Euler's Method - The Maple Way

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

We begin again by entering the initial value problem.

```
> deq:=diff(y(x),x) = 5/sqrt(x+25);
  IC:=y(0)=0;
```

$$deq := \frac{d}{dx} y(x) = \frac{5}{\sqrt{x+25}}$$

$$IC := y(0) = 0$$

We solve the equation, change the right-hand side to a function using unapply, and create the plot structure for our domain. Here we choose not to initialize the array structures.

```
> Y:=dsolve({deq,IC},y(x));
  Y:=unapply(rhs(Y),x);
  p[1]:=plot(Y(x),x=0..700):
```

$$Y := y(x) = 10\sqrt{x+25} - 50$$

$$Y := x \rightarrow 10\sqrt{x+25} - 50$$

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.

```
> Y100:=dsolve({deq, IC}, y(x), type=numeric,
          method=classical[foreuler], start=0.0,stepsize=100);
```

$$Y100 := \mathbf{proc}(x\_classical) \ \ldots \ \mathbf{end\ proc}$$

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 $x = 200$.

```
> Y100(200);
```

$$[x = 200., y(x) = 144.721359549996]$$

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

```
> eval([x,y(x)],Y100(200));
```

$$[200., 144.721359549996]$$

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

```
> eval(y(x),Y100(200));
```

$$144.721359549996$$

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

```
> for i from 0 to 7 do
       Y100(i*100);
  od;
```

$$[x = 0., y(x) = 0.]$$

$$[x = 100., y(x) = 100.]$$

$$[x = 200., y(x) = 144.721359549996]$$

$$[x = 300., y(x) = 178.054692883329]$$

$$[x = 400., y(x) = 205.789702694591]$$
$$[x = 500., y(x) = 230.043265198224]$$
$$[x = 600., y(x) = 251.865054221823]$$
$$[x = 700., y(x) = 271.865054221823]$$

We can also choose to see this as a listing of data points.

```
> for i from 0 to 7 do
      eval([x,y(x)],Y100(100*i));
  od;
```

$$[0., 0.]$$
$$[100., 100.]$$
$$[200., 144.721359549996]$$
$$[300., 178.054692883329]$$
$$[400., 205.789702694591]$$
$$[500., 230.043265198224]$$
$$[600., 251.865054221823]$$
$$[700., 271.865054221823]$$

Suppose we ask for a value, such as $x = 250$, which falls between the computed points. Maple uses linear interpolation, that is, it follows the line segment joining the nearest two computed points.

```
> Y100(250);
```

