Improving and Extending ODE Module

Table of Contents

Personal Details

Contact Information

Name:- Milan Jitendra Jolly

University:- Indian Institute of Technology, Patna

Email:- milan.cs16@iitp.ac.in, milan.jolly18@gmail.com

Github Profile:- **https://github.com/mijo2**

Timezone:- IST (UST + 5:30)

Personal Background

I am Milan Jolly, an undergraduate student at the Indian Institute of Technology, Patna pursuing Bachelor of Technology in Computer Science.

I have successfully completed two internships till this point and both of them involved Machine Learning and Deep Learning. Python is one of the prime choices for Machine Learning practitioners. Thus, I have worked on lots of projects where the primary language used was Python. Other than that, I have a keen interest in theoretical Machine Learning and reinforcement learning where Mathematics is heavily involved in finding logical solutions to highly dynamic problems.

Along with working on Machine Learning projects, I have experience in creating web applications by building both the frontend and backend of the websites using HTML, CSS, Bootstrap, MySQL and PHP. I can work with C and C++ to solve programming challenges. I am able to adapt quickly and if need arises, I can learn a new language as well. I have been using Linux terminal and Git for a while now and I am fairly competent with the same. Moreover, contributing to SymPy for the past couple of months have honed these skills even further.

Relevant courses that I have taken:

- Linear Algebra
- Probability and Statistics
- Algorithms
- Introduction to Data Structures
- Abstract Algebra
- Discrete Mathematics

- Optimization techniques.

Platform Details

1. **OS:-** Ubuntu 18.04
2. **Hardware Details:-** Asus K401u; i7-6500U; 8 GB RAM
3. **Preferred Code Editor:-** Visual Studio Code
4. **IDE:-** Python - PyCharm

Contributions to SymPy

- **Merged PRs**
  - Addition of a test to verify if replace works with Equality/Relational or not
  - function: Added new proportional tolerance in test_comb_factorial.py
  - Added min_fixed and max_fixed options for Float in printers
  - Integrals of summations(nested)
  - Added new test cases for limits
  - Added PyPI installation procedure in README
  - atoms() method return value updated
  - Updated the docstring of some functions in util.py
  - Documentation update in stats/joint_rv_types.py
  - Docstring update in polysys
  - Fix for numpy arrays when the size 1 arrays were treated as scalars
- **Open PRs**
  - implementation of chinese remainder theorem over cartesian product of vectors
  - Added the argmax and argmin in sympy.calculus
  - Added floor implementation in solvers
  - Addition of schur number in combinatorics
  - Using matrix exp for solving linear differential equations

After some initial experience with easy to fix and good first issues, I have mainly focused my efforts on adding new features to SymPy after discussing the relevant design details with the maintainers.

Project

Abstract

There are many Ordinary differential equation solvers in SymPy. But, it is possible to eliminate many of these solvers by introducing some new solvers that handle more of the general cases of ODEs. The current solvers handle special cases more than the general ones, and many of them can be replaced.

Along with that, solving a system of equations can be greatly enhanced by dividing the system of equations based on weakly connected components and strongly connected components. The general purpose solvers that will be introduced use matrices to solve the system and it is very efficient if we can divide the system of equations and solve these divided sub-systems separately. Also, a function that will reduce higher order ODEs to first order ODEs will help greatly extend the ODE module.

The solvers that will be introduced are given as follows:-

1. n equations linear first order constant coefficient homogeneous and non-homogeneous solvers.
2. n equations linear first order symmetric coefficient homogeneous and non-homogeneous solvers.
3. Special case nonlinear solvers.

Along with that the following are the functionalities that will be added for enhancement of the module:

1. Component division that divides the system of equations into weakly connected components and their corresponding strongly connected components which can help reduce the computational complexity of solving large equations.
2. nth order to 1st order functionality that reduces any nth order ODE to first ODE by introducing new dependent variables.

In this project, apart from adding new solvers and methods of enhancements, one crucial aspect is always replacing the unwanted solvers carefully and ensuring that removing a solver doesn't lead to a failure in a case where the original solver could have succeeded. Along with that, adding new test cases, updating and migrating old test cases, keeping track of which issues and/or PRs have been taken care of by introducing the current set of solvers and updating the documentation is another important part of this project which will consume time and should be done along with every new addition of solvers/techniques. That is the reason why in every phase, this task is added so that it doesn't become overwhelming in the future.

Motivation

Several solvers which handle systems of ODEs in the ODE module are special case solvers and there is only one general case solver which also works when a special condition is met. This makes it hard to understand, develop and work with the ODE module. The module can be greatly enhanced by introducing general case solvers.

Most of the special case solvers handle 2 or 3 equations at a time which greatly increases the size of the module when it can be made more compact with a lot more capability. The only n equations general solver is _linear_neq_order1_type1 which handles only the case where the coefficient matrix is not defective. But, it is possible to implement a general purpose solver that solves a system of linear first order constant coefficient ODEs without the constraint that the coefficient matrix be non-defective. This can be achieved by using the Jordan Normal form of a matrix to get the exponential of both the defective and the non-defective matrices.

We can also extend the module by adding a Python function that reduces a system of higher order ODEs to first order ODEs so that it is possible to solve higher order ODEs using the solvers implemented for first order ODEs.

Also, since we are using matrices to solve the general cases of ODEs, it will be useful and efficient to reduce the size of the matrices since matrix operations can be computationally heavy for large matrices, even if there are more number of matrices to compute solutions for. This can be achieved by dividing the system of equations into subsystems by using the concept of weakly connected and strongly connected components of a graph.

Many of the techniques and solvers that will be implemented are very new to the library and this will allow the users to solve many systems of ODEs, some that are relatively new. Instead of one incomplete general solver, when this project will be at its completion, there will be 4 more new general solvers.

By adding all of the n equation solvers, the capabilities of SymPy for solving systems of ODEs will be expanded significantly. An end user, who is using SymPy for solving systems of ODEs will be able to easily identify if the system of ODEs that he/she is trying to solve can be solved by the library or not, since few general solvers will replace many special case solvers and this will make it easier for someone to identify which systems of ODEs are solvable using SymPy. The users will now be able to solve all the systems of ODEs which are n equations linear constant coefficient ODEs of any order. Along with that since the module will be greatly extended with new solvers like n equations linear non constant coefficient solver.

Also, the systems of ODEs which can be divided and solved separately can't be solved with the current ode solvers and this will be addressed by dividing the systems into independent subsystems(component division). Thus, in principle, an end user may also use this to their advantage by passing a list of ODEs that is required to be solved where all the ODEs don't particularly form a system but rather are independent to each other. This will also allow the module to give partial solutions if it is possible to divide the system into subsystems where some of the subsystems can be solved using techniques available. By adding these general solvers and component division technique, not only the capabilities but the ease of solving systems of ODEs is also greatly enhanced.

Theory

Jordan Normal Form

All the matrices mentioned below are defined over a field K where 0 and 1 are additive and multiplicative inverses respectively. But the examples mentioned after the definitions and explanations are of matrices over Real numbers.

Matrices that can be represented as:-

$$A = P * D * P^{-1}$$

Are known as Diagonalizable matrices [1] where D is a diagonal matrix and P is an invertible matrix.

But, not every matrix is diagonalizable. A $n \times n$ matrix is diagonalizable when the matrix has n linearly independent eigenvectors [2]. Still, it is possible to represent any matrix in the form mentioned in the above equation.

$$A = P * J * P^{-1}$$

Where J is known as the Jordan normal form of the matrix A [3]. A Jordan form of a matrix is a square matrix in which only non-zero elements are in the diagonal elements and super-diagonal of the matrix J and everywhere else the elements are zero. Here, instead of all the diagonal elements being non-zero, we have Jordan blocks, which are of a particular form, as the diagonals of a matrix.

A Jordan block is a square matrix with diagonals having a value λ and all the elements in the super-diagonal having value 1 and everywhere else, the elements have value 0 [4]. Infact, a diagonal matrix is a special case of Jordan matrix where all the Jordan blocks are of size 1×1.

Notation for a Jordan block is given as $J_{\lambda,k}$ where λ is the element found in the diagonals and k stands for the size of the block.

A Jordan normal form has Jordan blocks on its diagonals which is represented as:

$$J = J_{\lambda_1,n_1} \oplus \ldots \oplus J_{\lambda_r,n_r}$$

where J is the Jordan normal form and $J_{\lambda_1,n_1}, \ldots, J_{\lambda_r,n_r}$ are the Jordan blocks.

Now, we will look into some examples to grasp the concept of Diagonal matrix and Jordan normal form.

Consider the matrix below:

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 5 & 12 & 11 \end{bmatrix}$$

Here, the matrix has eigenvalues: $1, 6 - \sqrt{42}, 6 + \sqrt{42}$. Also, all the eigenvectors are independent, and hence the matrix given above is diagonalizable with the diagonal elements as the eigenvalues.

But consider the matrix given below:

$$A = \begin{bmatrix} 24 & 5 & -1 \\ 12 & 11 & -1 \\ 204 & 55 & -5 \end{bmatrix}$$

This matrix has two unique eigenvalues $6, 12$ where 12 has an algebraic multiplicity as 2, hence there are only 2 independent eigenvectors of the above matrix thus the above matrix is not diagonalizable. But the Jordan normal form using the eigenvalues found above is:

$$J = \begin{bmatrix} 6 & 0 & 0 \\ 0 & 12 & 1 \\ 0 & 0 & 12 \end{bmatrix}$$

Here the Jordan blocks are $J_{6,1}$ and $J_{12,2}$ and $J = J_{6,1} \oplus J_{12,2}$.

It can be solved and proved that there exists an invertible matrix namely P given below such that $A = P * J * P^{-1}$ where A and J are the matrices mentioned above.

$$P = \begin{bmatrix} 0 & 6 & \frac{1}{12} \\ \frac{1}{5} & 0 & 1 \\ 1 & 72 & 0 \end{bmatrix}$$

Matrix Exponentials

Here, we are assuming that the elements of the matrices used in definitions and explanations belong to an algebraically closed field K where 0 and 1 are additive and multiplicative inverses respectively. In the examples, the elements of the matrices belong to the field of Real numbers.

Matrix exponentials [5] are functions over square matrices analogous to the ordinary exponential functions. The function is defined as follows:

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} X^k$$

Now, computing a matrix exponential naively means that we will have to infinitely compute the sum mentioned above. Naturally we have better techniques to compute the matrix exponential.

We can compute the Jordan normal form of the matrix to compute the matrix exponential. This can be easily proved. Let us assume that we have a matrix A for which we have to compute the matrix exponential. We will find the Jordan normal form of the matrix A along with the invertible matrix P and then substitute A as $P * J * P^{-1}$. Now, the below expression can be easily proved by just substituting the value of A in terms of P and J.

$$e^A = P * e^J * P^{-1}$$

Let us consider a variable t. Now, it is evident that $e^{At} = P * e^{Jt} * P^{-1}$. Note that t is an element of field K.

Exponential of a matrix of Jordan form is easy to compute. This can be demonstrated by looking at the cases mentioned below [6].

$$J = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix}$$

The exponential e^{Jt} is:

$$\begin{bmatrix} e^{\lambda t} & 0 \\ 0 & e^{\lambda t} \end{bmatrix}$$

For

$$J = \begin{bmatrix} \lambda & 1 \\ 0 & \lambda \end{bmatrix},$$

e^{Jt} is

$$\begin{bmatrix} e^{\lambda t} & te^{\lambda t} \\ 0 & e^{\lambda t} \end{bmatrix}$$

For

$$J = \begin{bmatrix} \lambda & 1 & 0 \\ 0 & \lambda & 1 \\ 0 & 0 & \lambda \end{bmatrix},$$

e^{Jt} is

$$\begin{bmatrix} e^{\lambda} & t * e^{\lambda} & t * *2 * e^{\lambda}/2 \\ 0 & e^{\lambda} & t * e^{\lambda} \\ 0 & 0 & e^{\lambda} \end{bmatrix}$$

It can be seen and proved that exponential of a Jordan block is simpler than computing the exponential as a sum.

It is also true that if $J = J_{\lambda_1,n_1} \oplus \ldots \oplus J_{\lambda_r,n_r}$ where $J_{\lambda_1,n_1}, \ldots, J_{\lambda_r,n_r}$ are the Jordan blocks, then:

$$e^{Jt} = e^{J_{\lambda_1,n_1}} \oplus \ldots \oplus e^{J_{\lambda_r,n_r}}$$

Computing an exponential of a Jordan matrix is simpler than computing an exponential of a normal matrix. The complexity of computing the exponential of a Jordan matrix is O(n^2) where n is the size of the matrix which is better as compared to the naive method of evaluating the sum.

Now, we will consider an example, namely of the matrix that we used to compute the Jordan normal form.

$$A = \begin{bmatrix} 24 & 5 & -1 \\ 12 & 11 & -1 \\ 204 & 55 & -5 \end{bmatrix}$$

If $A = P * J * P^{-1}$, then $e^{At} = P * e^{Jt} * P^{-1}$ and since for the matrix above,

$$P = \begin{bmatrix} 0 & 6 & \frac{1}{12} \\ \frac{1}{5} & 0 & 1 \\ 1 & 72 & 0 \end{bmatrix}, \quad J = \begin{bmatrix} 6 & 0 & 0 \\ 0 & 12 & 1 \\ 0 & 0 & 12 \end{bmatrix}$$

Then, using the cases and properties discussed above,

$$e^{Jt} = \begin{bmatrix} e^{6t} & 0 & 0 \\ 0 & e^{12t} & te^{12t} \\ 0 & 0 & e^{12t} \end{bmatrix}$$

and thus e^{At} is given as follows:

$$\begin{bmatrix} 12te^{12t} + e^{12t} & 5te^{12t} & -te^{12t} \\ 2e^{12t} - 2e^{6t} & \frac{5e^{12t}}{6} + \frac{e^{6t}}{6} & -\frac{e^{12t}}{6} + \frac{e^{6t}}{6} \\ 144te^{12t} + 10e^{12t} - 10e^{6t} & 60te^{12t} - \frac{5e^{12t}}{6} + \frac{5e^{6t}}{6} & -12te^{12t} + \frac{e^{12t}}{6} + \frac{5e^{6t}}{6} \end{bmatrix}$$

n equations linear first order constant coefficient homogeneous ODEs

For solving the homogeneous case represented as:

$$X' = AX$$

Where X is a vector of dependent variables, A is the constant coefficient matrix and X' is its derivative with respect to t. t is the independent variable.

Solution for such a system is given by:

$$X = e^{At}C$$

Where C is a vector of arbitrary constants. If the initial conditions are given as $X(0)$ then,

$$X = e^{At}X(0)$$

Exponential of At can be calculated using the technique discussed above.

Let us consider the system of differential equations:

$$\frac{dx}{dt} = 24x + 5y - z,$$

$$\frac{dy}{dt} = 12x + 11y - z,$$

$$\frac{dz}{dt} = 204x + 55y - 5z,$$

Here, x, y, z are functions of the independent variable t. So, using the solution discussed above and the exponential of the matrix At discussed in the previous section, the solution of the system above is:

$$X = \begin{bmatrix} 12te^{12t} + e^{12t} & 5te^{12t} & -te^{12t} \\ 2e^{12t} - 2e^{6t} & \frac{5e^{12t}}{6} + \frac{e^{6t}}{6} & -\frac{e^{12t}}{6} + \frac{e^{6t}}{6} \\ 144te^{12t} + 10e^{12t} - 10e^{6t} & 60te^{12t} - \frac{5e^{12t}}{6} + \frac{5e^{6t}}{6} & -12te^{12t} + \frac{e^{12t}}{6} + \frac{5e^{6t}}{6} \end{bmatrix} * \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

where $X = [x, y, z]$ and C_1, C_2, C_3 are arbitrary constants.

n equations linear first order constant coefficient non-homogeneous ODEs

For solving the non-homogeneous case as represented below:

$$X' = AX + f(t)$$

Where A is a constant coefficient matrix, X is a vector of independent variables, t is the independent variable and $f(t)$ is the non-homogeneous component of the system of ODEs.

