# Programming Concepts in Scientific Computing ODE Solver

Francesco Sala, Nicolò Viscusi

EPFL, 2022-2023

**Abstract**

The project hereby presented focuses on object-oriented programming in C++. An ODE solver was implemented, which enables to solve 1st order ODEs, alongside a thorough documentation in Doxygen, and some GoogleTest test suites.

## Contents

## 1 Introduction

The scope of this project was to implement several 1st order ODE solvers, focusing on object-oriented programming and polymorphism. The resulting program solves Cauchy problems in the form:

$$\begin{cases} y' = f(t, y) \\ y(t_0) = y_0,\ t \in [t_0, t_f] \end{cases} \tag{1}$$

The parameters that define the problem are the solving method, the function $f(t, y)$, the values of $t_0$, $t_f$, the stepsize $h$, the initial value $y_0$, and, for the multistep or multistage methods, the desired number of steps/stages. The methods implemented are the following:

1. Forward Euler, the most simple explicit method;
2. Adams-Bashforth multistep, a linear explicit multistep method, with steps $s \in [1, 4]$;
3. Backward Euler, an implicit method;
4. Explicit Runge-Kutta, multistage explicit method. In this case the specified stages can span from 1 to 4;
5. Backward Differentiation Formulæ, a family of linear multistep implicit methods, with steps $s \in [1, 4]$;
6. Adams-Moulton multistep, a linear implicit multistep method, with steps $s \in [1, 4]$.

These methods allow to compute the numerical solution of the Cauchy problem: for each $t_n$, the value of $y_n$ is computed, with $n = 1, ..., N$, $N = \lfloor \frac{t_f - t_0}{h} \rfloor$. See also [QSG10] for an insight into these numerical methods. Alongside the program itself, documentation of the code was written using Doxygen, and a total of 22 tests were implemented using GoogleTest.

## 2 Cloning the repository and compiling

The repository can be cloned using the following command:

```
~$ git clone https://github.com/Fra-Sala/Project_PCSC.git
```

After moving into the created directory, a second command is necessary in order to be able to run the Google tests:

```
~/Project_PCSC$ git submodule update --init
```

Now, in order to compile the program, it is first necessary to create a directory where the executables will be created:

```
~/Project_PCSC$ mkdir build
```

Such directory can have any name (for CLion, the automatic name is `cmake-build-debug`). At this point, move into that directory:

```
~/Project_PCSC$ cd build
```

Finally, type the following (first, make sure that cmake is installed):

```
~/Project_PCSC/build$ cmake ..
```

```
~/Project_PCSC/build$ make
```

Now, the project is compiled and can be run as:

```
~/Project_PCSC/build$ ./ProjectOdeSolver
```

Similarly, tests can be run as follows:

```
~/Project_PCSC/build$ ./test_main
```

If using an IDE such as CLion, it is then sufficient to compile the target `ProjectOdeSolver` for the ODE solver, `test_main` for the test suites.

## 3 Producing documentation

Documentation is produced using Doxygen. Hence, first of all, one must ensure that Doxygen is installed. To do so, open a terminal and type[1]:

```
~$ sudo apt-get update
```

```
~$ sudo apt-get install doxygen
```

Now that Doxygen is installed, GraphViz should be installed too, to visualize inheritance diagrams:

```
~$ sudo apt install graphviz
```

Now, documentation can be generated either by typing:

```
~/Project_PCSC$ doxygen
```

or by installing Doxywizard. In that case:

```
~$ sudo apt-get install doxygen-gui
```

Now, Doxywizard can be run as follows:

```
~/Project_PCSC$ doxywizard
```

In the window that appears the user has to specify the working directory from which Doxygen will run and the directory where the generated documentation will be put into. In both cases, once the process is completed, two new directories, `html` and `latex`, will be visible. Documentation can be visualized for example by looking for the file `index.html` in the `html` directory:

```
~$ cd html
```

```
~/html$ open index.html
```

## 4 Typical program execution

The user can define all the required parameters in 3 different ways, as described in the following sections.

### 4.1 Guided by the program

If the user runs only the executable (i.e.: `./ProjectOdeSolver`), the program itself will ask what kind of solver one would like to use, the function $f(t, y)$, $t_0$, $t_f$, $h$, $y_0$. If the user selects a method that depends on the number of steps/stages this information will be asked by the program. If the user selects an implicit method, the program asks whether the user wants to use a value for `tol` and `nmax` different from the default values (default values: tol $= 1 \times 10^{-8}$, nmax $= 1000$). What follows is the output if the executable is run:

```
~$ ./ProjectOdeSolver
```

```
 ------------- Ode solver -------------

This program solves the Cauchy problem in the form:
                 dy/dt = f(t,y)
         y in [t0, tf], y(t0) = y0
Please choose a solving method:
```

[1]in Linux.

```
1) Forward Euler
2) Adams Bashforth
3) Backward Euler
4) Explicit Runge Kutta
5) BDF schemes
6) Adams Moulton
```

By inserting a number from 1 to 6, the program will guide the user into the setting of all the other parameters and solution visualization.

#### 4.1.1 Controls on the user input

If the program is run as described above, i.e. the user is guided through the program, simple controls were implemented to check that the user enters valid input. For example, if the user enters a method that is not between 1 and 6, the program will keep asking input until a valid selection in entered. Similarly, if the user happens to mistype $f(t, y)$ and would like to insert it again, the program allows to do it by prompting a question. Additionally, if the user enters a $t_f$ which is smaller than $t_0$ the program asks the user to provide a valid input. One more control is made on the $h$ selected: if the value entered is larger than $t_f - t_0$, the program will ask again to provide a valid input. Finally, one last control concerning the number of steps/stages selected is implemented: in this case, if the input inserted is not between 1 and 4, the program will keep asking to provide input until a valid choice is selected.
Obviously, these controls can only manage simple unwanted inputs: if a letter is provided instead of a number, the program has an undefined behavior.

### 4.2 All parameters specified from command line

The user can run the executable specifying from command line all the required parameters. The calling format must be:

```
~$ ./ProjectOdeSolver method "f(t,y)" t0 tf h y0 steps/stages tol nmax
```

where method is an integer from 1 to 6 as listed above, steps/stages are required only for method 2, 4, 5, 6 and `tol` and `nmax` can be omitted. Note the quotation marks on $f(t, y)$, and that the program can only treat a function where only $t$ and $y$ appear as variables (it is not possible to use a different variable, e.g. $x$). A possible call is given by the following listing.

```
~$ ./ProjectOdeSolver 6 "cos(t)" 0 5 0.1 0 3 1e-9 1e4
```

Here the user opted for Adams-Moulton (6), with $f(t, y) = \cos(t)$, $t_0 = 0$, $t_f = 5$, $h = 0.01$, $y_0 = 0$, number of steps $s = 3$, and specific values of tolerance and maximum number of iterations, `tol` $= 1 \times 10^{-9}$, `nmax` $= 1 \times 10^4$.

### 4.3 All parameters specified from file

The user can run the executable as follows:

```
~$ ./ProjectOdeSolver file "name_of_file.txt"
```

where `name_of_file.txt` is a text file (located in the same folder of the executable) in the format:

```
————————— "name_of_file.txt" ——————————
     method f(t,y) t0 tf h y0 steps/stages
————————————————————————————————————————
```

Once again, `method` is an integer from 1 to 6 as listed above, `steps/stages` are required only for method 2, 4, 5, 6. Note that if using a file, `tol` and `nmax` will have the default values (and cannot be changed).

Evidently, the most user-friendly option is the first one presented.

## 5 Program structure and features

### 5.1 Managing the user input

`ManageInput` class was implemented to manage the user input. Its customized constructor takes as parameters the arguments inserted from command line by the user and sets all the members of the class accordingly, depending on how the user decided to run the program (see section 4). It also contains one method that allocates memory for the particular solver chosen, called `ConstructSolver()`: firstly, it allocates memory for the member `funObject` of the solver class `AbstractOdeSolver` and then, depending on the method chosen (here a `switch` statement is used), a solver object pointer is allocated using the corresponding inherited constructor (from `AbstractOdeSolver`). Hence, such pointer (`ptr`) is returned and the ODE can be solved (`ptr->solve()`).

### 5.2 Implementation of solving algorithms

Class inheritance was used to implement all the solving algorithms. In particular, each algorithm is implemented as a class which inherits either from `AbstractOdeSolver` or, in the case of implicit methods, from `AbstractImplicitOdeSolver`, which in turn inherits from the very same `AbstractOdeSolver`. The corresponding structure is shown in fig. 1.

The `AbstractImplicitOdeSolver` class is implemented in order to store methods to solve non-linear equations. The idea behind this is that, since the equation that must be solved for the implicit solvers is always in the form:

$$g(y_{n+1}) = y_{n+1} + a + bf(t_{n+1}, y_{n+1}) = 0 \tag{2}$$

where $y_{n+1}$ is the unknown, it is only necessary to update at each time step the values of the constants $a$ and $b$, and then to know the evaluation of such expression, to be able to solve numerically the equation[2]. In particular, as sketched in fig. 1, the class `AbstractImplicitOdeSolver` has a method `NonLinearEquation()` which, given a value of $t_{n+1}$ and $y$, returns the evaluation of Eq. 2. The values of $a$ and $b$, protected members of `AbstractImplicitOdeSolver`, are set by the child classes in their `solve()` method at each time-step.

Then, the same `solve()` method calls `SolveNonLinearEquation()`, which in turn will simply call a non-linear equation solver: at present, fixed point iterations and Broyden method are implemented, and are tried in this order. An exception is thrown if fixed point iterations fails to converge[3]. The solution of the differential equation is stored in the member `sol` which is a standard map. The keys are the time-steps, the values are the corresponding $y_n$. This reflects the fact that at each time-step, the solution is defined both by $t_n$ and $y_n$, and therefore it seemed logical to use an object for which each entry is a pair. Additionally, not only the dynamic allocation is handled automatically, but any $y_n$ could be easily fetched given a time $t_n$. Multistep solver objects have a method `NameOfSolverNstep(int NSteps)` which is used to evaluate the new $y_{n+1}$ using the specified number of steps `int NSteps`. This is particularly useful for the first $s - 1$

[2]The value $t_{n+1}$ is treated as a constant and not as an independent variable, since it is always known.

[3]About the convergence check: if the absolute difference between two consecutive $y_{n+1}$ grows for three times in a row, the algorithm is believed to be diverging, and the exception is thrown.

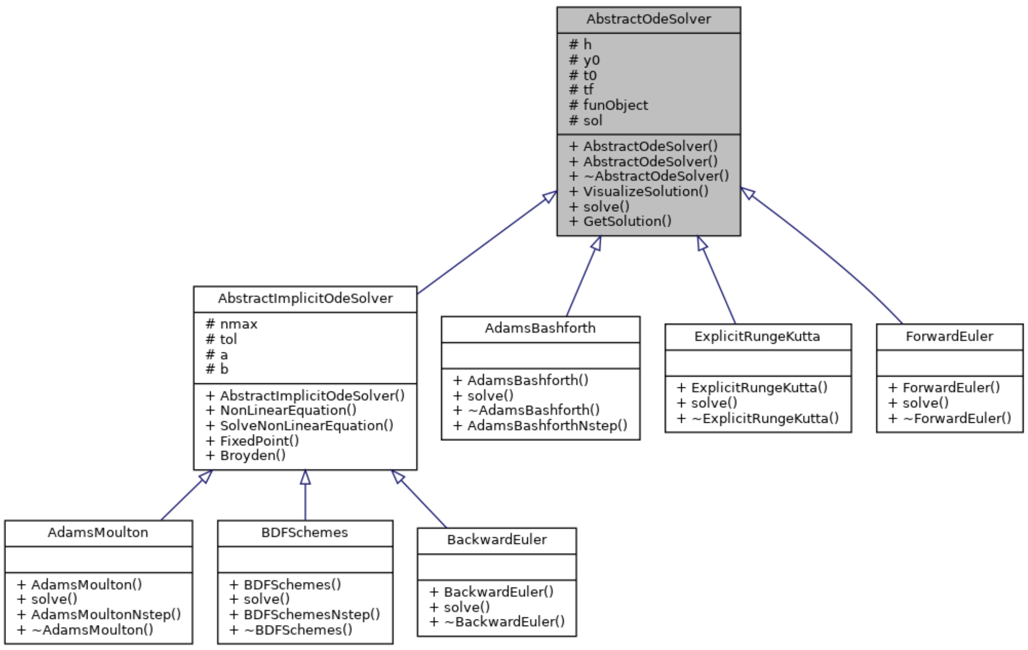

Figure 1: Class hierarchy for ODE solvers (UML generated with Doxygen)

values of $y$: in order to be able to compute them, such method is called, with an increasing number of steps.

## 5.3 About function parsing

In order to be able to evaluate the function passed by the user as a string, a function parser is necessary. In the repository, the folder *fparser4.5.2* contains the headers and source files of this function parser for C++. The program was thought to be potentially extended: the `AbstractParser` class was created for this scope. The child class `Fparser` uses the above mentioned library to evaluate the chosen function: other child classes could be added to use different function parsers. See fig. 2 for the UML inheritance of the function parser.
In particular, here is what happens when the function $f(t, y)$ needs to be evaluated. `AbstractOdeSolver` has a member `AbstractParser* funObject` which is a pointer to an object of type `AbstractParser`, and a virtual method `EvaluateFun(double t, double y)` (overridden by the child class of the chosen solver, e.g. an object `solver` of type `AdamsBashforth`). When a solver-object needs to evaluate the function, it will access such method of the mother class, e.g. `solver.EvaluateFun(t_n, y_n)`, which will make the parsing using the right child class of `AbstractParser`, which will vary depending on what type of parser was used when instantiating the object `solver`[4].

## 6 List of implemented tests

The following is the list of implemented tests, which can be run singularly or all together. The executable `test_main` is to be run to test them all together. Otherwise, by accessing the file source `test/test.cc` it is possible to run each test individually.

- A set of 18 tests, one for each solving method: in this case, the function $f = \cos(t)$ is used to set the Cauchy problem, alongside the parameters $t_0 = 0$, $t_f = 1$, $y_0 = 0$,

[4]Note that at the moment, at line 134 in `ManageInput.cpp` the function object is declared as a pointer of type `Fparser`. If a new parser were to be added, then this instantiation would need to be changed, potentially according to a choice made by the user.

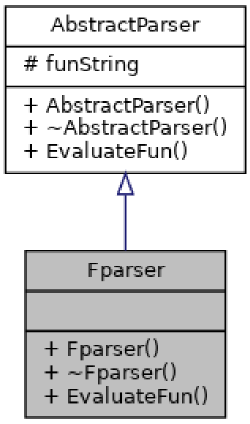

Figure 2: Class hierarchy for function parsing

$h = 0.01$. Then, using a `for_each` loop, it is checked through an `ASSERT_NEAR` that at each time step the computed solution is sufficiently close (within $1 \times 10^{-2}$) to the exact solution $y = sin(t)$. To test the solvers, test fixture was exploited, since the preparation of each solver is always the same. A class `AbstractMethodTest` that inherits from `::testing::TestWithParam<int>` has been created. Here, the previous problem is set, i.e. every member is initialized. Subsequently, for every solver a child class that inherits from `AbstractMethodTest` is created: in this way, when a new solver is implemented, only one new child class is to be created. Multistep/multistage methods are tested with parameterized tests: using `INSTANTIATE_TEST_SUITE_P` and the method `GetParam()` allows to set the same solver with a different number of steps/stages. The UML diagram of such classes is shown in fig. 3;

