Numerical Computing :: Project Two

Julia Troni

1. Using the quadratic formula, compute the roots of $f(x) = 4x^2 - 3x - 3$. Show your work.
2. Implement bisection for root finding.
3. Transform the function f into an appropriate function g for a fixed point problem. Show your work.
4. Implement the fixed point method. Make sure you use good stopping criteria; see Sauer, section 1.2.4.
5. Using your implementations, compare the two root finding methods forfinding the root of f. Which is faster? Can you find a function such that the other root finding method is faster? For f, you know the true roots. What if you didn't know the true roots? How do you compute accuracy if you don't know the true answer?

In [1]:
```
import math
import timeit
```

Method implementations

Implement the methods used in the numerical experiment with comments

1. Quadratic Formula

$$f(x) = 4x^2 - 3x - 3$$

$$x = \frac{3 \pm \sqrt{9 - 4(4)(-3)}}{2(4)}$$

$$x = \frac{3 \pm \sqrt{9 + 48}}{8}$$

$$x = \frac{3 \pm \sqrt{57}}{8}$$

In [2]:
```
def quadEq(a,b,c):
    pos= (-b+ math.sqrt(b**2-(4*a*c)))/(2*a)
    neg= (-b - math.sqrt(b**2-4*a*c))/(2*a)
    return (pos,neg)
```

In [3]:
```
rts=quadEq(4,-3,-3)
print ("The roots found with quadratic method are ", rts)
```

```
The roots found with quadratic method are  (1.3187293044088437, -0.5687293044088437)
```

2. Bisection Method

I first implemented bisection method following Sauer 1.1 and I noticed that the method looks very similar to binary search by curring in half at each iteration

In [5]:
```
# Code from Sauer 1.1 Bisection Method and adapted to python
# Computes approximate solution of f(x)=0
# Input: function handle f; a,b such that f(a)*f(b)<0, and tolerance tol
# Output: Approximate solution xc

def bisect(f,a,b,tol):
    if (f(a))*(f(b)) >= 0:
        return print("f(a)f(b)<0 not satisfied!")
    fa=f(a);
    fb=f(b);
    while (b-a)/2>tol:
        c=(a+b)/2;
        fc=f(c);
        if fc == 0: #c is a solution, done
            return c
        if (fc)*(fa)<0: #a and c make the new interval
            b=c;
            fb=fc;
        else: #c and b make the new interval
            a=c;
            fa=fc;
    xc=(a+b)/2; #new midpoint is best estimate

    return(xc)
```

In [6]:
```
f = lambda x: 4*x**2-3*x-3
bisect(f, 0, 2, 0.0005)
```

Out[6]:
```
1.31884765625
```

In [7]:
```
bisect(f, -2, 0, 0.0005)
```

Out[7]:
```
-0.56884765625
```

Then I was curious how other implementations would compare so I found https://pythonnumericalmethods.berkeley.edu/notebooks/chapter19.03-Bisection-Method.html#:~:text=The%20bisection%20method%20uses%20the,interval%20(a%2Cb) and ran that

In [8]:
```
import numpy as np

def my_bisection(f, a, b, tol):
    # approximates a root of function f bounded by a and b to within tolerance
    # | f(m) | < tol with m the midpoint between a and b

    # check if a and b bound a root meaning check f(a)*f(b)>0
    if np.sign(f(a)) == np.sign(f(b)):
        raise Exception(
```

```
        "The scalars a and b do not bound a root")

    # get midpoint
    m = (a + b)/2

    if np.abs(f(m)) < tol:
        # stopping condition, report m as root
        return m
    elif np.sign(f(a)) == np.sign(f(m)):
        # case where m is an improvement on a.
        # Make recursive call with a = m
        return my_bisection(f, m, b, tol)
    elif np.sign(f(b)) == np.sign(f(m)):
        # case where m is an improvement on b.
        # Make recursive call with b = m
        return my_bisection(f, a, m, tol)
```

In [9]:

```
f = lambda x: 4*x**2-3*x-3

r1 = my_bisection(f, 0, 2, 0.0005)
print("The first root =", r1)

r2 = my_bisection(f, -1, 1, 0.0005)
print("The second root =", r2)
```

```
The first root = 1.3187255859375
The second root = -0.5687255859375
```

I had to play around with choosing an appropiate a and b such that f(a)*f(b)>0

- First I tried -1 and 1.9, however f(a)*f(b)<0 so then I tried -1 and 1.5 which also failed the check
- Then I realized that the bounds needed to be less than the larger root that I already found, in other words for calculating r2 any b works as long as both a and b < the first root
- So I chose a=-1 and b=1 for simplicity

Comparison

It found the same roots (roughly, and I assume this is likely due to floating point rounding error)

- for first implementation using bisect:

$$r1 = 1.31884765625, r2 = -0.56884765625$$

- for second implementation using my_bisection:

$$r1 = 1.3187255859375, r2 = -0.5687255859375$$

Transform the function f into a function g for a fixed point problem:

First transformation

$$4x^2 - 3x - 3 = 0$$

$$4x^2 - 3 = 3x$$

$$\frac{4x^2 - 3}{3} = x$$

So I let

$$g(x) = \frac{4x^2 - 3}{3}$$

However, as I found below this does not converge so I transformed f into many other acceptable g's, however unfortunately they do not converge to both roots.

