```
> restart;
```

The Euler and trapezoidal stencils to solve

$$\frac{d}{dx}y(x) = f(x, y(x))$$

The purpose of this worksheet is to derive the three simplest numerical stencils to solve the first order equation $\frac{d}{dx}y(x) = f(x, y(x))$, and study some of their properties. We first write down the ODE using the D notation:

```
> ode := D(y)(x) = f(x,y(x));
```

$$ode := D(y)(x) = f(x, y(x)) \quad (1)$$

In this expression, f is assumed to be a known function of the independant variable x and the function that we are trying to solve for $y(x)$. The simplest numerical stencils to solve this equation will give us an approximation to y at some point $x = X + h$ given some knowledge of y at $x = X$. All of these stencils are based on the Taylor series approximation for $y(x)$ about $x = X$ to linear order:

```
> eq1 := y(x) = series(y(x),x=X,2);
```

$$eq1 := y(x) = y(X) + D(y)(X)\,(x - X) + O\left((x - X)^2\right) \quad (2)$$

Let us define h as the difference between x and X, and then get rid of x in the above expression:

```
> eq2 := h = x - X;
  eq3 := subs(isolate(eq2,x),eq1);
```

$$eq2 := h = x - X$$

$$eq3 := y(h + X) = y(X) + D(y)(X)\,h + O\left(h^2\right) \quad (3)$$

Now, we can remove the first derivative of y by making use of the differential equation:

```
> eq4 := subs(x=X,ode);
  eq5 := subs(eq4,eq3);
```

$$eq4 := D(y)(X) = f(X, y(X))$$

$$eq5 := y(h + X) = y(X) + f(X, y(X))\,h + O\left(h^2\right) \quad (4)$$

The forward Euler algorithm involves discarding the terms of order h^2 and higher in this expression and hence obtain an approximation to $y(X + h)$ in terms of $y(X)$. We can accompish this by converting the above series into a polynomial using the **convert/polynom** command. Also, we can take $X = x_i$ as the i^{th} point on an evenly spaced lattice $x_i = x_0 + ih$, where h is the lattice spacing and i is an integer; it then follows that $X + h = x_{i+1}$. Furthermore, we label the numeric approximation of $y(x)$ at $x = x_i$ as $y_i \approx y(x_i)$. Implementing these steps and notational changes:

```
> eq6 := convert(eq5,polynom);
  eq7 := [y(h+X)=y[i+1],y(X)=y[i],X=x[i]];
  eq8 := subs(eq7,eq6);
```

$$eq6 := y(h + X) = y(X) + f(X, y(X))\,h$$

$$eq7 := \left[y(h + X) = y_{i+1}, y(X) = y_i, X = x_i\right]$$

$$eq8 := y_{i+1} = y_i + f\left(x_i, y_i\right) h \quad (5)$$

The last expression is the forward Euler stencil for solving **ode**. It is called an explicit algorithm because the quantity we wish to calculate y_{i+1} is given explicitly in terms of y_i. The backward Euler stencil is

obtained in a similar fashion, except we identify h, y_i, and y_{i+1} differently:

```
> eq9 := h = X - x;
  eq10 := subs(isolate(eq9,x),eq1);
  eq11 := subs(eq4,eq10);
  eq12 := convert(eq11,polynom);
  eq13 := [y(-h+X)=y[i],y(X)=y[i+1],X=x[i+1]];
  eq14 := subs(eq13,eq12);
```

$$eq9 := h = X - x$$

$$eq10 := y(-h + X) = y(X) - D(y)(X)\,h + O\left((-h)^2\right)$$

$$eq11 := y(-h + X) = y(X) - f(X, y(X))\,h + O\left((-h)^2\right)$$

$$eq12 := y(-h + X) = y(X) - f(X, y(X))\,h$$

$$eq13 := \left[y(-h + X) = y_i, y(X) = y_{i+1}, X = x_{i+1}\right]$$

$$eq14 := y_i = y_{i+1} - f\left(x_{i+1}, y_{i+1}\right) h \qquad \textbf{(6)}$$

This stencil is explicit because it will not in general be possible to alegraically isolate what we want to calculate, i.e. $y_{i+1} \approx y\left(x_{i+1}\right)$, except for very specific forms of f. We can rearrange the stencils to better demonstrate their geometric meaning:

```
> ForwardEuler := collect(expand(isolate(eq8,f(x[i],y[i]))),h);
  BackwardEuler := collect(expand(isolate(eq14,f(x[i+1],y[i+1]))),h);
```

$$ForwardEuler := f\left(x_i, y_i\right) = \frac{y_{i+1} - y_i}{h}$$

$$BackwardEuler := f\left(x_{i+1}, y_{i+1}\right) = \frac{y_{i+1} - y_i}{h} \qquad \textbf{(7)}$$

These forms illustrate that the forward Euler stencil (first equation) is a simple approximation of the first derivative of $y(x)$ in the interval $x \in \left[x_i, x_{i+1}\right]$ using **ode** evaluated at the lefthand side of the interval. Conversely, the backward Euler stencil uses the differential equation to evalulate the derivative at the righthand side of the interval. There is another stencil we can derive that is the average of these two approaches:

```
> Trapezoidal := (ForwardEuler+BackwardEuler)/2;
```

$$Trapezoidal := \frac{1}{2} f\left(x_i, y_i\right) + \frac{1}{2} f\left(x_{i+1}, y_{i+1}\right) = \frac{y_{i+1} - y_i}{h} \qquad \textbf{(8)}$$

