

# lab\_taylor\_newton

September 17, 2019

## 1 Taylor Approximation and Newton's Method

### 1.1 Taylor Approximation

For analytic function equivalence between value of a function at each point or value of all derivatives at a single point.

We consider approximating the function

$$h(t) = t^2 e^{3t}$$

in the neighbourhood of some value  $t = t_0$ . The first-order Taylor approximation to  $h(t)$  is

$$h_1(t) = h(t_0) + h'(t_0)(t - t_0),$$

where  $h'(t)$  is the derivative of  $h(t)$ .

We define the symbolic function  $h(t)$  and find its symbolic derivative  $h'(t)$ :

```
In [2]: import sympy as sp
        sp.init_printing()
        from IPython.display import display

        t = sp.symbols('t');
        h = sp.exp(3*t)*t**2; # h(t)
        hp = h.diff(t); # h'(t)
        hp
```

Out [2]:

$$3t^2 e^{3t} + 2te^{3t}$$

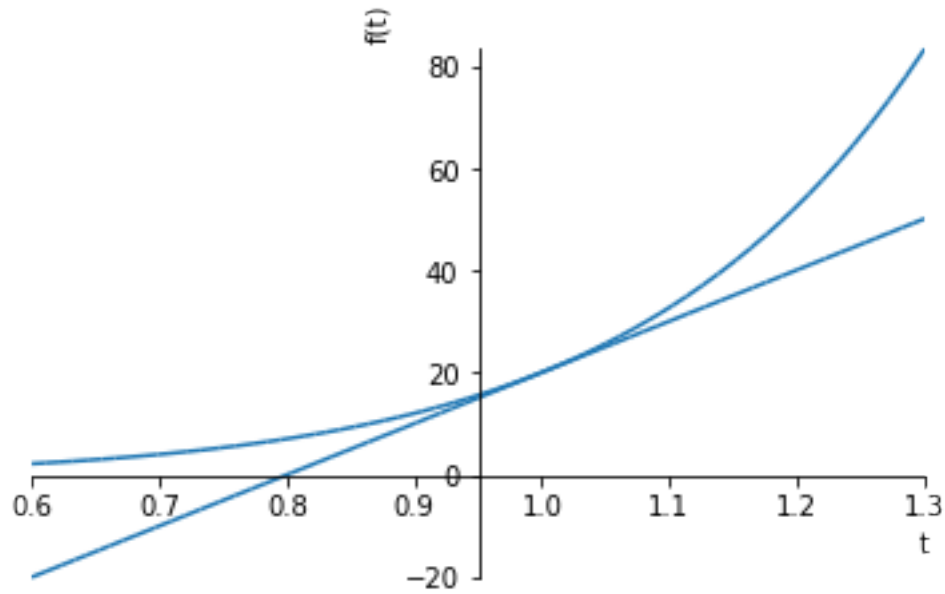
We want to evaluate  $h(t_0)$  and  $h'(t_0)$  so that we can use the approximation. We don't yet have to put in a value so we just substitute the symbol  $t_0$ . The result will then be a symbolic expression in  $t$  and  $t_0$ :

```
In [8]: t0 = sp.symbols('t0');
        h0 = h.subs(t,t0); # h(t0)
        h1 = h0 + hp.subs({t:t0})*(t-t0); # h1(t) Taylor approximation at t0
        display(h1);
```

$$t_0^2 e^{3t_0} + (t - t_0) (3t_0^2 e^{3t_0} + 2t_0 e^{3t_0})$$

For a fixed value for  $t_0$  we can plot the symbolic function  $h(t)$  and the first-order Taylor expansion  $h'(t)$  at that point:

```
In [9]: h1s = h1.subs(t0, 1); # h1(t) for value t0=1
        sp.plot(h, h1s, (t,0.6,1.3));
```



The previous cell used a symbolic plotting function. We could instead convert the symbolic expressions to functions that can be evaluated numerically using

```
In [5]: lam_h = sp.lambdify(t, h, modules=['numpy'])
        lam_h1s = sp.lambdify(t, h1s, modules=['numpy'])
```

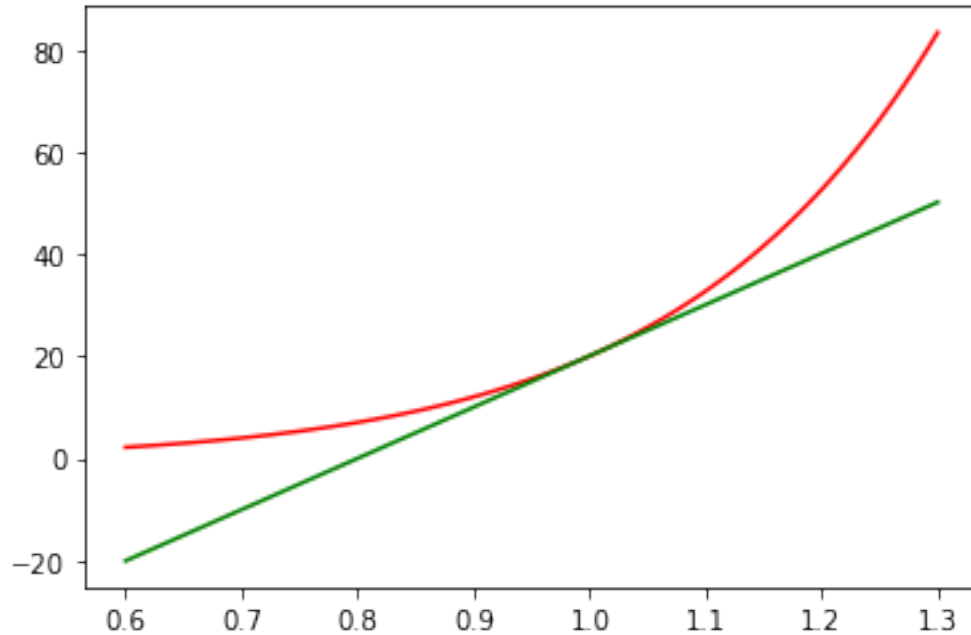
and then plot from samples of  $t$ :

```
In [10]: import numpy as np
         import matplotlib.pyplot as plt
```

```

tv = np.linspace(0.6, 1.3, 100);
h_vals = lam_h(tv); h1s_vals = lam_h1s(tv)
fig, ax = plt.subplots(1,1);
ax.plot(tv, h_vals, 'r-', tv, h1s_vals, 'g-');

```



Higher-order approximations are given by the series

$$h_n(t) = h(t_0) + h'(t_0)(t - t_0) + \frac{1}{2}h''(t_0)(t - t_0)^2 + \frac{1}{6}h'''(t_0)(t - t_0)^3 + \dots \frac{1}{n!}h^{(n)}(t_0)(t - t_0)^n$$

We can write this as the following sum:

$$h_n(t) = \sum_{i=0}^n \frac{1}{i!} h^{(i)}(t_0) (t - t_0)^i$$

An expression for the  $n$ th-order Taylor approximation of a symbolic function  $h$  about a point  $t_0$  can be calculated using the sympy function `series`:

```
hnt = sp.series(h,t,t0,n).removeO
```

The code below plots the first- and fifth-order approximations of  $h(t) = t^2 e^{3t}$

```

In [21]: %run src/labX_preamble.py # For internal notebook functions
         # %%writefileexec src/lab_taylorexp-1.py -s # dump cell to file before execute

```

```

import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Symbolic function and derivatives
h, t = sp.symbols('h t');
h = sp.exp(3*t)*t**2;
hp = h.diff(t);
#hpp = hp.diff(t);

# Taylor expansions around point t0
t0 = 1;
h0 = h.subs(t,t0);
h1 = h0 + hp.subs({t:t0})*(t-t0);
#h2 = h0 + hp.subs(t,t0)*(t-t0) + 1/2*hpp.subs(t,t0)*(t-t0)**2;

# Direct Taylor expansion using sympy
h5s = sp.series(h, t, t0, 6).removeO();
print("Taylor 5: ")
display(h5s)

# Convert symbolic to functions that can be evaluated
lam_h = sp.lambdify(t, h, modules=['numpy']);
lam_h1 = sp.lambdify(t, h1, modules=['numpy']);
lam_h5s = sp.lambdify(t, h5s, modules=['numpy']);

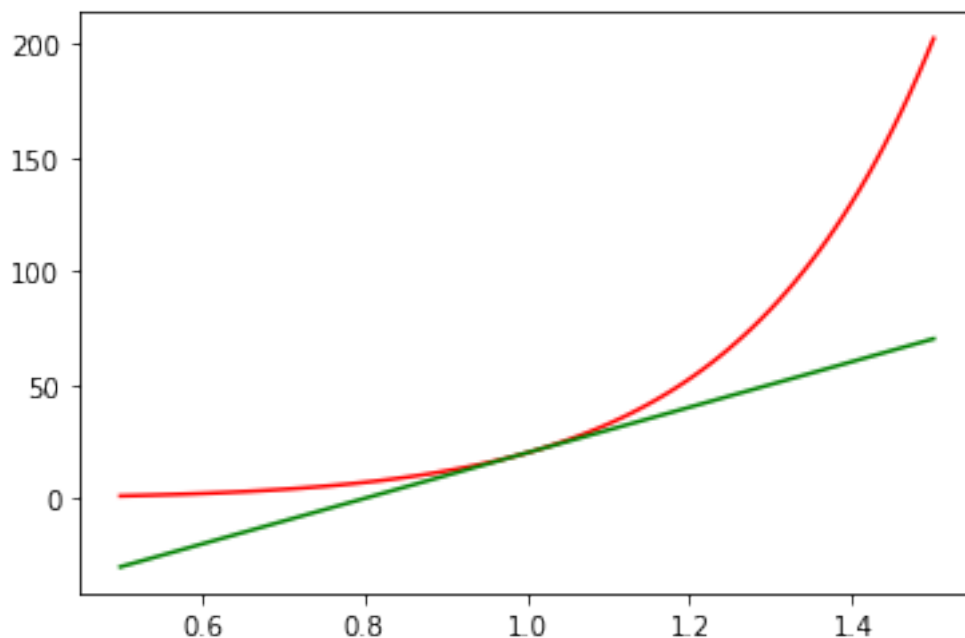
# Plots
fig, ax = plt.subplots(1,1);
t_vals = np.linspace(0.5, 1.5, 100);
ax.plot(t_vals, lam_h(t_vals), 'r');
ax.plot(t_vals, lam_h1(t_vals), 'g');
#ax.plot(t_vals, lam_h5s(t_vals), 'b');

# Symbolic plotting also probably works
#sp.plot(h, h1, (t, 0, 1.5))

```

Taylor 5:

$$\frac{531(t-1)^5 e^3}{40} + \frac{135(t-1)^4 e^3}{8} + \frac{33(t-1)^3 e^3}{2} + \frac{23(t-1)^2 e^3}{2} + 5(t-1)e^3 + e^3$$



## 1.2 Newton's Method for finding roots

Suppose we want to solve the equation

$$\cos(x) = x^3$$

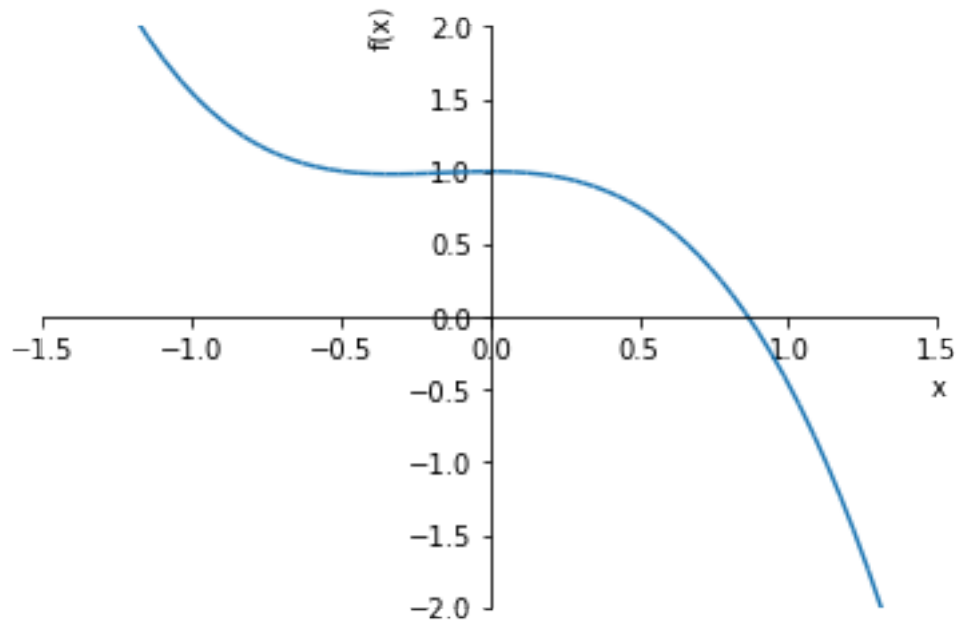
for the variable  $x$ . This is equivalent to finding the root or zero of  $h(x) = \cos(x) - x^3$ .

In general the function  $h(x)$  might be expensive to evaluate, so plotting it and looking for the zero is not feasible. For this demonstration, though, we can make a graph of the function over some finite domain. The cell below defines the equation as a symbolic expression and generates the plot. We observe that the desired solution is in the  $[0.5, 1]$  interval.

```
In [14]: import sympy as sp

x = sp.symbols('x');
h = sp.cos(x) - x**3;
display(h);
sp.plot(h, (x,-1.5,1.5), ylim=(-2,2));
```

$$-x^3 + \cos(x)$$



One might hope to just find the solution using a symbolic solver. However, when I uncomment the code below and run it an error is thrown saying there is no algorithm implemented to solve an equation of this type.

```
In [ ]: #sp.solve(h, x);
```

Only certain categories of functions can be handled by the solver. Our  $h$ , with polynomial and trigonometric parts, doesn't have the required structure.

The function  $h(x)$  is infinitely differentiable and analytic and can be locally approximated around  $x_0$  by a Taylor series expansion.

This is a polynomial in  $x$  and is quite tractable to deal with. In particular, we know how to quite easily find the zeros of low order polynomials.

This suggests an iterative strategy for finding the zero of  $h(x)$ . Start with a guess or initial estimate  $x_0$ . Locally approximate  $h(x)$  around this point using a low-order Taylor expansion, and find the zero for this approximation. We know that the expansion is accurate for values of  $x$  near  $x_0$ , so if the zero is very close to  $x_0$  then it's a good solution and we can terminate. Otherwise we hope that the zero of our approximation is close to the actual zero, move  $x_0$  to this new location, and repeat the process. This is the essence of Newton's method [https://en.wikipedia.org/wiki/Newton's\\_method](https://en.wikipedia.org/wiki/Newton's_method).

If a first-order Taylor approximation is used then given a center  $x_0$  we have

$$h(x) \approx h(x_0) + h'(x_0)(x - x_0)$$

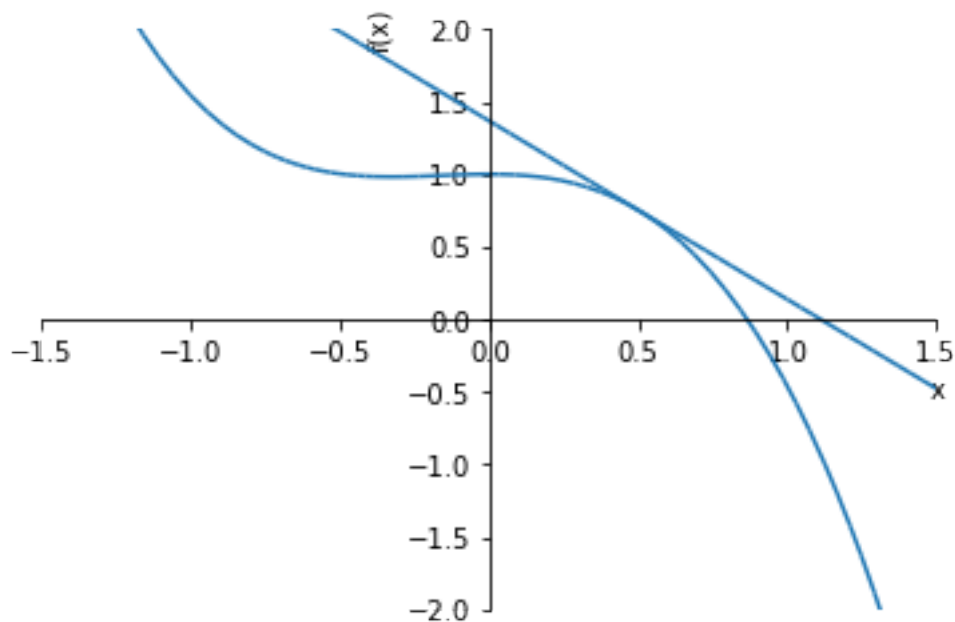
The right-hand side of this approximation is implemented symbolically in the cell below for the given  $h$ .

```
In [15]: hp = sp.diff(h, x);
        x0 = sp.symbols('x0');
        h1 = h.subs(x,x0) + hp.subs({x:x0})*(x-x0); # first-order Taylor at x0
        display(h1);
```

$$-x_0^3 + (x - x_0)(-3x_0^2 - \sin(x_0)) + \cos(x_0)$$

In the code below we plot the function and its first-order Taylor expansion for the case of  $x_0 = 0.5$ .

```
In [16]: x0c = 0.5; # current x0 value
        h1s = h1.subs(x0,x0c);
        sp.plot(h, h1s, (x,-1.5,1.5), ylim=(-2,2));
        display(h1s)
```



$$1.36729533119247 - 1.2294255386042x$$

A linear equation is a special case of a polynomial, and the roots function knows how to find zeros of these. There is only a single zero, which is found below.

```
In [17]: zs = sp.roots(h1s);
        x0n = list(zs.keys())[0];
        x0n
```

Out [17]:

1.11214163709727

This could also just have been calculated numerically directly from the Taylor approximation: if at  $x = x_0$  we know the function value  $h(x_0) = a$  and the derivative  $h'(x_0) = b$  then

$$h(x) \approx h(x_0) + h'(x_0)(x - x_0) = a + b(x - x_0),$$

which has a zero at

$$x = -a/b + x_0 = -\frac{h(x_0)}{h'(x_0)} + x_0.$$

```
In [18]: lam_h = sp.lambdify(x, h, modules=['numpy']);
        lam_hp = sp.lambdify(x, hp, modules=['numpy']);
        x0nn = -lam_h(x0c)/lam_hp(x0c) + x0c;
        x0nn
```

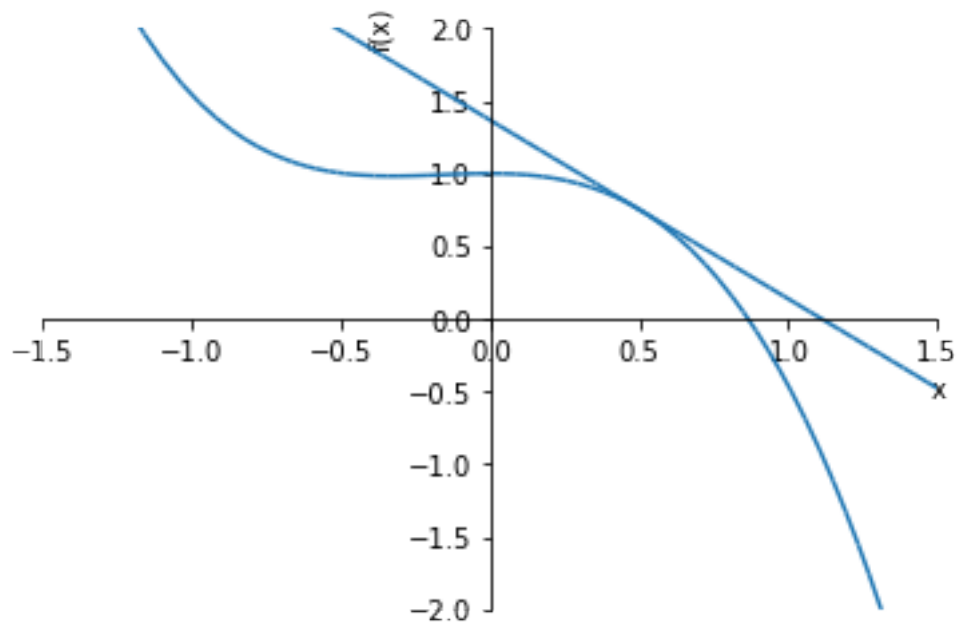
Out [18]: 1.1121416370972725

Thus we see that in order to perform the update step all we need is a method of evaluating  $h(x)$  and  $h'(x)$  for some given  $x_0$ . The lambda functions above obtained from the symbolic expression provide this utility, but if the expressions are known then could also just be coded directly.

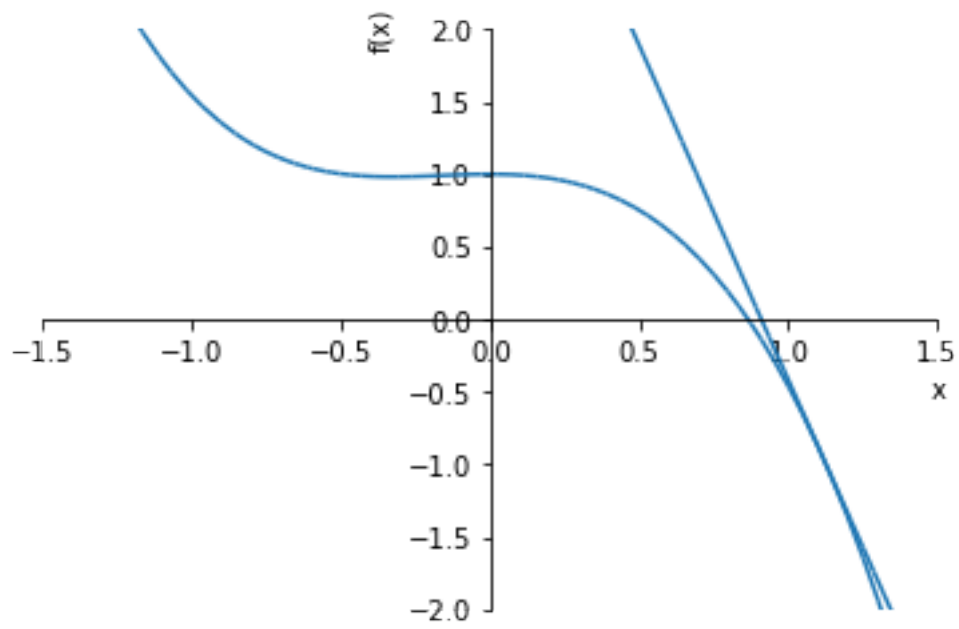
The code below shows a sequence of plots demonstrating Newton's method for finding the root of  $h(x)$  given an initial estimate of  $x_0 = 0.5$ .

```
In [19]: x0c = 0.5; # initial
        for iter in range(0,4):
            h1s = h1.subs(x0,x0c)
            sp.plot(h, h1s, (x,-1.5,1.5), ylim=(-2,2));
            zs = sp.roots(h1s);
            x0c = list(zs.keys())[0];
            print('Crossing: x=%f' % x0c);
```

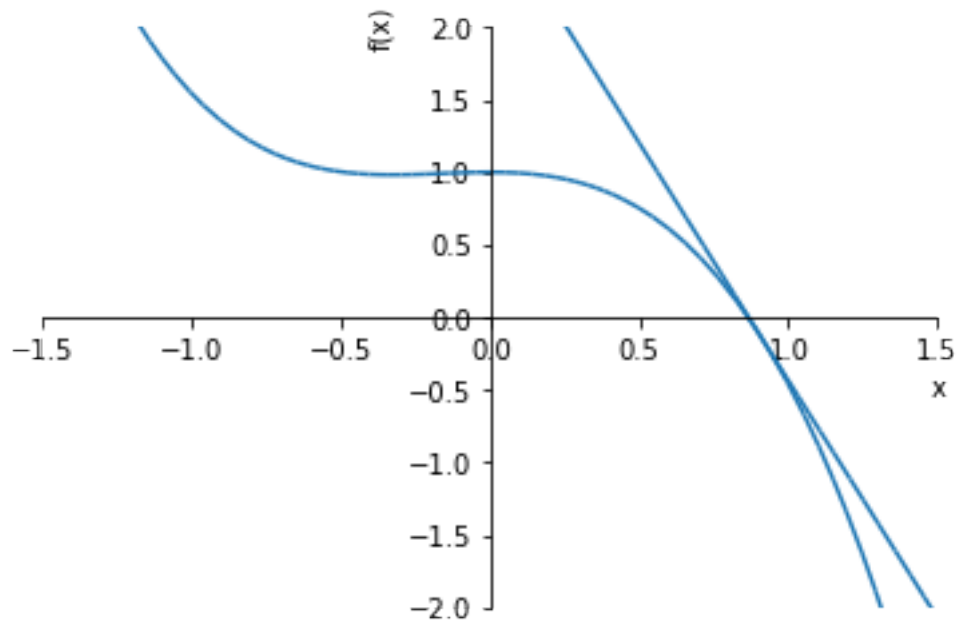




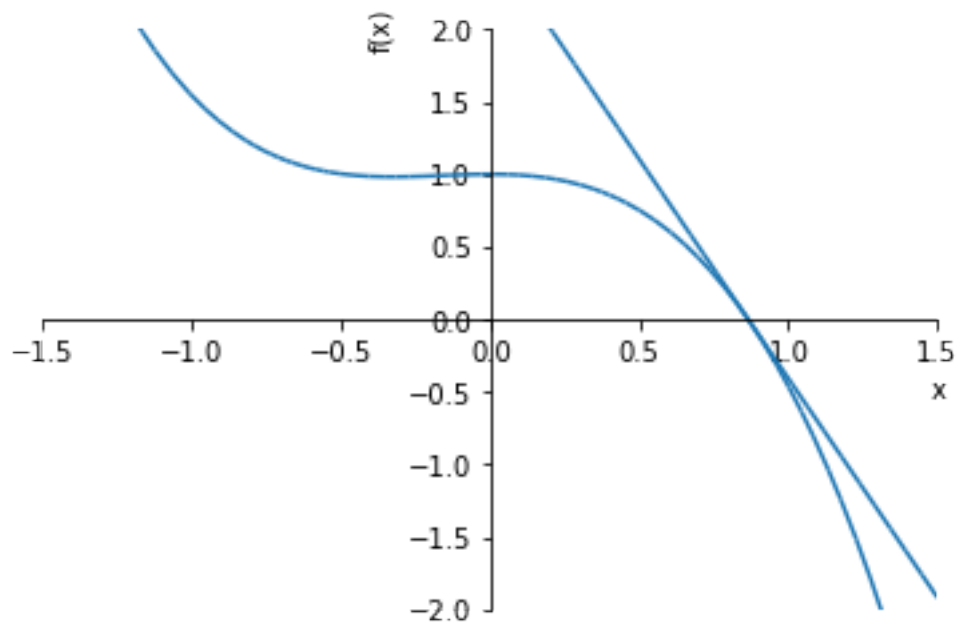
Crossing:  $x=1.112142$



Crossing:  $x=0.909673$



Crossing:  $x=0.867264$



Crossing:  $x=0.865477$

## 2 Tasks

These tasks involve writing code, or modifying existing code, to meet the objectives described.

1. Generate Taylor series expansions of order 5, 9, and 13 centered around the point  $t = 0$  for the function  $x(t) = \sin(t)$ . Plot all of these functions on the same set of axes over the domain  $t = -2\pi$  to  $t = 2\pi$ , with a range from  $-1.5$  to  $1.5$ .
2. Write code to recursively find the solution to  $\cos(x) = x^3$  using Newton's method. Iterate the procedure described until the change in the estimate of the zero location is small enough. Start the process from the initial point  $x_0 = 0.5$ . Easy mode: show five iterations. Challenge: stop when the magnitude of the change in the zero location drops below the threshold  $1e-4$ .
3. The method described uses a very rough first-order approximation to the function, and a large number of low-complexity iterations will need to be done to find the root. You could imagine that it might be better to use a higher-order approximation, and do a smaller number of higher-complexity iterations. Generate a single plot of the function  $h(x)$  and of the second-order Taylor approximation with center point  $x_0 = 0.5$ , and find the roots for the approximation. Developing a numerical method around this approach is not trivial, though, mainly because the quadratic approximation to the function could have two roots (which one do you choose?) or zero roots (what do you do then?).
4. Use Newton's method to find the value of  $\sqrt{13}$ . You should have confidence in your solution to 5 decimal places.

In [ ]: