

Euler's Method

```
> restart:with(plots):with(DEtools):
```

Euler's method is designed to approximate the solution to the initial value problem, $\frac{dy}{dt} = f(t, y)$, $y(a) = \alpha$, over a closed interval $x=a..b$.

We will restrict our work in this course to initial value problems where there is a unique solution curve through the initial point (a, α) . That is where we will start. With $t_0 = a$, we then pick N other equally spaced values t_1, t_2, \dots, t_N of the independent variable t with $t_N = b$ at which to approximate the solution. We will use h to represent the **step size**, with N the **number of steps**. Then $h = \frac{b - a}{N}$.

In general, Euler's method uses the recursion formulas below to move from one step to the next.

$$t_{i+1} = t_i + h \quad w_{i+1} = w_i + hf(t_i, w_i)$$

with w_i approximating $y(t_i)$.

As an example, we will consider the IVP from Problem 3a on page 190 of the text.

$$\frac{\partial}{\partial t} y = \frac{y}{t} - \left(\frac{y}{t}\right)^2, \quad t=1..2, \quad y(1) = 1, \quad \text{with } h = 0.1.$$

This value of h implies $N = 10$. We enter the IVP.

```
> deq:=diff(y(t),t) = y(t)/t-(y(t)/t)^2;
```

$$deq := \frac{d}{dt} y(t) = \frac{y(t)}{t} - \frac{y(t)^2}{t^2}$$

```
> IC:=y(1)=1;
```

$$IC := y(1) = 1$$

Let's see if we can find an exact solution here.

```
> soln:=dsolve({deq,IC},y(t));
```

$$soln := y(t) = \frac{t}{\ln(t) + 1}$$

We wish to express our solution as a function, using the [unapply](#) command.

```
> soln:=unapply(rhs(soln),t);
```

$$soln := t \rightarrow \frac{t}{\ln(t) + 1}$$

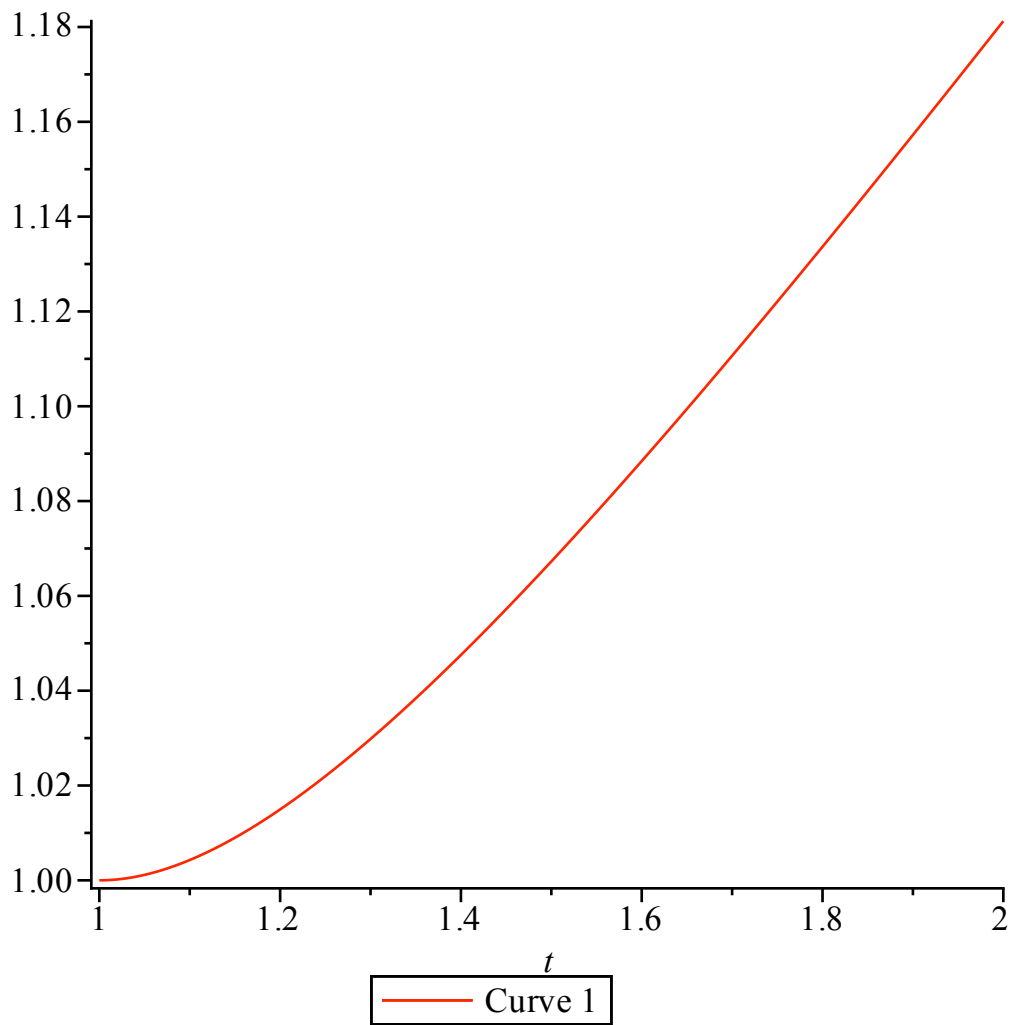
We draw a plot of this solution. When I assign a [plot](#) to a variable, the plot does not display. Instead, we get the Maple instructions for drawing the plot. Typically, we will suppress this output by ending the statement with a [colon](#).

```
> p[1]:=plot(soln(t),t=1..2);
```