$$[x = 250., y(x) = 161.388026216662]$$

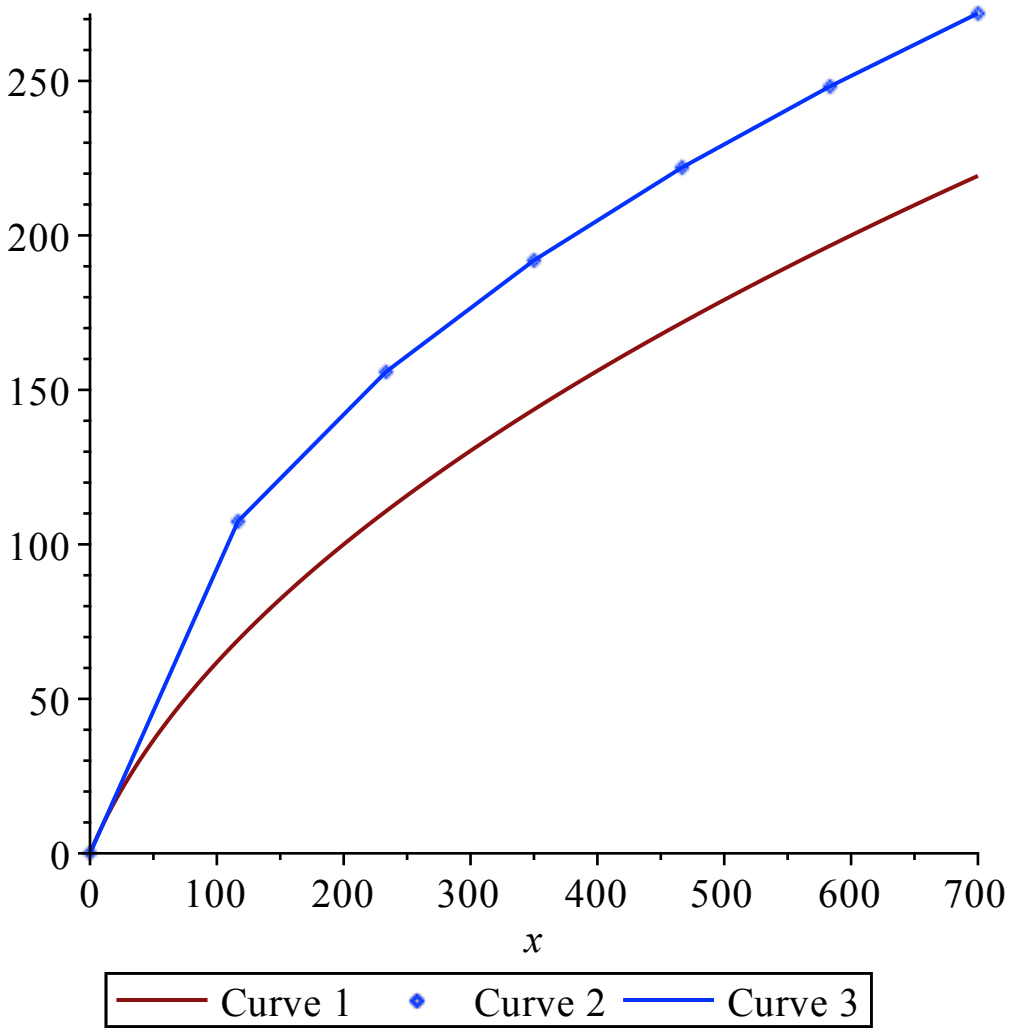
We can again choose to see this simply as a data point.

```
> eval([x,y(x)],Y100(250));
```

$$[250., 161.388026216662]$$

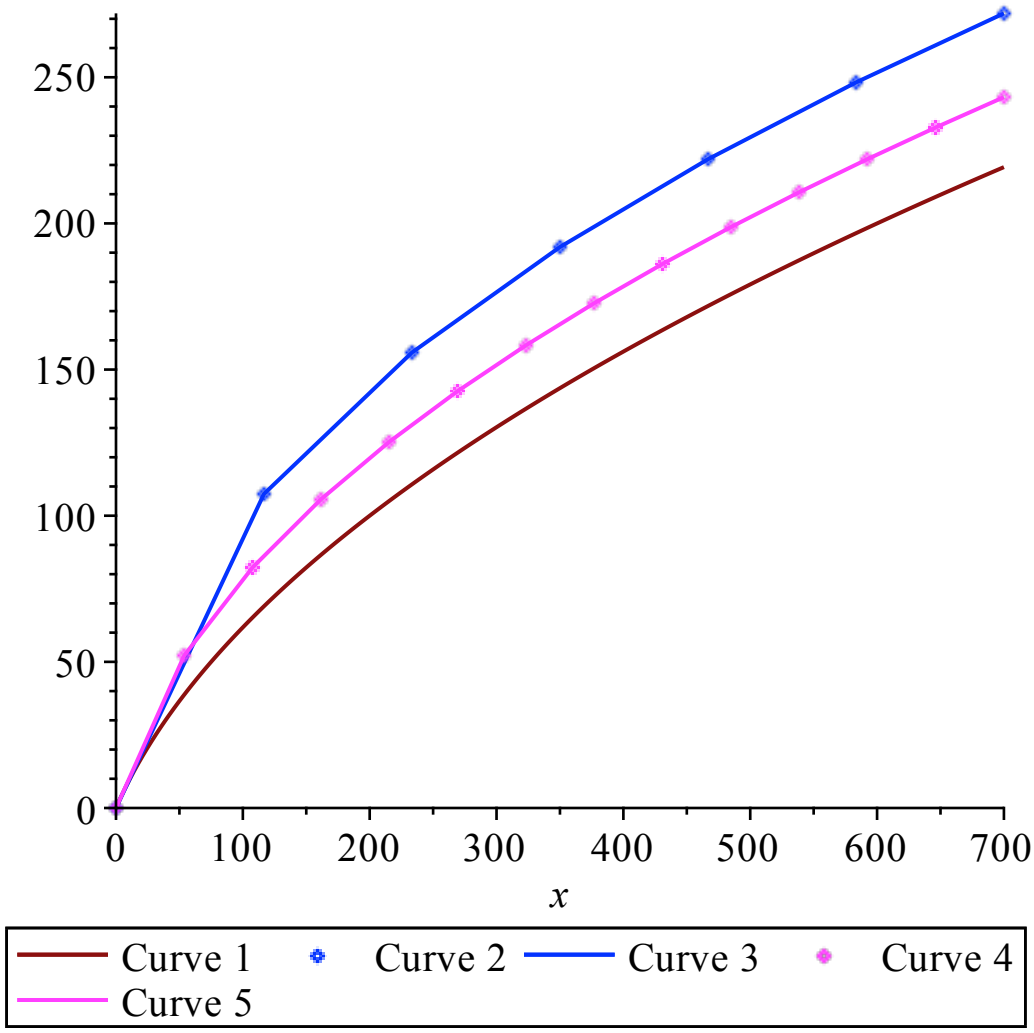
We will use odeplot from the plots package to create first the point plot and then the line plot to connect the points.

```
> p[2]:=odeplot(Y100,[x,y(x)],0..700, numpoints=7, style=point,color=
  blue):
  p[3]:=odeplot(Y100,[x,y(x)],0..700, numpoints=7, style=line,color=
  blue):
  display(seq(p[n],n=1..3));
```



