Scientific Computing

A practical Companion

2nd Notebook

This notebook can be downloaded from the location:
http://staff.science.uva.nl/~walter/SC/Notebooks/SC08-2.nb

Author:

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

January - February, 2008

Ordinary differential equations II

Improved simple numerical algorithm ("Backward Euler").

The improvement we propose leads to the so called Backward Euler method, which is the simplest example of an implicit method.

it is based on the following approximation:

$$\frac{x_{n+1} - x_n}{\Delta t} \approx x'(t_{n+1}).$$

From this we arrive at:

$$x_{n+1} = x_n + \Delta t\, x'(t_{n+1}).$$

Mostly this leads to a non linear equation for x_{n+1} which can only be solved iteratively.

Here we concentrate on the differential equation from our simple example:

$$x'(t) = 10\, x(t) - 10\, x(t)^2$$

Recall the picture for the vectorfield with this equation.

In[1]:=

```
<< VisualDSolve`
```

In[2]:=

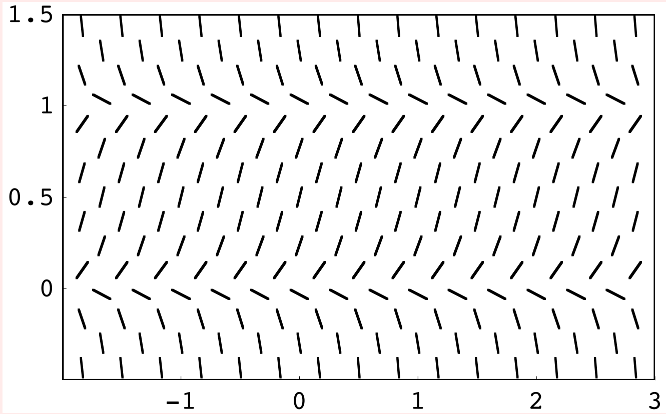
```
SetOptions[VisualDSolve,
  PlotStyle → {{Blue, Thickness[0.01]}, {Red, Thickness[0.01]},
    {Green, Thickness[0.01]}, {Magenta, Thickness[0.01]},
    {Yellow, Thickness[0.01]}}];
```

In[3]:=

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

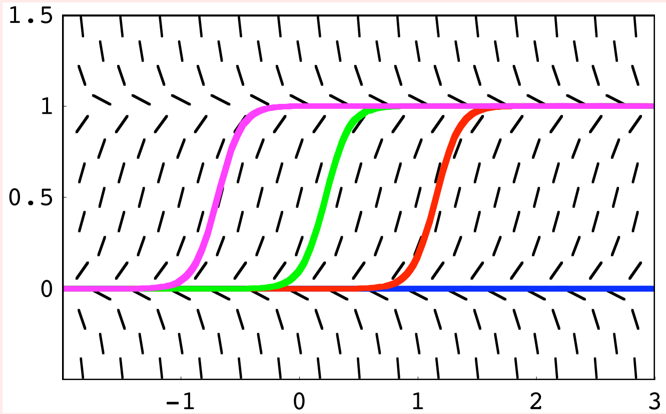
In[4]:=

```
VisualDSolve[x'[t] == 10 x[t] - 10 x[t]^2,
    {t, -2, 3}, {x, -0.5, 1.5}, DirectionField → True];
```



In[5]:=

```
VisualDSolve[x'[t] == 10 x[t] - 10 x[t]^2,
    {t, -2, 3}, {x, -0.5, 1.5}, DirectionField → True,
    InitialValues → {{0, 0}, {0, 0.00001}, {0, 0.1},  {0, 0.999}}];
```



Mathematica has solved the ODE and drawn the pictures using a sophisticated formula for solving ODE's.

Application of Backward Euler to solve the equation leads to

$$x_{n+1} = x_n + \Delta t\left(10\, x_{n+1} - 10\, x_{n+1}{}^2\right)$$

Using the well known formula for solving quadratics and again putting `a = 10 Δt`, we arrive at:

In[6]:=

```
ImplFunction[a_][x_] := (a - 1 + Sqrt[(1 - a)^2 + 4 * a * x]) / (2 * a);
```

In[7]:=

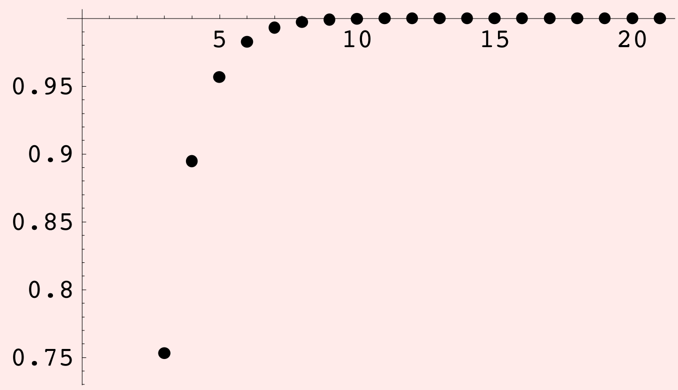
```
h[x_] := ImplFunction[1.5][x];
```

In[8]:=

```
NestList[h, 0.1, 20]
ListPlot[%, PlotStyle -> PointSize[0.02]];
```

Out[8]=

```
{0.1, 0.473985, 0.752984, 0.894518, 0.956681,
 0.982488, 0.992966, 0.997182, 0.998872,
 0.999549, 0.999819, 0.999928, 0.999971, 0.999988,
 0.999995, 0.999998, 0.999999, 1., 1., 1., 1.}
```



