Differential Equations Computational Task

Maksim Surkov

April 2018, Innopolis University

Contents

1 Introduction

1.1 Motivation

Universe of differential equations is diverse and full of analytically unsolvable equations. Actually mathematicians have formulated general solutions for only a subset of standard equations. But there is still need in real world to apply unsolved ones and the best (and actually close enough) we can get is through approximation, i.e. Numerical methods. In the presented work we will go through solution of one ordinary differential equation and analyse efficiency of three numerical methods: Euler method, Improved Euler method and Runge Kutta method. Firstly we'll describe methods, then solve the problem analytically, finally describe the implementation and assess its efficiency.

1.2 General (numerical methods)

All methods rely on idea of finding each next point based on previous one. As a starting point we have to know initial value $f(x_0) = y_0$. Each next point depends on previous with a relation $y_n = g(x_{n-1}, y_{n-1}, h)$ where h is a step between x_{n-1} and x_n. Function g determines the calculation approach and is the only part that is different in three methods presented below.

1.3 Euler method

To determine point (x_n, y_n) we build a line that goes through point (x_{n-1}, y_{n-1}) and tangent of its angle to ox is equal to $f'(x_{n-1}, y_{n-1})$. [1]

1.4 Improved Euler method

We can improve Euler method by adding additional line. Thus to determine point (x_n, y_n) we build a line that goes through point (x_{n-1}, y_{n-1}) and tangent of angle to ox is equal to $f'(x_{n-1}, y_{n-1})$. Then at resulting point (x_n, y_n) we construct complementary line with tangent of angle to ox $f'(x_n, y_n)$. Now we find avarage of these two lines and shift resulting line to go through (x_{n-1}, y_{n-1}). [2]

1.5 Runge-Kutta method

This method improves effeciency of calculation even more than Improved Euler method by adding more complementary graphs. Firstly construct original line as in Euler method (call it k_1), take half the oy shift (i.e. $(y_n - y_{n-1})/2$ at x_n and add it to point at $x_{n-1} + h/2$ (actually it has y coordinate equal to y_{n-1}), line at this point will be k_2. Extend k2 by h on ox so we get shift on oy, add half that shift to new point at $x_{n-1} + h/2$, line at this point will be k_3. Finally extend k_3 by h on ox and add half the oy shift to new point at x_n, line at this point will be k_4. Finally balance these lines with coefficients 1,2,2,1 to get resulting line. [3]

2 Equation analysis

Now we can start solving actual equation. Problem is formulated in the following way: (Equation 19) (Note: we do not consider complex roots)

$$\frac{dy}{dx} = 2 * \sqrt{y} + 2 * y \quad for \quad x_0 = 0,\ y_0 = 1,\ X = 9$$

Clearly that is first-order separable ODE and we can rewrite and solve it as:

$$\frac{dy}{2 * \sqrt{y} + 2 * y} = dx \ \ , \ 2 * \sqrt{y} + 2 * y \neq 0$$

$$\int \frac{dy}{2 * \sqrt{y} + 2 * y} = \int dx$$

$$\frac{1}{2} \int \frac{dy}{\sqrt{y} + y} = \int dx$$

$$\frac{1}{2} \int \frac{dy}{\sqrt{y}(1 + \sqrt{y})} = \int dx$$

$$t = 1 + \sqrt{y}\ ,\ \frac{dt}{dy} = \frac{1}{2 * \sqrt{y}}\ ,\ 2 * \sqrt{y} dt = dy$$

$$\int \frac{dt}{t} = \int dx$$

$$ln(t) = x + C$$

$$Back\ substitution : ln(1 + \sqrt{y}) = x + C$$

$$1 + \sqrt{y} = e^{x+C}$$

$$\boxed{\mathbf{y = (e^{x+C} - 1)^2\ ,\ y \geqslant 0}}$$

$$Let's\ check \ \ 2 * \sqrt{y} + 2 * y = 0$$

$$\sqrt{y} = -y \ \Rightarrow \ y = 0\ which\ is\ included\ into\ general\ solution.$$

Function exists on $y \in [0, +\infty)$ and $x \in (-\infty, +\infty)$, it is monotonically increasing and continuous on that interval. Also for computations we need to explicitly define C:

$$y = (e^{x+C} - 1)^2$$

$$\begin{cases} \sqrt{y} = e^{x+C} - 1\ , \\ -\sqrt{y} = e^{x+C} - 1 \end{cases}$$

$$\begin{cases} 1 + \sqrt{y} = e^{x+C}\ , \\ 1 - \sqrt{y} = e^{x+C} \end{cases}$$

$$\begin{cases} ln(1 + \sqrt{y}) = x + C\ , \\ ln(1 - \sqrt{y}) = x + C \end{cases}$$

$$\begin{cases} C = ln(1 + \sqrt{y}) - x\ , \\ C = ln(1 - \sqrt{y}) - x \end{cases}$$

$$Both\ equations\ have\ limitation\ y \geqslant 0,\ latter\ equation\ also\ has\ limitation\ y \neq 1.$$

$$Let's\ use\ former\ equation\ as\ solution\ in\ application.$$

$$\boxed{\mathbf{C = ln(1 + \sqrt{y}) - x\ ,\ y \geqslant 0}}$$

We will use concluding expressions from this area for building analytical plot in application.