$$p_1 := PLOT(...)$$

We view the plot.

```
> display(p[1]);
```



We create a loop to implement Euler's method.

```
> f:=(t,w)->(w/t)-(w/t)^2;
t[0]:=1;w[0]:=1;
for i from 0 to 9 do
t[i+1]:=t[i]+.1;
w[i+1]:=w[i]+.1*f(t[i],w[i]);
end do;
```

$$f := (t, w) \rightarrow \frac{w}{t} - \frac{w^2}{t^2}$$

$$t_0 := 1$$

$$w_0 := 1$$

$$t_1 := 1.1$$

$$w_1 := 1.$$

$$t_2 := 1.2$$

$$w_2 := 1.008264463$$

$$t_3 := 1.3$$

```

w3 := 1.021689472
      t4 := 1.4
w4 := 1.038514735
      t5 := 1.5
w5 := 1.057668193
      t6 := 1.6
w6 := 1.078461094
      t7 := 1.7
w7 := 1.100432165
      t8 := 1.8
w8 := 1.123262052
      t9 := 1.9
w9 := 1.146723597
      t10 := 2.0
w10 := 1.170651570

```

Now we use `dsolve` to implement **Euler's method** by setting the type to **numeric**, the method to **classical** [**foreuler**] (classical would be sufficient since `foreuler`, standing for forward Euler method, is the default), and providing a starting point for the independent variable and a stepsize.

```

> t:='t':
> eul:=dsolve({deq, IC}, type=numeric,
              method=classical[foreuler], start=1.0, stepsize=.1);
      eul := proc(x_classical) ... end proc

```

The output in this case is a procedure from which the data we want can be extracted. Suppose we simply wanted to find the data for when $x = 1.5$, which happens to be one of our mesh points.

```

> eul(1.5);
      [t = 1.5, y(t) = 1.05766819214087637]

```

We can do the following to create an ordered pair representing a single data point.

```

> eval([t, y(t)], eul(1.5));
      [1.5000000000000000, 1.05766819214087637]

```

We can also simply extract the y value at a data point.

```

> eval(y(t), eul(1.5));
      1.05766819214087637

```

Another method for extracting the same data point.

```

> rhs(eul(1.5)[2]);
      1.05766819214087637

```

We can do the following to see the whole set of data.

```

> for i from 0 to 10 do
      eval([t, y(t)], eul(1+.1*i));
od;

```

```
[1., 1.]
[1.1, 1.]
[1.2, 1.00826446280991733]
[1.3, 1.02168947172703750]
[1.4, 1.03851473424817797]
[1.5, 1.05766819214087637]
[1.6, 1.07846109363175469]
[1.7, 1.10043216469946592]
[1.8, 1.12326205158126324]
[1.9, 1.14672359652952638]
[2.0, 1.17065156956466465]
```

Suppose we ask for a value, such as $x = 1.24$, which falls between the computed points. Maple uses linear interpolation, that is, it follows the line segment joining the nearest two computed points.

```
> eul(1.24);
[t = 1.24, y(t) = 1.01363446637676535]
```

We note that doing linear interpolation by hand gives the same answer.

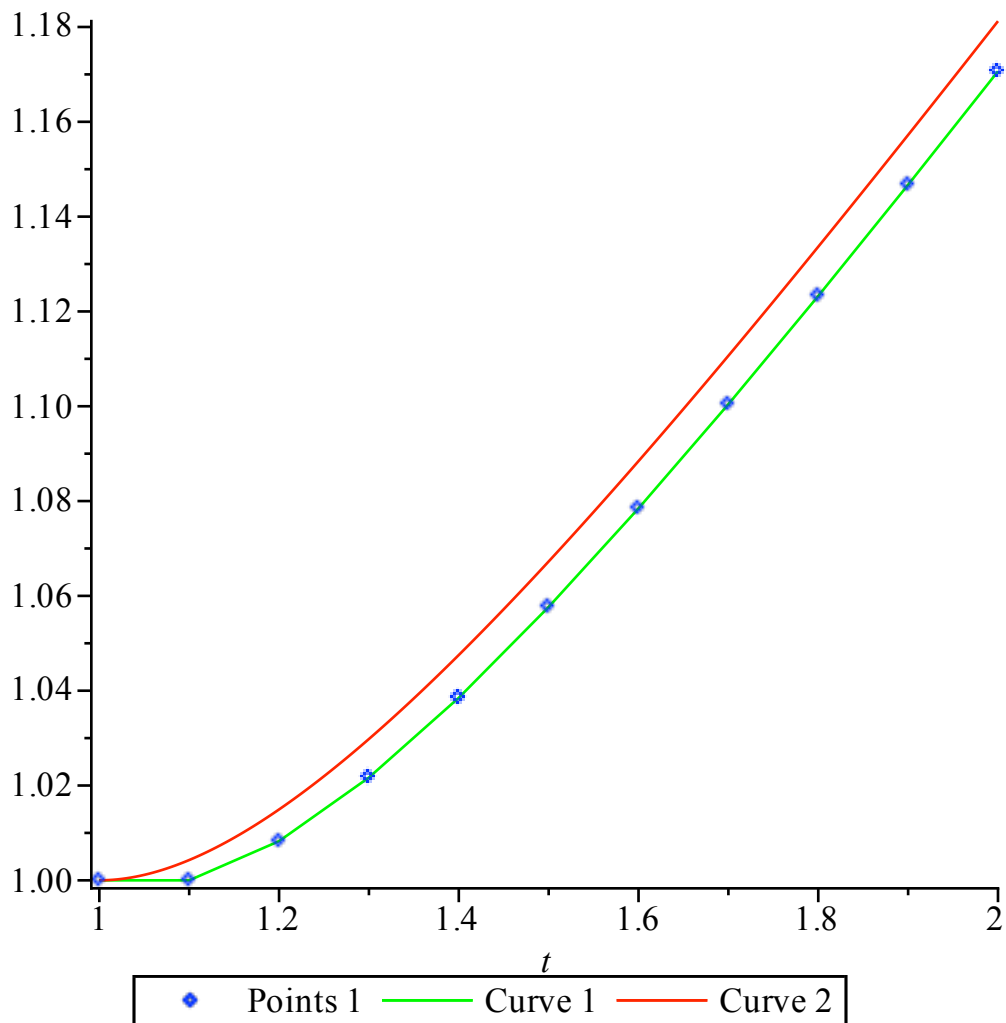
```
> eval(y(t), eul(1.2)) + .4 * (eval(y(t), eul(1.3)) - eval(y(t), eul(1.2)));
1.013634467
```

We will use [odeplot](#) from the [plots](#) package to create first a point plot and then the line plot to connect the points.

```
> p[2] := pointplot([seq(eval([t, y(t)], eul(1+.1*i)), i=0..10)], style=
point, color=blue);
p[3] := pointplot([seq(eval([t, y(t)], eul(1+.1*i)), i=0..10)], style=
line, color=green);
```

To show one or more plots simultaneously that have been assigned to variables, we use the [display](#) command.

```
> display({p[1], p[2], p[3]});
```



We can see the error involved in the process from the graph. Let's repeat the whole process for a stepsize of .05, giving 20 steps.

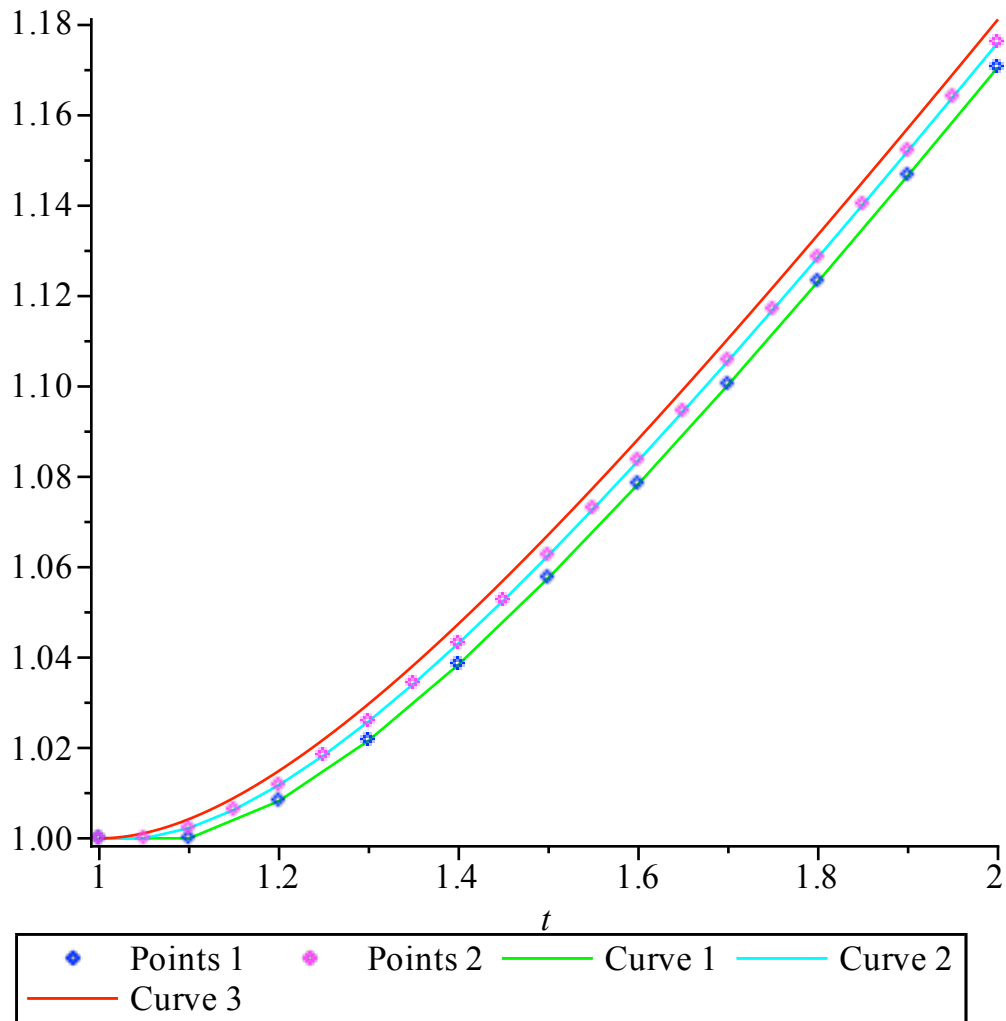
```
> eul2:=dsolve({deq, IC}, type=numeric,
               method=classical[foreuler], start=1.0, stepsize=.05);
               eul2 := proc(x_classical) ... end proc
```

We create the plots for the new step size.

```
> p[4]:=pointplot([seq(eval([t,y(t)],eul2(1+.05*i)),i=0..20)],style=
point,color=magenta):
p[5]:=pointplot([seq(eval([t,y(t)],eul2(1+.05*i)),i=0..20)],style=
line,color=cyan):
```

We view all graphs simultaneously.

```
> display({seq(p[i],i=1..5)});
```



It appears that the smaller stepsize gives a better approximation.

Finally, let's make a comparative table of our results.

```
> printf("t[i]  y[i]                w[i](h=.1)    y[i]-w[i]    w2[i](h=.05)  y[i]-w2[i]\n");
printf("\n");
for i from 0 to 10 do
printf("%4.1f  %12.9f  %12.9f  %12.9f  %12.9f  %12.9f\n",eval(t,eul(1+.1*i)),soln(eval(t,eul(1+.1*i))),eval(y(t),eul(1+.1*i)),soln(eval(t,eul(1+.1*i)))-eval(y(t),eul(1+.1*i)),eval(y(t),eul2(1+.1*i)),soln(eval(t,eul(1+.1*i)))-eval(y(t),eul2(1+.1*i)));
od;
t[i]  y[i]                w[i](h=.1)    y[i]-w[i]    w2[i](h=.05)  y[i]-w2[i]
1.0   1.000000000        1.000000000    0.000000000    1.000000000
0.000000000
1.1   1.004281728        1.000000000    0.004281728    1.002267574
0.002014154
1.2   1.014952314        1.008264463    0.006687851    1.011781883
0.003170431
1.3   1.029813689        1.021689472    0.008124217    1.025941944
0.003871745
1.4   1.047533919        1.038514734    0.009019185    1.043219551
0.004314368
```

```

1.5    1.067262354    1.057668192    0.009594162    1.062660432
0.004601922
1.6    1.088432687    1.078461094    0.009971593    1.083640054
0.004792633
1.7    1.110655052    1.100432165    0.010222887    1.105734163
0.004920889
1.8    1.133653557    1.123262052    0.010391505    1.128645717
0.005007840
1.9    1.157228433    1.146723597    0.010504836    1.152161503
0.005066930
2.0    1.181232218    1.170651570    0.010580648    1.176125268
0.005106950

```

It is clear that the smaller step size results in less global error. A scanning of the results also shows that the global error accumulates linearly.

Let's see how our results compare with the error bound in the text:

$$|y(t_i) - w_i| \leq \frac{hM}{2L} \{e^{L(t_i - a)} - 1\}.$$

We have values for h and a .

```
> h:=.1;a:=1;
```

```
h := 0.1
```

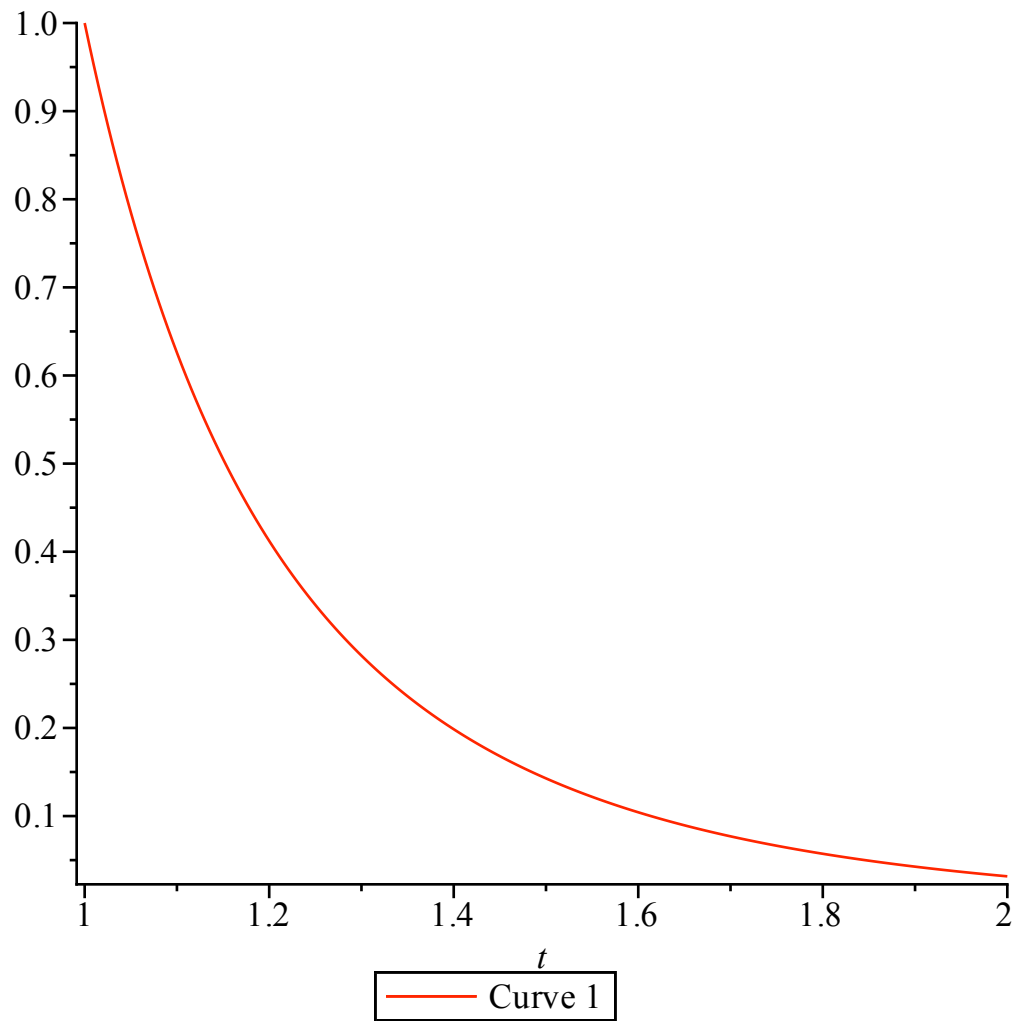
```
a := 1
```

M is a bound for $|y''(t)|$ on $t=1..2$.

```
> y2:=(D@@2)(soln);
```

$$y2 := t \rightarrow -\frac{1}{(\ln(t) + 1)^2 t} + \frac{2}{(\ln(t) + 1)^3 t}$$

```
> plot(y2(t),t=1..2);
```



```
> M:=1;
```

$$M := 1$$

L is the Lipschitz constant, where $|\frac{\partial}{\partial y} f| \leq L$ on $t=1..2$.

```
> f:=subs(y(t)=y,rhs(deq));
```

$$f := \frac{y}{t} - \frac{y^2}{t^2}$$

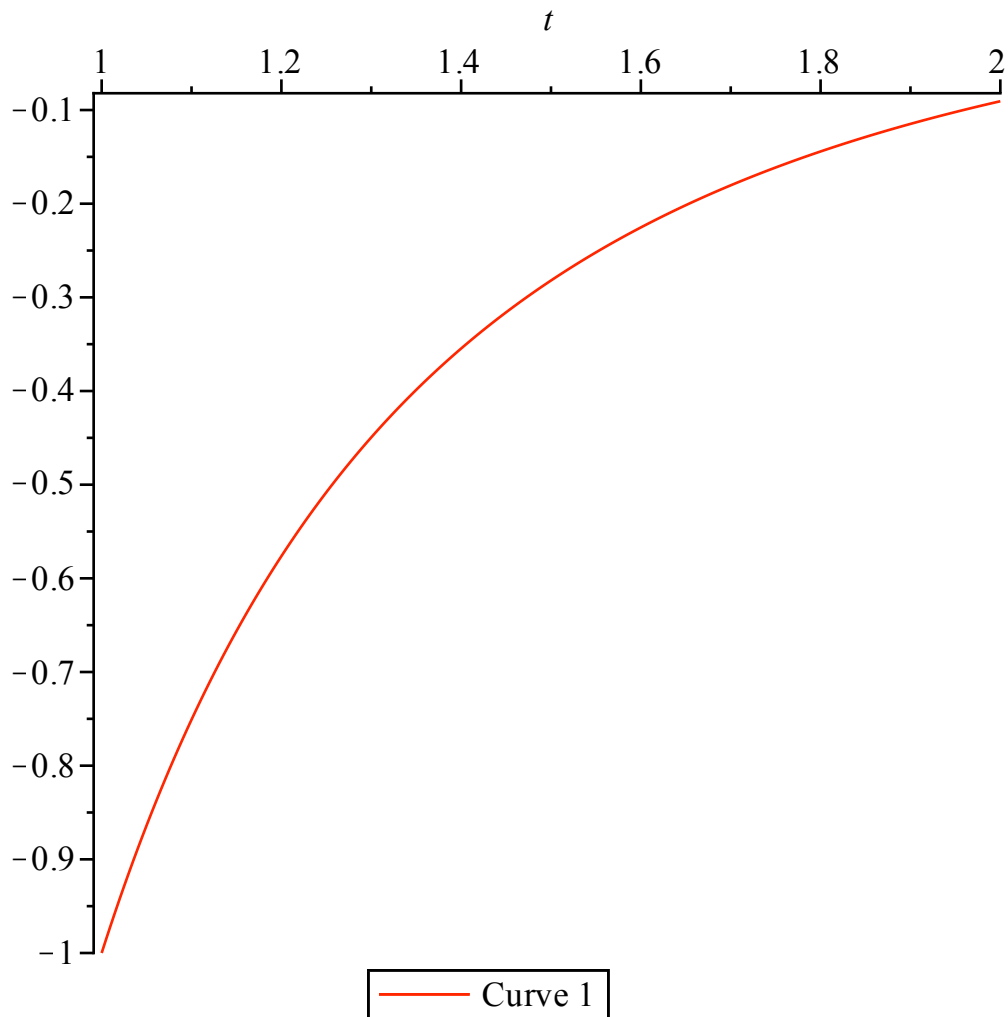
```
> fy:=diff(f,y);
```

$$fy := \frac{1}{t} - \frac{2y}{t^2}$$

```
> fy:=subs(y=soln(t),fy);
```

$$fy := \frac{1}{t} - \frac{2}{t(\ln(t) + 1)}$$

```
> plot(fy,t=1..2);
```



```
> L:=1;
```

```
L := 1
```

We write the bound as a function of the index i .

```
> bound:=i->(h*M)/(2*L)*(exp(L*(1+.1*i-a))-1);
```

$$\text{bound} := i \rightarrow \frac{1}{2} \frac{h M (e^{L(1+0.1i-a)} - 1)}{L}$$

The following table compares the actual accumulated or global error at each step with the error bound.

```
> printf("t[i]  y[i]-w[i]      bound[i]\n");
printf("\n");
for i from 0 to 10 do
printf("%4.1f  %12.9f  %12.9f\n",eval(t,eul(1+.1*i)),soln(eval(t,
eul(1+.1*i)))-eval(y(t),eul(1+.1*i)),bound(i));
od;
```

t[i]	y[i]-w[i]	bound[i]
1.0	0.000000000	0.000000000
1.1	0.004281728	0.005258546
1.2	0.006687851	0.011070138
1.3	0.008124217	0.017492940
1.4	0.009019185	0.024591235
1.5	0.009594162	0.032436064
1.6	0.009971593	0.041105940

1.7	0.010222887	0.050687635
1.8	0.010391505	0.061277046
1.9	0.010504836	0.072980156
2.0	0.010580648	0.085914091

Our error clearly stays within the given bounds.

Numerical Analysis

```
> with(Student[NumericalAnalysis]);
```

```
[AbsoluteError, AdamsBashforth, AdamsBashforthMoulton, AdamsMoulton, AdaptiveQuadrature,
AddPoint, ApproximateExactUpperBound, ApproximateValue, BackSubstitution, BasisFunctions,
Bisection, CubicSpline, DataPoints, Distance, DividedDifferenceTable, Draw, Euler, EulerTutor,
ExactValue, FalsePosition, FixedPointIteration, ForwardSubstitution, Function,
InitialValueProblem, InitialValueProblemTutor, Interpolant, InterpolantRemainderTerm,
IsConvergent, IsMatrixShape, IterativeApproximate, IterativeFormula, IterativeFormulaTutor,
LeadingPrincipalSubmatrix, LinearSolve, LinearSystem, MatrixConvergence,
MatrixDecomposition, MatrixDecompositionTutor, ModifiedNewton, NevilleTable, Newton,
NumberOfSignificantDigits, PolynomialInterpolation, Quadrature, RateOfConvergence,
RelativeError, RemainderTerm, Roots, RungeKutta, Secant, SpectralRadius, Steffensen, Taylor,
TaylorPolynomial, UpperBoundOfRemainderTerm, VectorLimit]
```

The basic command in this package for solving initial value problems is [InitialValueProblem](#). For the **Euler** method, this command would be used with **method=euler**. Or one could simply use the command [Euler](#). We need to reset t since it was changed to an array above. We use $t = 2$ to indicate our target point, **numsteps=10** to indicate the number of steps to take to our target (the default is 5), **digits=9** to indicate the number of digits returned values will be rounded to (the default is 4). Also, **output=** can be followed by any of **solution**, **Error**, **plot**, or **information**.