In[10]:=

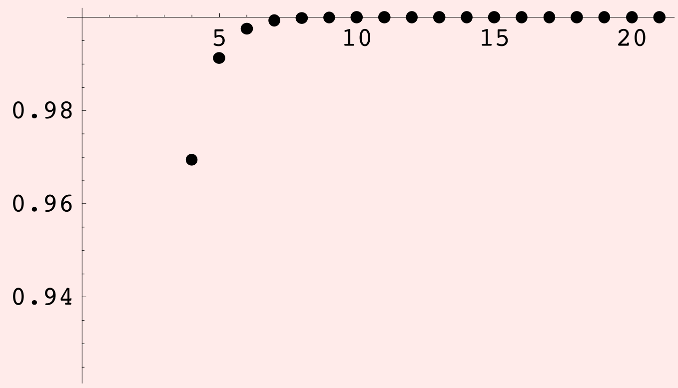
```
h[x_] := ImplFunction[2.5][x];
```

In[11]:=

```
NestList[h, 0.1, 20]
ListPlot[%, PlotStyle -> PointSize[0.02]];
```

Out[11]=

```
{0.1, 0.660555, 0.895166, 0.969377, 0.991195, 0.99748,
 0.99928, 0.999794, 0.999941, 0.999983, 0.999995,
 0.999999, 1., 1., 1., 1., 1., 1., 1., 1., 1.}
```



In[13]:=

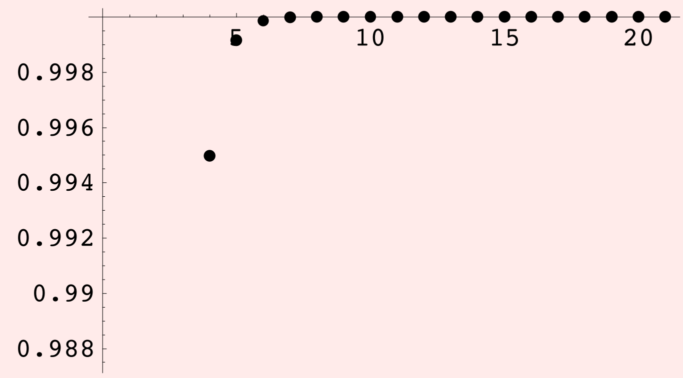
```
h[x_] := ImplFunction[5][x];
```

In[14]:=

```
NestList[h, 0.1, 20]
ListPlot[%, PlotStyle -> PointSize[0.02]];
```

Out[14]=

```
{0.1, 0.824264, 0.969959, 0.994972, 0.999161,
 0.99986, 0.999977, 0.999996, 0.999999, 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}
```



In[16]:=

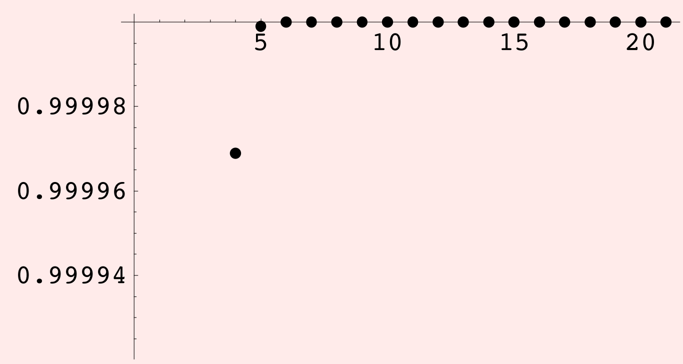
```
h[x_] := ImplFunction[30][x];
```

In[17]:=

```
NestList[h, 0.1, 20]
ListPlot[%, PlotStyle -> PointSize[0.02]];
```

Out[17]=

```
{0.1, 0.970103, 0.999035, 0.999969, 0.999999, 1., 1.,
 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.}
```



It appears that, in contrast to Forward Euler, we can use arbitrarily large steps in Backward Euler without the numerical solution going to oscilate or even become chaotic as we have seen in the example with Forward Euler.

Stepsize has to do with

- Stability

  and

- Accuracy.

Stability of a numerical algorithm for solving ODE's.

We look for a method to judge the stability of numerical algorithms. The two algorithms we know until now are Forward and Backward Euler. To judge the stabilty of a given ODE-solver, scientists have settled for considering a "standard problem" and give a report for the quantitative behaviour of the ODE-solver under consideration for this standard problem.

The standard problem is:

Compute an approximation of

$$u = u(t) \text{ for } t > 0$$

for the standard differential equation

$$u' = \lambda u \quad \text{with} \quad u(0) = u_0 \,.$$

The general solution for this problem is:

$$u = u_0 \, e^{\lambda t} \quad .$$

Many problems in science are qualitatively of this type.

