Part III. First Order Ordinary Differential Equations

Section 4. Approximate Solutions

In this section looping constructs are used to approximate solutions to initial value problems (IVP). Once created, and tested for accuracy, these loops can be easily be made into procedures that can be used on any IVP. Throughout the section discussions will refer to the standard IVP

$$y'(t) = f(t, y(t)) \ , \ y(\mathrm{t0}) = \mathrm{y0} \ .$$

One step at a time: Euler's algorithm

The Euler one-step algorithm generates IVP approximations by moving from point to point along tangent lines in the slope field (Simmons/Krantz, Chapter 9, Section 2). Movement from point (t, y) is horizontal by a displacement h and vertical by the displacement $f(t, y)\, h$. The horizontal distance h stays the same, it is called the "step size". Thus t increases to $t + h$ and y changes to $y + f(t, y)\, h$ yielding the famous Euler iteration formulas:

$$t[n] = t[n\text{-}1] + h$$

$$y[n] = y[n\text{-}1] + h\, f(\, t[n\text{-}1], y[n\text{-}1]\,)$$

Euler's algorithm will be implemented for the IVP

$$y'(t) = \mathrm{cost}(t)\, y(t)\ ,\ y(0) = 1$$

We begin by entering the differential equation and then defining the function f. The equation is entered so we can solve it later to check the algorithm. We also want to make sure that you see how the function f is obtained from the differential equation (which must be in normal form).

```
In[1]:= DE = y'[t] ==  Cos[t]*y[t]
        f[t_,y_] := Cos[t]*y
Out[1]= y′[t] == Cos[t] y[t]
```

We will use the symbols T and Y to denote the indexed variables that contain the values of t and y respectively. Both T and Y are defined recursively in a special way that tells *Mathematica* to remember their values.

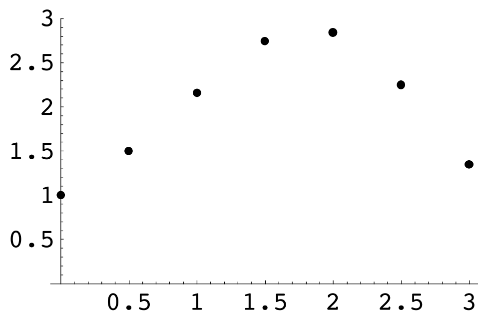
Begin with step size $h = 0.5$.

The next entries first define h, T[0], and Y[0]. Then T[n] and Y[n] are defined as indexed variables that remember their values. See Section 2.5.9 of *The Mathematica Book*.

```
In[3]:= h = 0.5; T[0] = 0; Y[0] = 1;
        T[n_] := T[n] = T[n-1] + h;
        Y[n_] := Y[n] = Y[n-1] + h*f[T[n-1],Y[n-1]];
```

Once defined we can plot the approximation points using ListPlot. See below where the first 6 approximation points are plotted.

In[6]:=
```
ListPlot[ Table[{T[n],Y[n]}, {n,0,6}], PlotStyle->PointSize[0.02],
          PlotRange->{0,3} ]
```



Out[6]= - Graphics -

The following entry shows how to display the approximate solution values in matrix form.

In[7]:=
```
Table[{T[n],Y[n]}, {n,0,6}]//MatrixForm
```

Out[7]//MatrixForm=

$$\begin{pmatrix} 0 & 1 \\ 0.5 & 1.5 \\ 1. & 2.15819 \\ 1.5 & 2.74122 \\ 2. & 2.83818 \\ 2.5 & 2.24763 \\ 3. & 1.34729 \end{pmatrix}$$

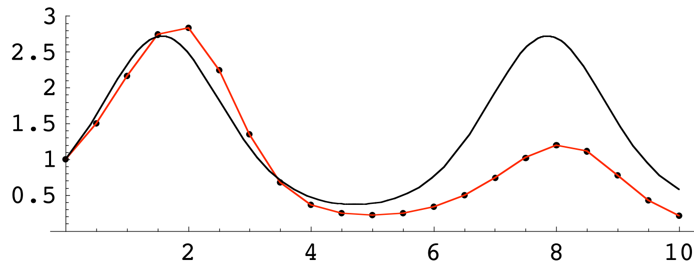
To check the algorithm we will obtain the solution formula and then add its graph to a plot of the approximation points. We will to use the solution several more times, so make it into a function named g.