Now, let us try to solve this equation:

Bringing $A * X$ term from right hand side to left hand side of the equation, we get:

$$X' - AX = f(t)$$

Now, we multiply both the sides by e^{-At} and since:

$$\frac{d}{dt}(e^{-At}X) = e^{-At}X' - Ae^{-At}X$$

we get:

$$e^{-At}X' - Ae^{-At}X = f(t)$$

$$\frac{d}{dt}(e^{-At}X) = e^{-At}f(t)$$

Integrating on both the sides and then rearranging, we get X as:

$$X = e^{At}(\int e^{-At}f(t)dt + C)$$

Where, C is a vector of arbitrary constants.

Now, let us consider an example of a non-homogeneous system that can be solved using the solution discussed above. The system of differential equations is given as follows:

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = t + x$$

So, our A is

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and $f(t)$ is

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

So, e^{At} is

$$\begin{bmatrix} \frac{e^t}{2} + \frac{e^{-t}}{2} & \frac{e^t}{2} - \frac{e^{-t}}{2} \\ \frac{e^t}{2} - \frac{e^{-t}}{2} & \frac{e^t}{2} + \frac{e^{-t}}{2} \end{bmatrix}$$

And e^{-At} is

$$\begin{bmatrix} \frac{e^t}{2} + \frac{e^{-t}}{2} & -\frac{e^t}{2} + \frac{e^{-t}}{2} \\ -\frac{e^t}{2} + \frac{e^{-t}}{2} & \frac{e^t}{2} + \frac{e^{-t}}{2} \end{bmatrix}$$

Now, substituting values into the solution, we get:

$$X = \begin{bmatrix} \left(\frac{e^t}{2} - \frac{e^{-t}}{2}\right)\left(C_2 + \frac{e^t}{2} - \frac{e^{-t}}{2}\right) + \left(\frac{e^t}{2} + \frac{e^{-t}}{2}\right)\left(C_1 - \frac{e^t}{2} - \frac{e^{-t}}{2}\right) \\ \left(\frac{e^t}{2} - \frac{e^{-t}}{2}\right)\left(C_1 - \frac{e^t}{2} - \frac{e^{-t}}{2}\right) + \left(\frac{e^t}{2} + \frac{e^{-t}}{2}\right)\left(C_2 + \frac{e^t}{2} - \frac{e^{-t}}{2}\right) \end{bmatrix}$$

The example discussed above is from the ODE systems roadmap [7].

n equations linear first order non-constant coefficient homogeneous ODEs

For a system of ordinary differential equations given below in matrix format:

$$X' = A(t) * X$$

Now, if $B(t)$ is the antiderivative and it commutes with $A(t)$, then it can be proved that:

$$\frac{d}{dt}(e^{B(t)}) = A(t)e^{B(t)}$$

Now, it is clear by looking at both the equations that $e^{B(t)}C$ is the general solution of X.

Let us consider a system of differential equations of this case and get the solution:

$$\frac{dx}{dt} = t * y$$

$$\frac{dy}{dt} = t * x$$

Here,

$$A(t) = \begin{bmatrix} 0 & t \\ t & 0 \end{bmatrix}$$

Since, $A(t)$ is symmetric, then we can use the solution discussed above. Now antiderivative $B(t)$ is:

$$B(t) = \begin{bmatrix} 0 & \frac{t^2}{2} \\ \frac{t^2}{2} & 0 \end{bmatrix}$$

Now, exponential of this matrix is:

$$e^{B(t)} = \begin{bmatrix} \cosh\left(\frac{t^2}{2}\right) & \sinh\left(\frac{t^2}{2}\right) \\ \sinh\left(\frac{t^2}{2}\right) & \cosh\left(\frac{t^2}{2}\right) \end{bmatrix}$$

Thus, our solution is:

$$X(t) = e^{Bt}C$$

$$X(t) = \begin{bmatrix} C_1 \cosh\left(\frac{t^2}{2}\right) + C_2 \sinh\left(\frac{t^2}{2}\right) \\ C_1 \sinh\left(\frac{t^2}{2}\right) + C_2 \cosh\left(\frac{t^2}{2}\right) \end{bmatrix}$$

n equations linear first order non-constant coefficient non-homogeneous ODEs

For the system of ODE given below:

$$X' = A(t)X + f(t)$$

where $A(t)$ is a coefficient matrix dependent only on the variable t and $f(t)$ is the non-homogeneous term of the system of ODE.

Now, we bring $A(t)X$ term in the left hand side of the equation and then multiplying both the sides with e^{-Bt} where $B(t)$ is the antiderivative of $A(t)$, we get:

$$e^{-B(t)}X' - A(t)e^{-B(t)}X = e^{-B(t)}f(t)$$

Now, from the section above, we know that if $B(t)$ and $A(t)$ commute, then:

$$\frac{d}{dt}e^{-B(t)} = A(t)e^{-B(t)}$$

And from this, we can infer using simple derivative calculation that:

$$\frac{d}{dt}(e^{-B(t)}X) = e^{-B(t)}X' - A(t)e^{-B(t)}X$$

Substituting this value into the left hand side of the equation discussed above, we get:

$$\frac{d}{dt}(e^{-B(t)}X) = e^{-B(t)}f(t)$$

Integrating on both the sides, we get:

$$e^{-B(t)}X = \int e^{-B(t)}f(t)dt + C$$

where C is a vector of arbitrary constants. Multiplying both the sides by $e^{B(t)}$ we get X as:

$$X = e^{B(t)}\int e^{-B(t)}f(t)dt + e^{B(t)}C$$

The above is the solution for the system defined in this section.

Now, we will look at an example of this system:

$$\frac{dx}{dt} = t * y + t$$

$$\frac{dy}{dt} = t * x$$

Here, $A(t)$ and its antiderivative $B(t)$ is the same as the example discussed in the section above, so we don't need to compute the exponentials again. The only difference between the two examples is that this example system of ODEs has a non-homogeneous term $f(t)$ as:

$$\begin{bmatrix} t \\ 0 \end{bmatrix}$$

Now, using the solution discussed above, we can get the solution. First, lets compute the integral part of the solution:

$$\int e^{-B(t)}f(t)dt = \begin{bmatrix} \frac{e^{\frac{t^2}{2}}}{2} - \frac{e^{-\frac{t^2}{2}}}{2} \\ -\frac{e^{\frac{t^2}{2}}}{2} - \frac{e^{-\frac{t^2}{2}}}{2} \end{bmatrix}$$

Thus, substituting this value in the solution along with the value of $e^{B(t)}$ computed in the above section, we can get the solution:

$$X = \begin{bmatrix} C_1 \cosh\left(\frac{t^2}{2}\right) + C_2 \sinh\left(\frac{t^2}{2}\right) \\ C_1 \sinh\left(\frac{t^2}{2}\right) + C_2 \cosh\left(\frac{t^2}{2}\right) - 1 \end{bmatrix}$$

Higher order ODEs to first order ODEs

In the above sections, the solutions that were discussed were of first order ODEs and they don't directly work on higher order ODEs. But, there is a way to convert the higher order ODEs to first order ODEs. This will allow us to use the methods defined for obtaining the solutions of first order ODEs to solve higher order ODEs.