For our *Stability Standard*, we only consider this type of problems with $\mathrm{Re}(\lambda) < 0$, which says that the solution decays with increasing values of t.

What does it imply for Forward Euler?

$$\frac{u_{n+1} - u_n}{\Delta t} \approx u'(t_n).$$

For advancing one step we find:

$$u_{n+1} = u_n + \Delta t\, u'(t_n) = u_n + \lambda \Delta t\, u_n$$
$$= (1 + \lambda \Delta t)\, u_n$$

and we demand that possible inaccuracies in approximations for u_n are not magnified to spoil the approximation for u_{n+1} ; this sets a requirement on the stepsize Δt in this formula:

$$\left| 1 + \lambda \Delta t \right| < 1$$

The factor $(1 + \lambda\Delta t)$ is called "the amplification factor".

For problems with real (negative) values of λ this states

$$-1 < 1 - |\lambda| \Delta t < 1$$

or

$$-2 < -|\lambda| \Delta t < 0$$

which comes down to

$$\Delta t < \frac{2}{|\lambda|} .$$

How realistic is this bound?

One should be aware of the fact that for $\Delta t > \frac{1}{|\lambda|}$ the factor $(1 + \lambda\Delta t)$ becomes negative (although in absolute value less than 1). So already for values of Δt (much) less than the stability bound, we have a sequence of approximations that is alternating in sign, although decreasing in absolute value, but this is not very relevant.

The alternating approximations are useless, but to be strict: the row of approximations is decreasing in absolute value.

The important fact is, that indeed there is an upperbound for the stepsize, but long before reaching this stepsize, the accuracy has become abominable!

Let us illustrate this by an experiment.

Consider the equation $u' = -15\,u$ with $u(0) = 1$ to be solved on [0,1].

We are going to compute the value in t = 1, by succesively applying Forward Euler using various stepsizes.

According to theory, the stability bound gives $\Delta t < 2/15 \sim 0.13$.

Long before that, we shall see that any accuracy is completely lost.

Application of Forward Euler yields

$$u_0 = 1$$

$$u_{n+1} = u_n + \Delta t\,(-15\,u_n) = u_n\,(1 - 15\,\Delta t)$$

For a number of values for Δt we compute the following results:

In[19]:=

```
n = 1024 * 5;
```

In[20]:=

```
F = Exp[-15.]
Do[
  (h = 1.0/n; u = 1;

    Do[
      u = u (1 - 15 h)
     , {n}
    ];

    Print[
     "App: ", u,
     " , RErr: ", (F - u)/u,
     " ,  Aerr:  ", F - u,
     " , d t = ", N[1/n],
     " , n: ", n
        ];
    n = n/2
  ), {11}
]
```

Out[20]=

3.05902×10^{-7}

App: 2.99241×10^{-7} , RErr: 0.0222598 , Aerr: 6.66105×10^{-9} , d t = 0.000195313 , n: 5120

App: 2.927×10^{-7} , RErr: 0.0451054 , Aerr: 1.32023×10^{-8} , d t = 0.000390625 , n: 2560

App: 2.7997×10^{-7} , RErr: 0.0926253 , Aerr: 2.59323×10^{-8} , d t = 0.00078125 , n: 1280

```
App: 2.55875×10^-7 , RErr: 0.195515 ,  Aerr:
 5.00274×10^-8 , d t = 0.0015625 , n: 640

App: 2.12792×10^-7 , RErr: 0.437564
  ,  Aerr:  9.31102×10^-8 , d t = 0.003125 , n: 320

App: 1.44439×10^-7 , RErr: 1.11787
  ,  Aerr:  1.61464×10^-7 , d t = 0.00625 , n: 160

App: 6.10759×10^-8 , RErr: 4.00856
  ,  Aerr:  2.44826×10^-7 , d t = 0.0125 , n: 80

App: 6.84228×10^-9 , RErr: 43.7077
  ,  Aerr:  2.9906×10^-7 , d t = 0.025 , n: 40

App: 9.09495×10^-13 , RErr: 336342.
  ,  Aerr:  3.05901×10^-7 , d t = 0.05 , n: 20

App: 0.000976563 , RErr: -0.999687
  ,  Aerr:  -0.000976257 , d t = 0.1 , n: 10

App: -32. , RErr: -1. ,  Aerr:  32. , d t = 0.2 , n: 5
```

And now for the situation with Backward Euler.

$$\frac{u_{n+1} - u_n}{\Delta t} \approx u'(t_{n+1}).$$

For advancing one step we find:

$$u_{n+1} = u_n + \Delta t\, u'(t_{n+1}) = u_n + \lambda \Delta t\, u_{n+1}$$
$$u_{n+1} - \lambda \Delta t\, u_{n+1} = u_n$$
$$u_{n+1} = u_n / (1 - \lambda \Delta t)$$

So now the requirement on Δt should be

$$\left| \frac{1}{1 - \lambda \Delta t} \right| < 1$$

And for problems with real (negative) values of λ this states

$$1 + |\lambda| \Delta t > 1$$

So that with respect to stability, no bound is imposed on the value of Δt .

