section_1

February 7, 2025

1 Section 1 - Python code review

Question (i) **(i)** This is a copy of the Python Code Review notebook

[7]:

```
#Code CELL #1
import sympy
def compute_tangent_line(my_func, x0):
  x = sympy.Symbol('x')
  # Convert the lambda function to a symbolic expression
  func_expr = sympy.sympify(my_func(x))
  # Evaluate my_func at the given point
  y0 = my_func(x0)
  # Compute the derivative of my_func symbolically
  m = sympy.diff(func_expr, x).subs(x, x0)
  # Define function object for equation of tangent line
  tangent_line = lambda x_val: m * (x_val - x0) + y0
  return tangent_line
```

Question (ii) **(ii)** We're defining a function 'compute_tangent_line', it's return values and types aren't explicitly declared, but given the naming we can assertain that the first argument is a function that takes and returns a single floating point value, that the second is a point x_0, a value x from which to cast the tangent line.

Reading further we can assertain the return type to be a function, a line, that takes and returns singular floating points

Question (iii) **(iii)** We're modeling the function

$$f(x) = \frac{1}{2}x(x-2)(x+2)$$

And calling 'compute_tangent_line' with $x_0 = -1.5$ and the function defined input

[8]:

```
#Code CELL #2

cubic_function = lambda x : (1/2)*x*(x-2)*(x+2)
tangent_line_func = compute_tangent_line(cubic_function, -1.5)
```

Question (iv) **(iv)** Now were going to evaluate at $x = 10$

[9]:

```
#Code CELL #3

tangent_line_func(10)
```

[9]:

17.125

Question (v) **(v)** This illustrates the use of numpy's linspace to generate a range

[10]:

```
#Code CELL #4
import numpy as np
import matplotlib.pyplot as plt

x_vals = np.linspace(0,2,5)

print(x_vals)
```

```
[0.  0.5 1.  1.5 2. ]
```

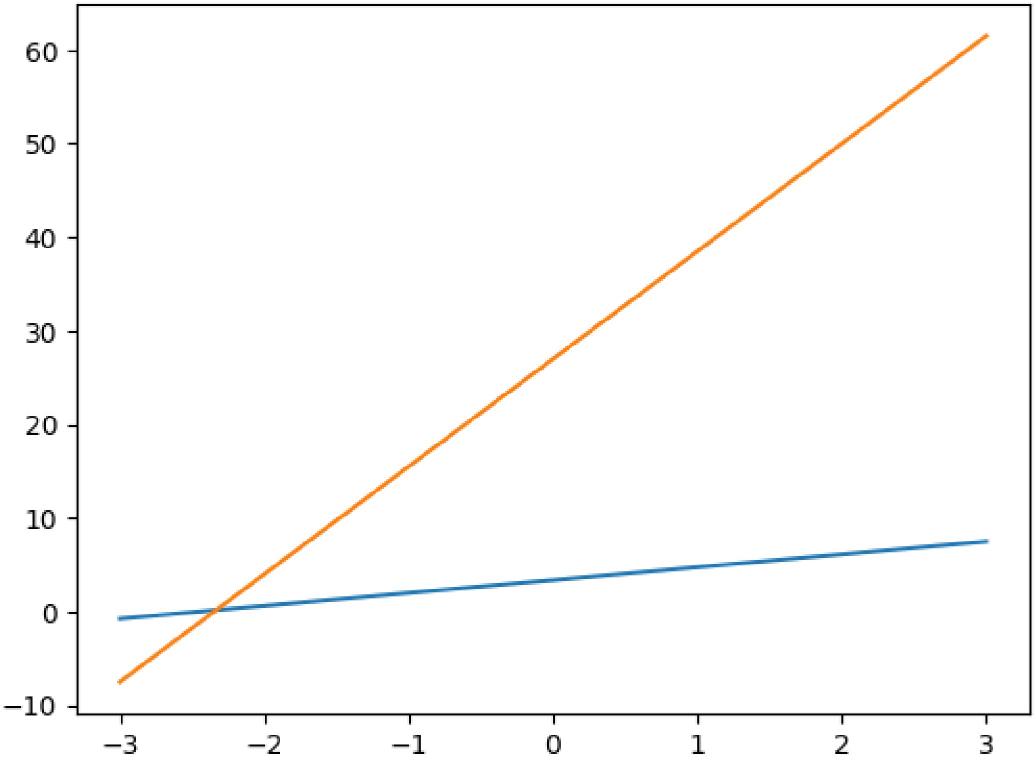
Question (vi) **(vi)** Lastly, we're going to take what's done so far and apply it to a different set of numbers - We're generate a new set of x values, between -3, 3, 100 samples - We generate a line on our plot using the initial intercept of $x = -1.5$ - We then compute a new tangent line from $x = -3$ - Then plot that over the same range of x values

[11]:

```
x_vals = np.linspace(-3, 3, 100)

tan_y_vals = tangent_line_func(x_vals)

plt.plot(x_vals, tan_y_vals)

#Computing the tangent line instead now for -3
tangent_line_func = compute_tangent_line(cubic_function, -3)
plt.plot(x_vals, tangent_line_func(x_vals))
```

[11]:

```
[<matplotlib.lines.Line2D at 0x777ca1fe3c50>]
```


60
50
40
30
20
10
0
−10
−3
−2
−1
0
1
2
3

section_2

February 7, 2025

1 Section 2. Error Analysis Using Plots

This notebook is designed to guide you through plotting error for rootfinding methods.

Please submit your answers to the questions below along with screenshots of the plots you make while working through this.

Let’s continue working with the very simple equation $f(x) = x^2 - 2$ with solution $p = \sqrt{2}$.

Since we know the exact solution to this equation, we can compare the exact answer to the value of the approximation at each iteration and calculate the absolute error:

$$\text{Absolute Error} = |\text{Approximate Solution - Exact Solution}|.$$

The code cell below defines the function we devloped in class to carry out the bisection method with one small change.

[1]:

```python
import numpy as np

def my_bisection_method(my_func,a,b,TOL,exact_root):

  #check if initial point satisfies the initial criteria
  if my_func(a) == 0:
    print("The root is {}".format(a))
    return a

  if my_func(b) == 0:
    print("The root is {}".format(b))
    return b

  if my_func(a)*my_func(b) > 0:
    print("Sign of f(a) and sign of f(b) must be opposite")
    return None

  curr_iter = 1
  error_vec = []

  #Looping Bisection Method
  while (b-a)/2 > TOL:
     print("The current iteration is {}".format(curr_iter))
     print("The current error is = {}".format((b-a)/2))
     p = a + (b-a)/2
     abs_error = np.abs(p-exact_root)
     error_vec.append(abs_error)
     print("The approximation p_k = {}".format(p))
     print("f(p_k) = {}".format(my_func(p)))

     if my_func(p)==0:
       print("The root is {}".format(p))
       return p

     if my_func(a)*my_func(p)>0:
        a = p
     else:
        b = p

     curr_iter = curr_iter + 1
  print("Current error is {}".format((b-a)/2))
  return p, error_vec
```

Question 1: How many inputs does the function have? What are they? How many outputs does the function have? What are they?

