Project 2

November 6, 2017

```
In [1]: # Just to get started on the notebook, let's bring in numpy
        import numpy as np
```

0.0.1 Problem 1

```
In [2]: # We first define a python generator that returns the bounds
        # for the next guess.
        def bisection_guesses( f, a, b ):
            while True:
                midpoint = (a+b)/2
                # If the sign of the midpoint is the same as the sign
                # of f(a), then update b
                if np.sign( f(midpoint) ) != np.sign( f(a) ):
                    b = midpoint
                #Otherwise, update a
                else:
                    a = midpoint
                #yield the next guess and wait for the next call.
                yield (a,b)
```

```
In [3]: # Because we defined the bisector generator above,
        # this function just needs to do 3 things:
        # 1 - validate input
        # 2 - provide an exit condition
        # 3 - ouput results
        def bisection_method( f, a, b, tol=1.0e-5, ittr_max=100):
            if( f(a) * f(b) > 0):
                print("You useless nincompoop! That's not a valid interval!")
            bis = bisection_guesses( f, a, b)
            ittr_count = 0
            while np.abs(a - b) > tol and ittr_count < ittr_max:
               a, b = next(bis)
               ittr_count += 1
            root = (a + b) / 2
            print( "Root is {}. Took {} iterations.\n\tThe resulting error was {}"
                  .format( a, ittr_count, b-a ) )
            return (a + b) / 2
```

0.0.2 Problem 2

```
In [4]: f = lambda x: np.log(x) - 2
        bisection_method( f , 6, 8, tol=1e-7, ittr_max=200)
```

```
Root is 7.389056086540222. Took 25 iterations.
        The resulting error was 5.960464477539063e-08
```

```
Out[4]: 7.3890561163425446
```

0.0.3 Problem 3

```
In [5]: #Just like in 1, we first define a generator.
        def newtons_guesses( f, f_prime, x0 ):
            current_x = x0;
            while True:
                yield current_x
                next_x = current_x - (f(current_x) / f_prime(current_x))
                current_x = next_x
```

```
In [6]: # And then a function that consumes guesses from that
        # generator and provides a nice (although far from kind)
        # user experience.
        def newtons_method( f, f_prime, x0, tol=1.0e-5, ittr_max=100 ):
            guesses = newtons_guesses( f, f_prime, x0 )
            ittr_count = 1
            current_guess = next(guesses) # it yields x0 first
            next_guess = next(guesses)
            while (np.abs(next_guess - current_guess) > tol
                   and ittr_count < ittr_max):
                current_guess = next_guess
                next_guess = next(guesses)
                ittr_count += 1
            print( "Root is {}. Took {} iterations.\n\t The result error was{}"
                  .format( next_guess, ittr_count, next_guess - current_guess ) )
            return next_guess
```

0.0.4 Problem 4

```
In [7]: f = lambda x: np.log(x) - 2
        f_prime = lambda x: 1/x
        newtons_method( f, f_prime, 6, tol=1e-7, ittr_max=200)
```

```
Root is 7.38905609893065. Took 5 iterations.
         The result error was1.7763568394002505e-15
```

```
Out[7]: 7.3890560989306504
```

0.0.5 Problem 5

```
In [8]: #Helper methods + variables for question 5
        def tan(x):
            return np.tan(x)

        def tan_p(x):
            return 1 / ( np.cos(x) ** 2 )

        bottom = 63 * np.pi / 2
        top = 65 * np.pi / 2
```

Problem 5.a)

```
In [9]: newtons_method(tan, tan_p, 100, tol = 1e-7, ittr_max = 20)
        newtons_method(tan, tan_p, 101, tol = 1e-7, ittr_max = 20)
        newtons_method(tan, tan_p, 102, tol = 1e-7, ittr_max = 20)
```

```
Root is 100.53096491487338. Took 4 iterations.
         The result error was1.1603162874962436e-10
Root is 100.53096491487338. Took 4 iterations.
         The result error was-4.575895218295045e-12
Root is 100.53096491487338. Took 8 iterations.
         The result error was0.0
```

```
Out[9]: 100.53096491487338
```

In each case it seems to find the answer after only a small number of iterations, though the number increases quite a bit as we move further from the 'true' answer.

Problem 5.b)

```
In [10]: for k in range(10):
             delta = 10 ** (-1 * k)
             print( "With k={} (delta={}), we search from x in [{}, {}].\n\t=> tan(x) in [{}, {}
                   .format( k, delta, bottom + delta, top - delta, tan( bottom + delta), tan( to
             bisection_method( tan, bottom + delta, top - delta, tol = 1e-8)
             newtons_method( tan, tan_p, bottom + delta, tol = 1e-8 )
```

```
With k=0 (delta=1), we search from x in [99.96016858807849, 101.10176124166827].
        => tan(x) in [-0.6420926159343285, 0.6420926159343175])
Root is 100.53096491487338. Took 27 iterations.
        The resulting error was 8.505523396706849e-09
Root is 100.53096491487338. Took 4 iterations.
         The result error was7.537153123848839e-10
With k=1 (delta=0.1), we search from x in [99.06016858807848, 102.00176124166828].
        => tan(x) in [-9.96664442325966, 9.966644423258874])
Root is 100.53096491487338. Took 29 iterations.
```

```
        The resulting error was 5.4791371439932846e-09
Root is 100.53096491487338. Took 8 iterations.
         The result error was0.0
With k=2 (delta=0.01), we search from x in [98.9701685880785, 102.09176124166827].
        => tan(x) in [-99.99666664437837, 99.99666664429998])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.814428050143761e-09
Root is 100.53096491487338. Took 11 iterations.
         The result error was1.318767317570746e-11
With k=3 (delta=0.001), we search from x in [98.9611685880785, 102.10076124166827].
        => tan(x) in [-999.9996666603981, 999.9996666525604])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.847951456416922e-09
Root is 100.53096491487338. Took 15 iterations.
         The result error was0.0
With k=4 (delta=0.0001), we search from x in [98.96026858807849, 102.10166124166827].
        => tan(x) in [-9999.999966187557, 9999.999965403784])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.851291007274995e-09
Root is 100.53096491487338. Took 18 iterations.
         The result error was0.0
With k=5 (delta=1e-05), we search from x in [98.96017858807849, 102.10175124166827].
        => tan(x) in [-99999.99995021097, 99999.99987183357])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.8516320677881595e-09
Root is 100.53096491487338. Took 21 iterations.
         The result error was5.016431714466307e-12
With k=6 (delta=1e-06), we search from x in [98.96016958807849, 102.10176024166827].
        => tan(x) in [-1000000.0010529909, 999999.9932152512])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.85166048949759e-09
Root is 100.53096491487338. Took 25 iterations.
         The result error was0.0
With k=7 (delta=1e-07), we search from x in [98.96016868807848, 102.10176114166828].
        => tan(x) in [-10000000.446538439, 9999999.662764478])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.851674700352305e-09
Root is 100.53096491487338. Took 28 iterations.
         The result error was0.0
With k=8 (delta=1e-08), we search from x in [98.96016859807848, 102.10176123166828].
        => tan(x) in [-100000048.06592856, 99999969.68851951])
Root is 100.53096491487338. Took 29 iterations.
        The resulting error was 5.851674700352305e-09
Root is 98.96016860807848. Took 1 iterations.
         The result error was9.999993721976352e-09
With k=9 (delta=1e-09), we search from x in [98.96016858907849, 102.10176124066827].
        => tan(x) in [-999994893.1389295, 999987055.5408959])
Root is 100.53096491487338. Took 29 iterations.
```

```
        The resulting error was 5.851674700352305e-09
Root is 98.9601685900785. Took 1 iterations.
         The result error was1.0000036354540498e-09
```

I might be missunderstanding the question, but if all that it's asking is what the smallest natural number k is such that `bottom + 10 ** (-1 * k)` to `top - 10 ** (-1 * k)` is a valid range for the bisection method, then I think the correct answer must be `k = 0` so `delta = 1`. As demonstrated above, this searches beteween `a = 99.960` and `b = 101.102`, which implies `tan(a) = -0.642` and `tan(b) = 0.642`, which is valid for the bisection method since `tan(a) * tan(b) < 0`.

I added in the newtons method results for comparison. Note that at `k >= 8`, this method fails to converge to the true answer.

Problem 5.c)

In [11]:
```
# Using k = 0 => delta = 1.
for j in range(1, 6): # from 1 to 5 inclusive
    print( "j={}".format( j ) )
    # Solve the bisection problem.
    bisection_guess = bisection_method(
        tan, bottom + 1, top - 1,
        tol = 10 ** (-1 * j) )
    # And plug it into Newtons method.
    newtons_guess = newtons_method(
        tan, tan_p, bisection_guess,
        tol = 1e-8 )
```

```
j=1
Root is 100.53096491487338. Took 4 iterations.
        The resulting error was 0.0713495408493543
Root is 100.53096491487338. Took 3 iterations.
         The result error was-2.842170943040401e-14
j=2
Root is 100.53096491487338. Took 7 iterations.
        The resulting error was 0.00891869260617284
Root is 100.53096491487338. Took 3 iterations.
         The result error was0.0
j=3
Root is 100.53096491487338. Took 11 iterations.
        The resulting error was 0.000557418287883138
Root is 100.53096491487338. Took 2 iterations.
         The result error was-1.4438228390645236e-11
j=4
Root is 100.53096491487338. Took 14 iterations.
        The resulting error was 6.967728597828682e-05
Root is 100.53096491487338. Took 2 iterations.
         The result error was-2.842170943040401e-14
```

```
j=5
Root is 100.53096491487338. Took 17 iterations.
        The resulting error was 8.709660747285852e-06
Root is 100.53096491487338. Took 2 iterations.
         The result error was0.0
```

In each case, Newton's method converged to the true answer.

Problem 5.d)

```
In [12]: times = np.matrix( '0.6295; 1.03725' ) # The given times
         # The iteration counts from the last question
         counts = np.matrix( '4 3; 7 3; 11 2; 14 2; 17 2' )
         counts * times # Matrix multiplication is always the best way to do it.
```

```
Out[12]: matrix([[  5.62975],
                 [  7.51825],
                 [  8.999  ],
                 [ 10.8875 ],
                 [ 12.776  ]])
```

Even though Newton's takes longer per iteration, it's still faster than the bisection method due to its quadratic convergence. Thus, j = 1 is the fastest, at just 5.62 ms.