These examples will be solving the differential equation $y' = y + x$, with initial value $y(0)=0$. Our goal is to find the value of $y(1)$.

Executable commands are placed after Maple prompts (">") and can be executed by pressing "Enter" after each prompt. Note that many commands will rely on information from previous commands, order of execution is important!
Hint: use Shift+Enter to enter a new line without creating a new prompt. Useful for making code more readable in loops, or when defining several variables at once.

First, we will use *restart* to clear all variables. *with()* is used to load additional Maple packages, we want to use commands in the *DETools* package.

```
> restart;
> with(DETools) :
```

Use Maple to find the exact solution to the differential equation.

Now, let's define some variables to use in solving the equation. y0 will represent y, y1 will represent y'. We need to define y0 as a function of x, and y1 as the derivative of y with respect to x.

```
> y0 := y(x);
  y1 := diff(y(x), x);
```

$$y0 := y(x)$$

$$y1 := \frac{d}{dx} y(x) \qquad \textbf{(1.1)}$$

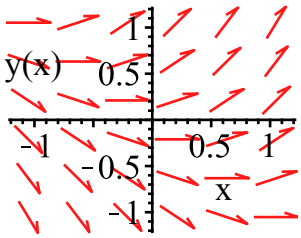
We can now store our differential equation (in terms of variables y0 and y1) to a new variable, call this ODE.

```
> ODE := y1 = x + y0;
```

$$ODE := \frac{d}{dx} y(x) = x + y(x) \qquad \textbf{(1.2)}$$

To plot a slopefield, use the *dfieldplot* command.

```
> dfieldplot(ODE, y0, x = -4..4, y = -4..4);
```



dsolve is used to find an exact solution for y(0)

```
> dsolve({ODE, y(0) = 0}, y0);
```

$$y(x) = -1 - x + e^{x} \qquad \textbf{(1.3)}$$

The *subs* command will input a value for x into the equation we just found. *evalf* will find a numerical solution using floating point arithmetic.
Hint: "%" is used to represent the last output generated, in our subs, it will use the results of our dsolve above. In evalf, this is using the results of our subs.

```
> subs(x = 1, %);
  evalf(%);
```

$$y(1) = -2 + e$$

$$y(1) = 0.718281828 \qquad \textbf{(1.4)}$$

Now we will solve the differential equation using numeric methods.

We will be using Euler's method and fourth-order Runge Kutta methods to estimate the value of *f(1)*

Define variables to use in calculating our estimates. In this case, we will a and fa to represent our inital value, $y(x_0)=y_0$
b represents our final *x* value, N represents our number of iterations.

Next, x[] and y[] will be used to store our values for x_n and y_n, h is our step size calculated using N.

```
> a := 0.0; fa := 0; b := 1.0; N := 1000;
```

$$a := 0.$$
$$fa := 0$$
$$b := 1.0$$
$$N := 1000 \quad \textbf{(2.1)}$$

```
> x[0] := a; y[0] := fa; h := (b - a)/N;
```

$$x_0 := 0.$$
$$y_0 := 0$$
$$h := 0.001000000000 \quad \textbf{(2.2)}$$

Recall: Euler's formula tells us that for a function where *y'(x) = f(x,y(x))*, given an initial value $y(x_0)=y_0$ we can find an approximate value of the function using the formula: $y_{n+1} = y_n + h*f(x_n,y_n)$

The following loop will perform the iterations of Euler's formula: add *h* to *x[n]* for each step, add *h*f(x[n],y[n])* to *y[n]* for each step
Then, use *print* to display our final output.

```
> for n from 0 to N - 1 do
  x[n + 1] := x[n] + h :
  y[n + 1] := y[n] + h·(x[n] + y[n]) :
  end do:
> print(N, x[N], y[N]);
```

1000, 1.000000000, 0.7169239329 **(2.1.1)**

In the previous example, we used (x[n]+y[n]) in our loop to represent f(x[n],y[n])=x[n]+y[n]. In the case of complex functions or repeating functions, it would be easier if we could use f (a,b) to calculate our outputs. There are a couple of ways to do this in Maple.

We can define a Maple procedure using *proc* if we wish to use *f(x,y)* more than once. *proc* will specify which inputs are being used, and allow us to create an algorithm for the output. In this case, we want to use a numeric input for *x* and *y*, and generate a numeric output (*x*+*y* from our differential equation.)

Then we will test the procedure that we just created with numeric values for *x* and *y*.

```
> f := proc(x :: numeric, y :: numeric) x + y end:
> f(2, 3);
```

5 **(2.2.1)**

```
> f(1, 1);
```

2 **(2.2.2)**

We can also use a functional operator to define a special sort of procedure. These are written using arrow notation ("->") First, specify a series of variables,

```
> g := (x, y) -> x + y :
> g(2, 3);
```

5 **(2.2.3)**

```
> g(1, 1);
```

2 **(2.2.4)**

Now we can rewrite our Euler's method loop using one of the procedures we just defined. Note that we will get the same values that we in 2.1.1.

```
> for n from 0 to N - 1 do
    x[n + 1] := x[n] + h :
    y[n + 1] := y[n] + h·f(x[n], y[n]) :
  end do:
> print(N, x[N], y[N]);
```

1000, 1.000000000, 0.7169239329 **(2.2.5)**

```
> h/6
```

0.0001666666667 **(2.2.6)**

Recall: Fourth-Order Runge Kutta tells us that for a function where $y'(x) = f(x,y(x))$, given an initial value $y(x_0)=y_0$ we can find an approximate value of the function using the formulae: $y_{n+1}=y_n+(h/6)(k_1+2k_2+2k_3+k_4)$ and $x_{n+1}=x_n+h$

k_i are given by:

$k_1=f(x_n, y_n)$

$k_2=f(x_n+h/2, y_n+hk_1/2)$

$k_3=f(x_n+h/2, y_n+hk_2/2)$

$k_4=f(x_n+h, y_n+hk_3)$

We will be using the procedure $f(x,y)$ defined above for Euler's formula, along with the same initial values and step size from 2.1 and 2.2.

The following loop will perform the iterations of fourth-order Runge Kutta, then we will output the values.

```
> for n from 0 to N - 1 do
  x[n + 1] := x[n] + h;
  K[1] := f(x[n], y[n]) :
  K[2] := f(x[n] + h/2, y[n] + h·K[1]/2) :
  K[3] := f(x[n] + h/2, y[n] + h·K[2]/2) :
  K[4] := f(x[n] + h, y[n] + h·K[3]) :
  y[n + 1] := y[n] + (h/6)·(K[1] + 2 K[2] + 2 K[3] + K[4]) :
  end do:
> print(N, x[N], y[N]);
```

1000, 1.000000000, 0.7182818303 **(2.3.1)**