- The function has inputs (my_func,a,b,TOL,exact_root), so five total, all required
  - The function we're trying to find a root for
  - The lower bound
  - The upper bound
  - The tolerance we're looking to get the root within
  - The actual root we're comparing against
- The function prototype isn't explicitly declared, but, analyzing the return statement 'return p, error_vec' we can determine that we have two outputs
  - The independent value of the root we've found
  - A list of error values (a vector)

Next I will show you how to call a function with two outputs.

[2]:

```
import numpy as np

my_func = lambda x: x**2-2
a = 0
b = 2
tol = .001
exact_root = np.sqrt(2)

approx_root, error_vec = my_bisection_method(my_func,a,b,tol,exact_root)

error_vec_length = len(error_vec)
```

```
print("The length of the error vector is {}".format(error_vec_length))
```

```
The current iteration is 1
The current error is = 1.0
The approximation p_k = 1.0
f(p_k) = -1.0
The current iteration is 2
The current error is = 0.5
The approximation p_k = 1.5
f(p_k) = 0.25
The current iteration is 3
The current error is = 0.25
The approximation p_k = 1.25
f(p_k) = -0.4375
The current iteration is 4
The current error is = 0.125
The approximation p_k = 1.375
f(p_k) = -0.109375
The current iteration is 5
The current error is = 0.0625
The approximation p_k = 1.4375
f(p_k) = 0.06640625
The current iteration is 6
The current error is = 0.03125
The approximation p_k = 1.40625
f(p_k) = -0.0224609375
The current iteration is 7
The current error is = 0.015625
The approximation p_k = 1.421875
f(p_k) = 0.021728515625
The current iteration is 8
The current error is = 0.0078125
The approximation p_k = 1.4140625
f(p_k) = -0.00042724609375
The current iteration is 9
The current error is = 0.00390625
The approximation p_k = 1.41796875
f(p_k) = 0.0106353759765625
The current iteration is 10
The current error is = 0.001953125
The approximation p_k = 1.416015625
f(p_k) = 0.005100250244140625
Current error is 0.0009765625
The length of the error vector is 10
```

Question 2: Is the number of iterations consistent with the theory?

Next we will make plots of the absolute error on the vertical axis and the iteration number on the horizontal axis.

Run the code cell below to redefine the bisection method function so it carries out more iterations than needed. This way our plots will look better.

[3]:

```
import numpy as np
#this code block defines the function called my_bisection_method
#INPUTS: function my_func, a, b, TOL, exact root
#OUTPUTS: root or a error message
def my_bisection_method(my_func,a,b,TOL,exact_root,max_iter):
  if my_func(a) == 0:
    print("The root is {}".format(a))
    return a#this end the function
  if my_func(b) == 0:
    print("The root is {}".format(b))
    return b#this end the function
  if my_func(a)*my_func(b) > 0:
    print("Sign of f(a) and sign of f(b) must be opposite")
    return None#this end the function
  curr_iter = 1
  error_vec = []
  for i in range(max_iter):#current interval divided by two bigger than tol
    print("The current iteration is {}".format(curr_iter))
    print("The current error is = {}".format((b-a)/2))
    p = a + (b-a)/2
    #compute |approximate root - exact root|
    abs_error = np.abs(p-exact_root)
    error_vec.append(abs_error)
    print("The approximation p_k = {}".format(p))
    print("f(p_k) = {}".format(my_func(p)))
    if my_func(p)==0:
      print("The root is {}".format(p))
      return p#this end the function
    if my_func(a)*my_func(p)>0:
      a = p
    else:
      b = p
    curr_iter = curr_iter + 1
  print("Current error is {}".format((b-a)/2))
  return p, error_vec
```

[4]:

```
import numpy as np
#define the inputs
my_func = lambda x: x**2-2
a = 0
b = 2
tol = .001
exact_root = np.sqrt(2)
```

```
max_iter = 30
#call the function
approx_root, error_vec =␣
 ↪my_bisection_method(my_func,a,b,tol,exact_root,max_iter)
```

```
The current iteration is 1
The current error is = 1.0
The approximation p_k = 1.0
f(p_k) = -1.0
The current iteration is 2
The current error is = 0.5
The approximation p_k = 1.5
f(p_k) = 0.25
The current iteration is 3
The current error is = 0.25
The approximation p_k = 1.25
f(p_k) = -0.4375
The current iteration is 4
The current error is = 0.125
The approximation p_k = 1.375
f(p_k) = -0.109375
The current iteration is 5
The current error is = 0.0625
The approximation p_k = 1.4375
f(p_k) = 0.06640625
The current iteration is 6
The current error is = 0.03125
The approximation p_k = 1.40625
f(p_k) = -0.0224609375
The current iteration is 7
The current error is = 0.015625
The approximation p_k = 1.421875
f(p_k) = 0.021728515625
The current iteration is 8
The current error is = 0.0078125
The approximation p_k = 1.4140625
f(p_k) = -0.00042724609375
The current iteration is 9
The current error is = 0.00390625
The approximation p_k = 1.41796875
f(p_k) = 0.0106353759765625
The current iteration is 10
The current error is = 0.001953125
The approximation p_k = 1.416015625
f(p_k) = 0.005100250244140625
The current iteration is 11
The current error is = 0.0009765625
The approximation p_k = 1.4150390625
```

```
f(p_k) = 0.0023355484008789062
The current iteration is 12
The current error is = 0.00048828125
The approximation p_k = 1.41455078125
f(p_k) = 0.0009539127349853516
The current iteration is 13
The current error is = 0.000244140625
The approximation p_k = 1.414306640625
f(p_k) = 0.0002632737159729004
The current iteration is 14
The current error is = 0.0001220703125
The approximation p_k = 1.4141845703125
f(p_k) = -8.200109004974365e-05
The current iteration is 15
The current error is = 6.103515625e-05
The approximation p_k = 1.41424560546875
f(p_k) = 9.063258767127991e-05
The current iteration is 16
The current error is = 3.0517578125e-05
The approximation p_k = 1.414215087890625
f(p_k) = 4.314817488193512e-06
The current iteration is 17
The current error is = 1.52587890625e-05
The approximation p_k = 1.4141998291015625
f(p_k) = -3.8843369111418724e-05
The current iteration is 18
The current error is = 7.62939453125e-06
The approximation p_k = 1.4142074584960938
f(p_k) = -1.726433401927352e-05
The current iteration is 19
The current error is = 3.814697265625e-06
The approximation p_k = 1.4142112731933594
f(p_k) = -6.474772817455232e-06
The current iteration is 20
The current error is = 1.9073486328125e-06
The approximation p_k = 1.4142131805419922
f(p_k) = -1.0799813026096672e-06
The current iteration is 21
The current error is = 9.5367431640625e-07
The approximation p_k = 1.4142141342163086
f(p_k) = 1.6174171832972206e-06
The current iteration is 22
The current error is = 4.76837158203125e-07
The approximation p_k = 1.4142136573791504
f(p_k) = 2.687177129701013e-07
The current iteration is 23
The current error is = 2.384185791015625e-07
The approximation p_k = 1.4142134189605713
```

```
f(p_k) = -4.056318516632018e-07
The current iteration is 24
The current error is = 1.1920928955078125e-07
The approximation p_k = 1.4142135381698608
f(p_k) = -6.845708355740499e-08
The current iteration is 25
The current error is = 5.960464477539063e-08
The approximation p_k = 1.4142135977745056
f(p_k) = 1.0013031115363447e-07
The current iteration is 26
The current error is = 2.9802322387695312e-08
The approximation p_k = 1.4142135679721832
f(p_k) = 1.583661290993632e-08
The current iteration is 27
The current error is = 1.4901161193847656e-08
The approximation p_k = 1.414213553071022
f(p_k) = -2.6310235545778937e-08
The current iteration is 28
The current error is = 7.450580596923828e-09
The approximation p_k = 1.4142135605216026
f(p_k) = -5.236811428943611e-09
The current iteration is 29
The current error is = 3.725290298461914e-09
The approximation p_k = 1.414213564246893
f(p_k) = 5.29990096254096e-09
The current iteration is 30
The current error is = 1.862645149230957e-09
The approximation p_k = 1.4142135623842478
f(p_k) = 3.154454475406965e-11
Current error is 9.313225746154785e-10
```

