Part III. First Order Ordinary Differential Equations

Section 4. Approximate Solutions

In this section, **for..do** loops are used to approximate solutions to initial value problems (IVP). Once created, and tested for accuracy, these loops can be turned into procedures for application to any IVP. Throughout the section, discussions will refer to the standard IVP

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

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 direction field (Ledder, Chapter 2, Section 5). Movement from a point (t, y) is horizontal by a displacement h and vertical by the displacement $f(t, y)h$. The horizontal distance h, called the "step size", stays the same. Thus, t increases to $t + h$ and y changes to $y + f(t, y)h$, yielding the famous Euler iteration formulas:

$$t_{n+1} = t_n + h$$

$$y_{n+1} = y_n + f(t_n, y_n)\, h$$

Euler's algorithm will be implemented for the IVP

$$\frac{d}{dt} y(t) = \cos(t)\, y(t) \ , \ y(0) = 1 \ .$$

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.

```
> DE := diff(y(t),t) = cos(t)*y(t);
  f := (t,y) -> cos(t)*y;
```

$$DE := \frac{d}{dt} y(t) = \cos(t)\, y(t)$$

$$f := (t, y) \rightarrow \cos(t)\, y$$

We will use the symbols T and Y to denote the indexed variables appearing in the iteration formulas. Using t and y can confuse Maple because these symbols are also used to define DE.

The indexed variables are initialized with two assignments.

```
> T[0] := 0;
  Y[0] := 1;
```

$$T_0 := 0$$

$$Y_0 := 1$$

- Key Observation : By making these entries, we have told Maple to set up a "Table" named T and another **Table** named Y. Both T and Y can then be used to store any data we want, using any indices that we find convenient. Data is stored by making assignments via the square bracket notation displayed above. Note that indexed variables prettyprint as subscripted variables. Data that is stored in a **Table** can be overwritten at any time by making a new assignment. **Tables** are marvelous.

The following entry displays the initial point, thereby confirming that the correct initial values have been given.

• Key Observation. A point is entered using square brackets, making it officially into a list.

```
> [T[0],Y[0]];
```

[0, 1]

The next entry defines the step size (we take $h = 0.5$) and uses a **for..do** loop to generate the tabular entries $T[n]$ and $Y[n]$ for $n = 1, 2, \ldots, 20$. All output is suppressed.

```
> h := 0.5:
  for n from 0 to 19
    do
      T[n+1] := T[n] + h:
      Y[n+1] := Y[n] + f(T[n],Y[n])*h
    end do:
  unassign('n');
```

Two more observations:

1. A **for..do** loop is ended with "end do". The word "od" can also be used to signal the end of the loop (and must be used in earlier versions of Maple).

2. As a default, the count variable n increments by 1. This can be changed to any increment desired. For example, to increment by 0.5 type "by 0.5" between "n" and "from" or after "19". The loop ended when n was assigned the value 20. The variable n was unassigned so that it would be free for use later in the worksheet.

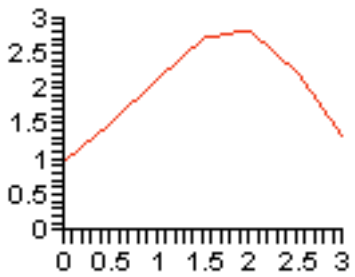
The following entry uses $T[n]$ and $Y[n]$ values to make a list named L that contains the initial point and the first 6 approximations. The 10 digit output is suppressed and then the output is displayed with 3 digit accuracy. There is little to be gained by using lots of significant figures to report the result of a crude approximation like this.

```
> L := [ [T[n],Y[n]] $ n=0..6]: evalf[3](%);
```

[[0., 1.], [0.5, 1.5], [1.0, 2.16], [1.5, 2.74], [2.0, 2.84], [2.5, 2.25], [3.0, 1.35]]

And here is how the points in L plot, in the default line style: The points are plotted and connected with line segments.

```
> plot( L, t=0..3, y=0..3);
```



To check the algorithm we will obtain the solution formula and then add its graph to the plot. Because we want

to use the solution several more times, the unapply procedure is used to make it into a function named g.

```
> soln := dsolve( {DE, y(0)=1} );
  g := unapply(rhs(soln),t);
```

$$soln := y(t) = \mathbf{e}^{\sin(t)}$$

$$g := \exp @ t \rightarrow \sin(t)$$

Don't be frightened by the appearance of the function g. It is Maple's way of saying that g is the composition of the sine function

$$t \rightarrow \sin(t)$$

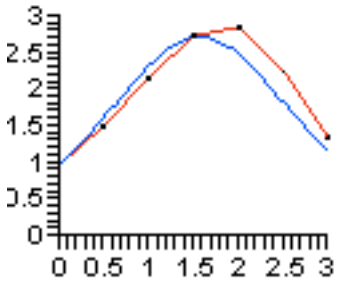
and the exponential function exp. The symbol @ is used to denote function composition.

```
> g(t);
```

$$\mathbf{e}^{\sin(t)}$$

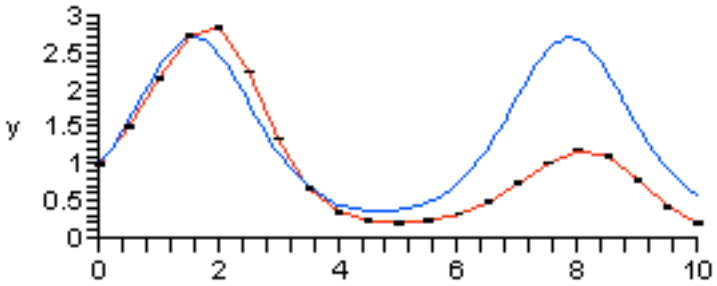
Here is the plot we promised. The seven points that determine the line segments are also plotted.

```
> plot( [L,g(t),L], t=0..3, y=0..3, style=[line$2,point],
        color=[red,blue,black]);
```



The next picture displays the Euler approximation extended over all 20 points generated in the **for..do** loop.

```
> L := [ [T[n],Y[n] ] $ n=0..20]:
  plot( [L,g(t),L], t=0..10, y=0..3, style=[line$2,point],
        color=[red,blue,black]);
```



Make it yours: User-defined procedures

Examination of the **for..do** loop shows that ituses 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 two tables of data named *T* and *Y*.

The following input makes this process into a user-defined Maple procedure with the name Euler. It is defined to take *f*, *t0*, *y0*, *h*, *N* as input and output the Euler approximation points in a list for easy plotting.

• Do not be intimidated by the code. Each line is explained below.

```
> Euler := proc(f,t0,y0,h,N)
    local n, T, Y;
      T[0] := t0;
      Y[0] := y0;
      for n from 0 to N-1
        do
          T[n+1] := T[n] + h:
          Y[n+1] := Y[n] + f(T[n],Y[n])*h:
        end do:
      [ [T[m],Y[m]] $ m=0..N ]
  end proc;
```

```
Euler := proc(f, t0, y0, h, N)
local n, T, Y;
      T[0] := t0;
      Y[0] := y0;
      for n from 0 to N - 1 do T[n + 1] := T[n] + h; Y[n + 1] := Y[n] + f(T[n], Y[n])*h; end do;
      [`$`([T[m], Y[m]], m = 0 .. N)];
end proc;
```

Read each line of the procedure definition to yourself like this:

Line 1: "Euler is defined to be a procedure that depends on the input f, t0, y0, h, N."

Line 2: "The variables n, T, and Y will be used locally within the procedure.

Lines 3 - 9: These lines use the input data to initialize T and Y and define the **for..do** loop.

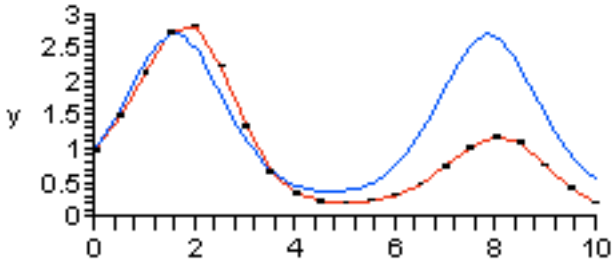
Line 10: "This line is the output of the procedure: A list of N+1 data points."

Line 11: "The procedure ends here."

"Euler" will be tested on the IVP given above. The function *f* has already been defined.