Let us illustrate this by the same experiment we used with Forward Euler.

Again consider the equation $u' = -15\,u$ with $u(0) = 1$ to be solved on [0,1].

The value in t = 1 is computed now by succesively applying Backward Euler using various stepsizes.

Application of Backward Euler yields

$$u_0 = 1$$

$$u_{n+1} = u_n + \Delta t\,(-15\,u_{n+1})$$

giving

$$u_{n+1} = u_n / (1 + 15\,\Delta t)$$

For the same values of Δt we compute the results:

In[22]:=

```
n = 1024 * 5;
```

In[23]:=

```
Exp[-15.]
Do[
  (h = 1.0/n; u = 1;

    Do[
      u = u/(1 + 15 h)
      , {n}
    ];

    Print[
      "App: ", u,
      " , RErr: ", (Exp[-15.] - u)/u,
      " , Aerr: ", Exp[-15.] - u,
      " , d t = ", N[ 1/n],
      " , n: ", n
         ];
    n = n/2
  ), {11}
]
```

Out[23]=

3.05902×10^{-7}

App: 3.12685×10^{-7} , RErr: -0.0216911 , Aerr: -6.78248×10^{-9} , d t = 0.000195313 , n: 5120

App: 3.1959×10^{-7} , RErr: -0.0428301 , Aerr: -1.36881×10^{-8} , d t = 0.000390625 , n: 2560

App: 3.33778×10^{-7} , RErr: -0.0835153 , Aerr: -2.78756×10^{-8} , d t = 0.00078125 , n: 1280

```
App: 3.63707 × 10^-7 , RErr: -0.158931 ,  Aerr:
 -5.78044 × 10^-8 , d t = 0.0015625 , n: 640

App: 4.30185 × 10^-7 , RErr: -0.288904 ,  Aerr:
 -1.24282 × 10^-7 , d t = 0.003125 , n: 320

App: 5.93075 × 10^-7 , RErr: -0.48421 ,  Aerr:
 -2.87172 × 10^-7 , d t = 0.00625 , n: 160

App: 1.06982 × 10^-6 , RErr: -0.714062
  ,  Aerr:  -7.63917 × 10^-7 , d t = 0.0125 , n: 80

App: 2.93692 × 10^-6 , RErr: -0.895842
  ,  Aerr:  -2.63102 × 10^-6 , d t = 0.025 , n: 40

App: 0.0000137797 , RErr: -0.9778
  ,  Aerr:  -0.0000134738 , d t = 0.05 , n: 20

App: 0.000104858 , RErr: -0.997083
  ,  Aerr:  -0.000104552 , d t = 0.1 , n: 10

App: 0.000976563 , RErr: -0.999687
  ,  Aerr:  -0.000976257 , d t = 0.2 , n: 5
```

Stiff problems

Forward and backward Euler for a stiff problem

Consider the equation:

$$x' = -1000\, x - e^{-t} \;;\; x(0) = 0;$$

The analytic solution is:

$$x(t) = \frac{1}{999}\left(e^{-1000\, t} - e^{-t}\right);$$

The solution of the ODE shows a behaviour that is ruled by timescales which are far apart: slowly and quickly varying in time. That is typical for a so called "stiff equation". With slowly varying phenomena, one can take large time-steps when integrating the differential equation. Because of the quickly varying part, this is not possible. In such cases an implicit formula (like Backward Euler) tends to allow much larger time-steps then an explicit formula, such as Forward Euler.

For the given equation we compare Forward and Backward Euler.

Forward Euler gives:

$$x_{n+1} = x_n + h\left(-1000\,x_n - e^{-n\,h}\right);\quad x_0 = 0;$$

Backward Euler gives:

$$x_{n+1} = \frac{1}{1 + 1000\,h}\left(x_n - h\,e^{-(n+1)\,h}\right);\quad x_0 = 0;$$

Implementation of Forward Euler on the interval [0, 0.1] gives:

In[25]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 0; n = 9; i = 5;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp; xp =
      xn + h * (-1000 * xn - Exp[-k * h]) ;
     k = k + 1
    ), {i}];
   Err = xp -
     (Exp[-1000 * (k * h)] - Exp[-(k * h) ]) /
      999;
  Print["node = ", k * h ,
    "   ", xp, "   ",
    "Error   ", Err, "   Rel.   ", Err / xp];