[5]:

```
# next we will plot the absolute error on the vertical axis and the iteration␣
 ↪number on the horizontal axis
import matplotlib.pyplot as plt
# "formula" for making a scatterplot
# plt.plot(x_vals, y_vals)
# in this case the x_vals should be the set of iterations x_vals =␣
 ↪[1,2,3,4,5,6,7,8,9,10]
# there are many ways to make collections of numbers like this in Pyhton
# we will keep using numpy
x_vals = np.arange(1,11)
print(x_vals)
# notice it leaves off the last number
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

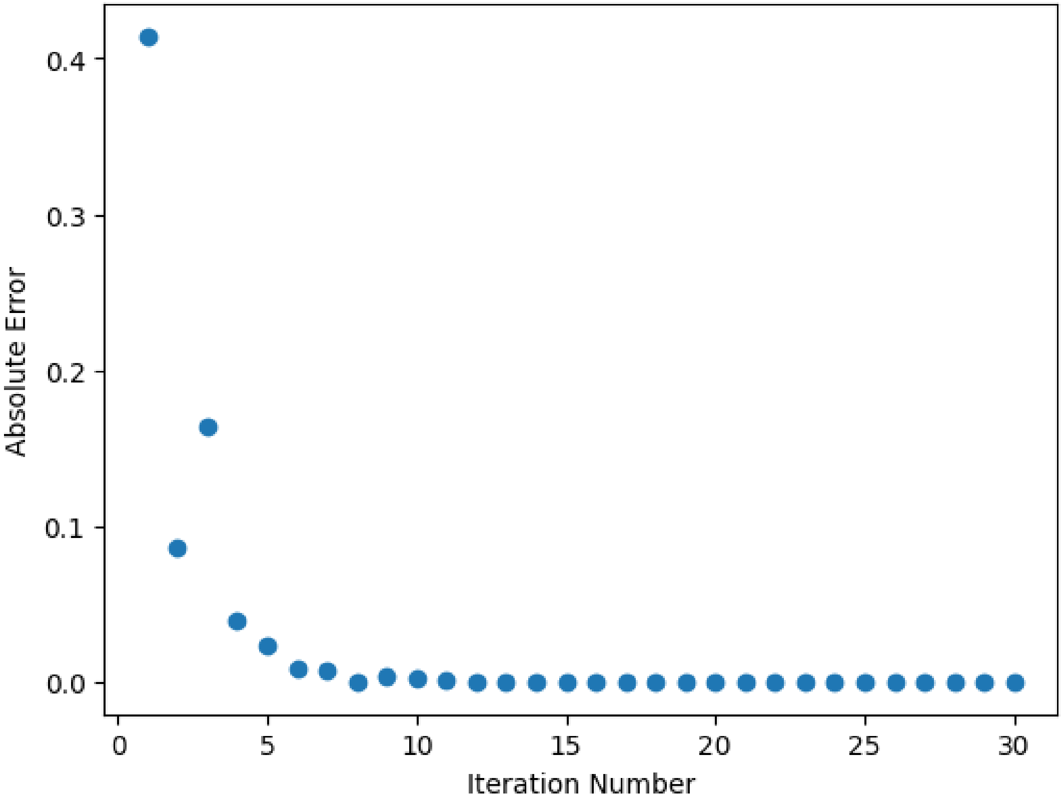
[6]:

```
# the best way to set up this vector is below
```

```
x_vals = np.arange(1,len(error_vec)+1)#the len(error_vec) function returns the␣
 ↪# of things in the error_vec array
print(x_vals)
# now we make the plot
plt.scatter(x_vals, error_vec)
# add some titles so we know what we are looking at
plt.xlabel("Iteration Number")
plt.ylabel("Absolute Error")
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30]
```

[6]:
```
Text(0, 0.5, 'Absolute Error')
```



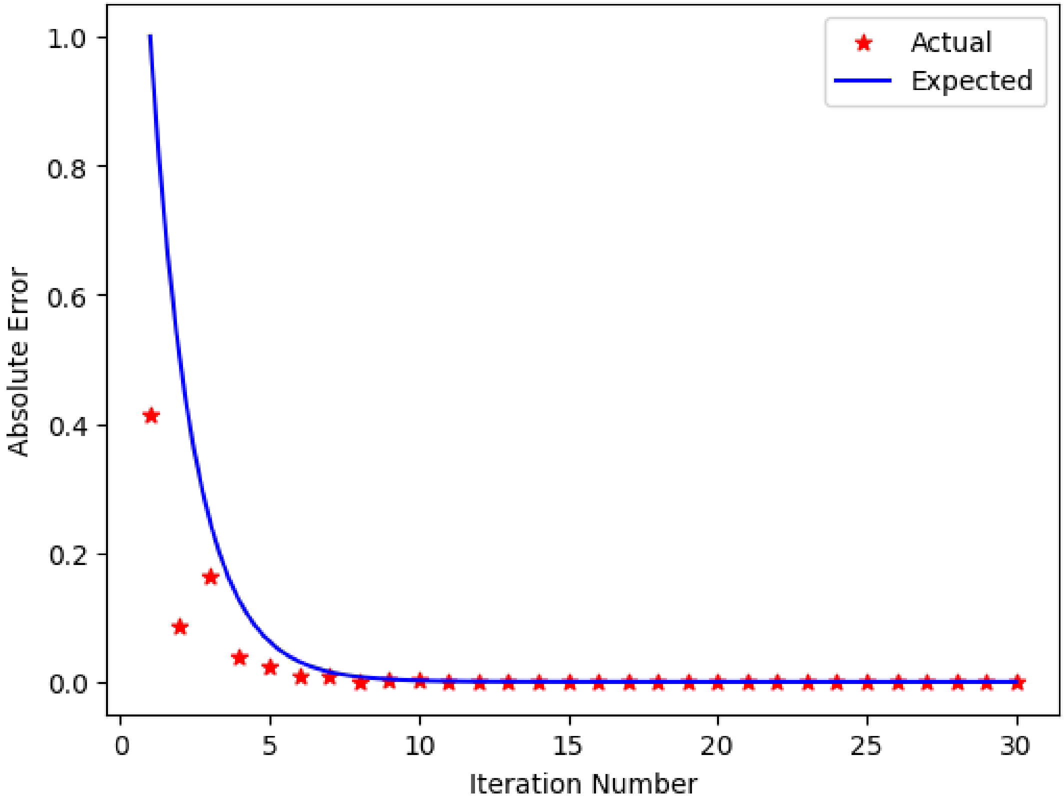
As expected, the absolute error follows an exponentially decreasing trend. Namely, $\frac{b-a}{2^n}$.

Notice that it isn't a completely smooth curve since we will have some jumps in the accuracy just due to the fact that sometimes the root will be near the midpoint of the interval and sometimes it won't be.

[7]:
```
# we can add the theoretical curve to our plot
a = 0
```