This is called the trapezoidal method, and it uses **ode** evaluated at both ends of the interval to approximate $y'(x)$. Like the backward Euler stencil, it represents an implicit scheme. As an example, we can look at what these stencils look like for the special case of $f(x, y) = \lambda y$ where λ is a constant:

```
> f := (x,y) -> lambda*y;
  Stencils := [ForwardEuler,BackwardEuler,Trapezoidal];
  Labels := ["Forward Euler","Backward Euler","Trapezoidal"];
```

$$f := (x, y) \to \lambda\, y$$

$$Stencils := \left[\lambda\, y_i = \frac{y_{i+1} - y_i}{h}, \lambda\, y_{i+1} = \frac{y_{i+1} - y_i}{h}, \frac{1}{2}\lambda\, y_i + \frac{1}{2}\lambda\, y_{i+1} = \frac{y_{i+1} - y_i}{h}\right]$$

$$Labels := [\text{"Forward Euler"}, \text{"Backward Euler"}, \text{"Trapezoidal"}] \qquad \textbf{(9)}$$

For this special case, we see that it is possible to isolate y_{i+1} for each stencil. Let's do this using a loop:

```
> for j from 1 to 3 do:
    Stencils[j] := factor(isolate(Stencils[j],y[i+1])):
    print(Labels[j],Stencils[j]):
  od:
```

$$\text{"Forward Euler"}, y_{i+1} = y_i\,(\lambda h + 1)$$

$$\text{"Backward Euler"}, y_{i+1} = -\frac{y_i}{\lambda h - 1}$$

$$\text{"Trapezoidal"}, y_{i+1} = -\frac{y_i\,(\lambda h + 2)}{\lambda h - 2} \qquad \textbf{(10)}$$

Now, lets go back to the general case by undefining *f*:

```
> f := 'f';
  Stencils := [ForwardEuler,BackwardEuler,Trapezoidal];
```

$$f := f$$

$$Stencils := \left[f(x_i, y_i) = \frac{y_{i+1} - y_i}{h}, f(x_{i+1}, y_{i+1}) = \frac{y_{i+1} - y_i}{h}, \frac{1}{2} f(x_i, y_i) + \frac{1}{2} f(x_{i+1}, y_{i+1}) = \frac{y_{i+1} - y_i}{h}\right] \qquad \textbf{(11)}$$

We now want to determine what the error is in each stencil. To do this, we need to rewrite each of them in the standard form $y_{i+1} - y_i - h\Phi(x_i, x_{i+1}, y_i, y_{i+1}) = 0$:

```
> for j from 1 to 3 do:
    Stencils[j] := Stencils[j]*h + y[i];
    Stencils[j] := (rhs-lhs)(Stencils[j])=0;
    print(StencilLabels[j],Stencils[j]):
  od:
```

$$StencilLabels_1, y_{i+1} - y_i - f(x_i, y_i)\,h = 0$$

$$StencilLabels_2, y_{i+1} - f(x_{i+1}, y_{i+1})\,h - y_i = 0$$

$$StencilLabels_3, y_{i+1} - h\left(\frac{1}{2} f(x_i, y_i) + \frac{1}{2} f(x_{i+1}, y_{i+1})\right) - y_i = 0 \qquad \textbf{(12)}$$

Note that these relations ***define*** our numeric stencils in these sense that they give exact relations between the approximations y_i and y_{i+1}. But if we replace the approximations with the exact values of $y(x)$ they are supposed to represent, the above will represent only approximate equalities. That is, if we make the changes $x_i \to x, x_{i+1} \to x + h, y_i \to y(x), y_{i+1} \to y(x+h)$, the lefthand sides of the above equations will only be approximately equal to zero. We call the magnitude of the discrepancy the "***one step error***" or "***local error***" in the stencil. Hence, the one step error in the various stencils will be:

```
> eq15 := [y[i]=y(x),y[i+1]=y(x+h),x[i]=x,x[i+1]=x+h];
  for j from 1 to 3 do:
    Error[Labels[j]] := subs(eq15,lhs(Stencils[j]));
  od;
```

$$eq15 := [y_i = y(x), y_{i+1} = y(x+h), x_i = x, x_{i+1} = x + h]$$

$$Error_{\text{"Forward Euler"}} := y(x+h) - y(x) - f(x, y(x))\,h$$

$$Error_{\text{"Backward Euler"}} := y(x+h) - f(x+h, y(x+h))\,h - y(x)$$

$$Error_{\text{"Trapezoidal"}} := y(x+h) - h\left(\frac{1}{2}f(x,y(x)) + \frac{1}{2}f(x+h,y(x+h))\right) - y(x) \qquad (13)$$

To estimate the magnitudes of these errors, we expand each of the above expression as a power series about $h = 0$ (since we are implicitly assuming h is a small quantity):

```
> for j from 1 to 3 do:
    Error[Labels[j]] := series(Error[Labels[j]],h,4);
  od;
```

$$Error_{\text{"Forward Euler"}} := (-f(x,y(x)) + D(y)(x))\,h + \frac{1}{2}D^{(2)}(y)(x)\,h^2 + \frac{1}{6}D^{(3)}(y)(x)\,h^3 + O(h^4)$$

$$Error_{\text{"Backward Euler"}} := (-f(x,y(x)) + D(y)(x))\,h + \left(\frac{1}{2}D^{(2)}(y)(x) - D_1(f)(x, y(x)) - D_2(f)(x,y(x))\,D(y)(x)\right)h^2 + \left(\frac{1}{6}D^{(3)}(y)(x) - \frac{1}{2}D_{1,1}(f)(x,y(x)) - D_{1,2}(f)(x,y(x))\,D(y)(x) - \frac{1}{2}D_{2,2}(f)(x,y(x))\,D(y)(x)^2 - \frac{1}{2}D_2(f)(x, y(x))\,D^{(2)}(y)(x)\right)h^3 + O(h^4)$$

$$Error_{\text{"Trapezoidal"}} := (-f(x,y(x)) + D(y)(x))\,h + \left(\frac{1}{2}D^{(2)}(y)(x) - \frac{1}{2}D_1(f)(x, y(x)) - \frac{1}{2}D_2(f)(x,y(x))\,D(y)(x)\right)h^2 + \left(\frac{1}{6}D^{(3)}(y)(x) - \frac{1}{4}D_{1,1}(f)(x, y(x)) - \frac{1}{2}D_{1,2}(f)(x,y(x))\,D(y)(x) - \frac{1}{4}D_{2,2}(f)(x,y(x))\,D(y)(x)^2 - \frac{1}{4}D_2(f)(x,y(x))\,D^{(2)}(y)(x)\right)h^3 + O(h^4) \qquad (14)$$