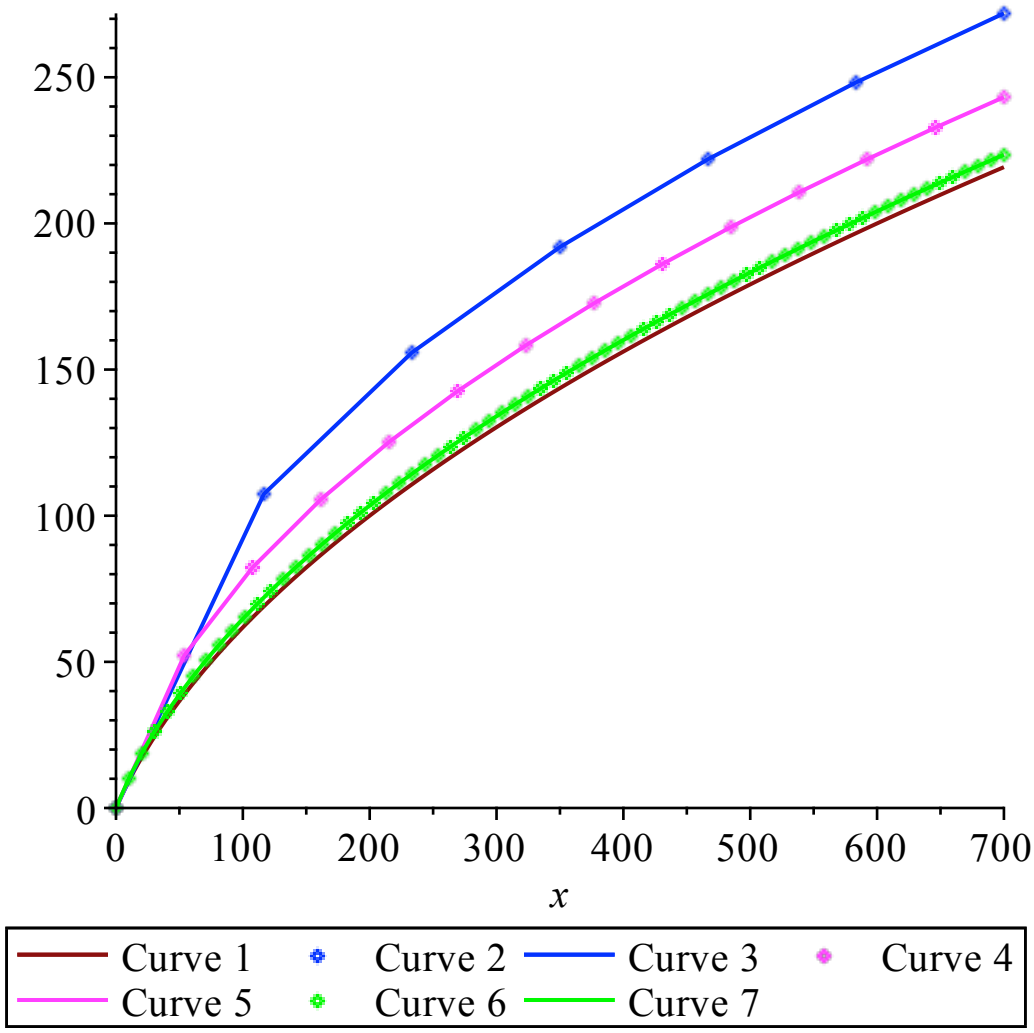
Let's repeat the whole process for a stepsize of 50, giving 14 steps.

```
> Y50:=dsolve({deq, IC}, y(x), type=numeric,
          method=classical[foreuler], start=0.0,stepsize=50):
  p[4]:=odeplot(Y50,[x,y(x)],0..700, numpoints=14, style=point,color=
  magenta):
  p[5]:=odeplot(Y50,[x,y(x)],0..700, numpoints=14, style=line,color=
  magenta):
  display(seq(p[n],n=1..5));
```