This technique will be explained by considering an example:

$$\frac{dx}{dt} = y+'$$

$$\frac{d^2y}{dt^2} = x$$

Here, we can see that the system of ODEs is that of mixed 2nd order. So, now, we will introduce another dependent term $p(t)$ along with $x(t)$ and $y(t)$ such that $p = y'$. Hence, the system of ODEs of two dependent variables and two equations becomes that of three dependent variables and 3 equations:

$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = p$$

$$\frac{dp}{dt} = x$$

Now, it is clear that this system of ODE is linear first order constant coefficient ODE of 3 equations.

The coefficient matrix is:

$$A(t) = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

where $X(t)$ is:

$$\begin{bmatrix} x \\ y \\ p \end{bmatrix}$$

So, the solution of this ODE is:

$$X(t) = e^{At}C$$

Now, when we get the solution, we will see that we don't need the value of p and we can just filter out the solutions for the dependent variables that we originally wanted the solution for.

Non-linear solvers

A general system of ODEs with two independent variables can be given as:

$$\frac{dx}{dt} = f(x, y, t)$$

$$\frac{dy}{dt} = g(x, y, t)$$

Now, we can take a look at a case when we can separate out the term of independent variable:

$$\frac{dx}{dt} = f(x, y) * h(t)$$

$$\frac{dy}{dt} = g(x, y) * h(t)$$

So, we can divide both of these equations and get two equations:

$$\frac{dx}{dy} = \frac{f(x,y)}{g(x,y)}$$

$$\frac{dy}{dx} = \frac{g(x,y)}{f(x,y)}$$

We can solve both of these equations to get x in terms of y and vice-versa. Finally, we can substitute x or y in the respective differential equation and get the solution for one of the variables and using that we can get the values for both of the dependent variables in terms of t.

Let us consider an example to demonstrate this technique:

$$\frac{dx}{dt} = xy$$

$$\frac{dy}{dt} = xy$$

Dividing both of these equations, we will get:

$$\frac{dx}{dy} = 1$$

By integrating on both the sides with respect to dy, we get the relation:

$$x = y + c_1$$

where c_1 is the integration constant. Now, substituting value of x in the second ODE:

$$\frac{dy}{dt} = (y + c_1) * y$$

which gives us the solution:

$$y = \frac{c_1 * e^{c_1 * (t + c_2)}}{1 - e^{c_1 * (t + c_2)}}$$

and finally using the value of y, we can easily get the value of x using the relation found:

$$x = c_1 + \frac{c_1 * e^{c_1 * (t + c_2)}}{1 - e^{c_1 * (t + c_2)}}$$

Weakly connected components

There might be some ODEs for which we don't have any direct solution. And for which we do have, solving them using big matrices can be computationally heavy as the matrix operations that we are doing are not O(n) but at least an order more than that(since e^{Jt} has complexity O(n^2)). So, to solve

both of these issues, we can divide a system of equations into subsystems such that there are methods to deal with subsystems and along with that, we have an added advantage of expending less computational resources for solving the system of ODEs.

Now, we will use the concept of weakly connected components to divide the system into logical subsystems. To explain this technique, let us consider a system of ODEs defined below:

$$\frac{d^2x}{dt^2} = x$$

$$\frac{d^2y}{dt^2} = y^2$$

Here, if we club the system of equations and try to solve them together, we won't be able to solve them as we have only some of the general linear ODE solvers(where we can't solve all the non constant coefficient linear ODEs) and this system mentioned above is clearly non-linear.

So, let us construct a graph where nodes are the dependent variables x and y and there is a directed edge between the nodes if the derivative of the highest order of a dependent variable is in the left hand side, and there is a term of another dependent variable in the right hand side. So, here there is an edge from the dependent variable which is on the left hand side to the dependent variable which is found in the right hand side.

The above is the graph that is constructed based on the rule defined in the above paragraph. Note that there is no edge between the nodes.

So, for a directed graph, a weakly connected component is defined as: weakly connected component is a subgraph of the original graph where all vertices are connected to each other by some path, ignoring the direction of edges [8].

Now, given the graph generated, it is clear that the weakly connected components are x and y. This means, we have found our two subsystems, namely they are just the equations where in the left hand side, we have order 2 terms for x and y. This means we will solve both the equations separately and independently. This will give us the values for x and y in terms of t.

So, the final solution will be:

$$x = c_1 * e^{-t} + c_2 * e^{t}$$

$$y = \frac{-1}{c_3 + t}$$

So, we are dividing the system of equations considering the assumption that the system is in the format where the left hand side has the highest order of a dependent variable and it is in this format: $x' = f(x, y, z, ...)$, $y' = g(x, y, z,)$ and so on.

Now, let us look at another example to see how dividing the system of equations helps:

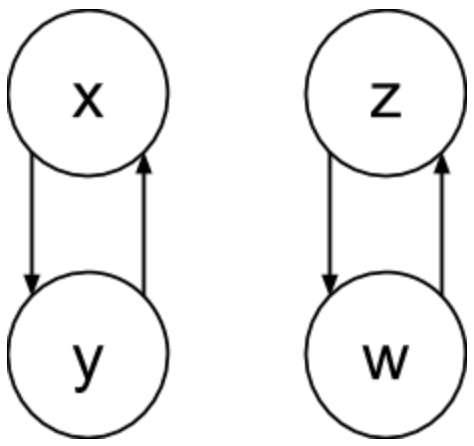
$$\frac{dx}{dt} = y$$

$$\frac{dy}{dt} = x$$

$$\frac{dz}{dt} = w$$

$$\frac{dw}{dt} = z$$

Constructing the graph based on the rules defined above, we get:



We can see that there are two weakly connected components, namely x, y and z, w. Hence, we will solve both of these subsystems separately and we will get the solutions in terms of t.

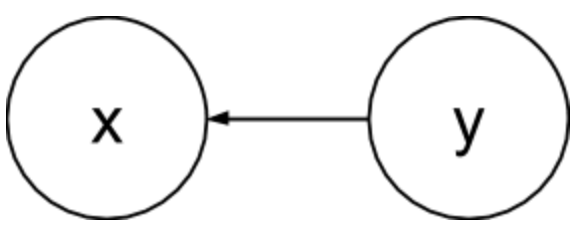
Strongly connected components

Now, we have come up with a procedure of dividing a system of ODEs into subsystems by defining a graph and using weakly connected components. But consider the system of ODEs defined below:

$$\frac{dx}{dt} = x^2$$

$$\frac{dy}{dt} = x$$

The graph for this ODE will be:



Here, we can see that the weakly connected component is the system itself but it's trivial to notice that we can first solve for x and then for y to get solutions for both.

So, we will use strongly connected components to solve both of these equations by dividing the weakly connected components further.

A strongly connected component is a subgraph such that every vertex in that subgraph is reachable from every other vertex in that subgraph [8].

We need to find the strongly connected components in reverse topological order. Then we will use this order to solve the components. But, unlike solving the weakly connected components separately and independently, we will have to solve the strongly connected components in the following manner:

Suppose we have strongly connected components named as a, b and c and the reverse topological order is: a, b, c. Now, these components a, b and c are systems of ODEs themselves. So, first we solve a, then we get the solutions for dependent variables in a and substitute the values in the system b. Then, we get the solutions for dependent variables in b and substitute the values of dependent variables found by solving systems a and b in the equations of system c. In this way, we can solve many strongly connected components.

Now, let us solve the system of ODEs introduced above:

First step would be to divide the system of ODEs into its weakly connected components. x, y is the weakly connected component.
Second step would be to divide each weakly connected component into its strongly connected components. Since, the weakly connected component for the example is x, y, the strongly connected components of this subsystem in the reverse topological order is: x, y.
Third step would be to solve the first element in the strongly connected component. This gives x as:

$$x = \frac{-1}{c_1 + t}$$

Finally, we will continue the chain of substituting the values of the dependent variables when solving the strongly connected components.

