Congratulations! You passed!

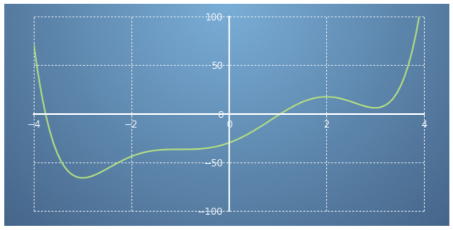
Grade received 80% **To pass** 80% or higher

Go to next item

1. In this quiz we shall explore using the Newton-Raphson method for root finding.

0 / 1 point

Consider the following graph of a function,



There are two places that this function goes through zero, i.e. two roots, one is near $x = -4$ and the other is near $x = 1$.

Recall that if we linearise about a particular point x_0, we can ask what the value of the function is at the point $x_0 + \delta x$, a short distance away.

$$f(x_0 + \delta x) = f(x_0) + f'(x_0)\delta x$$

Then, if we assume that the function goes to zero somewhere nearby, we can re-arrange to find how far away, i.e. assume $f(x_0 + \delta x) = 0$ and solve for δx. This becomes,

$$\delta x = -\frac{f(x_0)}{f'(x_0)}$$

Since the function, $f(x)$ is not a line, this formula will (try) to get closer to the root, but won't exactly hit it. But this is OK, because we can repeat the process from the new starting point to get even closer,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

This is the Newton-Raphson method, and it (or a variant) is used widely to find the roots of functions.

For the graph we showed above, the equation of the function is,

$$f(x) = \frac{x^6}{6} - 3x^4 - \frac{2x^3}{3} + \frac{27x^2}{2} + 18x - 30.$$

We'll explore the Newton-Raphson method for this function in this quiz, when it works, and how it can go wrong.

To start, differentiate the function $f(x)$, as we'll need $f'(x)$ later on.

(Type your answer as you would Python code, i.e with * to multiply and ** to raise to a power. e.g., 4*x**3 - 2*x**2/5)

$x^5 - 12x^3 - 6x^2 + 54x + 18$

x**5-12*x**3-6*x**2+54*x+18

Incorrect

Check your working carefully. Make use of the power rule.

2. We'll first try to find the location of the root near $x = 1$.

1 / 1 point

By using $x_0 = 1$ as a starting point and calculating $-f(1)/f'(1)$ by hand, find the first iteration of the Newton-Raphson method, i.e., find x_1.

Give your answer to 3 decimal places.

1.063

Correct

3. Let's use code to find the other root, near $x = -4$.

1 / 1 point

Complete the d_f function in the code block with your answer to Q1, i.e. with $f'(x)$. The code block will then perform iterations of the Newton-Raphson method.

```
def f (x) :
  return x**6/6 - 3*x**4 - 2*x**3/3 + 27*x**2/2 + 18*x - 30

def d_f (x) :
  return x**5-12*x**3-6*x**2+54*x+18 # Complete this line with the derivative y

x = -4.0

d = {"x" : [x], "f(x)": [f(x)]}
for i in range(0, 20):
  x = x - f(x) / d_f(x)
  d["x"].append(x)
  d["f(x)"].append(f(x))

pd.DataFrame(d, columns=['x', 'f(x)'])
```

Run

Reset

What is the x value of the root near $x = -4$? (to 3 decimal places.)

-3.760

Correct
Observe that the function converges in just a few iterations.

4. Let's explore where things can go wrong with Newton-Raphson.

1 / 1 point

Since the step size is given by $\delta x = -f(x)/f'(x)$, this can get big when f'(x) is very small. In fact $f'(x)$ is exactly zero at turning points of $f(x)$. This is where Newton-Raphson behaves the worst since the step size is infinite.

Use the code block in the previous question for a starting point of $x_0 = 1.99$ and observe what happens.

Select all true statements.

- [x] The method takes over 15 iterations to converge.

Correct
Contrast this to a starting value of -4 or 1 where convergence was very quick.

- [ ] None of the other statements are true.
- [ ] The method diverges to infinity.
- [ ] The method does not converge, instead oscillates without settling.
- [ ] The method converges to the root nearest $x = 1$
- [x] The method converges to the root nearest $x = -4$

Correct
Note that this is not the nearest root to the starting point.

5. Some starting points on the curve do not converge, nor do they diverge, but oscillate without settling. Try $x_0 = 3.1$ as a starting point; it does just this.

1 / 1 point

Again, this is behaviour that happens in areas where the curve is not well described by a straight line - therefore our initial linearisation assumption was not a good one for such a starting point.

Use the code block from previously to observe this.

In practice, often you will not need to hand craft optimisation methods, as they can be called from libraries, such as scipy. Use the code block below to test $x_0 = 3.1$.

```
from scipy import optimize

def f (x) :
  return x**6/6 - 3*x**4 - 2*x**3/3 + 27*x**2/2 + 18*x - 30

x0 = 3.1
optimize.newton(f, x0)
```

Run

Reset

Did it settle to a root?

- ○ No, the method diverged.
- ○ No, the method returned an error.
- ◉ Yes, to the root nearest $x = 1$.
- ○ Yes, to the root nearest $x = -4$.

Correct
Eventually this tricky starting point settles.