3 Application description

Application can be launched using *DifferentialEquationDemo.exe*. On start, basic view is opened and graph is built with default values.

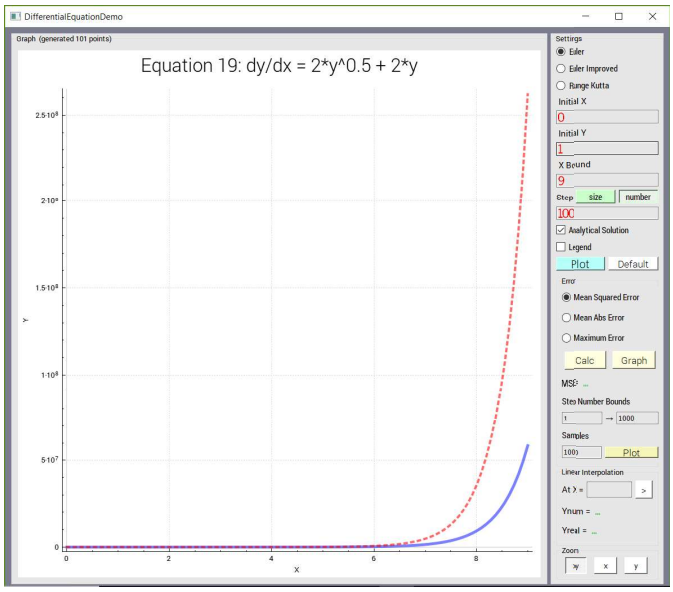
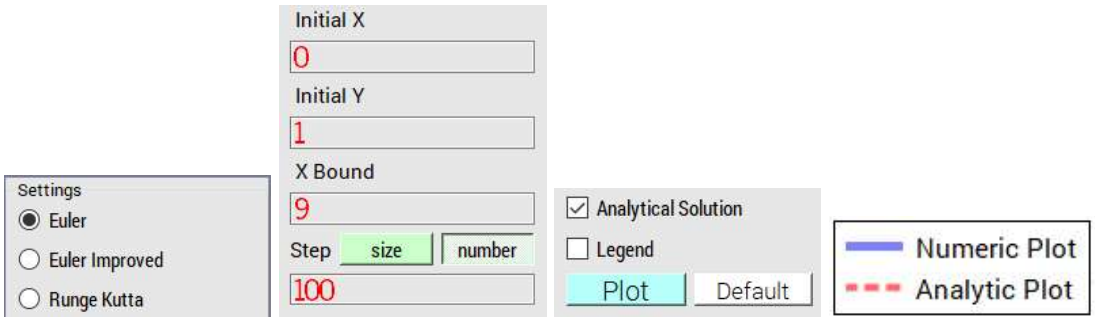


Figure 1: Starter UI view.

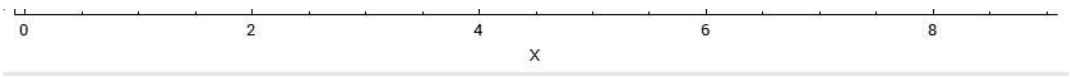
- On top is numeric method selection panel as shown on Fig 2(a). By setting one, whole application will plot graphs with respect to selected method.
- Next, panel with initial value and related parameters as shown on Fig 2(b). It's possible to select step size explcitly by checking **size** or implicitly selecting number of steps by checking **number** (latter is recommeded due to bound breaking that is intentionally tolerated in former).
- When we have filled settings, graph can be plotted by pressing **Plot** button as shown on Fig 2(c). Settings can be reset to default by pressing **Default** button. **Analytical Solution** checkbox allows to plot exact solution on same window with numerical, overwise numerical plot will be plotted only. Also there is an option to show plot legend similar to Fig 2(d), you can toggle it with **Legend** checkbox.



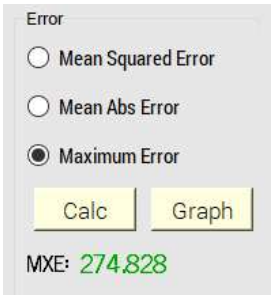
(a) Method selection. (b) Initial value settings. (c) Plotting menu. (d) Legend example.

Figure 2: Settings panel overview.

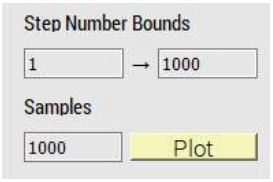
- There are three classes of error plotting.
  First - simple error calculation from currently shown plot. That requires both numerical and analytical plot to be shown in window. Activated with **Calc** button, result of calculation, i.e. single number, is shown on line just under the buttons as shown on Fig 3(b).
  Second is also related to current plot, it takes absolute errors on each of points and results separate plot. Activated with **Graph** button, result replaces plot in window. To use it again it's required to do numeric and analytic plots first.
  Third method uses initial value settings and settings from panel below, here you can select step number to be iterated as shown on Fig 3(c). E.g. for Step Number Bounds $1 \rightarrow 100$, calculation will begin from step size $\frac{X-x_0}{1}$ and end with $\frac{X-x_0}{100}$, while doing so it will generate number of points equal to **Samples**. So for 100 samples it will follow $\frac{X-x_0}{1}, \frac{X-x_0}{2}, \frac{X-x_0}{3}, \ldots, \frac{X-x_0}{99}, \frac{X-x_0}{100}$.
- Additionaly we can get exact value at some X for both numerical and analytical plots as shown on Fig 3(d), note that it does not apply to error graphs.
- Finally we have to mention navigation in window, you can pan the view with **left mouse button** and zoom with **mouse wheel**. Zooming mode can be chosen in menu shown on Fig 3(e). Thus by choosing either x or y you can limit zoom by one axis.



