Differential Equations and Numerical Approximation using C++

Matthias Gunz

11-25-2007

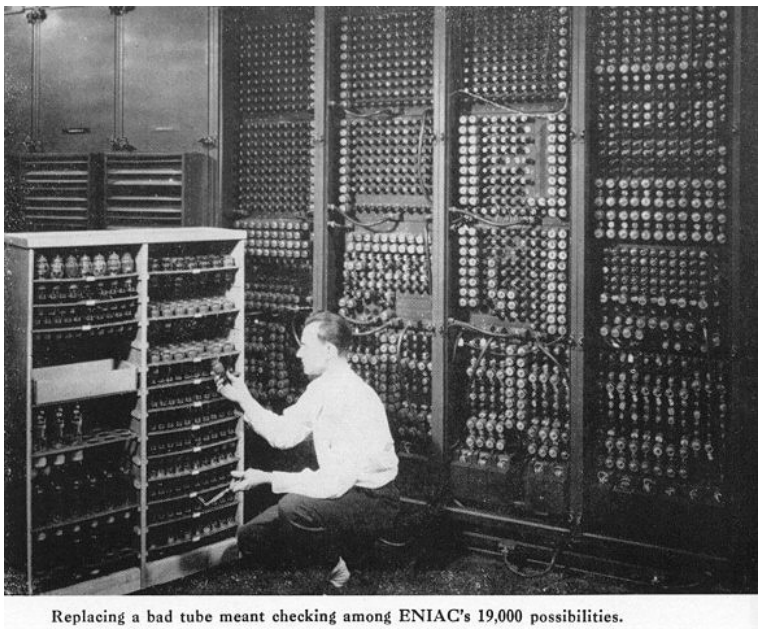
1 Idea

The fundamental idea of this project is, to show the different numerical methods of solving differential equations. DE's are very important in physics because they describe many problems like body systems, processes in electrical engineering, etc.
In this project i decided to use C++, because it is a very common programming language and i wanted to expand my poor knowledge about it. Another reason was, that i have had to write everything on my own and no math program like MatLab, Mathematica, etc. offered the algorythms to solve the equations. So, my first task was to get comfortable with the approximation methods and how i am able to translate them into C++ - Code.
This project paper should give an overview of these methods, their quality and their implementation. I tried to produce a graphical interface with Qt, so that it would have been possible to change the DE formula without editing the file, but it came out that this would have been a lot of additional work and i've already had enough problems with the C++ - Code.

2 Numerical Approximation Methods

2.1 Introduction

The very imortant and probably the oldest usage of computers in physics is to solve Differential Equations. The first full electronical mainframe called ENIAC - programmed with connectors - was used to solve partial DEs for nuclear fusion and made it possible to develop the hydrogen bomb. The americans didn't win the race to the moon because of their better rocket technology, but of their ability to build the first computer, light enough fitting into the spacecraft computing numerically the flight path.



ENIAC (Electronic Numerical Integrator and Computer)

2.2 Discretization

Computer are finite machines and therefore can only work with discrete problems. Firstly, in equations the independent variable x (or t in time dependent DEs) has to be discretized:

$$x = x_0 + jh;\ j \in \mathbb{N}$$

Where h is the step size in this iteration. The iteration index j adopts the role of the independent variable and one can use the following notation:

$$x_j = x_0 + jh,\ y_j = y(x_j),\ y_j^{'} = y'(x_j)$$

2.3 Euler Method

The root of every iteration method is the taylor expansion:

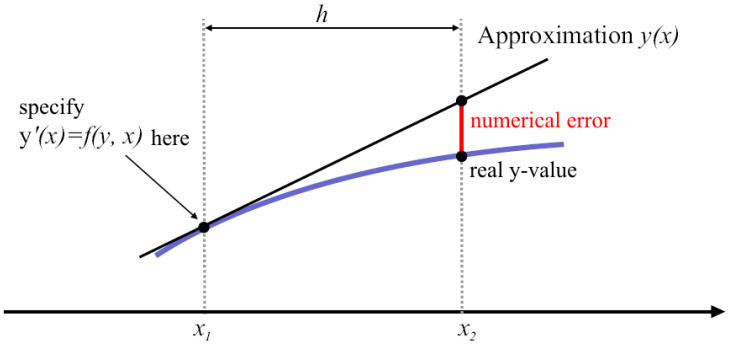
$$y_{j+1} = \sum_{k=0}^{\infty} \frac{h^k}{k!} y_j^{(k)} = y_j + h y_j^{'} + \frac{h^2}{2} y_j^{''} + ...$$

