Refactor the ODE module and make it fast

About Me

Contact Information

Name - Mohit Balwani
University - Gujarat Technological university
College - Adani Institute of Infrastructure Engineering
Email - mohitbalwani.ict17@gmail.com
Github Profile - Mohitbalwani26
CodeChef Profile - mohit4426
Codeforces Profile - mo-hit
Timezone - IST (UTC + 5:30)

Personal Background

I am Mohit Balwani, 3rd-year undergraduate pursuing a degree in Information and Communication Technology at AIIE. I have more than 4 years of experience with python, as I started out in high school.

I have been extensively participating in programming contests on platforms like CodeChef, code forces, etc. which further developed my algorithmic skills and proficiency in Python.

Some of the courses which I have taken in my academic curriculum: Basic Calculus, Vector Calculus, Linear Algebra, Differential Equations(Ordinary and Partial), Probability and Statistics, software engineering, object-oriented programming with C++ and Python.

Programming Background

I use Ubuntu 18.04 (LTS) as my operating system. For development, I use V.S. Code because of its Command Palette and inbuilt version control system features and for competitive programming, I use Sublime text 3.

I have been programming for the last 4-5 years. I am familiar with python, C++ and Java.
I have been using python since high school. I have also completed the python proficiency Task on Hackerrank and I also use it as my primary language in competitive programming. I have also explored Machine Learning and web development.

My Github repositories :

1. Machine learning by Andrew-ng:
    - It contains all the assignments and projects given in the course for better hands-on experience.
    - Github link: https://github.com/Mohitbalwani26/machine-learning-andrew-ng-

2. TinDog:
    - It is an assignment project in which I was asked to develop a website for dog owners who can willingly provide dogs for adoption.
    - Github link: https://github.com/Mohitbalwani26/TinDog

I also secured 1435th rank in Google kickstart'19 round D and 3897th rank in Google code jam'19.
I prefer coding in python as it has many advantages over other programming languages. The programmer-friendly syntax; the diversity of applications, from web to command-line utilities; that can be developed using Python. The best part being, it is open source so the whole community can use it to develop various packages and later on contribute to it.

According to me, the most advanced feature I have used is **@lru_cache**. As I regularly participate in algorithmic contests many times the problem solution requires the dynamic programming approach and with the help of **@lru_cache,** I am able to achieve the results in a very efficient way with just a simple implementation.

There are many features of SymPy which I really like and have been using them over one year but amongst them, my top two favorites are **Integrate** and **dsolve**. Integrate makes the complex integration solve within seconds and dsolve solves the complex ODEs and it also identifies the classification of that ODE very accurately. I used dsolve for an entire semester as I had ODE as a subject last year.

Example:

```
>>> from sympy import *
>>> from sympy.abc import x, y, z
>>> f = x**2 + y**2 - x*y
>>> integrate(f, (x, -1, 1), (y, -1, 1))
8/3
>>> f = Function('f')
>>> eq = f(x).diff(x)**2 + 2*f(x).diff(x) + 1
Eq(f(x), C1 - x)
```

Contributions

Merged

(Easy to Fix)

- Fixed docstring in latex parser
- Added codecov Badge
- Fixes Attribute Error in diophantine.py

(Related to ODE module)

- adding solve function in ode_factorable_match
- Fixed the way of creating trialset
- bug fix in checkodesol
- Added hyperbolic function in undetermined coefficients
- Adding Subcheck module in ODE
- Refactoring Factorable hint (This PR is an example of this project)
- refactor(dsolve): move nth_algebraic tests to test_single.py

- refactor(dsolve): move Factorable tests to test_single.py
- Bugfix_in_get_general_solutions_of_bernoulli
- refactor(dsolve): move Riccati_special_minus2 and bernoulli test to single.py

(Closed)

- refactoring 1st_linear and almost_linear using pattern matching (This PR is an example of this project)
- Added order in SingleODEproblem

Both of these PRs were merged in Oscarbenjamin's branch pr_dsolve_refactor and in SymPy master, they are commits see PR#18403.

Issues Raised

- dsolve hangs for linear differential equation
- classify_ode doesn't recognise nth_linear_constant_coeff_undetermined_coefficients

The Project

Motivation

Currently, the dsolve function in the **ODE** module is a bit messy, as whenever dsolve is called for solving an ODE, it first calls `classify_ode()` which tries to match each solver and after that, it again calls that particular solver for solving. If a particular solver matches the equation it should directly return the solution instead. So, sometimes the solver which returns the solution is much faster than running all the matches.