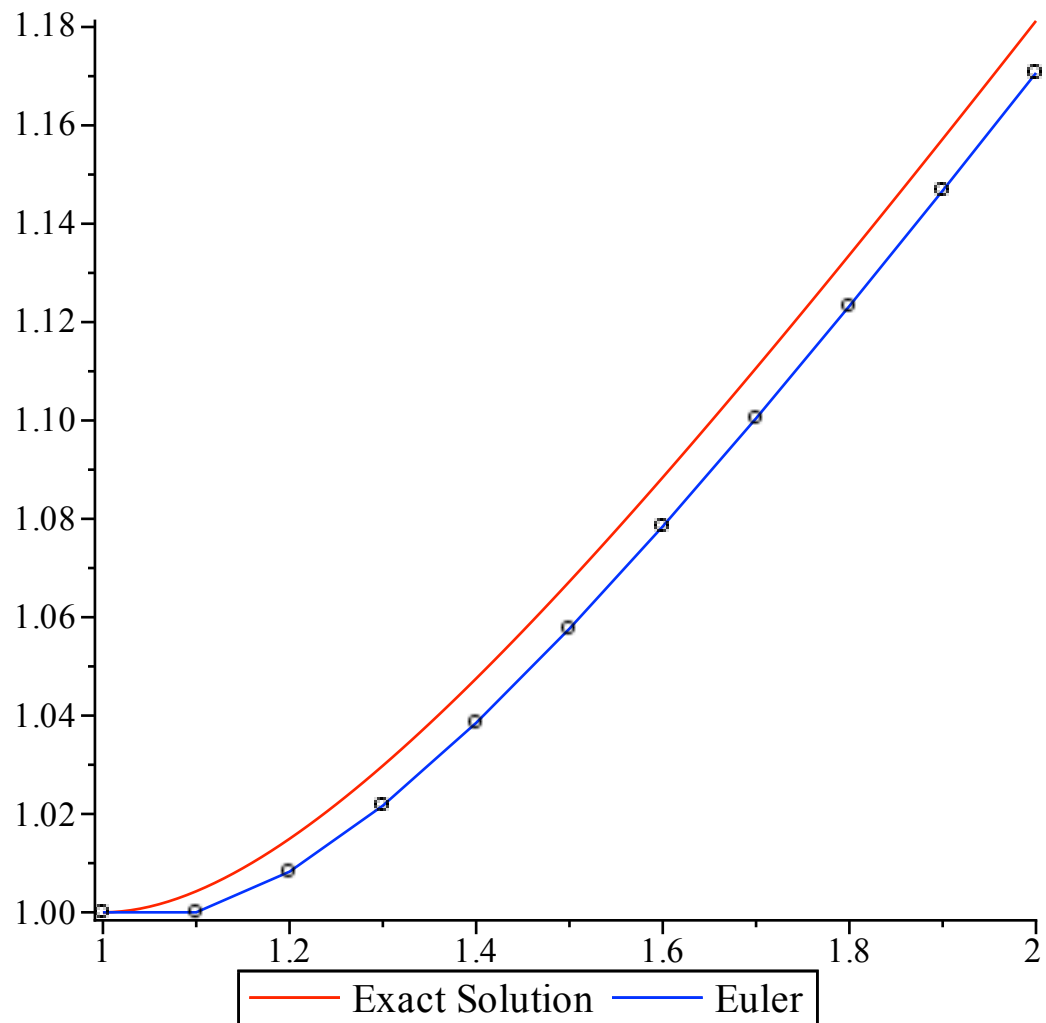
```
> t := `t`;
```

```
t := t
```

```
> Euler(deq, IC, t=2, numsteps=10, digits=10, output=solution);
1.170651570
```

```
> Euler(deq, IC, t=2, numsteps=10, digits=10, output=Error);
0.010580648
```

```
> Euler(deq, IC, t=2, numsteps=10, digits=10, output=plot);
```



```
> Euler(deq,IC,t=2,numsteps=10,digits=10,output=information);
```

```

1..12 x 1..4 Array
Data Type: anything
Storage: rectangular
Order: Fortran_order

```

If you get something like this, use the following command and then try again.

```
> interface(rtablesize=100);
```

```
10
```

```
> Euler(deq,IC,t=2,numsteps=10,digits=10,output=information);
```