If every derivation would be known, one could iterate exactly the Differential Equation. Unfortunately, only the first derivate is given through:

$$y_j^{'} = f(x_j, y_h)$$

The simplest method is therefore to cancel after the 1st order, what leads to an iteration error of the order $O(h^2)$. This method is called Euler:

$$y_{j+1} = y_j + h f(x_j, y_j) + O(h^2)$$



Functionality of the Euler Method

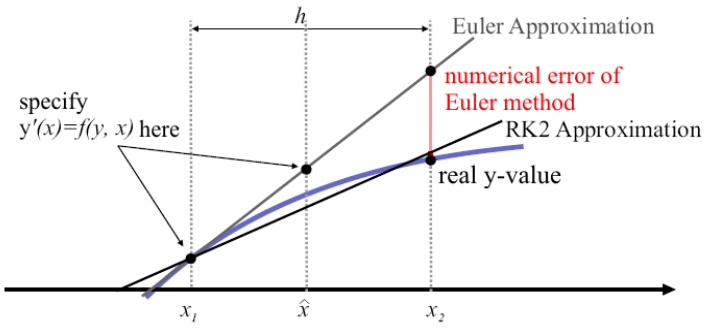
The Euler method is very popular in solving Integrations, but not very exact. To improve the accuracy, the step size h must be chosen very small.

2.4 Runge-Kutta 2nd Order

As a matter of fact, it is possible to improve the efficiency and accuracy of a numerical iteration with little effort, by using Runge-Kutta. The Runge-Kutta Method 2nd order doesn't use the gradient of the left/precedent point - like Euler - to proceed to the next point, but the gradient in the middle of the two points. Thereto you do an half iteration step $h/2$, compute the gradient using function f and use it for the step to the next point. Therefore the Runge-Kutta 2nd order is not much more complicated than Euler. On the one hand the iteration needs a bit more time, because the function f has to be computed twice, but the error reduces to the order $O(h^3)$. On the other hand it is possible to work at the same accuracy with a bigger step size and therefore in less time.
The functionality of RK2 is following:

$$k_1 = h \cdot f(x_j, y_j)$$

$$y_{j+1} = y_j + h \cdot f(x_j + \frac{h}{2}, y_j + \frac{k_1}{2}) + O(h^3)$$



Functionality of the RK2 method

2.5 Runge-Kutta 4th Order

Runge-Kutta 2nd order has an error of $O(h^3)$. On a related way you can define the Runge-Kutta methods of higher orders: A RK kth order would converge with decreasing step size with an error of $O(h^{k+1})$. But these methods of higher order are not clearly defined, there is not *the* Runge-Kutta kth order, rather a variety of approaches.
If you increase the order, you can expect a obvious improvement and a much more complicated algorithm. A good trade-off standing the test in practise is the Runge-Kutta method 4th order - alias known as the classical Runge-Kutta method:

$$k_1 = h \cdot f(x_j, y_j)$$

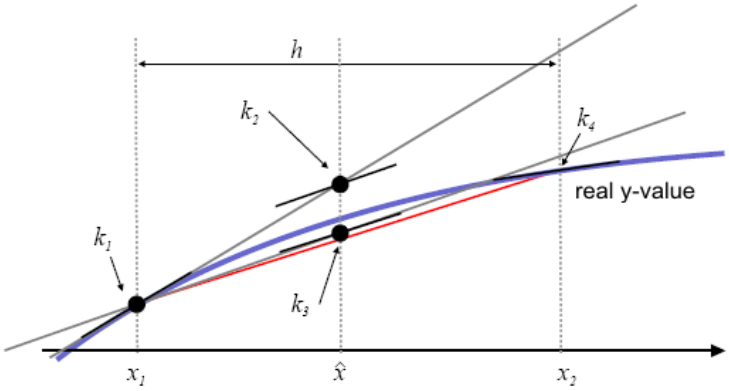
$$k_2 = h \cdot f(x_j + \frac{h}{2}, y_j + \frac{k_1}{2})$$