```
b = 2
error_func = lambda n: (b-a)/(2**n)
# now we make the plot with the error vals we computed vs. this function
plt.scatter(x_vals, error_vec,label = "Actual", marker = '*', color = 'r')
# I added a label so we could make a legend, we can also specify the marker␣
  ↪type and color
smooth_x_vals = np.linspace(1,max_iter,100)
plt.plot(smooth_x_vals, error_func(smooth_x_vals), label = "Expected", color =␣
  ↪'b')
plt.xlabel("Iteration Number")
plt.ylabel("Absolute Error")
# this uses the labels I defined in the plot call above
plt.legend()
```

```
[7]: <matplotlib.legend.Legend at 0x1f7411ad190>
```



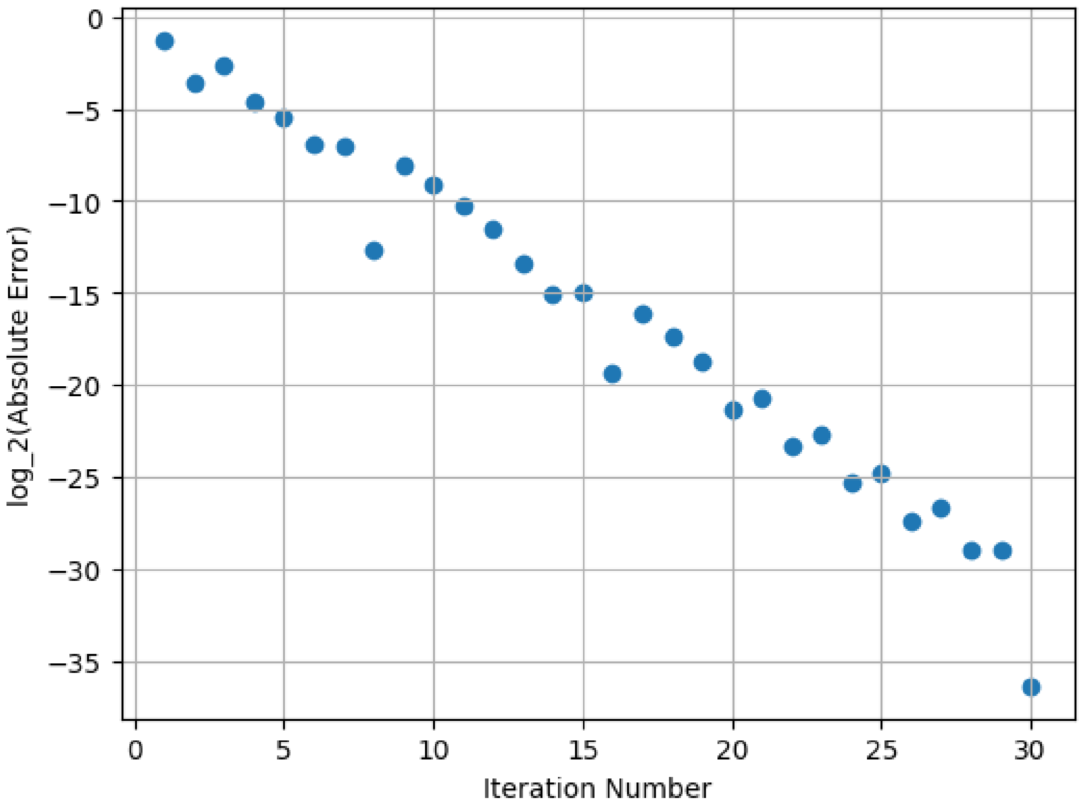
Without the theory, it would be really hard to determine what the exact behavior is in the exponential plot above. We know that the error will divide by 2 at every step, so if we instead plot

$$\log_2(errorvec)$$

against the iteration number we should see a linear trend.

[8]:
```python
plt.scatter(x_vals, np.log2(error_vec))
# Add grid lines
plt.grid(True)
plt.xlabel("Iteration Number")
plt.ylabel("log_2(Absolute Error)")
```

[8]:
```
Text(0, 0.5, 'log_2(Absolute Error)')
```



Question 3: What is the slope of the line in the plot above (estimate it)? Why does that slope make sense?

Another plot we can use to determine how an algorithm is behaving as it progresses is to compare the absolute error at iteration k to the absolute error at iteration $k + 1$.

Run the code below to print the lenght of the *error_vec*.

[9]:
```python
print(len(error_vec))
```

```
30
```

[10]:
```
# let's learn how to pull out values at certain indices of a vector like␣
  ↪error_vec
error_at_k = error_vec[0:len(error_vec)-1]#this pulls out the first 29 values
# Note that: 1) we start at 0 not 1, so the indices of the vector go from 0-29
# in python this code will go up to but not include the value in the 29th and␣
  ↪last index
print(len(error_at_k))
```

```
29
```

Question 4: How would you pull out the first five values?

[11]:
```
error_first_5_values = error_vec[0:5]
print(len(error_first_5_values))
```

```
5
```

Question 5: Make a vector called *error_at_k_plus_1* with the last 29 values of *error_vec*.

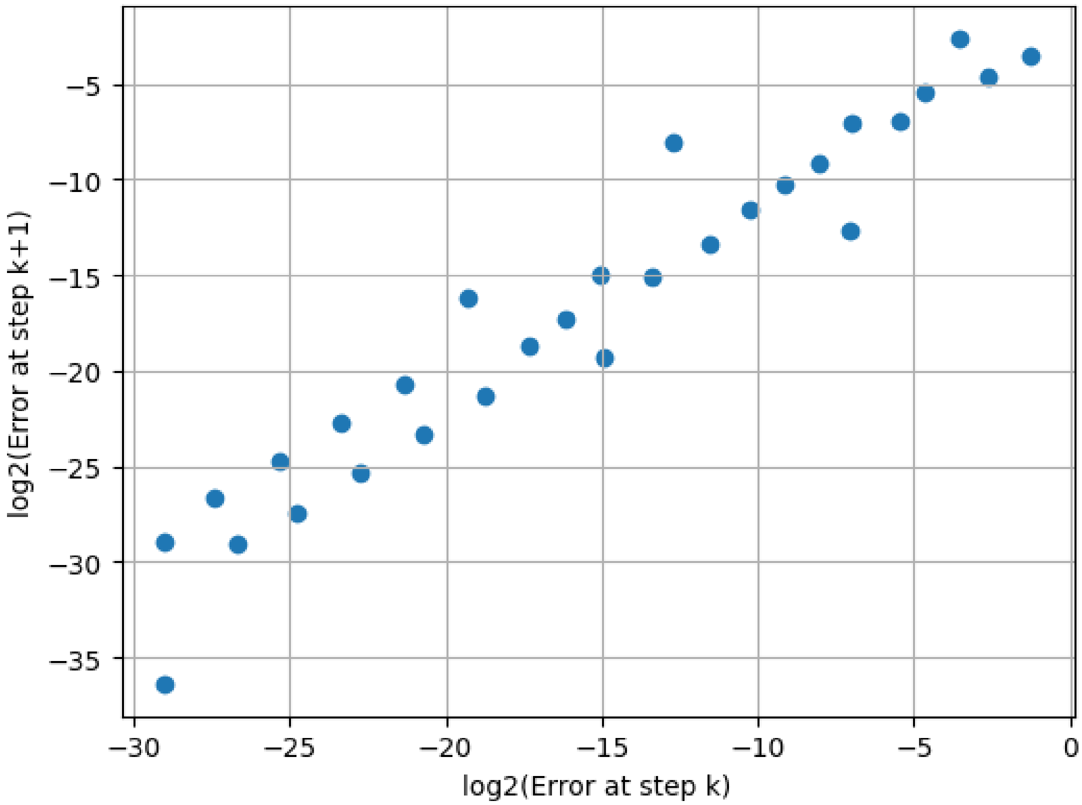
[12]:
```
error_at_k_plus_1 = error_vec[1:len(error_vec)]
print(len(error_at_k_plus_1))
```

```
29
```

Question 6: Which values does the line above pull out?

[13]:
```
# write code to plot error_at_k_plus_1 vs. error_at_k
plt.scatter(np.log2(error_at_k),np.log2(error_at_k_plus_1))
# Add grid lines
plt.grid(True)
plt.xlabel("log2(Error at step k)")
plt.ylabel("log2(Error at step k+1)")
```

[13]:
```
Text(0, 0.5, 'log2(Error at step k+1)')
```



Question 7 Interpret the graph above.

The graph is displaying linear behavoir, a straight line, let's break down the details. - We can see along the left side that our dependent variable is $log_2(x)$ of the step ahead - We can see along the bottom side that our independent variable is $log_2(x)$ of the current step - We're taking the error at both - Taken all together, we can see that the subsequent step is twice as accurate, half as inaccurate - We're demonstrating here the behavoir of the bisection method, we're throwing out half the search space each time, so, we're converging at $O(log_2(n))$

section_3

February 7, 2025

1 Section 3. Bisection method

[1]:

```python
from typing import Callable
from math import sin, log, exp

def bisection_method(
        f: Callable[[float], float],
        a : float,
        b : float,
        tolerance : float) -> float:

    if a == b and (not abs(f(a)) < tolerance):
        return None

    a, b = (a, b) if a < b else (b, a)

    guesses : list[tuple[float, float]] = []

    guess = bisect(a, b)
    while abs(f(guess)) > tolerance:
        guesses.append((guess, f(guess)))
        if f(guess) > 0:
            b = guess
        elif f(guess) < 0:
            a = guess
        guess = bisect(a, b)


    return guesses


def bisect(a : float, b : float) -> float:
    return a + ((b - a) / 2)
```

Question (i) **(i)** Find a bound for the number of iteration needed to achieve an approximation with accuracy 10^{-3} to the solution of $x^3 + x - 4 = 0$ on the interval $[1, 4]$. Find an approximation to the root with this degree of accuracy

[6]:

```
f_of_x = lambda x : x ** 3 + x - 4
root = bisection_method(f_of_x, 1, 4, 10 ** -3)[-1][0]

print(f"Root found at {root}, valued: {f_of_x(root)}")

print("Checking we're within tolerance...")

if f_of_x(root) > 10 ** -3:
    print("We're within tolerance")
else:
    print("We're outside of tolerance")
```

```
Root found at 1.37939453125, valued: 0.004008884658105671
Checking we're within tolerance…
We're within tolerance
```

We've found a root for the function at (1.37939453125, 0.004008884658105671)

section_4

February 7, 2025

1 Section 4. Existence and uniqueness of roots

1.1 Definintions

1.1.1 Intermediate value theorem

Let f be a continuous function defined on $[a, b]$, and let s be a number such that $f(a) < s < f(b)$. Then there exists some x between a and b such that \$f(x)=s > https://en.wikipedia.org/wiki/Intermediate_value_theorem

1.1.2 Rolle's theorem

Suppose $f \in C[a, b]$ and f is differentiable on (a, b). If $f(a) = f(b)$ then $\exists c \in (a, b)$ such that $f'(c) = 0 >$ https://drive.google.com/drive/folders/1UU0Re7ZDqMQLyX4_QSW5pb9W_3gnIHsp

1.1.3 Solving $expr_1(x) = expr_2(x)$

Rearrange so that $f(x) = expr_1(x) - expr_2(x) >$https://drive.google.com/drive/folders/1UU0Re7ZDqMQLyX4_Q

2 Work

(a) Consider the polynomial $f(x) = x^5 + x - 1$. Use IVT to show f has atleast one real root on $(0, 1)$ $f(0) = 0^5 + 0 - 1 = -1 < 0$

$f(1) = 1^5 + 1 - 1 = 1 > 0$

$f(0) * f(1) = (-1) * (1) = -1 < 0$

We can be sure that f crosses the axis because of the change in sign of its values

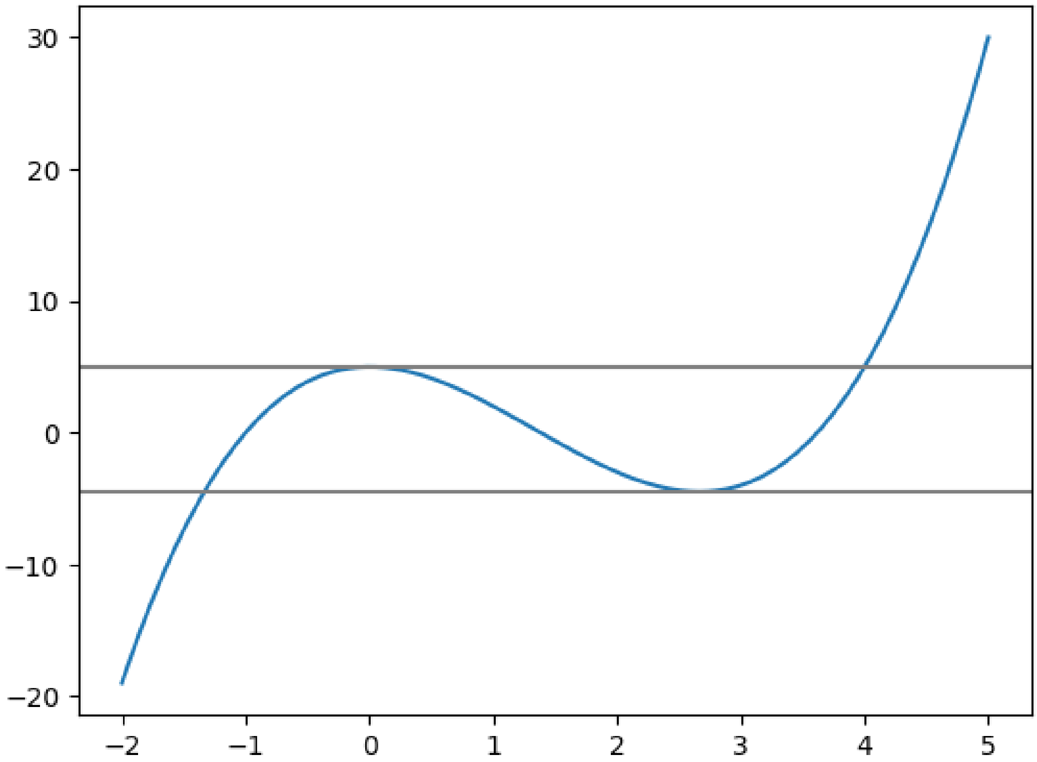
(b) Explain how you would use Rolle's Theorem to show f has at most one real root on all of $\mathbb{R}$

[16]:

```
import matplotlib.pyplot as plt
from numpy import linspace

f = lambda x : x**3 -4*(x**2) + 5
plt.plot(linspace(-2, 5, 100), f(linspace(-2, 5, 100)))
plt.axhline(y=5, color="gray")
plt.axhline(y=-4.48, color="gray")
```

[16]:

```
<matplotlib.lines.Line2D at 0x1238ce930>
```



Let's take a look at the plot. Rolle's theorem here works much in the same way as the Intermediate Value Theorem, but with the derivative. For the graph to have passed the x-axis multiple times, the derivative of the function must be 0 at some point, because if $f(x)$ were only strictly increasing or decreasing, then we'd only go past the x-axis at one point. Put another way

For there to exist points $a, b \in \mathbb{R}$ such that $a \neq b$ and $f(a) = f(b) = 0$, $f'(x)$ must be 0 at some point for the graph to return to cross the x-axis again

(c) Develop a general theory for how the number of solutions to $f'(x) = 0$ can be used to determine an upper bound for the number of roots of f

Examining Firstly, let's just disreguard the case where we have an infinite number of roots along a interval, if $[a, b] \in \mathbb{R}$. If $f'(x) = 0$ and $f(x) = 0$ for $x \in [a, b]$ where $a \neq b$. Then because of the properties of real numbers, we have an infinite number of roots in the interval.

We can do nothing with Rolle's to guarantee that there is a root, but, if $f'(x) \neq 0$ for $x \in [a, b]$ we can't return to cross the x-axis in the other direction, so, let's state this succinctly

Stating There are at most $n + 1$ roots for $x \in [a, b]$ for the n times $f'(x) = 0$

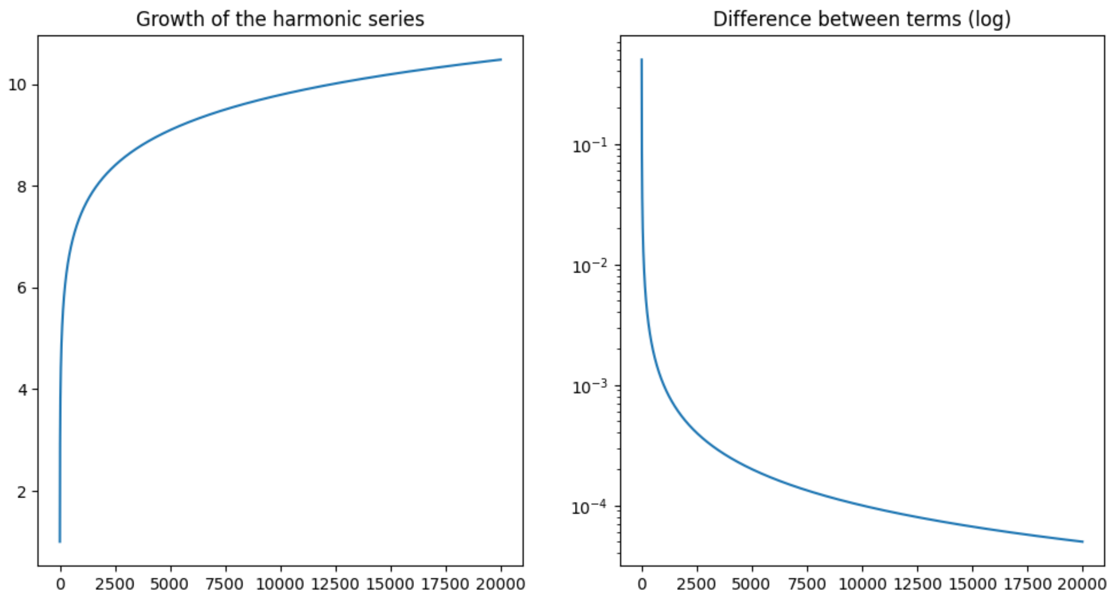
section_5

February 7, 2025

(i) Let p_n be the sequence defined by $p_n = \sum_{k=1}^{n} \frac{1}{k}$.

Show that p_n diverges even though $\lim_{n\to\infty}(p_n - p_{n-1}) = 0$

[7]:
```
import numpy as np
import matplotlib.pyplot as plt

def harmonic_series(n):
    return np.sum(1 / np.arange(1, n + 1))

n_vals = range(1, 20000)
p_n = [harmonic_series(x) for x in range(1, 20000)]
p_n_diffs = np.diff(p_n)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(n_vals, p_n)
plt.title("Growth of the harmonic series")

plt.subplot(1, 2, 2)
plt.plot(n_vals[1:], p_n_diffs)
plt.title("Difference between terms (log)")
plt.yscale('log')
```



Well, numerically this doesn't seem to be converging, though the difference between terms is decreasing, given 20,000 iterations, we'd expect to see something more

Proof by contradiction, the harmonic series diverges *Let* $H = \sum_{k=1}^{n} \frac{1}{k}$

Suppose p_n converges to a value, then

$H = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \ldots$

$H \geq 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{6} + \frac{1}{6} + \ldots$

Now grouping terms together

$H \geq 1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4}$

And substituting from above

$H \geq \frac{1}{2} + H$

We arrive at a contradiction, therefor, H diverges

https://web.williams.edu/Mathematics/lg5/harmonic.pdf

(ii) Let $f(x) = (x-1)^{10}$, $p = 1$, **and** $p_n = 1 + \frac{1}{n}$

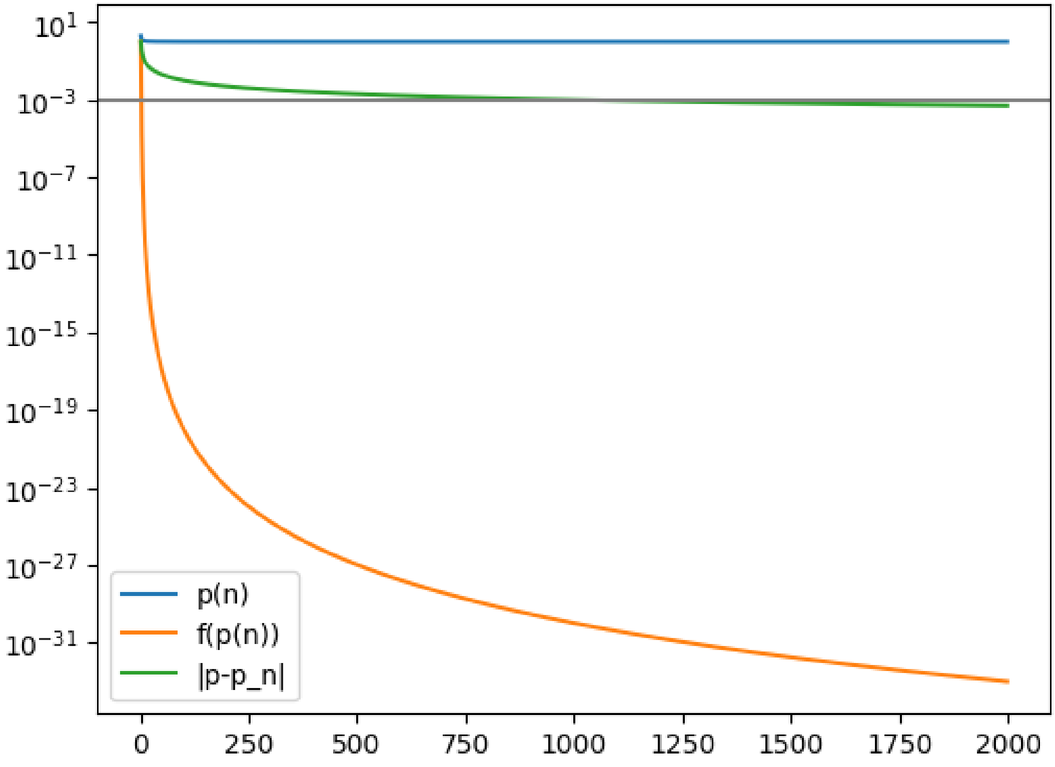
Show that $|f(p_n)| < 10^{-3}$ **whenever** $n > 1$ **but that** $|p - p_n| < 10^{-3}$ **requires that** $n > 1000$

[8]:

```
import numpy as np
import matplotlib.pyplot as plt

def exp_func(x):
    return (x - 1) ** 10
```

```
p = 1
n_values = np.arange(1, 2000)
p_n_values = 1 + (1 / n_values)

f_p_n = exp_func(p_n_values)
abs_p_minus_p_n = abs(p - p_n_values)

plt.plot(n_values, p_n_values, label='p(n)')
plt.plot(n_values, f_p_n, label='f(p(n))')
plt.plot(n_values, abs_p_minus_p_n, label='|p-p_n|')
plt.axhline(1e-3, color='grey')
plt.yscale('log')
plt.legend()
```

```
[8]: <matplotlib.legend.Legend at 0x211c7efbe90>
```



Alright, we've shown this numerically as above. Let's get down to moving some numbers around.

Substituting in to $|p - p_n| < 10^{-3}$ $\quad |1 - (1 + (\frac{1}{n}))| < 10^{-3} \Rightarrow |-\frac{1}{n}| < 10^{-3} \Rightarrow \frac{1}{n} < 10^{-3}$ $\Rightarrow n > 10^3$

Restating ■ $|p - p_n| > 10^{-3} \Leftrightarrow n > 1000$

section_6

February 7, 2025

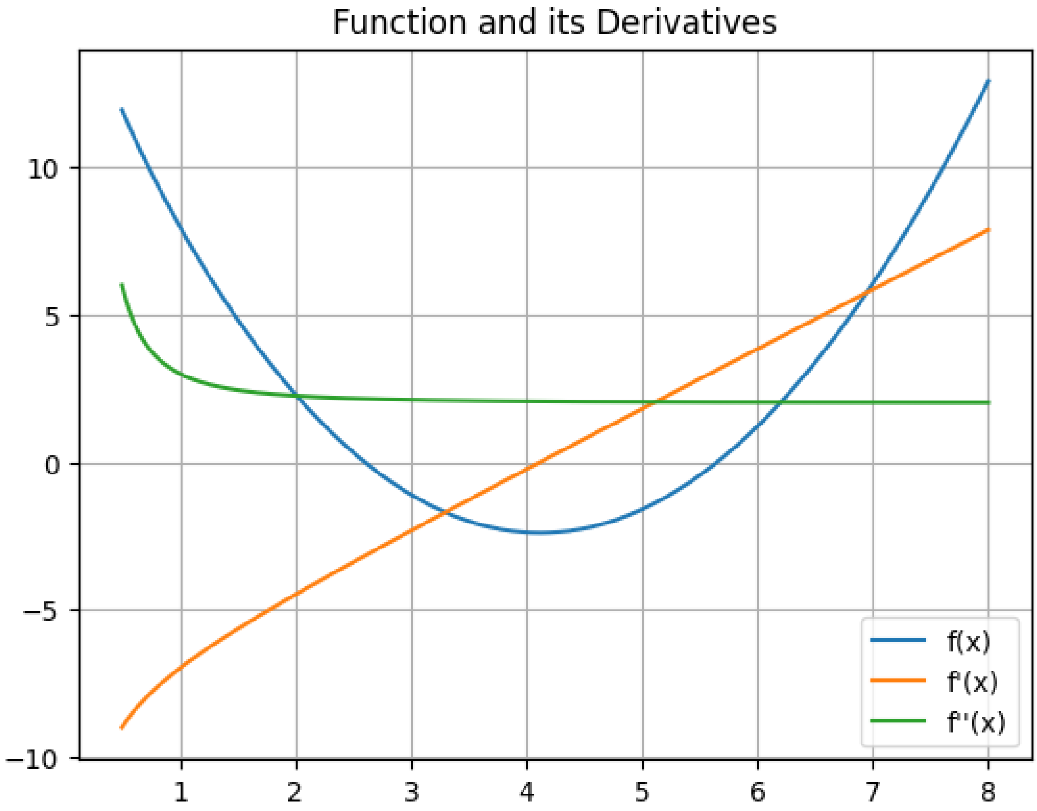
1 Section 6. Newton's method

(a) Consider the equation $ln(x) = (x-4)^2 - 1$

[50]:

```
import numpy as np
import matplotlib.pyplot as plt
import sympy

x = sympy.symbols('x')

f_of_x = (-sympy.log(x)) + ((x - 4) ** 2) - 1
f_prime_of_x = sympy.diff(f_of_x, x)
f_prime_prime_of_x = sympy.diff(f_prime_of_x, x)

# Lambdify the symbolic expressions to numerical functions
f_of_x_func = sympy.lambdify(x, f_of_x, 'numpy')
f_prime_of_x_func = sympy.lambdify(x, f_prime_of_x, 'numpy')
f_prime_prime_of_x_func = sympy.lambdify(x, f_prime_prime_of_x, 'numpy')


x_values = np.linspace(0.5, 8.0, 1000)

y_values = f_of_x_func(x_values)
y_prime_values = f_prime_of_x_func(x_values)
y_prime_prime_values = f_prime_prime_of_x_func(x_values)

f_prime_of_x_roots = sympy.solve(f_prime_of_x, x)
print(f"Roots of f': {f_prime_of_x_roots}")
for index, item in enumerate(f_prime_of_x_roots, 1):
    print(f'Root number {index} stated numerically: {item.evalf()}')
print(f"Root of f' stated numerically: {f_prime_of_x_roots[1].evalf()}")

plt.plot(x_values, y_values, label="f(x)")
plt.plot(x_values, y_prime_values, label="f'(x)")
plt.plot(x_values, y_prime_prime_values, label="f''(x)")
plt.legend()
plt.title("Function and its Derivatives")
plt.grid(True)
plt.show()
```

```
Roots of f': [2 - 3*sqrt(2)/2, 2 + 3*sqrt(2)/2]
Root number 1 stated numerically: -0.121320343559643
Root number 2 stated numerically: 4.12132034355964
Root of f' stated numerically: 4.12132034355964
```



Let $f(x) = (x-4)^2 - 1 - ln(x) \Rightarrow f'(x) = 2x^2 - 8x - 1 \Rightarrow f''(x) = x^{-2} + 2$

We can see the derivative crossing the x-axis only once, if we zoom out further, we'd see that $f'(x)$ is increasing from $x \approx 4.12$, we can also see that $f(x)$ is crossing the x-axis twice, for $f'(x)$ there's two roots, but, $ln(x)$ is *undefined* for $x \leq 0$

We could prove that there's no other roots to $f'(x)$ but, sympy has already told us otherwise. To borrow from work I've done previously, and tossing out the errogenous root, we know that as $f'(x)$ only crosses the x-axis once in the interval, we know we have at most two roots

■ There are at most two roots in the interval

(ii) **Let α be the smaller of the two solutions. Use Newton's Method to find an approximation of α with an absolute error of less than 10^{-3}**

[51]:

```python
from typing import Callable, Optional

def newtons_method(
        fOfX : Callable[[float], float],
        dfOfX : Callable[[float], float],
        guess : float,
        maxIterations : int = 100,
        maxError : float = 1e-6) -> Optional[float]:

    def formula() -> float:
        return guess - (fOfX(guess) / dfOfX(guess))

    if maxIterations == 0 and fOfX(guess) > maxError:
        return None
    elif fOfX(guess) < maxError:
        return guess
    else:
        return newtons_method(fOfX, dfOfX, formula(), maxIterations - 1, maxError)

f_of_x = lambda x : (-np.log(x)) + ((x - 4) ** 2) - 1
f_prime_of_x = lambda x : 2 * (x ** 2) - (8 * x) - 1
guess = newtons_method(
    fOfX = f_of_x,
    dfOfX = f_prime_of_x,
    guess = 1
)

print(f'Root at ({guess}, {f_of_x(guess)})')
```

```
Root at (2.601409918180815, 6.426907868117837e-07)
```

We can look at the graph before for a quick double check, this seems totally reasonable, the root is at $x \approx 2.6$

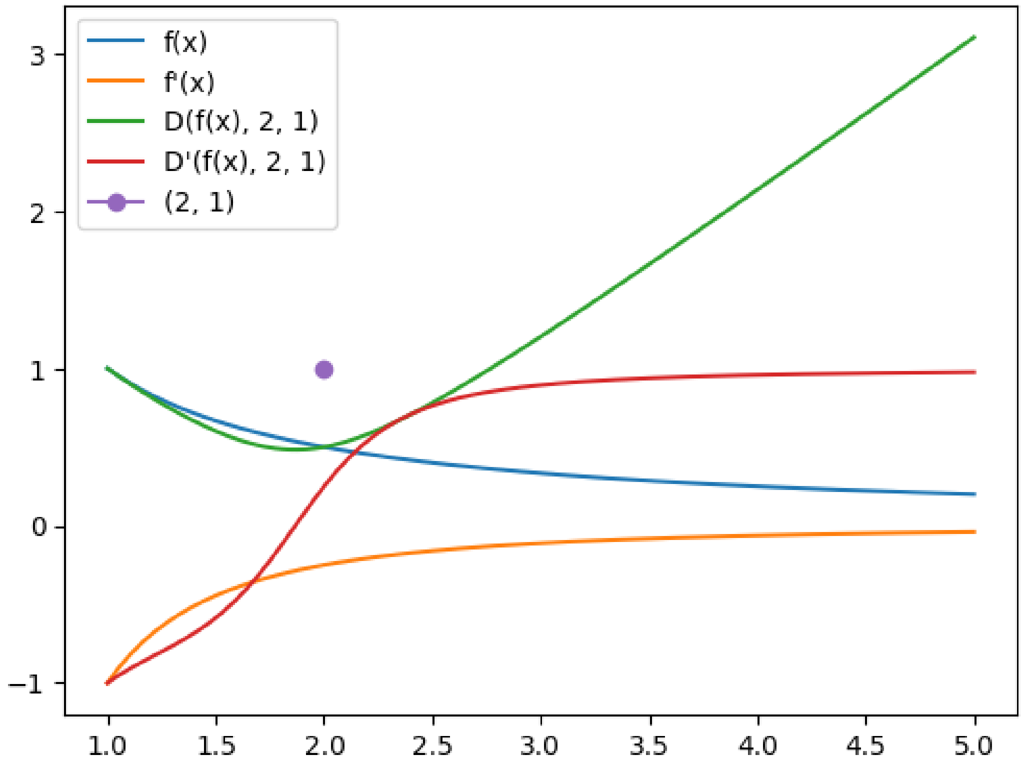
(b) **Use Newton's method to approximate to within 10^{-4}, the value of x that produces the point on the graph of $f(x) = \frac{1}{x}$ that is closest to $(2, 1)$.**

[52]:

```python
f_of_x_sympy = 1/x

distance_of_f_of_x_sympy = sympy.sqrt((x - 2)**2 + (f_of_x_sympy - 1)**2)
distance_prime_of_f_of_x_sympy = sympy.diff(distance_of_f_of_x_sympy, x)
distance_prime_prime_of_f_of_x_sympy = sympy.diff(distance_prime_of_f_of_x_sympy, x)
```

```
f_of_x = sympy.lambdify(x, f_of_x_sympy, 'numpy')
f_prime_of_x = sympy.lambdify(x, sympy.diff(f_of_x_sympy, x), 'numpy')

distance_of_f_of_x = sympy.lambdify(x, distance_of_f_of_x_sympy, 'numpy')
distance_prime_of_f_of_x = sympy.lambdify(x, distance_prime_of_f_of_x_sympy,␣
 ↪'numpy')
distance_prime_prime_of_f_of_x = sympy.lambdify(x,␣
 ↪distance_prime_prime_of_f_of_x_sympy, 'numpy')

x_values = np.linspace(1, 5, 1000)

plt.plot(x_values, f_of_x(x_values), label = "f(x)")
plt.plot(x_values, f_prime_of_x(x_values), label = "f'(x)")
plt.plot(x_values, distance_of_f_of_x(x_values), label = "D(f(x), 2, 1)")
plt.plot(x_values, distance_prime_of_f_of_x(x_values), label = "D'(f(x), 2, 1)")
plt.plot(2, 1, label = "(2, 1)", marker = 'o')
plt.legend()
guess = 2

guess = newtons_method(
    fOfX = distance_prime_of_f_of_x,
    dfOfX = distance_prime_prime_of_f_of_x,
    guess = 1
)

print(f'Root at ({guess}, {f_of_x(guess)})')
```

```
Root at (1, 1.0)
```



Alright, this is going to take a… little explaining - We write the distance equation $D(f(x), 2, 1) = \sqrt{(x-2)^2 + (\frac{1}{x} - 1)^2}$ - We could just minimize the distance equation, but, in order to use the Newton's method, we're gonna use the first and second derivative - Where the first derivative has a root, the distance equation is minimized - The root we found is at $f(x \approx 1.87) = 0.54$

(c) **Use Newton's method to approximation for λ, accurate to within 10^{-4}, for the population growth equation**

[64]:
```
f_of_x_lambda = lambda x: 1000000*sympy.exp(x) + (435000 / x) * (sympy.exp(x) - 1) - 1564000
f_of_x_sympy = sympy.sympify(f_of_x_lambda(x))
f_prime_of_x_sympy = sympy.diff(f_of_x_sympy, x)

f_of_x = sympy.lambdify(x, f_of_x_sympy)
f_prime_of_x = sympy.lambdify(x, f_prime_of_x_sympy)

root = newtons_method(
    f_of_x,
    f_prime_of_x,
    guess = 2,
    maxError = 10 ** (-4)
)
```

```python
print(f'Root found at ({root}, {f_of_x_lambda(root)}')
```

```
Root found at (0.10099792968651021, 0.000001018401235342O3
```