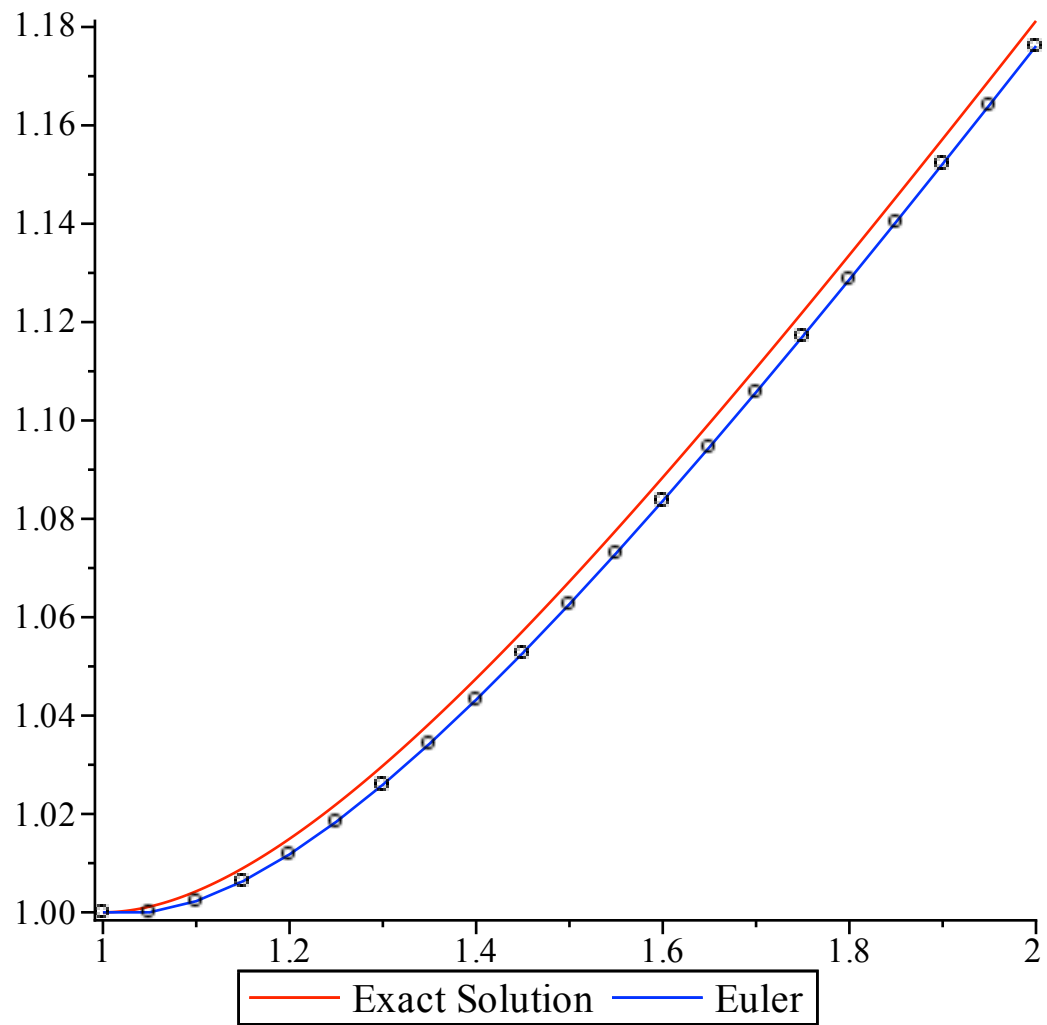
t	$y(t)$	[Euler]	[Error]
1.	1.	1.	0.
1.100000000	1.004281728	1.	0.004281728
1.200000000	1.014952314	1.008264463	0.006687851
1.300000000	1.029813689	1.021689472	0.008124217
1.400000000	1.047533919	1.038514734	0.009019185
1.500000000	1.067262354	1.057668192	0.009594162
1.600000000	1.088432687	1.078461094	0.009971593
1.700000000	1.110655052	1.100432165	0.010222887
1.800000000	1.133653557	1.123262052	0.010391505
1.900000000	1.157228433	1.146723597	0.010504836
2.	1.181232218	1.170651570	0.010580648

Now let's go to 20 steps, with $h = 0.05$.

```
> Euler(deq, IC, t=2, numsteps=20, digits=10, output=solution);
1.176125268
```

```
> Euler(deq, IC, t=2, numsteps=20, digits=10, output=Error);
0.005106950
```

```
> Euler(deq, IC, t=2, numsteps=20, digits=10, output=plot);
```



```
> Euler(deq,IC,t=2,numsteps=20,digits=10,output=information);
```

t	$y(t)$	[<i>Euler</i>]	[<i>Error</i>]
1.	1.	1.	0.
1.050000000	1.001153554	1.	0.001153554
1.100000000	1.004281728	1.002267574	0.002014154
1.150000000	1.008982628	1.006315261	0.002667367
1.200000000	1.014952314	1.011781883	0.003170431
1.250000000	1.021956907	1.018394233	0.003562674
1.300000000	1.029813689	1.025941944	0.003871745
1.350000000	1.038377995	1.034260514	0.004117481
1.400000000	1.047533919	1.043219551	0.004314368
1.450000000	1.057187611	1.052714457	0.004473154
1.500000000	1.067262354	1.062660432	0.004601922
1.550000000	1.077694897	1.072988064	0.004706833
1.600000000	1.088432687	1.083640054	0.004792633
1.650000000	1.099431749	1.094568732	0.004863017
1.700000000	1.110655052	1.105734163	0.004920889
1.750000000	1.122071226	1.117102676	0.004968550
1.800000000	1.133653557	1.128645717	0.005007840
1.850000000	1.145379178	1.140338945	0.005040233
1.900000000	1.157228433	1.152161503	0.005066930
1.950000000	1.169184349	1.164095447	0.005088902
2.	1.181232218	1.176125268	0.005106950