$$k_3 = h \cdot f(x_j + \frac{h}{2}, y_j + \frac{k_2}{2})$$

$$k_4 = h \cdot f(x_j + h, y_j + k_3)$$

$$y_{j+1} = y_j + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}$$

At first - like in RK2 - the gradient k_2 in the middle of the two points will be computed. With $k_2$another point at the half interval is approximated, where the gradient $k_3$is taken. At last, $k_4$is computed at the end of the inverval. With these four parameters a mean value is formed, used as the gradient for the whole step. It is easier to understand regarding the following figure.



Functionality of the RK4 method

2.6 Runge-Kutta-Fehlberg 4th Order

It often occurs, that in certain regions of a function a very accurate iteration is needed, while in other regions greater steps are adequate. So it would be an advantage if an algorithm would adjust the step size dynamically. Further on the adjustment should be done fast and not only in large distances and the half step method should be avoided, because of the great time effort. A method, that computes the error spontaniously with the k-values and fits the step size dynamically is the Runge-Kutta-Fehlberg method 4th order (aka. RKF45). This method uses following equations:

$$k_1 = h \cdot f(x_j, y_j)$$

$$k_2 = h \cdot f(x_j + \frac{h}{4}, y_j + \frac{k_1}{4})$$

$$k_3 = h \cdot f(x_j + \frac{3}{8}h, y_j + \frac{3}{32}k_1 + \frac{9}{32}k_3)$$

$$k_4 = h \cdot f(x_j + \frac{12}{13}h, y_j + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3)$$

$$k_5 = h \cdot f(x_j + h, y_j + \frac{439}{216}k_1 - 8 \cdot k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4)$$

$$k_6 = h \cdot f(x_j + \frac{1}{2}h, y_j - \frac{8}{27}k_1 + 2 \cdot k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5)$$

$$y_{j+1} = y_j + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$$

Beacause there are more interim values as needed for the computation of y_{j+1}, a better approximation of 5th order can be computed:

$$z_{j+1} = y_j + \frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6$$

If ϵ is the given tolerance, an adjustment of the step size h will be:

$$h = h \cdot \left(\frac{\epsilon \cdot h}{2\,|z_{j+1} - y_{j+1}|} \right)^{1/4}$$

3 Programming

The implementation of the numerical methods into the C++ code needed some research and getting involved in the way of programming this language. The main difficulties were: how am i able to return all the y(x) values from the functions to the main programm and how to implement an differential equation as a function. At first i decided to handle the DE-function as a pointer function and tried to stay away from the simple "return" of an expression:

```
16  //The ODE is defined here
17  void f*(double *x, double *y)
18  {
19      ydot=y+x-1;
20  }
```

Unfortunately i wasn't able to implement it in my numerical functions, because of the strange complexity of pointers and so i returned to the old style, by staying away from a pointer function:

```
16  //The ODE is defined here
17  double f(double x, double y)
18  {
19      return y+x-1;
20  }
```

The numerical approximation methods are functions as well and shown in the code below:
Euler in C++:

```
22  //Euler function to solve the ODE
23  void euler(double a, double b, double h, double y0, double eulerout[][2])
24  {
25      int N = (int)abs((b-a)/h);
26      double y = y0;
27      double x = a;
28
29      for (int i=0; i<=N; i++)
30      {
31          eulerout[i][1]=x;
32          eulerout[i][2]=y;
33          y = y + h*f(x,y);
34          x = x+h;
35      }
36  }
```

Runge-Kutta 2nd order in C++:

```
38  //Runge-Kutta 2nd order function to solve the ODE
39  void rk2(double a, double b, double h, double y0, double rk2out[])
40  {
41      int N = (int)abs((b-a)/h);
42      double y = y0;
43      double x = a;
44      double k1;
45
46
47      for (int i=0; i<=N; i++)
48      {
49          //rk2out[i][1]=x;  same x as in euler
50          rk2out[i]=y;
51          k1 = h*f(x,y);
52          y = y + h*f(x+(h/2.0),y+(k1/2.0));
53          x = x+h;
54      }
55  }
```