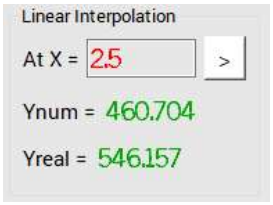
(a) Example of X axis.



(b) Error menu.



(c) Error-to-step menu.



(d) Get Y on selected X.

(e) Zoom settings.

Figure 3: Settings panel overview (cont).

- According to currently shown plot, names of axes will be shown. Example of X axis for default plot can be seen on Fig 3(a). Note that while plotting error-to-step plot and typing numbers of steps as metric, X axis unit will be converted to actual step size. E.g. for values $1 \rightarrow 100$, we can observe step sizes on X axis from $\frac{1}{100}$ to $\frac{1}{1}$ (lower values always on the left).

4 Code comments

Application is powered by C++ on top of Qt Framework [5], plotting is done through QCustomPlot addon[4]. Licensing information is attached to project files, it can be found in *Licenses* folder.
For better understanding let's divide application into several parts and describe each one.

4.1 Main Window

Related files: DifferentialEquationDemo.cpp/.h/.ui

This class is a central point of user interface. It contains bindings from UI elements to function slots. When a button press is registered, focused fields are checked to be non-empty, then data is fetched and checked to be valid (axis bounds and allowed range for mathematical operations checks). Finally it's thrown into new thread, which after finishing will perform a callback with result. Also all fields have RegExp for decimal numbers, thus meaningless input is not possible.

4.2 PlotDataObject Container

Related files: PlotDataObject.cpp/.h

This is a simple class that stores Plot as X and Y arrays. ($QVector < double >$ to be certain). It can be constructed and cleared to deallocate memory.

4.3 Numeric Methods

Related files: Solver.cpp/.h, Euler.cpp/.h, EulerImproved.cpp/.h, RungeKutta.cpp/.h

Core meaning of whole application is encapsulated in four classes that are going to be described here. There is a **Solver** class that declares virtual function *get_next_y*(...) and function *solve*() that takes initial values and iterates through defined x range. Each y_n is calculated with $f(x_{n-1}, y_{n-1})$ where *get_next_y*(...) acts as $f(...)$. Classes **Euler**, **Euler_Improved** and **RungeKutta** inherit Solver class and each of them provides definition for *get_next_y*(...) function. Simplified

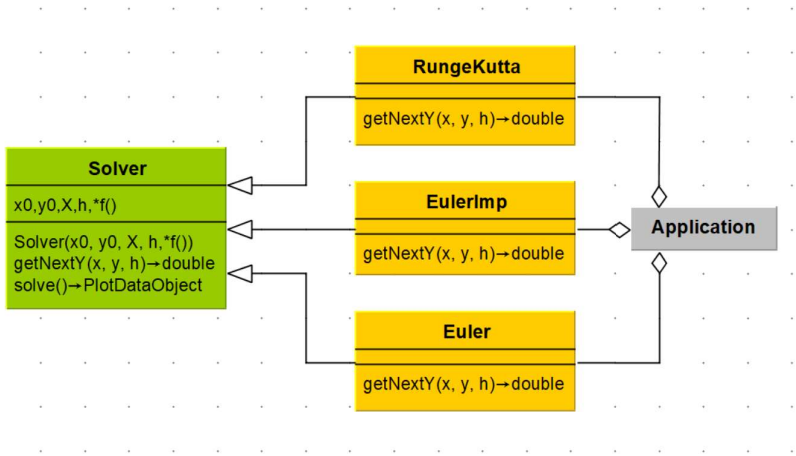


Figure 4: Approximate UML layout for Numeric Methods.

view of that structure is presented on Fig 4. So the final operation is the following: construct Solver with instance of either Euler, Euler_Improved or RungeKutta using initial value and additional parameters. Call *solve*() function which returns PlotDataObject. That output is inserted into plot window.
Looking closer, implementation is a bit more complex becuase Solver class inherits from QObject and provides definition for signals and slots which allows it to run as a thread.

4.4 Exact Plot

Related files: SimplePlot.cpp/.h

Analytical solution plotting does not require special algorithm, it is simple as $y_n = f(x_n, C(x_0, y_0))$. Calculation is run as a separate thread so class **SimplePlot** inherits from QObject and implements signals and slots for communication with main thread.

