Ordinary Differential Equations

Programming Concepts in Scientific Computing
Professor: Guillaume Anciaux

Ahmed Rockey Saikia
ahmed.saikia@epfl.ch

Andras Horkay
andras.horkay@epfl.ch

December 12, 2025

1 THIS IS AN OLD VERSION. PLEASE REFER TO README

2 Scientific Question

This projects aims to write efficient and modular C++ code to solve ordinary differential equations (ODEs) using various numerical methods. The general, non-linear form of an ODE is given by:

$$\frac{dy}{dt} = f(t, y) \tag{1}$$

where y is the dependent variable, t is the independent variable (often time), and $f(t, y)$ is a given function that defines the relationship between y and t. The goal is to implement numerical methods to approximate the solution of such equations over a specified interval. It remains to be seen if an initial value problem (IVP) or a boundary value problem (BVP) will be tackled in this project.

The algorithms that the project will implement will surely include:

- A basic Forward Euler method (explicit)
- A basic Backward Euler method (implicit)
- Multistep Adams-Bashforth method up to 4 steps (explicit)

Other advanced methods may be considered based on time and complexity, such as:

- Runge-Kutta methods (e.g., RK4)
- Backward Differentiation Formulas (BDF)
- Multistep Adams-Moulton method (implicit)

The project is due on **Friday 12th December 2025 at 12:00**. The various criteria are listed in the project description

3 Problem Formulation

3.1 Mathematical Formulation

The project will focus on solving ordinary differential equations (ODEs) of the form given in Equation **??**. The specific problem formulation will depend on whether an initial value problem (IVP) or a boundary value problem (BVP) is chosen. Apart from the different algorithms definitions, there is not much mathematical basis to be formulated here.

Euler's Forward Method is summarised in Alg. **??**, as defined in [**?**].

Algorithm 1 Forward Euler Method

Input: Initial vector $\mathbf{y}_0 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0\,]$, and $t = [\,t_0\,]$
for $i = 0$ to $n - 1$ **do**
    $\mathbf{y}_{i+1} = \mathbf{y}_i + h \cdot \mathbf{f}(t_i, \mathbf{y}_i)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

Adams-Bashforth Method The project will implement Adams-Bashforth methods up to 4 steps. That is, there are four different versions of the method, each using a different number of previous points to estimate the next point. The general formula for the Adams-Bashforth method is given by:

$$y_{n+1} = y_n + h \sum_{j=0}^{k-1} b_j f(t_{n-j}, y_{n-j}) \tag{2}$$

where k is the number of steps (1 to 4 in this case), and b_j are the coefficients specific to each version of the method. The b_j coefficients are chosen in such a way as to obtain the desired order of accuracy for the method. An Adams-Bashforth method with s steps has an order of accuracy of s. The coefficients do not need to be derived from scratch, as they are well-documented in numerical analysis literature. The coefficients taken in this project originate from [**?**]. The first step (1-step) is equivalent to the Forward Euler method, shown in Alg. **??**. The 2-step, 3-step, and 4-step Adams-Bashforth methods are summarised in Algs. **??**, **??**, and **??** respectively.

Notice that the $s \geq 2$ methods require $s-1$ initial values to start the iteration. These initial values can be obtained using a single-step method like the Forward Euler method or any other suitable method, depending on the desired accuracy. For highest accuracy, it is suggested to use a RK4 method to generate these initial values, as it is a 4th order method.

Algorithm 2 2-step Adams-Bashforth Method

Input: Initial vectors $\mathbf{y}_0, \mathbf{y}_1 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0, \mathbf{y}_1\,]$, and $t = [\,t_0, t_1\,]$
for $i = 1$ to $n - 1$ **do**
    $\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h}{2}\left(3\mathbf{f}(t_i, \mathbf{y}_i) - \mathbf{f}(t_{i-1}, \mathbf{y}_{i-1})\right)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

