# PCSC
# Vectorial ODEs

Daniele Hamm, Lionel Constantin

December 2019

This report describes a project that tackled the solution of initial value problems for vectorial ODEs of the form:

$$\boldsymbol{y}'(t) = \boldsymbol{A} \cdot \boldsymbol{y}(t) + \boldsymbol{g}(t).$$

where $\boldsymbol{A}$ is the right hand side (RHS) matrix and $\boldsymbol{g}(t)$ is the RHS function.

The following paragraphs discuss the main features of the library created for this purpose and show how it can be used.

## 1 Compiling the program

The program files are compiled as a library called ODElibrary. It can be added to any other C++ program by linking the library using cmake `https://cmake.org/`. To compile the code, do in a terminal open in pcscproject :
*cmake CMakeLists.txt*
*make*
This will create executable for the tests and also a *pcscproject* executable, that is calling the library.

### 1.1 External libraries needed

- Eigen : `https://eigen.tuxfamily.org/index.php?title=Main_Page`
- GoogleTests : `https://github.com/google/googletest`

## 2 Program execution flow and usage

The main function is called this way :
*pcscproject inputfilename outputfilename*
The argument *inputfilename* is mandatory, *outputfilename* is optional, if not specified the output file is called *"solution.dat"*. The input file contains all the information necessary to perform the integration of the ODE problem at hand. Specifically, it is organized by lines, each of which contains a string describing the

parameter and, separated by a space, its value/s. The input file parameters are summarized and described in the following table. It's important to remark that *initialValue*, *rhsFunction* need a number of values equal to *problemDimension*; *rhsMatrix* needs *problemDimension*$^2$ values. In the output file, the first column

| Parameter | Meaning and values |
|---|---|
| solver | 1.ForwardEuler, 2.RungeKutta, 3.AdamsBashforth, 4.AdamsMoulton, 5.BDF |
| initialTime | Starting time of integration. |
| finalTime | Ending time of integration. |
| numberOfSteps | Integration steps to be performed, integer >1 |
| stepSize | Discretization parameter |
| discretizationMethod | Define discretization using: 0.numberOfSteps, 1.stepSize |
| problemDimension | Dimension of ODE problem |
| initialValue | Values of vector of initial conditions, all on same line, separated by a space. |
| rhsFunction | Integers defining rhsFunction g(t), all on same line, separated by a space. $g_i(t)$: 0=0, 1=$t$, 2=cos($t$). Any combination can be chosen. |
| rhsMatrix | Values of rhs matrix, all on same line, separated by a space. |
| stepsMultistep | 1-5, number of steps of multistep methods. Ignored if solver $\neq 3, 4, 5$ |
| orderRK | 1-4, order of RungeKutta method to be applied. Ignored if solver $\neq 2$ |

corresponds to the time, and the next columns to the approximation of the solution.

The library ODElibrary was originally developed to be called from other C++ files; the implementation of the input file reading was added at the end and is not fully tested.

## 3 Features

### 3.1 Variety of solvers

Several solvers are implemented, both one-step and multistep, explicit and implicit. Specifically, implementations for Forward Euler, Runge Kutta, Adams Bashforth, Adams Moulton and BDF are provided.

### 3.2 Overall structure and its extensibility

The AbstractSolver class wraps up variables and methods common to all solvers. A MultistepAbstractSolver class, daughter of AbstractSolver, contains the features shared by the multistep methods implemented. A specific class for each method is then implemented. The one-step methods are daughter classes of AbstractSolver, the multistep ones of MultistepAbstractSolver. This class subdivision allows to factor together all what is not method-dependent. Such feature guarantees code maintainability, compactness and simple extensibility.

The methods are defined by how the update is performed at each integration step. A purely virtual function step() is defined inside the AbstractSolver class, and every solver overrides it providing a specific implementation for it. A solve() method iterates over the necessary time steps, calling the proper step() function at each iteration. The approximations of the exact solution computed at the times defined by the discretization are written to an output stream specified by the user, whose default is terminal. Any additional handling needed is managed by the class of the specific solver being used. This fairly simple approach allows the library to be easily extended to other integration methods.

### 3.3 Handling the inputs

When initializing a solver, several checks are performed on the parameters defining the integration to be carried on. Exceptions are thrown if invalid inputs are detected. This should prevent some possible reasons of failure of the algorithms and notify the user with error messages to help understanding what might be going wrong.

The time step can be defined in two different ways, to guarantee more flexibility: by setting a number of steps or by setting it explicitly. This may lead to problems if the step size is not a submultiple of the integration interval. It is therefore preferable to set the number of integration steps to be performed and let the library handle the computation of the step size.

For any needed vectorial operation the external library Eigen is used. The high quality and careful operation handling that it provides are exploited. Additional checks are performed on the dimensions of the elements involved, raising specific exceptions to help the user understand what was not properly set.

To generate the function from input in files, it was decided to use a simple notation, where a number correspond to a function. This was done by having a class that is able to generate *std::function* objects in *function.cpp.* Adding a different function is easy, as one can create it and add it to the *enum* that lists all the functions; adding it to the parser and getter methods following the example of the already implemented functions completes the job.

## 4 Tests

Methods of the AbstractSolver mother class are tested multiple times, once per each solver, to catch undesired modifications by the daughter classes.

Tests to check that good inputs are correctly set are provided, and also tests to check that bad inputs throw specific exceptions.

The class RungeKutta has specific tests for the setting of the method order. Another test checks if the error becomes smaller at the increase of the method's order.

Convergence of the methods is tested checking the accuracy of the approximation to the solution of a test problem corresponding to a mass attached to a spring with dampening and forced oscillations.

## 5 TODO's and perspectives

In section 3.3, it was discussed how to read the RHS function. Having a parser that would read from file and create the function object would allow a very general usage and more flexibility.

Better tests for convergence should be designed. In the tests the accuracy of the approximate solution at the final value is checked, but this is not at all a stringent condition for convergence. Plotting the exact solutions of the test problems along with the approximated ones, the "eye-test" tells there is convergence to the exact solution, but something more sophisticated should be designed.

Some specific tests for the multistep methods should be designed, since some of their specific functionalities are not tested by the testsolver, e.g. the method to compute missing initial values in order to initialize the solver. Such methods were tested by hand and they work, but a formal test should be designed and implemented.

Tests of the inputs of the main should be written. In particular, a check that the given parameters from the file are in the correct format.