Runge-Kutta 4th order in C++:

```
57  //Runge-Kutta 4th order function to solve the ODE
58  void rk4(double a, double b, double h, double y0, double rk4out[])
59  {
60      int N = (int)abs((b-a)/h);
61      double y = y0;
62      double x = a;
63      double k1,k2,k3,k4;
64
65      for (int i=0; i<=N; i++)
66      {
67          //rk4out[i][1]=x;  same x as in euler
68          rk4out[i]=y;
69          //cout << y << "\t" << rk4out[i][1] << endl;
70          k1 = h*f(x,y);
71          k2 = h*f(x+(h/2.0),y+(k1/2.0));
72          k3 = h*f(x+(h/2.0),y+(k2/2.0));
73          k4 = h*f(x+h,y+k3);
74          y = y + (k1/6.0)+(k2/3.0)+(k3/3.0)+(k4/6.0);
75          x = x+h;
76      }
77  }
```

Runge-Kutta-Fehlberg 4th order in C++:

```
79  //Runge-Kutta-Fehlberg 4th order function to solve the ODE
80  void rkf45(double a, double b, double h, double y0, double rkf45out[][2])
81  {
82      int N = (int)abs((b-a)/h);
83      double y = y0;
84      double x = a;
85      double tol = 0.1;    //tolerance of the dynamic step size
86      double k1,k2,k3,k4,k5,k6,z;
87
88      for (int i=0; i<=N; i++)
89      {
90          rk45out[i][1]=x; //the x-value could differ because of dynamic h
91          rk45out[i][2]=y;
92          k1 = h*f(x,y);
93          k2 = h*f(x+(h/4),y+(k1/4));
94          k3 = h*f(x+(3*h/8),y+(3*k1/32)+(9*k2/32));
95          k4 = h*f(x+(12*h/13),y+(1932*k1/2197)-(7200*k2/2197)+(7296*k3/2197));
96          k5 = h*f(x+h,y+(439*k1/216)-8*k2+(3680*k3/513)-(845*k4/4104));
97          k6 = h*f(x+(h/2),y-(8*k1/27)+2*k2-(3544*k3/2565)+(1859*k4/4104)-(11*k5/40));
98          y = y + (25*k1/216)+(1408*k3/2565)+(2197*k4/4104)-(k5/5);
99          z = y + (16*k1/135)+(6656*k3/12825)+(28561*k4/56430)-(9*k5/50)+(2*k6/55);
100         h = h*pow((tol*h)/2*abs(z-y), 1/4); //dynamic step size
101         x = x+h;
102     }
103 }
```

And finally the main-programm which calls all numerical functions and writes the computed iterations into a file called “out.dat”.

```
109  int main ()
110  {
111  double a=0.0;
112  double b=3.0;
113  double h=0.5;
114  double y0=1.0;
115
116  int N = (int)abs((b-a)/h);
117
118  double eulerout[N][2];
119  double rk2out[N];
120  double rk4out[N];
121  double rkf45out[N][2];
122
123  euler(a,b,h,y0,eulerout);
124  rk2(a,b,h,y0,rk2out);
125  rk4(a,b,h,y0,rk4out);
126  rk45(a,b,h,y0,rkf45out);
127
128  //write all data in a file
129  ofstream os;
130  os.open("out.dat");
131  if (not os) cout << "Unable to open/write file 'out.dat'." << endl;
132  else
133      {
134      for (int i=0; i<=N; i++)
135      {
136          os << i << "\t" << eulerout[i][1] << "\t" << eulerout[i][2] << "\t" <<
137      }
138      os.close();
139      cout << "File out.dat written successfully." << endl;
140      }
141  }
```

The output file shows the number of steps and the needed x and y values to plot all the different numerical approached data:

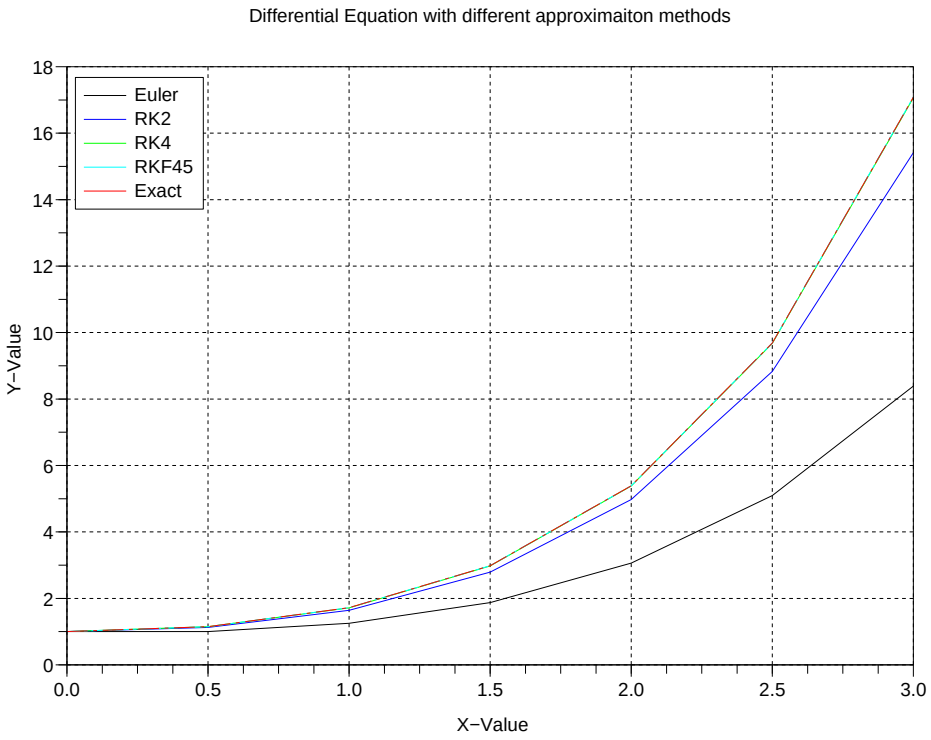
```
1  0   0   1        1        1        0   1
2  1   0.5 1        1.125    1.14844  0.5 1.14874
3  2   1   1.25     1.64062  1.71735  1   1.71834
4  3   1.5 1.875    2.79102  2.97938  1.5 2.98183
5  4   2   3.0625   4.9729   5.38397  2   5.38936
6  5   2.5 5.09375  8.83096  9.67201  2.5 9.68311
7  6   3   8.39062  15.4128  17.0648  3   17.0868
```