I am interested in ODE since they have important applications and are a powerful tool in the study of many problems in the natural sciences and in technology. This motivated me to contribute to this module and I learned and explored a lot during this process. In fact, I used this module as last year my curriculum had ODE as a subject and I think refactoring will make this module more consistent as adding new solvers will be easy and the code will become more maintainable and complex ODE can be solved efficiently with speed.

Project Idea

I would divide my project into the following broad categories so that it becomes easier to plan and execute the workflow:

- Refactoring of `test_ode.py`
- Refactoring ODE solvers which will use `SingleODESolver` as their parent class.
- Refactoring ODE solvers which will use `SinglePatternODESolver` as their parent class

I will start refactoring with `test_ode.py` because currently, it contains repetitive tests and after refactoring them using appropriate data structures we can easily find out the bugs of individual solvers and the tests will be consistent throughout. This idea is discussed here.

Benefits of refactoring test_ode.py

- All examples will be tested properly corresponding to the proper format and hints.
- In some examples, dsolve is used as `dsolve(eq)` without specifying the hint for which it is tested. So maintaining the consistent format will surely help in highlighting the bugs in the individual solvers.
- We can also use these examples for benchmarking to measure the performance of dsolve.

Comparison

Before refactoring (ode.py)

```
def dsolve(eq, func,...):
    # checks whether it is a system of ODE or single ODE
    # let us say it is single ODE

    hints = _dsolve(eq, func)
    '''This is a helper function to dsolve which returns a dictionary that contains fields which are set by
    classify_ode as TRUE'''

    return _helper_simplify(eq,hint)
    #Here hint stores the best hint return by _desolve() or indirectly we can say classify_ode()

def classify_ode(eq, func):
        # classifies single ODE by trying to match each solvers code

        ''' In this function to check for different solvers manipulations like dividing the equation by
        coefficient of the highest order or extracting terms which are used by some solvers to produce
        solution are just  performed before matching function which makes it less organized '''
```

Let us Now suppose that match function of every solver has a statement print("I am " + name_of_solver)

```
>>> from sympy import *
>>> from sympy.abc import x
>>> f = Function('f')
>>> eq = (f(x)**2 - 4) * (f(x).diff(x) + f(x))
>>> dsolve(eq)
>>> I_am_Factorable_solver
>>> I_am_separable_solver
>>> I_am_1st_exact_solver
>>> I_am_Factorable_solver
```

The reason it is written twice is as first this solver is invoked by classify_ode then when the best hint gets its value as Factorable then again dsolve calls that function for solving that equation.

```
>>> [Eq(f(x), 2), Eq(f(x), -2), Eq(f(x), C1*exp(-x))]
```

After refactoring(ode.py)

Single.py

This file will contain all the classes of solvers which make the code more systematic and maintainable.

```
class SingleODEProblem:
        def __init__(self, eq, func):
                # Store the inputs passed to dsolve
                self.eq = eq
                self.func = func

class SingleODEsSolver(SingleODEProblem):
    def match(self):
    # return True if this method can solve the equation

    def get_general_solution(self):
    # return the general solution

class SinglePatternODEsSolver(SingleODESolver):
    # Superclass for ODE solvers based on pattern matching
    def _wilds(self):
    # elements which are to be matched
    def _match(self):
    # return True if this method can solve the equation
    def _equation(self):
    # returns the general equation to be matched for
```

ODE.py

```
def dsolve(eq, func)
    ode = SingleODEProblem(eq_orig, func, x, prep=prep)

    solvers = {
        NthAlgebraic: ('nth_algebraic',),
        FirstLinear: ('1st_linear',),
        AlmostLinear: ('almost_linear',),
        Bernoulli: ('Bernoulli',),
        Factorable: ('factorable',),
        RiccatiSpecial: ('Riccati_special_minus2',),
    }

    for solvercls in solvers:
        solver = solvercls(ode)
        if solver.matches():
            return solver.get_solution()
```

Let us Now suppose that match function of every solver has a statement print(“I am ” + name_of_solver)

```
>>> from sympy import *
>>> from sympy.abc import x
>>> f = Function(‘f’)
>>> eq = (f(x)**2 - 4) * (f(x).diff(x) + f(x))
>>> dsolve(eq)
>>> I_am_Factorable_solver After Refactoring the first solver which gets matched will be returned. This will make the process much faster.
>>> [Eq(f(x), 2), Eq(f(x), -2), Eq(f(x), C1*exp(-x))]
```

Before refactoring **(test_ode.py)**

Let’s say this function is used for checking solutions of basic_odes.

```
def test_odes_basic():
    eq = Eq(f(x).diff(x), 0)
    sol = Eq(f(x), C1)
    assert dsolve(eq, f(x)) == sol
    assert checkodesol(eq, sol)

    eq = Eq(f(x).diff(x), 1)
    sol = Eq(f(x), x + C1)
    assert dsolve(eq, f(x)) == sol
    assert checkodesol(eq, sol)
```

After refactoring **(test_ode.py)**

```
_odesol_basic = {
    'basic1': {
        'eq': Eq(f(x).diff(x), 0),
        'sol': Eq(f(x), C1),
    }
    'basic2': {
        'eq': Eq(f(x).diff(x), 1),
        'sol': Eq(f(x), x + C1),
    }
}

def  _test_all_odesol(ode_examples, hint="default"):
    for example in ode_examples:
        eq = ode_examples[example]['eq']
        sol = ode_examples[example]['sol']
        assert our_hint in classify_ode(eq)
        sols = [sol,constant_renumber(sol)]
        sols += [sols[-1].expand()]
        assert dsolve(eq,hint=our_hint).rhs in sols
        assert checkodesol(eq, sol)[0]

def test_ode_basic():
    _test_all_odesol(_odesol_basic)
```

Note: here `_test_all_odesol_()` makes sure that each solver is tested with the examples properly and this will also reduce the repetition of code.

Implementation

In this section, I have tried to show how individual solvers are going to be refactored.

1. **1st Exact solver:** A 1st order differential equation is called exact if it is the total differential of a function. That is, the differential equation P(x,y)∂x+Q(x,y)∂y=0 is exact if there is some function `F(x, y)` such that P(x,y)=∂F/∂x and Q(x,y)=∂F/∂y. It can be shown that a necessary and sufficient condition for a first-order ODE to be exact is that ∂P/∂y=∂Q/∂x . So the solution can be given as:

$$F(x,y) = \int_{Xo}^{X} P(t,y)dt + \int_{Yo}^{Y} Q(Xo,t)dt$$

   <u>How the refactored code will look like:</u>

```python
class First_Exact(SinglePatternODESolver):

    hint = "1st_exact"
    has_integral = True

    def _wilds(self, f, x, order):
        P = Wild('P', exclude=[f(x).diff(x)])
        Q = Wild('Q', exclude=[f(x).diff(x)])
        return P, Q

    def _equation(self, fx, x, order):
        P, Q = self.wilds()
        return P + Q*fx.diff(x)

    def _verify(self, fx):
        '''Here we will check the additional conditions for 1st_exact equation and will also
           try to check if a given equation can be converted to 1st_exact form.'''

    def _get_general_solution(self, *, simplify: bool = True):
        P, Q = self.wilds_match()
        fx = self.ode_problem.func
        x = self.ode_problem.sym
        ''' Here general solution is calculated mentioned in docs. we should remove the global
        variable Y which is used in current implementation'''
        return [gensol]
```

Similar kind of approach will be followed for below-mentioned solvers:

A. **Separable**: This is any differential equation that can be written as $P(y)dy/dx = Q(x)$. The solution can then just be found by rearranging terms and integrating: $\int P(y)dy = \int Q(x)dx$.

   Why pattern matching for this: we will consider the general form as $a(x) * b(f(x)) * d/dx(f(x)) = c(x) * d(f(x))$. Here a, b, c, d, f are functions and these can be used in _wilds function to get their values and return equation to be matched for in method _equation.

B. **Separable_reduced:** A differential equation that can be reduced to the separable form. The general form of this solver is $y' + (y/x)H(x^n y) = 0$. This can be solved by substituting $u(y) = x^n y$. The equation then reduces to separable form which can be solved by pattern matching.

C. **Linear_coefficients:** A differential equation with linear coefficients. The general form of a differential equation with linear coefficients is : $y' + F((ax + by + c)/(dx + ey + f)) = 0$. This can be solved by substituting $x = x' + (e * c - b * f)/(d * b - a * e)$ and $y = y' + (a * f - d * c)/(d * b - a * e)$. This substitution reduces the equation to a homogeneous differential equation.

   Why pattern matching for this: As in general form we need to find out the values of a, b, c, d, e, f so that we can make the desired substitution and express the general solution appropriately.