Algorithm 3 3-step Adams-Bashforth Method

Input: Initial vectors $\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2\,]$, and $t = [\,t_0, t_1, t_2\,]$
for $i = 2$ to $n - 1$ **do**
    $\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h}{12}\left(23\mathbf{f}(t_i, \mathbf{y}_i) - 16\mathbf{f}(t_{i-1}, \mathbf{y}_{i-1}) + 5\mathbf{f}(t_{i-2}, \mathbf{y}_{i-2})\right)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

Algorithm 4 4-step Adams-Bashforth Method

Input: Initial vectors $\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0, \mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3\,]$, and $t = [\,t_0, t_1, t_2, t_3\,]$
for $i = 3$ to $n - 1$ **do**
    $\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h}{24}\left(55\mathbf{f}(t_i, \mathbf{y}_i) - 59\mathbf{f}(t_{i-1}, \mathbf{y}_{i-1}) + 37\mathbf{f}(t_{i-2}, \mathbf{y}_{i-2}) - 9\mathbf{f}(t_{i-3}, \mathbf{y}_{i-3})\right)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

Backward Euler Method is an implicit method, which means that at each time step, a root-finding problem must be solved to find the next value. The method is summarised in Alg. **??**, as defined in [**?**].

Algorithm 5 Backward Euler Method

Input: Initial vector $\mathbf{y}_0 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0\,]$, and $t = [\,t_0\,]$
for $i = 0$ to $n - 1$ **do**
    Solve the equation $\mathbf{y}_{i+1} - \mathbf{y}_i - h \cdot \mathbf{f}(t_{i+1}, \mathbf{y}_{i+1}) = 0$ for $\mathbf{y}_{i+1}$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

Runge-Kutta Methods may also be implemented, with the most common being the classical 4th-order Runge-Kutta method (RK4). Without creating too much difficulty for ourselves, we may only implement the RK4 method, but all RK algorithms up to order 4 are included below, see Algs. **??**, **??**, and **??** (order 1 is simply the Forward Euler method). The algorithms were taken from [**?**].

Algorithm 6 2nd-order Runge-Kutta Method (RK2)

Input: Initial vector $\mathbf{y}_0 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0\,]$, and $t = [\,t_0\,]$
for $i = 0$ to $n - 1$ **do**
    $k_1 = h \cdot \mathbf{f}(t_i, \mathbf{y}_i)$
    $k_2 = h \cdot \mathbf{f}(t_i + h, \mathbf{y}_i + k_1)$
    $\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{2}(k_1 + k_2)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

The shortcoming of RK methods is that the value of the function f needs to be evaluated multiple times per time step, which can be computationally expensive for complex functions. However, they offer higher accuracy and stability compared to single-step methods like Euler's method.

Algorithm 7 3rd-order Runge-Kutta Method (RK3)

Input: Initial vector $\mathbf{y}_0 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0\,]$, and $t = [\,t_0\,]$
for $i = 0$ to $n - 1$ **do**
    $k_1 = h \cdot \mathbf{f}(t_i, \mathbf{y}_i)$
    $k_2 = h \cdot \mathbf{f}(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{1}{2}k_1)$
    $k_3 = h \cdot \mathbf{f}(t_i + h, \mathbf{y}_i - k_1 + 2k_2)$
    $\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{6}(k_1 + 4k_2 + k_3)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

Algorithm 8 4th-order Runge-Kutta Method (RK4)

Input: Initial vector $\mathbf{y}_0 \in \mathbb{R}^m$, start time t_0, end time t_{end}, time step h
Output: Approximate solution vectors $\mathbf{y}_i$ at each time step
Initialise $n = (t_{end} - t_0)/h$, $\mathbf{y} = [\,\mathbf{y}_0\,]$, and $t = [\,t_0\,]$
for $i = 0$ to $n - 1$ **do**
    $k_1 = h \cdot \mathbf{f}(t_i, \mathbf{y}_i)$
    $k_2 = h \cdot \mathbf{f}(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{1}{2}k_1)$
    $k_3 = h \cdot \mathbf{f}(t_i + \frac{h}{2}, \mathbf{y}_i + \frac{1}{2}k_2)$
    $k_4 = h \cdot \mathbf{f}(t_i + h, \mathbf{y}_i + k_3)$
    $\mathbf{y}_{i+1} = \mathbf{y}_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
    $t_{i+1} = t_i + h$
    Append $\mathbf{y}_{i+1}$ to $\mathbf{y}$ and t_{i+1} to t
end for
Return $\mathbf{y}, t$