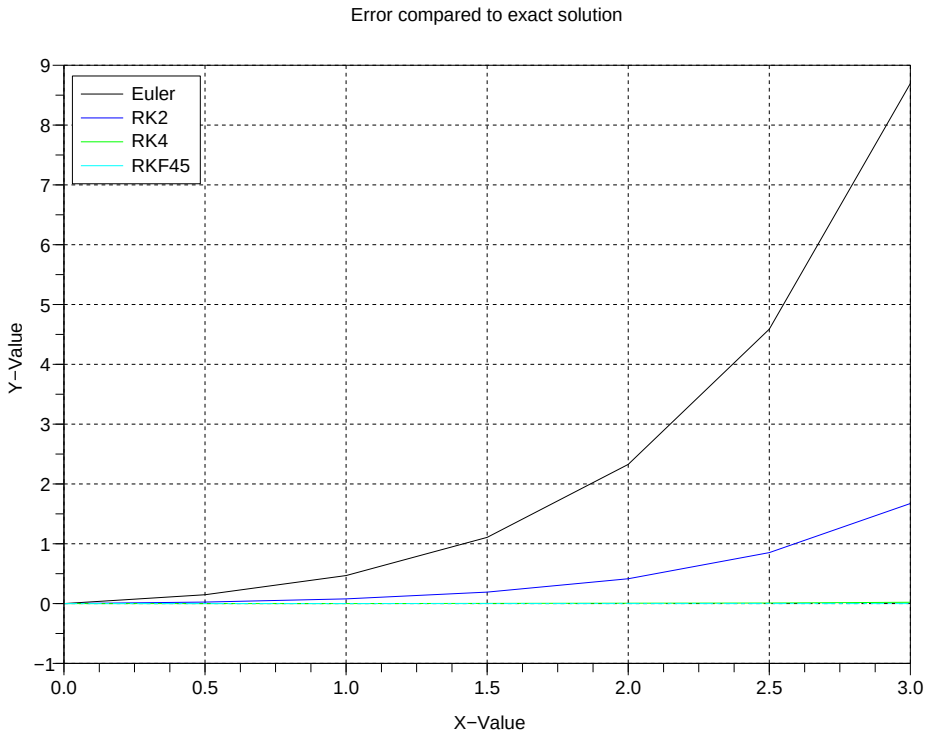
steps/x-value/Euler/RK2/RK4/x-RKF/RKF45

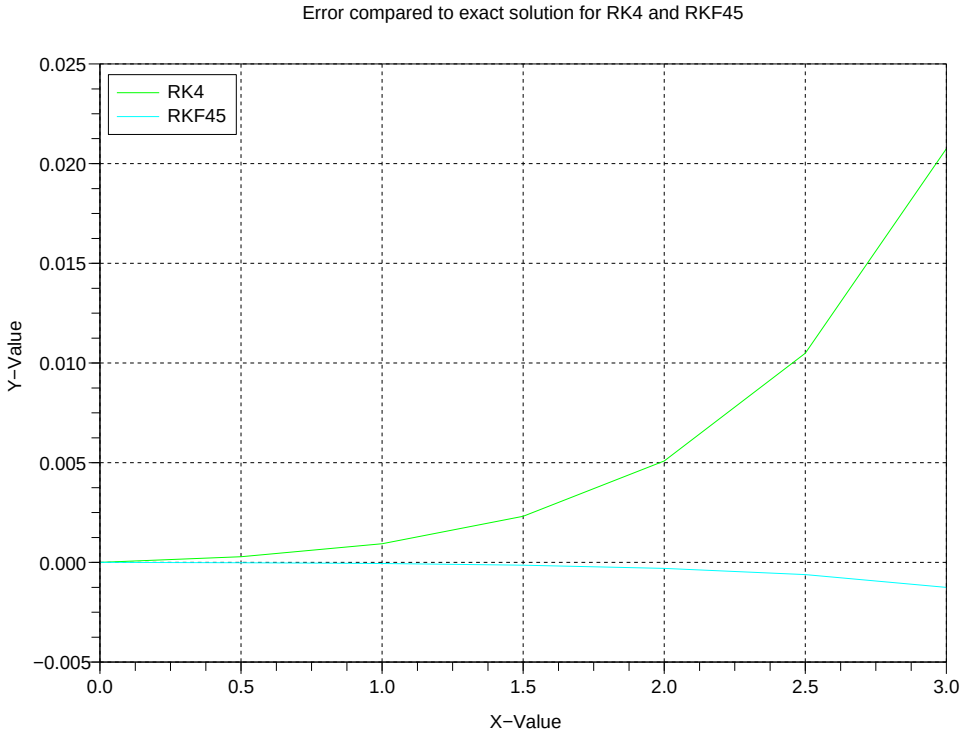
Here the Differential Equation $y' = y + x - 1$ was computed with the following preferences:

- area from $a = 0$ to $b = 3$
- step size $h = 0.5$
- start value $y_0 = 1.0$

Finally the data was plotted using SciLab:




Error compared to exact solution
Euler
RK2
RK4
RKF45
Y−Value
X−Value
9
8
7
6
5
4
3
2
1
0
−1
0.0
0.5
1.0
1.5
2.0
2.5
3.0


Error compared to exact solution for RK4 and RKF45
RK4
RKF45
Y−Value
X−Value
0.025
0.020
0.015
0.010
0.005
0.000
−0.005
0.0
0.5
1.0
1.5
2.0
2.5
3.0

I wrote a small script called "run" to do the work for you. All what you have to do is to change the DE in the "project.cc" as you like and in the Scilab file "dataplot.sce" to compare it to the numerical data in the output file of the C++ script, then start the script with "./run" and you should see the plotted graphs.