D. **Liouville:** The general form of a Liouville ODE is
$d^2y/dx^2 + g(y) * (dy/dx)^2 + h(x) * dy/dx.$

Why pattern matching for this: As in general form we can see that it will be easy to extract the functions h(x) and g(y) so the general solution can be directly written as $C1 + C2 * \int e^{-\int h(x)dx} dx + \int^{f(x)} e^{\int g(y)dy} dy = 0$.

E. **2nd_linear_airy:** Its general form is $d^2y/dx^2 + (a + bx) * y(x) = 0$ .Its general solution is expressed in terms of Airy special functions airyai and airybi.

F. **2nd_linear_bessel:** Its general form is
$x^2 * d^2y/dx^2 + x * dy/dx * y(x) + (x^2 - n^2) * y(x)$ Its general solution is expressed in terms of besselj and bessely.

G. **2nd_hypergeometric:** Current implementation is using pattern matching function but it is implemented for 2F1 type but it can be easily extended for 1F1 and 0F1. So I will also try to add these types.

H. **Nth_linear_constant_coeff_homogeneous:** This is an equation of the form $a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = 0.$ These equations can be solved in a general manner, by taking the roots of the characteristic equation.

I. **Nth_linear_euler_eq_homogeneous:** This is an equation with form $a_n x^n f^{(n)}(x) + a_{n-1} x^{n-1} f^{(n-1)}(x) + \cdots + a_1 x f'(x) + a_0 f(x) = 0.$ These equations can be solved in a general manner, by substituting solutions of the form $f(x) = x^r$ and then deriving a characteristic equation for r.

J. **Nth_linear_constant_coeff_undetermined_coefficients :** This is an equation of the form $a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = g(x).$

These equations can be solved in a general manner, by taking the roots of the characteristic equation and finding the particular integral. Here particular integral is evaluated using trialset i.e finite family of derivatives of g(x).

K. **Nth_linear_euler_eq_nonhomogenus_undetermined_coefficients:** This is an equation with form $a_n x^n f^{(n)}(x) + a_{n-1} x^{n-1} f^{(n-1)}(x) + \cdots + a_1 x f'(x) + a_0 f(x) = g(x)$. These equations can be solved in a general manner, by substituting solutions of the form $f(x) = x^r$ and then deriving a characteristic equation for r and rest process is same as nth_linear_constant_coeff_undetermined_coefficients.

L. **Nth_linear_constant_coeff_variation_of_parameters:** This is an equation of the form $a_n f^{(n)}(x) + a_{n-1} f^{(n-1)}(x) + \cdots + a_1 f'(x) + a_0 f(x) = g(x)$. These equations can be solved in a general manner, by taking the roots of the characteristic equation and finding the particular integral. Here the particular integral is evaluated using wronskian.

M. **Nth_linear_euler_eq_nonhomogeneous_variation_of_parameters:** This is an equation with form $a_n x^n f^{(n)}(x) + a_{n-1} x^{n-1} f^{(n-1)}(x) + \cdots + a_1 x f'(x) + a_0 f(x) = g(x)$. These equations can be solved in a general manner, by substituting solutions of the form $f(x) = x^r$ and then deriving a characteristic equation for r and rest process is the same as Nth_linear_constant_coeff_variation_of_parameters.

2. **nth_order_reducible:** For example any second order ODE of the form f''(x) = h(f'(x), x) can be transformed into a pair of 1st order ODEs g'(x) = h(g(x), x) and f'(x) = g(x). Usually the 1st order ODE for g is easier to solve. If that gives an explicit solution for g then f is found simply by integration.

How the refactored code will look like:

```python
class Nth_order_reducible(SingleODESolver):

    hint = "nth_order_reducible"
    has_integral = False

    def _matches(self):
        eq = self.ode_problem.eq
        f = self.ode_problem.func.func
        x = self.ode_problem.sym
        order =self.ode_problem.order
        df = f(x).diff(x)
        # returns true and false whether the given equation is solvable by nth_order_reducible


    def _get_general_solution(self, *, simplify: bool = True):
        func = self.ode_problem.func.func
        x = self.ode_problem.sym
        sols = []

        #Here the code will be written for writing general solution

        return sols
```

Similar kind of approach will be followed for below-mentioned solvers:

A. **1st_homogeneous_coeff_subs_indep_div_dep**:This is a differential equation $P(x, y) + Q(x, y)\, dy/dx = 0$. such that P and Q are homogeneous and of the same order. A function F(x, y) is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, F(x, y) can be rewritten as G(y/x) or H(x/y).
Why SingleODESolver for this: it is not necessary that we can directly extract elements from the general equation and return general solutions. We need to do a certain substitution like $u_2 =< independent\ variable > / < dependent\ variable >$. Also, the checking code like whether P and Q are of the same homogenous order can be easily implemented in _matches.