  ), {n}];
```

```
node = 0.0111111  -0.00372008
   Error  -0.00273015  Rel.  0.733897

node = 0.0222222  0.00646722
   Error  0.00744622  Rel.  1.15138

node = 0.0333333  -0.0212771
   Error  -0.0203089  Rel.  0.954496

node = 0.0444444  0.0544333
   Error  0.0553908  Rel.  1.01759

node = 0.0555556  -0.15202
   Error  -0.151074  Rel.  0.993771

node = 0.0666667  0.411103  Error  0.41204  Rel.  1.00228

node = 0.0777778  -1.12473  Error  -1.1238  Rel.  0.999177
```

```
node = 0.0888889   3.06416   Error   3.06507   Rel.   1.0003

node = 0.1   -8.36063   Error   -8.35972   Rel.   0.999892
```

Implementation of Backward Euler for the same interval [0, 0.1] gives:

In[29]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 0; n = 9; i = 5;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp;
     xp = ( xn - h * Exp[- (k + 1) * h] ) /
        (1 + 1000 h);
     k = k + 1
    ), {i}];
   Err = xp -
     (Exp[-1000 * (k * h)] - Exp[- (k * h) ]) /
      999;
   Print["node = ", k * h ,
    "   ", xp, "   ",
    "Error   ", Err, "   Rel.   ", Err / xp];

  ), {n}];
```

node = 0.0111111   -0.00098706
Error 2.8657×10^{-6} Rel.   -0.00290327

node = 0.0222222   -0.000978995
Error 7.20632×10^{-9} Rel. -7.36094×10^{-6}

node = 0.0333333   -0.000968185
Error -1.05375×10^{-9} Rel. 1.08838×10^{-6}

node = 0.0444444 -0.000957487
Error -1.06566×10^{-9} Rel. 1.11298×10^{-6}

node = 0.0555556 -0.000946907
Error -1.05395×10^{-9} Rel. 1.11305×10^{-6}

node = 0.0666667 -0.000936444
Error -1.04231×10^{-9} Rel. 1.11305×10^{-6}

node = 0.0777778 -0.000926097
Error -1.03079×10^{-9} Rel. 1.11305×10^{-6}

node = 0.0888889 -0.000915864
Error -1.0194×10^{-9} Rel. 1.11305×10^{-6}

node = 0.1 -0.000905744 Error
-1.00814×10^{-9} Rel. 1.11305×10^{-6}

Even using n=9, i=2 gives good results in the B.E. case.

Forward and backward Euler for a non-stiff problem

▽ A 'harmless' form of the previous equation is:

$$x' = -2x - e^{-t} \;;\; x(0) = 0;$$

The solution reads:

$$x(t) = e^{-2t} - e^{-t}\;;$$

Forward Euler gives:

$$x_{n+1} = x_n + h\left(-2x_n - e^{-nh}\right)\;;\; x_0 = 0;$$

Backward Euler gives:

$$x_{n+1} = \frac{1}{1+2h}\left(x_n - h\,e^{-(n+1)h}\right)\;;\; x_0 = 0;$$

Implementation of Forward Euler gives:

In[33]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 0; n = 5; i = 2;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp;
     xp =  xn  + h * (-2 * xn  - Exp[-k * h]) ;
     k = k + 1
    ), {i}];
   Err = xp -
     (Exp[-2 * (k * h)] - Exp[- (k * h) ]);
   Print["node = ", k * h ,
    "   ", xp, "   ",
    "Error   ", Err, "   Rel.   ", Err / xp];

  ), {n}];
```

```
node = 0.02  -0.0197005
 Error  -0.000291264  Rel.  0.0147846

node = 0.04  -0.0382308
 Error  -0.000557668  Rel.  0.0145869

node = 0.06  -0.0556449
 Error  -0.000800757  Rel.  0.0143905

node = 0.08  -0.0719945
 Error  -0.00102199  Rel.  0.0141954

node = 0.1  -0.0873294
 Error  -0.00122275  Rel.  0.0140016
```

Implementation of Backward Euler gives:

In[37]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 0; n = 5; i = 2;
h = (te / n) / i;
Do[(

    Do[
     (xn = xp;
      xp = ( xn  - h*Exp[-(k + 1)*h] ) /
         (1 + 2 h);
      k = k + 1
     ), {i}];
    Err = xp -
      (Exp[-2*(k*h)] - Exp[-(k*h) ]);
   Print["node = ", k*h ,
     "   ", xp, "   ",
     "Error   ", Err, "   Rel.   ", Err/xp];

  ), {n}];
```

```
node = 0.02  -0.0191258
 Error  0.000283393  Rel.  -0.0148173

node = 0.04  -0.0371303
 Error  0.000542808  Rel.  -0.014619

node = 0.06  -0.0540644
 Error  0.00077972  Rel.  -0.0144221

node = 0.08  -0.069977
 Error  0.000995528  Rel.  -0.0142265

node = 0.1  -0.0849151
 Error  0.00119155  Rel.  -0.0140323
```

▽ Another harmless variant is:

$$x' = -x - e^{-t};$$

Firts with initial value

$$x(0) = 1;$$

Then the solution reads:

$$x(t) = e^{-t} - t\,e^{-t};$$

Forward Euler gives:

$$x_{n+1} = x_n + h\left(-x_n - e^{-n h}\right); \quad x_0 = 1;$$

Backward Euler gives:

$$x_{n+1} = \frac{1}{1 + h}\left(x_n - h\,e^{-(n+1) h}\right); \quad x_0 = 1;$$

Implementation Forward Euler gives:

In[41]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 1; n = 5; i = 2;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp;
     xp =  xn + h * (-xn - Exp[-k * h]) ;
     k = k + 1
    ), {i}];
   Err =
    xp - ((1 - (k * h)) * Exp[-(k * h) ]);
   Print["node = ", k * h ,
    "   ", xp, "   ",
    "Error   ", Err, "   Rel.   ", Err / xp];

