In this notebook, we explore the Euler method for the numerical solution of first order differential equations. The Euler method is the simplest and most fundamental method for numerical integration. Unfortunately, it is not very accurate, so that in practice one uses more complicated but better methods such as Runge-Kutta. The main value of the Euler method is pedagogical -- it is a good introduction to the ideas used in the numerical integration of differential equations.

**Specification of the Equation**

The differential equation to be solved has the form $dy/dx = f(x,y)$, with an initial condition of the form $y(x_0) = y_0$. We define this equation for Mathematica by giving $f(x,y)$, $x_0$, and $y_0$. To consider a different example than the one given below, just change the three assignment statements below.

\[
\begin{align*}
  f[x_, y_] & := \text{Exp}[x] + y \\
x_0 & = 0; \\
y_0 & = 1;
\end{align*}
\]

**Exact Solution of Equation**

Mathematica can solve simple equations exactly, so it is always worth a try to find the exact solution. In particular, Mathematica can find the solution of the example above. Having the exact solution will allow us to check the accuracy of the numerical method. We use the command DSolve to find the exact solution of the above initial value problem.

\[
\text{ans} = \text{DSolve}\{y'[x] == f[x, y[x]], y[x_0] == y_0, y[x], x}\}
\]

\[
\{\{y[x] \to e^x (1 + x)\}\}
\]

The answer is in the form of a replacement rule. To get a function representing the answer, we construct it as follows:

\[
\text{exactsol}[x_] = y[x] \/. \text{Flatten}[\text{ans}];
\]

Let's check our solution. We first look at the function exactsol[x]:

\[
\text{exactsol}[x]
\]

\[
e^x (1 + x)
\]

Now we check the initial condition:
The initial condition is correct. Now we check that the function satisfies the differential equation, using the Mathematica command \texttt{D}, which carries out symbolic differentiation:

\begin{verbatim}
Simplify[D[exactsol[x], x] - f[x, exactsol[x]]]
\end{verbatim}

This shows that our equation is satisfied. The command \texttt{Simplify} tells Mathematica to combine algebraic terms in order to simplify the solution. In a more complicated problem, it is helpful in telling whether or not a long expression actually simplifies to zero. Let’s plot the solution from \(x = 0\) to \(x = 2\).

\begin{verbatim}
graphexact = Plot[exactsol[x], {x, 0, 2}, AxesLabel -> {"x", "y"}];
\end{verbatim}

\begin{center}
\includegraphics[width=0.6\textwidth]{plot.png}
\end{center}

\section*{Euler Method}

Now we implement the Euler method for numerical integration. We will let \(h\) be the step size, and we start with \(h = 0.1\).
We use the Euler method to integrate from the initial point $x_0$ to a final point $x_f$ which we must specify. For our example, we specify $x_f$ to be 2:

$$x_f = 2.0;$$

We are going to define our Euler method routine so that the output is simply a list of $\{x,y\}$ values generated by the numerical algorithm. The number of steps of size $h$ that we must take to reach $x_f$ is given by

$$\text{nsteps := Ceiling}\left[\frac{x_f - x_0}{h}\right]$$

The function Ceiling[$x$] gives the smallest integer larger than or equal to $x$. Thus if the division of interval length $(x_f-x_0)$ by $h$ doesn't come out even, we take enough steps to get at least as far as $x_f$.

The Euler code below constructs the list of the solution points. It assumes that $x$ is the independent variable, that $y$ is the dependent variable, that $f[x,y]$ is the slope function, that $h$ is the spacing, that $x_0$ is the initial $x$, that $y_0$ is the initial $y$, and that $x_f$ is the final $x$ value for the integration. A more flexible way to write the code would be to have all of these quantities be arguments of the function euler. The present way is simpler to use, however. The heart of the code is the first statement in the Do loop, which increments $y$ by $h*f$ and then increments $x$ by $h$. This is the basic stepping algorithm of the Euler method.

$$\text{euler := Module}\left[\left\{\text{ans}, i, x, y, \text{nsteps}\right\},\right.$$  
$$\text{ans = \{x_0, y_0\}; x = x_0; y = y_0; nsteps = nsteps;}\right.$$  
$$\text{Do[\{y = y + h*f[x, y];} \right.$$  
$$\text{x = x + h; ans = Append[ans, \{x, y\}], \{i, 1, nsteps\}]; ans}\right.$$  

Let's try this out, and name our solution list eulerans1.

$$\text{eulerans1 = euler;}$$

We can look at the list by typing its name.

$$\text{eulerans1}$$  
$$\{\{0, 1\}, \{0.1, 1.2\}, \{0.2, 1.43052\}, \{0.3, 1.69571\}, \{0.4, 2.00027\},\right.$$  
$$\{0.5, 2.34947\}, \{0.6, 2.74929\}, \{0.7, 3.20644\}, \{0.8, 3.72845\},\right.$$  
$$\{0.9, 4.32385\}, \{1., 5.0022\}, \{1.1, 5.77425\}, \{1.2, 6.65209\},\right.$$  
$$\{1.3, 7.64931\}, \{1.4, 8.78117\}, \{1.5, 10.0648\}, \{1.6, 11.5195\},\right.$$  
$$\{1.7, 13.1667\}, \{1.8, 15.0308\}, \{1.9, 17.1388\}, \{2., 19.5213\}\}$$

This gives the $\{x,y\}$ pairs in the numerical solution. How good are the numbers? We look at the last value and compare the exact and approximate.

$$\text{Last[Last[eulerans1]]}$$  
19.5213

$$\text{exactsol[xf]}$$  
22.1672

The exact answer is quite a bit larger than the Euler answer. The discrepancy is larger for large $x$. We can see that easily from a graphical comparison which we do now.
Let's decrease the step size and see if the Euler approximation gets better.

\[ h = 0.01; \]
\[ \text{eulerans2} = \text{euler}; \]
\[ \text{Last[Last[eulerans2]]} \]
\[ 21.875 \]
\[ \text{exactsol[xf]} \]
\[ 22.1672 \]

It is better, but there is still an error of around 1.5%. Let's look at the graph.
The dots are now so close together that the curve looks almost solid.

The solutions are very close graphically. We can reduce the error even further, at the expense of more computation. We set the spacing to be \( h = 0.001 \). The calculation (which has 2000 steps) takes awhile.

\[
h = 0.001;
\]
\[
euler3 = euler;
\]
\[
Last[Last[euler3]]
\]
\[
22.1376
\]
Even after 2000 steps, the answer is still off in the fourth digit. The Euler method is just not very accurate and requires an extremely small step for good results. Fortunately, there are other much more efficient methods, such as the one built-in to Mathematica in the form of the command NDSolve. We conclude this notebook by taking a quick look at NDSolve.

### Using NDSolve

NDSolve is Mathematica's built-in command for the numerical integration of differential equations. It is very efficient, very sophisticated and very fast. We will use it here to solve our same example equation. The syntax is similar to that of DSolve, with the main difference being that we also specify the endpoints of the x-interval on which we want the solution -- in this case, \( x_0 \) and \( xf \).

```plaintext
numans = NDSolve[{y'[x] == f[x, y[x]], y[x0] == y0, y[x], {x, x0, xf}}
                  [[y[x] -> InterpolatingFunction[{{0., 2.}}, <>][x]]]
```

The output from NDSolve is more sophisticated and more useful than a simple list of coordinates. It is an interpolating function, which basically is an algorithm to interpolate smoothly from the output list generated by the integration. All of this is transparent to the user, and if you want to define a function which represents the solution, you do this as follows:

```plaintext
numsol[x_] = y[x] /. Flatten[numans];
```

Now we can use numsol just like any analytically defined function. We check the initial and final values:

```plaintext
numsol[x0]
1.

numsol[xf]
22.1672

exactsol[xf]
22.1672
```

We see that NDSolve has given us six correct digits.