Now, we will get the differential equation by substituting the value of x:

$$\frac{dy}{dt} = \frac{-1}{c_1 + t}$$

Solving this equation gives:

$$y = c_2 - log(c_1 + t)$$

Implementation

Helper Functions

1. **_match_ode:**

This helper function is an important component of the overall solver. It takes the equations and the order of each dependent variable in the equations and determines important information about the system of ODEs which will be helpful later on to determine which type of system it is, get the important parts of the equation so that the solvers just need to access the information required to solve the system.

Parameters:

- eqs
    1. Type: List
    2. Explanation: List of equations that make up the system of ODEs that are needed to be solved.
- order
    1. Type: Dict
    2. Explanation: Dictionary where keys are the dependent variables and values are the maximum order of that dependent variable found in the system of ODEs.
- t
    1. Type: Symbol

2. Explanation: The independent variable in the system of ODEs

Returns:

- match
    1. Type: Dict
    2. Explanation: Dictionary where keys are the strings representing important information of the system of ODEs and values are the information needed.

       The keys and the nature of the values are given as follows:

        a. eqs: A list. Equations of the system of ODEs.
        b. t: A symbol. The independent variable in the system of ODEs.
        c. funcs: A list. List of dependent variables.
        d. order: A dictionary. Keys are the dependent variables and values is the maximum order.
        e. is_linear: A boolean. True if the system is linear, else False.
        f. is_constant. Boolean or None. True if the system is constant coefficient, False if it isn't and None the system is not linear or not first order.
        g. is_homogeneous: Boolean or None. True if the system is homogeneous, False if it isn't and None the system is not linear or not first order.
        h. is_first_order: Boolean or None. True if the system is first order, False if it isn't and None the system is not linear.
        i. f(t): Matrix or None. Matrix f(t) from the equation $X' = AX + f(t)$ if the system is linear and first order. None if it isn't.
        j. func_coeff: Matrix or None. Matrix A from the equation $X' = AX + f(t)$ if the system is linear and first order. None if it isn't.

2. **_commutative_anti_derivative:**

This helper function checks if the coefficient matrix $A(t)$ and its antiderivative $B(t)$ commute or not.

Parameters:

- A
    1. Type: Matrix
    2. Explanation: This is the coefficient matrix for which the check has to be made.
- t
    3. Type: Symbol
    4. Explanation: The independent variable in the system of ODEs

Returns:

- is_commutative
    3. Type: Boolean

4. Explanation: True if A and B commute, else False.

3. **_independent_rhs:**

This helper function checks if the equation passed is independent of the term passed in the second argument. But, this function is such that it will ignore terms like $x(t)$ but won't ignore terms like $x(t) + t$.

Parameters:

- eq
    1. Type: Equation
    2. Explanation: The equation that has to be tested.
- t
    1. Type: Symbol
    2. Explanation: The symbol required for the test.

Returns:

- is_independent
    1. Type: Boolean
    2. Explanation: True if the check passes, False otherwise.

4. **_nth_order_to_first_order:**

This helper function reduces a linear nth order system of ODEs to a first order system of ODEs by introducing more dependent variables. If the system is already in first order, then it doesn't change the equations.

Parameters:

- eqs
    1. Type: List
    2. Explanation: The equation that has to be reduced.
- order
    1. Type: Dictionary
    2. Explanation: The original dependent variables and their maximum order.

Returns:

- new_eqs
    1. Type: List
    2. Explanation: The reduced first order linear system of ODEs.
- new_funcs
    1. Type: List

2. Explanation: New list of dependent variables.

5. **_component_division:**

This helper function takes the system of ODEs (required to be in canonical form) and returns a 2D list where each element in this list is a sub-system. The original system is first divided as weakly connected components and each weakly connected component is divided into a list of strongly connected components. Strongly connected components are sorted in reverse topological order.

Parameters:

- eqs
    1. Type: List
    2. Explanation: The system that has to be divided
- funcs
    1. Type: List
    2. Explanation: The list of dependent variables in the system.

Returns:

- components
    1. Type: 2D List of tuples
    2. Explanation: 2D list of sub systems formed after dividing the original system. Each element in this 2D list is a tuple of 2 elements where the first element is the list of the equations to be solved and the second element is the list of dependent variables that are to be solved in the corresponding equations.

6. **_get_func_order:**

This helper function takes the system of ODEs and for each dependent variable finds the maximum order of that dependent variable found in the system of ODEs.

Parameters:

- eqs
    1. Type: List
    2. Explanation: The system of ODEs.
- funcs
    1. Type: List
    2. Explanation: The list of dependent variables in the system.
- t
    1. Type: Symbol
    2. Explanation: The independent variable in the system of ODEs

Returns:

- order
    1. Type: Dictionary
    2. Explanation: A dictionary where keys are the dependent variables and value is the maximum order of that dependent variable found in the system.

7. **_preprocess:**

This helper function takes the system of ODEs and preprocesses it to bring it to a particular format so that the solvers and the **_match_ode** function can work on it properly.

Parameters:

- eqs
    1. Type: List
    2. Explanation: The system of ODEs.

Returns:

- eqs
    3. Type: List
    4. Explanation: Preprocessed equations.

8. **_get_coeff_matrix:**

This function works only for linear systems of ODEs of from $X' = A(t)X + f(t)$. It returns the coefficient matrix and the homogeneous term. This function expects the equations in the form mentioned above, that is, the canonical form.

Parameters:

- eqs
    1. Type: List
    2. Explanation: The system of ODEs.
- funcs
    1. Type: List
    2. Explanation: The list of dependent variables in the system.

Returns:

- A_t
    1. Type: Matrix
    2. Explanation: Matrix of coefficients from the linear system of ODEs. May or may not be constant.
- f_t

1. Type: Matrix
2. Explanation: Vector of non homogeneous terms from the linear system of ODEs.

9. **_canonical_form:**

This function takes the system of ODEs along with the highest order of every dependent variable and solves the system for dependent variables in their highest order to reduce the system of ODEs to its canonical form.

Parameters:

- eqs
    1. Type: List
    2. Explanation: The system of ODEs.
- order
    1. Type: Dictionary
    2. Explanation: The dependent variables and their maximum order.

Returns:

- eqs
    1. Type: List
    2. Explanation: ODEs in their canonical form.

Solvers

1. **n equations linear first order constant coefficient homogeneous ODE:**

This solver solves the system of n equations where the system is linear, first order, constant coefficient and homogeneous. This solver is invoked by the function __linear_ode_solver when the match dict of the system indicates that this system is linear, constant coefficient, homogeneous and that every dependent variable is in its first order. Rough layout of the function is given as follows:-

```
def _neq_linear_first_order_const_coeff_homogeneous(match):

        # First, we get the coefficient matrix
        # from the system X' = A * X and the
        # independent variable.
        A = match["func_coeff"]
        t = match["t"]

        # Next thing, we get the last constant
        # index used to define our own constants
```

```
        const_idx = match["const_idx"]
        C = symbols("c{}:{}".format(const_idx + 1, const_idx + 1 + n))
        sol = (A*t).exp() * C
        const_idx += n

        # Finally we return the solution of the system
        return sol, const_idx
```

Parameters:

- match
    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system. The information that is required to solve this particular type of system is stored as values with key names: func_coeff (coefficient matrix), t (independent variable) and const_idx (last number of the constant used in solving previous ODEs).

Returns:

- sol
    1. Type: Matrix
    2. Explanation: This is the solution of the system of ODEs. It is in Matrix form because the solution is obtained through Matrix operations.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

2. **n equations linear first order constant coefficient non-homogeneous ODE:**

This solver solves the system of n equations where the system is linear, first order, constant coefficient and non-homogeneous. This solver is invoked by the function _linear_ode_sol when the match dict of the system indicates that this system is linear, constant coefficient, non-homogeneous and that every dependent variable has a maximum order of 1. This solver is very similar to the previous solver and hence the descriptions and pseudo code are also very similar. Rough layout of the function is given as follows:-

```
def _neq_linear_1st_order_const_coeff_non_homogeneous(match):

        # First, we get the coefficient matrix
        # from the system X' = A * X, the
        # independent variable and the term
```

```
        # f(t)
        A = match["func_coeff"]
        t = match["t"]
        f(t) = match[“non_homogeneous_term”]

        # Next thing, we get the last constant
        # index used to define our own constants
        const_idx = match["const_idx"]
        C = symbols("c{}:{}".format(const_idx + 1, const_idx + n + 1))

        exp_A = (A*t).exp()
        sol = exp_A * (integrate((-A*t).exp() * f(t), t) + C)
        const_idx += n

        # Finally we return the solution of the system
        return sol, const_idx
```

Parameters:

- match
    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system. The information that is required to solve this particular type of system is stored as values with key names: func_coeff (coefficient matrix), t (independent variable), non_homogeneous_term (the non-homogeneous term $f(t)$ in the system of ODEs) and const_idx (last number of the constant used in solving previous ODEs).

Returns:

- sol
    1. Type: Matrix
    2. Explanation: This is the solution of the system of ODEs. It is in Matrix form because the solution is obtained through Matrix operations.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

3. **n equations linear first order non-constant coefficient homogeneous ODE:**

This solver solves the system of n equations where the system is linear, first order, non-constant coefficient and homogeneous. This solver is invoked by the function _linear_ode_solver when the match dict of the system indicates that this system is linear, non-constant coefficient, homogeneous and that every dependent variable is in its first order. But this solver tests the coefficient matrix and

solves it after it satisfies a particular condition, that is the coefficient matrix and its antiderivative commute. Rough layout of the function is given as follows:-

```
def _neq_linear_1st_order_non_const_coeff_homogeneous(match):

        # First, we get the coefficient matrix
        # from the system X' = A(t) * X and the
        # independent variable
        A = match["func_coeff"]
        t = match["t"]

        # Here we check the coefficient matrix
        # using the helper function. If the check
        # is successful, then we solve the system
        # of ODEs using the solution defined below
        # this check.
        B, is_commuting = _commutative_anti_derivative(A, t)
        if is_commuting is not True:
           raise NotImplementedError("There is no available method to solve
this system of ODEs yet.")

        # Next thing, we get the last constant
        # index used to define our own constants
        const_idx = match["const_idx"]
        C = symbols("c{}:{}".format(const_idx + 1, const_idx + n + 1))

        exp_B = B.exp()
        sol = exp_B * C
        const_idx += n

        # Finally we return the solution of the system
        return sol, const_idx
```

Parameters:

- match
    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system. The information that is required to solve this particular type of system is stored as values with key names: func_coeff (coefficient matrix), t (independent variable), const_idx (last number of the constant used in solving previous ODEs).

Returns:

- sol
    1. Type: Matrix

2. Explanation: This is the solution of the system of ODEs. It is in Matrix form because the solution is obtained through Matrix operations.

- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

4. **n equations linear first order non constant coefficient non-homogeneous ODE:**

This solver solves the system of n equations where the system is linear, first order, non constant coefficient and non-homogeneous. This solver is invoked by the function _linear_ode_sol. This solver is very similar to the previous solver and has a similar check for solving the system of ODEs. Rough layout of the function is given as follows:-

```
def _neq_linear_1st_order_non_const_coeff_non_homogeneous(match):

        # First, we get the coefficient matrix
        # from the system X' = A * X, the
        # independent variable and the term
        # f(t)
        A = match["func_coeff"]
        t = match["t"]
        f(t) = match["non_homogeneous_term"]

        # Here we check the coefficient matrix
        # using the helper function. If the check
        # is successful, then we solve the system
        # of ODEs using the solution.
        B, is_commuting = _commutative_anti_derivative(A, t)
        if is_commuting is not True:
            raise NotImplementedError("There is no available method to solve
this system of ODEs yet.")

        # Next thing, we get the last constant
        # index used to define our own constants
        const_idx = match["const_idx"]
        C = symbols("c{}:{}".format(const_idx + 1, const_idx + n + 1))

        exp_B = B.exp()
        sol = exp_B * (integrate((-B).exp() * f(t), t) + C)
        const_idx += n

        # Finally we return the solution of the system
        return sol, const_idx
```

Parameters:

- match
    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system. The information that is required to solve this particular type of system is stored as values with key names: func_coeff (coefficient matrix), t (independent variable), non_homogeneous_term (the non-homogeneous term $f(t)$ in the system of ODEs) and const_idx (last number of the constant used in solving previous ODEs).

Returns:

- sol
    1. Type: Matrix
    2. Explanation: This is the solution of the system of ODEs. It is in Matrix form because the solution is obtained through Matrix operations.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

5. **2 equations nonlinear first order ODE:**

This solver solves the system of 2 first order ODEs where it is possible to eliminate the independent variable by simple division to get a direct relation between the two dependent variables. If the relation is possible, then this solver first solves the first dependent variable in terms of the second dependent variable, then uses the relation and solves for one dependent variable in terms of the independent variable by substituting the relation in one of the ODEs. Finally, after finding one of the dependent variables in terms of the independent variable, it is trivial to get the value of the second dependent variable using the relation established earlier. Note that this solver can be extended to handle 3 or more equations but as of now, only the solution for 2 non-linear equations is discussed. Rough layout of the function is given as follows:-

```
def _2eq_non_linear_first_order(match):

        # First, we get the equations,
        # the independent variable and
        # the two dependent variables
        eqs = match["eqs"]
        t = match["t"]
```

```
        x, y = [eq.lhs.args[0] for eq in eqs]
        const_idx = match["const_idx"]
        constants = symbols("c{}:{}".format(const_idx + 1, const_idx + 3))

        # Here we check if it is possible
        # to solve the system of ODEs.
        is_independent = _independent_rhs(eqs, t)
        if not is_independent:
            raise NotImplementedError("There is no available method to solve
this system of ODEs yet.")

        # In this section, we will divide the
        # two equations, get a differential equation
        # independent of the independent variable which
        # gives us the relation between two dependent variables
        # that we use to get the values of both the variables

        # Note: A rough code is not yet shown because it
        # still has to be implemented properly.

        # The answer is stored as a Matrix.
        z = Dummy()
        relation_eq = Eq(type(y)(z).diff(z), (eqs[1].rhs/eqs[0].rhs).subs({y:
type(y)(z), x:z}))
        relation_sol = dsolve(relation_eq, type(y)(z), ics={type(y)(0):
constants[0]})
        relation_sol = relation_sol.subs({type(y)(z): y, z: x})
        new_first_eq = eqs[0].subs(relation_sol.lhs, relation_sol.rhs)
        sol_first_eq = dsolve(new_first_eq, x, ics={type(x)(0):
constants[1]})

        sol = Matrix([x, y])
        const_idx += 2

        # Finally we return the solution of the system
        return sol, const_idx
```

Parameters:

- match

    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system. The information that is required to solve this particular type of system is stored as values with key names: eqs (ODEs), t (independent variable) and const_idx (last number of the constant used in solving previous ODEs).

Returns:

- sol
    1. Type: Matrix
    2. Explanation: This is the solution of the system of ODEs.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

Main functions

Now, we will look at how all of these pieces fit together and solve a given system of ODEs. First, we will look at two high level solvers, namely _linear_ode_solver and _non_linear_ode_solver. After, explaining these two solvers, we will finally look at the main solver functions ode_solver and its helper function _ode_component_solver.