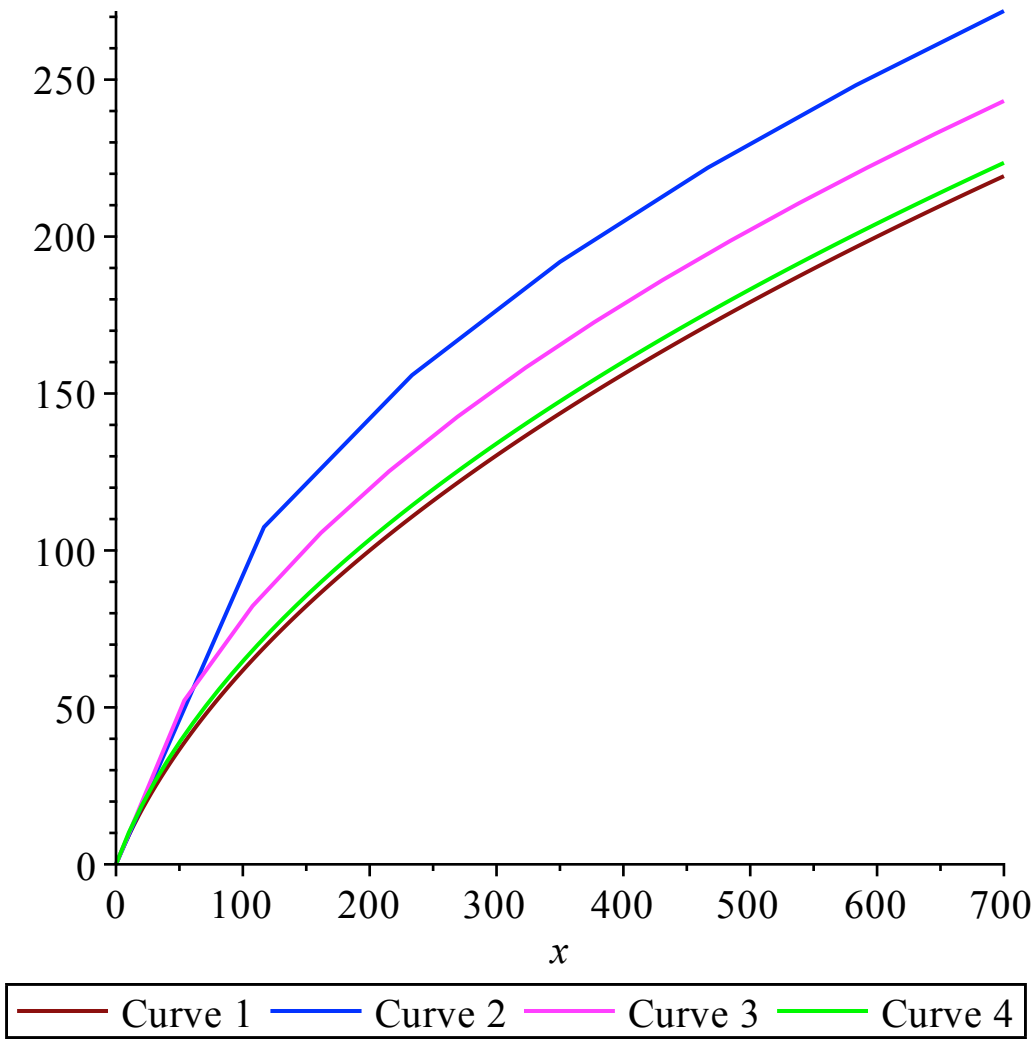
Finally, let's repeat it one more time with a stepsize of 10 for 70 steps.

```
> Y10:=dsolve({deq, IC}, y(x), type=numeric,
          method=classical[foreuler], start=0.0,stepsize=10):
  p[6]:=odeplot(Y10,[x,y(x)],0..700, numpoints=70, style=point,color=
  green):
  p[7]:=odeplot(Y10,[x,y(x)],0..700, numpoints=70, style=line,color=
  green):
  display(seq(p[n],n=1..7));
```