4.5 Error Calculation

Related files: ErrorCalc.cpp/.h, Solver.cpp/.h

There are two types of error calculation: single-step-size and step-size-range. Both methods support selection of error type. First uses ErrorCalc class (which is also implemented as a thread). After plotting axes data is stored in a buffer, getting error of currently shown plot can be done either as a single number or plot of actual error on each point ($Y_{ERRORn} = |Y_{EXACTn} - Y_{NUMn}|$). Second method uses range of step sizes with indicated samples number. For each step size it builds numerical plot, then for each point of resulting plot it gets bias from exact solution. At that point distributed error plot is converted into singe overall error via chosen method (MSE, MAE, MXE). For the range of step sizes overall errors are collected into new plot.

4.6 Interoperation

Related files: DifferentialEquationDemo.cpp/.h, Solver.cpp/.h, ErrorCalc.cpp/.h

Application is structured to run all calculations in separate thread, leaving main's thread functionality for UI management and handling user input. When user choses to plot a graph new object for requested calculations is constructed along with QThread object, former is attached to latter so that it can run as a separate thread. When thread finishes its calculations it performs an asynchronous callback (through a signal) to main's thread slot. Since UI is blocked for moment of running calculations, it's not possible to normally launch two separate concurrent threads.

4.7 Possible optimization and further improvement

Optimization methods are related to topic on making calculations faster on given hardware. The optimization could be done on vector operations. For example when we have numerical Y's and analytical Y's and we want to get Error plot as a difference of these two arrays. First idea is to use CUDA, however with fairly small plot sizes loading vectors into graphical memory would take lots of time, not

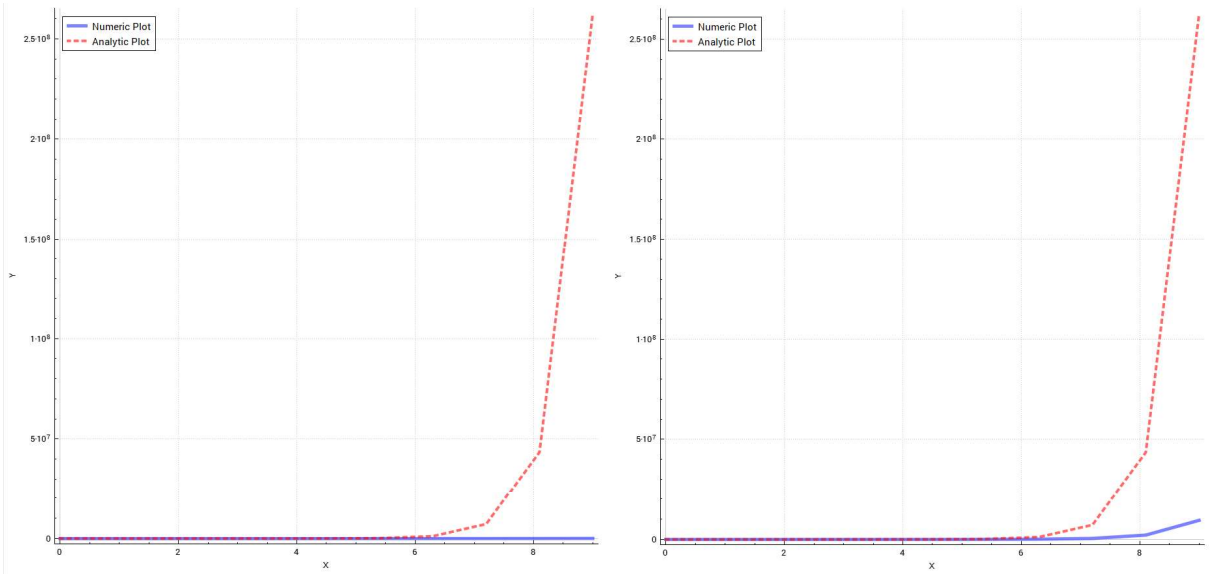
giving enough advantage. Another idea is to use either SSE2/AVX instructions natively supported by C++ (or by writing assembly inset, thus we can make 4 doubles operated at a time with AVX (2 for SSE2). That would speed up process by 10-20%.

5 Sample Plots

All plots are done with setting $x_0 = 0$, $y_0 = 1$, $X = 9$

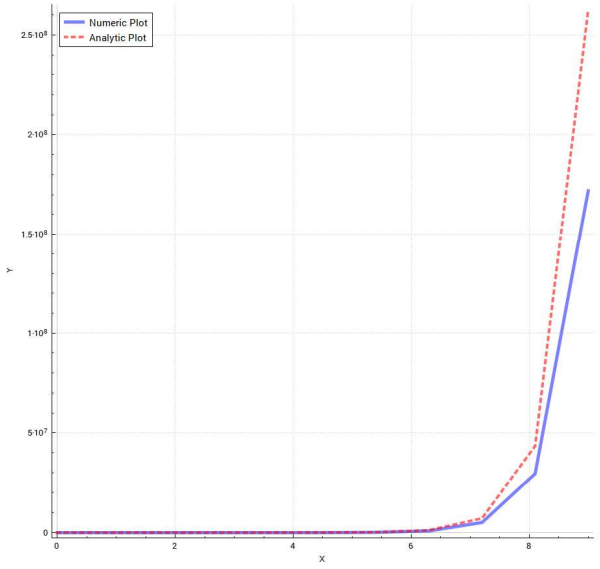
5.1 Numerical and Exact Plots

As a starting point let's take small number of steps, on which all methods do not converge. But we can already see difference between their behavior.



(a) Euler

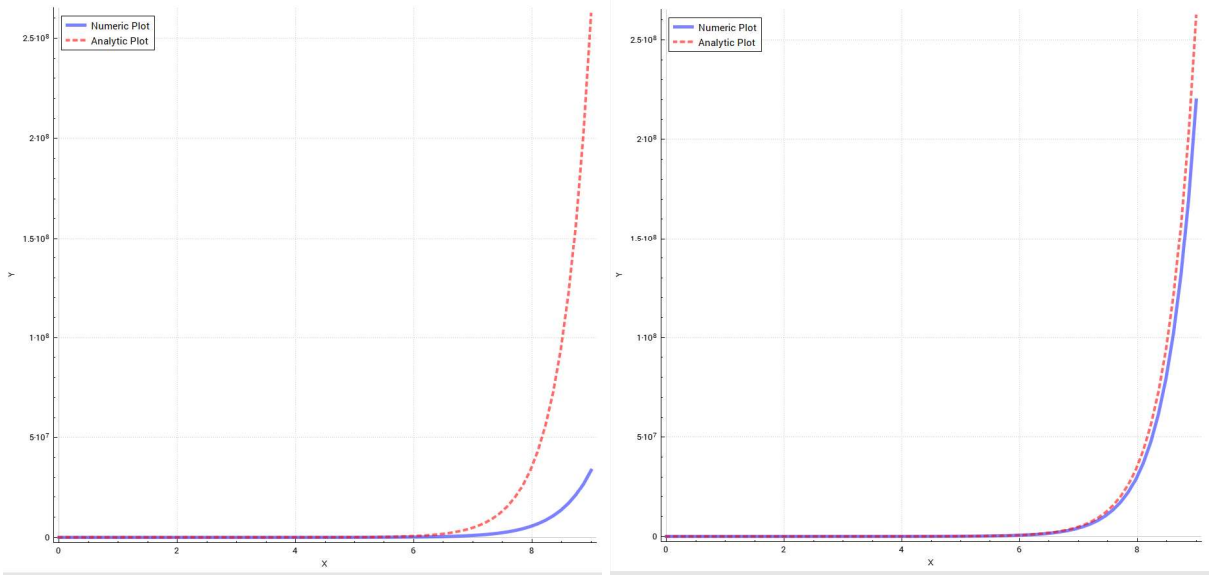
(b) Improved Euler



(c) Runge Kutta

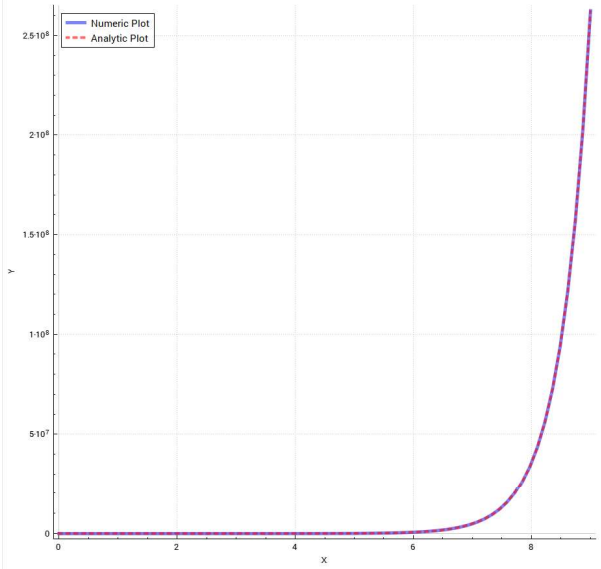
Figure 5: Numerical and Exact plots, 10 Steps (h=0.9)

At a small number of steps already Runge Kutta method shows good convergence.



(a) Euler

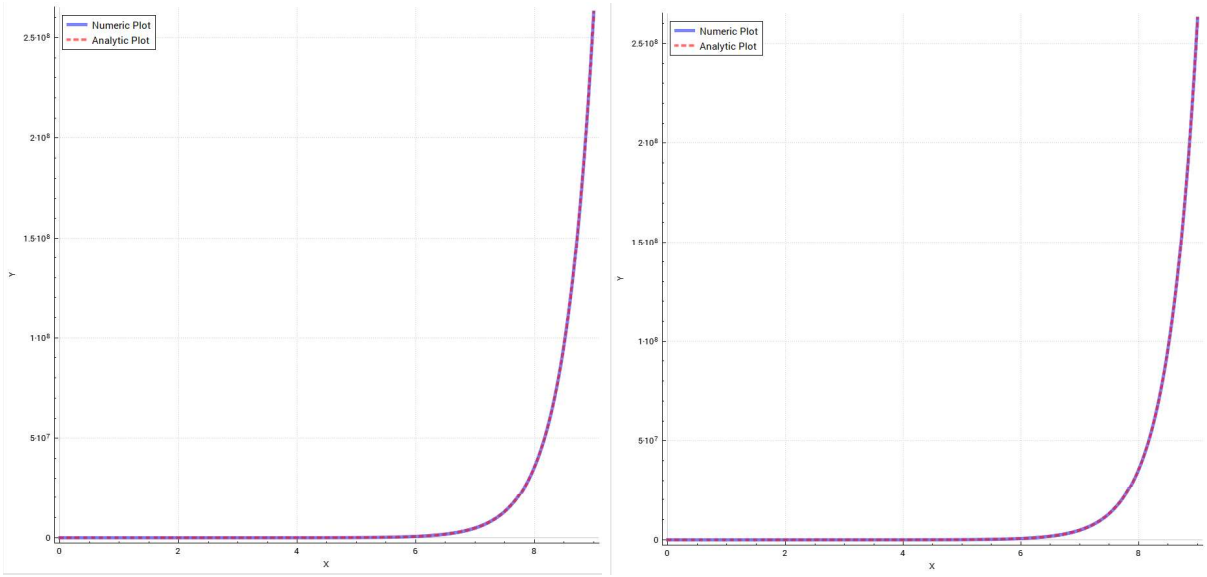
(b) Improved Euler



(c) Runge Kutta

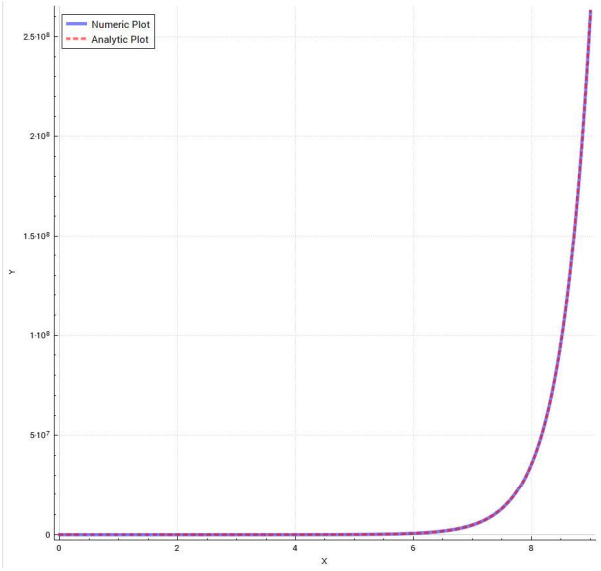
Figure 6: Numerical and Exact plots, 70 Steps (h=0.13)

And finally the point where Euler converges good.



(a) Euler

(b) Improved Euler

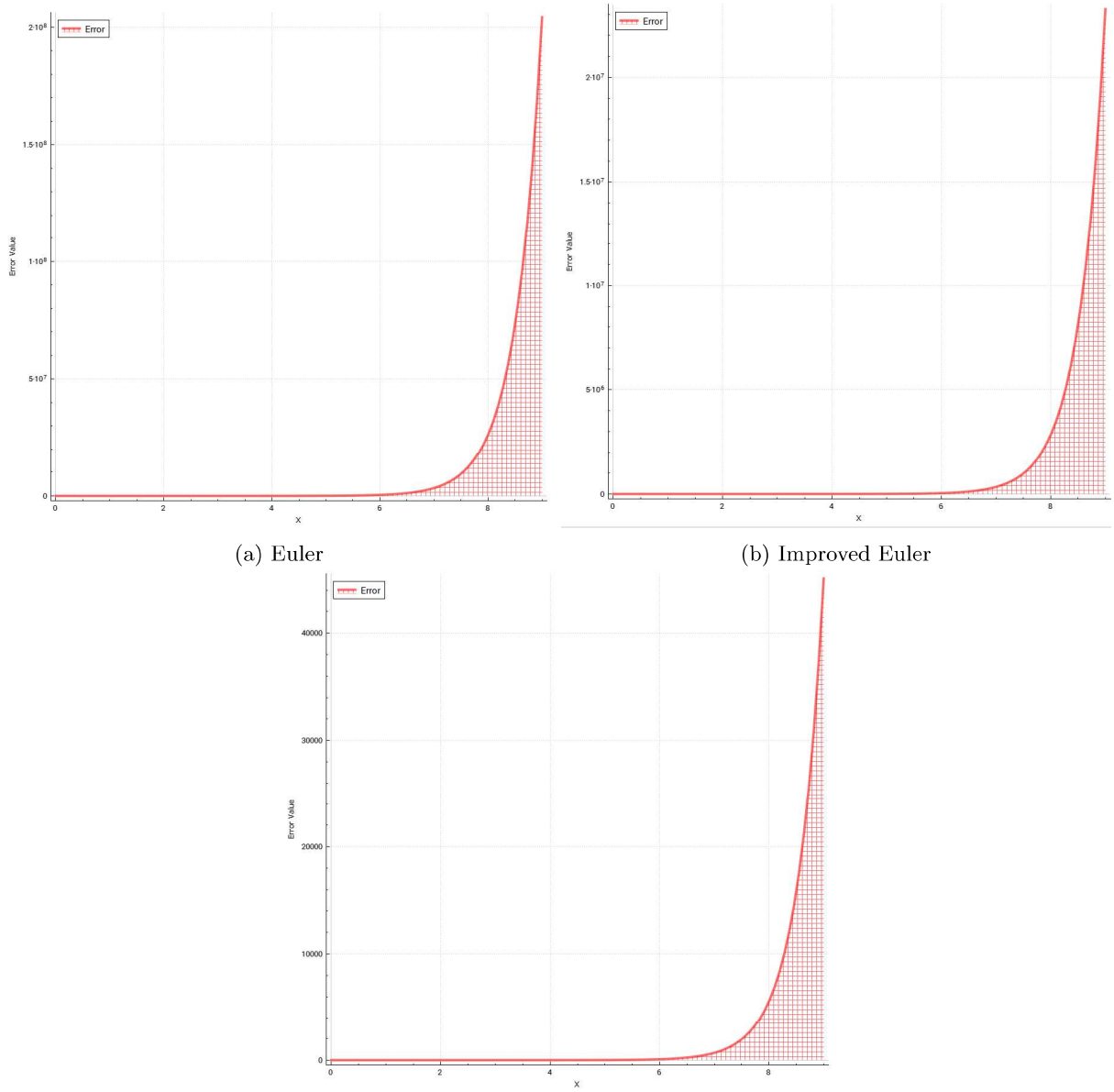


(c) Runge Kutta

Figure 7: Numerical and Exact plots, 1000000 Steps (h=0.000009)

5.2 Error Plot

Here we can see that behaviour of error along the plot is similar for all three methods, however maximum error (in this case rightmost point) varies dramatically.



(a) Euler

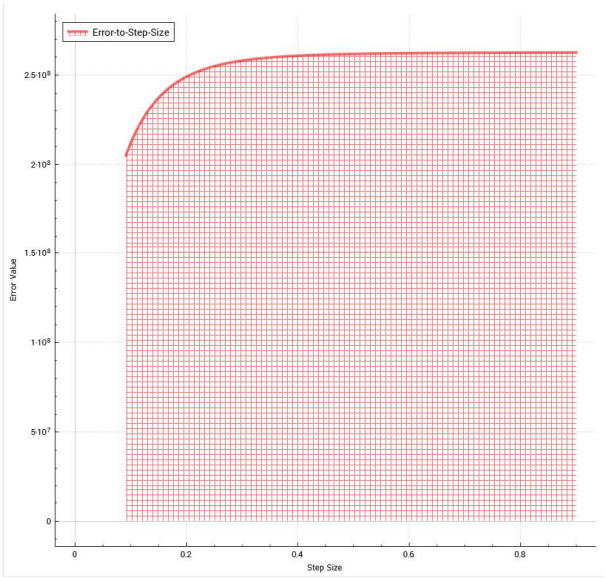
(b) Improved Euler

(c) Runge Kutta

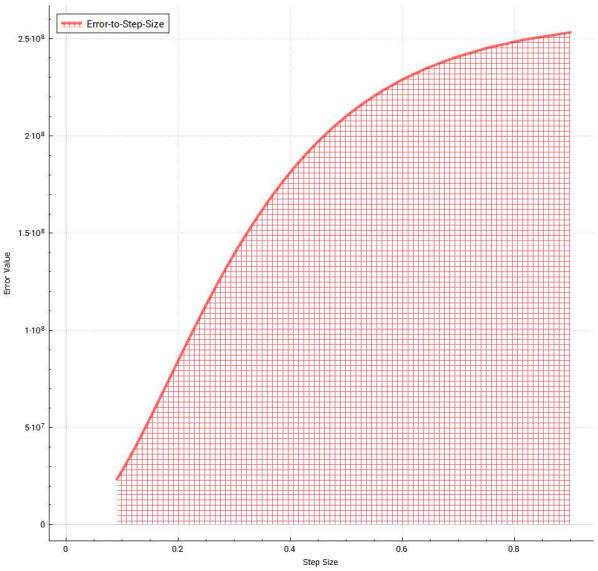
Figure 8: Actual Error, 100 Steps (h=0.09)

5.3 Error-To-Step-Size Plot

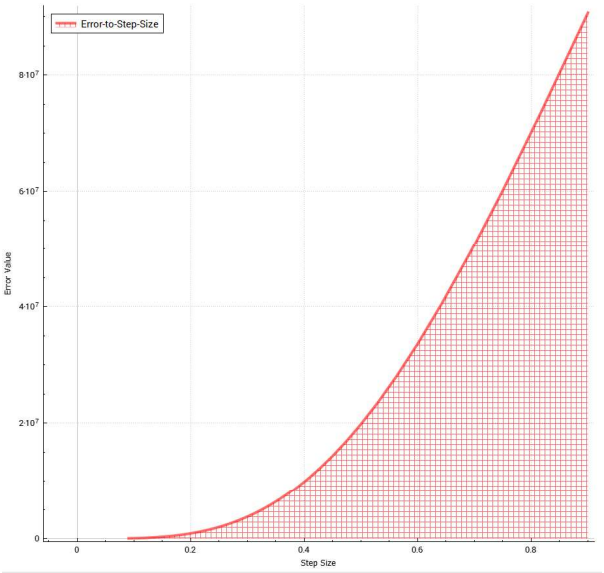
Now let's look how the step size variation affects each of methods. So, Euler method shows large error and gains convergence only on the very small step size less than 0.1, overall maximal error is around $2.5 * 10^8$. Improved Euler method is better, it gains convergence rate at step of 0.5 already, overall maximal error is around $1.7 * 10^8$. Runge Kutta method shows very good convergence on relatively large step of 0.8, overall maximal error is around $2.5 * 10^7$.



