Math 2280-1
Numerical solution to first order differential equations (sections 2.4-2.6 in book)
January 24 2007

Here are sample implementations in Maple of several numerical methods for solving first order differential equations. These should be helpful for the computing project. The same methods are used for solving systems of first order DEs (and thus nth-order DEs as well). We apply Euler's method, improved Euler's method and Runge-Kutta to one of the simplest DEs: dy/dx=y, y(0)=1, which has the solution y=exp(x). The interval of approximation we choose is [0,1].

Euler's method

```
> restart:       # Clear Maple's internal memory
> x0:=0; xn:=1.; # first and last points in the interval
  y0:=1;         # initial condition
  n:=5;          # number of steps
  h:=(xn-x0)/n;  # step size
```

$$x0 := 0$$
$$xn := 1.$$
$$y0 := 1$$
$$n := 5$$
$$h := 0.2000000000 \quad (1)$$

```
> f:=(x,y)->y;   # slope function (rhs in DE dy/dx = f(x,y), here
  f(x,y)=y)
```

$$f := (x, y) \rightarrow y \quad (2)$$

```
> x:=x0; y:=y0;  # initialize x,y
```

$$x := 0$$
$$y := 1 \quad (3)$$

```
> for i from 1 to n do
    k:= f(x,y):  # current slope
    y:= y + h*k: # new y value (Euler's method)
    x:= x+h:     # increase x
    print(x,y,exp(x)); # display current values and compare to
  true sol.
  od: # end for i loop
>
```

```
0.2000000000, 1.200000000, 1.221402758
0.4000000000, 1.440000000, 1.491824698
0.6000000000, 1.728000000, 1.822118800
0.8000000000, 2.073600000, 2.225540928
 1.000000000, 2.488320000, 2.718281828          (4)
```

To plot the true solution (exp(x)) against the computed one, we would have needed to store the computed values. Here is a modification of the above code to just do that.

```
> with(plots):          # load plotting package
  with(LinearAlgebra): # load linear algebra package
> x0:=0; xn:=1.;       # first and last points in the interval
  y0:=1;               # initial condition
  n:=5;                # number of steps
  h:=(xn-x0)/n;        # step size
```

$$x0 := 0$$
$$xn := 1.$$
$$y0 := 1$$
$$n := 5$$
$$h := 0.2000000000 \quad (5)$$

```
> f:=(x,y)->y;         # slope function (rhs in DE dy/dx = f(x,y),
  here f(x,y)=y)
```