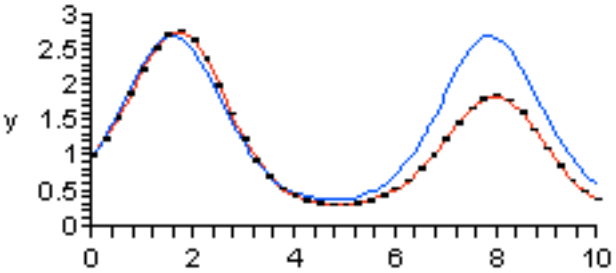
The following input assigns the name *L* to the output when Euler is applied to the appropriate input sequence. The plot should look familiar.

```
> L := Euler(f,0,1,0.5,20):
  plot( [L,g(t),L], t=0..10, y=0..3, style=[line$2,point],
         color=[red,blue,black]);
```



Cool. Now lets use Euler to plot the approximation for $h = 0.25$. In order to get all the way to $t = 10$, we must input $N = 40$.

```
> L := Euler(f,0,1,0.25,40):
  plot( [L,g(t),L], t=0..10, y=0..3, style=[line$2,point],
        color=[red,blue,black]);
```



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

Modify it: The Euler two step algorithm

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, then 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:

$$t_{n+1} = t_n + h$$

$$k = y_n + f(t_n, y_n)\, h$$

$$y_{n+1} = y_n + 0.5\,(f(t_n, y_n) + f(t_{n+1}, k))\, h$$

Ledder (pages 110-112) gives a clear explanation of why this algorithm is so much better.

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

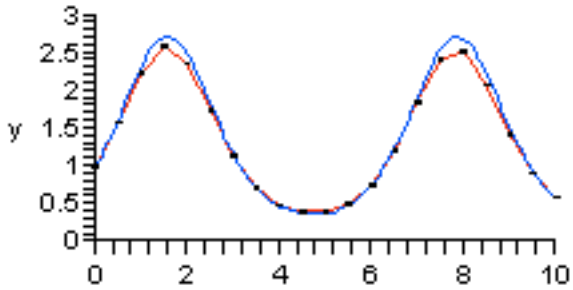
```
> ModEuler := proc(f,t0,y0,h,N)
    local n, k, T, Y;
      T[0] := t0;
      Y[0] := y0;
      for n from 0 to N-1
        do
          T[n+1] := T[n] + h:
```

```
        k := Y[n] + f(T[n],Y[n])*h:
        Y[n+1] := Y[n] + 0.5*(f(T[n],Y[n])+f(T[n+1],k))*h:
      end do:
    [ [T[m],Y[m]] $ m=0..N ]
end proc;
```

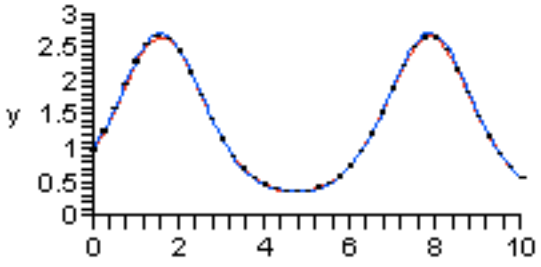
```
ModEuler := proc(f, t0, y0, h, N)
local n, k, T, Y;
      T[0] := t0;
      Y[0] := y0;
      for n from 0 to N - 1 do T[n + 1] := T[n] + h;
            k := Y[n] + f(T[n], Y[n])*h;
            Y[n + 1] := Y[n] + 0.5*(f(T[n], Y[n]) + f(T[n + 1], k))*h;
      end do;
      [`$`([T[m], Y[m]], m = 0 .. N)];
end proc;
```

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

```
> L := ModEuler(f,0,1,0.5,20):
  plot( [L,g(t),L], t=0..10, y=0..3, style=[line$2,point],
        color=[red,blue,black]);
```




```
> L := ModEuler(f,0,1,0.25,40):
  plot( [L,g(t),L], t=0..10, y=0..3, style=[line$2,point],
        color=[red,blue,black]);
```



Using dsolve to generate numeric solutions

The **dsolve** procedure can be used to generate numeric solutions in tabular form. Enter the IVP (in a set, as usual), the unknown function, and then either the keyword

```
numeric
```

or the equation

```
type=numeric
```

Then add an another optional equation of the form

```
output=array([ sequence of t values ])
```

The output is an array of approximate solution values.