(a) Euler



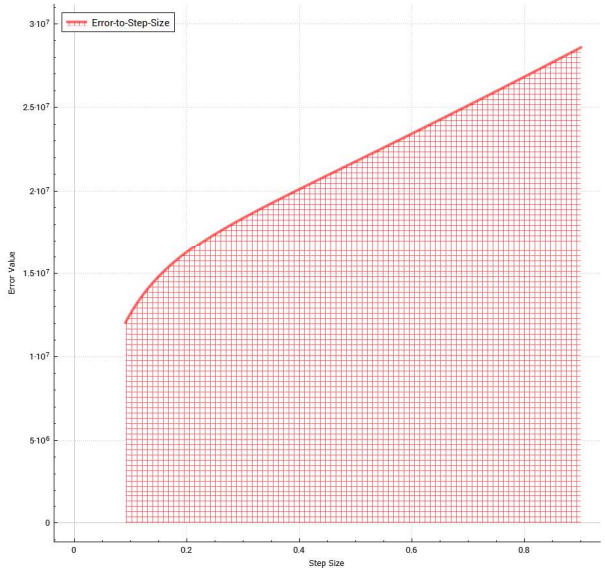
(b) Improved Euler



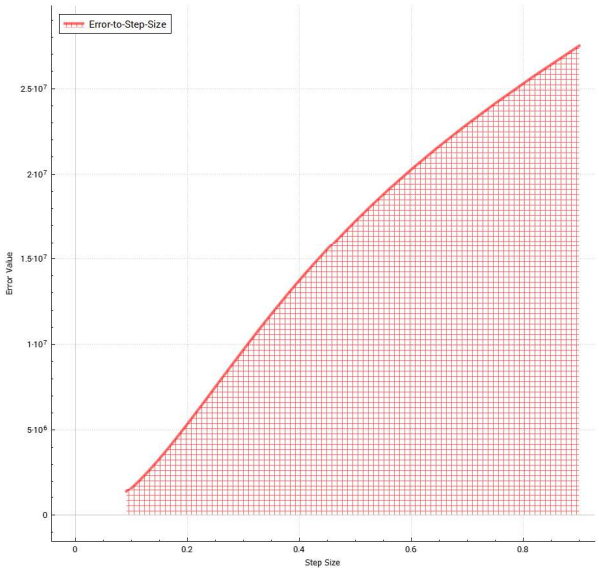
(c) Runge Kutta

Figure 9: Maximal Error, 10-100 Steps (h - 0.9 to 0.09)

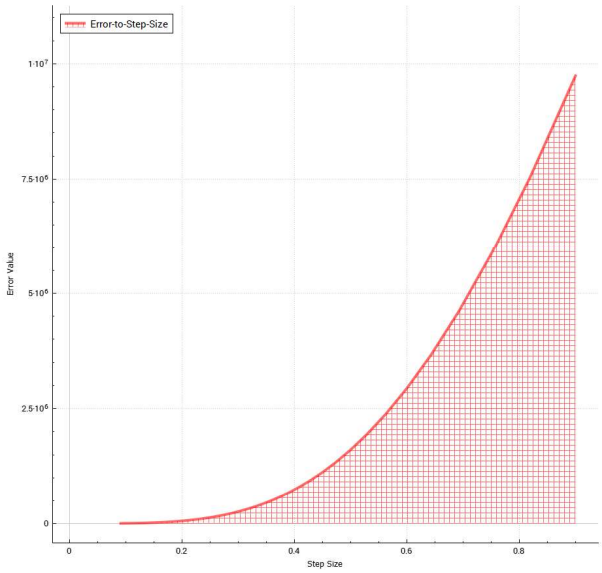
Also same plots for Mean Error.



(a) Euler



(b) Improved Euler



(c) Runge Kutta

Figure 10: Mean Error, 10-100 Steps (h - 0.9 to 0.09)

6 Conclusion

We have assessed usage of numerical methods to solve differential equations. Comparison between numerical solutions and exact solutions showed that their precision depend on the step size. Different numerical methods demonstrate completely different convergence scenario. However better methods do not require much more computational power, so there is actually little tradeoff. It's simple as better methods provide better convergence. But still numerical methods do not completely converge to exact solution, it's just very good approximation. However that approximation is absolutely useful in case we do not have exact solution.

7 References

[1] Euler method
[2] Improved Euler method
[3] Runge-Kutta method
[4] QCustomPlot website
[5] Qt website

Innopolis University, Russia. Document has been compiled with MiKTeX.