$$f := (x, y) \rightarrow y \quad (6)$$

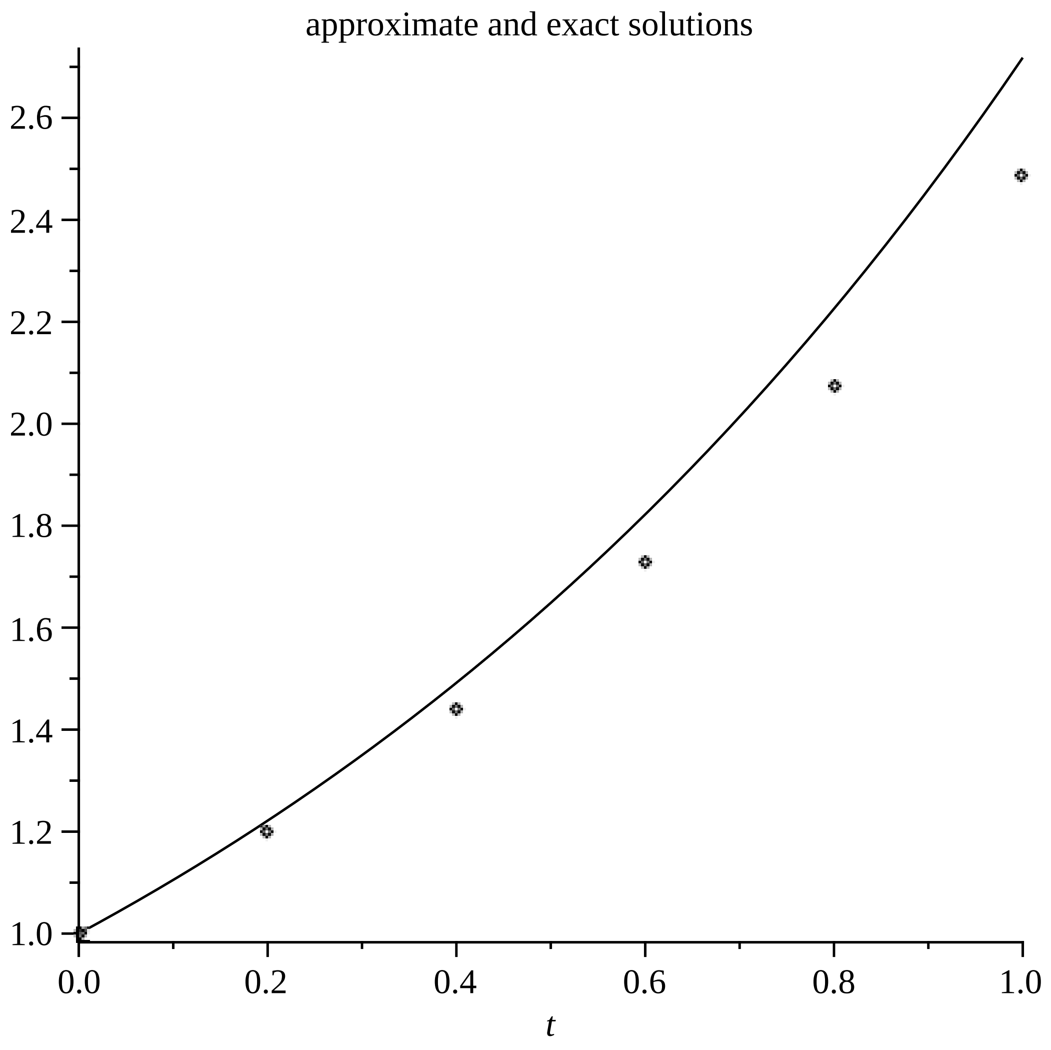
```
> xvals:=Vector(n+1); yvals:=Vector(n+1);
  xvals[1]:=x0; yvals[1]:=y0;
```

$$xvals := \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$yvals := \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$xvals_1 := 0$$
$$yvals_1 := 1 \quad (7)$$

```
> for i from 1 to n do
    x:=xvals[i]: y:=yvals[i]:
    k:= f(x,y):           # current slope
    yvals[i+1]:= y + h*k: # new y value (Euler’s method)
    xvals[i+1]:= x+h:     # increase x
  od: # end for i loop
> calcsol:=pointplot(Matrix([xvals,yvals])):
> exactsol:=plot(exp(t),t=0..1,color=black):
> display({calcsol,exactsol},title="approximate and exact
  solutions");
```



Decreasing the step size h should give, in exact arithmetic (no rounding error) a better approximation. However there are two things to keep in mind:

• Computers use floating point arithmetic, which means that all operations are rounded to a certain number of digits and thus we are introducing rounding error to each step. The smaller h is the more rounding error becomes a problem. By default Maple uses for floating point numbers the most common storage format dubbed ‘‘double precision’’. One can expect around 16 digits of accuracy (in C this data type is ‘‘double’’, in Fortran ‘‘double precision’’ or simply real*8). Here is the rounding error caught in fraganti, in exact arithmetic the result of the following expression should be 1e-16 (which means `10^-16`) but Maple gives 0!

```
> 1+1e-16-1;
                                 0.                                  (8)
```

- The smaller the step size the more steps we do, so the longer it takes to compute our approximation.

Here is the above Euler example with n=100 (instead of 5). The code is adapted so that we print the values every ten steps:

```
> x0:=0; xn:=1.; # first and last points in the interval
  y0:=1; # initial condition
  n:=100; # number of steps
  h:=(xn-x0)/n; # step size
```

$$x0 := 0$$
$$xn := 1.$$
$$y0 := 1$$
$$n := 100$$
$$h := 0.01000000000 \qquad (9)$$

```
> f:=(x,y)->y; # slope function (rhs in DE dy/dx = f(x,y), here f
  (x,y)=y)
```

$$f := (x, y) \rightarrow y \qquad (10)$$

```
> xvals:=Vector(n+1); yvals:=Vector(n+1);
  xvals[1]:=x0; yvals[1]:=y0;
```

$$xvals := \begin{bmatrix} \text{1 .. 101 } Vector_{column} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

$$yvals := \begin{bmatrix} \text{1 .. 101 } Vector_{column} \\ \text{Data Type: anything} \\ \text{Storage: rectangular} \\ \text{Order: Fortran\_order} \end{bmatrix}$$

$$xvals_1 := 0$$
$$yvals_1 := 1 \qquad (11)$$

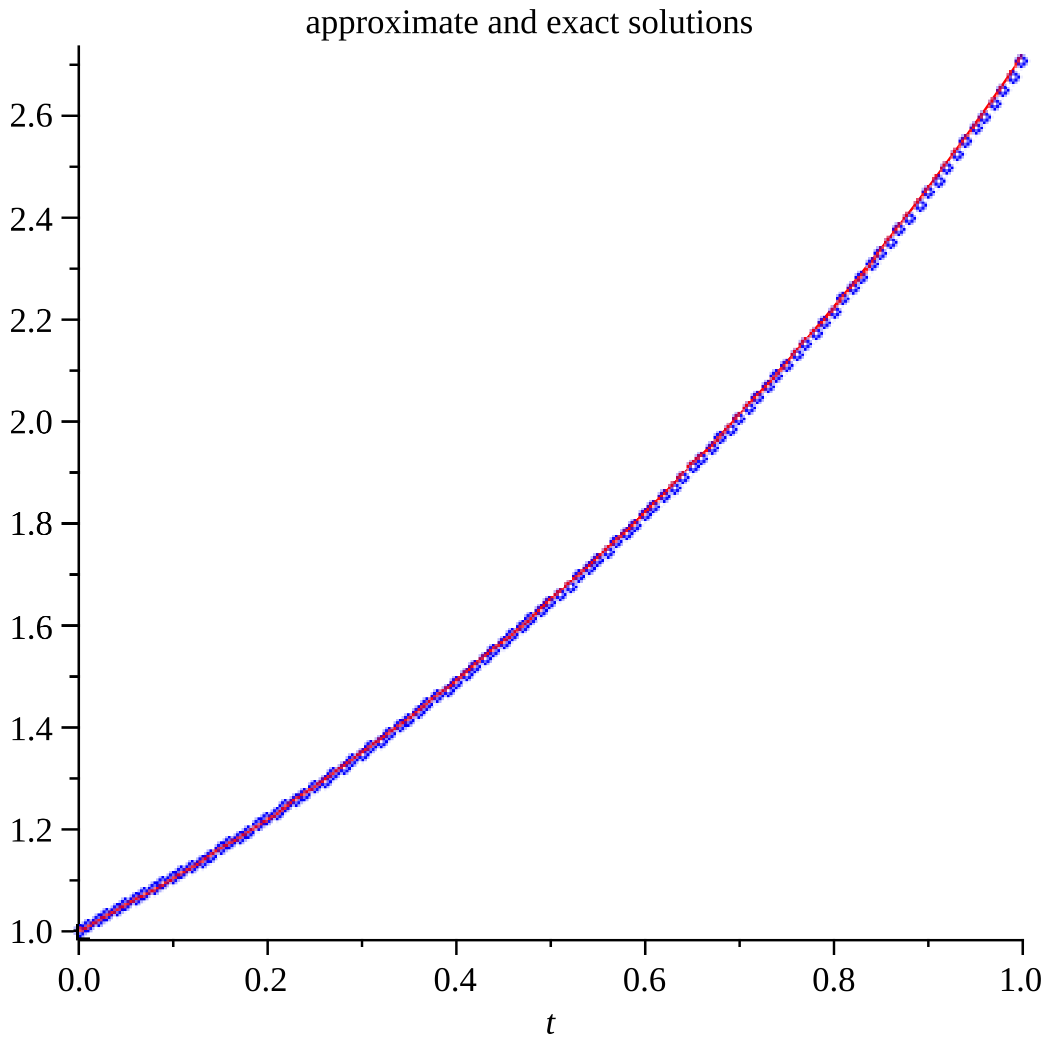
```
> for i from 1 to n do
    x:=xvals[i]: y:=yvals[i]:
    k:= f(x,y):  # current slope
    yvals[i+1]:= y + h*k: # new y value (Euler's method)
    xvals[i+1]:= x+h:     # increase x
    if frac(i/10)=0 then
     print(xvals[i+1],yvals[i+1],exp(xvals[i+1]));
    fi:
  od: # end for i loop
```

```
0.1000000000, 1.104622126, 1.105170918
0.2000000000, 1.220190040, 1.221402758
```

```
0.3000000000, 1.347848915, 1.349858808
0.4000000000, 1.488863734, 1.491824698
0.5000000000, 1.644631822, 1.648721271
0.6000000000, 1.816696698, 1.822118800
0.7000000000, 2.006763369, 2.013752707
0.8000000000, 2.216715219, 2.225540928
0.9000000000, 2.448632677, 2.459603111
 1.000000000, 2.704813833, 2.718281828                (12)
```

With the results above we get TWO digits of e (instead of only one)

```
> calcsol:=pointplot(Matrix([xvals,yvals]),color=blue):
> exactsol:=plot(exp(t),t=0..1,color=red):
> display({calcsol,exactsol},title="approximate and exact
  solutions");
```



Improved Euler’s method

uses a better approximation of the slope, based on the trapezoidal rule, see explanation in class or section 2.5.

```
> x0:=0; xn:=1.; # first and last points in the interval
  y0:=1; # initial condition
  n:=5; # number of steps
  h:=(xn-x0)/n; # step size
```

$$x0 := 0$$
$$xn := 1.$$
$$y0 := 1$$
$$n := 5$$
$$h := 0.2000000000 \qquad (13)$$

```
> f:=(x,y)->y; # slope function (rhs in DE dy/dx = f(x,y), here f
  (x,y)=y)
```

$$f := (x, y) \to y \qquad (14)$$

```
> x:=x0; y:=y0;
```

$$x := 0$$
$$y := 1 \qquad (15)$$

```
> for i from 1 to n do
    k1:= f(x,y):          # left hand slope
    k2:= f(x+h,y+h*k1): # approximation to right hand slope
    k:=(k1+k2)/2:         # averaged slope
    y:= y + h*k:          # improved Euler update
    x:= x+h:              # increase x
    print(x,y,exp(x));  # display current values and compare to
  true sol.
  od: # end for i loop
```

```
0.2000000000, 1.220000000, 1.221402758
0.4000000000, 1.488400000, 1.491824698
0.6000000000, 1.815848000, 1.822118800
0.8000000000, 2.215334560, 2.225540928
 1.000000000, 2.702708163, 2.718281828                (16)
```

With five points we get a result comparable to classical Euler with 100 points!

Runge Kutta

This method (or rather the particular version we use here) uses an approximation of the slope which is based on Simpson's integration rule (see explanation in class or 2.6). It is sometimes called RK4.

```
> x0:=0; xn:=1.; # first and last points in the interval
  y0:=1; # initial condition
  n:=5; # number of steps
  h:=(xn-x0)/n; # step size
```

$$x0 := 0$$
$$xn := 1.$$
$$y0 := 1$$
$$n := 5$$
$$h := 0.2000000000 \quad (17)$$

```
> f:=(x,y)->y; # slope function (rhs in DE dy/dx = f(x,y), here f
  (x,y)=y)
```

$$f := (x, y) \rightarrow y \quad (18)$$

```
> x:=x0; y:=y0;
```

$$x := 0$$
$$y := 1 \quad (19)$$

```
> for i from 1 to n do
    k1:= f(x,y):            # left hand slope
    k2:= f(x+h/2,y+h*k1/2): # midpoint slope: first approx.
    k3:= f(x+h/2,y+h*k2/2): # midpoint slope: second approx.
    k4:= f(x+h,y+h*k3):     # right hand slope approx.
    k:=(k1+2*k2+2*k3+k4)/6:       # Simpson's integration rule
    y:= y + h*k:        # RK4 update
    x:= x+h:            # increase x
    print(x,y,exp(x));  # display current values and compare to
  true sol.
  od: # end for i loop
```

```
0.2000000000, 1.221400000, 1.221402758
0.4000000000, 1.491817960, 1.491824698
0.6000000000, 1.822106456, 1.822118800
0.8000000000, 2.225520825, 2.225540928
 1.000000000, 2.718251136, 2.718281828          (20)
```

```
>
>
```

We get 5 digits with 5 points, not bad. The DE we have used to sample these numerical methods is particularly benign. There are cases when even RK4 can give a numerical approximation that is way off, especially in cases where the solution blows up (see project).