Tyler Gabb Math 475A Homework 3

We were tasked with using three different iterative methods to approximate $\sqrt{n}$. It was first necessary to alter the form of this equation such that we can treat it as a root finding problem:

Find x such that $x^2 = n$

Bisection:

I reduced the equation to the form $f(x) = n - x^2 = 0$ and used bisection for solve for when $f(x) = 0$. Below is the code used, and note that the interval $[a.b]$ is depenent on whether the argument $n \in [0,1]$ or $n \in (1, \infty)$.

```
def bisection(n):
    if n < 1:
        a = 0
        b = 1
    else:
        a = 1
        b = n

    f = lambda x: n - (x * x)

    iters = 0
    x_last = 0
    x = (b + a)/2
    while perc_rel_err(x,x_last) > TOL:
        iters += 1
        if iters > 10000:
            print("\tbisection is not converging within tolerance given")
            break
        if f(x) == 0:
            break
        if f(b)*f(x) < 0:
            a = x
        elif f(a)*f(x) < 0:
            b = x
        x_last = x
        x = (b + a)/2
    return (x,iters)
```

Where **perc_rel_err(x,x_last)** is the relative error between successive iterations, and **TOL** is a floating point tolerance passed in as a command line argument

Newtons method:

Show that Newtons Method with f(x) as above amounts to iterating $\boldsymbol{x_{n+1} = \frac{1}{2}(x_n + \frac{n}{x_n})}$

Newtons method: $x_{n+1} = x_n - \frac{f(x)}{f'(x)}$

$$f(x_n) = n - x_n^2$$

$$f'(x_n) = -2x_n$$

Plugging into Newton's method: $x_{n+1} = x_n - \frac{f(x)}{f'(x)} = x_n - \frac{n - x_n^2}{-2x_n} = x_n - \frac{x_n^2 - n}{2x_n} = \frac{1}{2}(x_n + \frac{n}{x_n})$

This is shown computationally in the results section of this write-up

Below is the code for my Newtons method

```
def newtons(n):
    f = lambda x: n - x * x
    dfdx = lambda x: -2*x
    iterator = lambda x: x - f(x)/dfdx(x)#causes problem if try to find sqrt(0)
    x = n #set initial guess to the right to optimize convergence
    x_last = 0
    iters = 0
    while perc_rel_err(x,x_last) > TOL:
        iters += 1
        x_last = x
        x = iterator(x)
        if(iters > 10000):
            print("\tnewtons is not converging within tolerance given")
            break
    return(x,iters)
```

Note that this method WILL fail if dfdx(x) ≈ 0

I wrote additional code to implement the arithmetic solution to newton's method. It is below.

```
def equivalent_newtons(n):
    iterator = lambda x: (x + n/x)/2
    x = n #set initial guess to n;
    x_last = 0
    iters = 0
    while perc_rel_err(x,x_last) > TOL:
        iters += 1
        x_last = x
        x = iterator(x)
        if(iters > 10000):
            print("\tequivalent_newtons is not converging within tolerance given")
            break
    return (x,iters)
```

In the results section of this write-up, the results of these two methods are compared, and it is determined that a majority of the time they are identical.

Note that to optimize convergence, an initial guess should be placed to the right of the solution, or additional iterations are required to converge. This is shown in results section

Functional Iterations:

We were provided with a functional iterative equation with additional parameter α to optimize convergence.

$$F(x) = (1 - \alpha)x + \alpha n/x$$

This equation is not adequate for calculation of any x, for when calculating the square-root of a number near or equal to 0, the division operator will introduce round-off errors and potentially cause an exception.

If we were to pick $\alpha = \frac{1}{2}$, our $F(x)$ equation becomes

$$F(x) = \frac{1}{2}x + \frac{\alpha n}{2x} = \frac{1}{2}\left(x + \frac{n}{x}\right) \rightarrow x_{n+1} = \frac{1}{2}(x_n + \frac{n}{x_n})$$

Which we showed was equivalent to newtons method. **I hypothesize that picking $\alpha = 0.5$ will produce results which follow closely with those of newton's method.**

Results:

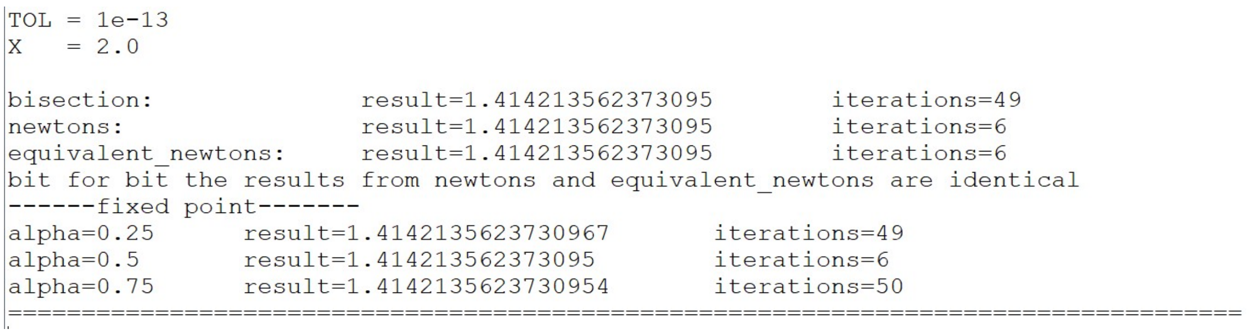
Below is a snippet of code used to compare all three methods. Note that both **n** and **TOL** are passed in as command line arguments

```
1def main():
2
3    n = float(sys.argv[1])
4    (result,iters) = bisection(n)
5    print('TOL = {0}'.format(TOL))
6    print('X   = {0}'.format(n))
7    print()
8    print('bisection:\t\tresult={0}\titerations={1}'.format(result,iters))
9    (result1,iters1) = newtons(n)
10   print('newtons:\t\tresult={0}\titerations={1}'.format(result1,iters1))
11   (result2,iters2) = equivalent_newtons(n)
12   print('equivalent_newtons:\tresult={0}\titerations={1}'.format(result2,iters2))
13   if result1 == result2 and iters1 == iters2:
14      print('bit for bit the results from newtons and equivalent_newtons are
        identical')

15  print('------fixed point-------')
16  for alpha in [0.25,0.5,0.75]:
17      (result,iters) = fixed_pt(n,alpha)
18      print('alpha={0}\tresult={1}\titerations={2}'.format(alpha,result,iters))
19
20  print('==============================================================')
```

Observe the conditional on line 13, this is used to print an indicative message when the two newtons methods yield the same results.

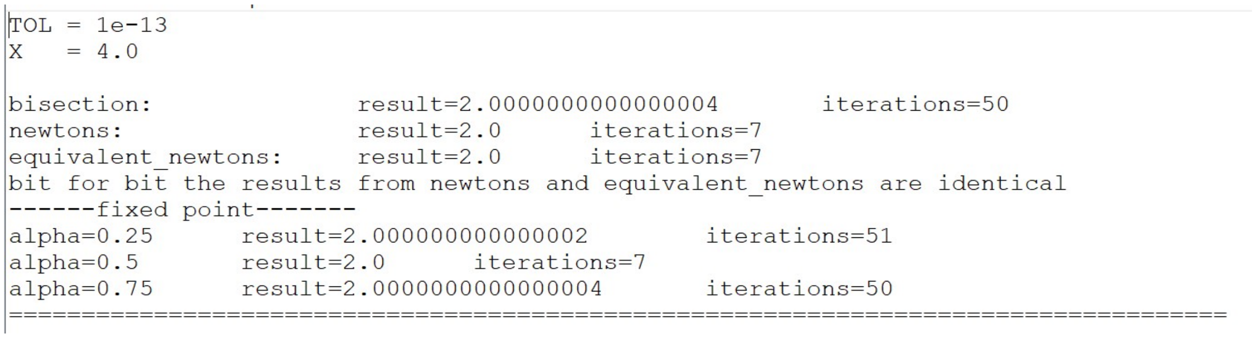
Below is the output for four tests

Test 1: n = 2 TOL = 1e-13



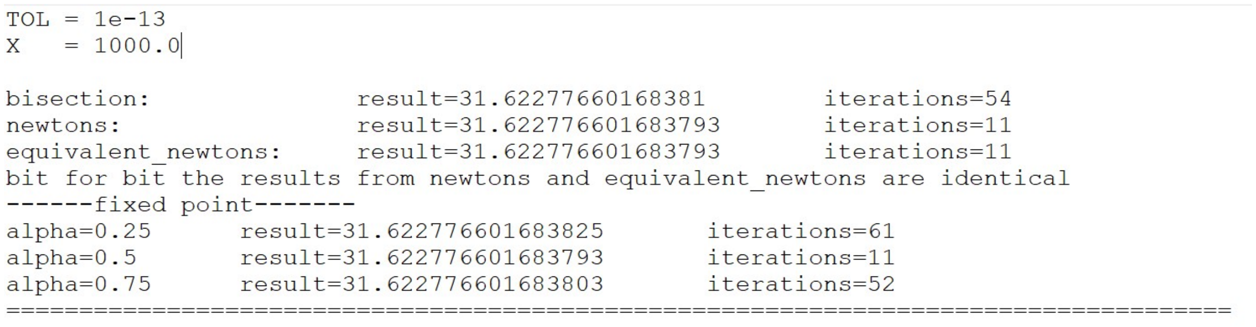
```
TOL = 1e-13
X   = 2.0

bisection:              result=1.414213562373095        iterations=49
newtons:                result=1.414213562373095        iterations=6
equivalent_newtons:     result=1.414213562373095        iterations=6
bit for bit the results from newtons and equivalent_newtons are identical
------fixed point-------
alpha=0.25      result=1.4142135623730967       iterations=49
alpha=0.5       result=1.414213562373095        iterations=6
alpha=0.75      result=1.4142135623730954       iterations=50
==================================================================================
```

