Differential Equations Computational Practicum Report - Variant 12

Tymur Lysenko

November, 2019

Contents

1 Solution to the provided differential equation

Solve the following initial value problem analytically:

$$\begin{cases} y' = 5 - x^2 - y^2 + 2xy \\ y(0) = 1 = y(x_0) = y_0 \end{cases} \tag{1}$$

The differential equation from the IVP above is a **Riccati** equation:

1. Find its any non-trivial particular solution y_1. Suppose the particular solution is of the form:

$$y_1 = c_1 + c_2 x \tag{2}$$

then

$$y_1' = c_2 \tag{3}$$

Substituting (2) and (3) into the (1) and rearranging the terms yields:

$$c_2 + c_1^2 + 2c_1c_2x + c_2^2x^2 - 2c_1x - 2c_2x^2 = 5 - x^2$$

Grouping of terms with the corresponding coefficients produces:

$$(c_2^2 - 2c_2)x^2 + 2c_1(c_2 - 1)x + c_1^2 + c_2 = 5 - x^2$$

To find c_1 and c_2 solve the below system of equations follows from the above equation:

$$\begin{cases} c_2^2 - 2c_2 = -1 \\ c_2 - 1 = 0 \\ c_1^2 + c_2 = 5 \end{cases} \iff \begin{cases} c_2 = 1 \\ c_1^2 + 1 = 5 \\ 1 - 2 = -1 \end{cases} \iff \begin{cases} c_1 = \pm 2 \\ c_2 = 1 \\ -1 = -1 \end{cases}$$

Since only 1 solution is enough, let the particular solution of (1) be:

$$y_1 = 2 + x \tag{4}$$

2. The solution to the Riccati equation has a form:

$$y = y_1 + u(x) = 2 + x + u, \text{where } u \text{ is a function of } x \tag{5}$$

So:

$$y' = 1 + u' \tag{6}$$

$$\begin{aligned} y^2 &= (2 + x + u)^2 \\ &= 4 + 2x + 2u + 2x + x^2 + ux + 2u + ux + u^2 \\ &= u^2 + 4u + x^2 + 4x + 2ux + 4 \end{aligned} \tag{7}$$

Substitute (5), (6) and (7) to the (1):

$$\cancel{1} + u' = \cancel{5} - \cancel{x^2} - u^2 - 4u - \cancel{x^2} - \cancel{4x} - \cancel{2ux} - \cancel{4} + \cancel{4x} + \cancel{2x^2} + \cancel{2xu}$$

The above equation is a **Bernoulli** equation, which needs to be solved with respect to u:

$$u' + 4u = -u^2 \tag{8}$$

(a) Solve the complementary equation to the (8):

$$\begin{aligned} u_c' &= -4u_c \\ \int \frac{du_c}{u_c} &= -4 \int dx, u_c \neq 0 \\ ln|u_c| &= -4x + c_3 \\ u_c &= c_4 e^{-4x} \end{aligned}$$

Since it is enough to have any solution to the complementary equation to solve the (8), for simplicity take $c_4 = 1$, so:

$$u_c = e^{-4x} \tag{9}$$

(b) Use (9) to find u by variating the parameter v:

$$u = u_c v = \frac{v}{e^{4x}}, \text{ where } v \text{ is a function of } x \tag{10}$$

$$u' = \frac{e^{4x}v' - 4e^{4x}v}{e^{8x}} \tag{11}$$

$$u^2 = \frac{v^2}{e^{8x}} \tag{12}$$

Substitute (10), (11) and (12) into the (8) to find v:

$$\frac{e^{4x}v' - 4e^{4x}v}{e^{8x}} + 4\frac{v}{e^{4x}} = -\frac{v^2}{e^{8x}}$$

Multiply both sides by e^{8x}:

$$\begin{aligned} e^{4x}v' - \cancel{4e^{4x}v} + \cancel{4e^{4x}v} &= v^2 \\ v' &= -\frac{v^2}{e^{4x}} \end{aligned}$$

Solve the above **separable** differential equation with respect to v:

$$\begin{aligned} \int \frac{dv}{v^2} &= -\int e^{-4x}dx \\ -\frac{1}{v} &= \frac{1}{4}e^{-4x} + c \\ v &= \frac{4}{c - e^{-4x}} \end{aligned} \tag{13}$$

The (13) can now be substituted to (10), which is the most general solution for (8):

$$u = \frac{4}{ce^{4x} - 1} \tag{14}$$

Now it is possible to get back to (5) and write **the most general solution for the differential equation from the IVP** (1), by substituting u with (14):

$$y = 2 + x + \frac{4}{ce^{4x} - 1} \tag{15}$$

The (15) is *undefined*, when:

$$\begin{aligned} ce^{4x} &= 1 \\ e^{4x} &= \frac{1}{c} \\ x &= \frac{ln(\frac{1}{c})}{4} \end{aligned} \tag{16}$$

