Scientific Computing

A practical Companion

3rd Notebook

This notebook can be downloaded from the location:
http://staff.science.uva.nl/~walter/SC/Notebooks/SC08-3.nb

Author:

Walter Hoffmann (Korteweg-de Vries Institute for Mathematics, UvA)

January - February, 2008

Accuracy, consistency and a higher order formula

Experiments with a non trivial ODE

We now consider an ODE which is still simple, but not as trivial as the example before. First we will show the solution curves of the ODE we are using.

In[93]:=

```
Clear["@"]
```

We use the package VisualDSolve (developed at Amstel Institute, UvA) that can be downloaded from

http://staff.science.uva.nl/~walter/SC/Notebooks/VisualDSolve.m

Loading the package by

In[94]:=

```
<< VisualDSolve`
```

Some parameters will be set.

In[95]:=

```
SetOptions[VisualDSolve,
  PlotStyle → {{Blue, Thickness[0.01]},
    {Red, Thickness[0.01]},
    {Green, Thickness[0.01]},
    {Magenta, Thickness[0.01]},
    {Yellow, Thickness[0.01]}}];
```

The package was developed under *Mathematica* 4; in *Mathematica* 5 a number of harmless messages are sort of a nuisance. We suppress their generation by:

In[96]:=

```
Off[NDSolve::precw]
```

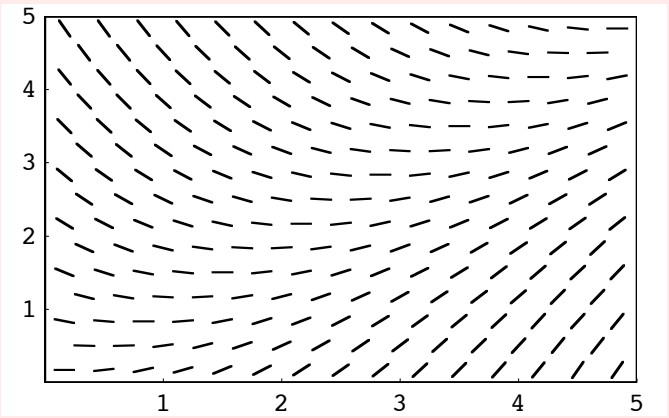
We consider the following ODE:

$$x'(t) = f(t, x) \equiv \frac{1}{2}(t - x(t))$$

Draw its vectorfield for a finite interval [0, 5] for both t and x:

In[97]:=

```
VisualDSolve[
  x'[t] == (t - x[t]) / 2, {t, 0, 5},
  {x, 0, 5}, DirectionField → True];
```

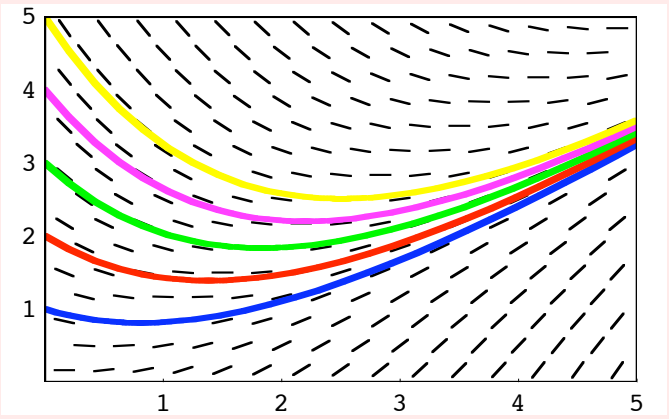


Some solution curves are drawn:

consider solutions with x(0) = 1, x(0) = 2, ... , x(0) = 5 :

In[98]:=

```
VisualDSolve[
  x'[t] == (t - x[t]) / 2, {t, 0, 5},
  {x, 0, 5}, DirectionField → True,
  InitialValues → {{0, 1}, {0, 2},
    {0, 3}, {0, 4}, {0, 5}}];
```



The exact solution for this ODE is:

$$x(t) = 3\,e^{-\frac{1}{2}t} + t - 2$$

On the correctness of this solution:

for the above function x(t) we have:

$$x'(t) = -\frac{3}{2}\,e^{-\frac{1}{2}t} + 1$$

so that indeed

$$x'(t) = f(t, x) \equiv \frac{1}{2}\,(t - x(t))$$

First order approximation

In[99]:=

```
der[t_][x_] := (t - x) / 2;
```

We will use Forward Euler (FE) to calculate an approximate solution on t=[0,5] for x(0)=1. FE reads:

$$x_{n+1} = x_n + \Delta t\, f(t_n, x_n).$$

As before, we are only interested in the final approximation error at the endpoint for an increasing number of intervals; we start with $\tau \equiv \Delta t = 1.0$ which corresponds with taking 5 steps.

In[100]:=

```
n = 5;
```

In[101]:=

```
Do[
  (tau = 5.0 / n; t = 0.0; x = 1;

   Do[
    ( x = x + tau * der[t][x]; t = t + tau
    ), {n}];

   Print["n = ", n , "   ",
    "Error  ",
    x - (3 * Exp[-t / 2] + t - 2)];

   n = n * 2
  ), {10}];
```

```
n = 5  Error  -0.152505
n = 10  Error  -0.0773145
n = 20  Error  -0.0386287
n = 40  Error  -0.019283
n = 80  Error  -0.00963116
n = 160  Error  -0.00481271
n = 320  Error  -0.00240561
n = 640  Error  -0.00120261
n = 1280  Error  -0.000601257
n = 2560  Error  -0.000300616
```

And again we observe the error being halved whenever the number of subintervals is doubled.

Second order, two step approximation

A simple formula having second order accuracy is the so called Leap Frog rule that needs both x0 and x1 to start with.

For the experiment on recognizing the order, we use the formula for the exact solution to initialize x1.

The Leap Frog scheme reads:

$$x_{n+1} = x_{n-1} + 2\,\Delta t\, f(t_n, x_n).$$

to get started we need both x0 and x1; for the time being we assume that next to x0 also x1 has been given.

As before, we start with an approximation over 5 intervals and we are going to double the number of steps in each experiment. Again, we only print the final approximation error.

In[102]:=

```
n = 5;
```

In[103]:=

```
Do[
  (tau = 5.0 / n; t = 0.0;
   x0 = 1; t = t + tau;
   x1 = 3 * Exp[-t / 2] + t - 2;

   Do[
    ( x2 = x0 + 2 * tau * der[t][x1];
     t = t + tau; x0 = x1; x1 = x2
    ), {n - 1}];

   Print["n = ", n , "   ",
    "Error   ",
    x2 - (3 * Exp[-t / 2] + t - 2)];

   n = n * 2
  ), {10}];
```

```
n = 5   Error   -0.148295
n = 10   Error   0.0402245
n = 20   Error   0.00670232
n = 40   Error   0.00109054
n = 80   Error   0.000189537
n = 160   Error   0.000036406
n = 320   Error   7.69363 × 10^-6
n = 640   Error   1.74524 × 10^-6
n = 1280   Error   4.13906 × 10^-7
n = 2560   Error   1.00668 × 10^-7
```

We observe that the error diminishes with at least a factor of 4 when the step size is halved.

Starting a two step method

In the experiment we just carried out, we were sort of cheating on the value of x1.

Suggestion: Compute x1 also by some approximating formula; let us try FE:

In[104]:=

```
n = 5;
```

In[105]:=

```
Do[
  (tau = 5.0 / n; t = 0.0;
   x0 = 1;
   x1 = x0 + tau * der[t][x0]; t = t + tau;


   Do[
    ( x2 = x0 + 2 * tau * der[t][x1];
     t = t + tau; x0 = x1; x1 = x2
    ), {n - 1}];

   Print["n = ", n , "   ",
    "Error   ",
    x2 - (3 * Exp[-t / 2] + t - 2)];

   n = n * 2
  ), {10}];
```

```
n = 5   Error   -1.74625
n = 10   Error   0.534507
n = 20   Error   0.140843
n = 40   Error   0.0356892
n = 80   Error   0.00895265
n = 160   Error   0.00224007
n = 320   Error   0.000560136
n = 640   Error   0.000140041
n = 1280   Error   0.0000350108
n = 2560   Error   8.75273 × 10^-6
```

We do see the effect of the error being devided by 4 when the number of steps is dubbled, but the less accurate result of the value for x1 remains visible in all remaining calculations.

There do exist one-step formulae with an order of accuracy higher than 1. For instance the Modified Euler method which is defined by:

$$x_{n+1} = x_n + \Delta t\, f\left(t_n + \frac{1}{2}\Delta t,\ x_n + \frac{1}{2}\Delta t\, f(t_n, x_n)\right).$$

In[106]:=

```
n = 5;
```

In[107]:=

```
Do[
  (tau = 5.0 / n; t = 0.0;
   x0 = 1;
   k1 = tau * der[t][x0];
   k2 = tau * der[t + tau / 2][x0 + k1 / 2];
   x1 = x0 + k2;
   t = t + tau;

   Do[
    ( x2 = x0 + 2 * tau * der[t][x1];
     t = t + tau; x0 = x1; x1 = x2
    ), {n - 1}];
   Print["n = ", n , "   ",
    "Error  ",
    x2 - (3 * Exp[-t / 2] + t - 2)];

   n = n * 2
  ), {10}];
```

```
n = 5  Error  0.128745
n = 10  Error  -0.0018092
n = 20  Error  0.00105539
n = 40  Error  0.000366
n = 80  Error  0.0000980171
n = 160  Error  0.0000249137
n = 320  Error  6.25404×10^-6
n = 640  Error  1.56511×10^-6
n = 1280  Error  3.91378×10^-7
n = 2560  Error  9.78509×10^-8
```

Apart from a slightly irregular behavior for smaller values of n, we nicely observe again the "factor 4 behavior". The final result is even slightly better than in the case that x1 was calculated from the theoretical solution.