1. **_linear_ode_solver:**

This function takes the match dictionary, extracts information and assigns a particular solver to solve the ODE if it is possible with the current solvers. This solver doesn't work for nonlinear systems of ODEs. This solver assumes that all the necessary information like the coefficient matrix or the non-homogeneous matrix is available in the match dictionary. This solver also reduces a nth order system to a first order system. This solver returns None if the system can't be solved using current techniques. Rough layout of the function is given as follows:-

```
def _linear_ode_solver(match):

        eqs = match["eqs"]
        funcs = match["funcs"]
        order = match["order"]
        t = match["t"]
        const_idx = match["const_idx"]
        is_first_order = match["is_first_order"]

        # First, we reduce the higher order
        # system of ODEs to first order system
        if is_first_order is False:
            new_eqs, new_funcs = _nth_order_to_first_order(eqs, order)
            A, f = _get_coeff_matrix(new_eqs, new_funcs)
            match["func_coeff"] = A
            match["non_homogeneous_term"] = f
```

```
        # Now, we will match the system of ODEs
        # with the corresponding solver
        solver = None
        if match["is_constant"]:
            if match["is_homogeneous"]:
                solver = _neq_linear_1st_order_const_coeff_homogeneous
            else:
                solver = _neq_linear_1st_order_const_coeff_non_homogeneous
        else:
            if match["is_homogeneous"]:
                solver = _neq_linear_1st_order_non_const_coeff_homogeneous
            else:
                solver = _neq_linear_1st_order_non_const_coeff_non_homogeneous

        try:
            sol, const_idx = solver(match)
        except NotImplementedError:
            return None

        sol = {func: value for func, value in zip(new_funcs, sol)}
        sol = {func: sol[func] for func in funcs}

        # Finally we return the solution of the system
        return sol, const_idx
```

Parameters:

- match
    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system.

Returns:

- sol
    1. Type: Dictionary
    2. Explanation: This is the solution of the system of ODEs. Here, keys are the dependent variables and values are the solutions.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

2. **_non_linear_ode_solver:**

This function is similar to __linear_ode_solver__ but is defined separately because this function is still incomplete and it is easier to deal with linear and nonlinear systems separately. Similar to the above

function, this function returns None if there is no technique to solve the current system of ODEs. Rough layout of the function is given as follows:-

```
def _non_linear_ode_solver(match):

        eqs = match["eqs"]
        funcs = match["funcs"]
        is_first_order = match["is_first_order"]

        solver = None
        if len(eqs) == 2 and is_first_order:
            solver = _2eq_non_linear_first_order

        # Note that more nonlinear solvers can be added
        # and then we just have to add appropriate code
        # in this function to solve the system.

        try:
            sol, const_idx = solver(match)
        except (TypeError, NotImplementedError):
            return None

        sol = {func: value for func, value in zip(funcs, sol)}

        return sol, const_idx
```

Parameters:

- match
    1. Type: Dict
    2. Explanation: match is a dictionary that contains the system of ODEs that need to be solved along with various other information about the system.

Returns:

- sol
    1. Type: Dictionary
    2. Explanation: This is the solution of the system of ODEs. Here, keys are the dependent variables and values are the solutions.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

3. **_ode_component_solver:**

This function uses the two solvers defined above to solve a particular subsystem of ODEs. This function is invoked by **ode_solver** and rough layout of the function is given as follows:-

```
def _ode_component_solver(eqs, funcs, t, const_idx=0):
        order = _get_func_order(eqs, funcs, t)
        match = _match_ode(eqs, order, t)
        match["const_idx"] = const_idx

        if match["is_linear"] is True:
            return _linear_ode_solver(match)

        return _non_linear_ode_solver(match)
```

Parameters:

- eqs
    1. Type: List
    2. Explanation: The ODEs that make the subsystem to be solved
- funcs
    1. Type: List
    2. Explanation: The list of dependent variables that are in the subsystem of ODEs.
- t
    1. Type: Symbol
    2. Explanation: The independent variable in the system of ODEs

Returns:

- sol
    1. Type: Dictionary
    2. Explanation: This is the solution of the system of ODEs. Here, keys are the dependent variables and values are the solutions.
- const_idx
    1. Type: int
    2. Explanation: This is the last numbered constant that was used as the integrating constant due to solving the system.

Or

- None

4. **ode_solver:**

This is the main function that is called to solve the system of ODEs. Two of the most important things that this function does is handles the strongly and the weakly connected components, and handles the numbering of the integration constants. For now, this function returns the solutions for the dependent variables that it can solve for. This function also keeps track of and returns the systems that aren't solved by this function. Rough layout of the function is given as follows:-

```
def ode_solver(eqs, funcs, t):
        # Firstly, we will do some preprocessing of the
        # equations as it is required for dividing the
        # system into subsystems and later for the solvers.
        eqs = _preprocess(eqs)
        order = _get_func_order(eqs)
        eqs = _canonical_form(eqs, order)

        # Initialising the solution dictionary, the
        # constant index and dividing the system into
        # subsystems.
        sol = {}
        const_idx = 0
        components = _component_division(eqs, funcs)

        # This list will be used to keep track of the systems
        # not solved by this function
        not_solved_systems = []

        for wcc in components:

            # Keeping track of the solutions obtained for each loop.
            loop_sol = {}

            for j, scc in enumerate(wcc):
                eqs, funcs = scc
                eqs = eqs.subs(loop_sol)
                component_solution =\
                       _ode_component_solver(eqs, funcs, t, const_idx)

                # If no solution is found for the scc, then
                # we break out of this loop as the other sccs for
                # this wcc won't be able solvable since they require
                # the solutions of this scc
                if component_solution is None:

                    # Including all the sccs after this scc
                    # in the not_solved_systems list since the
                    # other sccs depend on the solutions of this
                    # scc
                    not_solved_systems += wcc[j:]
```

```
                break

            temp_sol, const_idx = component_solution

            # Updating the loop sol with the new solutions.
            loop_sol.update(temp_sol)

        # Updating the original solution dictionary
        sol.update(loop_sol)

    return sol, not_solved_systems
```

Parameters:

- eqs
    1. Type: List
    2. Explanation: The ODEs that make the subsystem to be solved
- funcs
    1. Type: List
    2. Explanation: The list of dependent variables that are in the subsystem of ODEs.
- t
    1. Type: Symbol
    2. Explanation: The independent variable in the system of ODEs

Returns:

- sol
    1. Type: Dictionary
    2. Explanation: This is the solution of the system of ODEs. Here, keys are the dependent variables and values are the solutions.
- not_solved_systems
    1. Type: List
    2. Explanation: These are the subsystems of ODEs that weren't solved using the above solver. Each element is a tuple of 2 elements where the first element is the equations and the second element is the list of dependent variables found in those equations.

Solvers to be replaced

There are many solvers that can be replaced after the addition of the general solvers. Here, a list is made as to which solver should be replaced after some of the general solvers are added. The title is the general solver introduced and the list given below is that of the solvers that are currently present in the module which can be removed. The information about these solvers is given here [7].

1. **n equations linear first order constant coefficient homogeneous ODE:**
    - Linear, 2 equations, Order 1, Type 1 [9]
    - Linear, 3 equations, Order 1, Type 1 [10]
    - Linear, 3 equations, Order 1, Type 2 [11]
    - Linear, 3 equations, Order 1, Type 3 [12]
    - Linear, n equations, Order 1, Type 1 [13]

2. **n equations linear first order constant coefficient non-homogeneous ODE:**
    - Linear, 2 equations, Order 1, Type 2 [14]

3. **Higher order to first order ODE:**
    - Linear, 2 equations, Order 2, Type 1 [15]
    - Linear, 2 equations, Order 2, Type 2 [16]
    - Linear, 2 equations, Order 2, Type 3 [17]
    - Linear, 2 equations, Order 2, Type 4 [18]

4. **n equations linear first order non-constant coefficient homogeneous ODE:**
    - Linear, 2 equations, Order 1, Type 3 [19]
    - Linear, 2 equations, Order 1, Type 4 [20]
    - Linear, 2 equations, Order 1, Type 5 [21]
    - Linear, 2 equations, Order 2, Type 5 [22]
    - Linear, 2 equations, Order 2, Type 6 [23]
    - Linear, 2 equations, Order 2, Type 7 [24]
    - Linear, 2 equations, Order 2, Type 8 [25]
    - Linear, 2 equations, Order 2, Type 9 [26]
    - Linear, 2 equations, Order 2, Type 10 [27]
    - Linear, 3 equations, Order 1, Type 4 [28]

5. **n equations linear first order non constant coefficient non-homogeneous ODE:**

As of now, SymPy can't solve any linear first order non constant coefficient non-homogeneous system of ODEs.