B. **1st_homogeneous_coeff_subs_dep_div_indep:** This is a differential equation $P(x, y) + Q(x, y)\, dy/dx = 0$. such that P and Q are homogeneous and of the same order. A function F(x, y) is homogeneous of order n if $F(xt, yt) = t^n F(x, y)$. Equivalently, F(x, y) can be rewritten as G(y/x) or H(x/y).

   Why SingleODESolver for this: it is not necessary that we can directly extract elements from the general equation and return general solutions. We need to do a certain substitution like $u_1 =< independent\ variable > / < dependent\ variable >$. Also, the checking code like whether P and Q are of the same homogenous order can be easily implemented in _matches.

The hint **1st_homogeneous_coeff_best** returns the best solution to an ODE from the two hints **1st_homogeneous_coeff_subs_dep_div_indep** and **1st_homogeneous_coeff_subs_indep_div_dep**. It is determined by ode_sol_simplicity.

What is ode_sol_simplicity? It returns an extended integer representing how simple a solution to an ODE is. The following things are considered, in order from the most simple to least:

- solution is solved for function. (returns -2)
- solution is not solved for function, but can be if passed to solve (e.g., a solution returned by ``dsolve(ode, func, simplify=False``). (returns -1)
- solution is not solved nor solvable for function. (returns len(str(sol)))
- solution contains unevaluated integral. (returns ∞)

3. **Lie_Group:** currently, this hint implements the Lie group method of solving first-order differential equations. The aim is to convert the given differential equation from the given coordinate system into another coordinate system where it becomes invariant under the one-parameter Lie group of translations. The converted ODE can be easily solved by quadrature. It makes use of the infinitesimals function which returns the infinitesimals of the transformation.

4. **1st_power_series, 2nd_power_series_ordinary,**

2nd_power_series_regular: For now I have decided to implement them with the same approach as SingleODESolver but as suggested by oscar, they can have their own superclass which can be discussed with the mentor and then I will change accordingly.

The Timeline

Pre-GSoC Period

As I don't have any open PRs so from this phase only, I will start refactoring test_ode.py with the above-mentioned idea. Here is an example of my PR
In which I have started refactoring test_ode.py. Although I will try to submit several PRs for the same as it would be easy to review and code which is to be added to the master will get refined more properly by updating in a few parts.

GSoC Period

The tentative timeline for the project is given below. There are 3 phases, apart from the bonding period, in the official timeline of GSoC 2020. I will try to submit PRs quite often and mostly before the scheduled timeline so that reviewing documentation and implementation details become easy and will lead to maintainable code.

This is just a tentative schedule so we can incorporate changes whenever required and I will stick to the timeline.

Bonding Period (4th May - 31st May)

This period comprises 4 weeks. Since I have been contributing to the SymPy for a good amount of time, it would be easier for me to get into the community. So, in this period I will decide the finer details of the workflow with my mentor.

In the remaining two weeks I will start coding as this will provide a head start which is most important in such projects. I have my end semester exams from 17th May to 29th May tentatively. During my exams, I won't be able to submit new PRs but I will constantly discuss the approach for simplification of them and will focus on documentation as they will be quite manageable during my exam time.

Phase - I (1st June - 29th June)

The first phase comprises approximately 4 weeks and is the longest phase. As my exams will get over by 29th May (tentative). I might be behind by 10 days but I will start coding in the bonding period so it will keep me up with the schedule.

During this phase, test_ode.py will get refactored completely. During the initial 2 weeks, I will work on the refactoring of test_ode.py and bug fixing so that all my PRs which I have created from the bonding period get merged.
In later 2 weeks, I will start refactoring individual solvers based on the SingleODESolver.

In refactoring of test_ode, I won't be making PRs for individual solvers but I will manage to refactor 3-4 solvers simultaneously as test refactoring will not require much simplification process like ODE solvers.

I have divided the solvers based on their parent classes whether they will use SingleODESolver or SinglePatternODESolver.