Notice that first, second and third derivatives of y appear explicitly in these expressions. The first derivative terms can be removed by making use of **ode**. The second derivative can also be removed by examining the derivative of **ode**:

```
> eq16 := diff(ode,x);
  eq17 := convert(ode,diff);
  eq18 := subs(eq17,eq16);
```

$$eq16 := D^{(2)}(y)(x) = D_1(f)(x,y(x)) + D_2(f)(x,y(x))\left(\frac{d}{dx}y(x)\right)$$

$$eq17 := \frac{d}{dx}y(x) = f(x,y(x))$$

$$eq18 := D^{(2)}(y)(x) = D_1(f)(x,y(x)) + D_2(f)(x,y(x))\,f(x,y(x)) \qquad (15)$$

Similarly, the third derivative can be removed by differentiating **eq18**:

```
> eq19 := subs(eq17,diff(eq18,x));
```

$$eq19 := D^{(3)}(y)(x) = D_{1,1}(f)(x,y(x)) + D_{1,2}(f)(x,y(x))\,f(x,y(x)) \qquad (16)$$

$$+\left(D_{1,2}(f)\,(x,y(x))+D_{2,2}(f)\,(x,y(x))f(x,y(x))\right)f(x,y(x))+D_2(f)\,(x,$$
$$y(x))\left(D_1(f)\,(x,y(x))+D_2(f)\,(x,y(x))f(x,y(x))\right)$$

We now substitute our formulae for the derivatives of y into the error expressions:

```
> for j from 1 to 3 do:
    Error[Labels[j]] := subs(eq18,eq19,ode,Error[Labels[j]]);
  od;
```

$$Error_{\text{"Forward Euler"}} := \left(\frac{1}{2}D_1(f)\,(x,y(x))+\frac{1}{2}D_2(f)\,(x,y(x))f(x,y(x))\right)h^2$$
$$+\left(\frac{1}{6}D_{1,1}(f)\,(x,y(x))+\frac{1}{6}D_{1,2}(f)\,(x,y(x))f(x,y(x))+\frac{1}{6}\left(D_{1,2}(f)\,(x,\right.\right.$$
$$\left.y(x))+D_{2,2}(f)\,(x,y(x))f(x,y(x))\right)f(x,y(x))+\frac{1}{6}D_2(f)\,(x,$$
$$\left.\left.y(x))\left(D_1(f)\,(x,y(x))+D_2(f)\,(x,y(x))f(x,y(x))\right)\right)h^3+O\left(h^4\right)$$

$$Error_{\text{"Backward Euler"}} := \left(-\frac{1}{2}D_1(f)\,(x,y(x))-\frac{1}{2}D_2(f)\,(x,y(x))f(x,y(x))\right)h^2+\left(\right.$$
$$-\frac{1}{3}D_{1,1}(f)\,(x,y(x))-\frac{5}{6}D_{1,2}(f)\,(x,y(x))f(x,y(x))+\frac{1}{6}\left(D_{1,2}(f)\,(x,\right.$$
$$\left.y(x))+D_{2,2}(f)\,(x,y(x))f(x,y(x))\right)f(x,y(x))-\frac{1}{3}D_2(f)\,(x,$$
$$y(x))\left(D_1(f)\,(x,y(x))+D_2(f)\,(x,y(x))f(x,y(x))\right)-\frac{1}{2}D_{2,2}(f)\,(x,$$
$$\left.y(x))f(x,y(x))^2\right)h^3+O\left(h^4\right)$$

$$Error_{\text{"Trapezoidal"}} := \left(-\frac{1}{12}D_{1,1}(f)\,(x,y(x))-\frac{1}{3}D_{1,2}(f)\,(x,y(x))f(x,y(x))\right. \tag{17}$$
$$+\frac{1}{6}\left(D_{1,2}(f)\,(x,y(x))+D_{2,2}(f)\,(x,y(x))f(x,y(x))\right)f(x,y(x))$$
$$-\frac{1}{12}D_2(f)\,(x,y(x))\left(D_1(f)\,(x,y(x))+D_2(f)\,(x,y(x))f(x,y(x))\right)$$
$$\left.-\frac{1}{4}D_{2,2}(f)\,(x,y(x))f(x,y(x))^2\right)h^3+O\left(h^4\right)$$

We see that the local error for the forward and backward schemes is $O\left(h^2\right)$. Futhermore, the leading order terms for those stencils is the negative of the other one. Since the trapezoidal scheme is the average of the forward and backward algorithms, this explains why the leading order error for the trapeziodal scheme is $O\left(h^3\right)$. In other words, the trapezoidal scheme is more accurate than the other two. Note that if all we are after is the magnitude of the leading order terms in the errors, we can just expand the above in a low order series:

```
> for j from 1 to 3 do:
    Error[Labels[j]] := series(Error[Labels[j]],h,0);
  od;
```

$$Error_{\text{"Forward Euler"}} := O(h^2)$$

$$Error_{\text{"Backward Euler"}} := O(h^2)$$

$$Error_{\text{"Trapezoidal"}} := O(h^3) \quad \textbf{(18)}$$

We now turn our attention to actual numerical algorithms employing these stencils. Since the forward Euler scheme is explicit, it is particularly easy to develop some code for it that works for arbitrary functions f. It is useful to rewrite the stencil with y_{i+1} isolated on the LHS:

```
> eq20 := isolate(Stencils[1],y[i+1]);
```

$$eq20 := y_{i+1} = y_i + f(x_i, y_i)\, h \quad \textbf{(19)}$$

Now we can transcribe this as a mapping that takes the stepsize and the old values of y_i and x_i and returns the new value y_{i+1}:

```
> y_new := (h,x_old,y_old) -> y_old + f(x_old,y_old)*h;
```

$$y\_new := (h, x\_old, y\_old) \rightarrow y\_old + f(x\_old, y\_old)\, h \quad \textbf{(20)}$$

We could have equivalently accomplished this by using the **unapply** command, which converts expression into mappings without having to re-write them manually:

```
> y_new := unapply(rhs(eq20),h,x[i],y[i]);
```

$$y\_new := (h, y2, y3) \rightarrow y3 + f(y2, y3)\, h \quad \textbf{(21)}$$

Here is a procedure **EulerSol** that calculates the numerical solution of **ode** in the interval $x \in [0, 1]$ once the form of $f(x, y)$ has been fixed. It's arguments are initial data in the form $y(0) = y0$ and the number of steps N. The output is a list of points, which we compare in the plot to the output generated by **dsolve/numeric** for the same problem.

```
> f := (x,y) -> -y^2+2*x;
  ode;
  y0 := 2;
  N := 15;

  EulerSol := proc(y0,N)
       local h, x, y, i:

       h := evalf(1/N):
       x[0] := 0:
       y[0] := y0:
       for i from 1 to N do:
          x[i] := x[i-1] + h:
          y[i] := y_new(h,x[i-1],y[i-1]):
       od:
       [seq([x[i],y[i]],i=0..N)]:

  end proc:

  data := EulerSol(y0,N);

  dsolveSol := rhs(dsolve({ode,y(0)=y0},numeric,output=listprocedure)
  [2]);

  plot([data,dsolveSol(x)],x=0..1,axes=boxed,legend=[`Forward Euler`,
  `dsolve/numeric`]);
```

$$f := (x, y) \rightarrow -y^2 + 2x$$

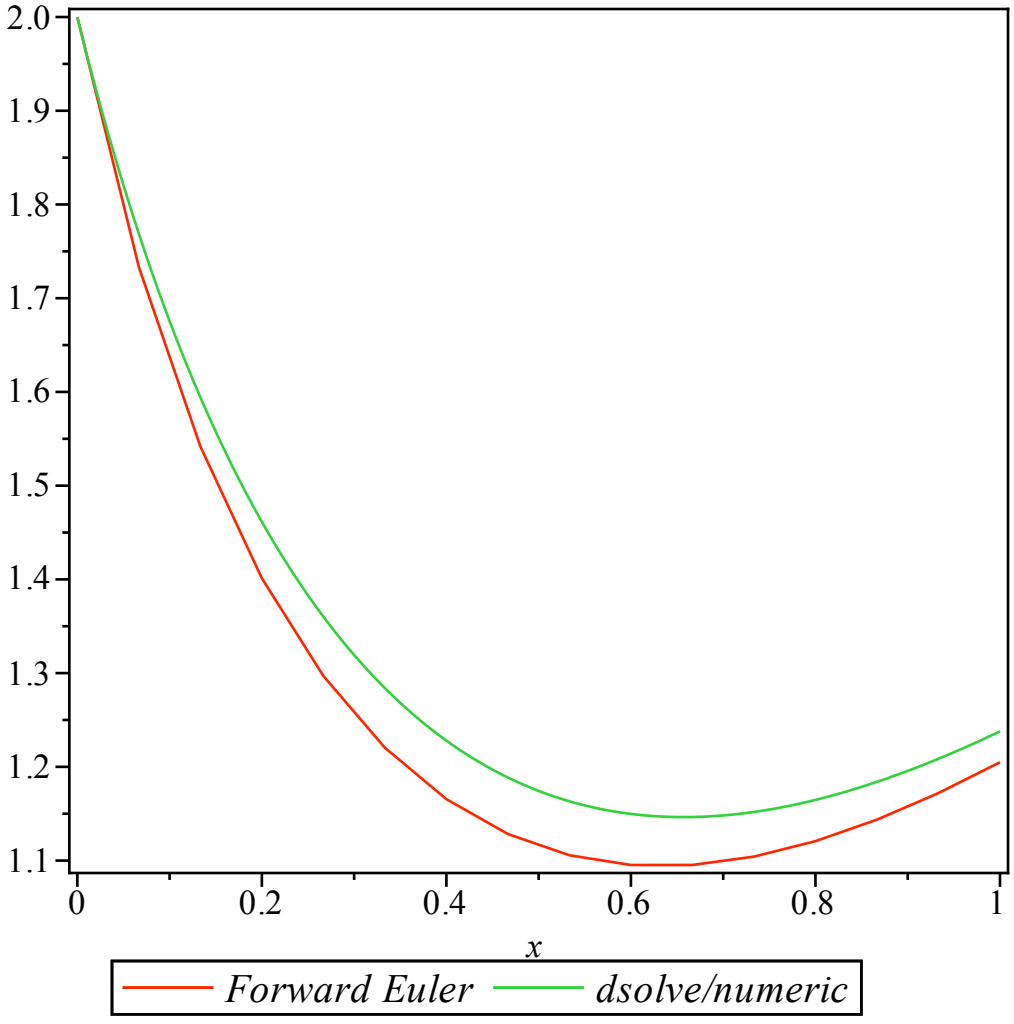
$$D(y)(x) = -y(x)^2 + 2x$$

$$y0 := 2$$

$$N := 15$$

$$data := [[0, 2], [0.06666666667, 1.733333333], [0.1333333333, 1.541925926], [0.2000000000, 1.401201333], [0.2666666667, 1.296976988], [0.3333333334, 1.220389256], [0.4000000001, 1.165543705], [0.4666666668, 1.128310896], [0.5333333335, 1.105660753], [0.6000000002, 1.095272817], [0.6666666669, 1.095297981], [0.7333333336, 1.104208359], [0.8000000003, 1.120701063], [0.8666666670, 1.143636338], [0.9333333337, 1.171998289], [1.000000000, 1.204870734]]$$

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



Of course, this plot is really comparing the results of two numeric approximations to the true solution of the problem. It is also useful to compare numeric answers to analytic results (if available). For the above choice of $f(x, y)$ there is an analytic solution in terms of Airy functions:

```
> analytic_sol := dsolve({ode,y(0)=y0}):
  analytic_sol := rhs(analytic_sol);
```

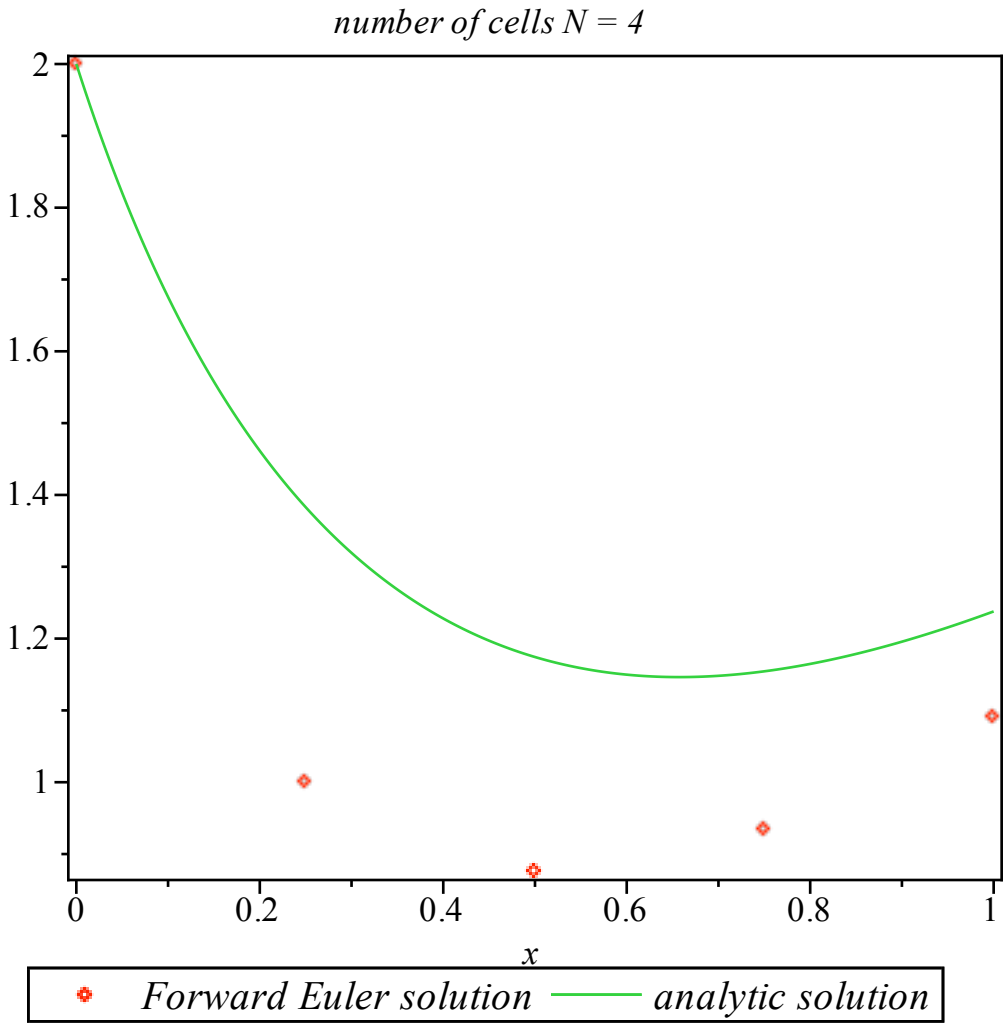
(22)

analytic_sol := **(22)**

$$2^{1/3}\left(\frac{\left(-4\pi 3^{5/6}+3\,2^{1/3}\Gamma\left(\frac{2}{3}\right)^2 3^{2/3}\right)\mathrm{AiryAi}\left(1,2^{1/3}x\right)}{4\pi 3^{1/3}+3\,2^{1/3}\Gamma\left(\frac{2}{3}\right)^2 3^{1/6}}+\mathrm{AiryBi}\left(1,2^{1/3}x\right)\right)$$
$$\frac{\left(-4\pi 3^{5/6}+3\,2^{1/3}\Gamma\left(\frac{2}{3}\right)^2 3^{2/3}\right)\mathrm{AiryAi}\left(2^{1/3}x\right)}{4\pi 3^{1/3}+3\,2^{1/3}\Gamma\left(\frac{2}{3}\right)^2 3^{1/6}}+\mathrm{AiryBi}\left(2^{1/3}x\right)$$

Here is some code that generates a movie comparing how the numerical solution converges to the analytic solution as the number of cells is increased:

```
> for j from 1 to 25 do:
    N := 4*j:
    p[j] := plot([EulerSol(y0,N),analytic_sol],x=0..1,title = cat
  (`number of cells N = `,N),axes=boxed,legend=[`Forward Euler
  solution`,`analytic solution`],style=[point,line]);
  od:
  plots[display](convert(p,list),insequence=true);
```



We would also like to examine the performance of the backward and trapezoidal stencils, so let's make a simple choice of $f(x,y)$ that admits an analytic solution allows each stencil to be solve for y_{i+1} explicitly:

```
> f := (x,y) -> lambda*y;
  y0 := 'y0';
  ode;
  analytic_sol := rhs(dsolve({ode,y(0)=y0}));

  for j from 1 to 3 do:
      Stencils[j] := isolate(Stencils[j],y[i+1]);
      print(Labels[j],Stencils[j]);
  od:
```

$$f := (x,y) \rightarrow \lambda\, y$$

$$y0 := y0$$

$$\mathrm{D}(y)(x) = \lambda\, y(x)$$

$$analytic\_sol := y0\, e^{\lambda x}$$

$$\text{"Forward Euler"}, y_{i+1} = \lambda\, y_i\, h + y_i$$

$$\text{"Backward Euler"}, y_{i+1} = \frac{y_i}{1 - \lambda h}$$

$$\text{"Trapezoidal"}, y_{i+1} = \frac{y_i + \frac{1}{2}\lambda y_i h}{1 - \frac{1}{2}\lambda h} \tag{23}$$

As for the above code, it will be useful to realize each stencil as a mapping with arguments h, λ and y_i. We can do this using a loop and the **unapply** command:

```
> for j from 1 to 3 do:
      stencil[j] := unapply(rhs(Stencils[j]),h,lambda,y[i]);
  od;
```

$$stencil_1 := (h, \lambda, y3) \rightarrow \lambda\, y3\, h + y3$$

$$stencil_2 := (h, \lambda, y3) \rightarrow \frac{y3}{1 - \lambda h}$$

$$stencil_3 := (h, \lambda, y3) \rightarrow \frac{y3 + \frac{1}{2}\lambda\, y3\, h}{1 - \frac{1}{2}\lambda h} \tag{24}$$

Conversely, we could have accomplished the same thing in one line by using the **map** command, which applies the same mapping onto each element of a list (in this case we are applying operations onto each element of **Stencils** to generate a new list **stencil**):

```
> stencil := 'stencil':
  stencil := map(u->unapply(rhs(u),h,lambda,y[i]),Stencils);
```

$$stencil := \left[(h, \lambda, y3) \rightarrow \lambda\, y3\, h + y3,\ (h, \lambda, y3) \rightarrow \frac{y3}{1 - \lambda h},\ (h, \lambda, y3) \rightarrow \frac{y3 + \frac{1}{2}\lambda\, y3\, h}{1 - \frac{1}{2}\lambda h} \right] \tag{25}$$

It is interesting to note that even though each of the stencils give different expressions for y_{i+1}, they actually agree with one another to order h^2. To see this, let's perform some Taylor series expansions:

```
> for j from 1 to 3 do:
      Labels[j],y[i+1] = series(stencil[j](h,lambda,y[i]),h,2);
  od;
```

$$\text{"Forward Euler"}, y_{i+1} = y_i + \lambda\, y_i\, h$$

$$\text{"Backward Euler"}, y_{i+1} = y_i + \lambda\, y_i\, h + O(h^2)$$

$$\text{"Trapezoidal"}, y_{i+1} = y_i + \lambda\, y_i\, h + O(h^2) \tag{26}$$

Now, here is a procedure **EulerSol2** that calculates a numeric solution for $y(x)$ for $x \in [0, 1]$ using N cells and assuming $y(0) = y0$ and a given value of λ. The value of **choice** dictates which stencil is used: 1 for forward Euler, 2 for backward Euler, and 3 for trapezoidal.

```
> EulerSol2 := proc(y0,N,lambda,choice)
      local h, x, y, i:
```

```
        h := evalf(1/N):
        x[0] := 0:
        y[0] := y0:
        for i from 1 to N do:
           x[i] := x[i-1] + h:
           y[i] := stencil[choice](h,lambda,y[i-1]):
        od:
        [seq([x[i],y[i]],i=0..N)]:

  end proc:
```

Here is a plot of the numeric output for each stencil compared to the actually solution. (Because we are going to plot 4 curves on our graph, I needed to augment **Labels** into **NewLabels** in order to gereate the legend.) It seems as if the trapezoidal method performs much better than the other two:

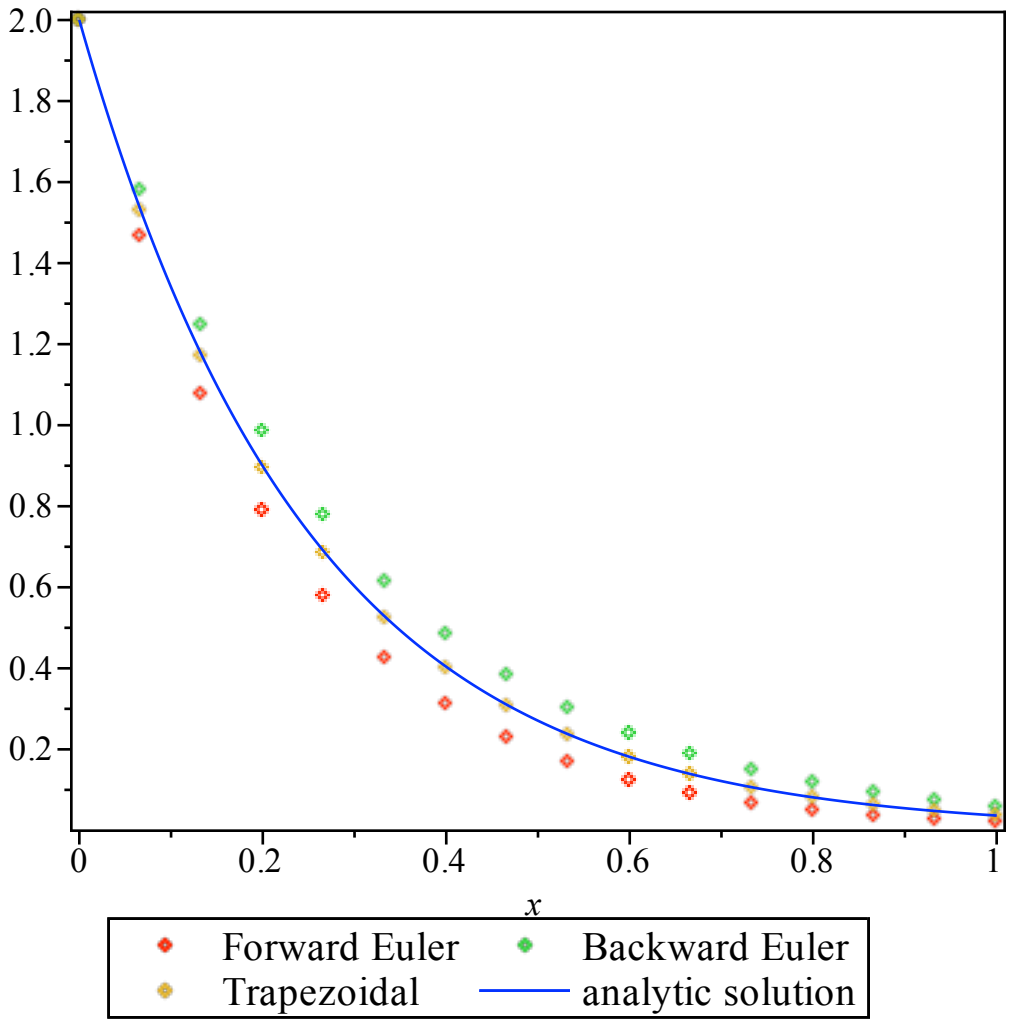
```
> N := 15;
  y0 := 2;
  lambda := -4;
  NewLabels := [op(Labels),"analytic solution"];

  plot([seq(EulerSol2(y0,N,lambda,j),j=1..3),analytic_sol],x=0..1,
  legend=NewLabels,axes=boxed,style=[point$3,line]);
```

$$N := 15$$

$$y0 := 2$$

$$\lambda := -4$$

$$NewLabels := [\text{"Forward Euler"}, \text{"Backward Euler"}, \text{"Trapezoidal"}, \text{"analytic solution"}]$$



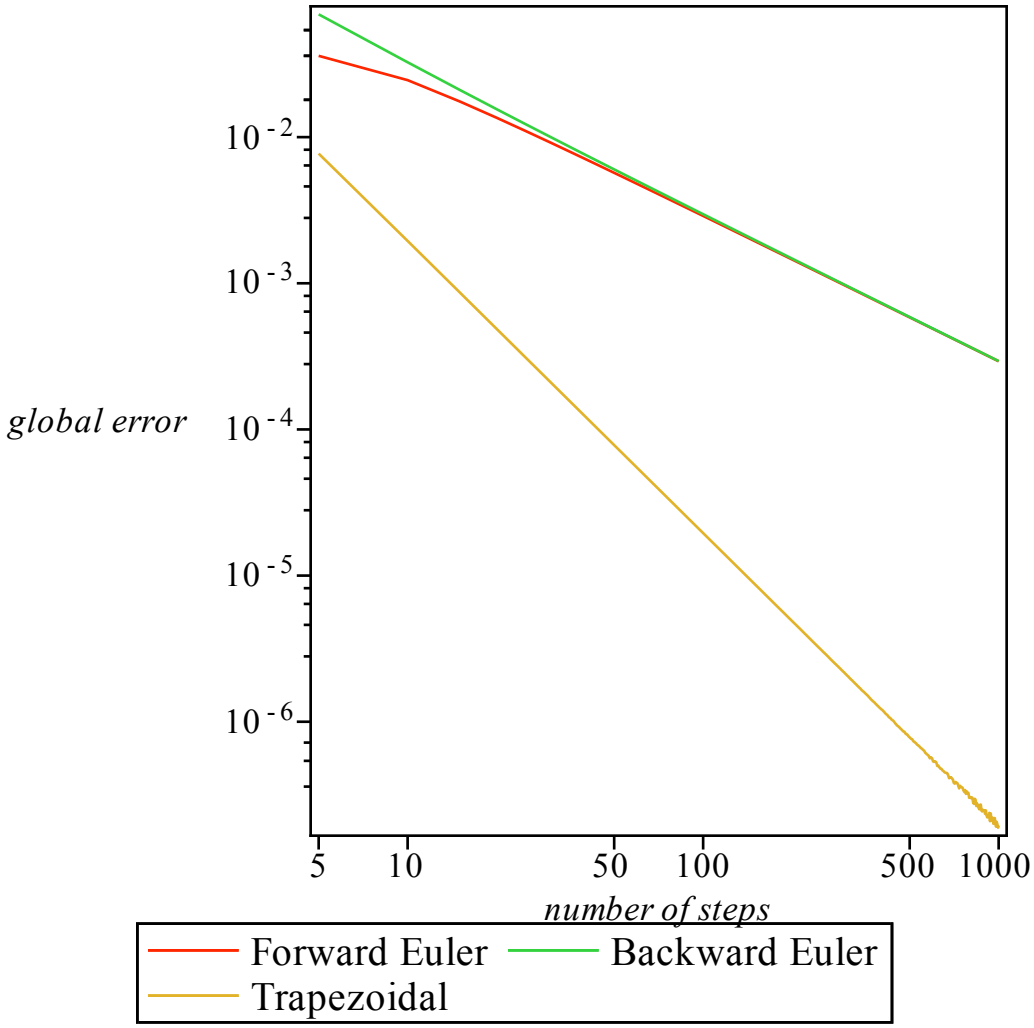
We can quantify exactly how well the various stencils are doing by comparing the numeric and actual values for y at some fixed value of x, say $x = 1$. We call this the global error in the numeric solution at $x = 1$, which we denote by ϵ:

```
> epsilon := proc(y0,N,lambda,choice)
        local numerical, actual:

        numerical := EulerSol2(y0,N,lambda,choice)[N+1][2]:
        actual := y0*exp(lambda):
        evalf(abs(numerical - actual)):

  end proc:
```