6. **3 equations nonlinear first order ODE:**

In this proposal, only 2 equations nonlinear solver is discussed but the method discussed can be extended and it is included in the plans.

- Nonlinear, 2 equations, Order 1, Type 1 [29]
- Nonlinear, 2 equations, Order 1, Type 2 [30]
- Nonlinear, 2 equations, Order 1, Type 3 [31]
- Nonlinear, 2 equations, Order 1, Type 4 [32]
- Nonlinear, 3 equations, Order 1, Type 1 [33]

- Nonlinear, 3 equations, Order 1, Type 2 [34]

7. **Connected components:**
    - Nonlinear, 2 equations, Order 1, Type 5 [35]

A total of 27 solvers can be replaced by adding just 5 new solvers and one new technique of solving systems of ODEs.

Proposed Timeline

The timeline proposed below indicates exact time intervals required to complete each component of the project.

Pre-GSoC Period (Present - 27 April)

During this period, I will keep working on my currently open PRs especially the ones where a new feature is added to the SymPy module. Along with that, this period will be used for working on PR #18720 which is the starting point for this project. The first solver can be added during this period and hence lots of work can be done ahead of time, but when it comes to this project, the top priority for this period would be efficient design of the main function **ode_solver.**

Community Bonding Period (4 May - 1 June)

Since I already have been contributing towards SymPy for a while now, I would like to focus this time on the initial phases of the project which include adding the implementation of the structure of the main function **ode_solver.** This period will be utilised for the following tasks:

1. Finalisation of the design of **ode_solver** if not done already.
2. The main functions like **ode_solver** will be defined along with all the necessary helper functions in a new PR. This is a high priority task.
3. Completion of the implementation of solver introduced in PR #18720.
4. Checking all the test cases of the old solvers that are in principle replaced by the new solver.
5. Fixes of the implementation if the solver fails any old test cases.
6. Adding test cases, removing unwanted solvers and updating the documentation.

Phase I (1 June - 3 July)

This phase will be the longest one with respect to the number of days.

1. In the first week, a new PR will be created for adding the second solver, that is the technique to solve linear first order constant coefficient non-homogeneous systems(n equations). Along with that, the reviews given by the mentor(s) and the community will be taken into account to update the old solver.

2. In the second week, a new PR will be created for adding the function that reduces linear higher order ODEs to linear first order ODEs. This function would then be thoroughly tested and updated based on the errors and suggestions.

3. In the third week, a new PR will be created for adding a special case solver when linear first order non-constant ODE has a commuting coefficient matrix, homogeneous case. Along with that, updating the old PRs according to mentor(s), community suggestions and reviews will be done. The important helper functions for this update also need to be taken care of.

4. In the fourth week and the remaining time left, test cases will be updated, the PRs added till now will be updated based on the reviews, work for detailed documentation will take place, unwanted solvers will be replaced and issues that are fixed by adding new general solvers will be closed.

For the Phase I evaluation, three new general solvers will be ready along with a working layout of the main function **ode_solver,** some of the issues will be closed and unwanted solvers will be removed.

Phase II (4 July - 31 July)

This phase is 3 weeks and 6 days long including the Phase II deadline.

1. In the first week, adding technique to solve non-constant non-homogeneous linear ODE based on the solver added by the end of Phase I. Along with that, this week, new test cases need to be added as this is a new addition to the library.

2. In the second week, the main focus would be evaluating and eliminating unnecessary solvers and closing related issues for linear systems of ODEs.

3. In the third week, all the PRs related to linear systems of ODE solvers will be updated and will be brought to its final stages. I want these PRs to be ready for getting merged. This is essential so that when work for other components begins, then it would be almost certain that there is no primary issue with the general solvers in case any error arises due to new additions.

4. In the fourth week, adding basic rearrangements to simplify the system of ODEs. Along with that, documentation will be updated and added wherever required.

For Phase II evaluation, all the solvers that attempt to solve linear first order systems of ODEs will be in their final stages.

Phase III (18 July - 17 August)

For this phase, the main priorities would be to add the nonlinear solvers and when all the solvers are ready and fully tested, adding the component division part.

1. Dividing the ODEs by evaluating which sub-systems are weakly and strongly connected and handling both of these cases accordingly.

2. In the first week, adding a special case solver where the independent variable can be eliminated and thus solving the system becomes easier. First step to complete this task would be to discuss with the mentor(s) about the exact method. Then, creating a PR for the same and updating the implementation based on the reviews provided by the mentor(s) and the community.

3. The second week would be devoted to implementing the final part of the project, that is the component division. This part will be done only after the nonlinear solver is well tested.

4. The final two weeks will be utilised to wrap things up, updating and adding test cases, updating the documentation wherever necessary, updating the PRs based on the reviews, time will be given to complete the pending work and working on the final evaluation.

For the final evaluation, we will have the non linear system solvers along with the component division functionality.

Time commitment

The current semester is my last one(8th). The semester will end mostly in the first week of June. I will still contribute even when my colleges are going on as I have been doing since January. I would have roughly 2 months free before my job starts but I would contribute my best irrespective of anything else and I am confident about my performance regarding the same even when my job starts, I will be able to contribute 40 hours weekly, by covering up on the weekends and working a little less in the weekdays. This will be only for the last month though and hence, I have planned this project in such a way that the parts that will take time to implement will be done in the first two phases. Along with that, I am also planning to work more than 40 hours weekly during the first two phases for backup incase during the last phase things become intense. Working more in the first two phases will give a head start for the last phase.

Post GSoC period

After the completion of the final evaluation and the project, I will still keep contributing to SymPy. I have grown fascinated after reading the theory that will be involved in this project and I wish to learn more. If possible, I would try to find out more about solving systems of ODEs to further advance the

ODE solvers in SymPy. Along with that, my contributions won't be limited towards solvers.dsolve and I would try to contribute towards other modules as well. I will also keep trying out new examples to identify the weaknesses and shortcomings of the functionalities available and report them on GitHub by creating an issue so that the community is also aware of the same and everyone can work on improving and upgrading the library.

My special thanks to the members of the organization, most importantly, Oscar Benjamin(oscarbenjamin), Aaron Meurer(asmeurer), Gagandeep Singh(czgdp1807), Kalevi Suominen(jksuom), S. Y. Lee(sylee957) and Christopher Smith(smichr) who have helped me a lot by giving their reviews to my PRs and providing guidance wherever it was necessary. Due to the members mentioned above and the community, I have learned a lot and this has motivated me to contribute to this community and become a part of it.

Side note: A lot of examples in the Theory section are in latex code and the Printing functionality of SymPy was very helpful in making these latex codes. Hence, I wish to contribute towards this module as a token of appreciation.

References

- https://en.wikipedia.org/wiki/Diagonalizable_matrix
- https://mathworld.wolfram.com/DiagonalizableMatrix.html
- https://en.wikipedia.org/wiki/Jordan_normal_form
- https://en.wikipedia.org/wiki/Jordan_matrix
- https://en.wikipedia.org/wiki/Matrix_exponential
- https://www.math24.net/method-matrix-exponential/
- https://github.com/sympy/sympy/wiki/ODE-Systems-roadmap
- https://en.wikipedia.org/wiki/Connectivity_(graph_theory)