3.2 Identification of Inputs, Outputs, and Data Structures

A key problem in the implementation of the project and the algorithms is that whenever we read an input, we must convert it to the correct data structure. All algorithms take as input an initial vector $\mathbf{y}_0 \in \mathbb{R}^m$ (or multiple initial vectors for multistep methods), start time t_0, end time t_{end}, and time step h. The output of all algorithms is a set of approximate solution vectors $\mathbf{y}_i$ at each time step, along with the corresponding time values t_i. The chosen data structures are as follows:

- Vectors $\mathbf{y}_i$:
- Time values t_i:
- Function $\mathbf{f}(\mathbf{t}, \mathbf{y})$:
- Other parameters (e.g., time step h, start time t_0, end time t_{end}):

We need to then decide what are the possible input formats for the user to provide the necessary data. Possible formats include:

- CSV files
- txt files
- CMD line arguments
- From a web interface (if time permits)?

Finally, we need to decide how to visualise the output data. Possible formats include:

- CSV files
- txt files
- CMD line output
- Plots using a library like Gnuplot in C++ (only for 1D ODEs)?
- Output to a web interface (if time permits)?

4 Algorithms Description

Here is a brief description of the various tasks and milestones that are planned for the project, along with ideas for code structure.

Task 1: Decide data structures Evidently, the first task is to decide on the data structures to be used for storing vectors, time values, and functions. This will involve choosing appropriate C++ data types or libraries (e.g., STL vectors, Eigen library, etc.) that can efficiently handle the required operations. It also needs to be decided how the function $f(t, y)$ will be represented in the code. Options include using function pointers, functors, or lambda functions.

Task 2: Handle input formats This task involves implementing functionality to read input data from various formats. The code should be able to parse CSV files, txt files, and command line arguments to extract the initial conditions, time parameters, and function definitions. The extraction of the data must then convert the data into the chosen data structures, as to simplify all subsequent operations.

Task 3: Handle the solver class and algorithms This task involves creating a solver class that encapsulates the various numerical methods for solving ODEs. The class should have as subclass the different algorithms (Forward Euler, Backward Euler, Adams-Bashforth, Runge-Kutta, etc.) as methods. Each method should implement the corresponding algorithm as described in the problem formulation section. The solver class should also handle the iteration over time steps and store the results in the chosen data structures.

This task may also involve implementing error handling and stability checks to ensure that the numerical methods are applied correctly and efficiently.

Task 4: Handle output formats and visualisation Once the numerical methods are implemented and validated, the next task is to handle the output formats. This involves writing the results to CSV files, txt files, or displaying them on the command line. If time permits, we may also implement visualisation of the results using a plotting library like Gnuplot for 1D ODEs. This will help in better understanding the behaviour of the solutions over time.

Task 5: Create tests and validation cases The test cases will be created to validate the correctness and accuracy of the implemented algorithms. This may involve comparing the numerical solutions obtained from the code with known analytical solutions for specific ODEs. Additionally, convergence tests can be performed by varying the time step h and observing how the solution changes. We should also consider edge cases (and error cases), such as very small or very large time steps, to ensure the robustness of the implementation.

4.1 Data Structures

The project will create a class called `ODE` which will encapsulate a general ODE problem. In a general ODE, we have a function $\mathbf{f}(t, \mathbf{y})$ that defines the relationship between the dependent variable $\mathbf{y}$ and the

independent variable t. Time is generally a `double`, while $\mathbf{y}$ is a vector which should be able to contain `double`, `Complex`, and other types for which ODE problems are generally defined. We therefore need to use templates to be able to handle different data types. Similarly, when solving an ODE numerically, we do not need to actually have access to the function $\mathbf{f}$ itself, but rather a list of its evaluations at different points in time and space. I found there is a neat way in doing this by using `std::variant` from the C++ STL library, which allows us to store different types in a single variable. At first, I will allow the function to be precomputed values tored at different time steps, and also allow functors to be passed. This decision is made so that we can handle the two dataformats: precomputed values from input files, and functions defined in the code itself. Furthermore, functors allow templating and therefore different data types. The decision not to use function pointers or lambda functions is made because they do not allow templating, and therefore we would be limited to only one data type (e.g., `double`). It could be included later, that we can also pass function pointers or lambda functions, but for now, functors should suffice. Apart from what is mentioned above, an `ODE` is associated to an initial value $\mathbf{y}_0$, and a start time t_0.

