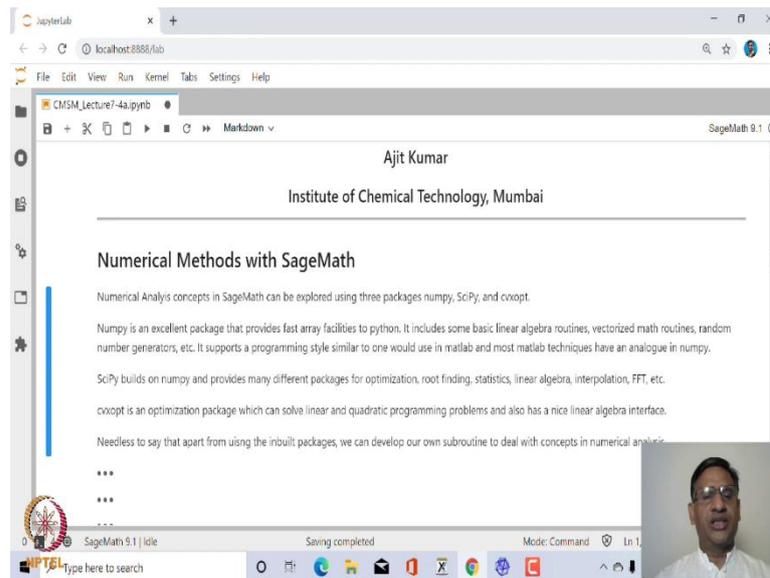


Lecture - 45
Finding Roots of algebraic and transcendental equations in SageMath

(Refer Slide Time: 00:15)



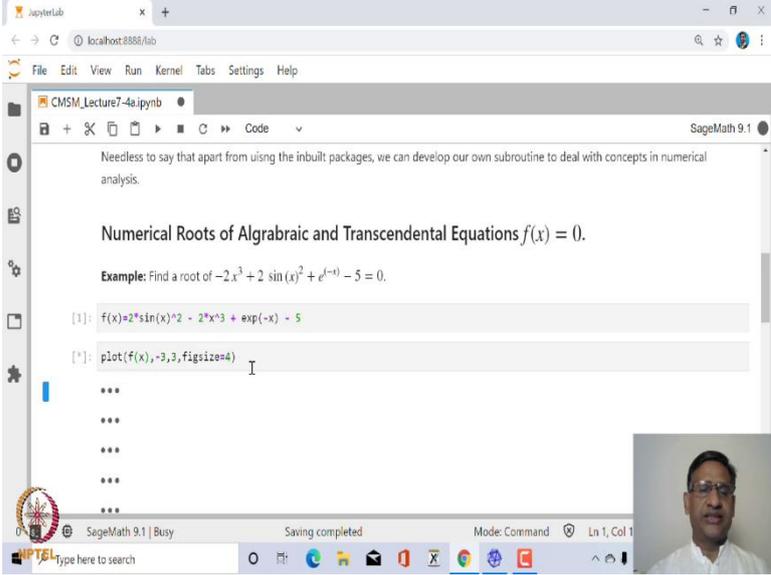
Welcome to the 45th lecture on Computational Mathematics with SageMath. In this lecture, we shall look at finding numerical roots of algebraic and transcendental equations. So, henceforth we shall look at some Numerical Methods in SageMath. So, let us get started. So, numerical analysis concepts in SageMath can be explored using, basically three packages; NumPy, SciPy, and cvxopt.

We have already seen NumPy and SciPy. NumPy is an excellent package in python that has facilities of fast array computation. It also includes linear algebra package, vectorized math routines, and random number generators. It has programming styles very similar to MATLAB.

On the other hand, SciPy builds on NumPy and provides many different packages for optimization, root finding, statistics, linear algebra, fast Fourier transforms etcetera. Cvxopt is an optimization package, which can solve linear and quadratic programming problems, and it also has nice linear algebra interface.

Of course, you can write your own programs in SageMath in order to solve problems in numerical analysis. And we will be doing for some of the methods.

(Refer Slide Time: 02:05)

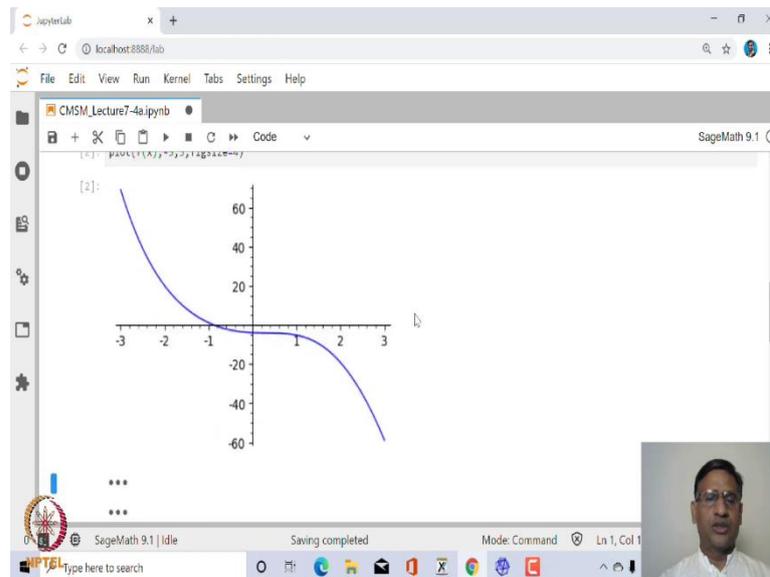


The screenshot shows a SageMath 9.1 JupyterLab interface. The browser address bar shows 'localhost:8888/lab'. The notebook title is 'CMSM_Lecture7-4a.ipynb'. The text in the cell reads: 'Needless to say that apart from using the inbuilt packages, we can develop our own subroutine to deal with concepts in numerical analysis. Numerical Roots of Algebraic and Transcendental Equations $f(x) = 0$. Example: Find a root of $-2x^3 + 2 \sin(x)^2 + e^{(-x)} - 5 = 0$.' Below the text, there are two code cells. The first cell contains the definition of the function: `f(x)=2*sin(x)^2 - 2*x^3 + exp(-x) - 5`. The second cell contains the plot command: `plot(f(x),-3,3,figsize=4)`. The interface also shows a 'Saving completed' message and a small video feed of a person in the bottom right corner.

So, we shall look at finding roots of algebraic and transcendental equations in one variable, namely $f(x)$ equal to 0. So, when I say algebraic equation, basically it means that $f(x)$ is a polynomial. Transcendental equations, on the other hand, will be combination of polynomials, exponential functions, algebraic, trigonometric functions, etcetera.

So, let us start with a problem. We want to find a root of minus $2x$ cube plus $2 \sin x$ square plus e to the power x minus 5 equal to 0. We want to find x for which this left-hand side is equal to 0. So, before we find a root, let us plot graph of this function. So, first, let us declare this function.

(Refer Slide Time: 03:15)



When we, after declaring this function, let us plot its graph between, let us say, minus 3 and 3, and figsize is equal to 4. So, this is what we see. So, we can see that between minus 3 and 3, it has a root, actually, the root lies between minus 1 and 0. So, we want to locate that root. We have already seen this in, in SageMath.

(Refer Slide Time: 03:34)

```
[3]: solve(f(x)==0,x)
[3]: [x^3 == 1/2*(2*e^x*sin(x)^2 - 5*e^x + 1)*e^(-x)]
[4]: f.find_root??

Source:
def find_root(self, a, b, var=None, xtol=10e-13, rtol=2.0**-50, maxiter=100, full_output=False):
    """
    Numerically find a root of self on the closed interval [a,b] (or
    [b,a] if possible, where self is a function in the one variable.
    Note: this function only works in fixed (machine) precision, it is not
    possible to get arbitrary precision approximations with it.

    INPUT:
    - ``a, b`` - endpoints of the interval
    - ``var`` - optional variable
    - ``xtol, rtol`` - the routine converges when a root
    is known to lie within xtol of the value return. Should be >= 0. The
    routine modifies this to take into account the relative precision
    of doubles.
```

So, one can try to use solve function. So, when, when I say solve $f(x)$ equal to 0 for x , it may or may not give a solution. So, for example, in this case, it is not finding a solution. It is unable to find x explicitly. So, this may not work. So, in that case, we have seen that

we can use what is called `f.find_root`. So, let us look at what is `find_root`, it is an inbuilt method in SageMath.

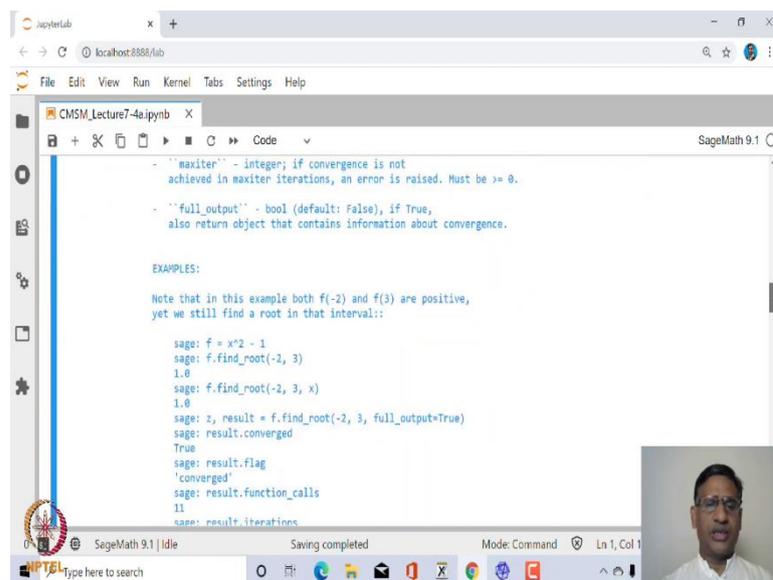
So, let us first take help on this. So, `f.find_root?`, which will give you detailed document. So, this says that `find_root` of self, that is the function, and `a` and `b` are the interval, the end point of the interval.

You need to mention the variable with respect to which you want to find a root, you have to also give the tolerance limit and the number of iterates, maximum number of iterates. By default, it takes 10 iterates.

And you can mention the output true or false, that is a Boolean option, and what is it? So, it finds numerically a root of a function `f` in `a`, in a closed interval `[a,b]` or `[b,a]`. So, it does not matter, you can give `[a,b]` where `a` could be less than `b`, or `a` could be bigger than `b`.

And if possible, where the self-function has one variable, is of one variable. So, basically, this works with only one variable. If you have multivariable function and they want to find roots, this does not work, right? And also, here it will find solution only when there is a root between `a` and `b`.

(Refer Slide Time: 05:46)



```

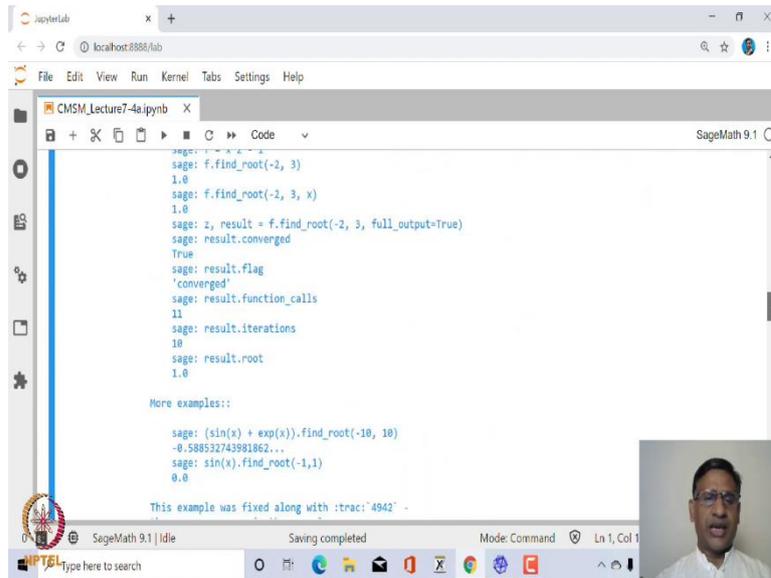
-- ``maxiter`` - integer; if convergence is not
achieved in maxiter iterations, an error is raised. Must be >= 0.
-- ``full_output`` - bool (default: False); if True,
also return object that contains information about convergence.

EXAMPLES:
Note that in this example both f(-2) and f(3) are positive,
yet we still find a root in that interval::

sage: f = x^2 - 1
sage: f.find_root(-2, 3)
1.0
sage: f.find_root(-2, 3, x)
1.0
sage: r, result = f.find_root(-2, 3, full_output=True)
sage: result.converged
True
sage: result.flag
'converged'
sage: result.function_calls
11
sage: result.iterations

```

(Refer Slide Time: 05:47)



The screenshot shows a JupyterLab window with a SageMath 9.1 kernel. The code in the cell is as follows:

```
sage: f = x^2 - 3
sage: f.find_root(-2, 3)
1.0
sage: f.find_root(-2, 3, x)
1.0
sage: z, result = f.find_root(-2, 3, full_output=True)
sage: result.converged
True
sage: result.flag
'converged'
sage: result.function_calls
11
sage: result.iterations
10
sage: result.root
1.0
```

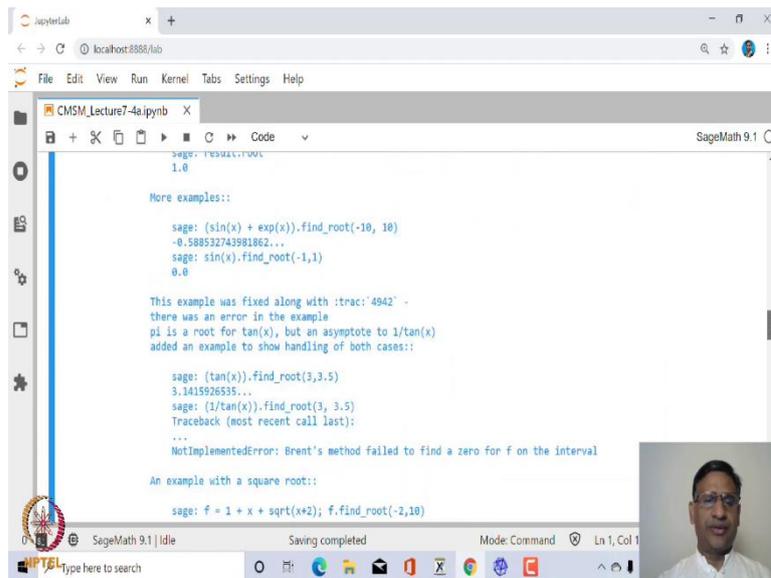
More examples::

```
sage: (sin(x) + exp(x)).find_root(-10, 10)
-0.588532743981862...
sage: sin(x).find_root(-1,1)
0.0
```

This example was fixed along with :trac:4942 -

The interface also shows a video feed of the presenter in the bottom right corner.

(Refer Slide Time: 05:48)



The screenshot shows a JupyterLab window with a SageMath 9.1 kernel. The code in the cell is as follows:

```
sage: result.root
1.0
```

More examples::

```
sage: (sin(x) + exp(x)).find_root(-10, 10)
-0.588532743981862...
sage: sin(x).find_root(-1,1)
0.0
```

This example was fixed along with :trac:4942 -
there was an error in the example
pi is a root for tan(x), but an asymptote to 1/tan(x)
added an example to show handling of both cases::

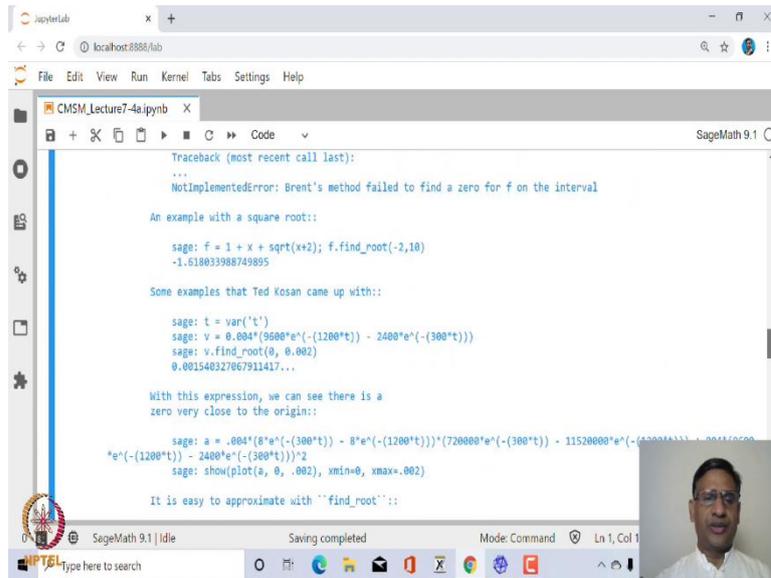
```
sage: (tan(x)).find_root(3,3.5)
3.1415926535...
sage: (1/tan(x)).find_root(3, 3.5)
Traceback (most recent call last):
...
NotImplementedError: Brent's method failed to find a zero for f on the interval
```

An example with a square root::

```
sage: f = 1 + x + sqrt(x+2); f.find_root(-2,10)
```

The interface also shows a video feed of the presenter in the bottom right corner.

(Refer Slide Time: 05:48)



The screenshot shows a JupyterLab window with a SageMath 9.1 kernel. The code cell contains the following text:

```
Traceback (most recent call last):
...
NotImplementedError: Brent's method failed to find a zero for f on the interval

An example with a square root::

sage: f = 1 + x + sqrt(x+2); f.find_root(-2,10)
-1.618033988749895

Some examples that Ted Kosan came up with::

sage: t = var('t')
sage: v = 0.004*(9600*e^(-(1200*t)) - 2400*e^(-(300*t)))
sage: v.find_root(0, 0.002)
0.001540327067911417...

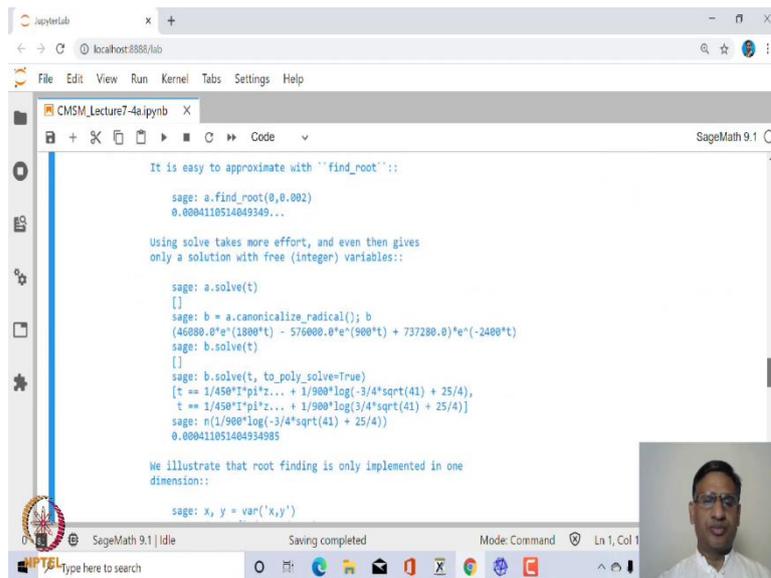
With this expression, we can see there is a
zero very close to the origin::

sage: a = .004*(e^(-300*t) - 8*e^(-(1200*t)))*(720000*e^(-(300*t)) - 11520000*e^(-(1200*t)))
sage: a.find_root(0, 0.002)
0.001540327067911417...

It is easy to approximate with "find_root"::
```

A video feed of a man with glasses is visible in the bottom right corner of the JupyterLab window.

(Refer Slide Time: 05:49)



The screenshot shows a JupyterLab window with a SageMath 9.1 kernel. The code cell contains the following text:

```
It is easy to approximate with "find_root"::

sage: a.find_root(0,0.002)
0.0004110514049349...

Using solve takes more effort, and even then gives
only a solution with free (integer) variables::

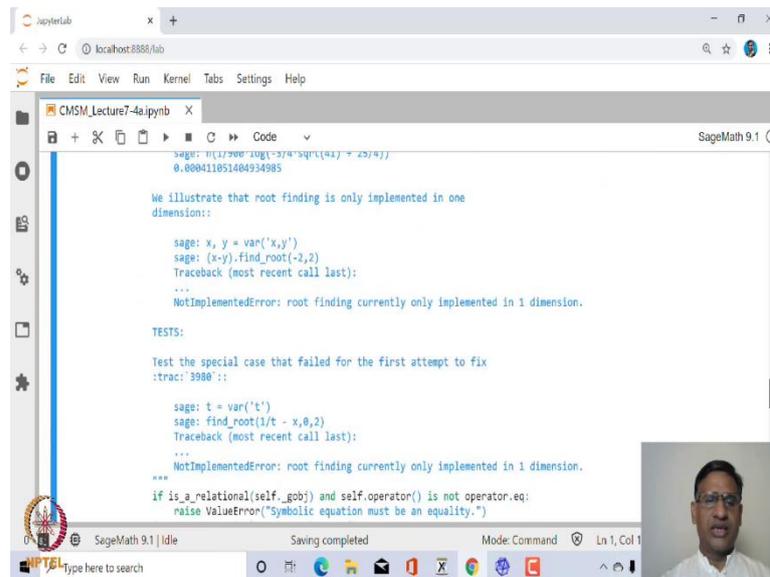
sage: a.solve(t)
[]
sage: b = a.canonicalize_radical(); b
(46080.0*e^(1800*t) - 576000.0*e^(900*t) + 737280.0)*e^(-2400*t)
sage: b.solve(t)
[]
sage: b.solve(t, to_poly_solve=True)
[t == 1/450*I*pi*t... + 1/900*log(-3/4*sqrt(41) + 25/4),
 t == 1/450*I*pi*t... + 1/900*log(3/4*sqrt(41) + 25/4)]
sage: n(1/900*log(-3/4*sqrt(41) + 25/4))
0.000411051404934985

We illustrate that root finding is only implemented in one
dimension::

sage: x, y = var('x,y')
```

A video feed of a man with glasses is visible in the bottom right corner of the JupyterLab window.

(Refer Slide Time: 05:50)



```
sage: m(1/3980*sqrt(-3/4*sqrt(41)+2/47))
0.0084110514084934985

We illustrate that root finding is only implemented in one
dimension::

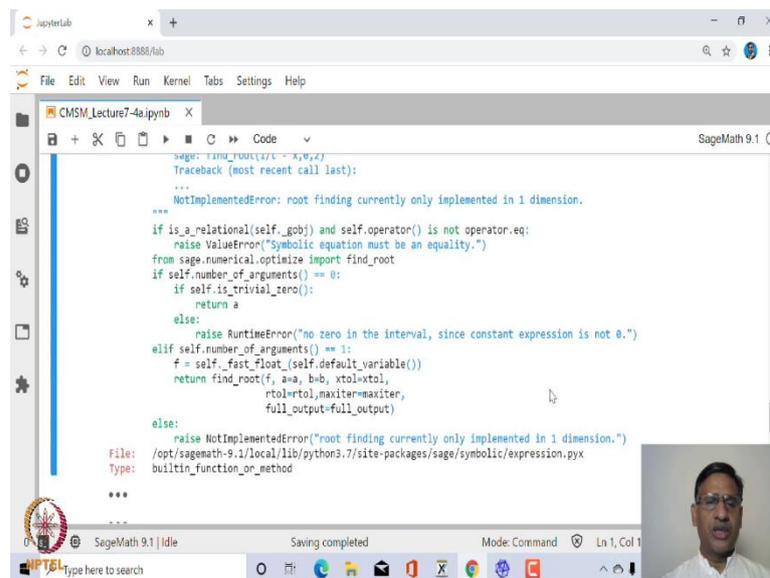
sage: x, y = var('x,y')
sage: (x-y).find_root(-2,2)
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.

TESTS:

Test the special case that failed for the first attempt to fix
:trac: 3980 ::

sage: t = var('t')
sage: find_root(1/t - x,0,2)
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.
===
if is_a_relational(self_gobj) and self.operator() is not operator.eq:
    raise ValueError("Symbolic equation must be an equality.")
```

(Refer Slide Time: 05:50)



```
sage: find_root(1/t - x,y,2)
Traceback (most recent call last):
...
NotImplementedError: root finding currently only implemented in 1 dimension.
===
if is_a_relational(self_gobj) and self.operator() is not operator.eq:
    raise ValueError("Symbolic equation must be an equality.")
from sage.numerical.optimize import find_root
if self.number_of_arguments() == 0:
    if self.is_trivial_zero():
        return a
    else:
        raise RuntimeError("no zero in the interval, since constant expression is not 0.")
elif self.number_of_arguments() == 1:
    f = self._fast_float(self.default_variable())
    return find_root(f, a=a, b=b, xtol=xtol,
                    rtol=rtol, maxiter=maxiter,
                    full_output=full_output)
else:
    raise NotImplementedError("root finding currently only implemented in 1 dimension.")
File: /opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/symbolic/expression.pyx
Type: builtin_function_or_method
...

```

So, you can go through this document and see there are several examples. It also tells you how this function is defined. So, basically, if you look at, this actually uses a function find root, which is defined inside SageMath numerical optimize package. So, if I, if I try to take a help on this package, then let us see what we are going to get.

(Refer Slide Time: 06:17)

```
NotImplementedError: root finding currently only implemented in 1 dimension.
'''
if is_a_relational(self._gobj) and self.operator() is not operator.eq:
    raise ValueError("Symbolic equation must be an equality.")
from sage.numerical.optimize import find_root
if self.number_of_arguments() == 0:
    if self.is_trivial_zero():
        return a
    else:
        raise RuntimeError("no zero in the interval, since constant expression is not 0.")
elif self.number_of_arguments() == 1:
    f = self.fast_float(self.default_variable())
    return find_root(f, a=a, b=b, xtol=xtol,
                    rtol=rtol, maxiter=maxiter,
                    full_output=full_output)
else:
    raise NotImplementedError("root finding currently only implemented in 1 dimension.")
File: /opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/symbolic/expression.pyx
Type: builtin_function_or_method

[5]: import sage.numerical.optimize
[ ]: sage.numerical.optimize.|
```

So, let us import, import sage dot numerical dot, right? Once we have imported this, then, let us see what we will get? If I say sage dot numerical dot optimize dot, and then tab. (Refer Slide Time: 06:38)

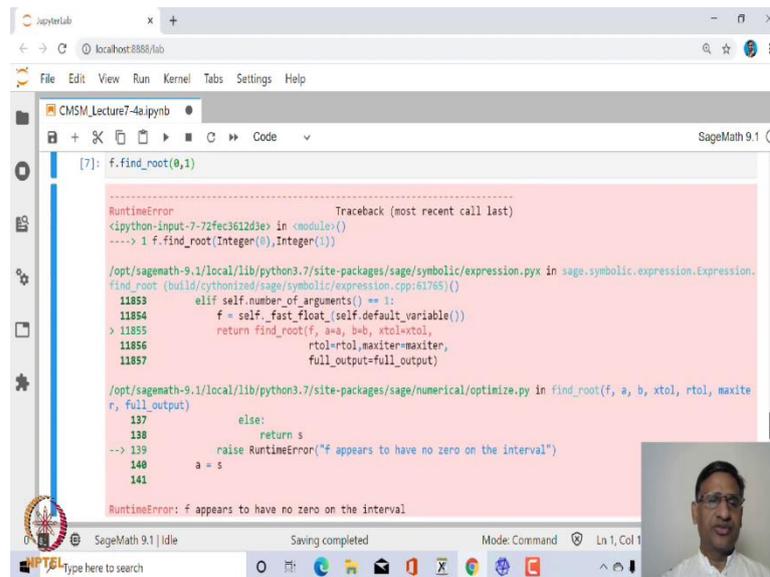
```
NotImplementedError: root finding currently only implemented in 1 dimension.
'''
if is_a_relational(self._gobj) and self.operator() is not operator.eq:
    raise ValueError("Symbolic equation must be an equality.")
from sage.numerical.optimize import find_root
if self.number_of_arguments() == 0:
    if self.is_trivial_zero():
        return a
    else:
        raise RuntimeError("no zero in the interval, since constant expression is not 0.")
elif self.number_of_arguments() == 1:
    f = self.fast_float(self.default_variable())
    return find_root(f, a=a, b=b, xtol=xtol,
                    rtol=rtol, maxiter=maxiter,
                    _output)
else:
    raise NotImplementedError("root finding currently only implemented in 1 dimension.")
File: /opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/symbolic/expression.pyx

[ ]: sage.numerical.optimize.|
```

- sage.numerical.optimize.binning
- sage.numerical.optimize.find_fit
- sage.numerical.optimize.find_local_maximum
- sage.numerical.optimize.find_local_minimum
- sage.numerical.optimize.find_root
- sage.numerical.optimize.linear_program
- sage.numerical.optimize.minimize
- sage.numerical.optimize.minimize_constrained
- sage.numerical.optimize.range
- sage.numerical.optimize.RDF

So, you can see here, this has many functions. One of them is find fit, which again we have made use of, when we wanted to find, find or fit best fit line or parabola, etcetera, find underscore local maximum, find local minimum, this also we have seen in, in calculus. It can also solve linear programs, that is linear programming problem, and it can find a minimizer, and similarly constrained minimization function also, right?

(Refer Slide Time: 08:11)



```
[7]: f.find_root(0,1)

RuntimeError                                Traceback (most recent call last)
<ipython-input-7-72fec3612d3e> in <module>()
----> 1 f.find_root(Integer(0),Integer(1))

/opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/symbolic/expression.pyx in sage.symbolic.expression.Expression.find_root (build/cythonized/sage/symbolic/expression.cpp:61765):
11853     elif self.number_of_arguments() == 1:
11854         f = self._fast_float(self.default_variable())
11855         return find_root(f, a=s, b=b, xtol=xtol,
11856                        rtol=rtol,maxiter=maxiter,
11857                        full_output=full_output)

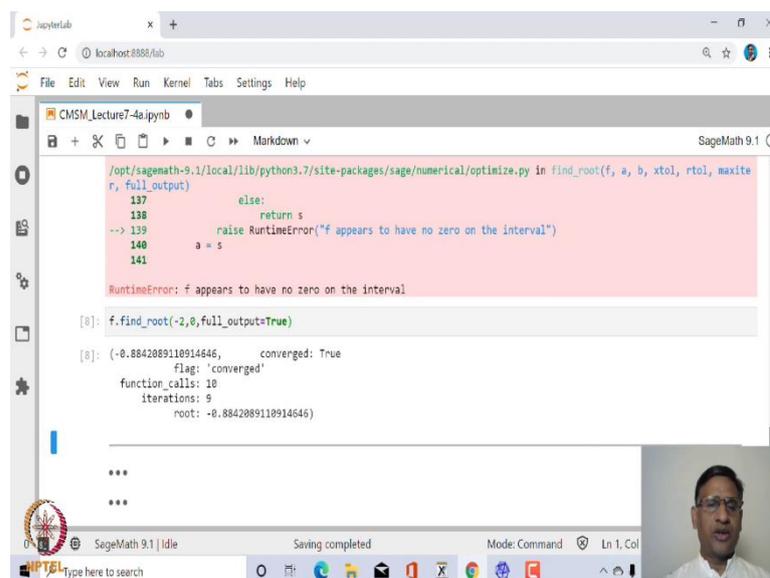
/opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/numerical/optimize.py in find_root(f, a, b, xtol, rtol, maxite
r, full_output)
137
138     else:
--> 139         return s
140     raise RuntimeError("f appears to have no zero on the interval")
141     a = s

RuntimeError: f appears to have no zero on the interval
```

In case we want to find a root between 0 and 1, if I say, if I give the interval 0 and 1, then we, we already know from the plot that this does not have a root, and therefore, it will give you an error.

This is an error. It says runtime error, and at the end, it tells you that f appears to have no zero on this interval. So, in case you are providing an interval which does not contain a root using find root function, then it will throw an error, right?

(Refer Slide Time: 08:51)



```
/opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/numerical/optimize.py in find_root(f, a, b, xtol, rtol, maxite
r, full_output)
137
138     else:
--> 139         return s
140     raise RuntimeError("f appears to have no zero on the interval")
141     a = s

RuntimeError: f appears to have no zero on the interval

[8]: f.find_root(-2,0,full_output=True)

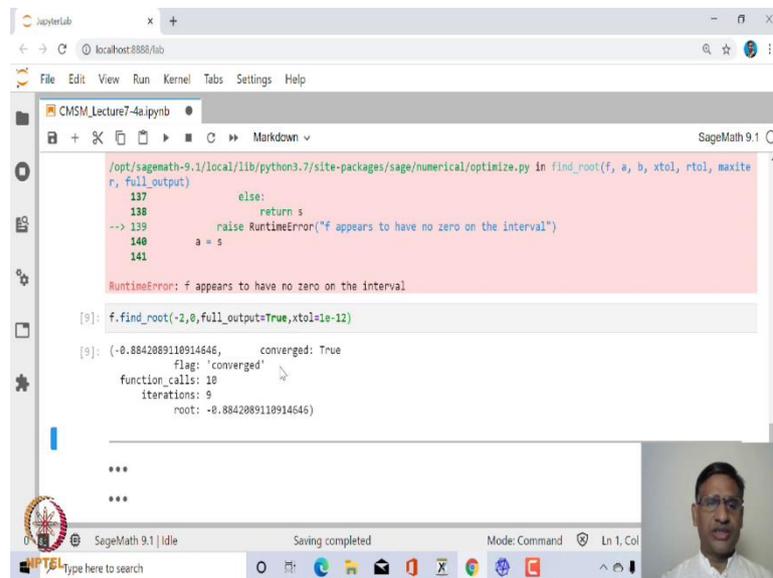
[8]: (-0.8842089110914646, converged: True
      flag: 'converged'
      function_calls: 10
      iterations: 9
      root: -0.8842089110914646)

...
...
```

You can also give an option called full output equal to True, in that case, what does it say? It, it gives you root along with an option converged, and it says that converged equal to True.

So, numerical method, when you applied any numerical methods, it may or may not converge. There are certain criteria when sequence of iterate converges to a solution. So, in case it has not converged, it will give you the answer false, and then it also tells you that how many times the function has been called. In this case, 10 times function were called, and the number of iterations which is required is, is 9, and the root lies at this. If I increase the tolerance limit?

(Refer Slide Time: 09:45)



```

/opt/sagemath-9.1/local/lib/python3.7/site-packages/sage/numerical/optimize.py in find_root(f, a, b, xtol, rtol, maxite
r, full_output)
   137     else:
   138         return s
--> 139     raise RuntimeError("f appears to have no zero on the interval")
   140     a = s
   141

RuntimeError: f appears to have no zero on the interval

[9]: f.find_root(-2,0,full_output=True,xtol=1e-12)

[9]: (-0.8842089110914646, converged: True
      flag: 'converged'
      function_calls: 10
      iterations: 9
      root: -0.8842089110914646)

...
...

```

So, for example, if I say tolerance, I think tolerance, I think it is called xtol is equal to, let us say, 10 to the power, let us say, 1 e minus 12, then it again it requires only 10 tols, right? So, let us, let us look at how the x tolerance, it is already 10 to the power 13. So, it does not, 10 power 10 is beyond that.

(Refer Slide Time: 10:22)

```

r, full_output)
137         else:
138             return s
--> 139         raise RuntimeError("f appears to have no zero on the interval")
140         a = s
141
RuntimeError: f appears to have no zero on the interval

[10]: f.find_root(-2,0,full_output=True,xtol=1e-20)
[10]: (-0.8842089110914646, converged: True
      flag: 'converged'
      function_calls: 10
      iterations: 9
      root: -0.8842089110914646)
...
...
...

```

If I say 10 power 20, it may require more number of iterates. Now it, it requires only the. So, it is, in this case, the, the convergence is quite fast. Whatever method it is using, it converges quite quickly, converges quite quickly, right? So, you, this is what you can use, namely find underscore fit, to a given function in order to find a root of this function.

So, Sage provides you very nice way of plotting the graphs. So, first you can plot graph of the function and locate the interval in which it has a root, and then apply find underscore fit, right? That is how, that is how you can find root of an algebraic and transcendental equations. As I said, you can write your own program. You can write python program and run in this Sage worksheet.

(Refer Slide Time: 11:16)

```

Bisection Method

[ ]: def bisection_method(f, a, b, tol=1.0e-5):
      a=float(a)
      b=float(b)
      tol = float(tol)
      intervals = [(a,b)]
      n = int(ceil(log(abs(b - a)/tol)/log(2.0)))
      i=0
      while i<=n:
          c = (a+b)/2.0
          fa = f(a); fb = f(b); fc = f(c)
          if fa*fc < 0:
              a, b = a, c
          elif fb*fc < 0:
              a, b = c, b
          intervals.append((a,b))
          i=i+1
      return c, intervals
...

```

So, for example, let us look at, suppose we want to develop our own program in order to find a root of this function using bisection method. So, bisection method is very simple, and I am sure all of you would have studied this bisection method. So, let me just show you this user-defined function for bisection methods, which requires input f, the interval a and b, and the tolerance is 10 to the power minus 5.

(Refer Slide Time: 11:54)

```

def bisection_method(f, a, b, tol=1.0e-10):
    a=float(a)
    b=float(b)
    tol = float(tol)
    intervals = [(a,b)]
    n = int(ceil(log(abs(b - a)/tol)/log(2.0)))
    i=0
    while i<n:
        c = (a+b)/2.0
        fa = f(a); fb = f(b); fc = f(c)
        if fa*fc < 0:
            a, b = a, c
        elif fc*fb < 0:
            a, b = c, b
        intervals.append((a,b))
        i=i+1
    return c, intervals

```

You can increase this tolerance to 10 to power whatever, 10 to the power, let us say, I will say 10 to power minus 10. And a and b are made float because in case you give this integer, then most often you may end up with rational number as output, which may look somewhat ugly.

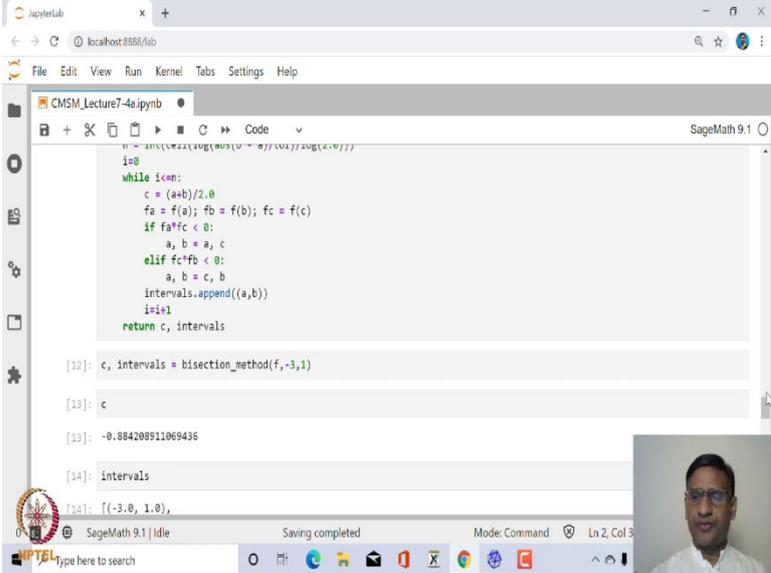
And then you, you also convert this tolerance to float, define initial interval as [a,b]. So, this will give you a tuple of intervals, and then the number of iterates which is required in order to find a root in an interval of length, let us say, a tolerance limit, in this case 10 to power minus 10, that is given by n is equal to integer part of, integer part of log of absolute value of b minus a divided by the tolerance limit divided by log of 2, right?

So, and then start with iteration 0, and then find the value of a plus b by 2, that is mid-point. Check value of f at a, f at b, and f at c. In case f at a and f at c product is less than 0, that means, root lies between a and c. So, you redefine a, b as a, c. And otherwise you redefine this as c, a, c, b.

Of course, you could also check if a, if c is a root, that is, if $f(c)$ is equal to 0, or value of $f(c)$ is less than the tolerance limit, then you can stop here itself. You can use break, or otherwise you keep on appending this interval a, b in the interval a, b, and then increment that the iteration by 1. And then at the end, you return the midpoint and also the set of the intervals, ok?

So, this is you can, of course, modify this, include more, more options here, for example, you can check whether a is less than b or not, you can also check whether a, [a,b] contains a root, all these things can be, can be included here, right?

(Refer Slide Time: 14:05)



```
def bisection_method(f, a, b):
    intervals = []
    i = 0
    while i <= 10:
        c = (a+b)/2.0
        fa = f(a); fb = f(b); fc = f(c)
        if fa*fc < 0:
            a, b = a, c
        elif fb*fc < 0:
            a, b = c, b
        intervals.append((a,b))
        i = i + 1
    return c, intervals

[12]: c, intervals = bisection_method(f, -3, 1)

[13]: c

[13]: -0.884208511069436

[14]: intervals

[14]: [(-3.0, 1.0),
```

Now let us call this function bisection method for the same function $f(x)$ which we chose earlier. So, this is, now let us see what is the value of c, c is this, and if I look at what is the interval, interval, inter, interval, that intervals is what I have given.

(Refer Slide Time: 14:38)

(Refer Slide Time: 14:56)

```
def f(x):  
    a, b = a, c  
    elif f*c < 0:  
        a, b = c, b  
    intervals.append((a,b))  
    i=i+1  
return c, intervals  
  
[15]: c, intervals = bisection_method(f, -3, 0)  
  
[16]: c  
-0.884208511083988  
  
[17]: intervals  
[(-3, 0, 0.0),  
 (-1.5000000000000000, 0.0),  
 (-1.5000000000000000, -0.7500000000000000),  
 (-1.1250000000000000, -0.7500000000000000),  
 (-0.9375000000000000, -0.7500000000000000),  
 (-0.9375000000000000, -0.8437500000000000),  
 ...]
```

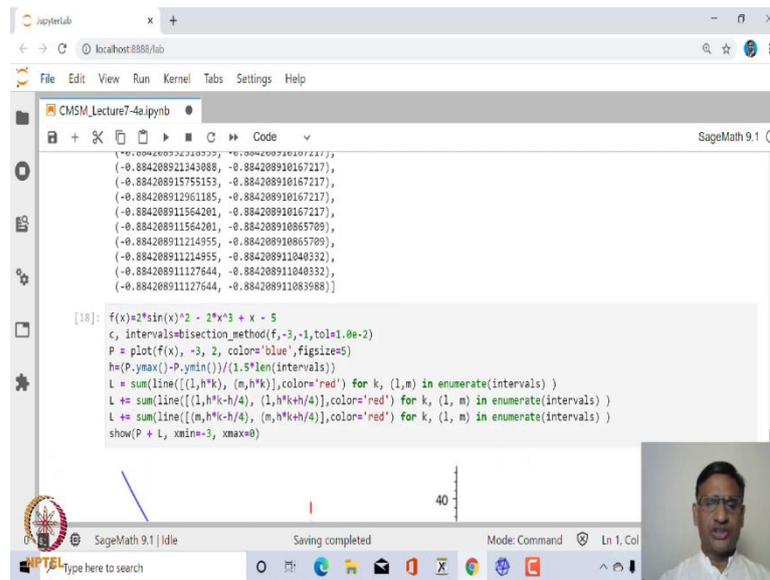
So, in this case, let me, let me, instead of minus 3 to 3, let me give you minus 3 to yeah, ok, minus 3 to 0, and this is what you get.

(Refer Slide Time: 15:07)

```
[(-0.884216308593750, -0.884032091250000),  
 (-0.884216308593750, -0.884124755859375),  
 (-0.884216308593750, -0.884179532265625),  
 (-0.884216308593750, -0.884193420401562),  
 (-0.884216308593750, -0.884204864501953),  
 (-0.884210586547852, -0.884204864501953),  
 (-0.884210586547852, -0.88420772524902),  
 (-0.884209156036377, -0.88420772524902),  
 (-0.884209156036377, -0.884208440780640),  
 (-0.884209156036377, -0.884208796408508),  
 (-0.884208977222443, -0.884208796408508),  
 (-0.884208977222443, -0.884208887815475),  
 (-0.884208932518959, -0.884208887815475),  
 (-0.884208932518959, -0.884208910167217),  
 (-0.884208921343088, -0.884208910167217),  
 (-0.88420891575153, -0.884208910167217),  
 (-0.884208912961185, -0.884208910167217),  
 (-0.884208911564201, -0.884208910167217),  
 (-0.884208911564201, -0.884208910865709),  
 (-0.884208911214955, -0.884208910865709),  
 (-0.884208911214955, -0.884208911040332),  
 (-0.88420891127644, -0.884208911040332),  
 (-0.88420891127644, -0.884208911083988)]
```

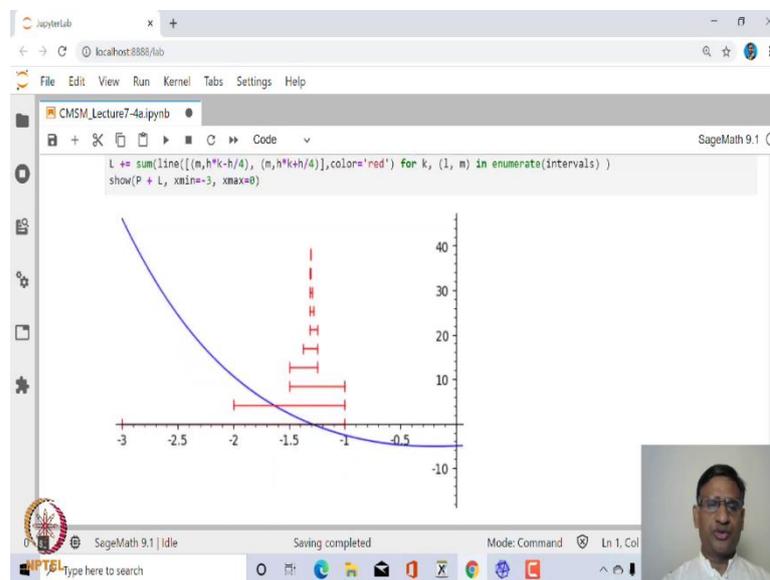
So, you can see here, there are so many iterations which is required in this case.

(Refer Slide Time: 15:10)



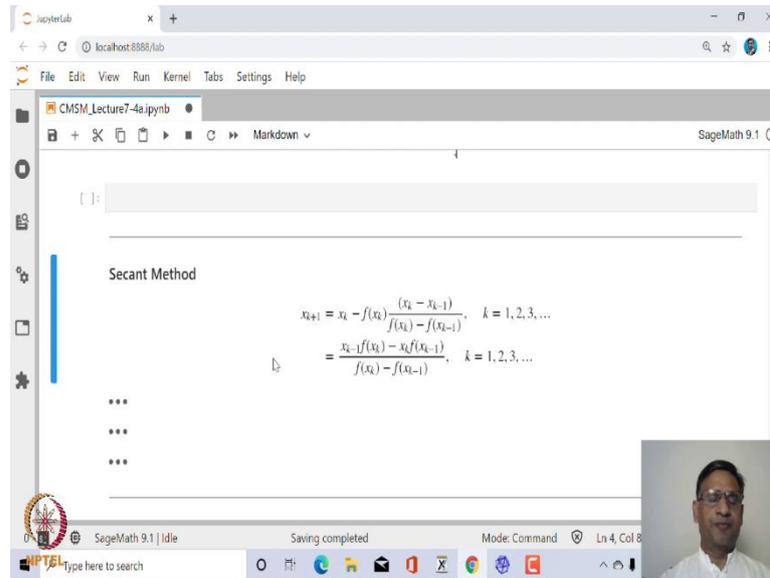
And one can even try to plot all these intervals bisecting intervals containing root, just to visualize how this iteration proceeds.

(Refer Slide Time: 15:23)



So, you can see here, the initial interval is minus 3 to minus 1, and then this is a root, and this is the 1st interval, this is the 2nd interval, 3rd interval, 4th, and so on, you can see here the length of the interval is becoming very small, right?

(Refer Slide Time: 15:44)



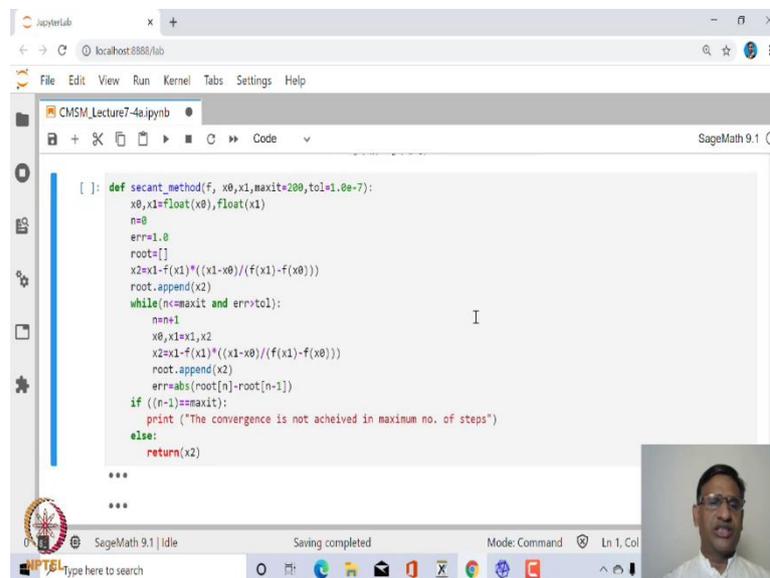
The screenshot shows a JupyterLab window with a SageMath 9.1 kernel. The main content area displays the title "Secant Method" followed by the mathematical formula for the next iteration:

$$x_{k+1} = x_k - f(x_k) \frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}, \quad k = 1, 2, 3, \dots$$
$$= \frac{x_k - f(x_k) \frac{(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}}{f(x_k) - f(x_{k-1})}, \quad k = 1, 2, 3, \dots$$

Below the formula are three vertical ellipses. The interface includes a file explorer on the left, a top menu bar, and a bottom status bar with a taskbar and a small video feed of the presenter.

Next let us write our own program for Secant Method. The secant method this a iteration scheme is given by x_{k+1} is equal to x_k minus f at x_k into x_k minus x_{k-1} divided by f of x_k minus f of x_{k-1} again this I am sure you must have seen this.

(Refer Slide Time: 16:04)



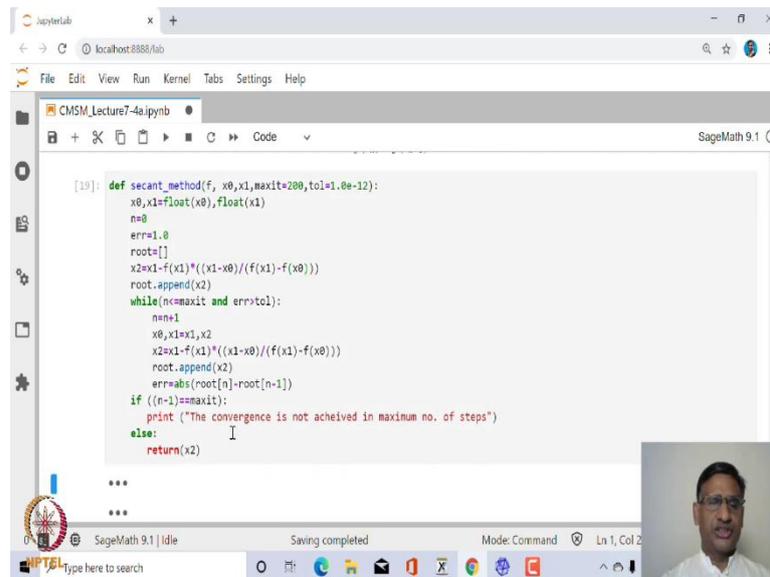
The screenshot shows the same JupyterLab window with a SageMath 9.1 kernel. The main content area displays a Python function definition for the secant method:

```
def secant_method(f, x0, x1, maxit=200, tol=1.0e-7):  
    x0, x1 = float(x0), float(x1)  
    n = 0  
    err = 1.0  
    root = []  
    x2 = x1 - f(x1) * ((x1 - x0) / (f(x1) - f(x0)))  
    root.append(x2)  
    while (n < maxit and err > tol):  
        n = n + 1  
        x0, x1 = x1, x2  
        x2 = x1 - f(x1) * ((x1 - x0) / (f(x1) - f(x0)))  
        root.append(x2)  
        err = abs(root[n] - root[n-1])  
    if ((n-1) == maxit):  
        print("The convergence is not achieved in maximum no. of steps")  
    else:  
        return(x2)
```

The function takes a function f and two initial points x_0 and x_1 as input. It iteratively refines the root estimate until it either reaches a maximum of 200 iterations or the error is within a tolerance of 1.0×10^{-7} . The interface includes a file explorer on the left, a top menu bar, and a bottom status bar with a taskbar and a small video feed of the presenter.

So, let us write a program for the secant method, again, very similar inputs we are giving, maximum number of iteration is 200, tolerance limit is 10 power minus 7.

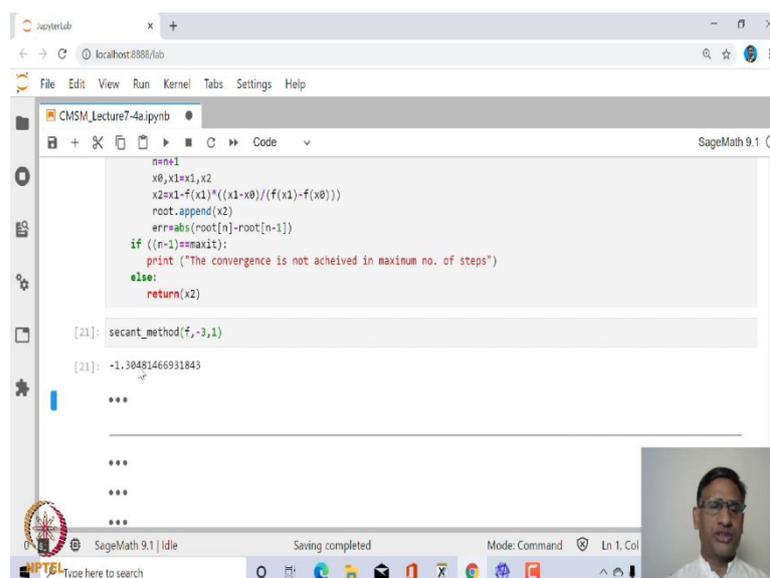
(Refer Slide Time: 16:16)



```
[19]: def secant_method(f, x0, x1, maxit=200, tol=1.0e-12):
      x0, x1 = float(x0), float(x1)
      n = 0
      err = 1.0
      root = []
      x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
      root.append(x2)
      while (n < maxit and err > tol):
          n = n + 1
          x0, x1 = x1, x2
          x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
          root.append(x2)
          err = abs(root[n] - root[n-1])
      if ((n-1) == maxit):
          print("The convergence is not achieved in maximum no. of steps")
      else:
          return(x2)
```

You can increase this to 10 to the power minus 12 and then let us execute this. In case you find that the number of iterate is equal to the maximum, and still you have not obtained this within the tolerance limit, you can print a message saying that the convergent is not achieved. And that is same as what you, you saw flag in case of inbuilt function find underscore root.

(Refer Slide Time: 16:44)



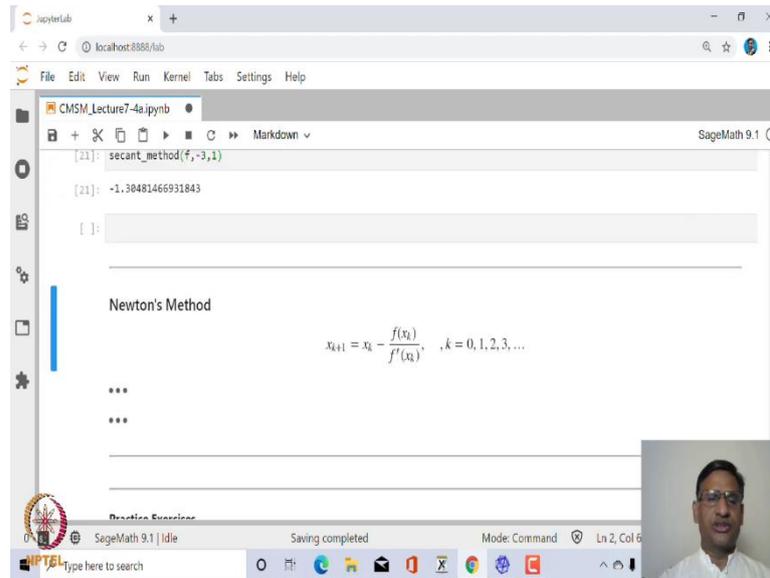
```
n = n + 1
x0, x1 = x1, x2
x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))
root.append(x2)
err = abs(root[n] - root[n-1])
if ((n-1) == maxit):
    print("The convergence is not achieved in maximum no. of steps")
else:
    return(x2)

[21]: secant_method(f, -3, 1)

[21]: -1.38481466931843
```

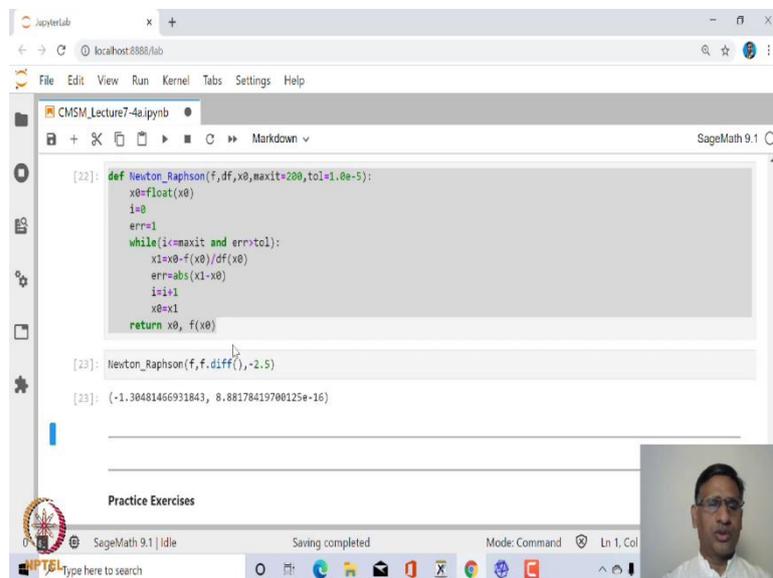
Now let us let us run this, and call this function for $f(x)$. This is again, the root is minus 3, minus 1.30 and so on, ok.

(Refer Slide Time: 17:00)



Similarly, you can, you can now find, write your own program for Newton's method. The iteration scheme in Newton's method is given by x_{k+1} is equal to x_k minus f of x_k divided by f dash at x_k .

(Refer Slide Time: 17:12)

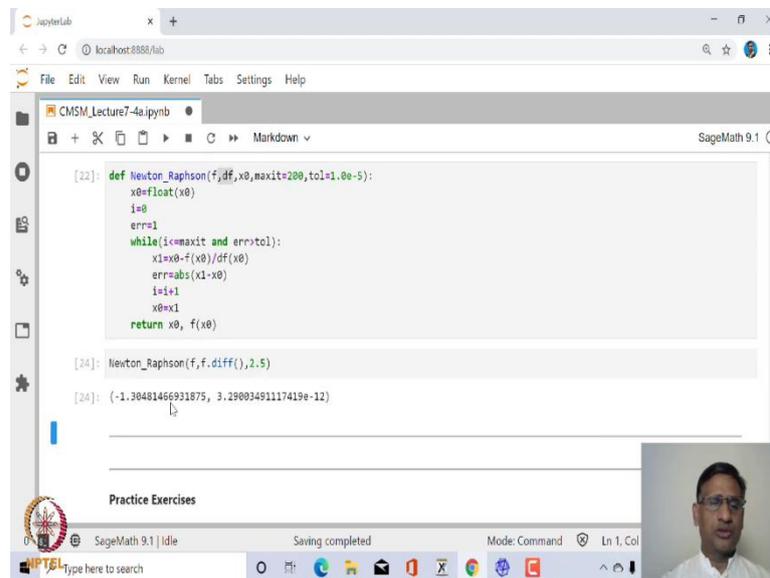


And you can create user-defined function for this. Fairly simple, very simple program, this we did in, in python also. And this, you can see here, this is exactly a python program, it

has nothing to do with Sage, we are not using any Sage functionality here. We are using Sage functionality only for plotting and visualization, right?

So, and then if we, if we run this, call this Newton's method, we, in this case, you need to input f and df that is derivative, right? Of course, without this also you can make use of, now, Sage functionality, and find the derivative to, define this df as f dot diff , and then, then run this, and in this case, you can see here we have passed df as f dot diff , and the initial guess is minus 2.5.

(Refer Slide Time: 18:09)



```
[22]: def Newton_Raphson(f,df,x0,maxit=200,tol=1.0e-5):
      x=float(x0)
      i=0
      err=1
      while(i<=maxit and err>tol):
          x1=x0-f(x0)/df(x0)
          err=abs(x1-x0)
          i=i+1
          x0=x1
      return x0, f(x0)

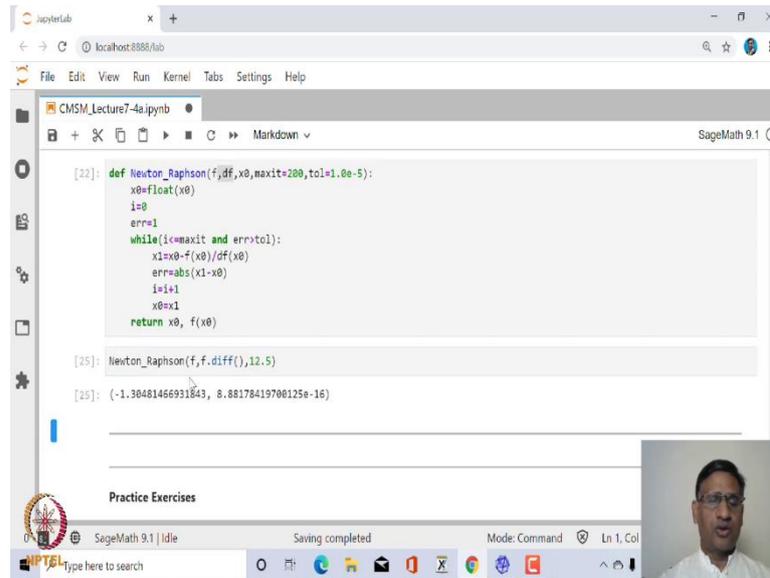
[24]: Newton_Raphson(f,f.diff(),2.5)

[26]: (-1.30481466931875, 3.29003491117419e-12)
```

Practice Exercises

If I give initial guess, let us say 2.5, then also it is able to find.

(Refer Slide Time: 18:16)



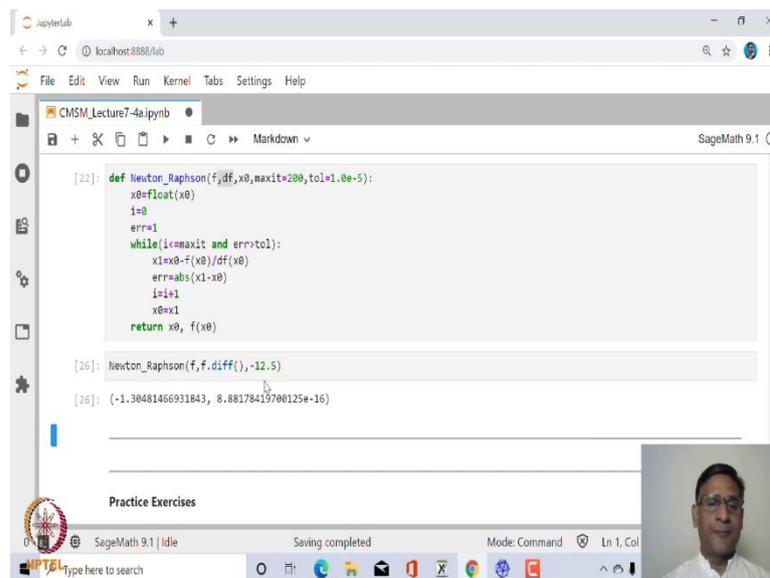
```
[22]: def Newton_Raphson(f,df,x0,maxit=200,tol=1.0e-5):
      x=float(x0)
      i=0
      err=1
      while(i<maxit and err>tol):
          x1=x0-f(x0)/df(x0)
          err=abs(x1-x0)
          i=i+1
          x0=x1
      return x0, f(x0)

[25]: Newton_Raphson(f,f.diff(),12.5)

[25]: (-1.30481466931843, 8.88178419700125e-16)
```

Practice Exercises

(Refer Slide Time: 18:20)



```
[22]: def Newton_Raphson(f,df,x0,maxit=200,tol=1.0e-5):
      x=float(x0)
      i=0
      err=1
      while(i<maxit and err>tol):
          x1=x0-f(x0)/df(x0)
          err=abs(x1-x0)
          i=i+1
          x0=x1
      return x0, f(x0)

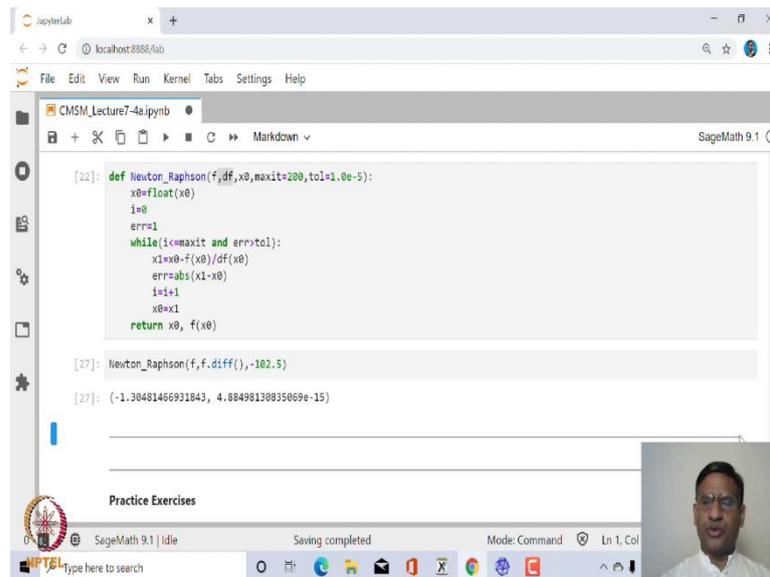
[26]: Newton_Raphson(f,f.diff(),-12.5)

[26]: (-1.30481466931843, 8.88178419700125e-16)
```

Practice Exercises

So, this is working, if I give you, let us say, 12.5, then also it is able to find, in case, if I give minus 12.5, then also it is able to find. So, this, this this function is quite nice, it seems that whatever initial guess you give, then it is able to find a root.

(Refer Slide Time: 18:34)

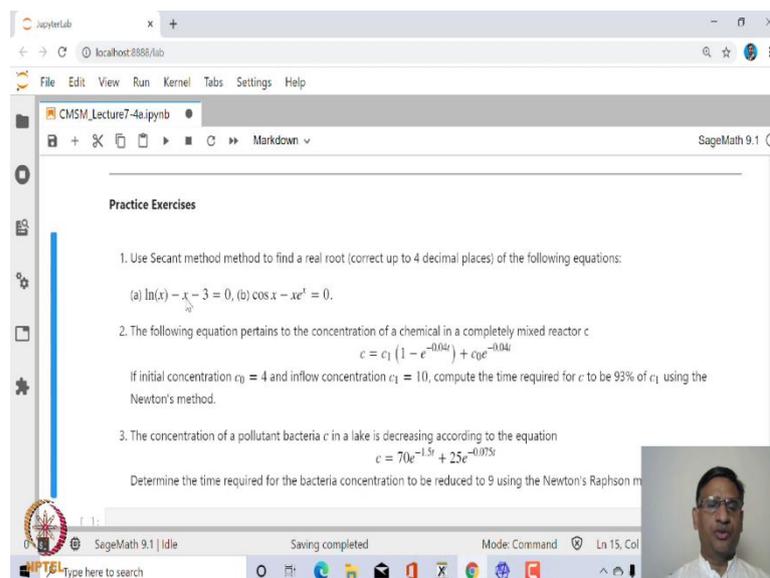


```
[22]: def Newton_Raphson(f,df,x0,maxit=200,tol=1.0e-5):
      x0=float(x0)
      i=0
      err=1
      while(i<=maxit and err>tol):
          x1=x0-f(x0)/df(x0)
          err=abs(x1-x0)
          i=i+1
          x0=x1
      return x0, f(x0)

[27]: Newton_Raphson(f,f.diff(),-102.5)

[27]: (-1.30481466931843, 4.88498130835069e-15)
```

So, let us say, for example, if I give minus 102.5, this, then also it is able to, to, to find, ok? So, this, because it is looking at 200 iterates, ok, fine? So, let me, so what it means is that you can write your own programs for all these methods, there are several other methods to find a root of a, a quadratic, root, root of an algebraic and transcendental equation, equations. So, either you can use find root, or you can create your own user-defined functions, right? (Refer Slide Time: 19:20)



Practice Exercises

- Use Secant method to find a real root (correct up to 4 decimal places) of the following equations:
(a) $\ln(x) - x - 3 = 0$, (b) $\cos x - xe^x = 0$.
- The following equation pertains to the concentration of a chemical in a completely mixed reactor $c = c_1 (1 - e^{-0.04t}) + c_0 e^{-0.04t}$. If initial concentration $c_0 = 4$ and inflow concentration $c_1 = 10$, compute the time required for c to be 93% of c_1 using the Newton's method.
- The concentration of a pollutant bacteria c in a lake is decreasing according to the equation $c = 70e^{-1.5t} + 25e^{-0.075t}$. Determine the time required for the bacteria concentration to be reduced to 9 using the Newton's Raphson method.

So, let me leave you with few simple exercises. So, try to solve these equations, find roots of these equations using secant method, this one using Newton's method, and this also

using Newton's method. And in each of these case, you can verify this using inbuilt function `find underscore root`.

Of course, you should also plot graph and then try to see the interval in which it has a root, ok? So, next time we will look at how to solve set of, or system of linear equations numerically.