Reason for dividing the same is for pattern matching, I kept in mind that we can extract the elements of our general solution from the equation with direct matching just like First_linear. And for `SingleODESolver` there will be proper logic checking whether the given equation matches or not.

Week-III PRs

1. refactor(dsolve): 1st_power_series
2. refactor(dsolve): 2nd_power_series_ordinary
3. refactor(dsolve): 2nd_power_series_regular

Note: These solvers are to be discussed whether they should have their own superclass.

Week-IV PRs

1. refactor(dsolve): 1st_homogeneous_coeff_subs_indep_div_dep
2. refactor(dsolve): nth_order_reducible

Phase - II (3rdJuly - 27th July)

The second phase comprises around 3 weeks, the shortest phase. The goal for this phase is to complete the refactoring of the ODE module by finishing up solves based on SingleODESolver and start pattern matching solvers and I will also try that all the PRs from phase-I get merged. As the pattern matching doesn't need much simplification that's why it is kept in the last week of this phase.

Week-I PR

1. refactor(dsolve): 1st_homogeneous_coeff_subs_dep_div_indep
2. refactor(dsolve): 1st_homogeneous_coeff_best (This might not be a separate PR as it just selects the best hint from above two methods)
3. refactor(dsolve): 1st_exact
4. refactor(dsolve): Lie_group

Week-II PRs

1. refactor(dsolve): nth_linear_constant_coeff_homogeneous
2. refactor(dsolve): nth_linear_euler_eq_homogeneous

Week-III PRs

1. refactor(dsolve):nth_linear_constant_coeff_undetermined_coefficients
2. refactor(dsolve):nth_linear_euler_eq_undetermined_coefficients
3. refactor(dsolve): nth_linear_constant_coeff_variation_of_parameters
4. refactor(dsolve):nth_linear_euler_eq_variation_of_parameters

Phase - III (1st August - 24th August)

This phase comprises around 3 weeks. The goal for this phase is to complete the refactoring of `ode.py`. Also, I will fix the bugs of individual solvers which will arise after refactoring `test_ode.py`.

I will try to complete this phase work way too early because after refactoring everything the performance enhancement of ODE is to be focused and ordering of hints such that dsolve always returns the correct solution as fast as possible.

Week-I PRs

1. refactor(dsolve): Liouville
2. refactor(dsolve): separable
3. refactor(dsolve): separable_reduced
4. refactor(dsolve): linear_coefficients

Week-II PRs

1. refactor(dsolve): 2nd_hypergeometrics
2. refactor(dsolve): 2nd_linear_airy
3. refactor(dsolve): 2nd_linear_bessel

Note: I have scheduled new PRs up to second last week only because there might be some PRs that are unmerged from previous phases due to various reasons. So on a safer side, it is better to have a plan for such unexpected work. If there aren't any then I will discuss with the mentor for further performance enhancement of dsolve.

Time Commitment

I have my reading vacation from the 30th of April. And as I have been regular with my classes so 1 month of reading vacation for end semester exams is more than enough so I will begin my project work as soon as the accepted projects will be announced. Since, I have no other plans this summer, giving more than 40-45 hours per week to the project will not be hard for me during my summer vacation.
My next semester's classes will commence on 22nd July. However, during the initial month of the semester, courses are just introduced, hence, we have a very little load during that phase.

Also, my university provides us backup classes for the students who have missed because of project works/internships. Hence, I would be able to work without hindrance, during the final phase of the project for around 40 hours per week. I will try to be ahead of deadlines so that we don't face pressure, however, if in case, due to some reasons, we lag behind the projected timeline, I will devote extra time to the project to cover the delay.

Post-GSoC Period

After the completion of the project, I will keep working and contributing to the library. I want to contribute to SymPy as much as possible since SymPy helped me a lot in gaining experience in the open-source community.
Also, I want to be part of the SymPy community as a full-time contributor.

Members of the organization, most importantly, Oscar (oscarbenjamin), Aaron Meurer(asmeurer), Smith(smichr), helped me a lot in contributing to the library by providing reviews to my PRs, their constant help motivated me to keep contributing for so long and I will be connected with the organization in the future also. I would be more than happy to become a mentor in SymPy in the future GSoC programs.

If time permits I would like to work on these additional things:

- Addition of shortcut tricks to solve linear ODEs (this is to be discussed with the mentor)

- Bug fixing in individual solvers after refactoring of `tests_ode.py`

References

- https://github.com/sympy/sympy/issues/18377
- https://github.com/sympy/sympy/issues/18348