

Euler's Method

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

From algebra we recall that there are precise methods to solve linear and quadratic equations. There are even formulas for cubic and quartic equations. However, we cannot find an exact solution to every algebraic equation. From Calculus, we recall that expressing an integral does not guarantee a closed form solution.

In each case we can use either numerical methods (for example, Newton's method or Simpson's Rule, respectively) or graphical methods to approximate solutions.

Similarly, a large store of techniques to solve differential equations does not guarantee an appropriate technique for every differential equation. However, there are several numerical methods to approximate solutions. In Chapter 3, Maple used numerical methods whenever we used the command [DEplot](#). A large number of numerical methods are built into Maple. In this course we shall consider only Euler's (forward) method, the simplest (and least accurate) method and leave the more advanced methods to numerical analysis.

>

Euler's method is designed to approximate the solution to the initial value problem, $\frac{dy}{dx} = f(x, y)$, $y(x_0) = y_0$, over a closed interval $x = x_0 .. b$.

We will restrict our work in this course to initial value problems where there is a unique solution curve through the initial point (x_0, y_0) . That is where we will start. We then pick N other equally spaced values x_1, x_2, \dots, x_N of the independent variable x with $x_N = b$ at which to approximate the solution and then connect those approximations with line segments. We will use h to represent the **step size**, with N the **number of steps**. Then $h = \frac{b - x_0}{N}$.

The First Step

>

Returning to the situation of the **Path of a Powerboat with Engine On** from the previous chapter, we have the initial value problem

$$\frac{dy}{dx} = \frac{5}{\sqrt{x+25}}, \quad x = 0 .. 700, \quad y(0) = 0.$$

Recall that we called this differential equation **motorboat**.

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

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

We enter the initial condition.

> **IC:=y(0)=0;**

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

We find the exact analytic solution.

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

$$\text{soln} := y(x) = 10\sqrt{x+25} - 10\sqrt{25}$$

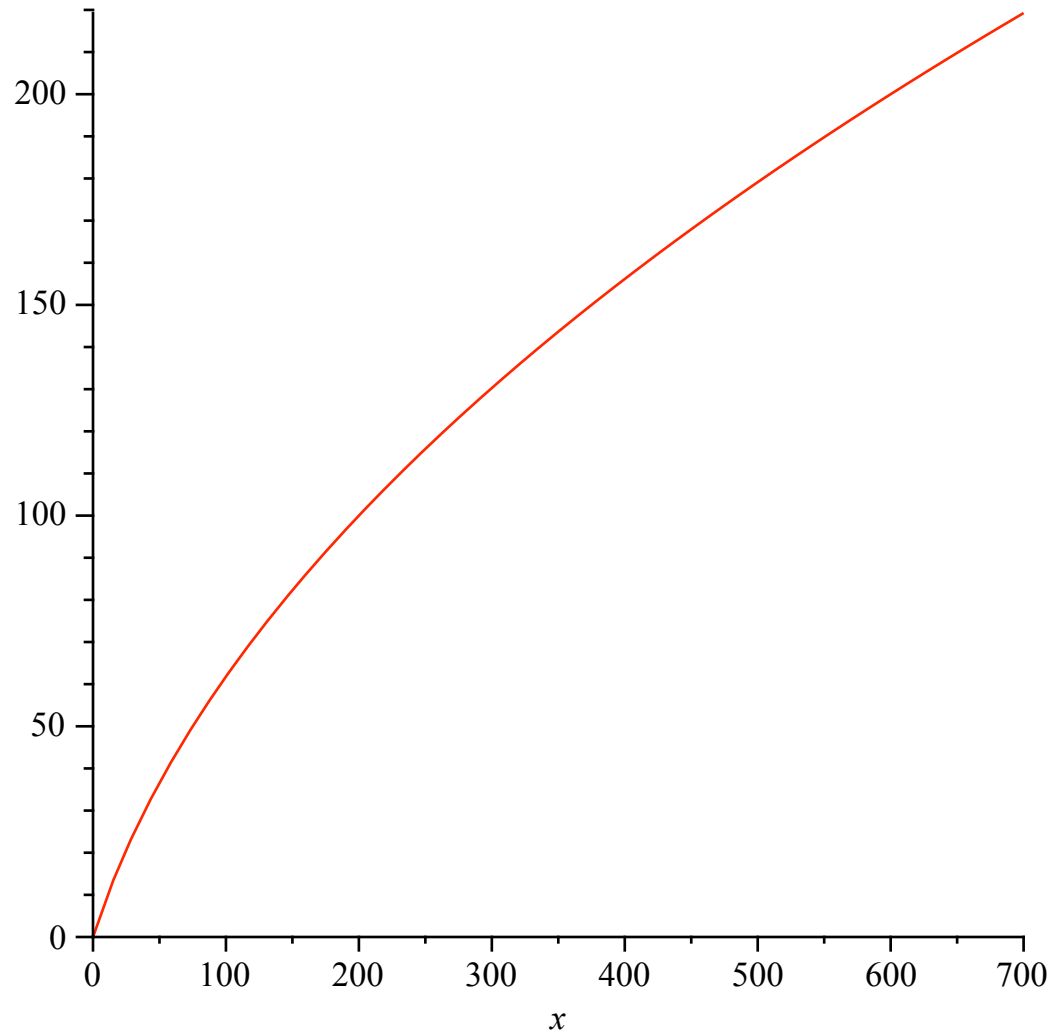
We draw a plot of this solution. When I assign a `plot` to a variable, the plot does not display. Instead, we get the Maple instructions for drawing the plot. Typically, we will suppress this output by ending the statement with a `colon`.

```
> p1:=plot(rhs(soln), x=0..700);
```

p1 := PLOT(...)

To show one or more plots simultaneously that have been assigned to variables, we use the `display` command.

```
> display(p1);
```



Suppose for the moment that we do not know the exact solution. However, we do know that the initial point of the solution is $(x_0, y_0) = (0, 0)$. We decide to use 7 steps to reach 700, so $h = \frac{700 - 0}{7} = 100$. Thus the first step is to approximate the "unknown" point $(x_1, y(x_1)) = (100, y(100))$ by a point $(x_1, y_1) = (100, y_1)$. How should we go about finding y_1 ?

```
>
```

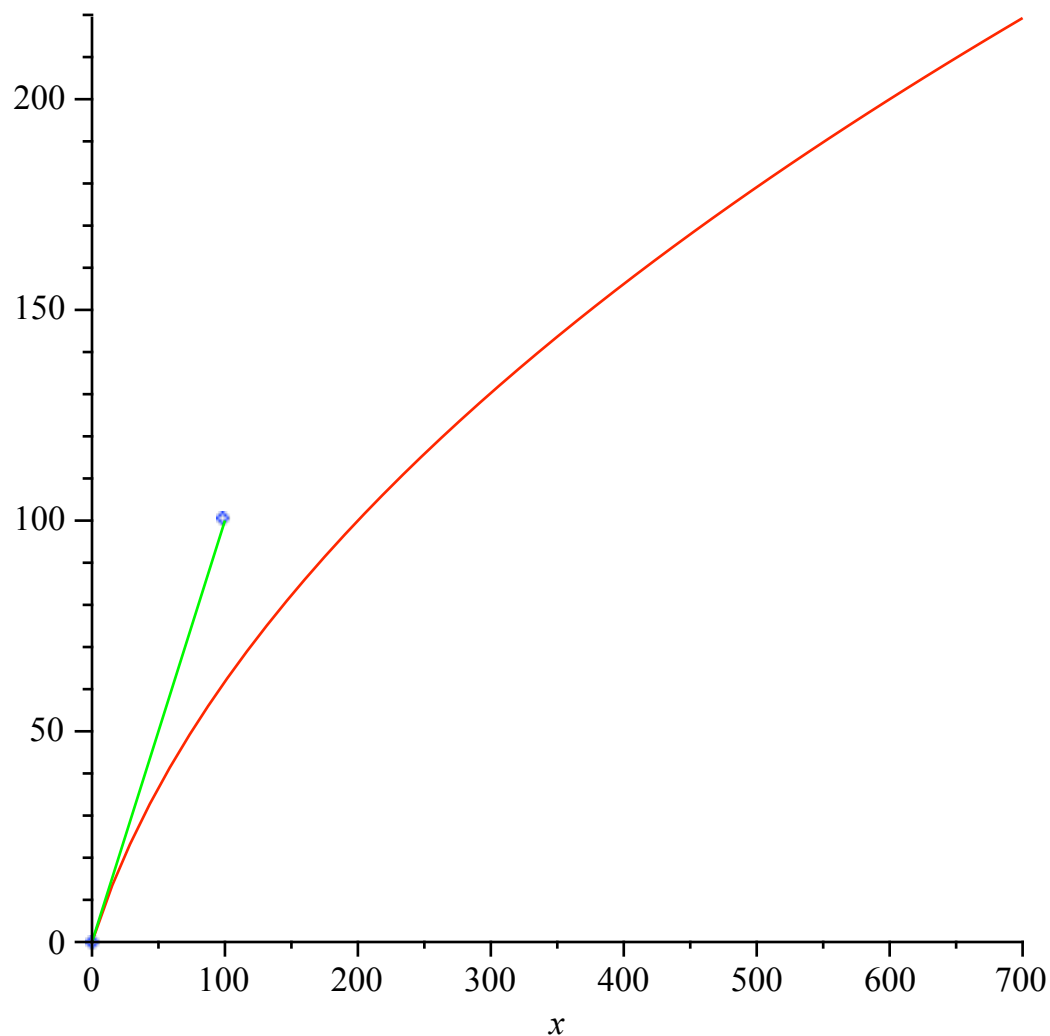
Besides the initial point $(0, 0)$, the only other information we have available to us is the differential equation, so we use that. We can use the right-hand side of the differential equation,

$f(x, y) = \frac{5}{\sqrt{x+25}}$, to find the slope of the tangent line to the solution curve going through any point (x, y) . In the direction fields of Chapter 3, this is the slope of the arrow whose base is at the point (x, y) . This slope will determine our direction of movement. We have $f(0, 0) = 1$, so we follow the line with slope 1 through the point $(0, 0)$ until we get to $x_1 = 100$. The slope-intercept form of this line is

$$y - 0 = 1(x - 0) \quad \text{or} \quad y = x,$$

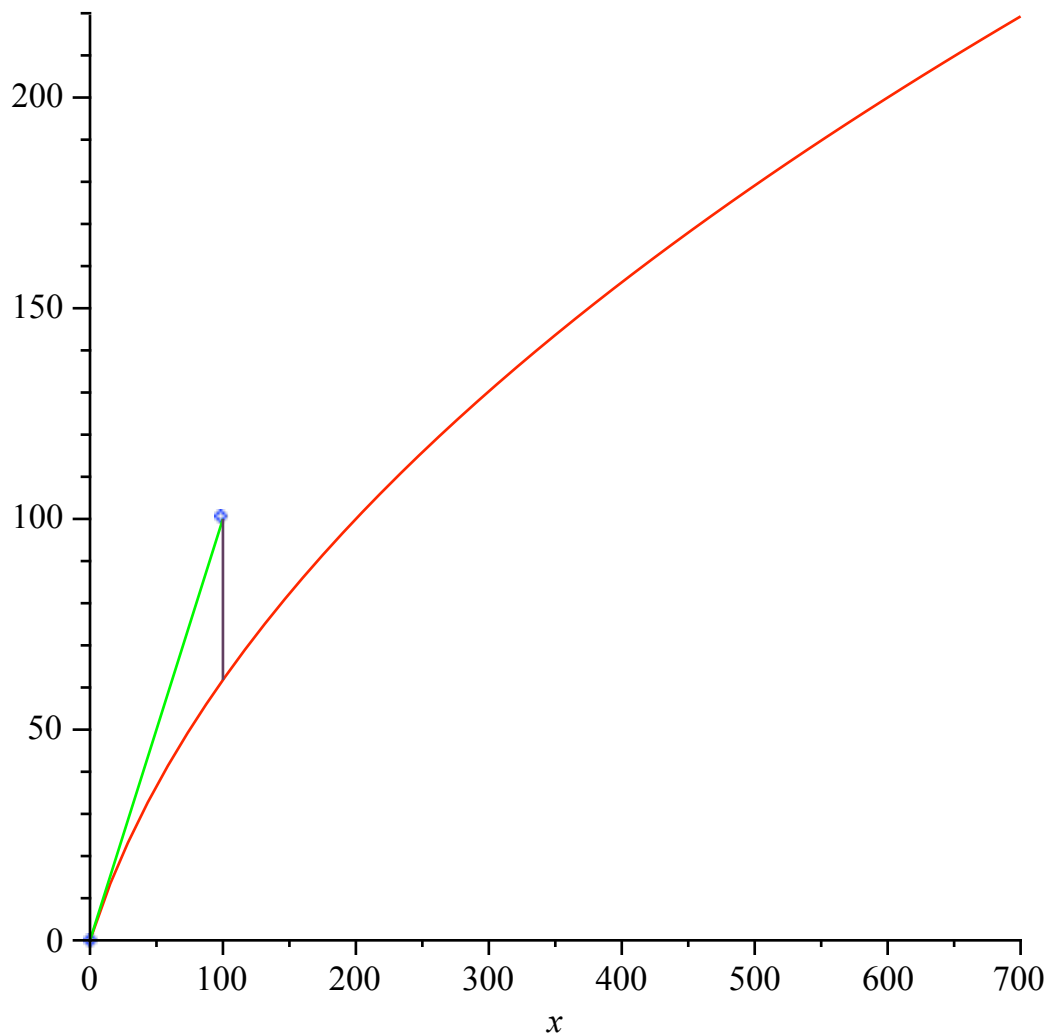
so $y_1 = x_1 = 100$. Thus we will use the point $(x_1, y_1) = (100, 100)$ to approximate the "unknown" point $(100, y(100))$ on the solution curve. We will use the `pointplot` command from the `plots` package to graph the line segment joining the initial point to the point $(100, 100)$, and display it along with the exact solution for comparison.

```
> p2:=pointplot([[0,0],[100,100]],color=blue):
p3:=pointplot([[0,0],[100,100]],style=line,color=green):
display(p1,p2,p3);
```



We can see that there is a fair amount of error, which is represented by the violet segment in the next graph.

```
> p4:=pointplot([[100,100],[100,subs(x=100,rhs(soln))]],style=
line,color=violet):
display(p1,p2,p3,p4);
```



It looks like we miss by about 40.

>

The Second Step

Beginning now at the point $(x_1, y_1) = (100, 100)$, we use the same method as above to find the point $(x_2, y_2) = (200, y_2)$ as an approximation to the point $(200, y(200))$ on the solution curve. We find that the slope of the tangent line to the solution curve through $(100, 100)$ is $f(100, 100) = \frac{5}{\sqrt{125}}$, which is approximately 0.4472135956. Thus we follow the line with equation

$$y - 100 = \frac{5}{\sqrt{125}}(x - 100).$$

For $x = x_2 = 200$, we solve this equation for y , using `evalf` to get a decimal approximation.

```
> evalf(solve(y-100 = 5/sqrt(125)*(200 - 100), y));
144.7213595
```

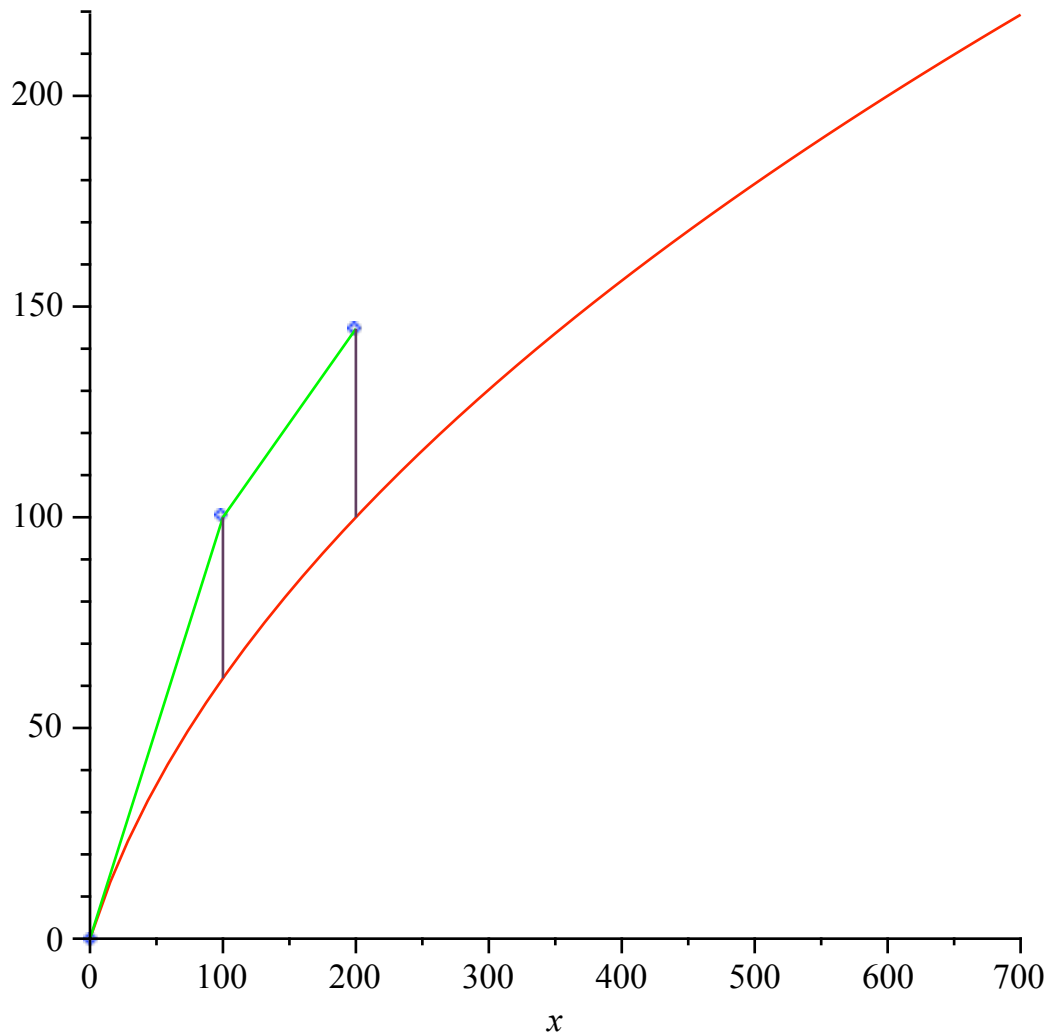
We get $y = y_2 = 144.7213595$. Thus the second point of our approximation is $(x_2, y_2) = (200, 144.7213595)$. We add to our graph.

```
> p2:=pointplot([[0,0],[100,100],[200,144.7213595]],color=blue):
p3:=pointplot([[0,0],[100,100],[200,144.7213595]],style=line,
```

```

color=green):
p5:=pointplot([[200,144.7213595],[200,subs(x=200,rhs(soln))]],
style=line,color=violet):
display(p1,p2,p3,p4,p5);

```



We note that we have

$$x_2 = x_1 + h \text{ and } y_2 = y_1 + hf(x_1, y_1).$$

>

Euler's Method

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

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

$$x_{n+1} = x_n + h \text{ and } y_{n+1} = y_n + hf(x_n, y_n)$$

We will restart from the beginning and use `for` loop structures to help carry us through to the end.

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

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

It will be useful to express the right-hand side of the differential equation as the function $f(x, y)$. To do

this, we use the [unapply](#) command.

```
> f:=unapply(rhs(deq), x, y);
```

$$f := (x, y) \rightarrow \frac{5}{\sqrt{x+25}}$$

```
> IC:=y(0)=0;
```

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

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

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

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

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

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

We create a plot of the exact solution, but do not display it at this time.

```
> p[1]:=plot(soln(x), x=0..700):
```

We initialize the point (x_0, y_0) from the initial conditions and set the step size h .

```
> x[0]:=0; y[0]:=0; h:=100;
```

$$X_0 := 0$$

$$Y_0 := 0$$

$$h := 100$$

The following loop uses Euler's method to find the values of the points $(x_1, y_1) \dots (x_7, y_7)$. Why do we use [evalf](#) here?

```
> for n from 0 to 6 do  
  x[n+1]:=x[n]+h;  
  y[n+1]:=evalf(y[n]+h*f(x[n], y[n]));  
end do;
```

