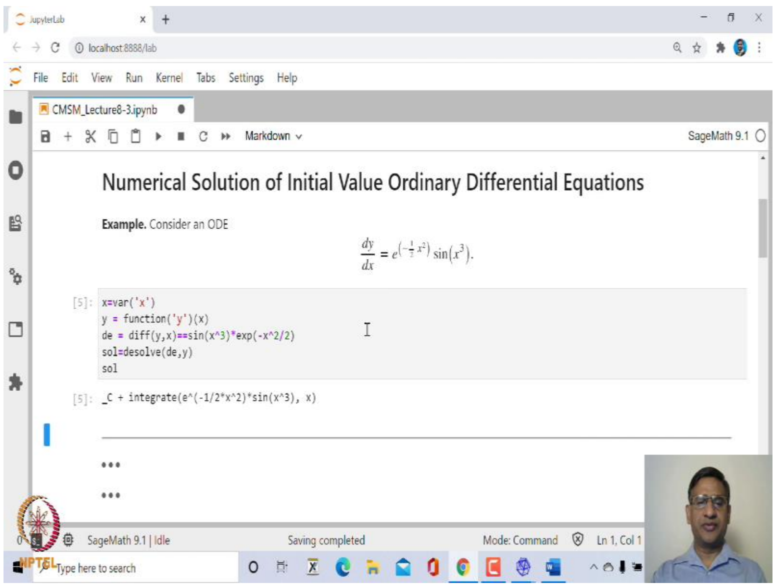
Computational Mathematics with SageMath

Prof. Ajit Kumar
Department of Mathematics
Institute of Chemical Technology, Mumbai

Lecture - 51
Euler's Method to solve 1st order ODE with SageMath

Welcome to the 51st lecture on Computational Mathematics with SageMath. In this lecture we are going to explore some methods of solving ordinary differential equations numerically or numerical solutions of ordinary differential equations. Not every differential equation can be solved analytically. So, in that case you need to appeal to numerical solutions or find some approximate solutions. Similar to what we saw in case of finding roots of an equation of the form f(x)= 0. In case f(x)=0, we are unable to solve explicitly, then we try to approximate a solution by means of numerical techniques.

(Refer Slide Time: 01:09)



Numerical Solution of Initial Value Ordinary Differential Equations

Example. Consider an ODE

$$\frac{dy}{dx} = e^{\left(-\frac{1}{2}x^2\right)}\sin\left(x^3\right).$$

```
[5]: x=var('x')
     y = function('y')(x)
     de = diff(y,x)==sin(x^3)*exp(-x^2/2)
     sol=desolve(de,y)
     sol

[5]: _C + integrate(e^(-1/2*x^2)*sin(x^3), x)
```

So, for example, let us start with a differential equation

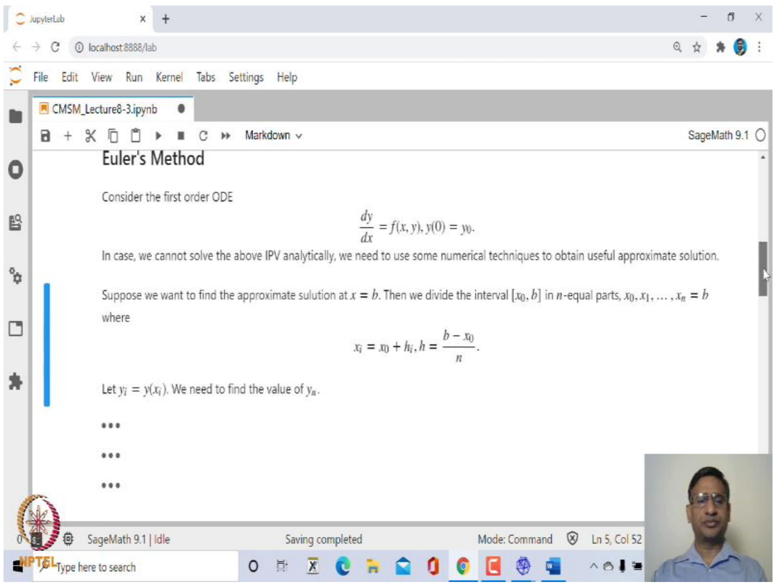
dy dx is equal to e to the power minus x square into sin x cube.

Now, if you try to solve this differential equation, in Sage using desolve, then you will not get explicit solution. It simply returns the constant and integral of this right hand side, which is not what we want.

This is one such example, there are several examples, where you cannot solve differential equation explicitly.

Fine! Solving differential equation of this form amounts to finding integral of a function and we have also seen that you cannot find integral of every function, even though the function is integrable. So, in that case we find numerical integrals. So, this is an example where Sage is unable to find a solution explicitly.

(Refer Slide Time: 02:23)



Euler's Method

Consider the first order ODE

$$\frac{dy}{dx} = f(x, y), y(0) = y_0.$$

In case, we cannot solve the above IPV analytically, we need to use some numerical techniques to obtain useful approximate solution.

Suppose we want to find the approximate sulution at $x = b$. Then we divide the interval $[x_0, b]$ in n-equal parts, $x_0, x_1, \dots, x_n = b$ where

$$x_i = x_0 + h_i, h = \frac{b - x_0}{n}.$$

Let $y_i = y(x_i)$. We need to find the value of y_n.

•••

•••

•••

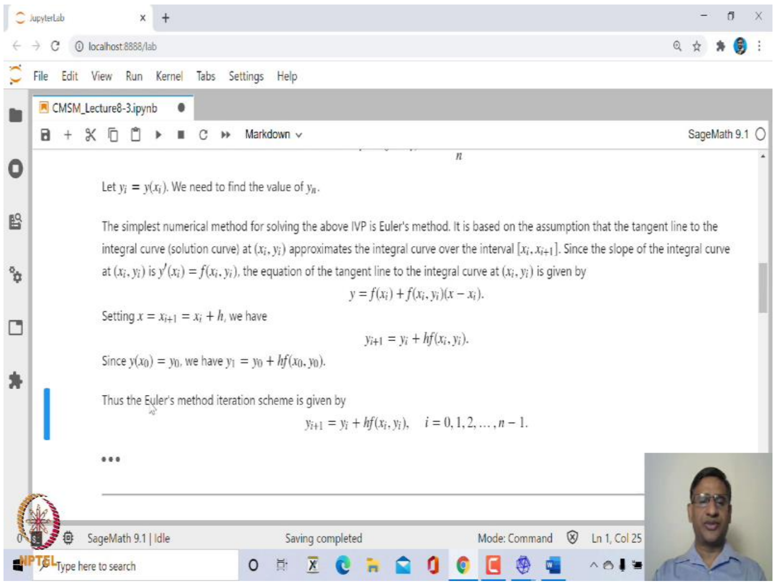
In that case we need to appeal to numerical methods. The first and the simplest numerical methods, we will look at is known as Euler's method. What is it? Suppose, you want to solve this differential equation

dy dx is equal to f(x, y) and with initial condition y(0) is equal to let us say, some number y0.

The idea is to obtained or approximate value of y, which is solution of this differential equation at some point starting with x=0.

How does it work? In order to find the approximate solution, an approximate solution of y at x equal to, let us say b. What we do? We divide this interval [x0,b], x0 is the starting point at which the value of the solution is given, in n equal parts, let us say x0, x1,…, xn. The length of each of this sub-interval will be nothing but, b minus x0 by n. Let us call that as h, which we call as a step size. Now, let yi be y at xi. This y at xi accept i equal to 0 is what we need to find out and in the process, we will be able to find y at b.

(Refer Slide Time: 04:16)



Let $y_i = y(x_i)$. We need to find the value of y_n.

The simplest numerical method for solving the above IVP is Euler's method. It is based on the assumption that the tangent line to the integral curve (solution curve) at (x_i, y_i) approximates the integral curve over the interval $[x_i, x_{i+1}]$. Since the slope of the integral curve at (x_i, y_i) is $y'(x_i) = f(x_i, y_i)$, the equation of the tangent line to the integral curve at (x_i, y_i) is given by

$$y = f(x_i) + f(x_i, y_i)(x - x_i).$$

Setting $x = x_{i+1} = x_i + h$, we have

$$y_{i+1} = y_i + hf(x_i, y_i).$$

Since $y(x_0) = y_0$, we have $y_1 = y_0 + hf(x_0, y_0)$.

Thus the Euler's method iteration scheme is given by

$$y_{i+1} = y_i + hf(x_i, y_i), \quad i = 0, 1, 2, \ldots, n-1.$$

So, let us see how we will do this? The basic idea behind this is to approximate the function that is y, by a linear approximation or by the tangent line. So, it is based on the assumption that the tangent line to the integral curve, which is the solution curve at any point (xi, yi) approximates the integral curve over the interval xi, x i+1.

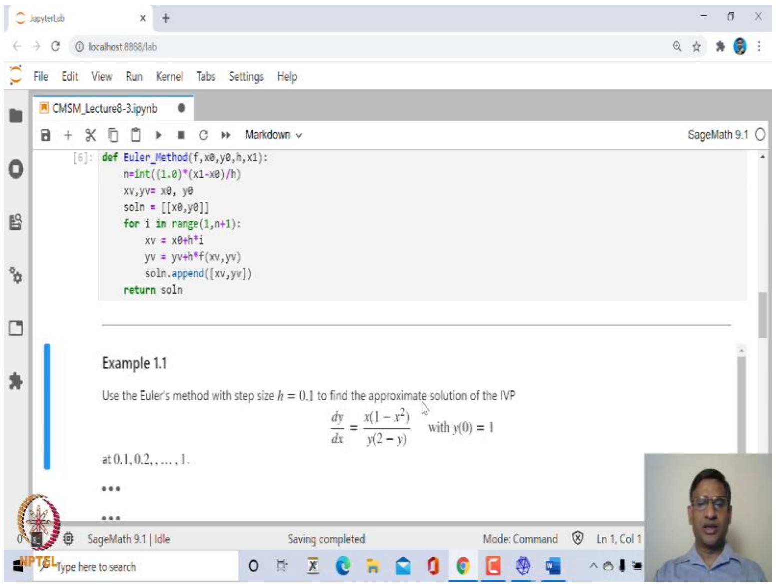
In case x i+1 is very close to xi. This approximation, which is linear approximation of the function y, will be reasonably good approximation. Now, since the slope of the integral curve at any point (xi, yi) is already given to us, that is y dash at xi, which is f(xi,yi), the equation of the tangent line in this case will be given by y is equal to f of xi plus f(xi, yi)(x-xi).

Now, suppose we want to evaluate at x equal to xi plus 1. So, if you substitute x equal to x i plus 1, that is, x i plus h. Then y i plus 1 will be y i plus h times f (xi,yi). So, what is this is? So, the first iterate namely y1 will be what? y1 is equal to y0 plus h times f(x0,y0).

So, once we know y1, then we can find y2, and then we can find y3 and so on. In general, we can find yn right. Therefore, what happens to the iteration scheme of this Euler's method, this is what is called Euler's method, and is given by y i plus 1, that is y at x i plus 1 is equal to y i, that is, approximate value of the solution at the previous point plus h times f (x i, yi).

So, that is is how this iteration schemes is defined, and i can take values 0, 1, 2, n-1.

(Refer Slide Time: 06:45)



```
[6]: def Euler_Method(f,x0,y0,h,x1):
         n=int((1.0)*(x1-x0)/h)
         xv,yv= x0, y0
         soln = [[x0,y0]]
         for i in range(1,n+1):
             xv = x0+h*i
             yv = yv+h*f(xv,yv)
             soln.append([xv,yv])
         return soln
```

Example 1.1

Use the Euler's method with step size $h = 0.1$ to find the approximate solution of the IVP

$$\frac{dy}{dx} = \frac{x(1-x^2)}{y(2-y)} \quad \text{with } y(0) = 1$$

at $0.1, 0.2, , \ldots, 1$.