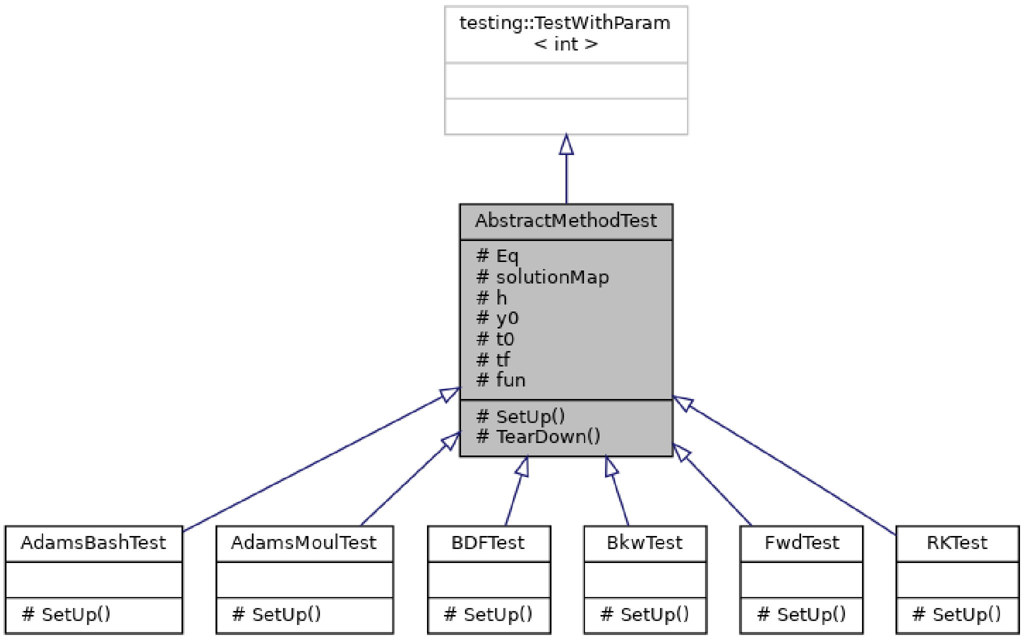

Figure 3: Class hierarchy tests (UML generated with Doxygen)

- 2 tests, to check the behaviour of the class `ManageInput`. These tests were made to ensure that the class `ManageInput` has the expected behavior when the user provides a file to be read or in the case one provides every argument from the command line directly. In the former case, a dummy file is produced and a the associated problem is set. Thus, the program is run and such file named `test.txt` is read by the program. The test checks that the solution obtained is sufficiently close to the one expected.

As a consequence, this feature of `ManageInput` can be considered safe. The test concerning arguments passed from the command line is analogous;

- 2 tests, one for `FixedPoint()` and one for `Broyden()` method. A one-timestep problem is solved using Backward Euler. Given $t_0 = 0$, $t_f = 0.5$, and $h = 0.5$, and $f(t, y) = cos(t)$ for fixed point iterations, $f(t, y) = y$ for Broyden, the exact solution of Eq. 2 is known, since it becomes a linear equation. Once again, `ASSERT_NEAR` is exploited. The choice of $f(t, y) = y$ is made so as to force fixed point iterations to fail to convergence, and therefore switch to the Broyden algorithm.

## 7 About visualization of the solution

The user can visualize the solution in two different ways. Once the solution is computed, the program will prompt the following message:

```
Where would you like to print the solution?
1) Screen
2) Matlab file
```

If option 1 is selected, the solution is printed to the standard output as two columns vector, one for the timesteps and the corresponding solution at each time-step. Here an example:

```
Where would you like to print the solution?
1) Screen
2) Matlab file
1
The solution is the following:
-----------------------------------
        t        ||        y
-----------------------------------
0.000e+00        ||      1.000e+00
1.000e-01        ||      1.100e+00
...
```

If option 2 is selected, a MATLAB `solution.m` file is created in the same folder where the executable is located. Such script is already formatted with the right MATLAB commands, and needs only to be run to produce the plot of the solution to the given Cauchy problem.

## 8 Future work and issues

The solver may be extended in many ways. First of all, about the visualization of the solution, a C++ library for plots and graphs may be added, in order to make the user able to visualize the solution directly when running the executable, without having to use an external software (such as MATLAB). An other possibility would be to write scripts for other languages to plot the solution (e.g.: `python`). Another feature that could be added is the possibility to specify a text file to be used as output for the solution. The problem of security was not particularly taken into account when realizing this project: at the moment, if the user does not follow the above instructions when setting the parameters (e.g. a letter is given as input for $t_0$), the behavior of the program becomes unpredictable. Finally, more in-depth tests could be implemented: specific features of the program could be tested, such as successful dynamic allocation of solver objects, or freeing of memory. Other ODE solvers could be added, along with other non-linear equation solvers. Finally, additional function parsers could be added too.

## References

[QSG10] A. Quarteroni, F. Saleri, and P. Gervasio. *Scientific Computing with MATLAB and Octave*. Texts in Computational Science and Engineering. Springer Berlin Heidelberg, 2010.