$$X_1 := 100$$

$$Y_1 := 100.0000000$$

$$X_2 := 200$$

$$Y_2 := 144.7213596$$

$$X_3 := 300$$

$$Y_3 := 178.0546929$$

$$X_4 := 400$$

$$Y_4 := 205.7897027$$

$$X_5 := 500$$

$$Y_5 := 230.0432652$$

$$X_6 := 600$$

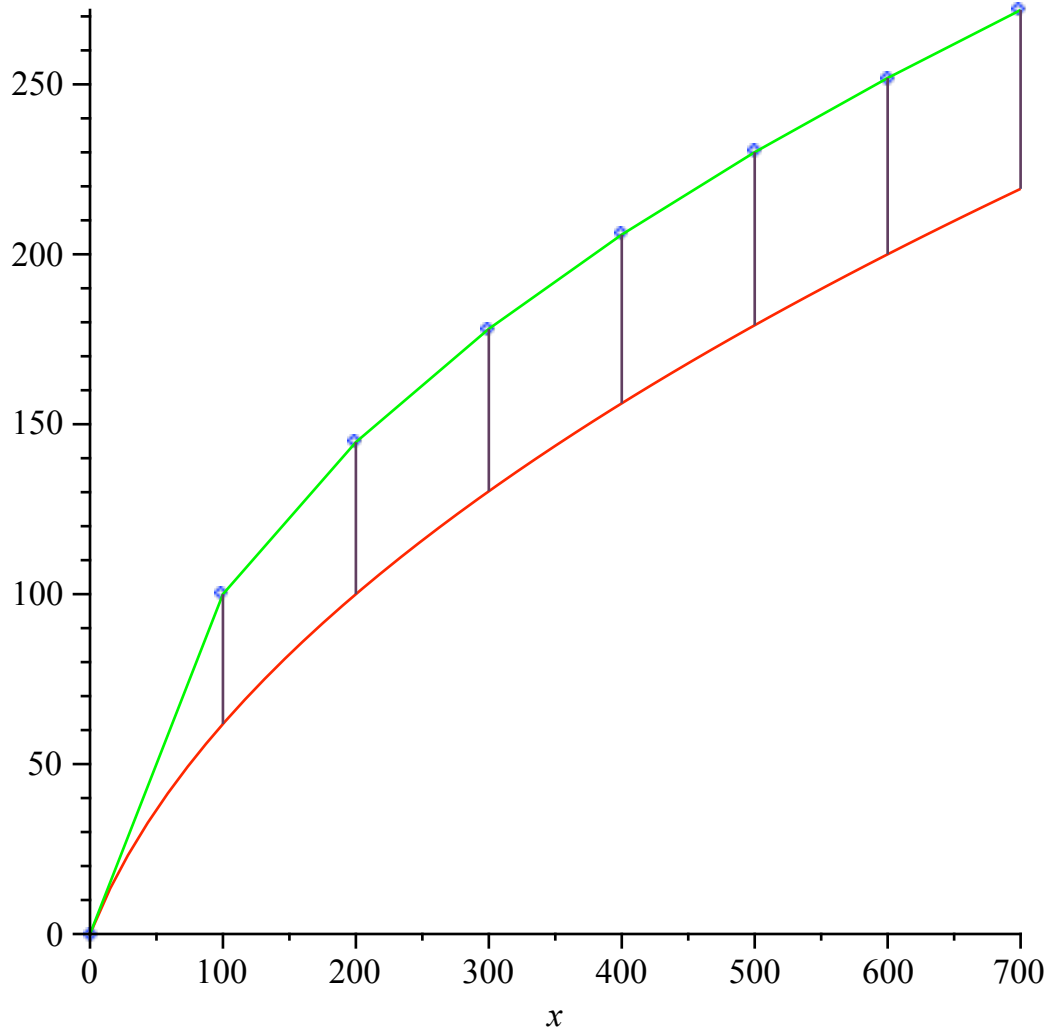
$$Y_6 := 251.8650542$$

$$X_7 := 700$$

$$Y_7 := 271.8650542$$

We create the plots of the points, the line segments connecting the points, and the line segments illustrating the errors at each step. We use a [for](#) loop for the latter. Finally, we display all of the plots.

```
> p[2]:= pointplot([seq([X[n],Y[n]],n=0..7)],color=blue):
p[3]:= pointplot([seq([X[n],Y[n]],n=0..7)],style=line,color=
green):
for n from 1 to 7 do
p[n+3]:=pointplot([[X[n],Y[n]],[X[n],soln(X[n])]],style=line,
color=violet):
end do:
display(seq(p[n],n=1..10));
```



Using the [printf](#) (like the **printf** in C) command, we create a table showing for each $X[n]$ both the computed value $Y[n]$ from Euler's method and the exact value $Y(X[n])$ from the solution function. We use [evalf](#) to express the exact Y values, the $Y(X[n])$, as decimals rounded to 4 places.

```
> printf(" n      X[n]      Y[n]      Y(X[n])");
printf("\n\n");
for n from 0 to 7 do
printf("%2d      %3d      %9.4f      %9.4f \n",n,X[n],Y[n],evalf(soln
(X[n],4)));
od;
n      X[n]      Y[n]      Y(X[n])
```

```

0      0      0.0000      0.0000
1     100     100.0000     61.8034
2     200     144.7214    100.0000
3     300     178.0547    130.2776
4     400     205.7897    156.1553
5     500     230.0433    179.1288
6     600     251.8651    200.0000
7     700     271.8651    219.2582

```

```
>
```

The Maple Way

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

Although all of the above was very illustrative of Euler's method, Maple provides much simpler ways of doing it all. 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):
```

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

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

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 := 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 $x=200$.

```
> Y100(200);
[x = 200., y(x) = 144.721359549995782]
```

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

```
> eval([x, y(x)], Y100(200));
[200., 144.721359549995782]
```

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

```
> eval(y(x), Y100(200));
144.721359549995782
```

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) = 144.721359549995782]
[x = 300., y(x) = 178.054692883329096]
[x = 400., y(x) = 205.789702694590545]
[x = 500., y(x) = 230.043265198223850]
[x = 600., y(x) = 251.865054221823101]
[x = 700., y(x) = 271.865054221823129]
```

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., 144.721359549995782]
[300., 178.054692883329096]
[400., 205.789702694590545]
[500., 230.043265198223850]
[600., 251.865054221823101]
[700., 271.865054221823129]
```

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.

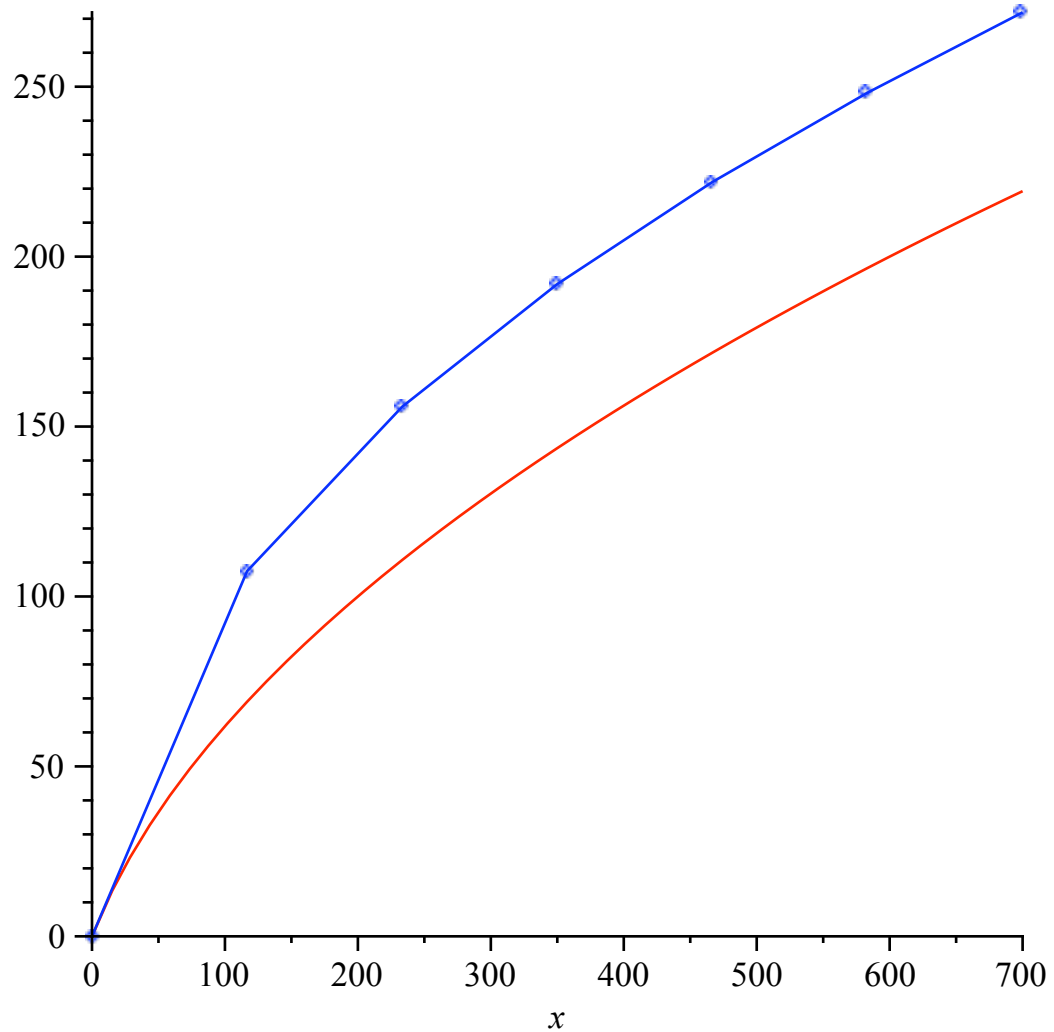
```
> Y100(250);
[x = 250., y(x) = 161.388026216662439]
```

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

```
> eval([x,y(x)],Y100(250));
[250., 161.388026216662439]
```

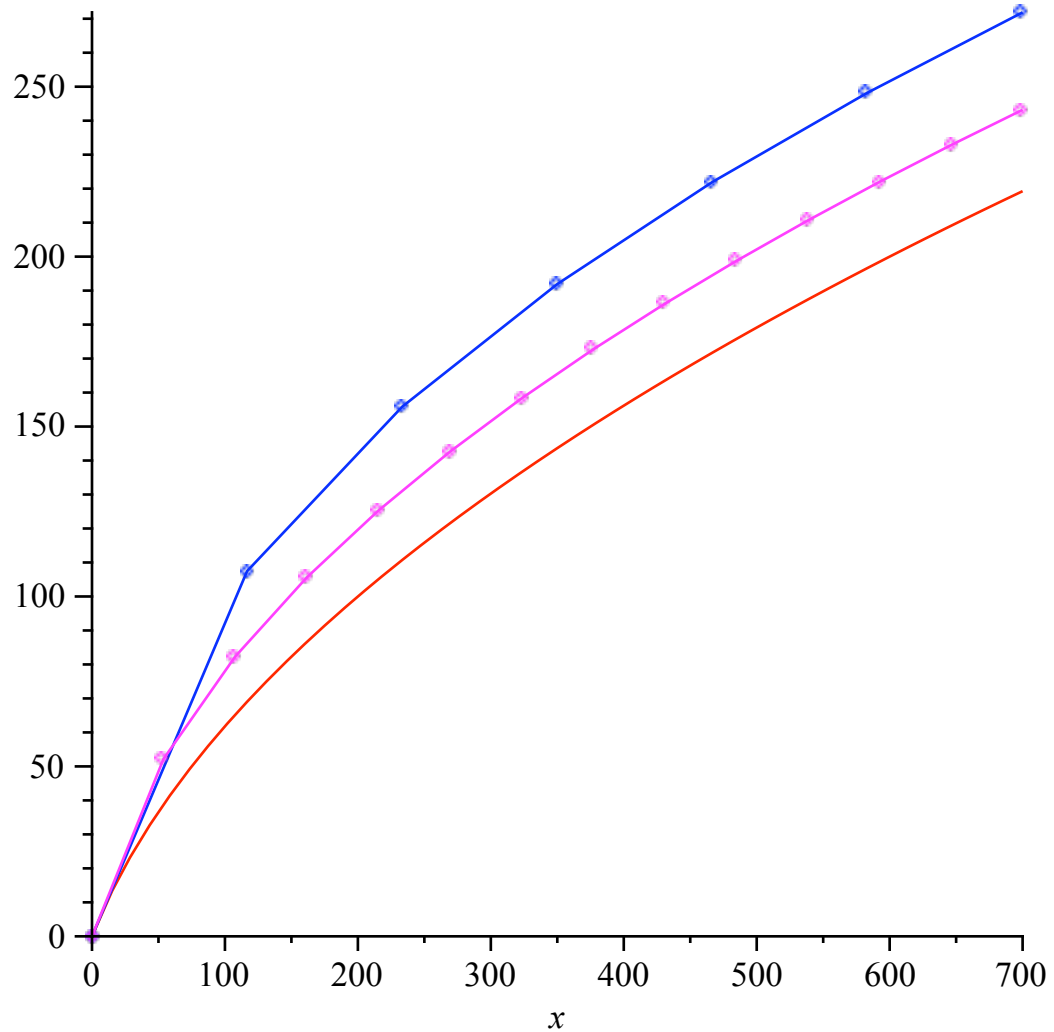
We will use `odeplot` from the `plots` package to create first the point plot and then the line plot to connect the points.

```
> 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));
```



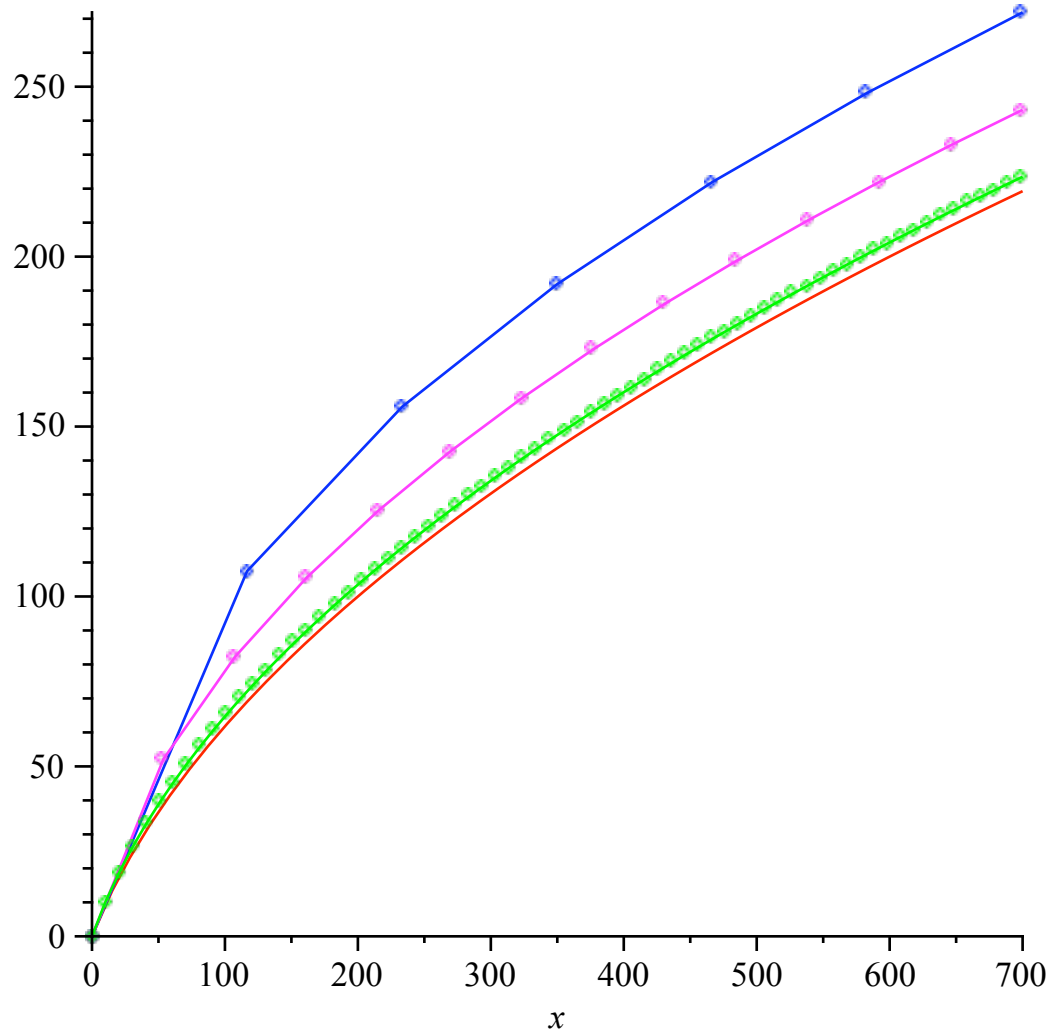
Let's repeat the whole process for a stepsize of 50, giving 14 steps.

```
> 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));
```



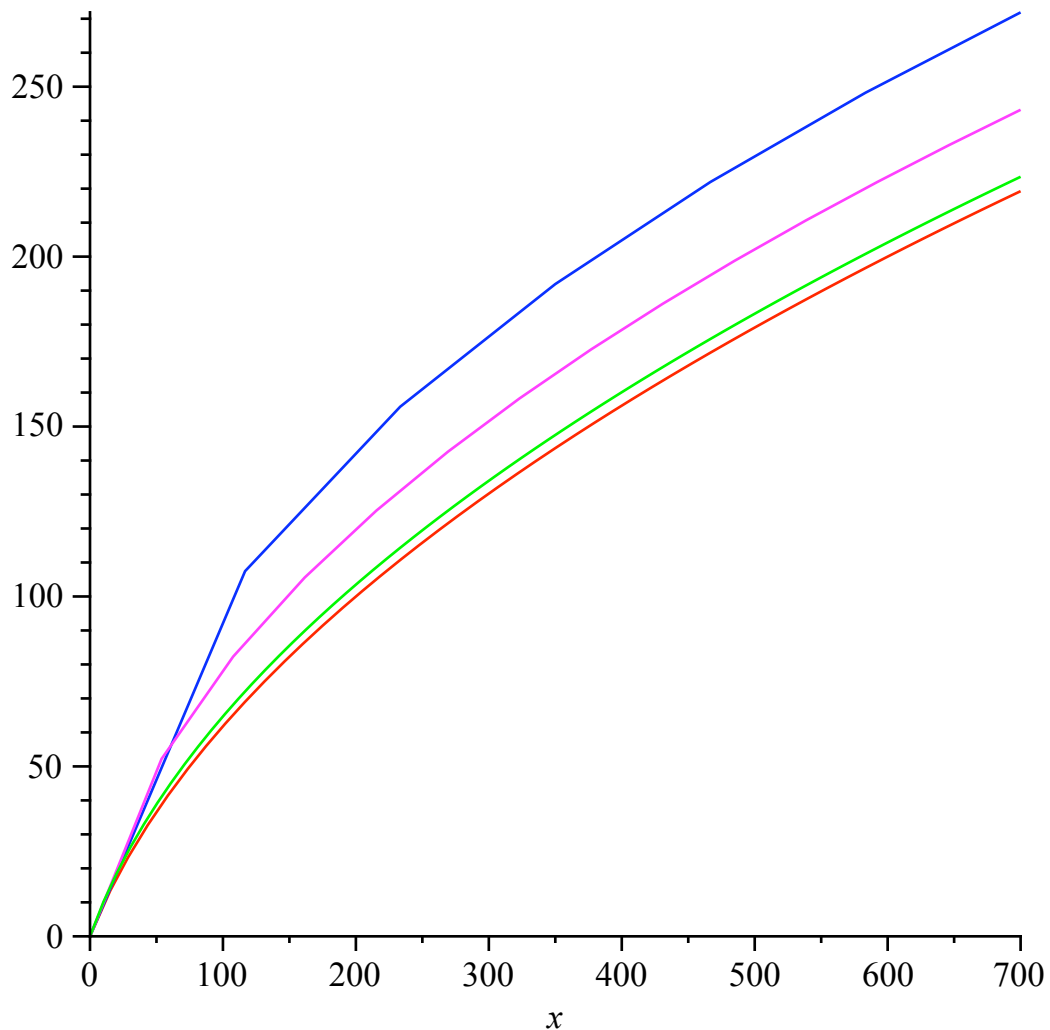
Finally, let's repeat it one more time with a stepsize of 10 for 70 steps.