Now let us write a Python program, user defined function for Euler's method. This is fairly simple. We want to input the value of the function f, the initial point at which the initial value is given, x0 and the value of y at x0, the step size h, and the point at which you want to evaluate, that in in this case, we call that as b. So, x1 is b here.

Then define what is n? n will be x1 minus x0 by h. Since, h is given. If you are dividing in n equal parts, h is given by b minus x0 upon n. Therefore, n will be given by b minus x0 by h. Here instead of b we have taken x1. So, this is what we have done.

Now, let us define xv and yv value of xv; xv will take value of x0, x1, x2 up to xn. And xi is nothing but x0 plus h times i; that is the nodes, we call them as node points or nodes. So, xv and yv initialize to x0 and y0. What we are going to do? We will find the solution at all these node points, that is, at every xi, we will find yi. We will put these values in a list namely xi, yi, we will create a list of xi's, yi's. Now i is going in the range of 1 to n +1, because, you have to go up to n plus 1. Because, xn is your b or x1 and these are the initial for x, for i equal to 0, it is already given.

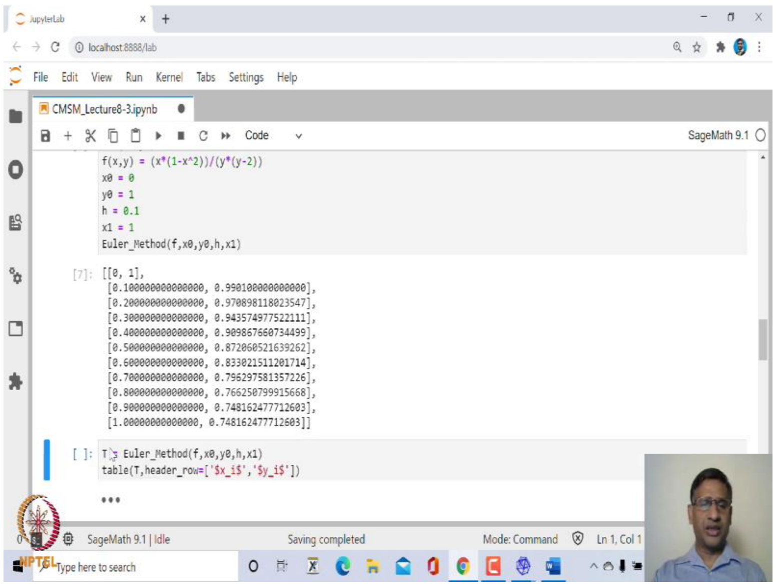
Therefore, we have to start from i=1 and then what is xv? xv is x0 plus h times i, that is, i-th node point. And the value of yv is, yv, that is the previous value, in this case for example, we it will take y0 and it will find y0 plus h times f(0 y0), and you append this xv, yv to this solution. At the end return this solution.

This program is fairly simple, very simple in fact. Now, let us call this method, the Euler's user defined function. Let us solve an ordinary differential equation of first order, namely

dy by dx is equal to x into 1 minus x square upon y into 2 minus y with initial condition y0 is equal to 0

let us find out value of y at node points, 0.1, 0.2 up to 1. So, here step size is 0.1.

(Refer Slide Time: 09:50)



```
f(x,y) = (x*(1-x^2))/(y*(y-2))
x0 = 0
y0 = 1
h = 0.1
x1 = 1
Euler_Method(f,x0,y0,h,x1)
```

```
[7]: [[0, 1],
      [0.100000000000000, 0.990100000000000],
      [0.200000000000000, 0.970898118023547],
      [0.300000000000000, 0.943574977522111],
      [0.400000000000000, 0.909867660734499],
      [0.500000000000000, 0.872060521639262],
      [0.600000000000000, 0.833021511201714],
      [0.700000000000000, 0.796297581357226],
      [0.800000000000000, 0.766250799915668],
      [0.900000000000000, 0.748162477712603],
      [1.00000000000000, 0.748162477712603]]
```

```
[ ]: T = Euler_Method(f,x0,y0,h,x1)
     table(T,header_row=['$x_i$','$y_i$'])
```

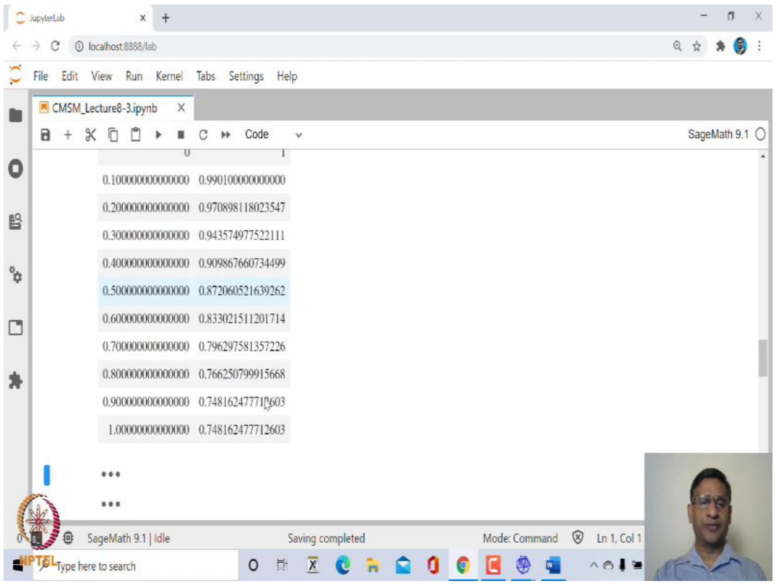
So, let us define x and y as variables, dy dx is equal to f(x, y). This is the right hand side, we will define as f(x, y) and x 0, y0 is 1, h which is step size is 0.1 and the x1 the end point at which we want to find the approximate the value of y is 1.

So, now let us call Eule_method(f, x0, y0, h,x1). This gives you a list of xi's, yi's.

So, if 0 at 0 it was 1, at 0.1 it is 0.9901 and so on. At 1 this is 0.748162 and so on. Right!

You can tabulate these values, you can put it in a tabular form using a function called table and then you need to give this as list. And then if you want the heading, you give the heading. Here we are saying the header headings are xi, yi.

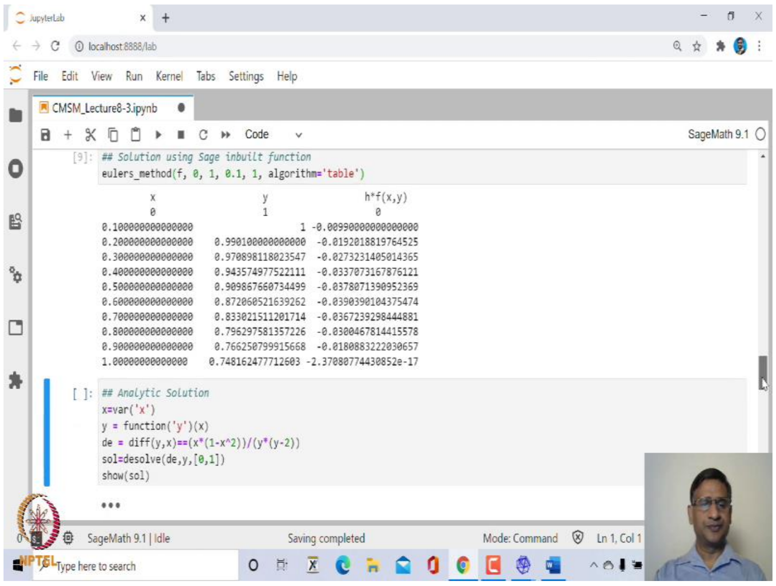
(Refer Slide Time: 10:57)



```
                0                  1
0.100000000000000  0.990100000000000
0.200000000000000  0.970898118023547
0.300000000000000  0.943574977522111
0.400000000000000  0.909867660734499
0.500000000000000  0.872060521639262
0.600000000000000  0.833021511201714
0.700000000000000  0.796297581357226
0.800000000000000  0.766250799915668
0.900000000000000  0.748162477712603
 1.00000000000000  0.748162477712603
```

Let us run this, this is what you get xi. So, i = 0, if you want you could have also printed the iteration number as 1 first column. So, this is at 0 it is 1, at 0.1 it is 0.999 and so on. So, that is how you can find the solution.

(Refer Slide Time: 11:20)



```
[9]: ## Solution using Sage inbuilt function
     eulers_method(f, 0, 1, 0.1, 1, algorithm='table')
```

```
               x                    y                 h*f(x,y)
               0                    1                        0
0.100000000000000                   1 -0.00990000000000000
0.200000000000000   0.990100000000000  -0.0192018819764525
0.300000000000000   0.970898118023547  -0.0273231405014365
0.400000000000000   0.943574977522111  -0.0337073167876121
0.500000000000000   0.909867660734499  -0.0378071390952369
0.600000000000000   0.872060521639262  -0.0390390104375474
0.700000000000000   0.833021511201714  -0.0367239298444881
0.800000000000000   0.796297581357226  -0.0300467814415578
0.900000000000000   0.766250799915668  -0.0180883222030657
1.00000000000000   0.748162477712603 -2.37080774430852e-17
```

```
[ ]: ## Analytic Solution
     x=var('x')
     y = function('y')(x)
     de = diff(y,x)==(x*(1-x^2))/(y*(y-2))
     sol=desolve(de,y,[0,1])
     show(sol)
```

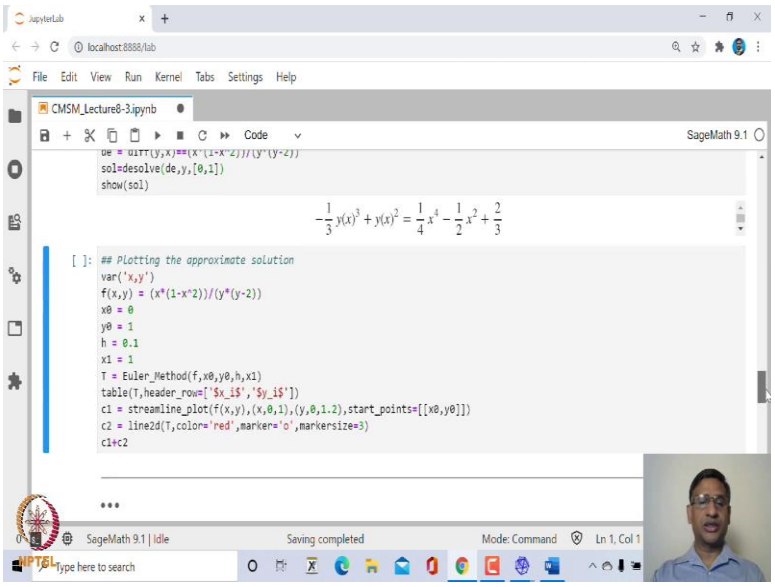
You can also use inbuilt function. In the last lecture, I told you Sage already has several inbuilt functions to find numerical solution of an ordinary differential equation. Euler_method is one such. So, let us call that Euler's_methods, that is an inbuilt function.

So, you have to give input f and then then the initial point x0, the value of f, y at x0, the step length and this is the end point. And it says that, it uses an option called table. So, it will tabulate the value exactly the same way as we have done in this case right. So, let us run this and see what we get.

Here by default this reports x value y value and h times f(x,y) value. This is just to see that, y plus h times f(x, y) will be the next iterate. That is what you can see here, this was 1 and this is 0.0099 and the next value comes as 0.9901 and so on. Right!

This is exactly very similar to what we have done, except that it is also reporting h times f(x,y) at every iterate. We have already seen how to solve this kind of differential equations analytically. So, if you try to solve this using desolve, then let us see what the solution we get?

(Refer Slide Time: 12:58)



```
de = diff(y,x)==(x*(1-x^2))/(y*(y-2))
sol=desolve(de,y,[0,1])
show(sol)
```

$$-\frac{1}{3}y(x)^3 + y(x)^2 = \frac{1}{4}x^4 - \frac{1}{2}x^2 + \frac{2}{3}$$

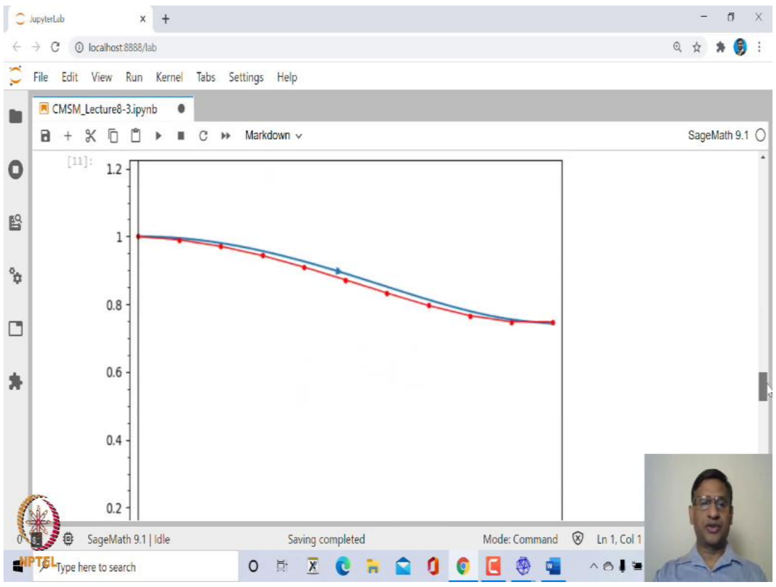
```
[ ]: ## Plotting the approximate solution
     var('x,y')
     f(x,y) = (x*(1-x^2))/(y*(y-2))
     x0 = 0
     y0 = 1
     h = 0.1
     x1 = 1
     T = Euler_Method(f,x0,y0,h,x1)
     table(T,header_row=['$x_i$','$y_i$'])
     c1 = streamline_plot(f(x,y),(x,0,1),(y,0,1.2),start_points=[[x0,y0]])
     c2 = line2d(T,color='red',marker='o',markersize=3)
     c1+c2
```

So, this is the solution, and is given in terms of y and x and this is implicitly defined. If you want to solve this explicitly, this will require again some another numerical technique, because in some cases you may be able to find y(x) explicitly. But in most of the cases it may not be possible.

Let us look at geometrically what it means? First let us plot the graph of the solution, and that we can do it using streamline_plot. We have seen this in the last class. We want to plot the integral curve passing through (x0, y0), that is the starting point. Then we also want to plot the set of points (xi, yi), obtained using Euler's Euler's method.

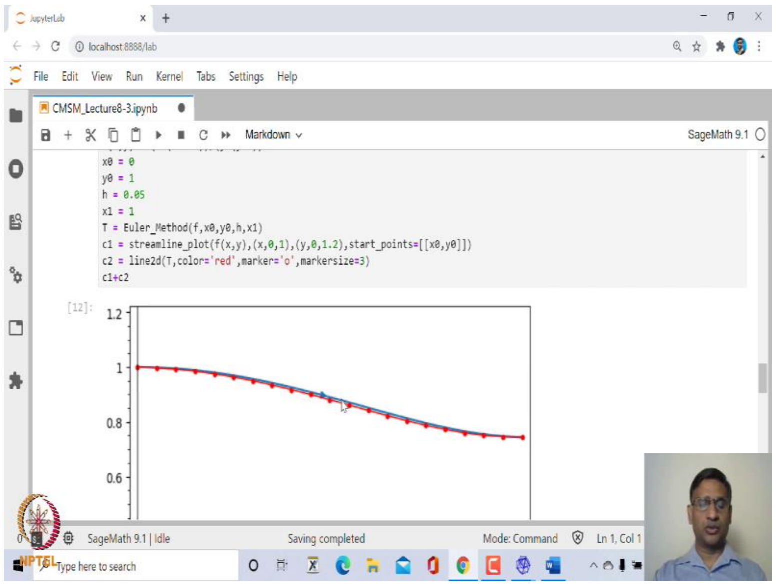
So, we are calling T to be the eulers_method and then create a table of these values. So, this actually, at present is not required. Let me get rid of this and then c1 is streamline plot, that is the solution curve or integral curve. You can think of this as explicit the curve or I mean solution obtained explicitly. And c2 to be the plot of the set of points in the table T, and put a marker as a dot and marker size is 3.

(Refer Slide Time: 14:47)



So, let us run this. When you run this, this is what you see. So, you can see here this blue one is the actual solution, blue one is the actual solution and the red one is the approximate solution.

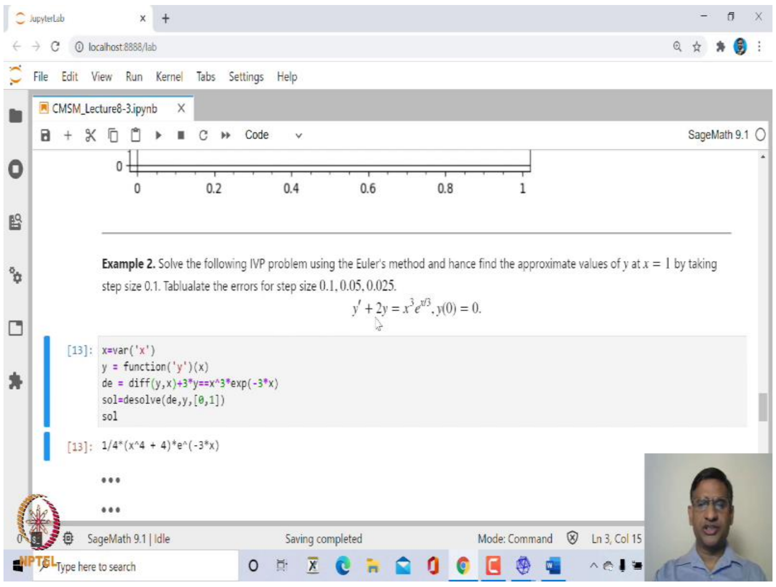
(Refer Slide Time: 15:07)



```
x0 = 0
y0 = 1
h = 0.05
x1 = 1
T = Euler_Method(f,x0,y0,h,x1)
c1 = streamline_plot(f(x,y),(x,0,1),(y,0,1.2),start_points=[[x0,y0]])
c2 = line2d(T,color='red',marker='o',markersize=3)
c1+c2
```

Now, in case you are going to decrease the step length, suppose I, decrease the step length and I make it 0.05 and then you will see that the solution curve, the approximate solution curve will be much closer to the actual solution. You can you can try with even smaller step size and then see the approximate solution.

(Refer Slide Time: 15:38)



Example 2. Solve the following IVP problem using the Euler's method and hance find the approximate values of y at $x = 1$ by taking step size 0.1. Tablualate the errors for step size $0.1, 0.05, 0.025$.

$$y' + 2y = x^3 e^{x/3}, y(0) = 0.$$

```
[13]: x=var('x')
      y = function('y')(x)
      de = diff(y,x)+3*y==x^3*exp(-3*x)
      sol=desolve(de,y,[0,1])
      sol

[13]: 1/4*(x^4 + 4)*e^(-3*x)
```

Now, you can also try to kind of tabulate the errors. At every each step size let us say 0.1, 0.05, 0.025, you are approximating the value of y. Right! You are approximating y, with these given step size.

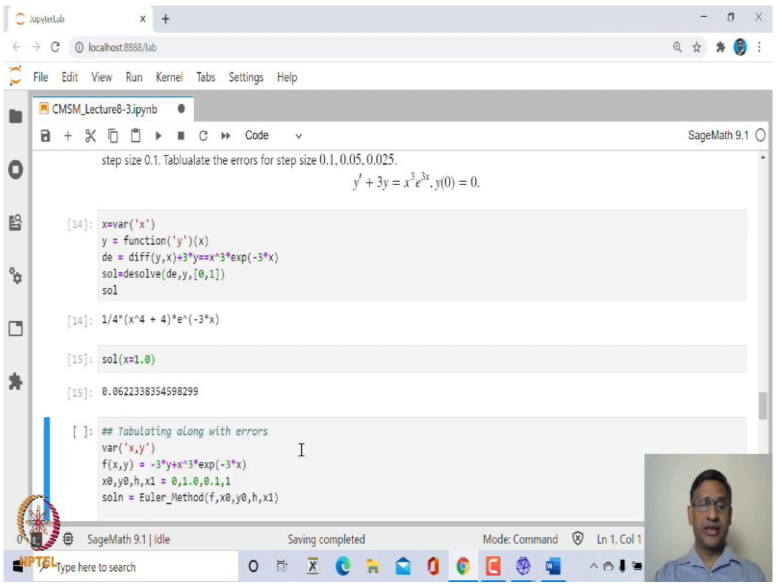
Now, what we would like to do is, let us also tabulate the error at each stage. Let us see, in the previous example, we did not have the solution in explicit form. So, evaluating explicit solution or exact solution will not be straight forward, still one can do numerically, you can approximate.

We will look at an example, where we have the explicit exact solution. In this case we are looking at

y dash plus 2y is equal to x cube into e to the power x by 3 and y at 0 is equal to 0.

So, let us let us first solve this analytically. In this case, the solution is of this initial value problem is 1 by 4 into x to the power 4 plus 4 into e to the power minus 3 x. I think this there is a small difference between this and this.

(Refer Slide Time: 17:12)



step size 0.1. Tablualate the errors for step size 0.1, 0.05, 0.025.

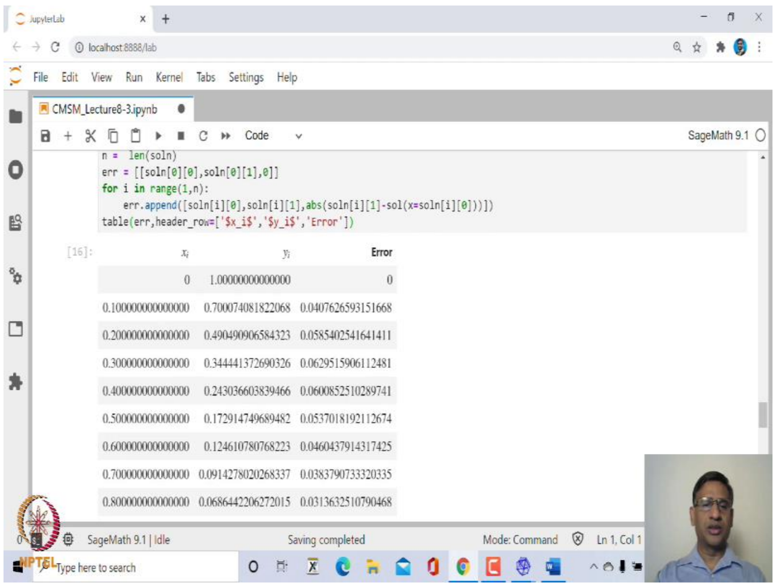
$$y' + 3y = x^3 e^{3x}, y(0) = 0.$$

```
[14]: x=var('x')
      y = function('y')(x)
      de = diff(y,x)+3*y==x^3*exp(-3*x)
      sol=desolve(de,y,[0,1])
      sol

[14]: 1/4*(x^4 + 4)*e^(-3*x)

[15]: sol(x=1.0)

[15]: 0.0622338354598299

[ ]: ## Tabulating along with errors
     var('x,y')
     f(x,y) = -3*y+x^3*exp(-3*x)
     x0,y0,h,x1 = 0,1.0,0.1,1
     soln = Euler_Method(f,x0,y0,h,x1)
```

Let me change this differential equation, let us change this to 3 y and this is 3 x. I would have first taken that differential equation, while solving I would have changed while experimenting.

This is the differential equation and its solution is given by 1 by 4 into x to the power 4 plus 4 and the whole thing multiplied by e to the power minus 3 x. That is the explicit solution. You can evaluate this solution at 1.0. This is what you get, this is the explicit solution. Of course, this is the approximation of this, this will also be an approximate value. But in any case this will also involve some error.

(Refer Slide Time: 18:02)



```
n =  len(soln)
err = [[soln[0][0],soln[0][1],0]]
for i in range(1,n):
    err.append([soln[i][0],soln[i][1],abs(soln[i][1]-sol(x=soln[i][0]))])
table(err,header_row=['$x_i$','$y_i$','Error'])
```

[16]:

x_i	y_i	Error
0	1.00000000000000	0
0.100000000000000	0.700074081822068	0.0407626593151668
0.200000000000000	0.490490906584323	0.0585402541641411
0.300000000000000	0.344441372690326	0.0629515906112481
0.400000000000000	0.243036603839466	0.0600852510289741
0.500000000000000	0.172914749689482	0.0537018192112674
0.600000000000000	0.124610780768223	0.0460437914317425
0.700000000000000	0.0914278020268337	0.0383790733320335
0.800000000000000	0.0686442206272015	0.0313632510790468

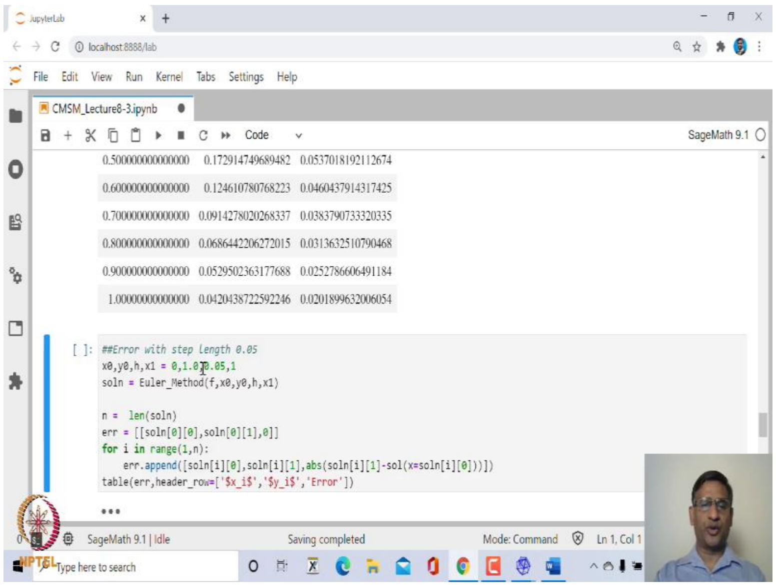
Now, let us tabulate the value of xi, yi, at each of these node points, for step length 0.1 to begin with. So here we have to write this differential equation as y dash is equal to f x, y. So, we need to push this, the left hand side 3 y to the right hand side.

So, that is why f(x,y) is equal to minus 3 y plus x cube into e to the power minus 3 x. Let us store this solution using Euler's method in soln, and then let us find the length of this. Then let us also define the error term. Error term is going to be to be the value of yi minus the explicit value or exact value of y at xi obtained, that is stored in soln. so soln at xi, that is what is done here, the initial error is 0.

So, we have xi, yi. So, this is x0, y0 and the initial error is 0. And in that, next time we will we will append a solution, first coordinate of the solution xi, and then second coordinate, which is yi and then the absolute value of the difference of yi with the exact solution. After that you print this using a table.

Here we have xi, yi and these are the error. So, you can see here, in this case, there is quite a bit of error. In this case almost 2 percent error here.

(Refer Slide Time: 19:59)

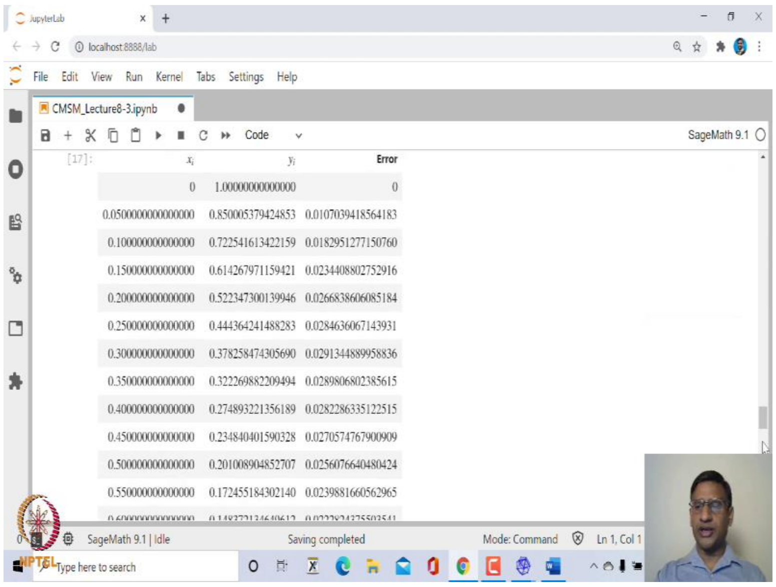


```
0.500000000000000  0.172914749689482 0.0537018192112674
0.600000000000000  0.124610780768223 0.0460437914317425
0.700000000000000 0.0914278020268337 0.0383790733320335
0.800000000000000 0.0686442206272015 0.0313632510790468
0.900000000000000 0.0529502363177688 0.0252786606491184
 1.00000000000000 0.0420438722592246 0.0201899632006054
```

```
[ ]: ##Error with step length 0.05
     x0,y0,h,x1 = 0,1.0,0.05,1
     soln = Euler_Method(f,x0,y0,h,x1)

     n =  len(soln)
     err = [[soln[0][0],soln[0][1],0]]
     for i in range(1,n):
         err.append([soln[i][0],soln[i][1],abs(soln[i][1]-sol(x=soln[i][0]))])
     table(err,header_row=['$x_i$','$y_i$','Error'])
```

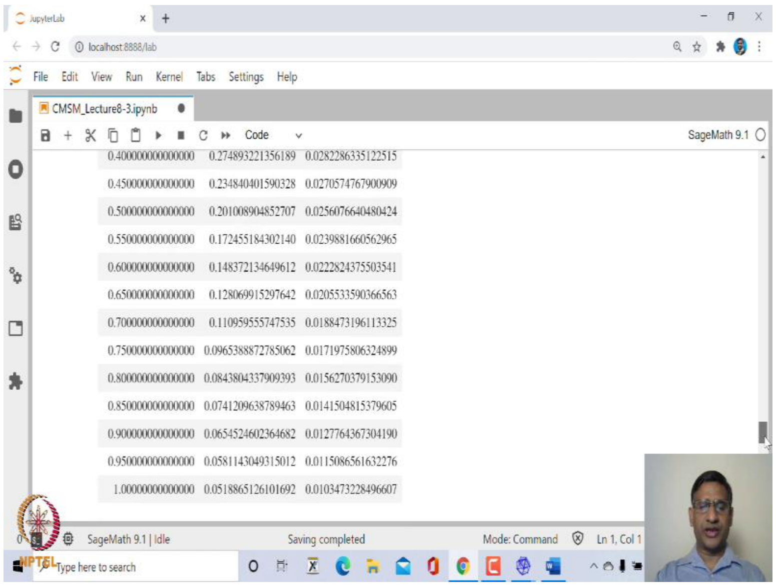
Now, suppose we decrease the step size, suppose we will take it 0.05 and again try to tabulate the error at each of this node point.

(Refer Slide Time: 20:11)



```
[17]:                 x_i                y_i               Error
                        0   1.00000000000000                   0
      0.0500000000000000  0.850005379424853 0.0107039418564183
       0.100000000000000  0.722541613422159 0.0182951277150760
       0.150000000000000  0.614267971159421 0.0234408802752916
       0.200000000000000  0.522347300139946 0.0266838606085184
       0.250000000000000  0.443364241488283 0.0284636067143931
       0.300000000000000  0.378258474305690 0.0291344889958836
       0.350000000000000  0.322269882209494 0.0289806802385615
       0.400000000000000  0.274893221356189 0.0282286335122515
       0.450000000000000  0.234840401590328 0.0270574767900909
       0.500000000000000  0.201008904852707 0.0256076640480424
       0.550000000000000  0.172455184302140 0.0239881660562965
```

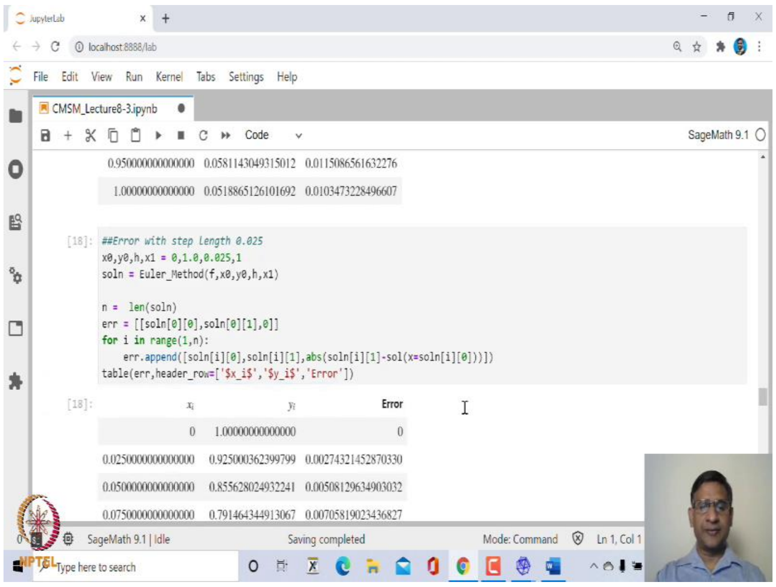
(Refer Slide Time: 20:16)



```
0.400000000000000  0.274893221356189  0.0282286335122515
0.450000000000000  0.234840401590328  0.0270574767900909
0.500000000000000  0.201008904852707  0.0256076640480424
0.550000000000000  0.172455184302140  0.0239881660562965
0.600000000000000  0.148372134649612  0.0222824375503541
0.650000000000000  0.128069915297642  0.0205533590366563
0.700000000000000  0.110959555747535  0.0188473196113325
0.750000000000000  0.0965388872785062 0.0171975806324899
0.800000000000000  0.0843804337909393 0.0156270379153090
0.850000000000000  0.0741209638789463 0.0141504815379605
0.900000000000000  0.0654524602364682 0.0127764367304190
0.950000000000000  0.0581143049315012 0.0115086561632276
1.00000000000000   0.0518865126101692 0.0103473228496607
```

You will be able to see that this error is much smaller than error using 0.5. You can see here in this case error is 0.01103, and in the previous case at 1 the error was 0.02. So, almost it has become half of the previous error.

(Refer Slide Time: 20:33)

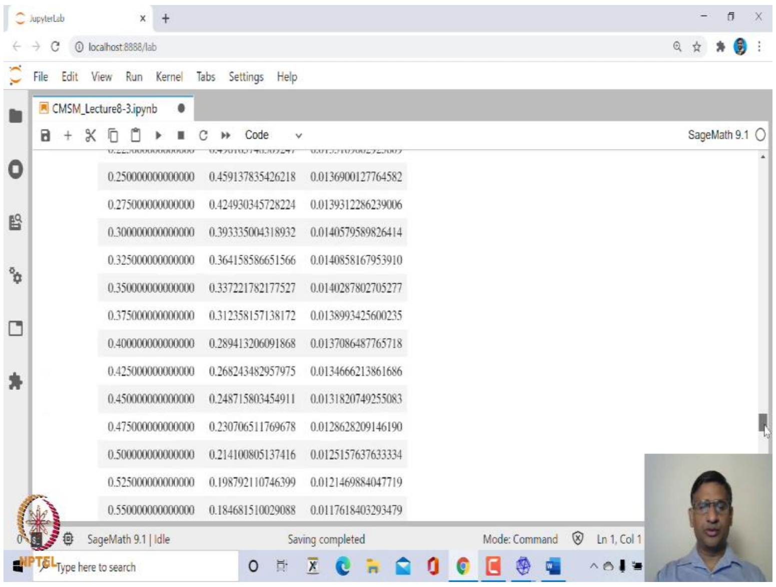


```
0.950000000000000  0.0581143049315012 0.0115086561632276
1.00000000000000   0.0518865126101692 0.0103473228496607
```

```
[18]: ##Error with step length 0.025
      x0,y0,h,x1 = 0,1.0,0.025,1
      soln = Euler_Method(f,x0,y0,h,x1)

      n =  len(soln)
      err = [[soln[0][0],soln[0][1],0]]
      for i in range(1,n):
         err.append([soln[i][0],soln[i][1],abs(soln[i][1]-sol(x=soln[i][0]))])
      table(err,header_row=['$x_i$','$y_i$','Error'])
```

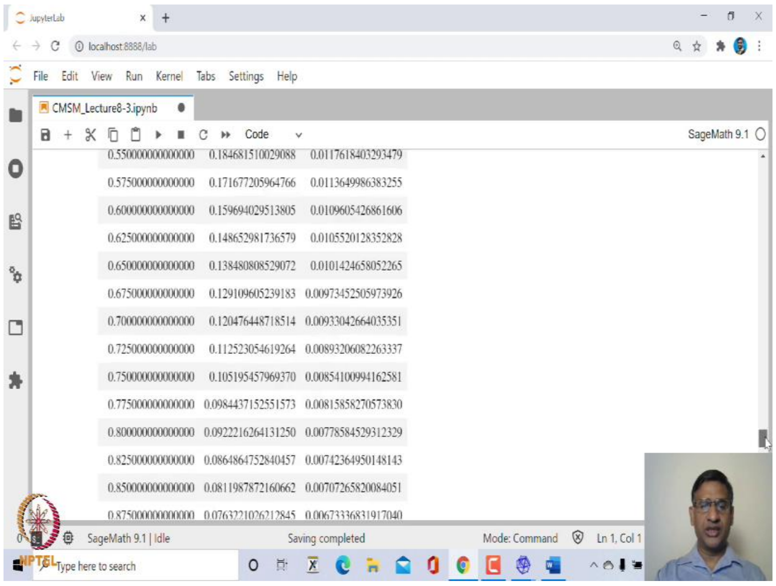
```
[18]:               x_i                y_i               Error
                      0   1.00000000000000                   0
     0.0250000000000000  0.925000362399799  0.00274321452870330
     0.0500000000000000  0.855628024932241  0.00508129634903032
     0.0750000000000000  0.791464344913067  0.00705819023436827
```

(Refer Slide Time: 20:40)



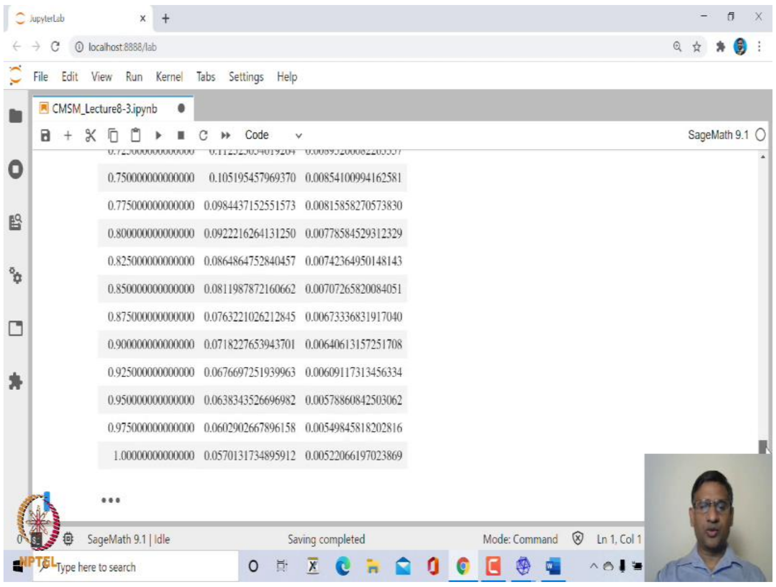
```
0.250000000000000  0.459137835426218  0.0136900127764582
0.275000000000000  0.424930345728224  0.0139312286239006
0.300000000000000  0.393335004318932  0.0140579589826414
0.325000000000000  0.364158586651566  0.0140858167953910
0.350000000000000  0.337221782177527  0.0140287802705277
0.375000000000000  0.312358157138172  0.0138993425600235
0.400000000000000  0.289413206091868  0.0137086487765718
0.425000000000000  0.268243482957975  0.0134666213861686
0.450000000000000  0.248715803454911  0.0131820749255083
0.475000000000000  0.230706511769678  0.0128628209146190
0.500000000000000  0.214100805137416  0.0125157637633334
0.525000000000000  0.198792110746399  0.0121469884047719
0.550000000000000  0.184681510029088  0.0117618403293479
```

(Refer Slide Time: 20:40)



```
0.550000000000000  0.184681510029088  0.0117618403293479
0.575000000000000  0.171677205964766  0.0113649986383255
0.600000000000000  0.159694029513805  0.0109605426861606
0.625000000000000  0.148652981736579  0.0105520128352828
0.650000000000000  0.138480808529072  0.0101424658052265
0.675000000000000  0.129109605239183  0.00973452505973926
0.700000000000000  0.120476448718514  0.00933042664035351
0.725000000000000  0.112523054619264  0.00893206082263337
0.750000000000000  0.105195457969370  0.00854100994162581
0.775000000000000  0.0984437152551573  0.00815858270573830
0.800000000000000  0.0922216264131250  0.00778584529312329
0.825000000000000  0.0864864752840457  0.00742364950148143
0.850000000000000  0.0811987872160662  0.00707265820084051
0.875000000000000  0.0763221026212845  0.00673336831917040
```

(Refer Slide Time: 20:41)

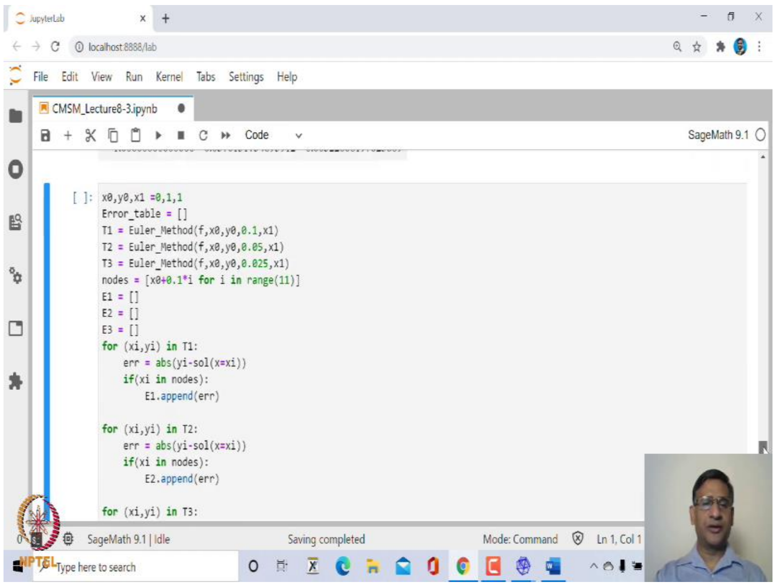


```
0.750000000000000  0.105195457969370  0.00854100994162581
0.775000000000000  0.0984437152551573 0.00815858270573830
0.800000000000000  0.0922216264131250 0.00778584529312329
0.825000000000000  0.0864864752840457 0.00742364950148143
0.850000000000000  0.0811987872160662 0.00707265820084051
0.875000000000000  0.0763221026212845 0.00673336831917040
0.900000000000000  0.0718227653943701 0.00640613157251708
0.925000000000000  0.0676697251939963 0.00609117313456334
0.950000000000000  0.0638343526696982 0.00578860842503062
0.975000000000000  0.0602902667896158 0.00549845818202816
1.00000000000000   0.0570131734895912 0.00522066197023869
```

Similarly, if I reduce this step size again by half and then try to tabulate the error, then you will be able to see that error would have decreased further. Again it is almost half of the previous error. Thus you can see here in this case the error seems to be reducing linearly. So, it is not very fast method in that sense.

Of course, you can look at the theory and see how this error can be obtained or error will be function of the step size h.

(Refer Slide Time: 21:14)

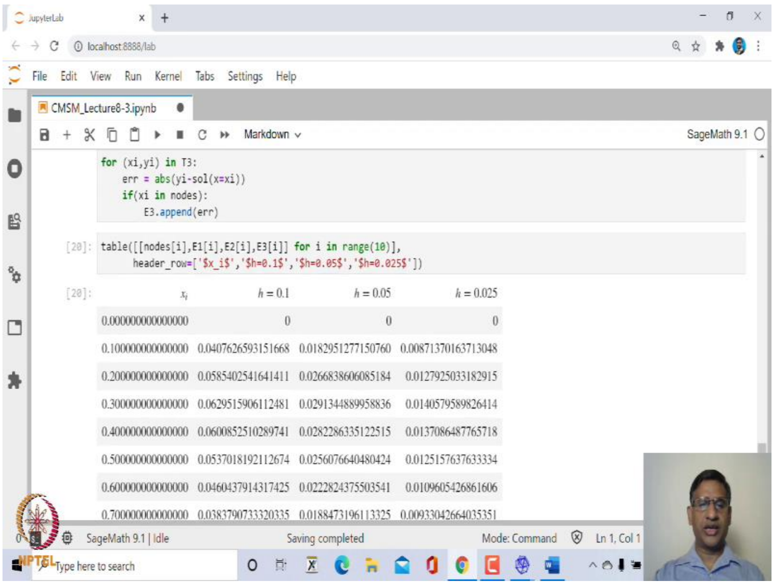


```
x0,y0,x1 =0,1,1
Error_table = []
T1 = Euler_Method(f,x0,y0,0.1,x1)
T2 = Euler_Method(f,x0,y0,0.05,x1)
T3 = Euler_Method(f,x0,y0,0.025,x1)
nodes = [x0+0.1*i for i in range(11)]
E1 = []
E2 = []
E3 = []
for (xi,yi) in T1:
    err = abs(yi-sol(x=xi))
    if(xi in nodes):
        E1.append(err)

for (xi,yi) in T2:
    err = abs(yi-sol(x=xi))
    if(xi in nodes):
        E2.append(err)

for (xi,yi) in T3:
```

You can even compare all the errors together, using very simple Sage code. So, we are defining x0, y0 and x1, the Error_Table, I am calling.

We will print value of xi and the error for step size 0.1, 0.05 and 0.025. These are the nodes. We are initializing these errors and then for first one let us append this error in E1 and then the second step size append this in error E2. And then similarly, do it with x3, that is, step size 0.025.

(Refer Slide Time: 22:02)



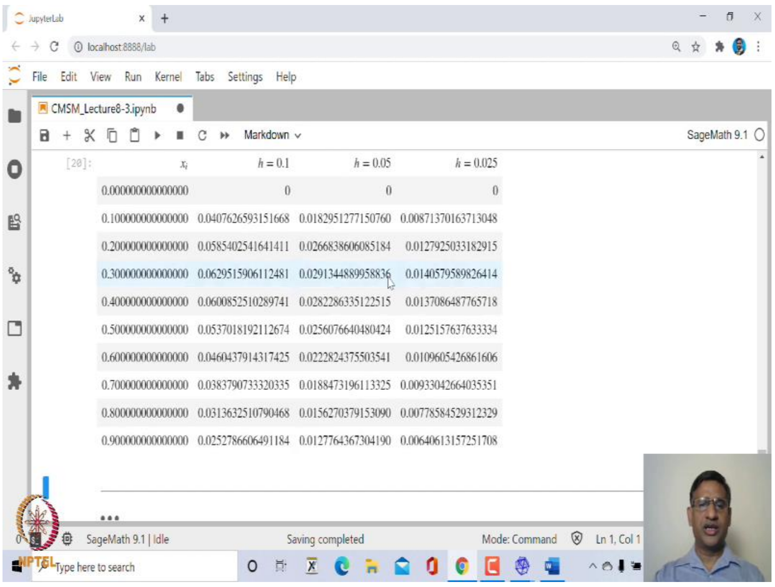
```
for (xi,yi) in T3:
    err = abs(yi-sol(x=xi))
    if(xi in nodes):
        E3.append(err)
```

```
[20]: table([[nodes[i],E1[i],E2[i],E3[i]] for i in range(10)],
          header_row=['$x_i$','$h=0.1$','$h=0.05$','$h=0.025$'])
```

[20]:

x_i	$h = 0.1$	$h = 0.05$	$h = 0.025$
0.000000000000000	0	0	0
0.100000000000000	0.0407626593151668	0.0182951277150760	0.00871370163713048
0.200000000000000	0.0585402541641411	0.0266838606085184	0.0127925033182915
0.300000000000000	0.0629515906112481	0.0291344889958836	0.0140579589826414
0.400000000000000	0.0600852510289741	0.0282286335122515	0.0137086487765718
0.500000000000000	0.0537018192112674	0.0256076640480424	0.0125157637633334
0.600000000000000	0.0460437914317425	0.0222824375503541	0.0109605426861606
0.700000000000000	0.0383790733320335	0.0188473196113325	0.00933042664035351

(Refer Slide Time: 22:09)



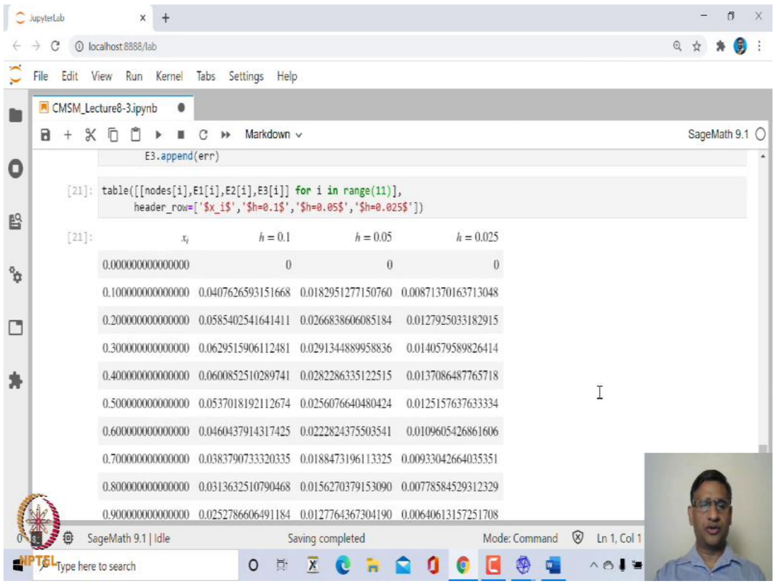
[20]:

x_i	$h = 0.1$	$h = 0.05$	$h = 0.025$
0.000000000000000	0	0	0
0.100000000000000	0.0407626593151668	0.0182951277150760	0.00871370163713048
0.200000000000000	0.0585402541641411	0.0266838606085184	0.0127925033182915
0.300000000000000	0.0629515906112481	0.0291344889958836	0.0140579589826414
0.400000000000000	0.0600852510289741	0.0282286335122515	0.0137086487765718
0.500000000000000	0.0537018192112674	0.0256076640480424	0.0125157637633334
0.600000000000000	0.0460437914317425	0.0222824375503541	0.0109605426861606
0.700000000000000	0.0383790733320335	0.0188473196113325	0.00933042664035351
0.800000000000000	0.0313632510790468	0.0156270379153090	0.00778584529312329
0.900000000000000	0.0252786606491184	0.0127764367304190	0.00640613157251708

Let us run this and let me tabulate this. This is what you get.

You can see here the error term at h equal to 0.1, h is equal to 0.05 and h equal to point 0.025. So, you can compare the error at every stage. These errors, we have computed only at node points 0.1, 0.2, 0.3. So, we have to say range 11 in this case.

(Refer Slide Time: 22:37)



```
        E3.append(err)
```

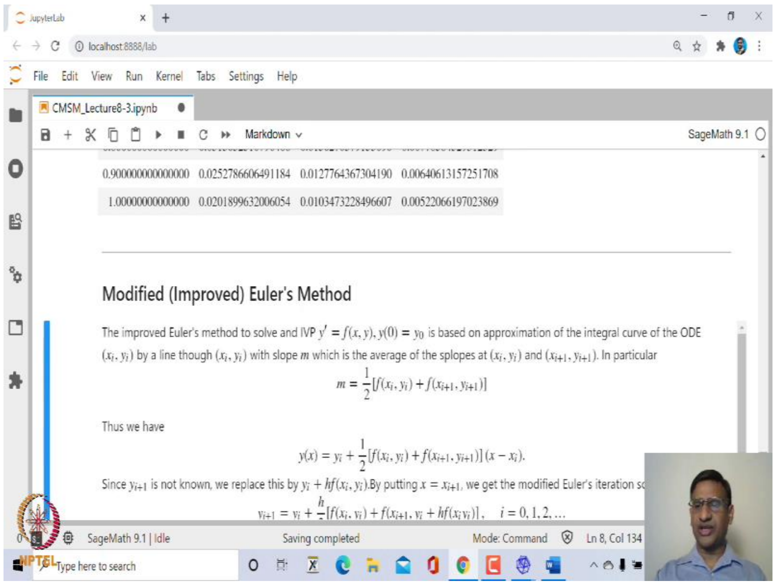
```
[21]: table([[nodes[i],E1[i],E2[i],E3[i]] for i in range(11)],
          header_row=['$x_i$','$h=0.1$','$h=0.05$','$h=0.025$'])
```

[21]:

x_i	$h = 0.1$	$h = 0.05$	$h = 0.025$
0.000000000000000	0	0	0
0.100000000000000	0.0407626593151668	0.0182951277150760	0.00871370163713048
0.200000000000000	0.0585402541641411	0.0266838606085184	0.0127925033182915
0.300000000000000	0.0629515906112481	0.0291344889958836	0.0140579589826414
0.400000000000000	0.0600852510289741	0.0282286335122515	0.0137086487765718
0.500000000000000	0.0537018192112674	0.0256076640480424	0.0125157637633334
0.600000000000000	0.0460437914317425	0.0222824375503541	0.0109605426861606
0.700000000000000	0.0383790733320335	0.0188473196113325	0.00933042664035351
0.800000000000000	0.0313632510790468	0.0156270379153090	0.00778584529312329
0.900000000000000	0.0252786606491184	0.0127764367304190	0.00640613157251708

So, now it will go up to 1, right! That is what we saw. Again , you can see here the error is almost becoming half of the error in previous case, the so this h is half of this and this h is half of this and error is also becoming half in this example.

(Refer Slide Time: 23:00)



0.900000000000000	0.0252786606491184	0.0127764367304190	0.00640613157251708
1.00000000000000	0.0201899632006054	0.0103473228496607	0.00522066197023869

Modified (Improved) Euler's Method

The improved Euler's method to solve and IVP $y' = f(x, y), y(0) = y_0$ is based on approximation of the integral curve of the ODE (x_i, y_i) by a line though (x_i, y_i) with slope m which is the average of the splopes at (x_i, y_i) and (x_{i+1}, y_{i+1}). In particular

$$m = \frac{1}{2}[f(x_i, y_i) + f(x_{i+1}, y_{i+1})]$$

Thus we have

$$y(x) = y_i + \frac{1}{2}[f(x_i, y_i) + f(x_{i+1}, y_{i+1})]\,(x - x_i).$$

Since y_{i+1} is not known, we replace this by $y_i + hf(x_i, y_i)$.By putting $x = x_{i+1}$, we get the modified Euler's iteration sc

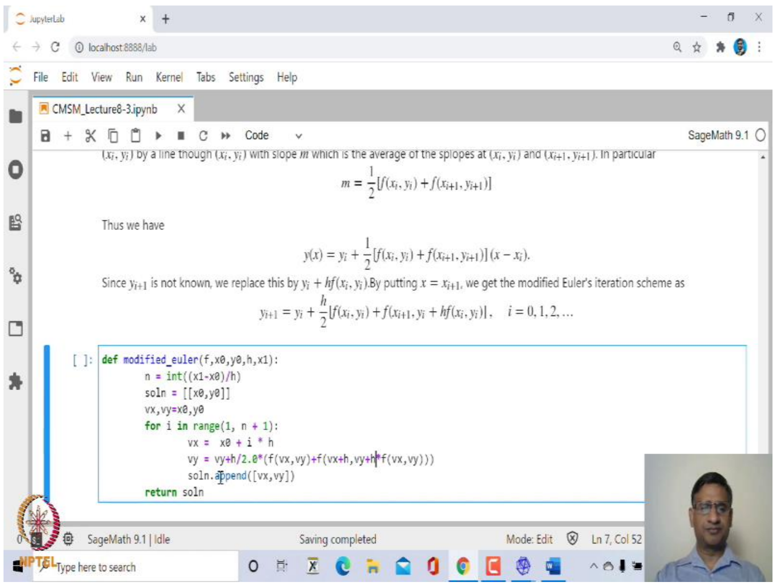
$$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, y_i + hf(x_i y_i)], \quad i = 0, 1, 2, \ldots$$

Now let us look at, you can make improvement in this Euler's method, in order to make the convergence faster or in order to reduce the error.

So, how do we do that? This is again fairly simple. Last time we approximated the value of y at xi, by the slope, f(xi, yi). But this time, what we do is, instead of taking the slope at (xi, yi), we take the average of the slope at (xi, yi) and the next point (x i+1, y i + 1). So, m is half of this and then we define the tangent at (xi, yi),with this as slope. So, this is that equation of the tangent at (xi, yi) with slope m.

But, if you notice that this requires value of f at x i plus 1, y at x i plus 1, but y at i plus 1 is not known.

(Refer Slide Time: 24:08)



(x_i, y_i) by a line though (x_i, y_i) with slope m which is the average of the splopes at (x_i, y_i) and (x_{i+1}, y_{i+1}). In particular

$$m = \frac{1}{2}[f(x_i, y_i) + f(x_{i+1}, y_{i+1})]$$

Thus we have

$$y(x) = y_i + \frac{1}{2}[f(x_i, y_i) + f(x_{i+1}, y_{i+1})](x - x_i).$$

Since y_{i+1} is not known, we replace this by $y_i + hf(x_i, y_i)$.By putting $x = x_{i+1}$, we get the modified Euler's iteration scheme as

$$y_{i+1} = y_i + \frac{h}{2}[f(x_i, y_i) + f(x_{i+1}, y_i + hf(x_i, y_i)], \quad i = 0, 1, 2, \ldots$$

```
[ ]: def modified_euler(f,x0,y0,h,x1):
         n = int((x1-x0)/h)
         soln = [[x0,y0]]
         vx,vy=x0,y0
         for i in range(1, n + 1):
             vx =  x0 + i * h
             vy = vy+h/2.0*(f(vx,vy)+f(vx+h,vy+h*f(vx,vy)))
             soln.append([vx,vy])
         return soln
```

We only know yi. Therefore, what we do is, we replace this by h times f(xi, yi), we replace it by this, there is a comma missing here. So, we replace this y i plus 1 by h f (xi, yi).

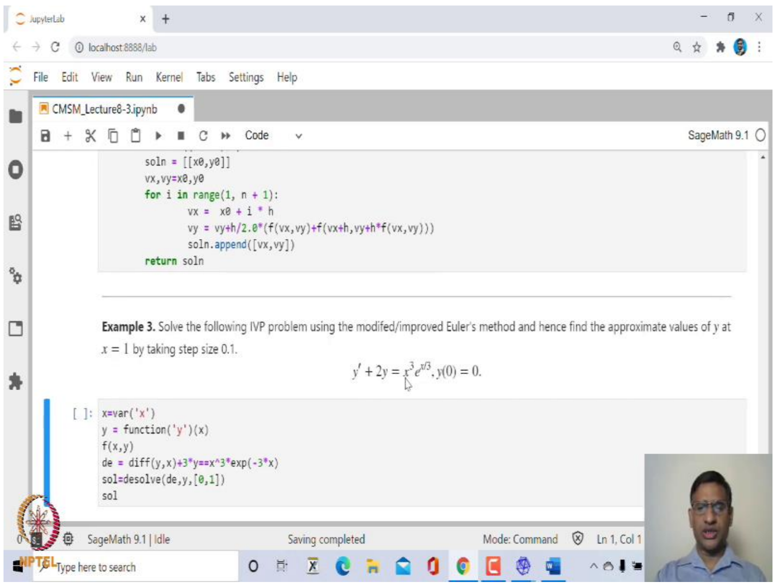
Therefore, this iteration scheme becomes

y i plus 1 is equal to y i plus h by 2 into f at xi yi, which is plus f at xi plus 1 comma y i plus h f at x i, y i.

This is the iteration scheme for modified Euler's method or improved Euler's method.

Let us implement this again. Let us write a Sage routine for this. This is exactly similar to the previous one, except that here computation of yi has changed.

(Refer Slide Time: 25:04)



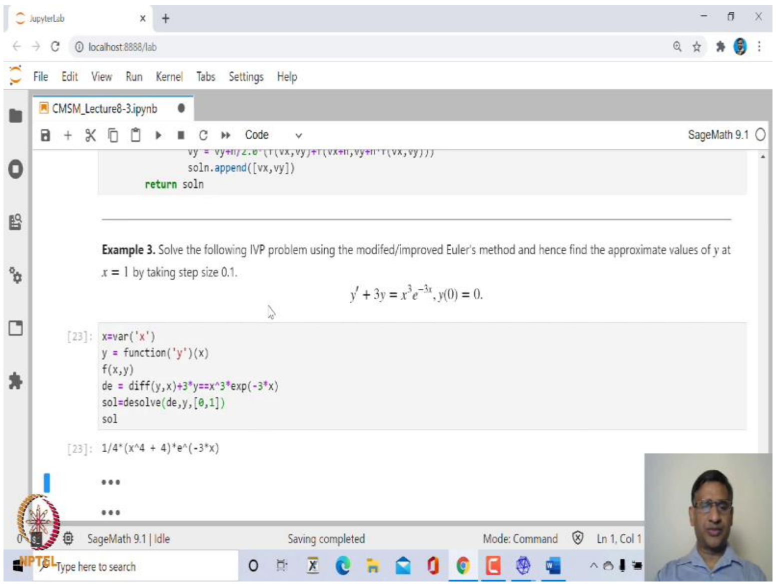
```
    soln = [[x0,y0]]
    vx,vy=x0,y0
    for i in range(1, n + 1):
            vx =  x0 + i * h
            vy = vy+h/2.0*(f(vx,vy)+f(vx+h,vy+h*f(vx,vy)))
            soln.append([vx,vy])
    return soln
```

Example 3. Solve the following IVP problem using the modifed/improved Euler's method and hence find the approximate values of y at $x = 1$ by taking step size 0.1.

$$y' + 2y = x^3 e^{x/3}, y(0) = 0.$$

```
[ ]: x=var('x')
     y = function('y')(x)
     f(x,y)
     de = diff(y,x)+3*y==x^3*exp(-3*x)
     sol=desolve(de,y,[0,1])
     sol
```

(Refer Slide Time: 25:19)



```
            soln.append([vx,vy])
    return soln
```

Example 3. Solve the following IVP problem using the modifed/improved Euler's method and hence find the approximate values of y at $x = 1$ by taking step size 0.1.

$$y' + 3y = x^3 e^{-3x}, y(0) = 0.$$

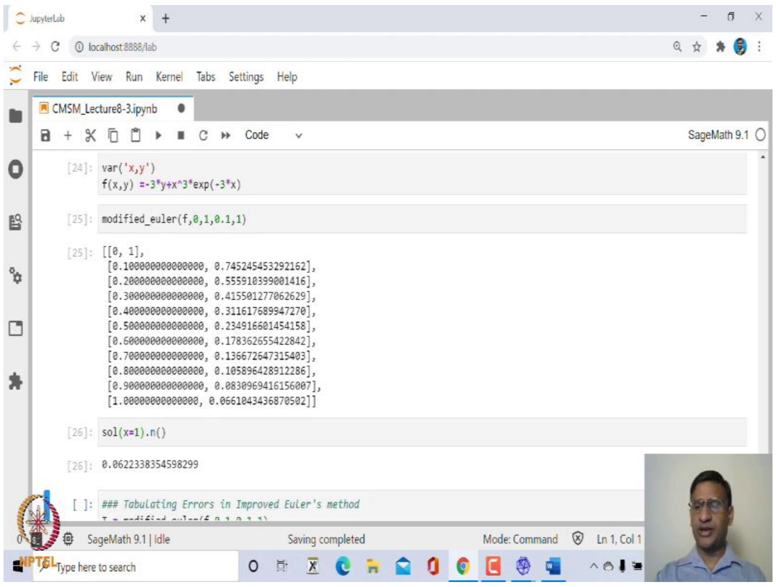
```
[23]: x=var('x')
      y = function('y')(x)
      f(x,y)
      de = diff(y,x)+3*y==x^3*exp(-3*x)
      sol=desolve(de,y,[0,1])
      sol

[23]: 1/4*(x^4 + 4)*e^(-3*x)
```

So, let us run this and then let us use this to solve this particular differential equation. This is the exactly the same as the previous one. So, again I should change this 2 to 3y and here to minus 3x. That is the differential equation. In the previous case also, I had made the appropriate changes.

This was actually minus 3x. It should be minus 3x. So, let us come back to this next example.

So, we want to solve the same differential equation using modified Euler's method, compare the two solutions. Again this is the analytic solution which we have already obtained.

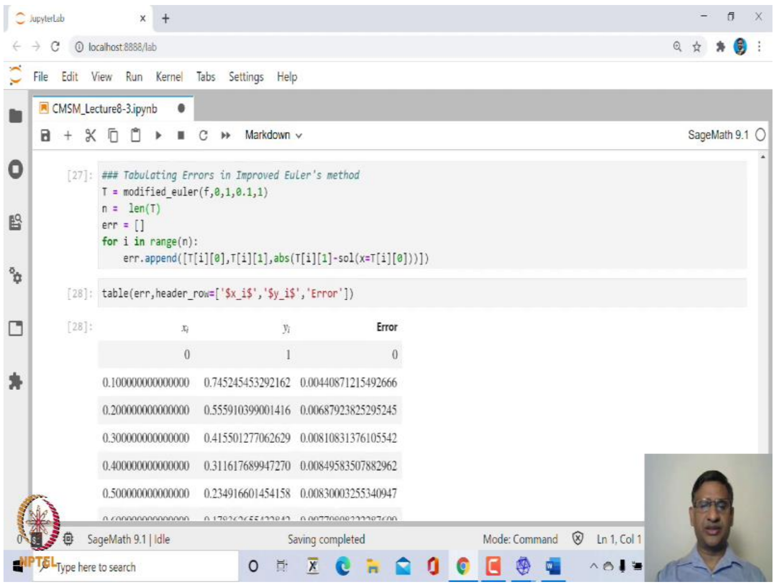
(Refer Slide Time: 26:02)



```
[24]: var('x,y')
      f(x,y) =-3*y+x^3*exp(-3*x)

[25]: modified_euler(f,0,1,0.1,1)

[25]: [[0, 1],
       [0.100000000000000, 0.745245453292162],
       [0.200000000000000, 0.555910399001416],
       [0.300000000000000, 0.415501277062629],
       [0.400000000000000, 0.311617689947270],
       [0.500000000000000, 0.234916601454158],
       [0.600000000000000, 0.178362655422842],
       [0.700000000000000, 0.136672647315403],
       [0.800000000000000, 0.105896428912286],
       [0.900000000000000, 0.0830969416156007],
       [1.00000000000000, 0.0661043436870502]]

[26]: sol(x=1).n()

[26]: 0.0622338354598299

[ ]: ### Tabulating Errors in Improved Euler's method
```

Now, let us let us define f(x,y) equals to minus 3 y plus x cube into e to power minus 3 x and then call this modified Euler's method. So, this is again giving you xi, yi and let us find out the exact value of the solution at x equal to 1, is this 0.622 and here also 0.61. So, it is actually matching up to two decimal places.

(Refer Slide Time: 26:35)



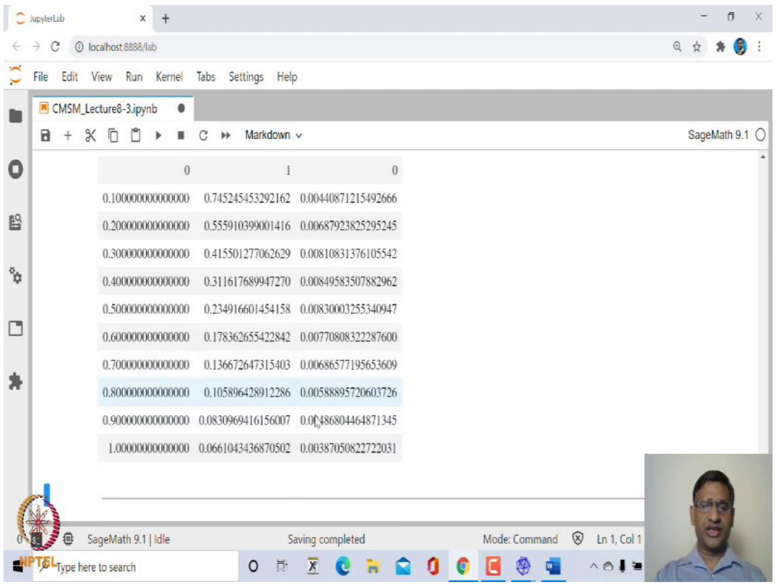
```
[27]: ### Tabulating Errors in Improved Euler's method
      T = modified_euler(f,0,1,0.1,1)
      n =  len(T)
      err = []
      for i in range(n):
          err.append([T[i][0],T[i][1],abs(T[i][1]-sol(x=T[i][0]))])

[28]: table(err,header_row=['$x_i$','$y_i$','Error'])
```

[28]:

x_i	y_i	Error
0	1	0
0.100000000000000	0.745245453292162	0.00440871215492666
0.200000000000000	0.555910399001416	0.00687923825295245
0.300000000000000	0.415501277062629	0.00810831376105542
0.400000000000000	0.311617689947270	0.00849583507882962
0.500000000000000	0.234916601454158	0.00830003255340947

And now let us tabulate this along with the error. So, exactly similar to previous one, except that we are calling modified Euler's method. So, let us tabulate the xi, yi with error.

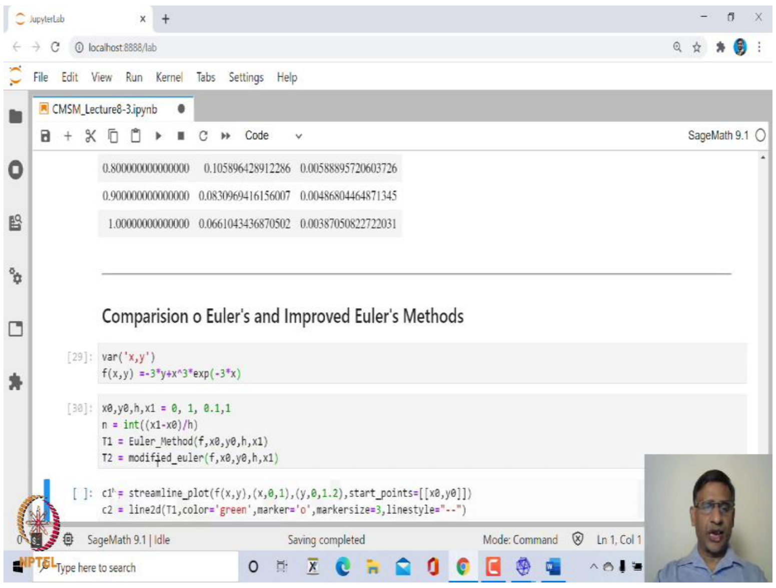
(Refer Slide Time: 26:53)



0	1	0
0.100000000000000	0.745245453292162	0.00440871215492666
0.200000000000000	0.555910399001416	0.00687923825295245
0.300000000000000	0.415501277062629	0.00810831376105542
0.400000000000000	0.311617689947270	0.00849583507882962
0.500000000000000	0.234916601454158	0.00830003255340947
0.600000000000000	0.178362655422842	0.00770808322287600
0.700000000000000	0.136672647315403	0.00686577195653609
0.800000000000000	0.105896428912286	0.00588895720603726
0.900000000000000	0.0830969416156007	0.00486804464871345
1.00000000000000	0.0661043436870502	0.00387050822722031

You can see this. This error if you compare with the Euler's method, it will be much smaller. And in fact, if you see here it is less than half.

(Refer Slide Time: 27:10)



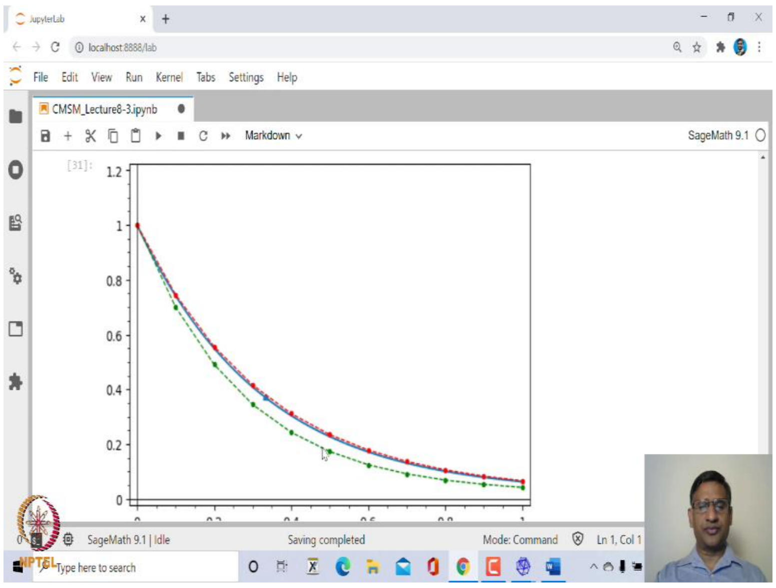
```
0.800000000000000   0.105896428912286  0.00588895720603726
0.900000000000000  0.0830969416156007  0.00486804464871345
 1.00000000000000  0.0661043436870502  0.00387050822722031
```

Comparision o Euler's and Improved Euler's Methods

```
[29]: var('x,y')
      f(x,y) =-3*y+x^3*exp(-3*x)

[30]: x0,y0,h,x1 = 0, 1, 0.1,1
      n = int((x1-x0)/h)
      T1 = Euler_Method(f,x0,y0,h,x1)
      T2 = modified_euler(f,x0,y0,h,x1)

[ ]:  c1 = streamline_plot(f(x,y),(x,0,1),(y,0,1.2),start_points=[[x0,y0]])
      c2 = line2d(T1,color='green',marker='o',markersize=3,linestyle="--")
```

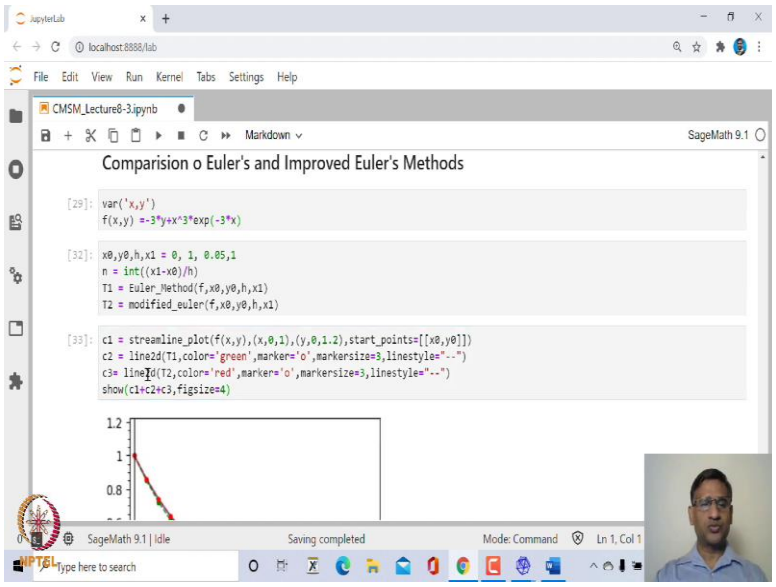
Now you can compare these wo methods. So, with the same function and let us try to tabulate T1 as the solution using Euler's method and T2 using modified Euler's method, and let us tabulate this two together and then plot the 2 graphs.

(Refer Slide Time: 27:33)



Here this blue one is the exact solution, the red one is solution obtained using modified Euler's method, and the green one is the solution obtained using Euler's method. You can see here, the green one is quite far away from the blue one.

(Refer Slide Time: 28:03)

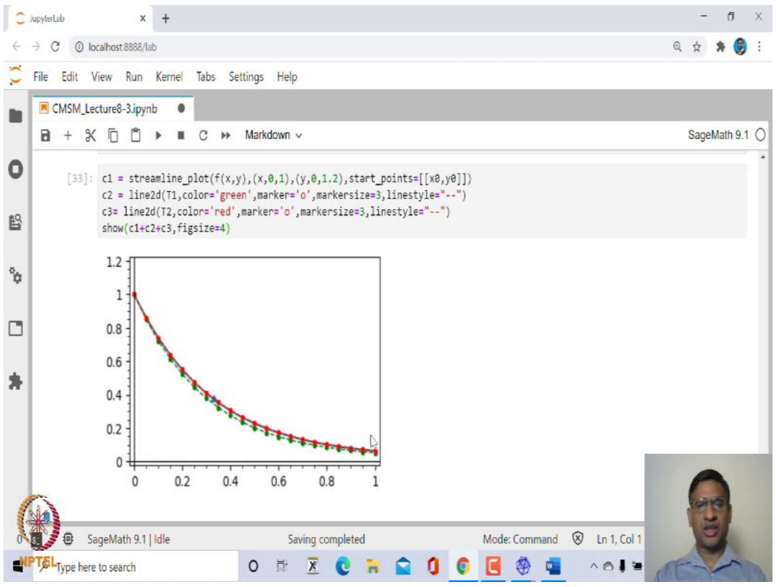


Comparision o Euler's and Improved Euler's Methods

```
[29]: var('x,y')
      f(x,y) =-3*y+x^3*exp(-3*x)

[32]: x0,y0,h,x1 = 0, 1, 0.05,1
      n = int((x1-x0)/h)
      T1 = Euler_Method(f,x0,y0,h,x1)
      T2 = modified_euler(f,x0,y0,h,x1)

[33]: c1 = streamline_plot(f(x,y),(x,0,1),(y,0,1.2),start_points=[[x0,y0]])
      c2 = line2d(T1,color='green',marker='o',markersize=3,linestyle="--")
      c3= line2d(T2,color='red',marker='o',markersize=3,linestyle="--")
      show(c1+c2+c3,figsize=4)
```

So, the error in case of Euler's method is much more. However, if you if you kind of reduce this, let me make it 0.05, and then try to let me make the figure size small, show this and fig size is equal to, let us say 4.

(Refer Slide Time: 28:24)

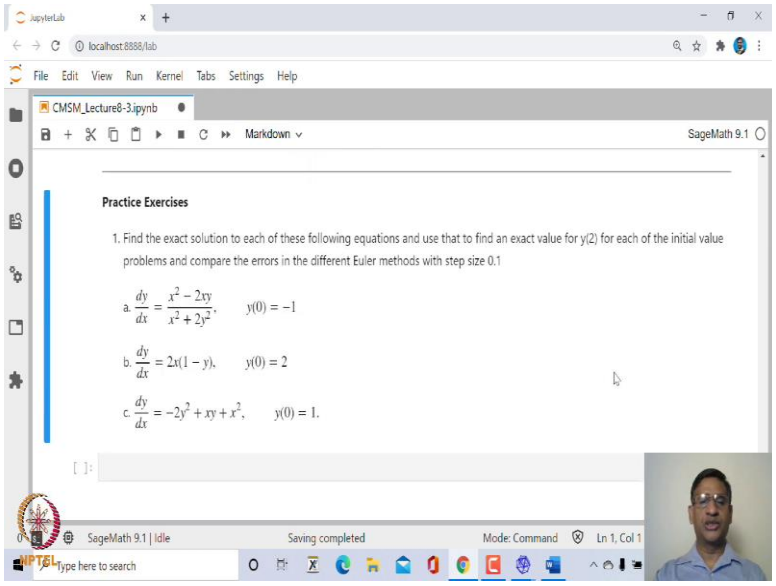


```
[33]: c1 = streamline_plot(f(x,y),(x,0,1),(y,0,1.2),start_points=[[x0,y0]])
      c2 = line2d(T1,color='green',marker='o',markersize=3,linestyle="--")
      c3= line2d(T2,color='red',marker='o',markersize=3,linestyle="--")
      show(c1+c2+c3,figsize=4)
```

Now you will see, they are much closer to each other, but still this Euler's method has lot more error right.

So, you can explore more examples, and try abulate error using Euler's method and Euler's error using modified Euler's method. So, it will also show you that the error in case of modified Euler's method is much smaller.

(Refer Slide Time: 28:55)



Practice Exercises

1. Find the exact solution to each of these following equations and use that to find an exact value for y(2) for each of the initial value problems and compare the errors in the different Euler methods with step size 0.1

   a. $\frac{dy}{dx} = \frac{x^2 - 2xy}{x^2 + 2y^2}, \qquad y(0) = -1$

   b. $\frac{dy}{dx} = 2x(1 - y), \qquad y(0) = 2$

   c. $\frac{dy}{dx} = -2y^2 + xy + x^2, \qquad y(0) = 1.$

[ ]:

Let me leave you with some few simple exercises. These are all exercises to solve this numerically using Euler's method and modified Euler's method. So use both of them and also tabulate the value xi yi and try to also plot the graph. So, take the different step size. That is the simple exercise.

Let me stop here. Next time we will look at another method which is known as RK-4 method.