3. To finish solving the initial value problem (1) express the general formula for c in terms of the initial condition y_0 and x_0 and the substitute the given corresponding values:

$$\begin{aligned} y_0 - 2 - x_0 &= \frac{4}{ce^{4x_0} - 1} \\ ce^{4x_0} - 1 &= \frac{4}{y_0 - 2 - x_0} \end{aligned}$$

$$c = \frac{2 + y_0 - x_0}{e^{4x_0}(y_0 - x_0 - 2)} \tag{17}$$

$$c = \frac{2 + 1 - 0}{e^{4*0}(1 - 0 - 2)} = \frac{3}{1 * (-1)} = -3 \tag{18}$$

From (17) it follows that the IVP (1) is not solvable, when the initial condition satisfies the equation:

$$y_0 = x_0 + 2 \tag{19}$$

So, **the solution to the IVP** (1) is:

$$y = 2 + x - \frac{4}{3e^{4x} + 1} \tag{20}$$

Because of the (16) and (18) the solution (20) is defined $\forall x \in \mathbb{R}$.

2 Application demonstration

2.1 General description

The application's source code is available on Github: `https://github.com/TymurLysenkoIU/DE`

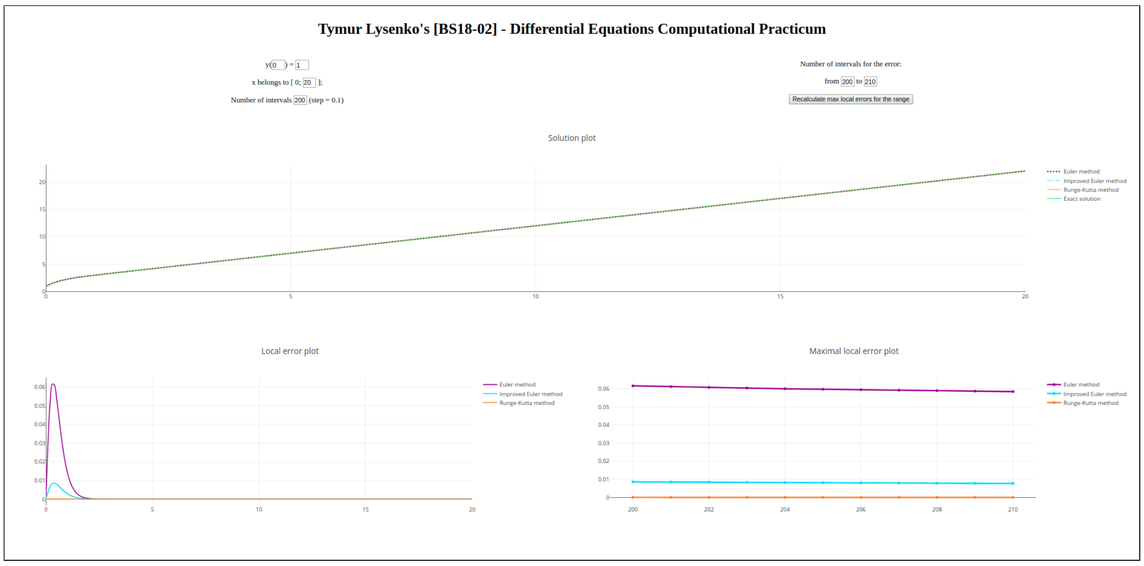
The application is implemented in **Scala.js** with the use of **Binging.scala** for UI and **Plotly.js** for plotting graphs. It implements the:

- **Euler**
- **improved Euler**
- **Runge-Kutta**

numerical methods and applies the to the IVP from the section 1. As the result, it plots the calculated values of the *solution*, *local truncation error* and *maximal local truncation error* for the specified interval of number of intervals.

UML diagram of the application generated by IntelliJ IDEA can be found in the **./img/class-uml-diagram.png**.

2.2 UI


Tymur Lysenko's [BS18-02] - Differential Equations Computational Practicum

y(0) = 1

x belongs to [0; 20];

Number of intervals 200 (step = 0.1)

Number of intervals for the error:

from 200 to 210

Figure 1: Application UI

The above figure demonstrates the UI of the implemented application. The application enables to input the *initial conditions for the IVP*, *interval that x belongs to* and a *number of intervals* (which affect the step). The application reacts to changes of these values and redraws the *solution* and *local error* plots.

The UI also allows specifying the *interval for the number of intervals*. The application does not react to changes the interval, instead, the corresponding button needs to be clicked to redraw the *maximal local truncation error* plot.