We need to template the class itself then, to allow the different data types for $\mathbf{y}$ and for the function evaluations. This is achieved by defining the class as `template<typename T, typename F>` where `T` is the data type for $\mathbf{y}$ (e.g., `double`, `Complex`, etc.) and `F` is the data type for the function evaluations (e.g., `std::vector<T>`, functor, etc.).

I would argue that an `ODE` should not have an end time t_{end} or a time step h, as these are properties of the numerical solver rather than the ODE itself. Therefore, these parameters will be passed to the solver class instead. The solver class will be able to take an `ODE` object as input, along with the time parameters, and will then perform the numerical solution accordingly, but more on this later. The class diagram for the `ODE` class is shown in Fig. **??**.

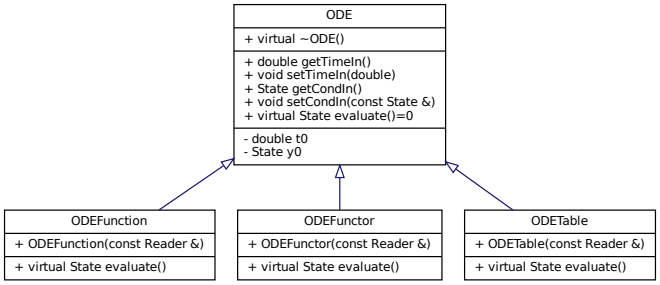

Figure 1: Class diagram for the ODE class.

Questions

I was thinking that I can create a constructor that takes as input a function, then create a sort of wrapper that turns the function into a functor, so I don't need an extra condition in my std::variant. Is this a good idea? Or should I just allow functions to be passed as well, making my variant have three options?

4.2 Input Formats

We need to create a parent class called `Reader` that will handle all input formats. This class will have subclasses for each input format, such as `CSVReader`, `TXTReader`, and `CMDReader`. Each subclass will implement methods to read the specific format and convert the data into the chosen data structures. Fig. **??** shows a simple class diagram for the input handling structure.

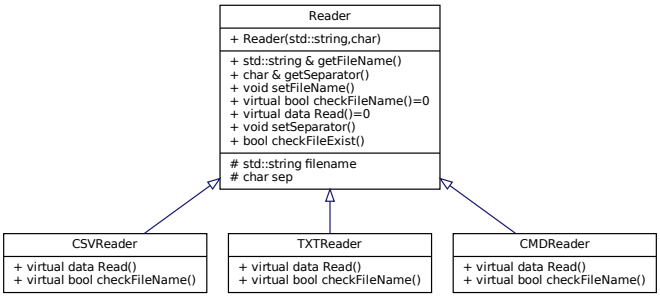

Figure 2: Class diagram for input handling structure.

4.3 Solver Class

Again, we need to create a parent class called `Solver` that will handle all numerical methods. We can further split this into two subclasses: `ExplicitSolver` and `ImplicitSolver`. Each subclass will have methods for the specific algorithms, such as `ForwardEuler`, `AdamsBashforth`, and `BackwardEuler`. Fig. **??** shows a simple class diagram for the solver structure. The reason for splitting into explicit and implicit solvers is that implicit methods often require solving equations at each time step, which may involve additional data structures and methods (e.g., for root-finding).