Test 2: n = 4 TOL = 1e-13



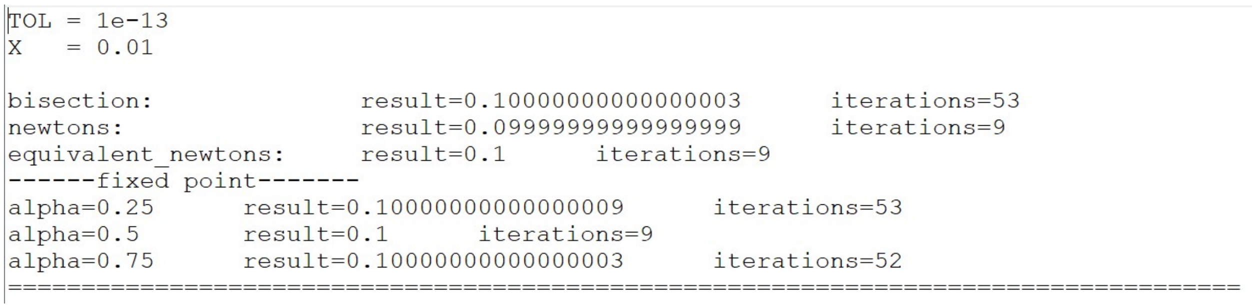
```
TOL = 1e-13
X   = 4.0

bisection:              result=2.0000000000000004       iterations=50
newtons:                result=2.0      iterations=7
equivalent_newtons:     result=2.0      iterations=7
bit for bit the results from newtons and equivalent_newtons are identical
------fixed point-------
alpha=0.25      result=2.000000000000002        iterations=51
alpha=0.5       result=2.0      iterations=7
alpha=0.75      result=2.0000000000000004       iterations=50
==================================================================================
```

Test 3: n = 1000 TOL = 1e-13



```
TOL = 1e-13
X   = 1000.0

bisection:              result=31.62277660168381        iterations=54
newtons:                result=31.622776601683793       iterations=11
equivalent_newtons:     result=31.622776601683793       iterations=11
bit for bit the results from newtons and equivalent_newtons are identical
------fixed point-------
alpha=0.25      result=31.622776601683825       iterations=61
alpha=0.5       result=31.622776601683793       iterations=11
alpha=0.75      result=31.622776601683803       iterations=52
==================================================================================
```

Test 4: n = 0.01 TOL = 1e-13



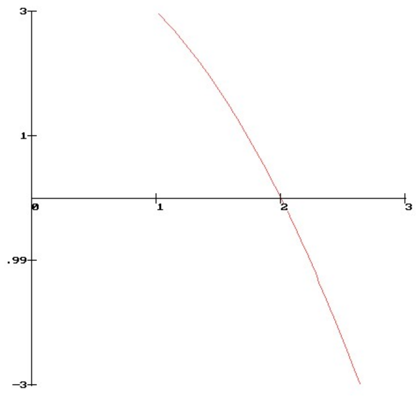
```
TOL = 1e-13
X   = 0.01

bisection:              result=0.10000000000000003      iterations=53
newtons:                result=0.09999999999999999      iterations=9
equivalent_newtons:     result=0.1      iterations=9
------fixed point-------
alpha=0.25      result=0.10000000000000009      iterations=53
alpha=0.5       result=0.1      iterations=9
alpha=0.75      result=0.10000000000000003      iterations=52
==================================================================================
```

Observations:

(1) Note test 4. No message was printed out for equivalence. I expected newtons method to fail before equivalent_newtons method. Notice that equivalent_newtons converged to the absolute solution in 9 iterations, where newtons was one bit off.

(2) In every test case, when alpha was set to 0.5, the fixed_pt iteration method converged to the same solution as newton's method, in the same number of iterations. $\alpha = 0.5$ seems to be the optimal value. This agrees with my hypothesis

(3) I mentioned earlier in this write-up that the initial guess for newtons method should be placed to the right of the anticipated solution to optimize convergence. Let us look at a plot of $f(x) = 4 - x * x$, the solution of which is 2



When an initial guess is placed to the left of 2, newton's method points the next iteration to the right of the solution, then additional iteraitons converge to the correct solution.

This could lead to to **at least** one more iteration than that of an initial guess placed to the right of the anticipated solution. (but of course not too far to the right)