It seems clear that the smaller the stepsize, the more accuracy one gets. However, even our smallest stepsize here is still quite large. If one does not include the stepsize command, Maple uses the minimum of $h = 0.005$ and $h = \frac{b - x_0}{3}$ where $x = x_0 \,..\, b$. Let's look at the line display without the points.

```
> display(p[1],p[3],p[5],p[7]);
```



Finally, let's print a table of our values. In the table, *y100*, for instance, refers to the *y*-values from a step size of 100, while *y* refers to the exact function values.

```
> printf(" n    x[n]       y100(x[n])       y50(x[n])        y10(x[n]
  )          y(x[n])");
  printf("\n\n");
  for n from 0 to 7 do
  printf("%2d    %4.0f    %12.4f     %12.4f     %12.4f     %12.4f
  \n",n,eval(x,Y100(100*n)),eval(y(x),Y100(100*n)),eval(y(x),Y50(100*
  n)),eval(y(x),Y10(100*n)),evalf(Y(100*n)));
  od;
 n    x[n]       y100(x[n])       y50(x[n])        y10(x[n])          y
(x[n])

 0       0          0.0000           0.0000           0.0000
0.0000
 1     100        100.0000          78.8675          64.7175
61.8034
 2     200        144.7214         120.1264         103.4922
100.0000
 3     300        178.0547         151.8687         134.0523
```

```
130.2776
 4     400         205.7897          178.6461          160.1053
156.1553
 5     500         230.0433          202.2437          183.2011
179.1288
 6     600         251.8651          223.5803          204.1638
200.0000
 7     700         271.8651          243.2028          223.4938
219.2582
```

In Section 1.3, we used DEplot to draw direction fields along with solutions. Those solutions were not the exact solutions through a point, but used a numeric method called the fourth order classical Runge-Kutta method as a default. We can choose other methods such as the Euler method. It is also good to know that the default stepsize is $h = \frac{b-a}{20}$ where x = a .. b. Let's compare the actual solution (red) with both Euler's method (green) and the Runga-Kutta method (blue).

```
> YY10:=dsolve({deq, IC}, y(x), type=numeric,
          method=classical[rk4], start=0.0,stepsize=10):
  p[8]:=odeplot(YY10,[x,y(x)],0..700, numpoints=70, style=line,color=
  blue):
  display(p[1],p[7],p[8]);
```