In the solver class, we have to hold a reference to an `ODE` object, as well as the time parameters t_{end}, h and other parameters like tolerances for implicit methods, max iteration numbers for root-finding, etc. A difficulty arises mainly when thinking of root-finding algorithms for the implic methods. We need to also know the derivative of the function $\mathbf{f}$ with respect to $\mathbf{y}$, i.e., the Jacobian matrix $\frac{\partial \mathbf{f}}{\partial \mathbf{y}}$. This is necessary for methods like Newton-Raphson, which require the Jacobian to update the solution iteratively. Therefore, we may need to extend the `ODE` class to include a method for computing the Jacobian, or we may need to pass it as an additional functor or precomputed values.

Questions

How should we handle the Jacobian problem for implicit methods?

- Should we extend the `ODE` class to include a method for computing the Jacobian?
- Should we pass the Jacobian as an additional functor or precomputed values? both?

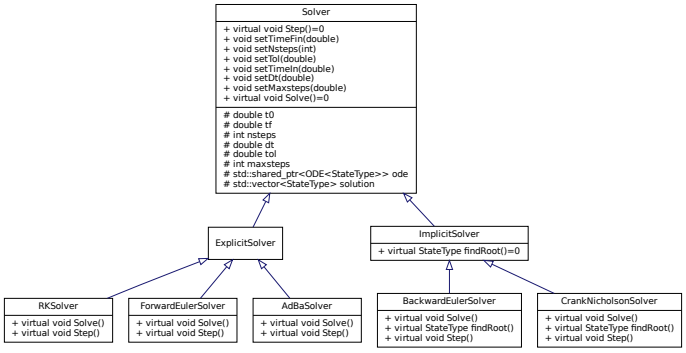

Figure 3: Class diagram for solver structure.

- Should it be an extra argument simply in the solver methods for implicit methods, or should it be a property of the solver class itself?

My initial plan is to pass it as a class attribute to the implicit solver class, as a std::variant that can either be precomputed values or a functor. This way, we can handle both cases where the Jacobian is known in advance or needs to be computed. However, what if someone resets the ODE in the solver? Then the Jacobian may no longer be valid. Should we then make the Jacobian a property of the ODE class itself? But then, what if someone wants to use different Jacobians for the same ODE (e.g., approximated vs exact)?

4.4 Output Formats

Similar to the input handling, we need to create a parent class called `Writer` that will handle all output formats. This class will have subclasses for each output format, such as `CSVWriter`, `TXTWriter`, and `CMDWriter`. Each subclass will implement methods to write the results in the specific format. Fig. **??** shows a simple class diagram for the output handling structure.

5 Implementation Details

We will be hosting on Github, link unknown for the time being. We need to decide on a coding style, and find out if we can use any libraries (e.g., Eigen for linear algebra, Gnuplot for plotting, etc.). We will then need to structure the repository with folders for source code, tests, documentation, and examples. We will also set up a build system (e.g., CMake) to manage the compilation process.

6 Revisions done pre-coding

Here is the final agreed upon outline and data structures before starting the coding phase. These were agreed upon 0n the 28.11.2025 after the class

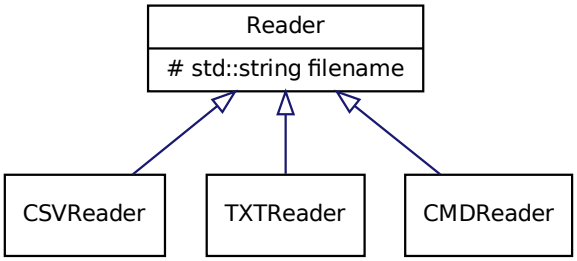

Figure 4: Class diagram for output handling structure.

Our project is structured around four main classes: `Reader`, `ODE`, `Solver`, and `Writer`. First, the `Reader` class is responsible for handling various input formats and creating an object that can be easily used by the `ODE` class. The `ODE` class defines the ODE problem using data extracted from the `Reader` object. The `Solver` class takes in an `ODE`, as well as the numerical parameters needed for solving the ODE (like time step, end time, etc.), and implements the numerical algorithms to solve the ODE. Finally, the `Writer` class takes the results from the `Solver` and writes them to the desired output format.

