Programming Concept in Scientific Computing : Ordinary Differential Equations

Nicolas Lesimple - Caroline Violot

***Abstract*—The goal of this project is to implement explicit methods to solve vectorial Ordinary Differential Equation (ODEs). Thus, in this report, an introduction of the overall vectorial ODE problem will be done. Then, in a second time the program compilation and the program execution will be explained. In a third part, an overview of all the classes implemented with the corresponding hierarchy will be described and implemented tests will also be illustrated. The report will be concluded by showing the several issues faced during this project and the potential improvements that can be added. Thus, this ReadMe file explains how to use the implemented solvers. Html and Latex documentation were created thanks to Doxygen. Everything is stored in the Documentation folder. If the user wants to re compile it, he just has to enter the following command in the project folder : doxygen Doxyfile.**

I. Introduction/Problem

A differential equation is a mathematical equation that relates some function with its derivatives. In applications, the functions usually represent physical quantities, the derivatives represent their rates of change, and the equation defines a relationship between the two. Because such relations are extremely common, differential equations play a prominent role in many disciplines including engineering, physics, economics, and biology.

In pure mathematics, differential equations are studied from several different perspectives, mostly concerned with their solutions - the set of functions that satisfy the equation. Only the simplest differential equations are solvable by explicit formulas; however, some properties of solutions of a given differential equation may be determined without finding their exact form.

If a self-contained formula for the solution is not available, the solution may be numerically approximated using computer methods. In this last case, Adam-Bashforth or Runge-Kutta methods play an important role : they are the definition of the solver of the system.

As we said in the abstract, we focused on vectorial ODEs, only using a linear function and we implemented several different methods to approach a solution.

The vectorials ODE are in the form :

$$y'(t,x) = A * x + g(t)$$

II. Compile the program

In the CMakeList.txt, the main executable is called main. Three test cases were implemented in the executable using different systems but with different parameters. The overall number of step is fixed to 10 and thus this output solution matrix will have 10 columns.

The user will be able to defined some parameters of the simulation but the prescribed inputs are:

- **InitialCondition** : Each column corresponds to one initial condition for each variable. For Bashforth solvers the number of columns should match the order of the solver used. For Runge-Kutta just one initial condition suffices.

- **StateMatrix.csv**: Defines the coefficient for each variable of the system. The RHS of the ODE is taken to be $A * x + g(t)$, A is the state matrix.

- **FunctionMatrix.csv**: The RHS of the ODE is taken to be $A * x + g(t)$, g is the FunctionMatrix. Each column corresponds to one time step. The other parameters needed to be initialized, the system can be ask to the user or taken directly from the code.

In fact, when user will run the program, he will directly be asked to choose one option between the following three :

- **Keyboard** : User will need to enter with the keyboard some parameter of the simulation. Timestep, NumberOfStep, Order, Dimension and WriteOutputTimestep are choosen by the user thus it allows him to have a big set of way to solve the system. InitialConditionMatrix, StateMatrix and FunctionMatrix are respectively : $\{\{6\},\{4\}\}$, $\{\{1,-1\},\{-1,1\}\}$ and a 2x10 matrix of zero. The solution of this defined system is :

  $$5 + \exp 2 * t \tag{1}$$

  and

  $$5 - \exp 2 * t \tag{2}$$

  If the user choose this option, it is more likely that the program stops due to an assertion. In fact, input have rules to be well defined and if the user did not follows theses rules, the program will throw assertions. Moreover, just in this keyboard case the user will have to enter the entire path to an output csv file to allow the program to save the solution in it. If the path is not correct, an assertion will be thrown.

- **SimpleTest** : User will just have to select the method he wants to use and the output will be printed in the command line. Timestep = 1, NumberOfStep = 10, Order = 1, Dimension = 4 and WriteOutputTimestep

= 1. Moreover, InitialConditionMatrix, StateMatrix and FunctionMatrix are respectively : $\{\{2\},\{2\},\{2\},\{2\}\}$, $\{\{0,0,0,0\},\{0,0,0,0\},\{0,0,0,0\},\{0,0,0,0\}\}$ and a 4x10 matrix of zero. The solution of this defined homogeneous system where g(t)=0 is : y(t) = 2.

- **ComplexTest** : User will just have to select the method he wants to use and the output will be printed in the command line. The solution of the system is more complex as g(t)=cos(t). Timestep = 1, NumberOfStep = 10, Order = 1, Dimension = 2 and WriteOutputTimestep = 1. Moreover, InitialConditionMatrix, StateMatrix and FunctionMatrix are respectively : $\{\{0\},\{0\}\}$, $\{\{0,0\},\{0,0\}\}$ and $\{\{1, 0.54, -0.41, -0.99, -0.66, 0.28, 0.96, 0.75, -0.14, -0.91\}, \{1, 0.54, -0.41, -0.99, -0.66, 0.28, 0.96, 0.75, -0.14, -0.91\}\}$ The solution of this system is more complex as the system is not homogeneous.

Thus after obtaining the solution, the output can be compared to the 'groundthruth equation' values to see the performance of each method on the different problems, varying time step parameters. If you put for each run the same filepath, output goes to the same file in the Output folder. Thus when the program is run, the solution of the previous run will be erase to save the new one.

III. Typical program execution

A. Input

The first step of the program execution is the declaration and the initialization of the Input object. In fact, Input object contains all the information needed to define and solve the system of differential equation. In fact, upon executing, the user needs to input the following values (all the italics writing are the requirements per variable) :

- **Timestep** - *DOUBLE* - Defines the size of the time step in seconds. This variable can be a double or a integer.

- **System dimension** - *INTEGER* - Defines the dimension of the system, which means the different number of states. Thus, the number of rows in the input matrices that defines the system needs to be superior or equal to this dimension value. If the input matrices have additional rows, they are ignored.

- **Solver Order** - *INTEGER* - Defines the order of the solver used. By taking into account the several methods implemented in the project, this number need to be in the range of [1,4].

- **Overall Number of Timesteps** - *INTEGER and need to be aa multiple of WriteOutputTimestep* - Defines the total number of steps considered for one system. This value should be at most the number of columns in the FunctionMatrix. Like before, if it is lesser the additional values are ignored. Moreover, a distinction needs to be done between the 2 implemented methods : in opposition to Adam-Bashforth method, Runge-Kutta will need one additional time step. For example, if the integration is carried out for 10 steps, the FunctionMatrix will need to contain 10+1 values which is equal to 11 columns.

- **WriteOutputTimestep** - *INTEGER* - Defines when the solution is written in the solution object saving the different step of the solver . This is a kind of interval of time steps or a quantity of time step we need to wait before this event.

In addition to theses variables, input object needs three matrices. Theses 3 matrices information are hard coded in the program to allow a working demo of our project. This information can be easily changed in the input object constructor or in the main (depending on the example you choose).

- **InitialConditionMatrix** - *VECTOR<VECTOR<DOUBLE>>* - Defines the matrix containing the initial condition of the system. The number of rows of this matrix needs to be superior or equal to the system dimension. Otherwise, an assertion will be thrown.

- **FunctionMatrix** - *VECTOR<VECTOR<DOUBLE>>* - Defines the matrix containing the values of the function defining the system for all the step we want to compute. The expression of this function can transform the system from a homogeneous ODE to a inhomogeneous ODE. In fact if this function is always 0, the system is in the case of homogeneous solution. If not, which means if the function is not null, the system is inhomogeneous.

- **SystemMatrix** - *VECTOR<VECTOR<DOUBLE>>* - Defines the matrix containing the information to declare and construct the RHS (rigth hand side) part of the ODE.

To define the input, two constructors were made. One of them is the simplest one where user need to specify each variable before in the code and put it as argument in the constructor. This is done in Input class. This is a kind of 'hardcoded usage'. Another way to declare this input is just to declare a Input class. The constructor don't have any arguments but user will need to enter the parameters he wants for the calculations in the command line. Thus in this case user needs to write information on the interface.

B. Solution

Filename - *STRING* - Moreover, at the end of simulations where user choose variables, the solution is written in an output file. Thus, the program needs the output filename. Be careful about the filename : the program do not only need the filename but it needs the entire filepath, the filename and the format of the output file in the same string. Here is a example : C:/Users/nicol/Desktop/Group15 /pcsc_VectorialODE_group15/Output/solution.csv. In fact, if you don't define the path, errors can happen and the file can

be created in unwanted folder of the computer. The advised format is .csv. Each column of the output corresponds to the state at print time. If a file of the same name already exists, the previous content is erased.

IV. List of features

A. Overview

To represent the matrix input needed to define the system, we decided to use the vector class. In fact, thanks to this class, a vector can be define as vector<double> and a matrix can be defined as vector<vector<double>>. One problem of doing that is the fact that usual operator like + or - or * can not be applied between this type of variable. Thus we need to create function allowing this kind of operation. This was done in the VectorialODE class which includes all members of the classes while children classes implement the solver method. Two general solvers are inherited from VectorialODE class which are Runge-Kutta and Bashforth class. Theses classes are abstract classes making difference between the two type of solver. Then, the individual solvers are inherited from one of these two parent classes. As we said before, all variables needed to initialize the system are stored in an input object coming from Input class and the solution is stored in a solution object coming from Solution class.

B. Input

This class is used to keep all the information about the system of ODE the program has to solve. This class defined several public methods such as constructors, a destructor, few getters and an asking function. The input is never modified during the execution of the program. The more important functions are the two different constructors. The first one takes as arguments all the variables described above and thus everything need to be hard coded in the main function of the executable. In this case, the user is not asked to enter any information and the solver is based on hard coded information. The other constructor do not need any argument. In fact, during the initialization of the input object, when the constructor is called, input variables will be asked to the user. He will have to enter the desired simulation parameter in the interface thanks to the keyboard. The 3 matrices input (which are InitialConditionMatrix, StateMatrix and FunctionMatrix) and the order were hard coded in this function to show a defined example of usage of our code.

C. Vectorial ODE

The Vectorial ODE class is a virtual class as the method SolveVectorialODE will be defined in the child classes. However, this class is used to implement all the algorithms used to solve the differential equations. Two types of solver were defined with two virtual classes which are Bashforth and Runge-Kutta. Constructors of theses classes are used to create assertions and thus to check coherence of the variables.

The vectorial ODE class constructor needs two argument : an input object and a solution object to define the system and write the solution.

1) Bashforth class: The Bashforth classes inherit from the VetorialODE class, an abstract class with a virtual SolveVectorialODE method, facilitating the definition of its inherited classes.

The Adams Bashforth method is an explicit linear multistep method. It can be made of different order (up to four in our case) where the order is the number of previous results (y_{n-1}, y_{n-2}, ...) used to approach the current step result (y_n). It implies the need of different vectorial operations (addition, multiplication with scaler, etc.) and the need to keep a few of the previous solutions in memory.

Therefore the general Bashforth class from which the different Bashforth classes inherit include a few methods for handling the vectorial operations.

Then in each Bashforth*Number*Step classes the method SolveVectorialODE is implemented using the following definitions :

- *BashforthFirstStep* : $\vec{x}(t_{k+1}) = \vec{x}(t_k) + hf(x(t_k), t_k)$
- *BashforthSecondSteps* : $\vec{x}(t_{k+1}) = \vec{x}(t_k) + \frac{3h}{2}f(x(t_k), t_k) - \frac{h}{2}f(x(t_{k-1}), t_{k-1})$
- *BashforthThirdSteps* : $\vec{x}(t_{k+1}) = \vec{x}(t_k) + \frac{23h}{12}f(x(t_k), t_k) - \frac{4h}{3}f(x(t_{k-1}), t_{k-1}) + \frac{5h}{12}f(x(t_{k-2}), t_{k-2})$
- *BashforthFourthSteps* : $\vec{x}(t_{k+1}) = \vec{x}(t_k) + \frac{55h}{24}f(x(t_k), t_k) - \frac{59h}{24}f(x(t_{k-1}), t_{k-1}) + \frac{37h}{24}f(x(t_{k-2}), t_{k-2}) - \frac{3h}{8}f(x(t_{k-3}), t_{k-3})$

2) Runge-Kutta: The Runge-Kutta classes inherit from the VectorialODE abstract class as well.

This method is also an explicit iterative method to solve ODEs, but there is only a single step needed (y_{n+1} only depends on y_n). Indeed, instead of using multistep to get closer of the accurate solution, Runge-Kutta methods are based on evaluating the function a certain number of time at each step, this number being the order of the method. The solution of each step is then a combination of the different evaluation of the function at this step.

In each Runge-Kutta*Number*Step classes the method SolveVectorialODE is implemented using the following definitions :

- 2nd order Runge-Kutta (*RK2*) : $\vec{x}(t_{k+1}) = \vec{x}(t_k) + hk_2$ with $k_2 = f(x(t_k) + \frac{hk_1}{2}, t_k + \frac{\Delta t}{2})$ and $k_1 = f(x(t_k), t_k)$
- 4th order Runge-Kutta (*RK4*) : $\vec{x}(t_{k+1}) = \vec{x}(t_k) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ with $k_1 = f(x(t_k), t_k)$, $k_2 = f(x(t_k) + \frac{h}{2}k_1, t_k + \frac{h}{2})$, $k_3 = f(x(t_k) + \frac{h}{2}k_2, t_k + \frac{h}{2})$ and $k_4 = f(x(t_k) + hk_3, t_k + h)$

D. Solution and output file

This class is made to store the solution for each writing step and to save it in a output file. The solution class contains just a few private features :

- *number_of_rows* : an integer representing the number of row wanted in the solution matrix. This number needs to correspond to the system dimension.

- *number_of_columns* : an integer representing the number of columns wanted in the solution matrix. This number need to correspond to the overall number of time steps.

- *solutionODE* : a matrix (vector<vector<double>>) where the solution will be stored.

In addition to constructors, destructors and getter public methods, it also has a ModifySolutionByColumns method to write values by columns in the solutionODE variable for each writing timestep and a SolutionToFile method which exports the solution in a csv file (our output file). The getter allows to access private attribute.

V. List of tests:

Googletest was used to test a majority of the function of the program. In fact, it is a cross platform system that provides automatic test discovery. In other words, we do not have to enumerate all of the tests in our test suite manually. It supports a rich set of assertions such as fatal assertions (ASSERT_), non-fatal assertions (EXPECT_), and death test which checks that a program terminates expectedly. In total 16 tests have been implemented. To make it run, the user needs to compile the executable called Test. All the tests should run and the user should see the output of all the test : how many failed and how many passed.

Here is a list of the implemented test.

- *MultiplyMatrixScalar* : Test the function MultiplyMatrixScalar() defined in the VectorialODE class.
- *Addition* : Test the function MultiplyMatrixScalar() defined in the VectorialODE class.
- *MultiplyMatrixScalar* : Test the function Addition() defined in the VectorialODE class
- *MultiplyMatrixVector* : Test the function MultiplyWithVectorByRight() defined in the VectorialODE class.
- *MultiplyVectorAndScalar* : Test the function MultiplyVectorAndScalar() defined in the VectorialODE class.
- *AddTwoVector* : Test the function AddTwoVector() defined in the VectorialODE class.
- *SubtractTwoVector* : Test the function SubtractTwoVector() defined in the VectorialODE class.
- *BashforthFirstStep* : Test the function SolveVectorialODE() defined in the BashforthFirstStep class.
- *BashforthSecondStep* : Test the function SolveVectorialODE() defined in the BashforthSecondStep class.
- *BashforthThirdStep* : Test the function SolveVectorialODE() defined in the BashforthThirdStep class.
- *BashforthFourthStep* : Test the function SolveVectorialODE() defined in the BashforthFourthStep class.
- *RungeKuttaOrder2* : Test the function SolveVectorialODE() defined in the RungeKuttaOrder2 class.
- *RungeKuttaOrder4* : Test the function SolveVectorialODE() defined in the RungeKuttaOrder4 class.
- *GetColumnsOfMatrix* : Test the function GetColumnsOfMatrix() defined in the vectorialODE class.
- *GetSolutionValueFromIndex* : Test the function GetSolutionValueFromIndex() defined in the Solution class.
- *ModifySolutionByColumns* : Test the function ModifySolutionByColumns() defined in the Solution class.
- *ModifySolutionByColumns* : Test the class Input by testing all the getter in teh same test.

We try to apply the concept of unit testing and thus we try to reach a high percentage of coverage of the code. Here, the more important tests are the one testing the solver definition like BashforthFirstStep or RungeKuttaOrder2. To test them, we used the 'simple' hard coded example. In fact output coming from this system definition should always be 2. Thus, it makes things easy to test.

Performing these tests allow to show that the algorithms are properly implemented and that the approximation using proper time step and solver is correct.

VI. Issues and perspectives

The most important issue we faced was to choose the format of input we used to define our system. Theses three matrices representation comes with the definition of the RHS found on several online documentations. This way of doing is optimal for the program but the declaration of theses matrices is harder. In fact, for now, user can choose some parameters of the simulation but all theses input matrices are hard coded in the code. The InitialConditionMatrix and the StateMatrix could have been asked to the user using interface and command line. However, the problematic part is the FunctionMatrix. Indeed, in our project, it is impossible to produce this kind of matrix. The solution could be to create a dictionary with keys corresponding to a string describing the function and values of the dictionary would be the FunctionMatrix. External libraries could also be used to define this matrix automatically. In addition, more solver could have been implemented like Runge-Kutta order 3.