In this procedure, note that the `[N+1][2]` suffix after `EulerSol2(y0,N,lambda,choice)` has the effect of picking out the last element of the list (which is itself a list of 2 quantities), and then picking out the second element of that sub-list (which is the numerical answer for $y(1)$). We now generate a log-log plot of the global errors for each stencil as a function of N:

```
> data := 'data':

  for j from 1 to 3 do:
      data[j] := [seq([N,epsilon(y0,N,lambda,j)],N=5..1000,5)];
  od:
```

```
plots[loglogplot](convert(data,list),axes=boxed,labels=[`number of
steps`,`global error`],legend=Labels);
```



For $N \gtrsim 100$, the error curves look linear on this log-log plot. This implies that above some threshold number of steps, the errors obey a approximate power law $\epsilon \approx aN^b$ where a and b are constants. We can determine the values of these parameters by fitting a power law to our error data using **Statistics/PowerFit**. We first need to regenerate our data for $N \geq 100$; i.e., the range for which we believe a power law ought to be valid:

```
> data := 'data':

  for j from 1 to 3 do:
      data[j] := [seq([N,epsilon(y0,N,lambda,j)],N=100..1000,5)];
  od:
```

We then use the **Statistics[PowerFit]** command to perform the fit. Note that this function requires the input to be a matrix, which is why we use the **convert/Matrix** structure. The raw output of the fitting algorithms is shown for each stencil, as well as the value of b in the power-law $\epsilon \approx aN^b$.

```
> for j from 1 to 3 do:
      fit[j] := Statistics[PowerFit](convert(data[j],Matrix),_N,
  output=[leastsquaresfunction,parametervalues]):
      print(Labels[j],fit[j],b=fit[j][2][2]);
  od:
```

$$\text{"Forward Euler"}, \left[\frac{0.284665691523995}{\_N^{0.995851786083571}}, \begin{bmatrix} -1.25643979942728 \\ -0.995851786083571 \end{bmatrix}\right], b = -0.995851786083571$$

$$\text{"Backward Euler"}, \left[\frac{0.301341961047440}{\_N^{1.00397729890028}}, \begin{bmatrix} -1.19950957587335 \\ -1.00397729890028 \end{bmatrix}\right], b = -1.00397729890028$$

$$\text{"Trapezoidal"}, \left[\frac{0.196182279176729}{\_N^{2.00065716714713}}, \begin{bmatrix} -1.62871105613015 \\ -2.00065716714713 \end{bmatrix}\right], b = -2.00065716714713 \qquad \textbf{(27)}$$

We see that for asyptotically large numbers of steps $N \gg 1$, the global errors are $\mathrm{O}\left(N^{-1}\right) = \mathrm{O}(h)$ for the forward and backward Euler stencils, and $\mathrm{O}\left(N^{-2}\right) = \mathrm{O}\left(h^2\right)$ for the trapezoidal method. This confirms the general expectation that for a stencil with one step error $\mathrm{O}\left(h^p\right)$, we expect the global error to obey

$$\begin{aligned} &\text{global error } \sim \text{ (number of steps)} \times \text{(one-step error for each step)} \\ &= N \times \mathrm{O}\left(h^p\right) = \mathrm{O}\left(h^{-1}\right) \times \mathrm{O}\left(h^p\right) = \mathrm{O}\left(h^{p-1}\right). \end{aligned}$$

[Recall that we showed in **(18)** above that $p = 2$ for forward and backward Euler, and $p = 3$ for the trapezoidal algoirthm.]

NOTE: the above syntax for `Statistics[PowerFit]` only works in Maple 15. For previous versions, you need to do this:

```
> for j from 1 to 3 do:
      M := convert(data[j],Matrix):
      X := LinearAlgebra[Column](M,1):
      Y := LinearAlgebra[Column](M,2):
      fit[j] := Statistics[PowerFit](X,Y,_N,output=
  [leastsquaresfunction,parametervalues]):
      print(Labels[j],fit[j],b=fit[j][2][2]);
  od:
```

$$\text{"Forward Euler"}, \left[\frac{0.284665691523995}{\_N^{0.995851786083571}}, \begin{bmatrix} -1.25643979942728 \\ -0.995851786083571 \end{bmatrix}\right], b = -0.995851786083571$$

$$\text{"Backward Euler"}, \left[\frac{0.301341961047440}{\_N^{1.00397729890028}}, \begin{bmatrix} -1.19950957587335 \\ -1.00397729890028 \end{bmatrix}\right], b = -1.00397729890028$$

$$\text{"Trapezoidal"}, \left[\frac{0.196182279176729}{\_N^{2.00065716714713}}, \begin{bmatrix} -1.62871105613015 \\ -2.00065716714713 \end{bmatrix}\right], b = -2.00065716714713 \qquad \textbf{(28)}$$