Here are some of the other functions g that I found that sadly only converge to the negative root

$$g1(x) = \frac{3}{4x - 3}$$

$$g2(x) = \frac{2}{x - 1} - 3x - 1$$

I could not figure out why they would not converge to 1.3187, which frustuated me as these g's only found -0.5687. I ended up using g2 in my implemtation below and for the comparison

3. Fixed Point Method

In [10]:

```
#https://www.codesansar.com/numerical-methods/fixed-point-iteration-python-program-outp

import math

def f(x):
    return (4*x**2)+(3*x)-3


# Re-writing f(x)=0 to x = g(x)
def g(x):
    #return (4*x**2-3)/3 #this g is not convergent
    return (3)/(4*x-3) #this g only finds 1 root


# Implementing Fixed Point Iteration Method
# x0 is initial guess, N is number of iterations, e is the tolerance
def fixedPointIteration(x0, e, N):
    print('\n\n*** FIXED POINT ITERATION ***')
    step = 1
    flag = 1
    condition = True
    while condition:
        x1 = g(x0)
        print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1, f(x1)))
        x0 = x1

        step = step + 1

        if step > N:
            flag=0
```

```
            break

        condition = abs(f(x1)) > e

    print (x1)
    if flag==1:
        print('\nRequired root is: %0.8f' % x1)
    else:
        print('\nNot Convergent.')

fixedPointIteration(-1,0.005,20)
```

```
*** FIXED POINT ITERATION ***
Iteration-1, x1 = -0.428571 and f(x1) = -3.551020
Iteration-2, x1 = -0.636364 and f(x1) = -3.289256
Iteration-3, x1 = -0.540984 and f(x1) = -3.452298
Iteration-4, x1 = -0.580952 and f(x1) = -3.392834
Iteration-5, x1 = -0.563506 and f(x1) = -3.420362
Iteration-6, x1 = -0.570991 and f(x1) = -3.408850
Iteration-7, x1 = -0.567756 and f(x1) = -3.413881
Iteration-8, x1 = -0.569150 and f(x1) = -3.411724
Iteration-9, x1 = -0.568548 and f(x1) = -3.412656
Iteration-10, x1 = -0.568807 and f(x1) = -3.412255
Iteration-11, x1 = -0.568696 and f(x1) = -3.412428
Iteration-12, x1 = -0.568744 and f(x1) = -3.412353
Iteration-13, x1 = -0.568723 and f(x1) = -3.412386
Iteration-14, x1 = -0.568732 and f(x1) = -3.412372
Iteration-15, x1 = -0.568728 and f(x1) = -3.412378
Iteration-16, x1 = -0.568730 and f(x1) = -3.412375
Iteration-17, x1 = -0.568729 and f(x1) = -3.412376
Iteration-18, x1 = -0.568729 and f(x1) = -3.412376
Iteration-19, x1 = -0.568729 and f(x1) = -3.412376
Iteration-20, x1 = -0.568729 and f(x1) = -3.412376
-0.5687293218019169

Not Convergent.
```

Time Comparison

I ran the following code (I have commented it out so my pdf is not hundreds of pages long) but I got these results

```
%%timeit
fixedPointIteration(-1,0.005,20)
>>4.41 ms ± 565 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

my_bisection(f, 0, 2, 0.0005)
>> #262 µs ± 22.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)

my_bisection(f, -1, 1, 0.0005)
>> 258 µs ± 18.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops
each)
```

```
bisect(f, -2, 0, 0.0005)
>> 24 µs ± 1.52 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
each)
```

- Thus, fixedPointIteration not only takes more human time to compute, but it takes significantly more computer time to compute compared to both bisection implementations
- I thought it was interesting that bisect (my first implementation of bisection) was significantly faster than my_bisection (second implementation of bisection). This makes sense because my_bisection uses recursive calls which is undoubtely slower and more cumbersome

In [11]:

```
# %%timeit
# fixedPointIteration(-1,0.005,20)
# 4.41 ms ± 565 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

#my_bisection(f, 0, 2, 0.0005)
#262 µs ± 22.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

#my_bisection(f, -1, 1, 0.0005)
#258 µs ± 18.5 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

# bisect(f, -2, 0, 0.0005)
# 24 µs ± 1.52 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

References

- Sauer Numerical Analysis
- Python Numerical Methods for my_bisection https://pythonnumericalmethods.berkeley.edu/notebooks/chapter19.03-Bisection-Method.html#:~:text=The%20bisection%20method%20uses%20the,interval%20(a%2Cb)
- For fixed point implementation https://www.codesansar.com/numerical-methods/fixed-point-iteration-python-program-output.html
- And of course the one and only Paul https://www.youtube.com/playlist?list=PLSHpTCHnqCKGNQpiLC4Ka5tdvkRmvnN34

In [ ]: