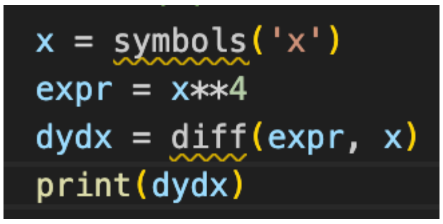
CS7CS2 Week 2 Assignment

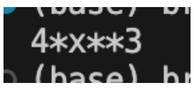
Brendan Jobus 18321740

Part A

(i)



```
x = symbols('x')
expr = x**4
dydx = diff(expr, x)
print(dydx)
```

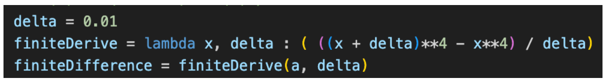


```
4*x**3
```

Screenshot of the code used to acquire the derivative and the output of the code

Using simply I obtained the derivative for x^4 to be $4x^3$. I utilised the sypmy function diff to do this, which takes an expression built with a sympy symbol, and differentiates the given function to respect of whatever symbol it is given, in this case x.

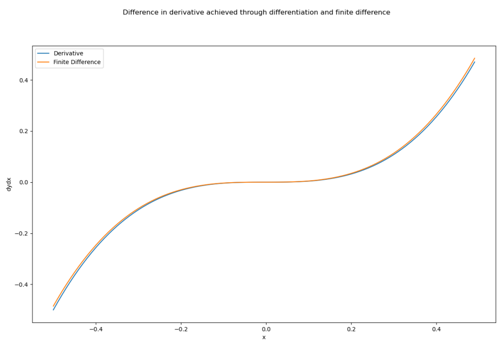
(ii)



```
delta = 0.01
finiteDerive = lambda x, delta : ( ((x + delta)**4 - x**4) / delta)
finiteDifference = finiteDerive(a, delta)
```

Code for finite difference

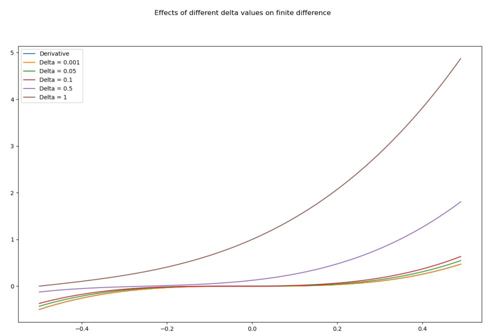
Here, I implement finite difference, we can prove that this is an approximation of

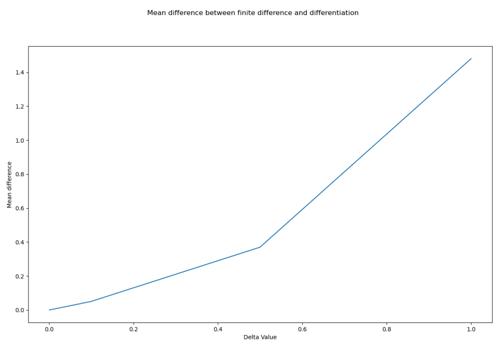


The plot shows how the finite difference and derivative compare over the range of -0.5 ≤ x < 0.5

As can be seen from the plot, the finite difference at delta = 0.01 does quite a good job at approximating the derivative for this function while being substantially faster.

(iii)



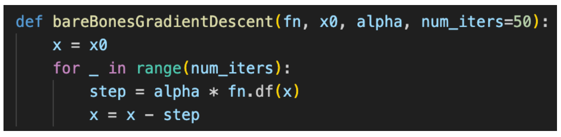


The raw functions on the left and the mean difference between them and the derivative on the right.

As can be seen from the plots, a larger value of delta causes the finite difference to map far worse to the actual derivative.

Part B

(i)

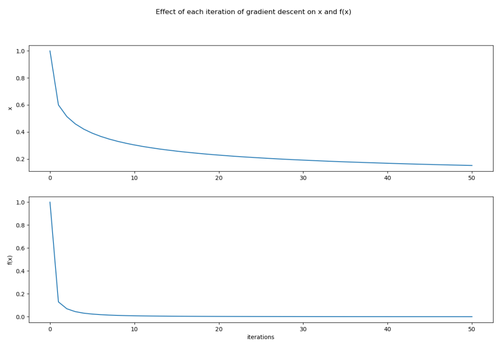


```python
def bareBonesGradientDescent(fn, x0, alpha, num_iters=50):
    x = x0
    for _ in range(num_iters):
        step = alpha * fn.df(x)
        x = x - step
```

Implementation of gradient descent without metadata for graphing

In my actual implementation, I utilised the code from the lecture slides which include some processing of metadata for plotting, I removed them for this section to be clearer for explanation. In the gradient descent algorithm, we simply select some starting point x0, and then attempt to find the minimum value of the function fn. We do this by looking at the derivative of fn at the current point, and since the derivative is the slope, if the minimum is to the left of the current point, the slope will be positive, and thus subtracting it from the current point will get us closer to the minimum value of the function. We use a further variable alpha to control the size of the steps that we take, because, if the function has a very steep change in slope, from very large to very small, using the slope may shoot us past the minimum we are trying to get to, and we would never converge to the minimum.

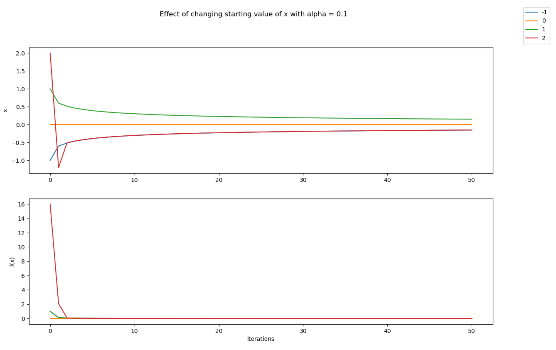
(ii)

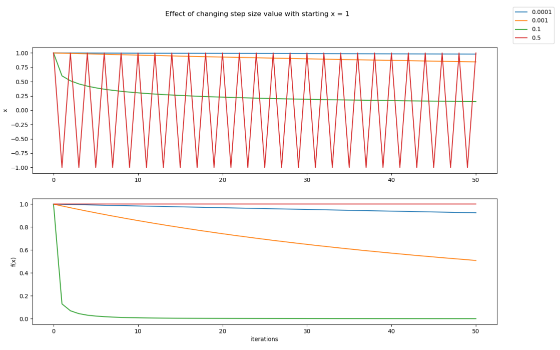


Value of x and f(x) with respect to gradient descent iterations

From the plots above we can see two things, the minimum is clearly at point 0, and also, that after a certain point(roughly x = 0.6), decreasing the value of x has diminishing returns on how close we are to the minimum.

(iii)



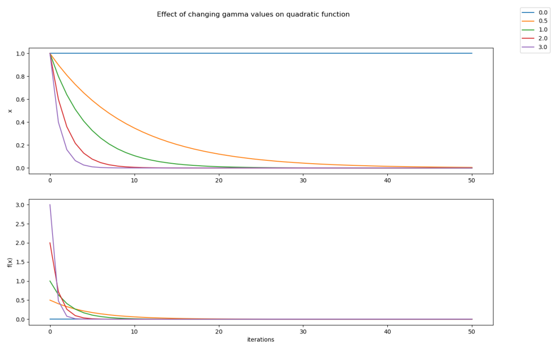


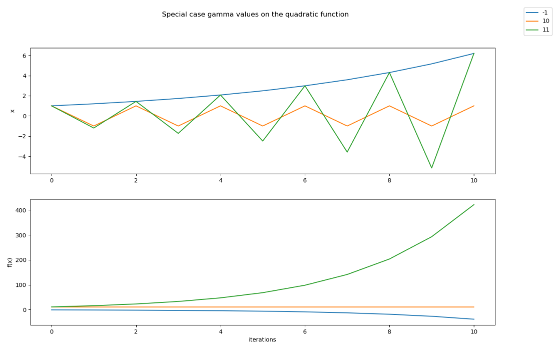
From the graph on the left, we can see that the starting point of x effects how fast we can get to the local minimum in this instance, however, in general can also effect whether we get there at all. All of them made it to the minimum, however, as we can see from when we started at 2, there was a massive jump caused by the slope being so large, and our alpha not being small enough, that we jumped past the local minimum, and have to turn back around. In this case we were lucky enough that the place that we landed had a small enough slope that we stabilised and were able to smoothly converge afterwards, however, it should be obvious from this, that there are cases that if the starting point and alpha are picked poorly, we could stall in an endless loop. What is not shown in these graphs either is that, for this function, I my laptop was not able to handle a starting point of 3 or more, due to the size of the numbers that we start getting, this tells us that the selection of the correct starting point effects how fast we converge, whether we converge, and can cause hardware limitations for depending on the function and hardware.

From the graph on the right, we can see that different values of alpha have massive effects on gradient descent. If our value is too small, they take forever to converge to the local minimum, and if the value is too large, then it won't converge at all, in this case, overshooting the local minimum and being stuck in an endless loop at the same place.

Part C

(i)

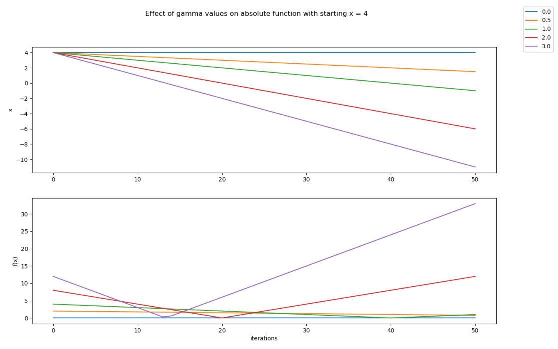




In the case of a quadratic equation, the addition of some scalar will in simply act to scale the slope itself, and thus, with a constant step size of alpha, a larger gamma value causes us to head downhill much more rapidly.

The right plot shows the values of gamma that breaks gradient descent. Firstly, if the value of gamma is negative, then the function doesn't have a minimum value, and since our gradient descent is always looking to go down hill, it will simply minimise forever. Secondly, at gamma = 6, we get the same bouncing as when we set the starting x to 2 in the last graph, and from gamma = 7 to gamma = 10, we never converge as we jump past the local minimum and the mixture of the slope and alpha value causes us to be stuck. From gamma = 11 and on, the slope is so large, that we we continue to get further and further from the minimum. Note, I had to limit the iterations in this plot to 10 so that the gamma of 10 is visible as both gamma = -1 and 11 will explode and make it impossible to see what is happening in gamma = 10.

(ii)


Effect of gamma values on absolute function with starting x = 4
x
f(x)
iterations