Newton-worksheet

October 31, 2023

1 Newton's method

First run the following code to load math functions into your Python environment.

[5]:
```
# importing the required modules
import math
import sympy as sym
import numpy as np
```

Review of basic math operations in Python:

[64]:
```
# addition
3+4
```

[64]: 7

[66]:
```
# subtraction
3-4
```

[66]: -1

[68]:
```
# multiplication
3*4
```

[68]: 12

[70]:
```
# division
3/4
```

[70]: 0.75

[72]:
```
# exponentiation (powers)
3**4
```

[72]: 81

Now we move on to solving the equation $g(x) = 0$

[75]:
```
# tell Python that we want to use x as a variable
x = sym.Symbol('x')

# Define the function g(x)
def g(x):
    return(x**3-2*x+1)

display(g(x))

print(g(0))
print(g(1))
print(g(2))
```

$x^3 - 2x + 1$

```
1
0
5
```

[77]:
```
# Define the derivative (calculate the derivative by hand first)
def gprime(x):
    return(3*(x**2)-2)

display(gprime(x))

print(gprime(0))
print(gprime(1))
print(gprime(2))
```

$3x^2 - 2$

```
-2
1
10
```

Define a function that returns the x-intercept of the linearization to $g(x)$ at $x = a$. We found that the x-intercept occurs at $a - \frac{g(a)}{g'(a)}$.

[79]:
```
def linint(a):
    return(a-(g(a)/gprime(a)))

print(linint(0))
print(linint(0.5))
print(linint(0.6))
```

```
0.5
0.6
0.6173913043478261
```

1.0.1 Problem 1.

Use `linint()` to generate a sequence of better and better guesses at a solution to $g(x) = 0$ (other than the solution $x = 1$). Stop when you think your guess is correct to 5 decimal points.

[81]:
```
guess=linint(0)
print(guess)
guess=linint(guess)
print(guess)
guess=linint(guess)
print(guess)
guess=linint(guess)
print(guess)
guess=linint(guess)
print(guess)
guess=linint(guess)
print(guess)
```

```
0.5
0.6
0.6173913043478261
0.6180330952207308
0.6180339887481617
0.6180339887498949
```

It appears that there's a solution to $x^3 - 2x + 1 = 0$ at about $x = 0.618033$

1.0.2 Problem 2.

Write a loop to generate guesses at solutions to $g(x) = 0$. An example of a `for` loop that you can modify is below. This iterative process is called **Newton's Method**.

[83]:
```
guess = linint(0) # start with our initial guess
for n in range(10):
    print(guess)
    guess = linint(guess) # generate a new guess by plugging the old guess into␣
 ↪linint()
```

```
0.5
0.6
0.6173913043478261
0.6180330952207308
0.6180339887481617
0.6180339887498949
0.6180339887498949
0.6180339887498949
0.6180339887498949
0.6180339887498949
```

1.0.3 Problem 3.

Try starting your loop with different guesses: 1. $a = -2$ 1. $a = -1$ 1. $a = -0.5$ 1. $a = 1$ 1. $a = 1.5$

Look at the graph $y = x^3 - 2x + 1$ in Desmos and explain what's going on. Can you find a starting point that doesn't give a root?

[37]:
```
guess = linint(-2) # start with our initial guess
for n in range(10):
    print(guess)
    guess = linint(guess) # generate a new guess by plugging the old guess into␣
 ↪linint()
```

```
-1.7
-1.623088455772114
-1.6180550397179787
-1.6180339891173305
-1.618033988749895
-1.618033988749895
-1.618033988749895
-1.618033988749895
-1.618033988749895
-1.618033988749895
```

[39]:
```
guess = linint(-1) # start with our initial guess
for n in range(10):
    print(guess)
    guess = linint(guess) # generate a new guess by plugging the old guess into␣
 ↪linint()
```

```
-3.0
-2.2
-1.7808306709265176
-1.6363032029562203
-1.6183045780943337
-1.6180340494407914
-1.618033988749898
-1.618033988749895
-1.618033988749895
-1.618033988749895
```

[41]:
```
guess = linint(-0.5) # start with our initial guess
for n in range(10):
    print(guess)
    guess = linint(guess) # generate a new guess by plugging the old guess into␣
 ↪linint()
```

```
1.0
1.0
```

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

[43]:
```
guess = linint(1) # start with our initial guess
for n in range(10):
    print(guess)
    guess = linint(guess) # generate a new guess by plugging the old guess into␣
 ↪linint()
```

```
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
```

[45]:
```
guess = linint(1.5) # start with our initial guess
for n in range(10):
    print(guess)
    guess = linint(guess) # generate a new guess by plugging the old guess into␣
 ↪linint()
```

```
1.2105263157894737
1.063279586248859
1.0089960378008913
1.0002316806665188
1.0000001608290732
1.0000000000000777
1.0
1.0
1.0
1.0
```

What's going on here? We generate guesses by following tangent lines to their x-intercepts. If the tangent line happens to have an x-intercept at $x = 1$, then we get stuck there because $x = 1$ is a solution to $x^3 - 2x + 1 = 0$. **Starting points matter**.

We still have our three solutions: $x \approx -1.618033988749895$, $x \approx 0.6180339887498949$, and $x = 1$. These are very close to the exact roots of $x = \frac{-1-\sqrt{5}}{2}$ and $x = \frac{-1+\sqrt{5}}{2}$. Python's decimals expansions

for these numbers agree exactly with the solutions we get from Newton's method:

[48]:
```
(-1-5**(0.5))/2 # calculate the first solution
```

[48]:
```
-1.618033988749895
```

[50]:
```
(-1+5**(0.5))/2 # calculate the second solution
```

[50]:
```
0.6180339887498949
```

1.0.4 Problem 4.

Some functions are more sensitive to starting points that others. Experiment with different starting points for the following functions: 1. $g(x) = xe^{-(x^2)/2}$. The code below names $e \approx 2.718281828459045$ so you can use it in your function. 1. $g(x) = x^{1/3}$

One way to do this is to define a functions `g(x)` and `gprime(x)` and then reuse the code you used for the last problem. Look at graphs in Desmos to find a starting guess and to figure out what's going on.

[1]:
```
(-1)**(1/3) # This version of a cube root turns out to not be the one we want
```

[1]:
```
(0.5000000000000001+0.8660254037844386j)
```

[7]:
```
math.cbrt(-1) # This cube root works better
```

[7]:
```
-1.0
```

[23]:
```
def newton(f, fprime, a, n=10): # define a function to streamline the process
    guess = a
    for n in range(n):
        if fprime(guess) == 0:
            break # avoid division by 0
        else:
            guess = (guess-(f(guess)/fprime(guess))) # update the guess␣
 ↪(equivalent to linit() from before)
        print(guess)
    return(guess)

newton(g, gprime, 0) # check that the function works and agrees with what we␣
 ↪did above
```

```
0.5
0.6
0.6173913043478261
0.6180330952207308
0.6180339887481617
0.6180339887498949
0.6180339887498949
```

```
0.6180339887498949
0.6180339887498949
0.6180339887498949
```

[23]:
```
0.6180339887498949
```

[27]:
```
e=math.exp(1)
e

def f(x):
    return(x*e**(-(x**2)/2))

display(f(x))

def fprime(x):
    return((1-x**2)*e**(-(x**2)/2))

display(fprime(x))
```

$2.71828182845905^{-\frac{x^2}{2}}x$

$2.71828182845905^{-\frac{x^2}{2}} \cdot (1 - x^2)$

[29]:
```
print('Starting at 0.2:')
newton(f, fprime, 0.2)

print('Starting at 0.6:')
newton(f, fprime, 0.6)

print('Starting at 0.8:')
newton(f, fprime, 0.8)
```

```
Starting at 0.2:
-0.008333333333333331
5.787438942528311e-07
-1.93864665802828e-19
0.0
0.0
0.0
0.0
0.0
0.0
0.0
Starting at 0.6:
-0.3374999999999999
0.04338520543114088
-8.181693517807925e-05
5.47683467046027e-13
```

```
0.0
0.0
0.0
0.0
0.0
0.0
Starting at 0.8:
-1.4222222222222232
-2.812854766886634
-3.2197981893500427
-3.5635329694508293
-3.8681405320348308
-4.14517812816601
-4.4013300014281045
-4.640901114409191
-4.866868075186778
-5.081396031685296
```

[29]:
```
-5.081396031685296
```

We get convergence to the root at $x = 0$ for values of x close to 0. There's a cutoff for convergence somewhere between $x = 0.6$ and $x = 0.8$. we can investigate this:

[31]:
```
for n in range(10):
    a=0.7+0.001*n
    print('Starting at ', a, ':')
    newton(f, fprime, a)
```

```
Starting at  0.7 :
-0.672549019607843
0.5554521878248875
-0.24783629145005914
0.016219024057858855
-4.267646250295104e-06
7.772543730594911e-17
0.0
0.0
0.0
0.0
Starting at  0.701 :
-0.6772960642864022
0.5740129175611993
-0.2820721933350081
0.024383024271815046
-1.4505108830463465e-05
3.0518478504959826e-15
0.0
0.0
```

```
0.0
0.0
Starting at  0.702 :
-0.6820803160908209
0.5933929784334966
-0.32249978989359207
0.037435467596400374
-5.253622216162129e-05
1.4500284518026083e-13
0.0
0.0
0.0
0.0
Starting at  0.703 :
-0.6869021532609317
0.613641411017375
-0.3706349566616568
0.05902210736522606
-0.00020632872235879646
8.783732150918072e-12
0.0
0.0
0.0
0.0
Starting at  0.704 :
-0.6917619591422406
0.6348111364248292
-0.4284977028532435
0.09637135547111009
-0.0009034335874550797
7.373760915014779e-10
0.0
0.0
0.0
0.0
Starting at  0.705 :
-0.6966601222724785
0.6569593457400126
-0.49883629005001373
0.16524955802979735
-0.0046392237095800315
9.984936171891085e-08
-9.926167350636332e-22
0.0
0.0
0.0
Starting at  0.706 :
-0.7015970364699218
```

```
0.6801479372692201
-0.5854819045590456
0.3053765237308872
-0.03140665514468832
3.100942030145881e-05
-2.9818168604828915e-14
0.0
0.0
0.0
Starting at  0.707 :
-0.7065731009235211
0.7044440085960383
-0.6939320278717646
0.6445207915836461
-0.4579912938727134
0.12156553732647912
-0.0018234648678493448
6.063084835193286e-09
0.0
0.0
Starting at  0.708 :
-0.7115887202848801
0.7299204116256401
-0.8323550896141487
1.877266397847115
2.6209947501969832
3.0675309248322606
3.4322899715410866
3.7506660704045203
4.0376886245749555
4.3015393234429435
Starting at  0.709 :
-0.7166443047621344
0.7566563802293601
-1.0134196338699564
-38.52059401434701
-38.546557822372186
-38.57255458961357
-38.59844564543065
-38.62463772064892
```

The threshold for convergence is between 0.707 and 0.708.

[91]:
```
def h(x):
    return(math.cbrt(x)) # This returns the real-valued cube root (not a␣
 ↪complex root)

def hprime(x):
```

```
    return((1/3)*(1/math.cbrt(x))**2)

newton(h, hprime, 0.01, 10)

#for n in range(10):
#    a=0.1**n
#    print('Starting at ', a, ':')
#    newton(h, hprime, a)
```

```
-0.01999999999999999
0.04
-0.07999999999999993
0.1599999999999999
-0.31999999999999973
0.6399999999999995
-1.2799999999999991
2.5600000000000005
-5.120000000000003
10.24000000000001
```

[91]:
```
10.24000000000001
```

This one never converges. Newton's method fails to find the root at $x = 0$ no matter the starting point.

[45]:
```
### This is tangentially-related, but not really part of the project
def quietnewton(f, fprime, a, n=10): # define a version of newton that doesn't␣
 ↪print intermediate steps
    guess = a
    for n in range(n):
        if fprime(guess) == 0:
            break # avoid division by 0
        else:
            guess = (guess-(f(guess)/fprime(guess))) # update the guess␣
 ↪(equivalent to linit() from before)
#         print(guess)
    return(guess)

# Use Newton's method to calculate cube roots by solving x^3 - a = 0.
def crt(a):
    def f(x):
        return(x**3 - a)
    def fprime(x):
        return(3*(x**2))
    return(quietnewton(f, fprime, a, 100))

print(crt(-1))
```

```
print(crt(-0.5))
print(crt(1/8))
```

```
-1.0
-0.7937005259840997
0.5
```

[234]:
```
def h(x): # use our own cube root function to work with h(x) = x^(1/3)
    return(crt(x))

def hprime(x):
    return((1/3)*crt(x**(-2)))

newton(h, hprime, 0.5, 10)
```

```
-0.9999999999999998
1.9999999999999998
-4.0
8.000000000000002
-16.000000000000007
32.00000000000002
-64.00000000000006
128.00000000000009
-256.0000000000002
512.0000000000006
```

[234]:
```
512.0000000000006
```

1.0.5 Problem 5.

Use Newton's method to finish the last worksheet by solving $\frac{1}{3} - (8-x)\left[36+(8-x)^2\right]^{-\frac{1}{2}} = 0$

[51]:
```
def g(x):
    return(1/3-(8-x)*((36+(8-x)**2)**(-1/2)))

display(g(x))

def gprime(x):
    return(36*(36+(8-x)**2)**(-3/2))

display(gprime(x))

guess=4 # Guess that Elvis should run halfway down the beach
print(guess)
for i in range(10):
    guess=guess-(g(guess)/gprime(guess))
    print(guess)
```

```
print('The exact answer is', 8-3*(2**(-1/2))) # The exact answer
```

$$-\frac{8-x}{\left((8-x)^2+36\right)^{0.5}}+0.333333333333333$$

$$\frac{36}{\left((8-x)^2+36\right)^{1.5}}$$

```
4
6.305765438442085
5.891179239973325
5.878691886229645
5.878679656452109
5.878679656440357
5.878679656440357
5.878679656440357
5.878679656440357
5.878679656440357
5.878679656440357
The exact answer is 5.878679656440357
```

[57]:
```
newton(g, gprime, 4) #should do the same thing as the code above
```

```
6.305765438442085
5.891179239973325
5.878691886229645
5.878679656452109
5.878679656440357
5.878679656440357
5.878679656440357
5.878679656440357
5.878679656440357
5.878679656440357
```

[57]: 5.878679656440357