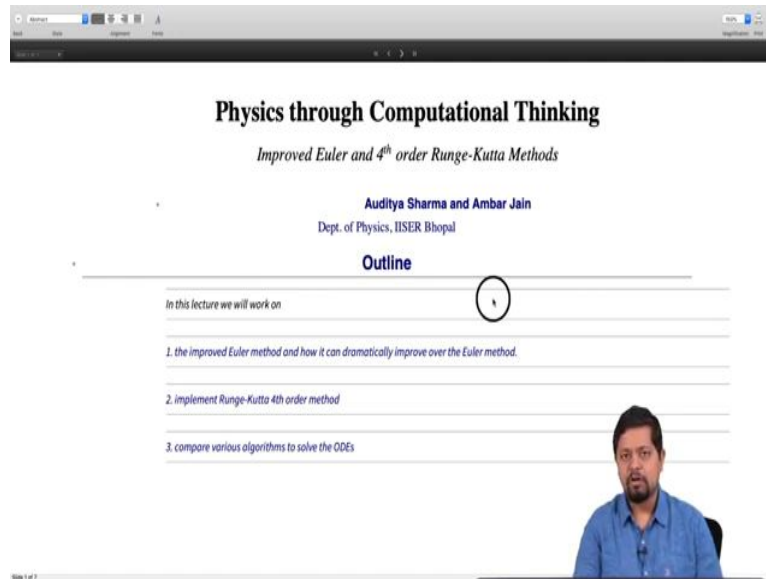


**Physics through Computational Thinking**  
**Professor Dr. Auditya Sharma**  
**Dr. Ambar Jain**  
**Department of Physics**  
**Indian Institute of Science Education and Research, Bhopal**  
**Lecture 29**  
**Improved Euler (RK2) and RK4 Methods for solving ODEs**

(Refer Slide Time: 00:28)



The screenshot shows a presentation slide with the following content:

- Physics through Computational Thinking**
- Improved Euler and 4<sup>th</sup> order Runge-Kutta Methods*
- Auditya Sharma and Ambar Jain
- Dept. of Physics, IISER Bhopal
- Outline**
- In this lecture we will work on
- 1. the improved Euler method and how it can dramatically improve over the Euler method.
- 2. implement Runge-Kutta 4th order method
- 3. compare various algorithms to solve the ODEs

A small video inset of a man in a blue shirt is visible in the bottom right corner of the slide.

Welcome back to Physics through Computational Thinking. Today we are going to talk about Improved Euler method. Last time we looked at Euler method and we saw that it takes to get an accurate calculation it takes a long time. So, look at the Improved Euler method which can get us desired accuracy in less number of steps or less number of, for a larger value of this step size  $h$ . We will further into this method using by using Range-Kutta Fourth order method.

Range-Kutta Fourth order method can get you even higher accuracy with even larger step size. And we will compare this methods for solving Ordinary Differential Equations. So, this will be a target of this particular lecture. So, let us go ahead and get started.

(Refer Slide Time: 01:13)

**Improved Euler's Method**

- Improved Euler's method improves the Euler method by reducing the local error to order  $h^3$  and global error to order  $h^2$ .
- This is how Improved Euler Method is defined:

$$\begin{aligned} t_{n+1} &= t_n + h \\ x_{n+1} &= x_n + h f(t_n, x_n) \\ x_{n+1} &= x_n + h \frac{f(t_n, x_n) + f(t_{n+1}, x_{n+1})}{2} \end{aligned} \quad (1)$$

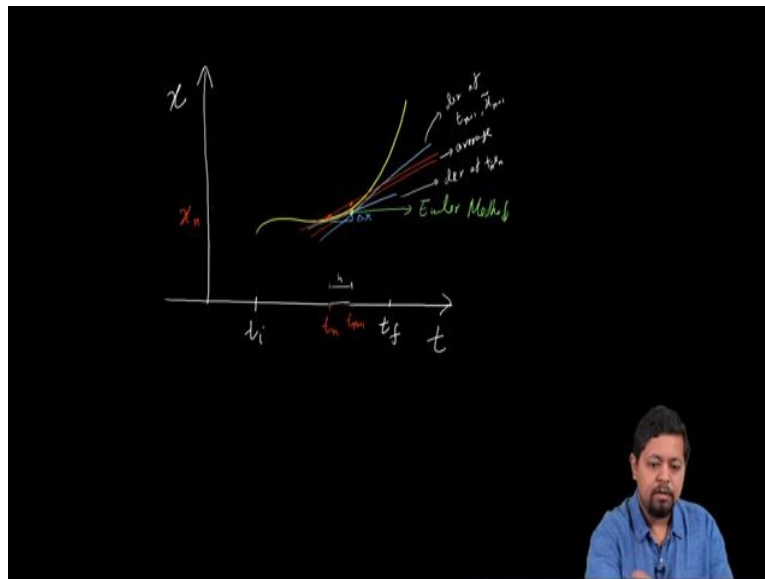
- This can also be written as

$$\begin{aligned} t_{n+1} &= t_n + h \\ x_{n+1} &= x_n + h \frac{f(t_n, x_n) + f(t_n + h, x_n + h f(t_n, x_n))}{2} \end{aligned} \quad (2)$$

- The improved Euler method is also known as the second-order Runge Kutta (RK) method.
- Implementation

```

in[ ] := eulerImp[F_, X0_, tf_, nMax_] :=
Module[{h, datalist, prev, next1, next, rate, rate1},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] < nMax,
      AppendTo[datalist, next1]
  ]
  ]
  
```



So, improved Euler method has a local error of order  $h^3$  and global error of order  $h^2$ . In comparison to Euler method which has a local error of order  $h^2$  and a global error of order  $h$ . This is slightly better because the local error here is order  $h^3$  and global error is order  $h^2$ .

So, let us see how Euler, improved Euler method defined, then I am going to give you a proof of this but we will illustrate it through a graphical image. So the (Euler), notice the difference in comparison to the Euler method. Time increment happens in the same way as the Euler method. And in the Euler method, we determined the next value of  $x$  by adding to the  $x$  and the previous value of  $x$   $h$  times derivative.

Evaluate the previous value  $t_n, x_n$ . So, this was the Euler method, the first two lines, define

the Euler method. In the improved Euler method, what we do is, we take, we determine  $x_{n+1}$  by adding to  $x_n$  improved value of the derivative. The improved value of the derivative is obtained by the value of the derivative at the previous point  $t_n$   $x_n$  plus the value of the derivative at the new point  $t_{n+1}$  and  $\tilde{x}_{n+1}$  is the value of  $x$ , that is determined by the Euler method.

So, this is the value of derivative at the previous point and here we have got the value of derivative at the point which is determined by the Euler method. We take the average of these two points and then multiply by the step size  $h$  and that gives me the step in  $x$ , I add that to  $x_n$  and that gives me the value of  $x_{n+1}$  according to the improved Euler method.

Now, the value of  $x_{n+1}$  determined this way is more accurate than value of the  $x_{n+1}$  determined by the Euler method that is by this step. I will give you a graphical proof of this or a graphical illustration of this rather. So, let us go ahead and take a look at that.

So, let us take  $x$  axis as time axis and  $y$  axis as the  $x$  axis and we want to find the solutions from some  $t_i$  to  $t_f$ . Now, let us say, our solution, let us say our true solution that is something that we assume to already know. Then the true solution let us say it looks like that from  $t_i$  to  $t_f$ . And this is some value of  $t_n$ .

This is  $t_n$  and corresponding to that this is  $x_n$ . I want to determine the next value, that is the  $t_n$  plus 1. This is  $t_{n+1}$ . The distance between  $t_n$  and  $t_{n+1}$  is  $h$  and I want to determine the value of the function over here using the derivatives. So, I for this problem the ODE, I know the derivatives I do not know the true function but for the illustration purpose I am drawing the true function assuming that I know it.

So, let us go ahead and see what Euler's method does and then we will compare with improved Euler's method. In the Euler's method, we take the derivative at  $t_n$ . The derivative is this, we multiply that derivative by  $h$ , so we, so multiply that derivative by  $h$ , that gives us the segment  $\Delta x$ .

This is  $h$  times  $f$  evaluated at  $t_n$  and  $x_n$  and that gives me the height change or the step size and so from this point of view, my determined value of  $x_n$  is over here. In fact, in order to make it more dramatic so that you can see the difference. Let us go ahead and make this

curve have a higher gradient.

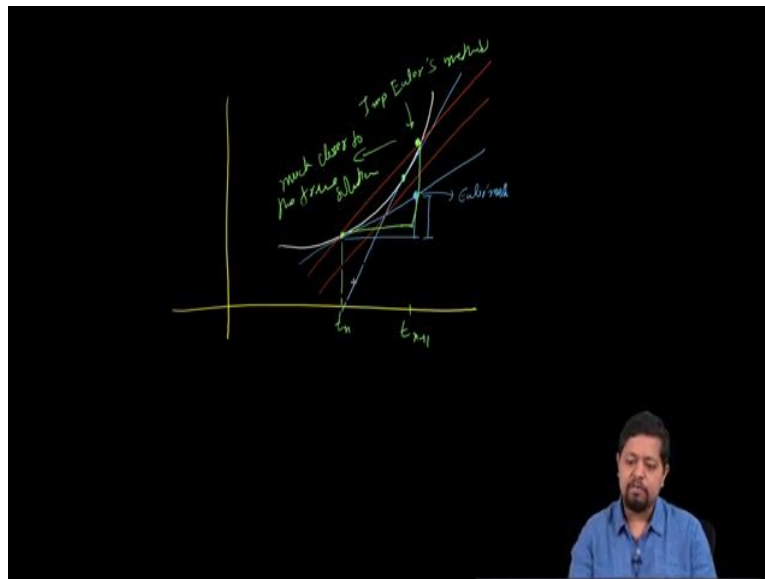
So, let us go ahead and say a function goes like that. And at this point  $t_n$ , the derivative goes like that. And for a distance  $h$ , this is  $h$ , I go ahead and I calculate  $\Delta x$ , so my new value, this is my value according to Euler method. So the improved Euler method says that, just do not take the first derivative but also take another derivative.

So, this was a derivative because the blue line is the derivative because of at the previous point but improved Euler method says that, take also the derivative, take also the second derivative which is evaluated at  $t_{n+1}$  and  $\tilde{x}_{n+1}$ .

So, this is  $t_{n+1}$  and  $\tilde{x}_{n+1}$ , at this point, the derivative has, has the value given by this line. Then the Euler's method says, that find the average of these two derivatives, which is given by the red line over here. so, let me label this. This is, this is derivative at  $t_n$  and  $x_n$ . This is derivative at  $t_{n+1}$   $\tilde{x}_{n+1}$  and this is the average of the two.

The average of the two, we take the average of the two, we translate that to the point over here, to the previous point and then we multiply the step size  $h$  in it and that gives me a value over here which answer being closer to the true value. So, let me do this again. Let me go ahead and erase this whole thing and let me do this, demonstrate this again.

(Refer Slide Time: 08:42)



**Improved Euler's Method**

- **Improved Euler's method** improves the Euler method by reducing the local error to order  $h^3$  and global error to order  $h^2$ .
- This is how Improved Euler Method is defined:

$$\begin{aligned}
 t_{n+1} &= t_n + h \\
 \tilde{x}_{n+1} &= x_n + h f(t_n, x_n) \\
 x_{n+1} &= x_n + h \frac{f(t_n, x_n) + f(t_{n+1}, \tilde{x}_{n+1})}{2}
 \end{aligned}
 \tag{1}$$

- This can also be written as

$$x_{n+1} = x_n + h \frac{f(t_n, x_n) + f(t_n + h, x_n + h f(t_n, x_n))}{2}
 \tag{2}$$

- The improved Euler method is also known as the second-order Runge Kutta (RK) method.
- **Implementation**

```

tr[ ] := eulerImp[F_, X0_, tf_, nMax_] :=
Module[{h, datalist, prev, next1, next, rate, ratel},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] < nMax,
    AppendTo[datalist, next1

```

Let us say this is the function, let us make h very big so that it is more clear, this is my point  $t_n$  and let us take  $t_{n+1}$  farther away. This is  $t_{n+1}$ , so Euler's method says, take the derivative over here, take the h multiply by that and this is the predicted point new predicted point by the Euler's method. So, this is the solution because of Euler's method. Now, improved

Euler's method says, that also take the derivative at this point.

This point that is the  $t_{n+1}$ . So, let me actually correct that. So, at  $t_{n+1}$ , the functions value is over here. so, let me take the derivative at this point and find the average of the two derivatives, which is this translate that to this point and then multiply by  $h$  which is travel the distance  $h$  so, therefore it lands up somewhere over here.

So, you see that the blue value and this blue dot and this green dot. Blue dot is further away from the true value where the green dot is solution or predicted point because of improved Euler's method and this is much closer to the true solution.

So, we see that improved Euler's method can significantly improve over the Euler's method. And this is just you take derivative at previous point, derivative at the predicted point and then calculate a new derivative of the average of these derivatives and then take predict the point again because of this average derivatives and this new value turns out to be much more accurate.

Alright, so let us go back and see how to implement this. This is my improved Euler's method. I can also write improved Euler's method in the following way. In so,  $t_{n+1}$  is  $t_n + h$  is before and  $x_{n+1}$  is given by  $x_n$  plus  $h$  times this fraction. This fraction is average of two derivative, the first one determined at the previous point and the second one determined at  $t_n + h$  and  $x_n + hf(t_n, x_n)$ .

This is nothing but  $x_{n+1}$  determined by the Euler's method. So, we can go ahead and implement this in one single we can write this in a single line and this is what we will use to implement. And implementation is very similar to what we have done before.

(Refer Slide Time: 12:19)

$$x_{n+1} = x_n + h f(t_n, x_n) \quad (1)$$

$$x_{n+1} = x_n + h \frac{f(t_n, x_n) + f(t_n + h, x_n + h f(t_n, x_n))}{2} \quad (2)$$

• This can also be written as

$$x_{n+1} = x_n + h \frac{f(t_n, x_n) + f(t_n + h, x_n + h f(t_n, x_n))}{2} \quad (2)$$

• The improved Euler method is also known as the second-order Runge Kutta (RK) method.

• Implementation

```

In[ ]:= eulerImp[F_, X0_, tF_, nMax_] :=
Module[{h, datalist, prev, next1, next, rate, rate1},
  h = (tF - X0[[1]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] <= nMax,
    AppendTo[datalist, next],
    prev = Last[datalist];
    rate = Through[F @@ prev];
    next1 = prev + h rate;
    rate1 = Through[F @@ next1];
  ]

```

```

In[403]:=
eulerImp[F_, X0_, tF_, nMax_] :=
Module[{h, datalist, prev, next1, next, rate, rate1},
  h = (tF - X0[[1]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] <= nMax,
    AppendTo[datalist, next],
    prev = Last[datalist];
    rate = Through[F @@ prev];
    next1 = prev + h rate;
    rate1 = Through[F @@ next1];
    next = prev + h/2 (rate + rate1);
  ];
  Return[datalist];
]

```

So, this time I will call the functions improved Euler method just to make a note that improved Euler method is also known as the second order Range-Kutta method. So to improve to define improved Euler method, we will work exactly as in the case of the Euler method but we will make one distinction. We will introduce few more variables, local variables in the module construct. We will introduce next1, next, rate1 and rate.

So let us let us see how this works, h is calculated as before,  $\frac{(t_f - t_i)}{n} Max$ . Initialization of data list is as before. Condition checking is as before and incrementation is also as before, to data list we add the next point but in the body we calculate the next point and this is how we do it. These are the 4 steps of the body or 5 steps of the body which is where we are calculating the

next step.

So, first in order to calculate the next step, we find out the previous step, previous step is the last element of data list then we calculate the rate by using the Through function, applying F on the previous point. F is the set of derivatives, applying that on the previous point. Then next is determined by next1 is determined by previous plus h rate. This is nothing but the determination because of Euler method.

Now, we determine a new rate called the rate1 by finding out the derivative at the point next1 using Through function F applied at next1 and then we find the next point according to improved Euler method by taking average of the rate and rate1 and multiplying by h and adding previous to it. Once you got that in over here, it is appended to data list and so on. So this is the implementation of Euler method, the only changes here in the body and we return the data list.

So, let me go ahead and execute that and that defines my Euler method. Let us go and check out this Euler method, if it provides significant improvement over Improved Euler method, if it provides significant improvement over Euler's method.



(Refer Slide Time: 14:31)

**Application of Improved Euler to Solve Damped Oscillator**

• We want to solve the IVP:

$$\begin{aligned} \frac{dQ}{dt} &= I \\ \frac{dI}{dt} &= -\frac{L}{R^2 C} Q - I \end{aligned} \quad (1)$$

Initial conditions:  $Q(0) = 1$ ,  $I(0) = 0$

• Implementation: Lets take the ratio  $w = L/(R^2 C)$

```

In[ ]:= w = 10;
beta = Sqrt[w - 1/4];
2.0 Pi
beta
id[t_, charge_, current_] = 1;
chargeDot[t_, charge_, current_] = current;
currentDot[t_, charge_, current_] = -w charge - current;
initial = {0, 1, 0};

```

Out[ ]:= 2.01223

```

In[ ]:= data = eulerImp[{id, chargeDot, currentDot}, initial, 10, 100];
In[ ]:= Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[e^{-t/2} Cos[beta t], {t, 0, 10}, PlotRange -> Full]

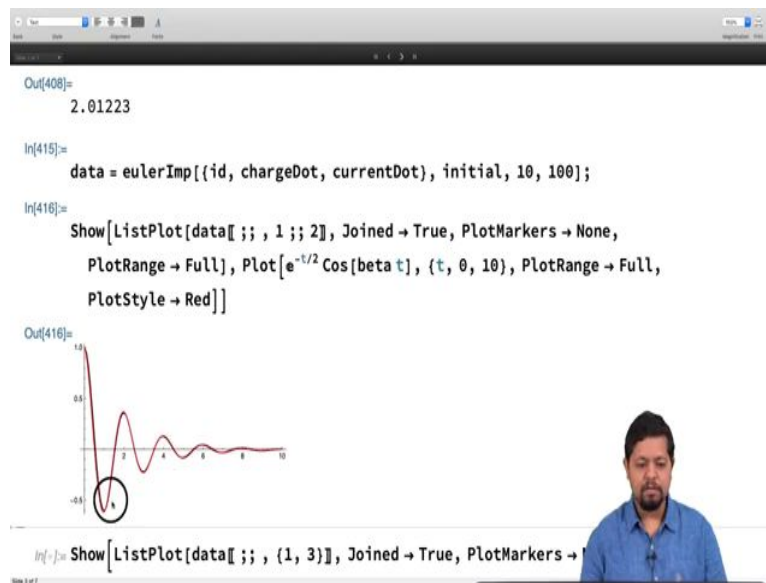
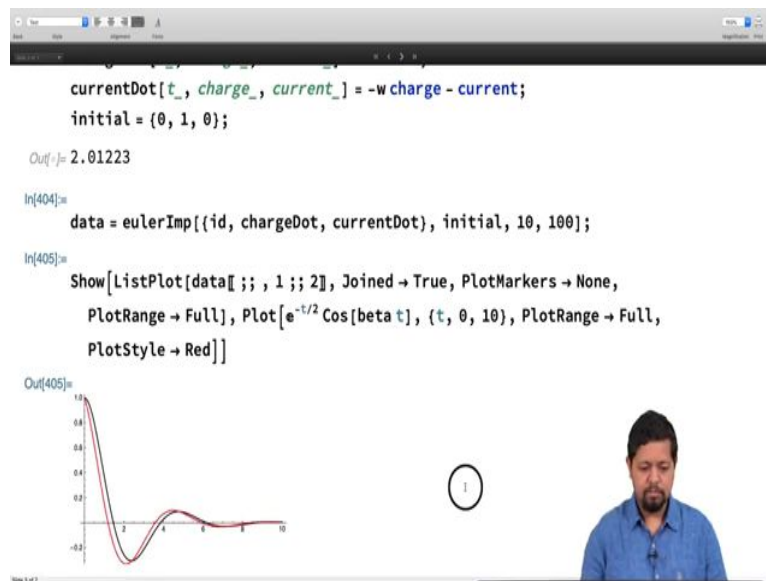
```

So here is the application of Improved Euler method for solving the damped oscillator. Just remind you damp oscillator equation was  $\frac{dQ}{dt} = I$ . And  $\frac{dI}{dt} = \frac{L}{R^2 C} Q - I$ .

This is the term that is responsible for oscillation and minus I is responsible for damping, initial conditions was  $Q(0) = 1$ . And  $I(0) = 0$  and we decided to take the ratio  $\frac{L}{R^2 C} = w$  and it shows the large value at  $w$ ,  $w \gg \frac{1}{4}$ . So, this is what we have taken from our code that we wrote last time.

So, we take  $w = 10$ ,  $\beta = \sqrt{w - \frac{1}{4}}$  then rest is same as before. We define the identity function chargeDot and the current dot and initial value at  $t = 0, Q = 1, I = 0$ .

(Refer Slide Time: 15:35)



And now calculate data with improved Euler function that we have just defined rather than Euler general function that we defined last time. So, let me go ahead and do that. And when I do this, here is what I get. So, let me go ahead and this time evaluate or calculate my data points using the improved Euler method this is same call as before, except that rather than using the original method, I am using the improved Euler method.

So let me go and execute that and then I am going to do a comparison of plots over here and we see that this is a fantastic agreement between the red curve and the black curve.

Where red curve is the expected solution and black curve is the numerical solution that we have obtained using the improved Euler method.

(Refer Slide Time: 16:39)

The image shows two screenshots of a Mathematica notebook. The top screenshot displays the following code and output:

```

initial = {0, 1, 0};
Out[408]=
2.01223
In[417]=
data = eulerGen[{id, chargeDot, currentDot}, initial, 10, 100];
In[418]=
Show[ListPlot[data[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[e^{-t/2} Cos[beta t], {t, 0, 10}, PlotRange -> Full,
PlotStyle -> Red]]
Out[418]=

```

The plot shows a black oscillating curve and a red curve that does not match it well. A circled '1' is visible in the plot area.

The bottom screenshot displays the following code and output:

```

In[425]=
data = eulerImp[{id, chargeDot, currentDot}, initial, 10, 100];
In[426]=
Show[ListPlot[data[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[e^{-t/2} Cos[beta t], {t, 0, 10}, PlotRange -> Full,
PlotStyle -> Red]]
Out[426]=

```

The plot shows a black oscillating curve and a red curve that closely matches it. A circled '1' is visible in the plot area.

The bottom part of the notebook shows the following code:

```

In[ ]:= Show[ListPlot[data[;;, {1, 3}], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[-\frac{1}{2} e^{-t/2} Cos[t beta] - e^{-t/2} beta Sin[t beta], {t, 0, 10}, PlotRange -> Full, PlotStyle -> Red]]

```

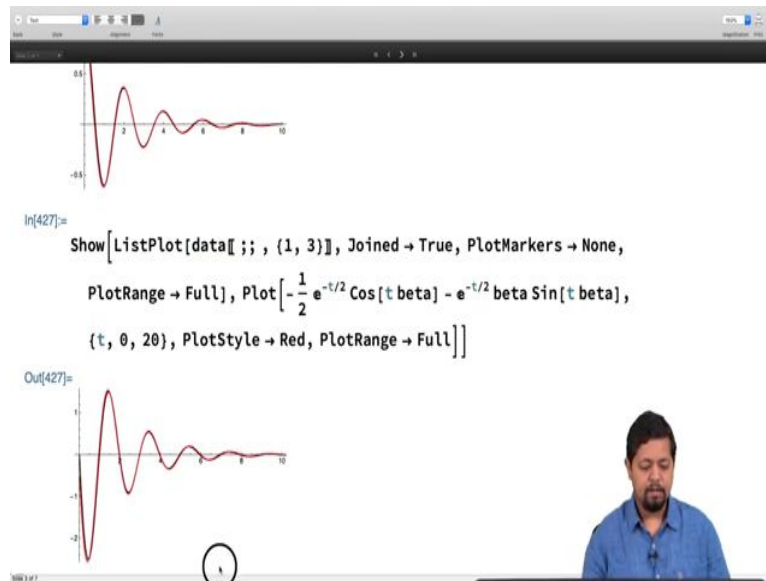
Just to show you that, what happens when you use the original method that we use previous last time. We see that we get a poor agreement. For  $n_{\text{Max}} = 100$ . Notice that we are using  $n_{\text{Max}} = 100$ . We are getting a poor agreement with Euler general function.

We had to go to higher value of  $n_{\text{max}}$  with the Euler general function to get some kind of agreement but the moment we go to improved Euler method, we can do this with much lesser points with only about 100 points we can get a fantastic agreement. Much better agreement with the expected curve [compares to the Euler improved method which completely fails.

Sorry, Euler general method which completely fails for 100 points. To show you again, is the Euler general method with  $n_{\text{max}}$  equal to 100 and  $t_f$  equal to 10 therefore,  $h$  is 0.1. For  $h$

equal to 0.1, we see that the, the Euler general method gives me a oscillation without any damping which is completely incorrect. While Euler improved method gives a fantastic agreement. You can play around with this.

(Refer Slide Time: 17:52)



We can also compare the current, so this is the call for current where I extract from data time and the first and the third component or the first and the third column and that also is in fantastic agreement. So, this is current versus time and this is charge versus time.

(Refer Slide Time: 18:14)

Implementation: Lets take the ratio  $w = L/(R^2 C)$

```
In[428]:=
w = 20;
beta = Sqrt[w - 1/4];
2.0 Pi
beta
id[t_, charge_, current_] = 1;
chargeDot[t_, charge_, current_] = current;
currentDot[t_, charge_, current_] = -w charge - current;
initial = {0, 1, 0};

Out[430]=
1.41383

In[425]:=
data = eulerImp[{id, chargeDot, currentDot}, initial, 10, 100];

In[426]:=
Show[ListPlot[data[[;;, 1, 3]], Joined -> True, PlotMarkers -> None,
```

---

```
In[437]:=
data = eulerImp[{id, chargeDot, currentDot}, initial, 10, 200];

In[438]:=
Show[ListPlot[data[[;;, 1, 3]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[e^-t/2 Cos[beta t], {t, 0, 10}, PlotRange -> Full,
PlotStyle -> Red]]

Out[438]=


In[427]:=
Show[ListPlot[data[[;;, {1, 3}], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[-1/2 e^-t/2 Cos[t beta] - e^-t/2 beta Sin[t be
```

So, let us go ahead and try a different value of  $w$ . Let us go ahead and say, make  $w = 20$ , so that we see a few more oscillations, we will keep  $n_{\text{Max}}$  as 10 as 100 and  $t_f$  as 10. Let us go and try this out and we see that this is again in good agreement.

But now that number of oscillations are higher, we need to increase this further so let us go and make this 200. So, with  $n_{\text{Max}} = 200$ , let us go and try this out and we see this is a fantastic agreement again. And there is something that will be impossible with Euler general method. So, you see that Euler's improved method provides significantly significant improvement over Euler general method or Euler method. So, let us go ahead and further improve this using the Fourth order Runge-Kutta method.

(Refer Slide Time: 19:04)

**The Fourth-order Runge-Kutta Method**

- The fourth-order Runge-Kutta method provides a significant improvement in accuracy, giving a local error of the order  $h^5$  while the global error is order  $h^4$ .
- When the efficiency increases, the complexity of the method also increases. The RK4 method is often taken to provide a good balance between efficiency and complexity.
- RK4 method is given by the following prescription:

$$\begin{aligned} t_{n+1} &= t_n + h \\ k_1 &= h f(t_n, x_n) \\ k_2 &= h f\left(t_n + \frac{h}{2}, x_n + \frac{1}{2}k_1\right) \\ k_3 &= h f\left(t_n + \frac{h}{2}, x_n + \frac{1}{2}k_2\right) \\ k_4 &= h f(t_n + h, x_n + k_3) \end{aligned} \quad (4)$$

$$x_{n+1} = x_n + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \quad (5)$$

- In order to implement it efficiently we will rephrase the method's algorithm in terms of rates  $r_1, r_2$  etc. rather than shifts  $k_1, k_2$  etc.

$$\begin{aligned} t_{n+1} &= t_n + h \\ r_1 &= \frac{k_1}{h} = f(t_n, x_n) \\ r_2 &= \frac{k_2}{h} = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}r_1\right) \\ r_3 &= \frac{k_3}{h} = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}r_2\right) \\ r_4 &= \frac{k_4}{h} = f(t_n + h, x_n + hr_3) \end{aligned} \quad (6)$$

$$x_{n+1} = x_n + h \frac{r_1 + 2r_2 + 2r_3 + r_4}{6} \quad (7)$$

- In order to implement it efficiently we will rephrase the method's algorithm in terms of rates  $r_1, r_2$  etc. rather than shifts  $k_1, k_2$  etc.

$$\begin{aligned} t_{n+1} &= t_n + h \\ r_1 &= \frac{k_1}{h} = f(t_n, x_n) \\ r_2 &= \frac{k_2}{h} = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}r_1\right) \\ r_3 &= \frac{k_3}{h} = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}r_2\right) \\ r_4 &= \frac{k_4}{h} = f(t_n + h, x_n + hr_3) \end{aligned} \quad (6)$$

$$x_{n+1} = x_n + h \frac{r_1 + 2r_2 + 2r_3 + r_4}{6} \quad (7)$$

- This was for the case of one dynamical quantity. When we have several dynamical quantities, where we expressed set of equations in the matrix form  $\dot{X} = F(X)$  where  $X = (t, x, y, z, \dots)^T$ . The RK4 method can be written as

$$\begin{aligned} R_1 &= F(X_n) \\ R_2 &= F\left(X_n + \frac{h}{2}R_1\right) \\ R_3 &= F\left(X_n + \frac{h}{2}R_2\right) \\ R_4 &= F(X_n + hR_3) \end{aligned} \quad (8)$$

$$X_{n+1} = X_n + h \frac{R_1 + 2R_2 + 2R_3 + R_4}{6} \quad (9)$$

- Now the implementation of RK4 method is straightforward:

```
Clear["Global`*"]
```

Fourth order Range-Kutta method is generalisation of improved Euler method or the second order Range-Kutta method. In fourth order Range-Kutta method, local error goes like order  $h^5$  while the global error goes as order  $h^4$  and that is the reason this is called the 4<sup>th</sup> order Range-Kutta method.

The global error reduces down to  $h^4$ . So, in the Euler method, it is  $h$ , in improved Euler method or second order Runge-Kutta, it is  $h^2$  and the fourth order Runge-Kutta method is  $h^4$ . The idea behind the fourth order Runge-Kutta method is similar to the improved Euler method. Again we find a more accurate value of the derivative that will give me a better prediction for the next evaluation of the of the step size.

So, this is how it works in 1 dimension,  $t_{n+1} = t_n + h$ . Then we calculate  $k_1 = hf(t_n, x_n)$ . So, this is the expected  $\Delta x$ . Then we calculate  $k_2 = hf(t_n + \frac{h}{2}, x_n + \frac{1}{2}k_1)$ . So,  $k_1$  was the step in  $\Delta x$  it was expectation in  $\Delta x$  or it was an estimations of  $\Delta x$  and that estimation of  $\Delta x$  is being used here to evaluate new value of  $k_2$ .

So,  $k_2$  is again and again another estimation of  $\Delta x$ . And that is used to evaluate  $k_3$ ,  $k_3 = hf(t_n + \frac{h}{2}, x_n + \frac{1}{2}k_2)$ , where  $k_2$  was another expectation or another estimation of  $\Delta x$ . And then  $k_4 = hf(t_n + h, x_n + k_3)$ . So, this is another so  $k_3$  was another estimation of  $\Delta x$  and  $k_4$  is another estimation of  $\Delta x$ . So, there are 4 different estimations of  $\Delta x$  I calculated over here.

$k_1, k_2, k_3$  and  $k_4$  then  $x_{n+1}$  is evaluated as taking the weighted average of these 4 expectations of  $\Delta x$  or 4 estimations of  $\Delta x$ ,  $k_1, k_2, k_3$  and  $k_4$ . So, weighted average  $k_1$  has a multiplier factor 2,  $k_2$  is a multiplier factor,  $k_1$  has a multiplier factor 1,  $k_2$  and  $k_3$  is a multiplier factor 2, and  $k_4$  is a multiplier factor 1.

We add all the  $k$ 's and then divide by 6, this the weighted average of all those estimations of  $\Delta x$  we add that to  $\Delta x$ . And that gives me determination of  $x_{n+1}$ . So, we are now going to present the proof of this, even graphical representation is bit tricky which you invited to try on your own graphical representation.

One will prove this mathematically, where the proof is extremely lengthy, so what we are going to do is, we are not going to go over the mathematical proof is not in the interest of this course. But what we will do is, we will take this for its face value, we will evaluate it and try it out and see how it provides a improvement.

So, let us go ahead and implement this. In order to implement this, what we will do is, we will redefine this Runge-Kutta method in the following way in terms of rather than estimations of  $x_{n+1}$ , what we will do is, we define a terms of rates because that is how we

have implemented our code.

So, what we will do is, we will define  $r_1$  as  $k\frac{1}{h}$ , that is,  $r_1$  is my first rate, that is evaluated at  $(t_n, x_n)$ . Then  $r_2$  is another estimation of rate that is evaluated at  $t_n + \frac{h}{2}$  and  $x_n + (\frac{h}{2})r_1$ ,  $(\frac{h}{2})r_1$  is estimation for  $\Delta x$ . Then  $r_3$  is again  $t_n + \frac{h}{2}$  and  $x_n + (\frac{h}{2})r_2$  and  $r_4$  is  $k\frac{4}{h}$  which is  $f$  evaluated at  $t_n + h$  and  $x_n + hr_3$ .

So, now taking these 4 rates, we will find the weighted average of rates by using this formula. Multiply that with  $h$  and add to  $x_n$ . So, let us go ahead and use this to implement our code. Now this was done for 1 dimensional problem. For a general  $n$  dimensional problem, we have got multiple dynamical quantities and our equation can be written in matrix form as  $\dot{X} = Fx$  as we discussed before and we can take  $x$  is the transpose of  $t, x, y, z$  etc.

Then RK4 method, we can write in terms of this matrix equations as follows.  $r_1$  is  $f$  evaluated  $x_n$ ,  $r_2$  is  $f$  evaluated at  $x_n + (\frac{h}{2})r_1$ ,  $r_3$  is  $f$  evaluated  $x_n + (\frac{h}{2})r_2$ . And  $r_4$  is  $f$  evaluated  $x_n + hr_3$ . Then we calculate  $x_{n+1}$  as weighted average of these 4 rates with each of these rates the matrix or column vector and we multiply that with  $h$ , add it to  $x_n$ . So, that gives us  $x_{n+1}$ . So, let us go ahead and implement this RK4 method.



(Refer Slide Time: 24:42)

```
In[441]:= Clear["Global`*"]

rk4[F_, X0_, tf_, nMax_] :=
Module[{h, datalist, prev, rate1, rate2, rate3, rate4, next},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] <= nMax,
    AppendTo[datalist, next],
    prev = Last[datalist];
    rate1 = Through[F @@ prev];
    rate2 = Through[F @@ (prev +  $\frac{h}{2}$  rate1)];
    rate3 = Through[F @@ (prev +  $\frac{h}{2}$  rate2)];
    rate4 = Through[F @@ (prev + h rate3)];
    next = prev +  $\frac{h}{6}$  (rate1 + 2 rate2 + 2 rate3 + rate4);
  ];
Return[datalist];
```

To do that, I am going to clear everything that in there, this clear command just take of the fact that, older definitions are all erased. Just in case there are some older definitions that can conflict with what we are going to do now. So, just erase all the older definitions that are stored in mathematica context called global. This will clear that up. And we go ahead and define RK4 method.

So, here is RK4 method. This is same as the Euler method or the Euler improve method. The only difference here is that is going to be in the body again. So, let us just review the body part. Here is the body of the follow which is being changed.

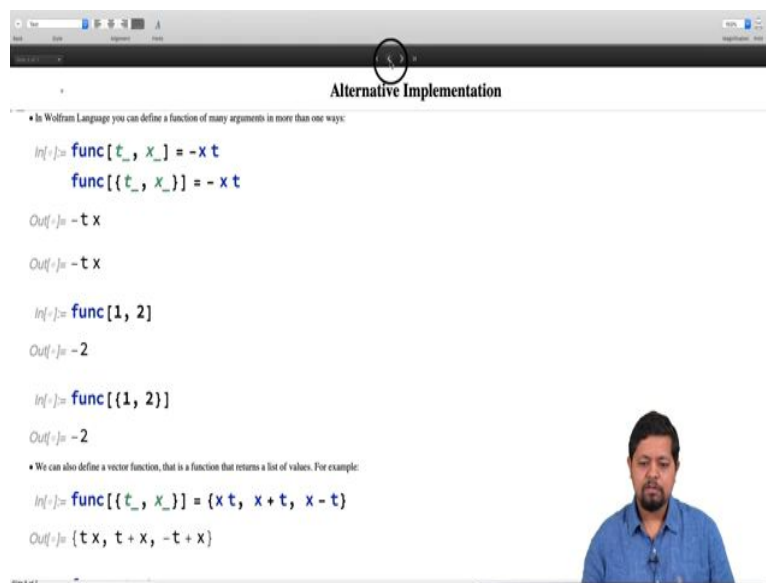
So, previous value is the last of the data list. Rate1 is F applied on previous, rate2 is f applied

on previous plus  $h/2$  times rate1. Rate3 is F applied previous plus  $h/2$  times rate2. And rate4 is F applied previous plus  $h$  times rate3.

And then we take the weighted average of the rates, multiplied by  $h$  and add to previous. So, it is as simple as that go ahead and write down this code yourself, implement it and once you are done, you can we can test it out.

So, this is a pretty much straight forward generalization of the Euler methods. Let me go ahead and execute that so that is loaded in the context and let us go ahead and when you are ready, let us go ahead and so, before we actually try it out, I want to simplify a few things.

(Refer Slide Time: 26:19)



The screenshot shows a Mathematica notebook window titled "Alternative Implementation". The notebook content includes the following text:

- In Wolfram Language you can define a function of many arguments in more than one way:  
`In[ ]:= func[t_, x_] = -x t`  
`func[{t_, x_}] = -x t`  
`Out[ ]:= -t x`  
`Out[ ]:= -t x`
- `In[ ]:= func[1, 2]`  
`Out[ ]:= -2`
- `In[ ]:= func[{1, 2}]`  
`Out[ ]:= -2`
- We can also define a vector function, that is a function that returns a list of values. For example:  
`In[ ]:= func[{t_, x_}] = {x t, x + t, x - t}`  
`Out[ ]:= {t x, t + x, -t + x}`

A small video inset in the bottom right corner shows a man in a blue shirt speaking.

So, we will what we will do is, we will do an alternative implementation. We will do an alternative implementation of the RK4 method. To do that, so let us go back to the RK4 method and I want to show you something.

(Refer Slide Time: 26:41)

```
h = (tf - X0[[1]]) / nMax // N;  
For[datalist = {X0},  
  Length[datalist] < nMax,  
  AppendTo[datalist, next],  
  prev = Last[datalist];  
  rate1 = Through[F @ prev];  
  rate2 = Through[F @@ (prev +  $\frac{h}{2}$  rate1)];  
  rate3 = Through[F @@ (prev +  $\frac{h}{2}$  rate2)];  
  rate4 = Through[F @@ (prev + h rate3)];  
  next = prev +  $\frac{h}{6}$  (rate1 + 2 rate2 + 2 rate3 + rate4);  
];  
Return[datalist];  
]
```

Here, you see that I am using the Through function and F applied. I take previous and all these other things and take their arguments and apply F on that. This is an alternate way of writing this body which is somewhat simpler and also slightly more general because we are working with vectors here. Previous is the vector, weight is a vector.

So, in order to actually make things slightly simpler, what we will do is, we will do an alternative implementation.

(Refer Slide Time: 27:08)

```
Alternative Implementation  
• In Wolfram Language you can define a function of many arguments in more than one way:  
In[ ]:= func[t_, x_] = -x t  
func[{t_, x_}] = -x t  
Out[ ]:= -t x  
Out[ ]:= -t x  
In[ ]:= func[1, 2]  
Out[ ]:= -2  
In[ ]:= func[{1, 2}]  
Out[ ]:= -2  
• We can also define a vector function, that is a function that returns a list of values. For example:  
In[ ]:= func[{t_, x_}] = {x t, x + t, x - t}  
Out[ ]:= {t x, t + x, -t + x}  
In[ ]:= func[{1, 2}]
```

```

func[t_, x_] = -x t
func[{t_, x_}] = x + t
Out[448]=
-t x
Out[449]=
t + x
In[446]=
func[3, 2]
Out[446]=
-6
In[451]=
func[{1, 2, 3}]
Out[451]=
func[{1, 2, 3}]

```

• We can also define a vector function, that is a function that returns a list of values. For example:

```

In[ ]:= func[{t_, x_}] = {x t, x + t, x - t}
Out[ ]:= {t x, t + x, -t + x}

```

• We can also define a vector function, that is a function that returns a list of values. For example:

```

In[473]=
func[{t_, x_}] = {x t, x + t, x - t}
Out[473]=
{t x, t + x, -t + x}
In[475]=
func[{2, 2}]
Out[475]=
{4, 4, 0}

```

• We can avoid the use of **Through** function and make a vector definition of  $F$  directly with its argument also being a vector. This is slightly more general in notation and makes function calling a little easier.

• We will define rate function  $F$  as follows

$$F[\{t_, x_, y_]\} := \{f(t, x, y), g(t, x, y)\}$$

• where  $f$  and  $g$  are some function of the arguments. This way we can define the rate function  $F$  in one go. Code also appears to be slightly simpler. Here is the implementation

```

Clear["Global`*"]
In[ ]:= rk4[F_, X0_, tf_, nMax_] :=
Module[{h, datalist, prev, rate1, rate2, rate3, rate4, next},

```

To do this alternative implementation, we will see the multiple ways of defining a function. That is a same result but they are multiple ways we can define it in wolfram language and will make use of that. So, here is an example of a definition of function of t and x.

Let us say, function of t and x is  $-x t$ . We can make the same alternate definition in the following way. You can define function rather has to 2 arguments, t and x. we can have 1 argument to the function, that is the list. But this list can have 2 elements, t and x. And both the functions are defined in the same way. And let me go ahead and execute this. And when you call a function of 1, 2 with 2 arguments, 1 and 2, I will use the first definition and give me evaluation of 2.

We can go and change it to 3 and function of 3, 2 will give me -6,  $-3 * 2$ . But if I pass this

function, the arguments 1 and 2 inside a list it is going to give me this definition. And I can go ahead and execute this. Now, if you want to test it out that these two actually are different. We can make this  $x + t$ .

So, when I call the function  $f$  with 2 arguments, with single argument that is a list, but the list containing 2 elements, I will get  $x + t$  which is  $1 + 2$  that is 3. At the same now if I add another third argument, this is going to give me just the same thing back because this is not been defined in the context.

We have defined function as a single argument that is a list, which contains 2 elements, and that is being defined as  $x + t$ . But we have never defined a function with 3 arguments. So, there are multiple ways of defining a function and since we are working with vectors, what we will do is, we will use this vector definition, where we will pass a single vector to the function, and define a function in terms of its arguments or the elements of the list.

So, to do that what we can also do is, we can define a vector function, that is a function is the arguments of the functions is a vector and also its output is a vector. So, we can for example, this is the definition where the arguments of function is a list  $t, x$ , it is got 2 elements and then the output is a list of 3 elements  $x \cdot t, x + t$  and  $x - t$ .

So, let us go and try this out and when we do this we get  $x \cdot t, 1 \cdot 2 = 2, 1 + 2 = 3$  and  $2 - 1 = 1$ . We will go and change that to 2 and the result changes to 4, 4 and 0. Where now this is a list of 3 elements, this is a vector. So, my input to a function is also a vector and my output of the function is also a vector.

So vector goes in vector comes out so, therefore this is a definition of a vector function and this is what we are going to use in order to define the problem that we have working on. And this when we use this, we can avoid the Through function by putting in a vector and getting out a vector.

So, what we will do is, we will define our derivative function in the following fashion. So, derivative function will have an argument. If it is 2 dimensional problem will have arguments a list of  $t, x$  and  $y$  and the output will be 1 because time derivative is constant, it is 1. Derivative of time with respect to itself is 1 then  $\dot{x}$  is some function  $f$  of  $t, x, y, z$  and  $\dot{y}$  is some function of  $g$  of  $t, x, y, z$ .

(Refer Slide Time: 31:01)

```
F[{x_, y_}] := (1., f[{x, y}], g[{x, y}])

• where f and g are some function of the arguments. This way we can define the rate function F in one go. Code also appears to be slightly simpler. Here is the implementation

In[478]:= Clear["Global`*"]

In[ ]:= rk4[F_, X0_, tf_, nMax_] :=
Module[{h, datalist, prev, rate1, rate2, rate3, rate4, next},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] < nMax,
    AppendTo[datalist, next],
    prev = Last[datalist];
    rate1 = F@prev;
    rate2 = F@ (prev + h/2 rate1);
    rate3 = F@ (prev + h/2 rate2);
```

```
Module[{h, datalist, prev, rate1, rate2, rate3, rate4, next},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] < nMax,
    AppendTo[datalist, next],
    prev = Last[datalist];
    rate1 = F@prev;
    rate2 = F@ (prev + h/2 rate1);
    rate3 = F@ (prev + h/2 rate2);
    rate4 = F@ (prev + h rate3);
    next = prev + h/6 (rate1 + 2 rate2 + 2 rate3 + rate4);
  ];
  Return[datalist];
}
```

So, let us go and add clear the global context again just because we are going to define the RK4 method again. So, once I clear the global context previously defined RK4 is gone and what we will do is we will redefine the RK4 and again this time, the changes in the body. So, let us go ahead and look at the body.

This is the body. Now, we see, I do not need to use the Through function. I can take the f and directly apply it as a single argument previous the list goes as a single argument F and the output of this is going to be a list. Before I needed to use a Through function to get an argument as a list.

Get the output as a list. But I do not need to do that now because my F is being defined. I am

assuming that my F is defined in this fashion which is something like this. A list goes in and a list comes out. So, therefore I can simply take F and apply it on the single argument previous again I can take F and apply on this.

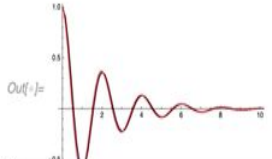
And at different point and then again rate3 in the same way fashion and rate4 in the same fashion. And I finally calculate the weighted average of rates multiplied by h and add it to previous and that gives me the new definition of RK4. So, let me go ahead and execute it. And this is new definition of RK4 that I am going to use. This is how we will apply it.

(Refer Slide Time: 32:28)

• This is how we will apply it now.

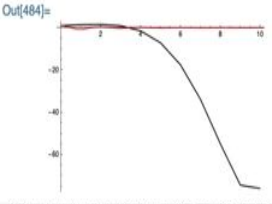
```
In[478]:=
w = 10;
beta =  $\sqrt{w - \frac{1}{4}}$ ;
rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};
initial = {0, 1, 0};
solx[t_] = e-t/2 Cos[beta t];

In[479]:= data4 = rk4[rateFunc, initial, 10, 70];
Show[ListPlot[data4[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full, PlotStyle -> Red]]
```

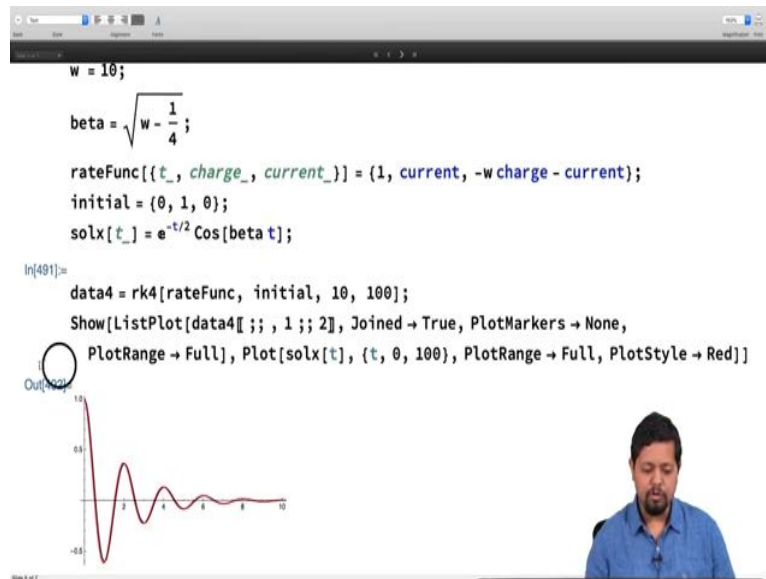


```
beta =  $\sqrt{w - \frac{1}{4}}$ ;
rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};
initial = {0, 1, 0};
solx[t_] = e-t/2 Cos[beta t];

In[485]:= data4 = rk4[rateFunc, initial, 10, 20];
Show[ListPlot[data4[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full, PlotStyle -> Red]]
```



• Now we will implement the Euler Method and RK2 (improved Euler) also in the same way. Then we will compare them for a couple of problems.



So, let us go ahead and we will go back to the problem we were working on the damped oscillator. For the damped oscillator, let us take  $w = 10$ ,  $\beta = \sqrt{w - \frac{1}{4}}$ . The difference here is, this time will define the rate function which is argument of  $t$ ,  $x$  and  $v$  as or this is actually the current so, the notation we are following was this was charge and this was current.

So rate function is the function of time, charge and current and  $t$  is 1,  $\dot{charge}$  is current and  $\dot{current}$  is minus  $W$  times charge and minus current is what we were doing. So, now we define 3 different rate functions for 3 different quantities. We define a single rate function for time, charge and current as a 3 tuple.

The 1<sup>st</sup> element is 1, the 2<sup>nd</sup> is current and  $\dot{current}$  is minus  $w$  times charge minus current. The initial condition is again a vector at  $t = 0$ ,  $charge = 0$  and  $current = 0$ . And the solution of this equation is  $e^{-2\cos(\beta t)}$  so let me go ahead and evaluate that.

We will call the variable `data4` and in that we will call the RK4, RK4 is now called with rate function. Rate function is the set of functions which defines the rate for charge, current and time. The initial condition which contains the initial time, the initial value of charge and current, final time `tf` and number of points `nMax` so let us start with a small number of points. Let us say, 10.

Let us go ahead and execute it. And you see this does not work. This is very poor. Because  $h$  is 1,  $t_i - t_f = 10$ .  $10/10 = 1$ . We are taking 2 few points. Let us increase the points to 20 and see what is that doing and you see that has already done a significant improvement with just



20 points. Rk4 method is already giving me almost shape of the true solution.

Let me go and increase this to 40. And you see that this is pretty much the shape of the solution. We go ahead and make it something like 70 and you see that this in pretty good agreement at 100. This will actually be fantastic. So, let us go ahead and compare this with other methods so the RK4 you see with only 100 points I am going to get a very reasonably good result. So, let me go ahead and check this out for Euler Methods.

(Refer Slide Time: 35:43)

• Now we will implement the Euler Method and RK2 (improved Euler) also in the same way. Then we will compare them for a couple of problems.

```
in[ ]:= euler[F_, X0_, tf_, nMax_] := Module[{h, datalist, prev},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] < nMax,
    AppendTo[datalist, prev + h (F@prev)],
    prev = Last[datalist];
  ];
  Return[datalist];
]
```

```
in[ ]:= rk2[F_, X0_, tf_, nMax_] := Module[{h, datalist, prev, rate1, rate2, next},
  h = (tf - X0[[1]]) / nMax // N;
  For[datalist = {X0},
    Length[datalist] < nMax,
    AppendTo[datalist, next],
    prev = Last[datalist];
    rate1 = F@prev;
    rate2 = F@ (prev + h rate1);
    next = prev +  $\frac{h}{2}$  (rate1 + rate2);
  ];
  Return[datalist];
]
```

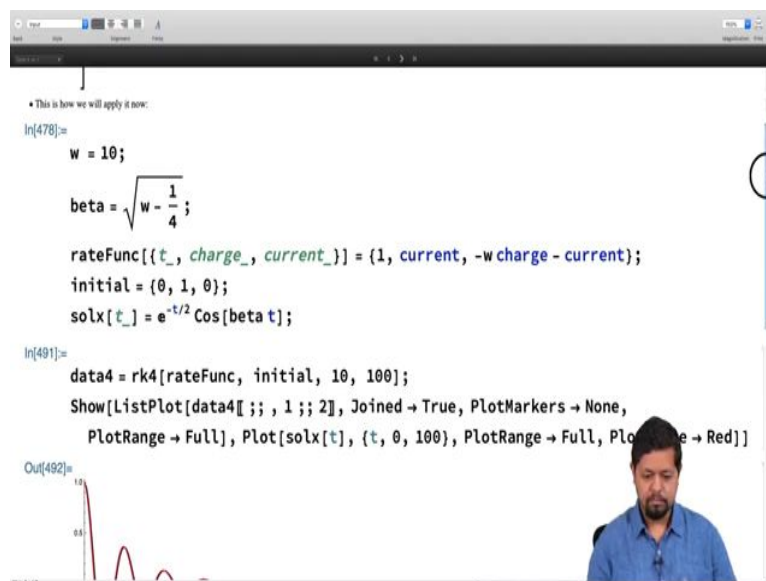
So, again we will define the Euler Method in the same way as you have defined the RK4 method that is, we will get rid of the Through function and we will have a rate function which whose output will be a vector.

It will give me rate of all the quantities. So, again the body of the Euler Method is being changed in the following way. We are redefining the Euler Method so that we can use the same structure for f that is the rate function and again all that we have done here is, is change the calculation of the next point is previous plus h times f at previous. This is the Euler Method.

So, let me go and execute it. And improved Euler or the second order Range-Kutta method which I am going to call now as RK2, this RK2 method I am going to redefine and this time the body changes in the following fashion. I have got rid of the Through function and f at at operation is just f at previous plus h times rate1. This is the 2 rates.

Take the average of the rates, multiply with h and add to previous and that is my improved Euler method. So, let me also execute this and load it into the context. So, now that I have defined the Euler Method, RK2 method and the improved Euler method, all 4 of them does the same style. Let me go ahead and do a comparison of all these methods.

(Refer Slide Time: 37:19)



The screenshot shows a Mathematica notebook interface. At the top, there is a title bar with standard window controls. Below the title bar, a small text box says "• This is how we will apply it now:". The main content area contains the following code:

```
In[478]:=
w = 10;
beta =  $\sqrt{w - \frac{1}{4}}$ ;
rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};
initial = {0, 1, 0};
solx[t_] = e-t/2 Cos[beta t];

In[491]:=
data4 = rk4[rateFunc, initial, 10, 100];
Show[ListPlot[data4[;;, 1 ;; 2], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full, PlotStyle -> Red]]
```

Below the code, the output of the last cell is shown as "Out[492]=", followed by a plot. The plot displays two data series: a red line representing the exact solution  $\text{solx}[t]$  and a blue line representing the numerical solution  $\text{data4}$ . Both series show a decaying oscillatory function starting at (0, 1). The plot axes range from 0 to 100 on the x-axis and 0 to 1.0 on the y-axis. A small inset plot in the bottom left corner shows a zoomed-in view of the initial part of the plot, highlighting the oscillations and the decay of the function.

```

rate2 = F0 (prev + h rate1);
next = prev +  $\frac{h}{2}$  (rate1 + rate2);
];
Return[datalist];
]

In[497]:= w = 10;
beta =  $\sqrt{w - \frac{1}{4}}$ ;

rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};
initial = {0, 1, 0};
solx[t_] = e-t/2 Cos[beta t];

In[498]:= data4 = rk4[rateFunc, initial, 10, 100];
Show[ListPlot[data4[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full]]

```

```

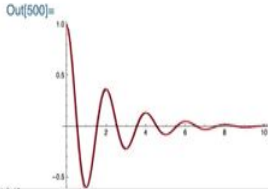
In[499]:= w = 10;
beta =  $\sqrt{w - \frac{1}{4}}$ ;

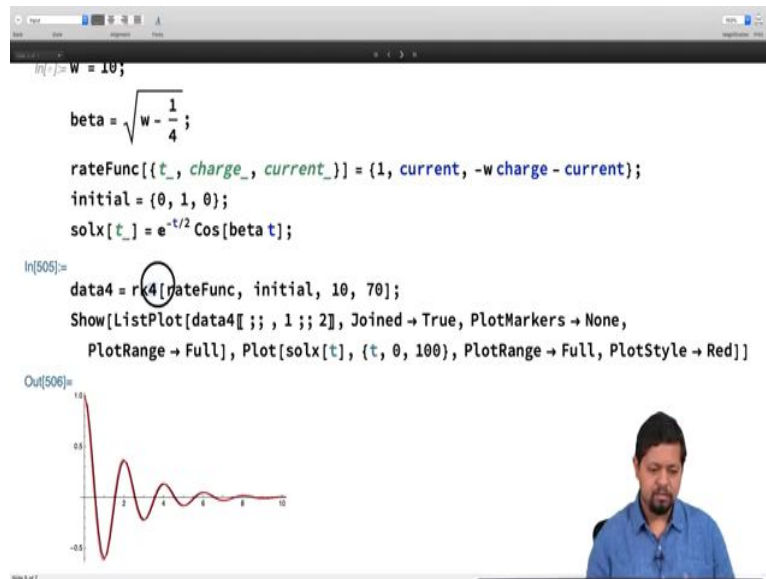
rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};
initial = {0, 1, 0};
solx[t_] = e-t/2 Cos[beta t];

In[499]:= data4 = rk4[rateFunc, initial, 10, 100];
Show[ListPlot[data4[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full, PlotStyle -> Red]]

Out[500]=

```





So, let me go ahead and do the comparison of all these methods. So, the same problem so, I will copy this part of the problem and I will put this at the end. Let me add this with the end over here, so, here is my definition of the problem, again I am choosing the same set of parameters  $w = 10$ , same rate function, same initial condition, the solution is also same.

Now rather than doing RK4, I can go ahead and try RK2 method and Euler Method. So, let me just go ahead with Euler Method first. With Euler method, 100 points let us see what happens? You see this is what happens with Euler Method on 100 points. We just tried this some time ago.


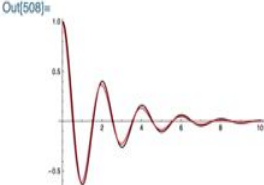
Euler Method with 100 points works in very poorly. Improved Euler method or RK2 method that we have defined with 100 points works reasonably well and RK4 method also works reasonably well and in fact RK4 method also works with lesser number of points may be at 50. So, here is RK4 at 50. Let us go and do this comparison again.

So, RK4 at 50, 70. So, let us go and do this. So, RK4 at 70 is pretty much as good as Euler Improved Euler Method or RK2.

(Refer Slide Time: 39:01)


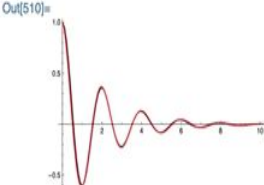
```
In[508]:= w = 10;  
beta =  $\sqrt{w - \frac{1}{4}}$ ;  
rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};  
initial = {0, 1, 0};  
solx[t_] =  $e^{-t/2} \text{Cos}[\text{beta } t]$ ;  
data4 = rk2[rateFunc, initial, 10, 100];  
Show[ListPlot[data4[;;, 1 ;; 2], Joined -> True, PlotMarkers -> None,  
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full, PlotStyle -> Red]]
```

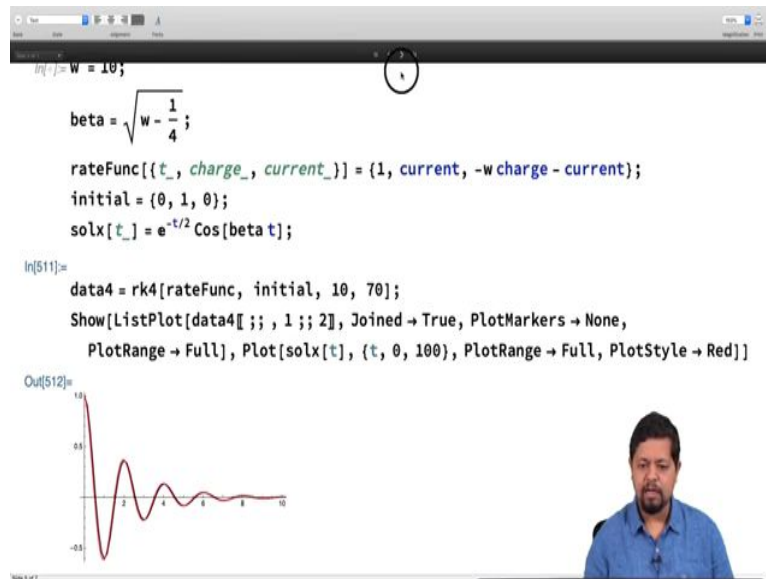
Out[508]=



```
In[509]:= w = 10;  
beta =  $\sqrt{w - \frac{1}{4}}$ ;  
rateFunc[{t_, charge_, current_}] = {1, current, -w charge - current};  
initial = {0, 1, 0};  
solx[t_] =  $e^{-t/2} \text{Cos}[\text{beta } t]$ ;  
In[509]= data4 = rk2[rateFunc, initial, 10, 100];  
Show[ListPlot[data4[;;, 1 ;; 2], Joined -> True, PlotMarkers -> None,  
PlotRange -> Full], Plot[solx[t], {t, 0, 100}, PlotRange -> Full, PlotStyle -> Red]]
```

Out[510]=





So, let us go ahead and see this at for the Euler method again in order to get a good solution in Euler Method, I require at least a 1000 points. Even with a 1000 points I am not getting a perfect agreement but with RK2 method, I can get a reasonably good agreement at just at 100 points.

And with RK4 method, we can do at even lesser number of points. Let us say 70, you get a pretty good agreement at 70. Let us do understand this comparison even better between these 3 methods by taking another example.

(Refer Slide Time: 39:41)

**Comparison of various algorithms using Driven Oscillator**

Equation of Motion for driven oscillator:  $m \frac{d^2 x}{dt^2} = -kx + F \cos(\omega t)$

EOM after non-dimensionalization:  $\frac{d^2 x}{dt^2} = -x + \cos(\omega t)$  (10)

Equations after reducing EOM to 1st ODEs:  $\begin{cases} \dot{x} = v \\ \dot{v} = -x + \cos(\omega t) \end{cases}$

Solution of EOM:  $x(t) = \frac{-\omega^2}{1-\omega^2} \cos(\omega t) + \frac{1}{1-\omega^2} \cos(\omega t)$

```

In[ ]:= ω = 0.8;
rateFunc[{t_, x_, v_}] = {1, v, -x + Cos[ω t]};
initial = {0, 1, 0};
solx[t_] :=  $\frac{-\omega^2 \text{Cos}[t] + \text{Cos}[\omega t]}{1 - \omega^2}$ ;

tf = 100;
nMax = 500;
data1 = euler[rateFunc, initial, tf, nMax];
data2 = rk2[rateFunc, initial, tf, nMax];
    
```

Equation of Motion for driven oscillator:  $m \frac{d^2 x}{dt^2} = -kx + F \cos(\omega t)$

EOM after non-dimensionalization:  $\frac{d^2 x}{dt^2} = -x + \cos(\omega t)$  (10)

Equations after reducing EOM to 1st ODEs:  $\begin{cases} \dot{x} = v \\ \dot{v} = -x + \cos(\omega t) \end{cases}$

Solution of EOM:  $x(t) = \frac{-\omega^2}{1-\omega^2} \cos(\omega t) + \frac{1}{1-\omega^2} \cos(\omega t)$

```

In[516]:=
D[ $\frac{-\omega^2}{1-\omega^2} \text{Cos}[t] + \frac{1}{1-\omega^2} \text{Cos}[\omega t]$ , {t, 2}] +  $\frac{-\omega^2}{1-\omega^2} \text{Cos}[t] + \frac{1}{1-\omega^2} \text{Cos}[\omega t]$ 
Out[516]:=
 $\frac{\text{Cos}[t \omega]}{1 - \omega^2} - \frac{\omega^2 \text{Cos}[t \omega]}{1 - \omega^2}$ 
In[ ]:= ω = 0.8;
rateFunc[{t_, x_, v_}] = {1, v, -x + Cos[ω t]};
initial = {0, 1, 0};
solx[t_] :=  $\frac{-\omega^2 \text{Cos}[t] + \text{Cos}[\omega t]}{1 - \omega^2}$ ;
    
```

Now we will compare the various methods, Euler Method, Euler, improved Euler or RK2 and RK4 method for the Rang-Kutta method. These various algorithms using driven oscillator, driven oscillator is something that we have covered in a different video.

So, go ahead and watch that, but here is a quick recap of the driven oscillator. The equation of motion of driven oscillator is given by this equation over here. Which is the first part of this equation is simply the equation of the simple harmonic oscillator to which we have added an external force.

So, this is an oscillator which is a simple harmonic oscillator but an external force is applied to drive it and when we non-dimensionalize it, this equation reduces down to this equation

where all the parameters, external parameters  $m$ ,  $k$  and  $f$  are gone and we are only left with single parameter  $\omega$ . Here,  $x$ ,  $t$  and  $\omega$  are dimensionless quantities.

When we reduce down this equation of motion, 2<sup>nd</sup> order equation of motion to 1<sup>st</sup> order coupled ODEs that is  $x' = v$  and  $v' = -x + \cos(\omega t)$ . We can use this to implement on the computer and or the, for the various methods that we have done. So, this is our set of first order ODEs. The solution of this equation of motion is given by this solution over here, which we will what we will do is quickly check.

So, here is our equation and we are claiming that this is the solution. So, in order to check that, what we will do is, we will use mathematica built-in method for calculating the derivatives. So, here is my claimed solution. So, this claimed solution is I am going to write as  $\frac{1}{\omega^2 - 1} \cos(\omega t)$ .

Take this guy multiply this with  $\cos t$  to this add  $\frac{1}{(1-\omega^2)}$  and then multiply that with  $\cos(\omega t)$  that is our claimed solution. This is what we are claiming that as a solution and in order to verify that this is actually the solution of this differential equation, we are going to calculate the 2<sup>nd</sup> derivative of this solution with respect to time.

In order to calculate the 2<sup>nd</sup> derivative, we will use the Mathematica's built-in function called `D`, which calculates the derivatives to calculate the single derivative with respect to time. I will just tell that this is the function, 1<sup>st</sup> argument of `D` is function, the 2<sup>nd</sup> argument is derivative with respect to the variable, so variable is  $t$ , so 1<sup>st</sup> derivative with respect to  $t$  is given by `D[f, t]`.

And that gives me this, but I am interested in the 2<sup>nd</sup> derivative for that, I will tell the `D` function that  $t$  is the argument, but I want the second derivative by putting all both these things in the list. If 1<sup>st</sup> argument is of this list is, 1<sup>st</sup> element of this list is  $t$ , second element is the order of derivative, when I do that I get the 2<sup>nd</sup> derivative.

Now, if to the 2<sup>nd</sup> derivative I add  $x$ , that is move this  $-x$  to the left hand side, it becomes  $\frac{d^2 x}{dt^2} + x = \cos(\omega t)$ . So, let me go ahead and add this solution to this derivative, so I take this, add this solution to this derivative and when I execute this, I get the sum and if you look at this sum, this sum can simplify further.



(Refer Slide Time: 43:32)

The image shows two screenshots of a Mathematica notebook. The top screenshot shows the following code and output:

```

Solution of EOM:  $x(t) = \frac{-\omega^2 \cos(t)}{1-\omega^2} + \frac{1}{1-\omega^2} \cos(\omega t)$ 

In[517]:=
D[ $\frac{-\omega^2}{1-\omega^2} \cos[t] + \frac{1}{1-\omega^2} \cos[\omega t]$ , {t, 2}] +  $\frac{-\omega^2}{1-\omega^2} \cos[t] + \frac{1}{1-\omega^2} \cos[\omega t]$  //
Simplify

Out[517]=
Cos[t  $\omega$ ]

In[ ]:=  $\omega = 0.8$ ;
rateFunc[{t_, x_, v_}] = {1, v, -x + Cos[ $\omega$  t]};
initial = {0, 1, 0};
solx[t_] :=  $\frac{-\omega^2 \cos[t] + \cos[\omega t]}{1-\omega^2}$ ;

tf = 100;
nMax = 500;
data1 = euler[rateFunc, initial, tf, nMax];
data2 = rk2[rateFunc, initial, tf, nMax];

```

The bottom screenshot shows the following code and output:

```

rateFunc[{t_, x_, v_}] = {1, v, -x + Cos[ $\omega$  t]};
initial = {0, 1, 0};
solx[t_] :=  $\frac{-\omega^2 \cos[t] + \cos[\omega t]}{1-\omega^2}$ ;

In[522]:=
tf = 100;
nMax = 500;
data1 = euler[rateFunc, initial, tf, nMax];
data2 = rk2[rateFunc, initial, tf, nMax];
data4 = rk4[rateFunc, initial, tf, nMax];

Table[Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full, Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],
ImageSize -> 400], {data, {data1, data2, data4}}]

```

Below the code in the bottom screenshot, a plot is visible showing the solution  $x(t)$  over time  $t$ . The plot shows a highly oscillatory signal with a sharp peak, indicating resonance. The y-axis ranges from 0 to 20,000, and the x-axis ranges from 0 to 100.

Let me once again go ahead and apply postfix simplify on this mathematica in-built ability to simplify equations. So, let me go ahead and simplify that and all we are left with  $\cos(\omega t)$ . So  $d^2 \frac{x}{dt^2} + x = \cos(\omega t)$  square  $x$  by  $dt$  square plus  $x$  is  $\cos$  of  $t$   $\omega$ . Therefore, the claimed solution that we have given over here is actually the solution of this equation. This was just a quick cross check of the solution.

And now, let us go ahead and add implement this. What we will do is, when  $\omega$  is close to 1, you see there is a divergence and that is called as resonance, the amplitude becomes very large as  $\omega$  approaches 1.

So, what we will do is, we will work with some value of  $\omega$  close to 1, so in this example,

what we have done is, we have we have taken  $\omega = 0.8$  and then we define the rate function, so we will take a free parameter  $\omega = 0.8$  and take the rate function as the argument of the rate function is the vector  $t, x, v$  and the value of this function is also the vector  $1, v, -x + \cos(\omega t)$ .

Where 1 is the rate of time and  $x' = v$  so that is why we have got the 2<sup>nd</sup> argument is  $v$  and  $v' = -x + \cos(\omega t)$ , that is why we have got the 3<sup>rd</sup> element over here is  $-x + \cos(\omega t)$ . So, that is our rate function for the forced oscillator or the driven oscillator and then the initial condition here is, 0 1 0 that is, at  $t = 0$ ,  $x = 1$  and  $v = 0$ .

So, that is we take the forced oscillator, we start out from extreme position or extended we extend the oscillator away from the mean position and we release it under the force  $f\cos(\omega t)$ . The solution is as we discussed is been given over here. So, this is our initialization of the problem. So let us go and execute that.

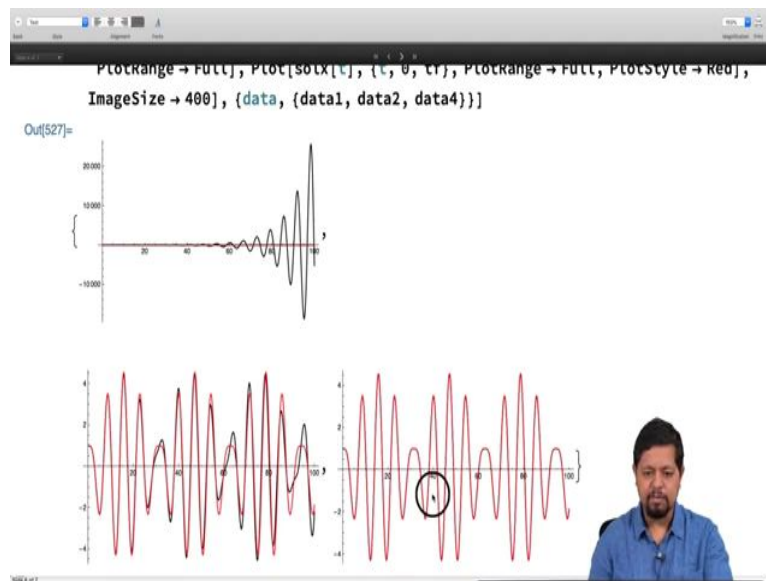
Then we want to  $tf = 100$ ,  $nMax = 500$  and for these 2 values, for these value of  $tf$  and  $nMax$ , I want to find the solution using the Euler method, RK2 method, and RK4 method and again you see we have defined the functions in the same way. So, we are using the same argument, same structure.

So Euler functions is called with rate function, the initial condition  $tf$  and  $n$  max, RK2 is called with again the same arguments, RK4 is called with same arguments, so we are keeping the  $tf$  and  $nMax$  fixed. So, let us go ahead and execute this. So,  $tf = 100$  and  $nMax = 500$ . Now, what we are going to do is, we are going to make a list plot of all of all 3 of these.

In order to do that, I am going to use the table construct. So, this part of the table construct makes the list plot for and compares it with the solution and to for this variable data, I am passing 3 possible choices.

So, table runs on the variable data where data takes the value data1, data2 and data4. And data1, data2 and data4 are calculated over here using the Euler method, RK2 and RK4 method. And when I execute this I am going to produce 3 list plot comparisons or 3 plots where I am comparison comparing the data from each of these methods with the analytical solution. Let me go ahead and execute that.

(Refer Slide Time: 47:06)



And the output is set of 3 graphs. This is because the Euler method, Euler Method compares very-very badly, you see Euler Method gives shows that the oscillations keeps going indefinitely. This would have happened if  $\omega$  was close to 1, but the  $\omega$  is actually 0.8.

So, this is a bad result from the Euler Method. The RK2 method gives a reasonable equal result but it does not show agreement and in all the regions. But RK4 method gives a fantastic agreement for the same value of nMax.

Now, all this different 3 different cases were done for same value of nMax. And we see that, for same value of nMax, RK4 gives the much better result compared to the to the RK2 method and the Euler method. Let us go ahead and check this out for slightly different values.

(Refer Slide Time: 47:59)

```
solx[t_] :=  $\frac{1 - \cos(\omega t)}{1 - \omega^2}$ ;
```

```
tf = 50;
```

```
nMax = 100;
```

```
data1 = euler[rateFunc, initial, tf, nMax];
```

```
data2 = rk2[rateFunc, initial, tf, nMax];
```

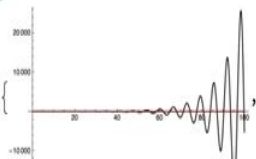
```
data4 = rk4[rateFunc, initial, tf, nMax];
```

```
In[527]= Table[Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
```

```
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],
```

```
ImageSize -> 400], {data, {data1, data2, data4}}]
```

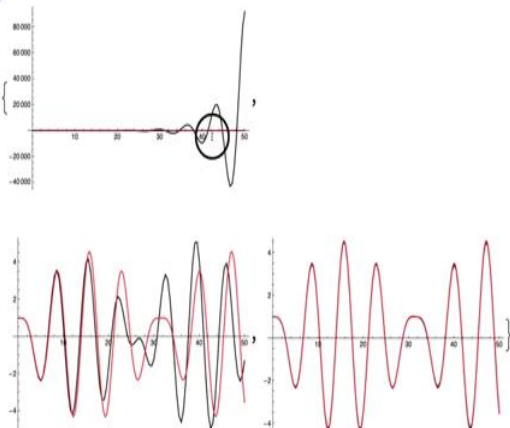
Out[527]=



```
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],
```

```
ImageSize -> 400], {data, {data1, data2, data4}}]
```

Out[533]=



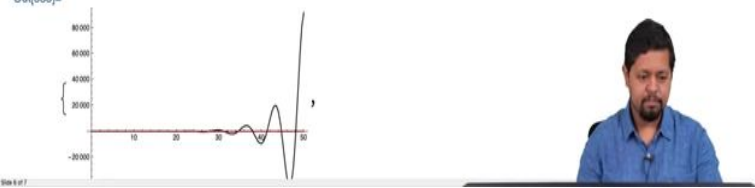
Let us go ahead and make  $tf$  equal to let us say, say 50 and  $nMax \leq 100$ . Let us go and do this comparison. You see again, for  $tf = 50$ ,  $nMax = 100$ , which is you can calculate  $h$ ,  $h = 50/100 = 0.05$  for  $h = 0.05$ , Euler Method is giving out poor result. Improved Euler Method is showing some increment in the beginning but then it is deviating from the true solution. However, RK4 method is doing the reasonably well.

(Refer Slide Time: 48:43)

```
In[534]=
tf = 50;
nMax = 200;
data1 = euler[rateFunc, initial, tf, nMax];
data2 = rk2[rateFunc, initial, tf, nMax];
data4 = rk4[rateFunc, initial, tf, nMax];

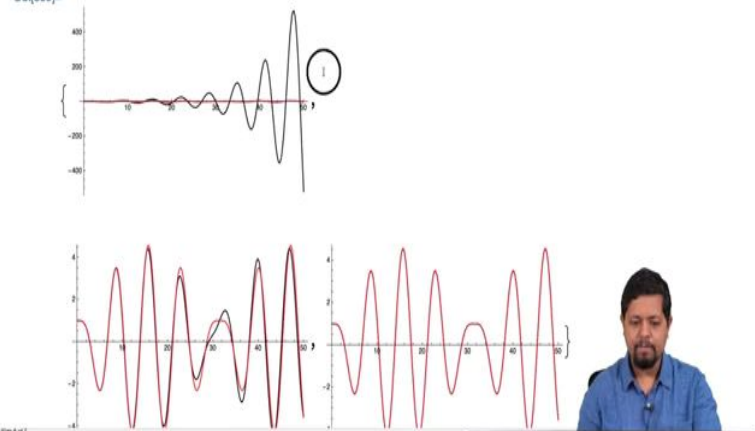
In[533]=
Table[Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],
ImageSize -> 400], {data, {data1, data2, data4}}]

Out[533]=
```



```
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],
ImageSize -> 400], {data, {data1, data2, data4}}]

Out[539]=
```



And as we increase this nMax from 100 to 200, there will be improvement for both the RK2 and the RK4 methods. RK4 method is now in a perfect agreement, RK2 method has improved but the Euler method is still not performing well. If you want Euler Method to perform well, we need to increase the nMax further. So, let us go ahead and check that out.

(Refer Slide Time: 49:10)

The image displays two screenshots from a Mathematica notebook. The top screenshot shows the following code and output:

```
tf = 50;
nMax = 1000;
data1 = euler[rateFunc, initial, tf, nMax];
data2 = rk2[rateFunc, initial, tf, nMax];
data4 = rk4[rateFunc, initial, tf, nMax];
```

In[539]=

```
Table[Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],
ImageSize -> 400], {data, {data1, data2, data4}}]
```

Out[539]=

The plot shows a red curve representing the solution  $\theta(t)$  over time  $t$  from 0 to 50. The y-axis ranges from -400 to 400. The plot shows a highly oscillatory function that grows in amplitude over time. A small circle highlights the initial condition at  $t=0$ .

The bottom screenshot shows the same code and output, but with a different plot configuration:

```
ImageSize -> 400], {data, {data1, data2, data4}}]
```

Out[545]=

The plot shows the same data as the top screenshot, but with a different y-axis range from -15 to 15. This provides a more detailed view of the oscillations. A small circle highlights the initial condition at  $t=0$ .

So, what we will do is, we will make nMax 1000 and see how does that do for the Euler method in this problem. We see now that the Euler method is starting to give some improvement at least in the beginning, whether it again deviates away for larger values, however improved Euler and the RK2 and RK4 do pretty well over here.

(Refer Slide Time: 49:30)

The image shows two screenshots of a Mathematica notebook. The top screenshot displays the following code:

```
tf = 50;  
nMax = 10000;  
data1 = Euler[rateFunc, initial, tf, nMax];  
data2 = rk2[rateFunc, initial, tf, nMax];  
data4 = rk4[rateFunc, initial, tf, nMax];
```

The input cell is:

```
In[545]= Table[Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,  
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],  
ImageSize -> 400], {data, {data1, data2, data4}}]
```

The output cell shows a plot with three curves: a red curve representing the true solution, and two black curves representing the Euler method (data1) and the RK4 method (data4). The Euler method curve shows significant oscillatory deviation from the true solution as time increases, while the RK4 method curve remains very close to the true solution.

The bottom screenshot shows the same code but with the Euler method curve removed. The input cell is:

```
In[551]= Table[Show[ListPlot[data[[;;, 1 ;; 2]], Joined -> True, PlotMarkers -> None,  
PlotRange -> Full], Plot[solx[t], {t, 0, tf}, PlotRange -> Full, PlotStyle -> Red],  
ImageSize -> 400], {data, {data1, data2, data4}}]
```

The output cell shows two plots. The top plot compares the true solution (red) with the RK2 method (black). The bottom plot compares the true solution (red) with the RK4 method (black). The RK4 method curve is nearly indistinguishable from the true solution.

Ofcourse, let us go and make it 10000 and for 10000, we see that Euler method starts to give decent result but as time grows, the deviations also grow form the true solution. Let us go ahead and extend this comparison further by doing the error analysis.

(Refer Slide Time: 49:51)

The slide is titled "Error Analysis" and contains the following content:

- A bullet point: "We implemented the following `err` function, a few weeks back, to compute the mean global error, given by equation
- A yellow highlighted equation: 
$$err = \frac{1}{N} \sum_{i=1}^N |y_i - F(t_i)| \quad (1)$$
- A Mathematica code block starting with `In[552]=`:

```
err[dataset_, func_] := Module[{tlist, xlist, Fxlist},
  tlist = dataset[[ , 1]]; (*Extract each time value*)
  xlist = dataset[[ , 2]]; (*Extract each x value*)
  Fxlist = func /@ tlist;
  (*Apply func to each time value to get list of func[t;]*)
  Return[xlist - Fxlist // Abs // Mean];
]
```
- A section titled "Scaling with  $h$ " with a bullet point: "Lets define the problem"
- Code defining the problem:

```
 $\omega = 0.2;$ 
rateFunc[{t_, x_, v_}] = {1, v, -x + Cos[ $\omega$  t]};
initial = {0, 1, 0};
```
- A video inset in the bottom right corner showing a man in a blue shirt speaking.

In each of the 3 cases. So, what we are going to do is, we will take this error, the mean absolute error that we defined earlier through this error function. Will execute this error function load it into the context.

Now, we will look at how does this error scales with  $h$ ? How does the mean error scales with  $h$  in these 3 functions?



(Refer Slide Time: 50:16)

```

In[553]:=
ω = 0.2;
rateFunc[{t_, x_, v_}] = {1, v, -x + Cos[ω t]};
initial = {0, 1, 0};
solx[t_] :=  $\frac{-\omega^2 \text{Cos}[t] + \text{Cos}[\omega t]}{1 - \omega^2}$ ;
• Next, we calculate the errors for each of the algorithms and check its scaling with h
• Euler Method
tf = 20;
Table[dataset = euler[rateFunc, initial, tf, 10^n];
h =  $\frac{tf}{10.0^n}$ ;
 $\frac{1}{h}$  err[dataset[;;, 1 ;; 2], solx], {n, 1, 4}]
{9.66266, 0.296082, 0.146228, 0.1376}
• Improved Euler/Runge Kutta 2nd order
tf = 20;

```

```

solx[t_] :=  $\frac{-\omega^2 \text{Cos}[t] + \text{Cos}[\omega t]}{1 - \omega^2}$ ;
• Next, we calculate the errors for each of the algorithms and check its scaling with h
• Euler Method
In[557]:=
tf = 20;
Table[dataset = euler[rateFunc, initial, tf, 10^n];
h =  $\frac{tf}{10.0^n}$ ;
 $\frac{1}{h}$  err[dataset[;;, 1 ;; 2], solx], {n, 1, 4}]
Out[558]:=
{9.66266, 0.296082, 0.146228, 0.1376}
• Improved Euler/Runge Kutta 2nd order
tf = 20;
Table[dataset = rk2[rateFunc, initial, tf, 10^n]; h =  $\frac{tf}{10.0^n}$ ;
 $\frac{1}{h^2}$  err[dataset[;;, 1 ;; 2], solx], {n, 1, 4}]

```

So, this is our definition of the problem. So, for Euler Method, we will take  $tf = 20$ , we will calculate the table where the data set is obtained by using the Euler method, and  $nMax$  is changed, extend by  $n$ , where  $n$  is 1 to 4.

In so, what do we do, we calculate dataset, we calculate the  $h$  value and both these outputs are suppressed by and then we output  $1/h$  times the error after comparing it with the solution. So, when we execute this, you see that the error /  $h$  becomes a constant. That is what we have expected for the Euler function.

The global error becomes a constant with  $h$  as  $h$  becomes smaller or  $nMax$  becomes larger. It converges to some number close to 0.14. So,  $h$  becomes a constant error over  $h$  becomes a

constant that means the error scales like  $h$ , or the global errors scales like  $h$  in the Euler method.

(Refer Slide Time: 51:24)

```

1/h err[dataset[;;, 1; 2], solx], {n, 1, 4}
Out[558]=
{9.66266, 0.296082, 0.146228, 0.1376}
• Improved Euler/Range Kutta 2nd order
In[559]:=
tf = 20;
Table[dataset = rk2[rateFunc, initial, tf, 10^n]; h = tf/10.0^n;
1/h err[dataset[;;, 1; 2], solx], {n, 1, 4}
Out[560]=
{2.58767, 0.0438949, 0.0442324, 0.0442784}
• Range Kutta 4th order
tf = 20;
Table[dataset = rk4[rateFunc, initial, tf, 10^n];
h = tf/10.0^n;

```

```

1/h^4 err[dataset[;;, 1; 2], solx], {n, 1, 4}
Out[560]=
{2.58767, 0.0438949, 0.0442324, 0.0442784}
• Range Kutta 4th order
In[561]:=
tf = 20;
Table[dataset = rk4[rateFunc, initial, tf, 10^n];
h = tf/10.0^n;
1/h^4 err[dataset[;;, 1; 2], solx], {n, 1, 4}
Out[562]=
{0.000918248, 0.00216844, 0.00219111, 0.00362541}
Comparison for fixed h
• Comparison of methods with each other for a fixed value of h:
tf = 20;
nMax = 1000;

```

Let us do the same thing for improved Euler method which is done this code over here and I will execute this. And this is the result we will get in this case again what we have done here, we have taken the error and divide by  $h^2$ . So,  $error/h^2$  in this case becomes a constant close to 0.04.

This constant is going to change depending on the method and the problem being used, but it will always be some constant. So, RK2 method, global error goes to  $h^2$ . This is the validation of that. And for RK4 method, we will take the error and divide by  $h^4$ . When we

execute that, we see that this number converges to some number close to 0.002 or 0.003, so in RK4 method we find the error does scale like  $h^4$  which is what we expected.

Let us do the comparison for fixed h that is, in this cases we were changing nMax and therefore we are changing x and we are looking h and we are looking at how does errors scale with h.

(Refer Slide Time: 52:38)

```

In[563]:=
tf = 20;
nMax = 1000;
h = (tf - 0.0) / nMax;
data1 = euler[rateFunc, initial, tf, nMax];
data2 = rk2[rateFunc, initial, tf, nMax];
data4 = rk4[rateFunc, initial, tf, nMax];
{err[data1[;;, 1 ;; 2], solx], err[data2[;;, 1 ;; 2], solx],
err[data4[;;, 1 ;; 2], solx]}

Out[565]=
0.02

Out[569]=
{0.00292457, 0.000017693, 3.50577 × 10-10}

Timing Analysis
• Let's tune nMax or h so that the errors for each of the methods is approximately comparable.
tf = 20:

```

Now, we are going to do is, we are going to fix the value of h and see for what value of n max, we are going to get results in each of the cases. So, we will take nMax = 1000, tf = 20 and h = tf/nMax.

So, and then we calculate data1, data2 and data4 using the Euler RK2 and RK4 methods. When we do that, we see that the h value is 0.02 and the errors for this fixed value of h that is step size is fixed. The error in the Euler method is 0.0029, 0.003 or  $10^{-3}$  is of the order of  $10^{-3}$ . For RK2 method, this error becomes of the order of  $10^{-5}$  and RK4 methods drops down to the order of  $10^{-10}$ .

Again this is of the order of h. This is of the order of  $h^2$  and this is of the order of  $h^4$ , h was 0.02 in this case is was  $10^{-2}$  order. This is slightly less than  $10^{-2}$  order. So, this order h, this is order  $h^2$  and this is order  $h^4$ . We can also do the timing analysis. See which of these methods is fast, how fast each of these methods are.

(Refer Slide Time: 54:00)

Out[569]=  
{0.00292457, 0.000017693, 3.50577 × 10<sup>-10</sup>}

**Timing Analysis**

- Let's tune  $\epsilon_{\text{Max}}$  or  $h$  so that the errors for each of the methods is approximately comparable.

In[580]=  
tf = 20;  
data1 = euler[rateFunc, initial, tf, 80 000];  
data2 = rk2[rateFunc, initial, tf, 2000];  
data4 = rk4[rateFunc, initial, tf, 100];  
{err[data1[;;, 1 ;; 2], solx], err[data2[;;, 1 ;; 2], solx],  
err[data4[;;, 1 ;; 2], solx]}

Out[579]=  
{0.000068458, 4.42581 × 10<sup>-6</sup>, 3.46951 × 10<sup>-6</sup>}

- Let's compare Time taken by each algorithm for solving the problem

```
euler[rateFunc, initial, 20, 30 000]; // Timing  
{2.23345, Null}
```

Slide 7 of 7

Out[569]=  
{0.00292457, 0.000017693, 3.50577 × 10<sup>-10</sup>}

**Timing Analysis**

- Let's tune  $\epsilon_{\text{Max}}$  or  $h$  so that the errors for each of the methods is approximately comparable.

In[580]=  
tf = 20;  
data1 = euler[rateFunc, initial, tf, 80 000];  
data2 = rk2[rateFunc, initial, tf, 2000];  
data4 = rk4[rateFunc, initial, tf, 100];  
{err[data1[;;, 1 ;; 2], solx], err[data2[;;, 1 ;; 2], solx],  
err[data4[;;, 1 ;; 2], solx]}

Out[584]=  
{0.0000342006, 4.42581 × 10<sup>-6</sup>, 3.46951 × 10<sup>-6</sup>}

- Let's compare Time taken by each algorithm for solving the problem

```
euler[rateFunc, initial, 20, 30 000]; // Timing  
{2.23345, Null}
```

Slide 7 of 7

```

tf = 20;
data1 = euler[rateFunc, initial, tf, 80000];
data2 = rk2[rateFunc, initial, tf, 2000];
data4 = rk4[rateFunc, initial, tf, 100];
{err[data1[;;, 1 ;; 2], solx], err[data2[;;, 1 ;; 2], solx],
err[data4[;;, 1 ;; 2], solx]}

Out[584]=
{0.0000342006, 4.42581 × 10-6, 3.46951 × 10-6}

• Let's compare Time taken by each algorithm for solving the problem

In[585]=
euler[rateFunc, initial, 20, 80000]; // Timing

Out[585]=
{46.0407, Null}

rk2[rateFunc, initial, 20, 2000]; // Timing

{0.034432, Null}

rk4[rateFunc, initial, 20, 100]; // Timing

```

---

```

{err[data1[;;, 1 ;; 2], solx], err[data2[;;, 1 ;; 2], solx],
err[data4[;;, 1 ;; 2], solx]}

Out[584]=
{0.0000342006, 4.42581 × 10-6, 3.46951 × 10-6}

• Let's compare Time taken by each algorithm for solving the problem

In[585]=
euler[rateFunc, initial, 20, 80000]; // Timing

Out[585]=
{46.0407, Null}

In[586]=
rk2[rateFunc, initial, 20, 2000]; // Timing

Out[586]=
{0.031795, Null}

In[587]=
rk4[rateFunc, initial, 20, 100]; // Timing

Out[587]=
{0.003105, Null}

• RK4 is the gold standard for solving ODEs when you want to achieve both good accuracy and high efficiency.

```

So, for that we will use the timing function. Built-in timing function in Mathematica. So, first what we will do is, we will tune the value of nMax for each of these cases so that we get approximately the same error. So, this is what we have what I have done here is, I played around with these values of nMax so, that I get the same error in all the 3 cases.

So, this will require you to play around a little bit with this nMax. When you do that, you see for, when I take nMax = 30000 for the Euler function, I get an error of  $10^{-5}$  and for RK2 I get an error of  $10^{-6}$ . For 2000 points and for RK4 I get an error of 10 to minus 6 for 100 points. So, what I have done is, I have tuned all these 3 methods to get an error of same order.

Error in the Euler method is slightly higher. In order to improve this further I may have to go to slightly more points but that does not really guarantee an improvement. But we can try this

out. Let me try it for 40000 points and this is still running. There we go okay the some improvement but not the error still of the order of  $10^{-5}$ . We need to probably in order to actually get to the order of  $10^{-6}$ , we may have to increase it further.

If we are lucky that might happen. So, let me go ahead and make it 80000 and this is going to take some time to run. You can see that, it is still running. Let us wait for this and see if there are some improvement and this is actually running because the first step has 80000 points that is what it is taking time.

If have to run the for loop for 80000 points. And for the Euler Method it is going to be pretty fast or improved Euler or RK2 is going to be pretty fast because it is 2000 and RK4 is going to be extremely fast.

So, there we go. We are done with this. So, still you see this is not much of an improvement. We could not really go down to  $10^{-6}$  but this is 3 times  $3 \times 10^{-5}$ . We will stick with that, but you see is 80000 points. So, let us go ahead and compare the timings. So, here let me go ahead and compare timings for 80000 and still running. So we have gone wait for this to finish and approximately my belief is this is going to be something around 10 seconds. Looks like, it is going more than 10 seconds.

Alright, it looks like it is even more than 20 seconds. There we go. So, it is 46 seconds. It took about 46 seconds to take to do 80000 steps using Euler Method to get an accuracy of  $3 \times 10^{-5}$ . Let us do this for RK2 method, where I get an accuracy of  $4 \times 10^{-6}$  with 2000 points.

So, let us execute that and you see it only take about 0.03 seconds. And for RK4 with 100 points which gives me the same accuracy. This only takes about 0.003 seconds on this computer. So, RK4 method is about 10 times faster than RK2 to get the same accuracy. And RK2 is about a 1000 times faster than the Euler method.

So, we see that those significant improvement when it comes to times, when we are using the RK4 method and that is why we often just stick to the RK4 method when we are solving ordinary differential equations. In fact, RK4 method is considered as a gold standard in order for solving ODEs to achieve both the high accuracy and high efficiency.

High accuracy means that I can get accuracy of order  $h^4$ , global error of the order  $h^4$  and

high efficiency means I can do this in less number of steps, that is, my code is going to do less amount of computation and I am going to get high accuracy.

So, RK4 method is considered as a gold standard and this is the method we are going to use in studying all the other problems related to ordinary differential equations that we are going to come across in this course. You can go ahead and play around with some others examples of the differential equation and we will see you next time with more examples with implementation of RK4 method.