In[8]:=
```
soln = DSolve[ {DE, y[0]==1}, y, t];
g := soln[[1,1,2]]
g[t]
```

Out[10]= $e^{\text{Sin}[t]}$

Here is a plot of g, the Euler approximation points from $t = 0$ to $t = 10$ and the broken line obtained by connecting the points (red).

```
In[11]:= pts = ListPlot[ Table[{T[n],Y[n]}, {n,0,20}],
                    PlotStyle->PointSize[0.01], PlotRange->{0,3} ];
         lines = ListPlot[ Table[{T[n],Y[n]}, {n,0,20}], PlotJoined->True,
                    PlotStyle->RGBColor[1,0,0], PlotRange->{0,3} ];
         Show[ pts, lines, Plot[ g[t], {t,0,10} ], AspectRatio->1/3 ]
```




```
Out[13]= - Graphics -
```

Make it yours: User defined procedures

Examination of the input that creates a Table of Euler approximation points shows that it requires the function *f*, the initial values t0 and y0, the step size *h*, and the number *N* of points that are to be generated. In other words, the process can be regarded as a function (or "procedure") that transforms the input *f*, t0, y0, *h*, *N* into a list of approximation points.

The following input makes this process into a user defined *Mathematica* function with the name euler. It is defined to take *f*, t0, y0, *h*, *N* as input and output the approximation points in a list for easy plotting. The official name of the procedure is a "module" and requires the Module function to define. Each line in the following entry is explained below.

```
In[62]:= euler[f_,t0_,y0_,h_,N_] :=
         Module[
                 {T,Y},
                 T[0] = t0; Y[0] = y0;
                 T[n_] := T[n] = T[n-1] + h;
                 Y[n_] := Y[n] = Y[n-1] + h*f[T[n-1],Y[n-1]];
                 Table[ {T[n],Y[n]}, {n,0,N} ]
               ]
```

What is going on here? Answer: An explanation for each line appears below.

1. The first line names the procedure and identifies the input variables. Finish it with := .

2. The second line is the Module function and its opening left bracket.

3. The third line is a list of the local variables, T and Y, that will be used in the procedure. These variables work independently of their definitions in the Notebook and will cease to exist once the procedure is run.

4 - 7. These are the lines defining the procedure. You will recognize each one from the input that created the Euler approximation above. The last entry is the output for the procedure: The Table of approximation points.

8. Line 8 contains the right bracket signalling the end of the Module function (and the procedure definition).

Let's test the function named euler on the data that created the matrix of approximation points that is displayed following the first graph appearing above.

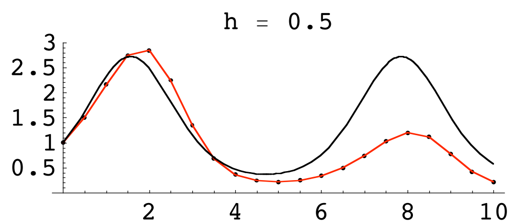
```
In[161]:= euler[f,0,1,0.5,6]//MatrixForm
```

Out[161]//MatrixForm=

$$\begin{pmatrix} 0 & 1 \\ 0.5 & 1.5 \\ 1. & 2.15819 \\ 1.5 & 2.74122 \\ 2. & 2.83818 \\ 2.5 & 2.24763 \\ 3. & 1.34729 \end{pmatrix}$$

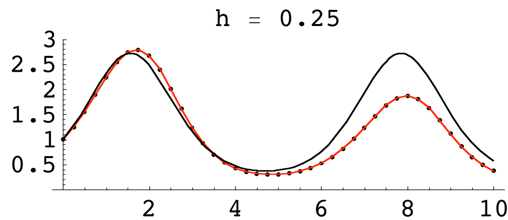
Cool. Now lets use euler to recreate the approximation for $h = 0.5$.

```
In[83]:= approx = euler[f,0,1,0.5,20];
pts = ListPlot[ approx, PlotStyle->PointSize[0.01], PlotRange->{0,3} ];
lines = ListPlot[ approx, PlotJoined->True, PlotStyle->RGBColor[1,0,0],
                  PlotRange->{0,3} ];
Show[ pts, lines, Plot[ g[t], {t,0,10} ], AspectRatio->1/3,
      PlotLabel→ "h = 0.5" ]
```



And here is the plot for the approximation when $h = 0.25$. In order to get to $t = 10$, $N = 40$

In[87]:=
```
approx = euler[f,0,1,0.25,40];
pts = ListPlot[ approx, PlotStyle->PointSize[0.01], PlotRange->{0,3} ];
lines = ListPlot[ approx, PlotJoined->True, PlotStyle->RGBColor[1,0,0],
                  PlotRange->{0,3} ];
Show[ pts, lines, Plot[ g[t], {t,0,10} ], AspectRatio->1/3,
      PlotLabel→ "h = 0.25" ]
```



Cutting the step size in half appears to have cut the error in half also. See Ledder, page 113.

Improve it: The Euler two step algorithm (Improved Euler)

The one step algorithm can be dramatically improved by making a very simple correction. Move from (t, y) along the line with slope $f(t, y)$ to the next Euler point (step one), use f to calculate the slope there, move back to (t, y), calculate the average of the two slopes, and take a second (and last) step in the averaged direction. The algorithm looks like this:

$$t[n] = t[n\text{-}1] + h$$

$$k[n] = y[n\text{-}1] + h\ f(t[n\text{-}1], y[n\text{-}1])$$

$$y[n] = y[n\text{-}1] + 0.5\ h\ (f(\ t[n\text{-}1], y[n\text{-}1] + f(t[n], k[n]\ )$$

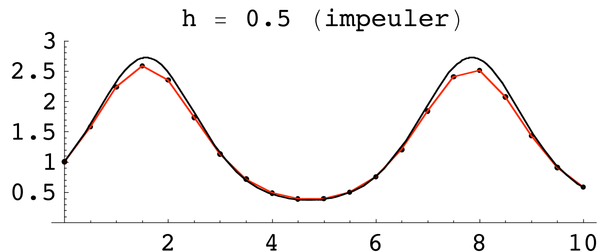
See Simmons/Krantz, Chapter 9, Section 3, for a clear explanation of why this algorithm is so much better.

The Improved Euler algorithm is easily implemented in a procedure as follows. One more local variable is needed.

In[14]:=
```
impeuler[f_,t0_,y0_,h_,N_] :=
Module[
        {T,K,Y},
        T[0] = t0; Y[0] = y0;
        T[n_] := T[n] = T[n-1] + h;
        K[n_] := K[n] = Y[n-1] + h*f[T[n-1],Y[n-1]];
        Y[n_] := Y[n] = Y[n-1] + 0.5*h*(f[T[n-1],Y[n-1]] + f[T[n],K[n]]);
        Table[ {T[n],Y[n]}, {n,0,N} ]
      ]
```

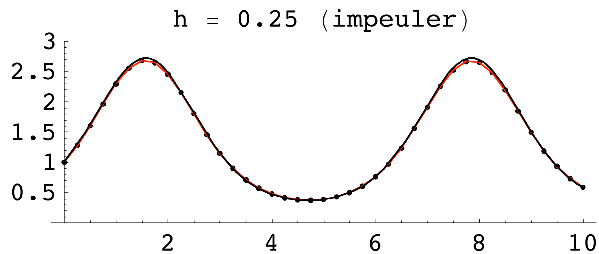
The next two plots show that modeuler works wonders on the example IVP.

```
In[19]:= approx = impeuler[f,0,1,0.5,20];
pts = ListPlot[ approx, PlotStyle->PointSize[0.01], PlotRange->{0,3} ];
lines = ListPlot[ approx, PlotJoined->True, PlotStyle->RGBColor[1,0,0],
            PlotRange->{0,3} ];
Show[ pts, lines, Plot[ g[t], {t,0,10} ], AspectRatio->1/3,
      PlotLabel→ "h = 0.5 (impeuler)" ]
```




```
Out[22]= - Graphics -
```

```
In[27]:= approx = impeuler[f,0,1,0.25,40];
pts = ListPlot[ approx, PlotStyle->PointSize[0.01], PlotRange->{0,3} ];
lines = ListPlot[ approx, PlotJoined->True, PlotStyle->RGBColor[1,0,0],
            PlotRange->{0,3} ];
Show[ pts, lines, Plot[ g[t], {t,0,10} ], AspectRatio->1/3,
      PlotLabel→ "h = 0.25 (impeuler)" ]
```




```
Out[30]= - Graphics -
```

Using NDSolve to generate numeric solutions

The function named NDSolve will generate numeric solutions to an IVP. Here, for example, is the numeric solution to the IVP used above. Recall that the equation looks like this.

```
In[103]:= DE
Out[103]= y′[t] == Cos[t] y[t]
```

The input to NDSolve is the same as DSolve except that the independent variable is replaced with an interval.

```
In[104]:= soln = NDSolve[ {DE,y[0]==1}, y, {t,0,10} ]
Out[104]= {{y → InterpolatingFunction[{{0., 10.}}, <>]}}
```

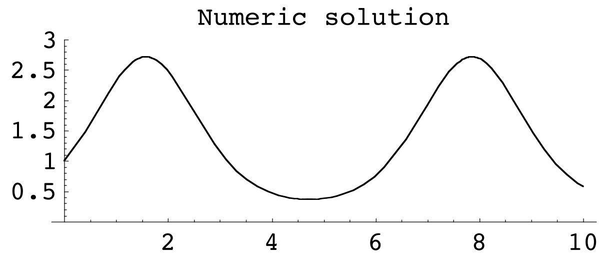
The output is a sequence of approximate solution values (in this case from $t = 0$ to $t = 10$) that will be interpolated by an "interpolating function" to any t value from 0 to 10.

Values can be obtained via substitution.

In[105]:=
```
y[2]/.soln
```
Out[105]=
```
{2.48258}
```

Plots are obtained the same way.

In[110]:=
```
Plot[ y[t]/.soln, {t,0,10}, PlotRange->{0,3}, AspectRatio->1/3,
      PlotLabel->"Numeric solution"]
```



The following matrix of values displaying *t*-values, approximate *y*-values, and exact *y*-values shows that the numeric algorithm used by *Mathematica* is very accurate.

In[124]:=
```
Join[{{t,approxy,exacty}},
     Table[ {t,y[t],g[t]}, {t,0,3,0.5} ]]/.soln[[1,1]]//MatrixForm
```
Out[124]//MatrixForm=

t	approxy	exacty
0	1.	1.
0.5	1.61515	1.61515
1.	2.31978	2.31978
1.5	2.71148	2.71148
2.	2.48258	2.48258
2.5	1.81934	1.81934
3.	1.15156	1.15156

The following was copied from *The Mathematica Book.*

For ordinary differential equations, NDSolve by default uses an LSODA approach, switching between a non-stiff Adams method and a stiff Gear backward differentiation formula method.

In addition

NDSolve supports explicit Method settings that cover most known methods from the literature.

See the Help page for NDSolve

In[133]:= **?NDSolve**

```
NDSolve[eqns, y, {x, xmin, xmax}] finds a numerical solution to the ordinary
   differential equations eqns for the function y with the independent
   variable x in the range xmin to xmax. NDSolve[eqns, y, {x, xmin,
   xmax}, {t, tmin, tmax}] finds a numerical solution to the partial
   differential equations eqns. NDSolve[eqns, {y1, y2, ... }, {x,
   xmin, xmax}] finds numerical solutions for the functions yi. More…
```

A preview of second order equations: *An aging spring*

Equations similar to the following second order differential equation are used as models for the position of a mass that is oscillating on an aging spring.

In[134]:= **agingspring = y''[t] + Exp[-0.2*t]*y[t] == 0**

Out[134]= $e^{-0.2\,t}\,y[t] + y''[t] == 0$

The exact solution formula for this equation requires special functions that are discussed in Appendix A2. Given an IVP, the NDSolve solution works just as well. The next entry generates that solution to the aging spring equation satisfying the initial conditions $y(0) = 1$ and $y'(0) = 0$.

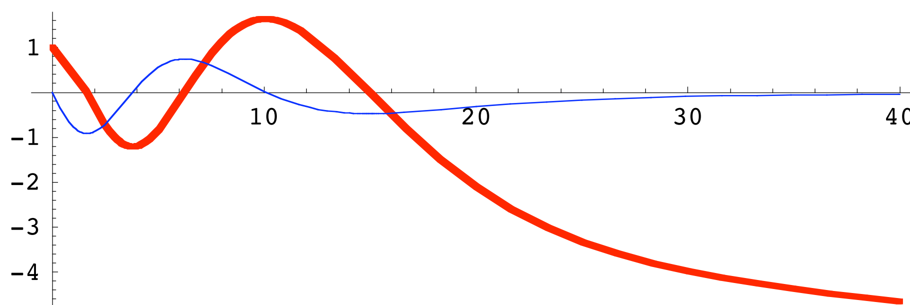
In[135]:= **soln = NDSolve[{agingspring, y[0]==1, y'[0]==0}, y, {t,0,40}]**

Out[135]= {{y → InterpolatingFunction[{{0., 40.}}, <>]}}

The following plot displays position and velocity of the spring for $t = 0$ to $t = 40$. Position is red and thick, velocity is blue.

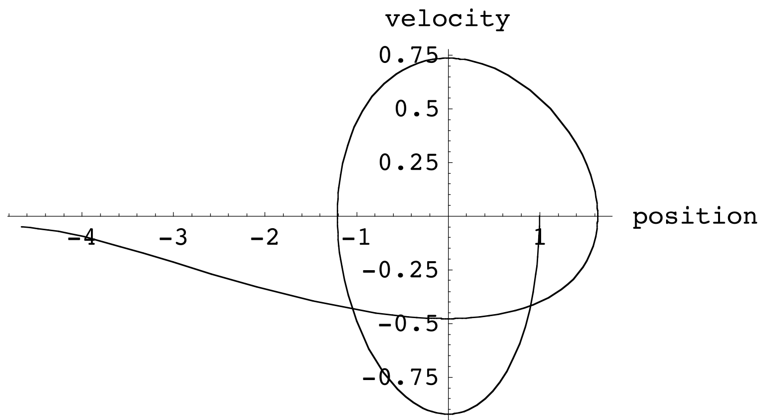
Note the use of the Evaluate function.

```
In[151]:= Plot[ Evaluate[{y[t],y'[t]}/.soln], {t,0,40}, AspectRatio->1/3,
          PlotStyle->{{RGBColor[1,0,0],Thickness[0.007]},RGBColor[0,0,1]} ]
```



The last plot of velocity versus position is called the phase plane trajectory. It requires the use of parametric plot.

In[153]:=
```
ParametricPlot[ {y[t],y'[t]}/.soln, {t,0,40},
                AxesLabel->{"position","velocity"} ]
```



Question: What happens to the object as the spring continues to age?

Answer: Study the phase plane trajectory carefully, it shows us were the object is going and how fast it is moving as it goes there.