The next entry generates approximate solution values for our IVP at $t = 0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0$.

```
> dsolve( {DE,y(0)=1}, y(t), numeric, output=array([k/2$k=0..6]));
```

$$\begin{bmatrix} [t, y(t)] \\ \begin{bmatrix} 0. & 1. \\ 0.500000000000000000 & 1.61514684446562962 \\ 1. & 2.31977874998471156 \\ 1.50000000000000000 & 2.71148236598134229 \\ 2. & 2.48257853876016776 \\ 2.50000000000000000 & 1.81933798345346220 \\ 3. & 1.15156383851474508 \end{bmatrix} \end{bmatrix}$$

Compare the approximations in the second column to the 5 digit representations of the exact solution values displayed below.

```
> Matrix([['t',t/2$t=0..6],[`g(t)`,g(t/2)$t=0..6]]): evalf[5](%);
```

$$\begin{bmatrix} t & 0. & 0.50000 & 1. & 1.5000 & 2. & 2.5000 & 3. \\ g(t) & 1. & 1.6152 & 2.3198 & 2.7115 & 2.4826 & 1.8193 & 1.1516 \end{bmatrix}$$

The dsolve/numeric *procedure*

As the default method, **dsolve/numeric** uses rkf45. The default output is a procedure. Give the procedure a name

```
> y_approx := dsolve( {DE,y(0)=1}, y(t), numeric);
```

$$y\_approx := \mathbf{proc}(x\_rkf45) \ \ldots \ \mathbf{end\ proc}$$

That way, an approximate y value at a specific t value, say $t = a$, can be obtained with an entry of the form

```
y_approx(a)
```

The next entry asks for the approximate y value when $t = 1.5$

```
> y_approx(1.5);
```

$$[t = 1.5, y(t) = 2.71148236598134229]$$

Observe that the output is a list containing equations identifying the t value and the approximate $y(t)$ value.

The entry

```
y_approx(1.5)[2]
```

outputs the second entry in the list.

```
> y_approx(1.5)[2];
```

$$y(t) = 2.71148236598134229$$

And the entry

```
rhs(y_approx)(1.5)[2])
```

outputs the approximate y value when $t = 1.5$.

```
> rhs(y_approx(1.5)[2]);
```

$$2.71148236598134229$$

This suggests that the following entry might output an approximate solution curve.

```
> plot( rhs(y_approx(t)[2]), t=0..10, 0..3);
```

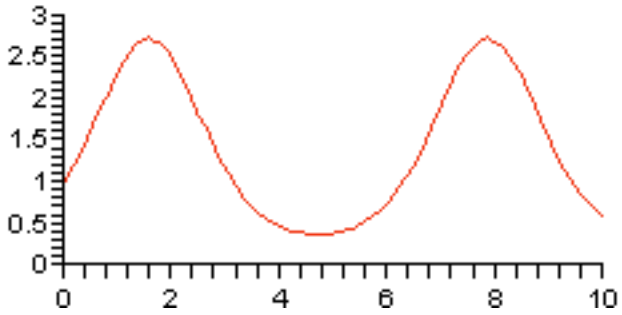
```
Error, invalid input: rhs expects its 1st argument, expr, to be of type
name, but received y_approx(t)[2]
```

Unfortunately, it does not work. However, the problem can be fixed by studying the Error message. The "rhs" procedure expects to get an equation as its input. It does not because the plot procedure tries to evaluate

```
rhs(y_approx(t)[2])
```

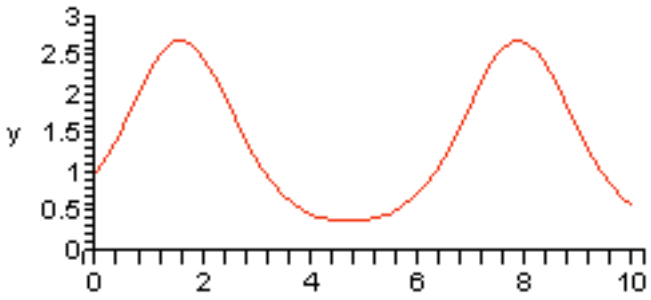
before it substitutes any t values. This problem can be fixed by enclosing rhs(....) in single forward quotes, thereby preventing premature evaluation. See below.

```
> plot( 'rhs(y_approx(t)[2])', t=0..10, 0..3);
```



If this seems like too much trouble (it certainly does to me), then use a special procedure in the plots package called **odeplot**.

```
> plots[odeplot]( y_approx, [t,y(t)], t=0..10, view=0..3);
```



Note that **odeplot** requires the entry [$t, y(t)$] so it knows what points to plot. In addition, the vertical range must be controlled with the equation

```
view=0..3
```

The nice thing about **odeplot** is the fact that if the differential equation is second order, then it will also plot the derivative function and the phase space trajectory.

A preview of second order equations: An aging spring

Equations similar to the following second order differential equation are used as models for an aging spring. See Ledder, Chapter 3, Section 7.

```
> aging_spring := diff(y(t),t,t) + exp(-0.2*t)*y(t) = 0;
```

$$aging\_spring := \left(\frac{d^2}{dt^2} y(t) \right) + \mathbf{e}^{(-0.2\,t)}\, y(t) = 0$$

The exact solution formula for this equation requires special functions that are discussed in Appendix A2. Given an IVP, the **dsolve/numeric** solution works just as well.

The next entry generates the rkf45 solution to the aging_spring equation satisfying the initial conditions

$$y(0) = 1 \text{ and } y'(0) = 0.$$

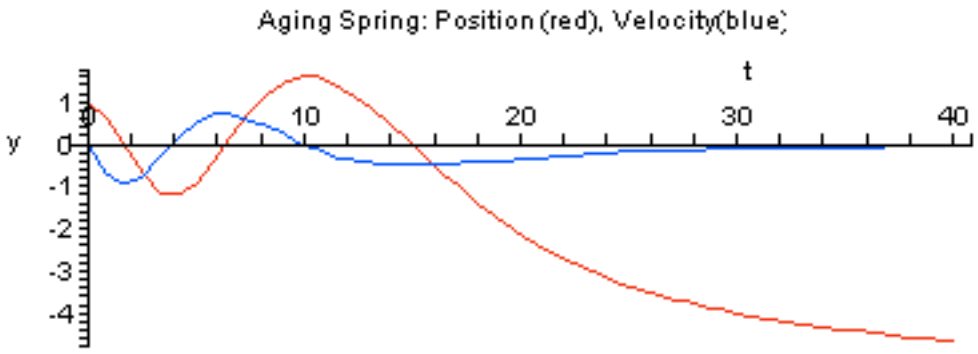
```
> y_approx := dsolve( {aging_spring, y(0)=1, D(y)(0)=0}, y(t), numeric );
```

$$y\_approx := \mathbf{proc}(x\_rkf45) \ \ldots \ \mathbf{end\ proc};$$

The following input (1) loads the plots package, (2) makes a plot named Position of the position function y, (3) makes a plot named Velocity the velocity function y', and (4) uses the display procedure to display them together.

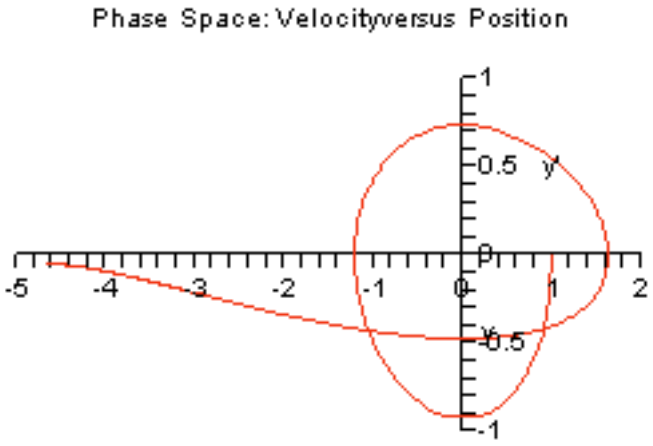
• Colons terminate the Position and Velocity definitions to suppress the ouput.

```
> with(plots):
  Position := odeplot( y_approx, [t,y(t)], t=0..40, color=red):
  Velocity := odeplot( y_approx, [t,diff(y(t),t)], t=0..40, color=blue):
  display( Position, Velocity,
           title="Aging Spring: Position (red), Velocity (blue)");
```



The following plot of velocity versus position is called the phase plane trajectory.

```
> odeplot( y_approx, [y(t),diff(y(t),t)], t=0..40, numpoints=400,
          title="Phase Space: Velocity versus Position",
          view=[-5..2,-1..1]);
```



The optional equation

```
numpoints=400
```

tells odeplot to plot 400 points (the default is 20 points, not enough to plot a smooth trajectory. Note also that the view window for the trajectory is controlled by the optional equation

```
view=[-5..2,-1..1]
```

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.