Ordinary Differential Equations

Programming Concepts in Scientific Computing
Professor: Guillaume Anciaux

Ahmed Rockey Saikia
ahmed.saikia@epfl.ch

Andras Horkay
andras.horkay@epfl.ch

November 19, 2025

1 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

2 Problem Formulation

2.1 Mathematical Formulation

The project will focus on solving ordinary differential equations (ODEs) of the form given in Equation 1. 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. 1, as defined in [2].

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 [3]. The first step (1-step) is equivalent to the Forward Euler method, shown in Alg. 1. The 2-step, 3-step, and 4-step Adams-Bashforth methods are summarised in Algs. 2, 3, and 4 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. 5, as defined in [1].

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. 6, 7, and 8 (order 1 is simply the Forward Euler method). The algorithms were taken from [4].

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 $\mathbf{y}, t$

2.2 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)?

3 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.

3.1 Data Structures

The data structure we use is shown in Fig. 1. This creates an ODE, defined by its function $f(t, y)$, initial condition(s), and time parameters. The data structure also includes methods to evaluate the function at given

points and to retrieve the initial conditions and time parameters.

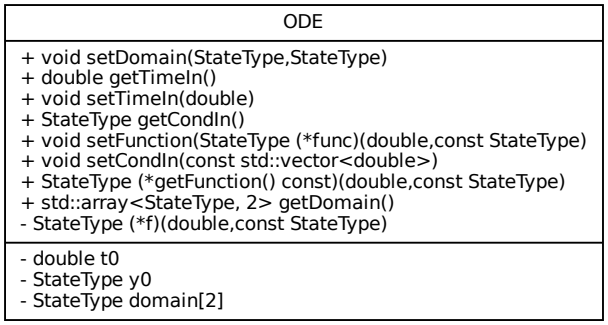

Figure 1: Class diagram for data structure.

3.2 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. 2 shows a simple class diagram for the input handling structure.

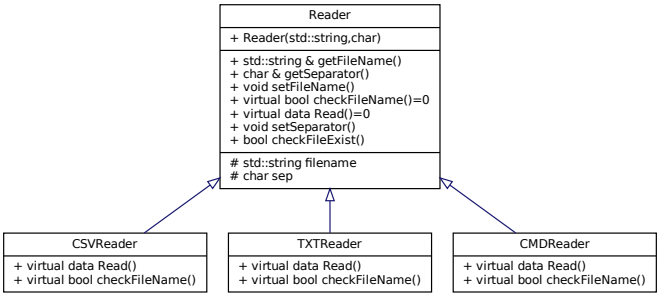

Figure 2: Class diagram for input handling structure.

3.3 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. 3 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).

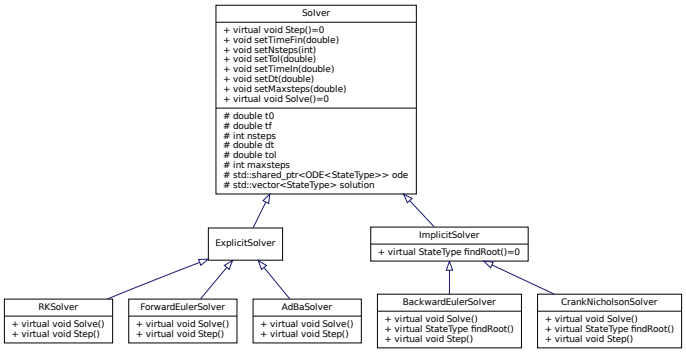

Figure 3: Class diagram for solver structure.

3.4 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. 4 shows a simple class diagram for the output handling structure.

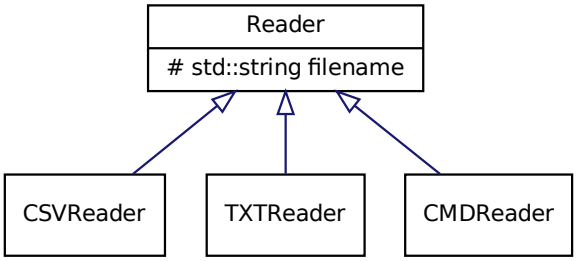

Figure 4: Class diagram for output handling structure.

4 Implementation Details

We will be hosting on Gitlab, 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.

References

[1] Backward Euler method, jul 26 2022. [Online; accessed 2025-11-14].

[2] Forward Euler method, jul 26 2022. [Online; accessed 2025-11-14].

[3] Wikipedia contributors. Linear multistep method — Wikipedia, the free encyclopedia, 2025. [Online; accessed 14-November-2025].

[4] Wikipedia contributors. List of runge–kutta methods — Wikipedia, the free encyclopedia, 2025. [Online; accessed 14-November-2025].