  ), {n}];
```

```
node = 0.02  0.9603  Error
 -0.000295198  Rel.  -0.000307402

node = 0.04  0.921781  Error
 -0.000576742  Rel.  -0.000625682

node = 0.06  0.884414  Error
 -0.000845096  Rel.  -0.000955544

node = 0.08  0.848166
 Error  -0.00110071  Rel.  -0.00129775

node = 0.1  0.81301  Error  -0.00134402  Rel.  -0.00165314
```

Implementation Backward Euler gives:

In[45]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 1; n = 5; i = 2;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp; xp =
      ( xn  - h * Exp[- (k + 1) * h] ) / (1 + h);
     k = k + 1
    ),  {i}];
   Err =
    xp -  ((1  - (k * h)) * Exp[- (k * h) ]);
   Print["node  =  ",  k * h ,
    "   ", xp, "   ",
    "Error   ", Err,  "   Rel.   ", Err / xp];

  ), {n}];
```

```
node = 0.02  0.960886
 Error  0.000290993  Rel.  0.000302838

node = 0.04  0.922926
 Error  0.000568581  Rel.  0.000616063

node = 0.06  0.886092
 Error  0.000833218  Rel.  0.000940329

node = 0.08  0.850352  Error  0.00108534  Rel.  0.00127635

node = 0.1  0.815679  Error  0.00132539  Rel.  0.00162489
```

With initial value x(0) = 0, the solution becomes different:

$$x' = -x - e^{-t} ;\ \ x(0) = 0;$$

The solution reads:

$$x(t) = -t\,e^{-t};$$

Forward Euler still gives:

$$x_{n+1} = x_n + h\left(-x_n - e^{-n\,h}\right);\quad x_0 = 0;$$

And backward Euler gives:

$$x_{n+1} = \frac{1}{1+h}\left(x_n - h\,e^{-(n+1)\,h}\right);\quad x_0 = 0;$$

Implementation of Forward Euler gives:

In[49]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 0; n = 5; i = 2;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp;
     xp =  xn  + h * (-xn  - Exp[-k * h]) ;
     k = k + 1
    ), {i}];
   Err = xp + (k * h) * Exp[- (k * h) ];
   Print["node  =  ",
    k * h , "   ", xp, "   ",
    "Error   ", Err,  "   Rel.   ", Err / xp];

  ), {n}];
```

```
node = 0.02  -0.0198005
 Error  -0.000196525  Rel.  0.00992525
node = 0.04  -0.0388149
 Error  -0.000383313  Rel.  0.00987541
node = 0.06  -0.0570666
 Error  -0.000560712  Rel.  0.00982557
node = 0.08  -0.0745784
 Error  -0.000729058  Rel.  0.00977573
node = 0.1  -0.0913724
 Error  -0.000888678  Rel.  0.00972589
```

Implementation of Backward Euler gives:

In[53]:=

```
te = 0.1; t0 = 0.0; k = 0;
xp = 0; n = 5; i = 2;
h = (te / n) / i;
Do[(

   Do[
    (xn = xp; xp =
      ( xn  -  h * Exp[- (k + 1) * h] ) / (1 + h);
     k = k + 1
    ),  {i}];
   Err = xp  +  (k * h) * Exp[- (k * h) ];
   Print["node  =  ",
    k * h , "    ", xp, "    ",
    "Error    ", Err,  "   Rel.    ", Err / xp];