A single step second order method

The Modified Euler formula can of course be used all over:

In[108]:=

```
n = 5;
```

In[109]:=

```
Do[
  (tau = 5.0 / n; t = 0.0; x = 1;

   Do[
    (k1 = tau * der[t][x];
     k2 = tau * der[t + tau / 2][x + k1 / 2];
     x = x + k2;
     t = t + tau
    ), {n}];

   Print["n = ", n , "   ",
    "Error  ",
    x - (3 * Exp[-t / 2] + t - 2)];

   n = n * 2
  ), {10}];
```

```
n = 5  Error  0.0398473
n = 10  Error  0.00785489
n = 20  Error  0.0017673
n = 40  Error  0.000420421
n = 80  Error  0.000102601
n = 160  Error  0.0000253471
n = 320  Error  6.29948×10^-6
n = 640  Error  1.57025×10^-6
n = 1280  Error  3.91986×10^-7
n = 2560  Error  9.79247×10^-8
```

To compare the cost of various formulae, as a rule, we count the number of function evaluations (evaluations of

the derivative) within one step of the loop.

For Leap Frog this is equal to one and for Modified Euler it is equal to two, which makes Modified Euler twice as expensive as Leap Frog for about the same accuracy.

A fourth-order Runge Kutta method

The principle of of evaluating the formula of the differential equation in internal points to calculate a better guess for stepping to the next point can be extended to higher order formulas. A whole class of formulas has been designed according to this idea; these are the so called Runge-Kutta*) formulas.

A famous formula from this class is the fourth-order Runge-Kutta formula or just for short the RK4 formula.

The RK4 formula is defined as follows:

$$
\begin{aligned}
k_1 &= \Delta t\, f(t_n,\ x_n)\\
k_2 &= \Delta t\, f\left(t_n + \frac{1}{2}\Delta t,\ x_n + \frac{1}{2}k_1\right)\\
k_3 &= \Delta t\, f\left(t_n + \frac{1}{2}\Delta t,\ x_n + \frac{1}{2}k_2\right)\\
k_4 &= \Delta t\, f(t_n + \Delta t,\ x_n + k_3)\\
x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4).
\end{aligned}
$$

*) C.D.T. Runge 1856 - 1927; M.W. Kutta 1867 - 1944

We show an example where it is applied for solving the same differential equation as before:

$$x'(t) = f(t, x) \equiv \frac{1}{2}(t - x(t))$$

to be solved on $t = [0, 5]$ with $x(0) = 1.$

and with solution

$$x(t) = 3\, e^{-\frac{1}{2}t} + t - 2$$

In[110]:=

```
n = 5; error = 1.;
```

In[111]:=

```
Do[
  (tau = 5.0 / n; t = 0.0; x = 1;

   Do[
    (k1 = tau * der[t][x];
     k2 = tau * der[t + tau / 2][x + k1 / 2];
     k3 = tau * der[t + tau / 2][x + k2 / 2];
     k4 = tau * der[t + tau][x + k3];
     x = x + (k1 + 2 k2 + 2 k3 + k4 ) / 6;
     t = t + tau
    ), {n}];

   olderror = error;
   error = x - (3 * Exp[-t / 2] + t - 2);
   fact = olderror / error;

   Print["n = ", n , "   ",
    "Error   ",
    error, " factor: ", fact];

   n = n * 2
  ), {10}];
```

```
n = 5   Error   0.000487946 factor: 2049.41
n = 10   Error   0.0000246983 factor: 19.7562
n = 20   Error   1.39024×10^-6 factor: 17.7654
n = 40   Error   8.24708×10^-8 factor: 16.8574
n = 80   Error   5.02178×10^-9 factor: 16.4226
n = 160   Error   3.09798×10^-10 factor: 16.2099
```

n = 320 Error 1.92375×10^{-11} factor: 16.1039

n = 640 Error 1.19504×10^{-12} factor: 16.0977

n = 1280 Error 7.81597×10^{-14} factor: 15.2898

n = 2560 Error 6.66134×10^{-15} factor: 11.7333

We more or less see the effect that the error is diminished by a factor of 16 each time the step size is cut in 2.

A comparison of costs tells us the following:

for each internal step we need four function evaluations, but doubling the number of steps increases the accuracy by a factor of 16.

To reach an accuracy of about 10^{-7} in RK4 we need n = 40, which corresponds with 160 function evaluations.

To reach the same accuracy with Modified Euler (a 2nd order method) we need n= 2560, which corresponds with 5120 function evaluations.

Close all sections