We choose that the only thing we allow is `vector<double>` as every data structure can be trated as a vector of doubles, through careful conversions when reading the input files. This greatly simplifies the implementation and compilation, as we do not need to template every class and method to handle different data types. Instead, we can focus on implementing the core functionality of reading, solving, and writing ODEs using a single data type. We choose vectors from the STL library, as they are flexible and easy to use for dynamic arrays. We choose them over fixed-size arrays because at compilation time, we do not know the size of the vectors needed for the ODE problems.

6.1 Structures

Here is a summary of some data structures that are necessary for the implementation of the classes.

6.2 The Reader Class

The reader class is possibly the most complex of all as it needs to handle the multiple input formats and create an object that can easily be used by the ODE class. The class diagram is shown in Fig. **??**, but we will explain the decisions and thought process here.

6.3 The ODE Class

The ODE class will take in a Reader object at construction, and extract the necessary data to define the ODE problem. The class diagram is shown in Fig. **??**, but we will explain the decisions and thought process here.

Figure 5: Final class diagram for the Reader class.

6.4 The Solver Class

The Solver class will take in an ODE object at construction, along with the time parameters and other solver-specific parameters. The class diagram is shown in Fig. **??**, but we will explain the decisions and thought process here.

6.5 The Writer Class

The Writer class will take in the results from the Solver class and write them to the desired output format. The class diagram is shown in Fig. **??**, but we will explain the decisions and thought process here.

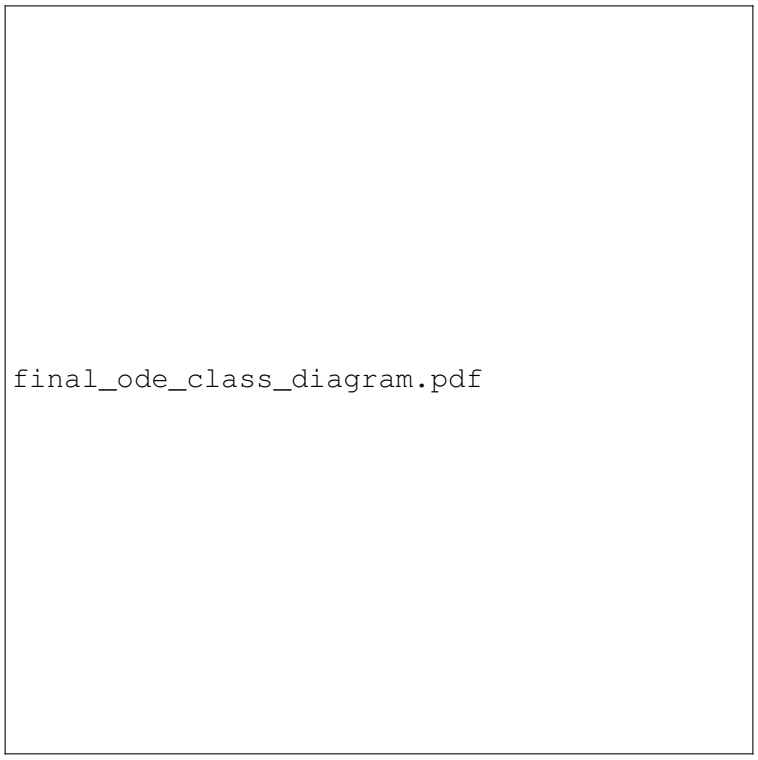

Figure 6: Final class diagram for the ODE class.

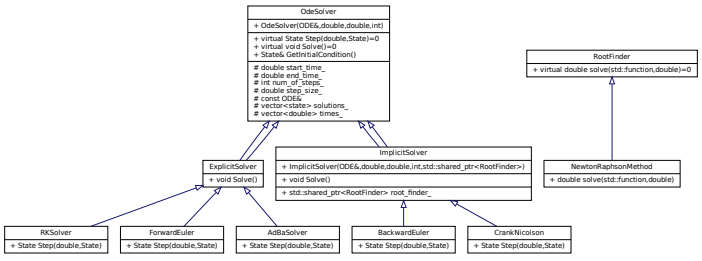

Figure 7: Final class diagram for the Solver class.

Figure 8: Final class diagram for the Writer class.