  ), {n}];
```

```
node = 0.02  -0.0194104
 Error  0.000193617  Rel.  -0.00997492

node = 0.04  -0.0380539
 Error  0.000377676  Rel.  -0.00992475

node = 0.06  -0.0559534
 Error  0.000552516  Rel.  -0.00987459

node = 0.08  -0.0731308
 Error  0.000718469  Rel.  -0.00982443

node = 0.1  -0.0896079
 Error  0.000875852  Rel.  -0.00977427
```

Order of approximation for a numerical ODE formula.

For the definition of the order of a formula for solving ODE's, we compare the formula for calculating x_{n+1} with Taylor's expansion for x_{n+1} and observe the smallest exponent of the stepsize that still occurs in the difference.

Definition:

The order of approximation (or simply 'order', for short) of a formula for numerically solving ODE's, is 1 less than the smallest exponent of the stepsize that is still present in a formula for the LOCAL error.

Example1:

For Forward Euler we have:

$$x_{n+1} = x_n + \Delta t\, x_n'$$

We find fromTaylor's expansion:

$$x_{n+1} = x_n + \Delta t\, x_n' + \frac{1}{2} (\Delta t)^2 x_n'' + O((\Delta t)^3)$$

The difference between the two being

$$\frac{1}{2} (\Delta t)^2 x_n'' + O((\Delta t)^3)$$

leads to declaring Forward Euler as a formula of order 1

(1 less then the smallest exponent of (Δt))

Example2:

For Backward Euler we have:

$$x_{n+1} = x_n + \Delta t\, x'_{n+1}$$

We first expand x'_{n+1} in a Taylor series:

$$x'_{n+1} = x'_n + \Delta t\, x_n'' + \frac{1}{2} (\Delta t)^2 x_n''' + O((\Delta t)^3)$$

From this we find for Backward Euler:

$$x_{n+1} = x_n + \Delta t\, x_n' + (\Delta t)^2 x_n'' + O((\Delta t)^3)$$

and the difference between this expansion and the approximating formula is

$$(\Delta t)^2 x_n'' + O((\Delta t)^3)$$

so that Backward Euler is also a formula of order 1

WARNING:

Be aware of the fact that we used the LOCAL error for defining the order of a formula. In practice, a formula is used over a succession of subintervals; a given interval [a,b] (say) is devided in a number of subintervals and to arrive at the final result, the approximating formula for solving the ODE is applied over all intervals in succession. That may give rise to a GLOBAL error which can be as large as the sum of all local errors.

So, only observing the order of magnitude, we have that the GLOBAL error can be as large as n times the local error. Symbolically this can be written as:

$$\text{Global\_error} \approx n * \text{Local\_error} = \frac{(b-a)}{\Delta t} * \text{Local\_error}$$

It is customary to have a factor $(b - a)$ in the expression for the global error and therefore the smallest exponent in the stepsize occurring in an expression for the global error is one less then in an expression for the local error.

Stepsize is mostly denoted by (Δt) for a formula having <u>temporal</u> significance with t as the independant variable or by h for a formula having <u>spatial</u> significance with x as the independent variable)

<u>Example:</u>

Forward Euler is a formula of order one: the Local error is

O ($(\Delta t)^2$)

Forward Euler is a formula of order one: the Global error is O ((Δt))

Accuracy related to Stepsize and Order

Consider some formula for approximating the solution of an ODE; this formula has a certain order.

Select a stepsize. We will consider the relation between accuracy and choice of stepsize.

First we do that for a very simple example.

We consider a trivial ODE in which the derivative does not depend on the solution:

$$y'(x) = \frac{1}{x};\; y(1) = 0$$

The solution of this problem follows from calculating:

$$y = \int_1^x \frac{1}{u}\, du$$

For numerical comparisons we will calculate the solution of this almost trivial ODE using Forward Euler and follow the solution curve to x = 2 . The answer should be an approximation for Log[2] (the logarithm with base *e*, which is the natural logarithm).

Our initial choice for stepsize h will be h = 0.1 , which is equivalent to saying that we use 10 subintervals which define 9 equidistant gridpoints x1, x2, ..., x9 between x0 = 1 and x10 = 2 .

The computation is based on the next formula and runs in *Mathematica* as is shown :

$$y_{k+1} = y_k + h \frac{1}{x_k}.$$

In[57]:=

```
n = 10;
```

In[58]:=

```
h = 1.0 / n;
```

In[59]:=

```
x = 1; y = 0;
Do[( y = y + h / x; x = x + h; Print[x, "   ", y]), {n}];
Print["Error  ", y - Log[2]];
```

```
1.1   0.1
1.2   0.190909
1.3   0.274242
1.4   0.351166
1.5   0.422594
1.6   0.489261
1.7   0.551761
1.8   0.610584
1.9   0.66614
2.   0.718771
Error  0.0256242
```

In fact we are not that much interested in the intermediate values; for this test we only want to see the error in the final approximated function-value (≈ Log[2]) when using a doubled number of intervals and observe the effect on the error.

This gives rise to the following experiment:

In[62]:=

```
n = 10;
```

In[63]:=

```
Log[2.]
Do[
  (h = 1.0/n; x = 1; y = 0;

    Do[
      (y = y + h/x; x = x + h
      ), {n}
    ];

    Print["n = ", n, "  Error: ", y - Log[2]];
    n = n * 2
  ), {5}
]
```

Out[63]=

```
0.693147

n = 10  Error: 0.0256242

n = 20  Error: 0.0126562

n = 40  Error: 0.00628906

n = 80  Error: 0.00313477

n = 160  Error: 0.00156494
```

When looking at the error, we indeed recognize the effect that the error is halved whenever the number of intervals is doubled. An approximation having this property, we call an

O(h) approximation.

If the value to be approximated is called *I* and the approximation is called A(N) then we can state:

$$I = A(N) + O(h)$$

or, if the value to be approximated can be written in a Taylor series expansion, we have:

$$I = A(N) + C\,h + O(h^2)$$

Now for the following observation.

If *I* can indeed be approximated in a Taylor expansion, then we may look at the approximation over double the number of intervals, giving:

$$I = A(2N) + \frac{1}{2}\,C\,h + O(h^2)$$

Comparing the two results, we can eliminate the error-term of order O(h) (without knowing the value of the constant C !) by subtracting the first formula from twice the latter formula, giving:

$$I = 2\,A(2N) - A(N) + O(h^2)$$

In this way we have calculated an approximation that is an order of magnitude more accurate.

What follows is an application of this idea; we use the approximations over 80 and 160 terms.

In[65]:=

```
n = 80;  y2 = 0;
```

In[66]:=

```
Log[2.]
Do[
  (h = 1.0/n; x = 1; y = 0;

    Do[
      (y = y + h/x; x = x + h
      ), {n}
    ];
    y1 = y2; y2 = y;

    Print["n = ", n , "  Error: ", y - Log[2]]; n = n * 2
  ), {2}
]
```

Out[66]=

```
0.693147
```

```
n = 80  Error: 0.00313477
n = 160  Error: 0.00156494
```

In[68]:=

```
y2
y1 - Log[2]
y2 - Log[2]
```

Out[68]=

```
0.694712
```

Out[69]=

```
0.00313477
```

Out[70]=

```
0.00156494
```

In[71]:=

```
y3 = 2 y2 - y1
```

Out[71]=

0.693142

In[72]:=

```
y3 - Log[2]
```

Out[72]=

-4.88265×10^{-6}

How many subintervals do we need to calculate a first-order approximation that is equaly accurate. So assume that we would like to know **the number of subintervals** such that the **error is at most 10^{-5}** .

For n = 80 we see that the error equals 3.1×10^{-3}. Each time the number of intervals is doubled, the error is reduced by a factor 2. This gives the relation:

In each approximation (that is to say for each number of subintervals) we can improve the Forward Euler result to a result that has the accuracy of a **second order** approximating formula in a very simple way (because we deal with pure integration).

If we imagine the result of Backward Euler (which would be trivial to calculate in this situation) then we can understand that the average of Forw. Euler and Backw. Euler would be equal to using the trapezoidal rule for integration and it is well known that this formula is of order 2.

To derive this from the preceding program, we should substract half the value in x0 and add half the value in xn:

In[76]:=

```
n = 10;
```

In[77]:=

```
Log[2.]
Do[
  (h = 1.0/n; x = 1; y = 0;

     Do[
       (y = y + h/x; x = x + h
       ), {n}];
                                   y = y - h/2 + h/4;

     Print["n = ", n , "  Error: ", y - Log[2]]; n = n*2
  ), {5}
]
```

Out[77]=

0.693147

n = 10 Error: 0.000624223

n = 20 Error: 0.000156201

n = 40 Error: 0.0000390594

n = 80 Error: 9.76543×10^{-6}

n = 160 Error: 2.44139×10^{-6}

Observe that the error is divided by 4 whenever the number of intervals is doubled.

Moreover, observe that for n = 80 subintervals, the result has an error of at most 10^{-5} .

Our overall conclusion is that using a formula with a higher order has a much stronger effect on the accuracy then increasing the number of subintervals. It is obvious that the effect on the number of calculations is also dramatic.

This holds under the assumption that the derivative does not vary to much on the subinterval where the approximation formula is used.

For this improved approximation, we can apply the same idea of improving the accuracy in the approximation by comparing the approximation using N subintervals with the approximation using 2N subintervals. This works as follows.

Compare:

$$I = B(N) + D h^2 + O(h^3)$$

and:

$$I = B(2N) + \frac{1}{4} h^2 + O\left(h^3\right)$$

giving:

$$3I = 4B(2N) - B(N) + O\left(h^3\right)$$

Application of this idea gives:

In[79]:=

```
n = 80;  y2 = 0;
```

In[80]:=

```
Log[2.]
Do[
    (h = 1.0/n; x = 1; y = 0;

      Do[
        (y = y + h/x; x = x + h
        ), {n}
      ];
                                    y = y - h/2 + h/4;
      y1 = y2; y2 = y;
      Print["n = ", n , "  Error: ", y - Log[2]];
      n = n*2
    ), {2}
  ];
```

Out[80]=

0.693147

n = 80 Error: 9.76543×10^{-6}

n = 160 Error: 2.44139×10^{-6}

In[82]:=

```
y3 = (4 y2 - y1)/3
y3 - Log[2]
```

Out[82]=

0.693147

Out[83]=

4.76759×10^{-11}

One would expect that it might be applied even one level deeper:

In[84]:=

```
n = 40;  y1 = 0; y2 = 0;
```

In[85]:=

```
Log[2.]
Do[
     (h  =  1.0/n; x = 1; y  =  0;

          Do[
            (y  =  y + h/x; x = x + h
            ),  {n}
          ];
                                                 y  =  y  −  h/2  +  h/4;
          y0 = y1; y1 = y2; y2 = y;
          Print["n = ",  n , "  Error: ", y  −  Log[2]];
          n = n * 2
     ),  {3}
  ];
z1 = (4 y1 − y0)/3
z2 = (4 y2 − y1)/3
z1 − Log[2]
z2 − Log[2]
w1 = (8 z2 − z1)/7
w1 − Log[2]
```

Out[85]=

0.693147

n = 40 Error: 0.0000390594

n = 80 Error: 9.76543×10^{-6}

n = 160 Error: 2.44139×10^{-6}

Out[87]=

0.693147

Out[88]=

0.693147

Out[89]=

7.62643×10^{-10}

Out[90]=

4.76759×10^{-11}

Out[91]=

0.693147

Out[92]=

-5.44622×10^{-11}

Close all sections