The plots are interactive, i. e. one can zoom them, move along some axis and are convenient to compare the numerical methods. For example, it is clear from the picture that **Euler method** has the greatest local error among other numerical methods. It is also possible to click the corresponding entry in the legend to display/hide the corresponding plot.

2.3 About the code

Since **Scala** combines both *object-oriented* and *functional programming*, the application's code is mostly **pure** (besides some UI-related code, because it must deal with side-effects), yet uses *OOP*. For example, consider the implementation of the **Euler method**:

```scala
trait DENumericalMethod {
  def solve(xs: NumericRange.Inclusive[BigDecimal], initialY: BigDecimal): Option[DENumericalSolution]
  def apply(xs: NumericRange.Inclusive[BigDecimal], initialY: BigDecimal): Option[DENumericalSolution] = solve(xs, initialY)
}

case class DENumericalSolution(xs: NumericRange.Inclusive[BigDecimal], ys: List[BigDecimal], le: List[BigDecimal])


case class EulerMethod(
  f: (BigDecimal, BigDecimal) => Option[BigDecimal],
  e: (BigDecimal, BigDecimal) => Option[BigDecimal],
  c: (BigDecimal, BigDecimal) => Option[BigDecimal]
) extends DENumericalMethod {
  override def solve(xs: NumericRange.Inclusive[BigDecimal], initialY: BigDecimal): Option[DENumericalSolution] = {
    def eulerMethodForValidData(r: NumericRange.Inclusive[BigDecimal], iy: BigDecimal) = {
      r.take(r.size - 1).foldLeft[Option[(List[BigDecimal], List[BigDecimal])]](Some((List(initialY), List(0.0)))) {
        (prev, x) =>
          val nextX = x + r.step
          for {
            p <- prev
            const <- c(xs.head, iy)
            exc <- e(nextX, const)
            fp <- f(x, p._1.head)
          } yield {
            val curY = p._1.head + (fp * r.step)
            val curLocalError = (curY - exc).abs
            (curY :: p._1, curLocalError :: p._2)
          }
      } map { res => DENumericalSolution(r, res._1.reverse, res._2.reverse) }
    }
    Some(eulerMethodForValidData(_, _)) filter { _ => xs.step < 1.0 } flatMap { _(xs, initialY) }
  }
}
```

2.3.1 Explanations to the implementation of the Euler method

On line 1 a `trait` (same as interface in other *OO* languages) is defined. It defines methods `solve` and `apply`, providing the default implementation for `apply` to be equivalent to the call of the `solve`. They accept `xs` of a type `NumericRange.Inclusive [BigDecimal]`, which is an inclusive range of floating point numbers with a particular step and `initialY` of a type `BigDecimal`, which is a value of the solution of a differential equation at the first element of the `xs` and possibly (if no error ocurred during the computations) returns an instance of `DENumericalSolution` (indicated by `Option[DENumericalSolution]`).

`DENumericalSolution` defined on line 6 stores the range of xs, corresponding y-s and local errors at each x.

Then, on line 9 the class implementing the **Euler method** is defined. It's constructor accepts a function `f` - which is the same as f in the expression $y' = f(x, y)$, it takes 2 numbers x and y and returns another number (if no error ocurred evaluating the function for the passed arguments). `e` stands for exact solution function, which depends on x and a constant. `c` - function that returns constants for the given initial condition y_0 and x_0.

`EulerMethod` extends `DENumericalMethod`, so it must implement the function `solve`, which solves an initial value problem for the provided initial conditions.

The method `solve` first defines an inner function `eulerMethodForValidData`, which does all the calculations. First, it discards the last elements of the passed range, because if it stays, then the next value of y will be calculated, which needs not to be present in the result. Then it left folds (goes from the beginning of a sequence to the and applies the passed function to each element of the sequence accumulating some value; the result of the operation - the accumulated value, when it reaches the end of the sequence) the range xs. The value accumulated by the fold is a pair of a list of the calculated ys for the respective xs and a list of local errors. The initial accumulated value is a pair of 2 singleton lists, first containing the y_0 and the second - 0. The function passed to `foldLeft` calculates the next x and uses `for` comprehension (monad comprehension in Scala) to get rid of the `Option` monad and calculate y_i and the corresponding local error for the point and adds it at the beginning of the list, i. e. if all calculations succeeded - the result will be `Some(List(...), List(...))`, otherwise `None`. The result of the function is `None`, if some of the calculations failed or `Some(DENumericalSolution(...))`, which contains the solution to an IVP.

Euler method diverges, if the given step is greater than 1, so the function solve must validate the step of the passed range `xs`. To validate this, on line 31 `Some` object is created, storing a function that accepts 2 arguments: a range of xs and y_0, checks the step to be less than 1 and if this is the case - the result is the result of `eulerMethodForValidData(xs, initialY)`, otherwise - `None`.