```
> 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])
0	0	0.0000	0.0000	0.0000
1	100	100.0000	78.8675	64.7175
2	200	144.7214	120.1264	103.4922
3	300	178.0547	151.8687	134.0523
4	400	205.7897	178.6461	160.1053
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

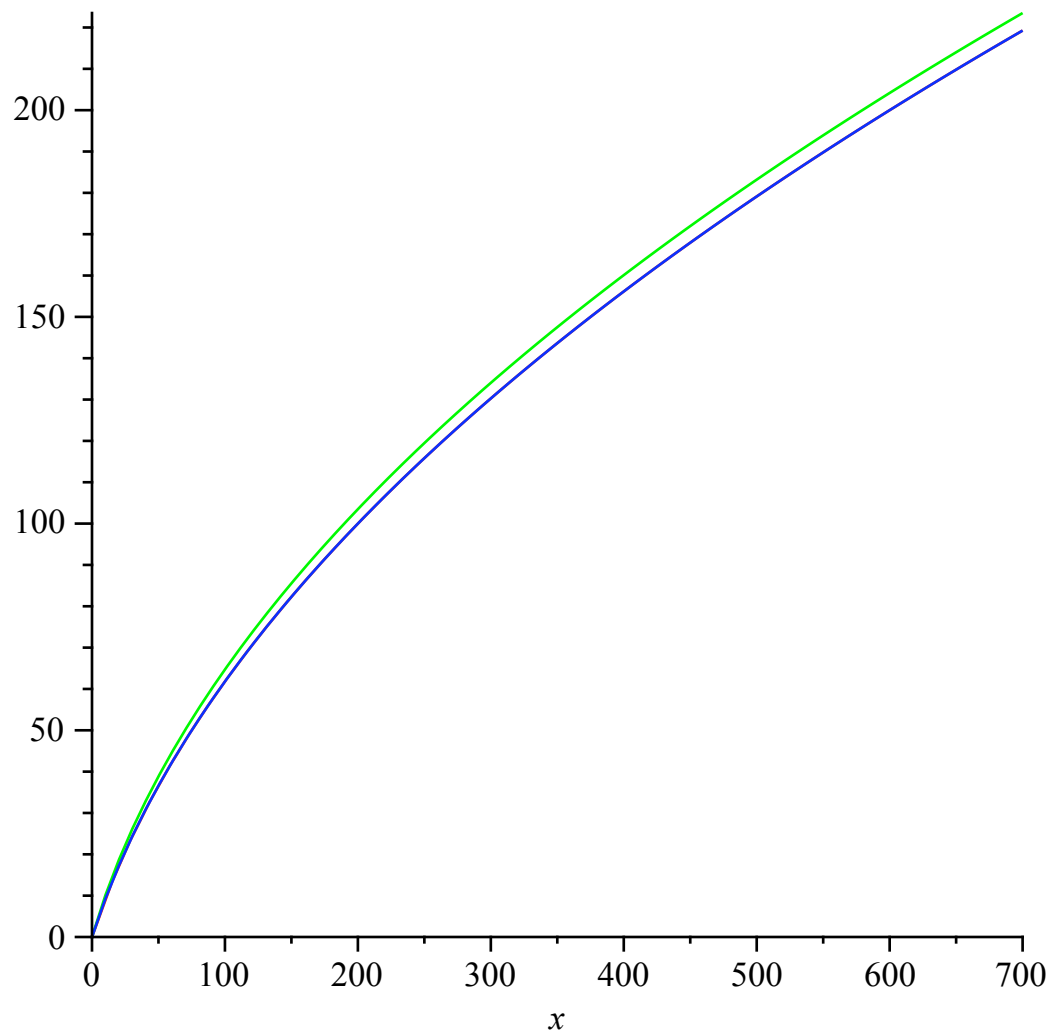
```

In Chapter 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 $x = a \dots b$. Let's compare the actual solution (red) with both Euler's method (green) and the Runge-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]);

```



```

>

```