Project 1, Raktim Biswas

1. Initial Approximations

Your input: solve the equation $-\frac{1}{100}+\frac{25}{4(x+7)^2}+\frac{9}{4(x+4)^2}+\frac{1}{100\left(x+\frac{3}{10}\right)^2}=0$ **for** x **on the interval** $(-\infty,\infty)$

ANSWER

$$x=\operatorname{RootOf}\{100x^6+2260x^5-66071x^4-795782x^3-2499897x^2-1343866x-260569,0\}\approx -35.4567304725451$$

$$x=\operatorname{RootOf}\{100x^6+2260x^5-66071x^4-795782x^3-2499897x^2-1343866x-260569,1\}\approx 23.0713631315813$$

Although we know lambda is greater than or equal to zero, our initial approximations don't necessarily need to be positive numbers to return a positive value for lambda. However, lambda_0 should not equal -7, -4, -0.3 to avoid dividing by zero due to the existence of vertical asymptotes in our function. Furthermore, when analyzing the results of our algorithms and studying how the quality of starting points affects the results and convergence, it's important to note that some of these behaviors depend on whether the function is increasing or decreasing as we approach either infinity or negative infinity along the x axis, as well as the derivative of the function. In our case, the function decreases to the horizontal asymptote at y=-0.01 as x increases, and the slope of the tangent line increases from negative values to 0 as x approaches infinity. To demonstrate the risk vs reward of each method I've listed scenarios that may or may not return our desired lambda value; however with either method we will see that we find the most stable and fastest results when our initial guesses are relatively close the value of lambda.

a. Newton: lambda_0 used for data table: 20.0

   i. Returns positive approximation: -6.4, -5.0, -4.1, -3.9, 20.0, 44.0, 57.0, 58.6

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [-6.4, -6.0916834058035985, -5.592558160972394, -4.395671264991286, -4.6081308924366535, -4.983
4525341620175, -6.185188221309151, -5.749680056643616, -4.899387205587536, -5.765285942862098, -4.938835047953379, -5.93
9719719753299, -5.316850482655894, 0.05305680064447049, 0.6498775083172057, 2.8572477653454667, 6.5897809552015225, 11.5
85450303270747, 17.12210652538348, 21.362499349685372, 22.92318590788866]
Initial guess:  -6.4
Final approximation:  22.92318590788866
Number of iterations:  20
```

1. Although using -6.4 as our initial approximation does not return a value within our desired tolerance, we see that even though our approximations bounce for

the first 11 iterations they begin to stabilize after this which demonstrates how newton's method can be slow if "good" points aren't used

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [-5.0, -6.294537300299734, -5.92622110167523, -5.290580437766467, 3.2212775383031325, 7.1102224
841799, 12.226161181938707, 17.72423208704493, 21.680936513988968, 22.972906559447846, 23.070862172797053, 23.0713631185
97688, 23.071363131581297, 23.071363131581297]
Initial guess:  -5.0
Final approximation:  23.071363131581297
Number of iterations:  13
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [-3.9, -3.8498621955578565, -3.774349228015504, -3.6601220343092726, -3.485883331555984, -3.216
1549523460264, -2.78896998543904, -2.0932640110988547, -0.9334612789501071, 2.2460654008860104, 5.69252042183321, 10.448
883280629799, 15.989637989925345, 20.68283636266474, 22.784131507220508, 23.067108591363933, 23.07136219514451, 23.07136
3131581254, 23.071363131581297]
Initial guess:  -3.9
Final approximation:  23.071363131581297
Number of iterations:  18
```

2. Despite the first approximation straying away from our desired value for lambda, using -5.0 as our initial approximation returns a value with our desired precision. This is interesting considering that -5.0 falls between two asymptotes on the graph of f(lambda). Similarly, -3.9 as an initial guess returns a desired lambda without bouncing although it falls between the asymptotes at x=-4, -0.3. This is because the tangents at these points are not so steep that we divide by values so large that would cause our approximations to converge slowly.

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [-4.1, -4.150168788878219, -4.225856026354654, -4.34101186059832, -4.520275070079996, -4.819603
952998355, -5.487629114747554, -3.8230289656016248, -3.7338348586710333, -3.598538781322794, -3.3911171280884007, -3.067
360379232382, -2.5488775271460438, -1.6956573533486397, -0.25250384758230293, -0.227232688830396, -0.18547111542552913,
-0.10793576379557522, 0.07630903972097582, 0.7488914080421948, 3.102858871812335]
Initial guess:  -4.1
Final approximation:  3.102858871812335
Number of iterations:  20
```

3. Because the tangent at lambda_0 = -4.1 is so steep, we see how slow Newton's method converges when approximations are made that are not local to the desired zero.

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [44.0, -4.947859410089151, -5.984571972554756, -5.401706650354685, -2.892189094665155, -2.26301
30814983698, -1.2185884218863348, 0.7154910026225876, 3.0227061915477256, 6.827443608069679, 11.879745019462515, 17.4020
0384177781, 21.514304977411868, 22.948128982729905, 23.070578522228907, 23.0713630997322, 23.0713631315813, 23.071363131
581297]
Initial guess:  44.0
Final approximation:  23.071363131581297
Number of iterations:  17
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [57.0, -59.34022503123654, 1.9903018201532348, 5.3051290288965225, 9.945865866779503, 15.463155
168117853, 20.332086171008264, 22.695107940762004, 23.064069958378457, 23.071360379942625, 23.071363131580906, 23.071363
131581297]
Initial guess:  57.0
Final approximation:  23.071363131581297
Number of iterations:  11
```

4. Despite not being local to lambda, 44.0 and 57.0 give us appropriate approximations; although we see more iterations and bouncing with 44.0, we have faster convergence with 57.0

ii. Returns negative approximation: -6.5, 45.0,50.0 56.0, 56.5, 58.7, 59.1

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [45.0, -7.97115988937103, -8.463878280114479, -9.212175471160535, -10.350093013180237, -12.072066886475774, -14.635644809198933, -18.316262786021777, -23.217908456864595, -28.79089143483879, -33.33500807094614, -35.22981565334765, -35.45407795023858, -35.45673010916895, -35.45673047254509, -35.4567304725451]
Initial guess:  45.0
Final approximation:  -35.4567304725451
Number of iterations:  15
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations:  [56.0, -53.917956512278224, -14.166950787530368, -17.65755136917148, -22.37973568007286, -27.92997429492588, -32.7800340270805, -35.0979089817006, -35.450107712919845, -35.45672820739612, -35.456730472544834, -35.456730472545104]
Initial guess:  56.0
Final approximation:  -35.456730472545104
Number of iterations:  11
List of approximations:  [56.5, -56.601458846049724, -6.823965012869955, -6.735820965464345, -6.603259462069121, -6.4030239955155945, -6.0963768987664135, -5.600621954845165, -4.428506059256329, -4.662122042894516, -5.093552401574966, -7.287185039715139, -7.431118987637782, -7.64768606047763, -7.9743327250479865, -8.468688448335582, -9.21948982024181, -10.36120585850361, -12.088795051930726, -14.660231740485939, -18.35062582662262]
Initial guess:  56.5
Final approximation:  -18.35062582662262
Number of iterations:  20
List of approximations:  [59.1, -71.45646564846305, 58.83753045188536, -69.8872639038591, 49.704310105335665, -24.67440777521206, -30.18446071875555, -34.106788687192335, -35.36405000582364, -35.45628730138698, -35.456730462401495, -35.456730472545104, -35.456730472545104]
Initial guess:  59.1
Final approximation:  -35.456730472545104
Number of iterations:  12
```

1. Interestingly, we find pockets of values between our previous guesses that give us the alternate x intercept of the function. For ranging from 45.0 to 56.0 we see quick convergence, although 56.5 shows us slow convergence that approaches the negative x intercept. For initial guess roughly between 58.7 and 59.1 we see that they bounce initially but converge within 20 iterations.

iii. Result too large: +59.2

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
Enter initial guess for p0 as a decimal value: 59.2
1 -72.05849503881934
2 62.491143033894005
3 -93.17973155833512
4 249.46962648900706
5 -9440.194817915124
6 493304383.30588466
7 -7.053186765935434e+22
8 2.0615629290389077e+65
9 -5.147901549356019e+192
```

```
dy = lambda x: -(12.5/(7+x)**3 + 4.5/(4+x)**3 + 0.02/(0.3+x)**3)
guess = float(input("Enter initial guess for p0 as a decimal value: "
#eps = float(
#maxit= int(i
#user inputs
#user input f
lamb = newton
```

```
Exception Thrown
(34, 'Result too large')
Copy Details
```

No issues f

1. The program stops running after the ninth iteration because we encounter overflow when trying to calculate the denominators in one or all terms of our derivative function, giving us extremely small values for the derivative function. Additionally, we can see that the approximations jump between positive and negative values as well as increasing in magnitude drastically with each iteration; the sign of pn+1 is opposite to the sign of f'(pn) because at all of these points f(pn) is less than 0. This behavior in our program demonstrates how we sacrifice reliability for speed by using Newton's method with "bad points". This can be explained by relating the idea that Newton's method is locally convergent and how f'(pn) cannot be equal to zero while using this method because we cannot divide by zero, and as f'(pn) approaches 0, the absolute value of f(pn)/f'(pn) approaches infinity.

b. Secant: lambda_0 and lambda_1 used for data table: 20.0, 24.0

i. Positive values for (p0, p1) that return the negative x inctercept: (50.0,55.0), (55.0, 0.0)

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [50.0, 55.0, -36.15912231122357, -42.010383345240164, -35.20991172752576, -35.54292053569076, -35.45782817997917, -35.456725583662646, -35.45673047282227, -35.4567304725451, -35.4567304725451]
Final approximation:  -35.4567304725451 Number of iterations:  9
```

1. As we will see later, we can select pairs (p0, p1) such that lambda is less than both values, and still return our desired lambda. In the case of the points where x= 50.0, 55.0 (order doesn't matter, behavior and speed are the same) we are

given the negative x intercept. The first approximation is relatively close to the negative x intercept and by the fourth iteration it begins to converge quickly.

ii. Values of (p0, p1) that are both greater than lambda:

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [40.0, 44.0, 0.8396704523412382, 42.58699081989373, 41.26012347648524, 0.7980227323055331, 40.03
410199909246, 38.88314223896054, 6.628511273640882, 35.108888593315044, 32.191373040140256, 16.71719749240222, 26.049849
655387064, 24.023177914407444, 22.921502002322278, 23.078775224404026, 23.071420553088466, 23.07136310956077, 23.0713631
31581364, 23.0713631315813]
Final approximation:  23.0713631315813 Number of iterations:  18
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [44.0, 40.0, 0.8396704523412382, 38.83322297725858, 37.739755372182394, 9.05600565905008, 32.910
360129584056, 29.61603812895961, 19.434888229900537, 24.31105502053926, 23.30089234109867, 23.05652153974856, 23.0715395
86694, 23.071363267060576, 23.07136313158006, 23.071363131581297, 23.071363131581297]
Final approximation:  23.071363131581297 Number of iterations:  15
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [45.0, 48.0, -12.684788185991408, 46.04570182711158, 44.21156618870666, -8.314062636708073, 44.1
1818811302393, 44.02515105268498, -5.1584578988114345, 43.932578849535574, 43.840355433946506, -4.615723925802882, 43.79
49155591784, 43.749561003725205, -4.284108633324884, 43.73855366912551, 43.72755138250622, -4.170874426979445, 43.723503
359921516, 43.71945602165234, -4.1374763441802, 43.716828813528714]
Final approximation:  43.716828813528714 Number of iterations:  20
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [48.0, 45.0, -12.684788185991408, 43.231334467861465, 41.57547263813066, -0.44004728097943513, 4
1.25804740616248, 40.945531492523756, 2.828565584993548, 38.81309460143023, 36.915514108425796, 9.916053385287341, 31.87
427274201942, 28.61470010842159, 20.344676296065526, 23.860814308685583, 23.181378486794156, 23.066847211344957, 23.0713
88850383215, 23.071363137589994, 23.07136313158129, 23.071363131581297]
Final approximation:  23.071363131581297 Number of iterations:  20
```

1. We see some interesting behavior with both pairs. In general, we see approximations bouncing back and forth between values that are not near lambda though they do ultimately converge. In both scenarios we see that if we set p0 greater than p1, we have slower convergence and in the case of (45.0, 48.0) we see very slow convergence that exceeds our maximum iterations

iii. Values of (p0, p1) that are both less than lambda: (0.0, 0.1), (0.1, 0.0), (20.0, 23.0), (23.0, 20.0)

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [0.0, 0.1, 0.6263892572771831, 1.7998855428168037, 4.006853832493138, 6.7275995624741025, 10.131
669527998028, 13.936806960672207, 17.724489097894484, 20.754332247321983, 22.45866075174022, 22.999147690237518, 23.0690
82870905362, 23.071354615765, 23.07136313057668, 23.071363131581297, 23.071363131581297]
Final approximation:  23.071363131581297 Number of iterations:  15
```

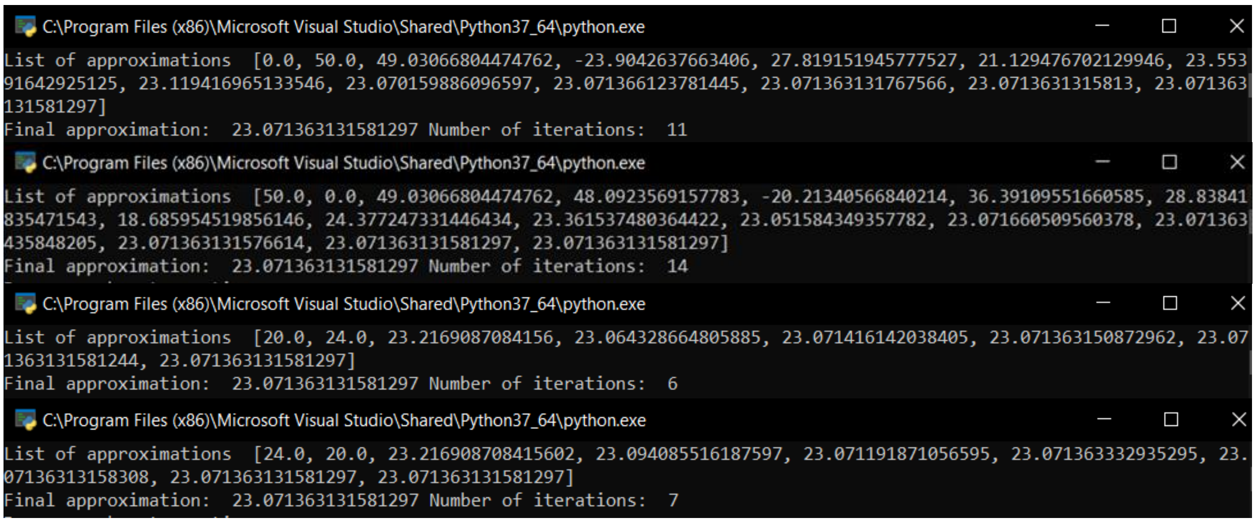
```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [0.1, 0.0, 0.6263892572771831, 1.4918620607097965, 3.7244744638558562, 6.2979144459272, 9.654179
037605086, 13.407918416071938, 17.24346186211677, 20.416335447327665, 22.309890050105984, 22.968814418901506, 23.0673429
6502737, 23.071341815500254, 23.07136312714791, 23.071363131581293, 23.071363131581297]
Final approximation:  23.071363131581297 Number of iterations:  15
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
List of approximations  [20.0, 23.0, 23.06023432098793, 23.071322062510852, 23.071363107936786, 23.07136313158125, 23.07
1363131581297]
Final approximation:  23.071363131581297 Number of iterations:  5
```

```
C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
23.0 20.0 23.06023432098793 23.069627101140462 23.07136213211388 23.07136313149153 23.0713631315813 23.071363131581297
List of approximations  [23.0, 20.0, 23.06023432098793, 23.069627101140462, 23.07136213211388, 23.07136313149153, 23.071
3631315813, 23.071363131581297]
Final approximation:  23.071363131581297 Number of iterations:  6
```

1. Like values of (p0, p1) that are both greater than lambda, we still have convergence and with both of these pairs we have stable convergence. There is only a slight difference in the rate convergence of convergence depending on the order of values chosen; it appears as though setting p0 less than p1 gives us slightly faster convergence

iv. Values of (p0, p1) such that lambda is between them: (0.0, 50.0), (50.0, 0.0), (20.0, 24.0), (24.0, 20.0)



1. Not surprisingly we have convergence when lambda is between p0 and p1, with quicker convergence as the distance between p0 and p1 decreases. We see that by setting p0 greater than p1, we get slower convergence, so the order of a pair of starting points matter.

2. Final Approximations

a. Newton

i. Lambda=23.071363131581297

b. Secant

i. Lambda=23.071363131581297

3. Table of Convergence Rates

a. Newton using 20.0 as lambda_0

iteration (n)	pn-1	pn	pn+1	p	|pn+1-p|	|pn-p|	|pn-1-p|	ln(|pn+1-p|/|pn-p|)	ln(|pn-p|/|pn-1-p|)	alpha
1	20	22.60016974	23.0599374	23.0713631 3	0.011425727	0.47193389	3.07136313 2	-3.719401028	-1.874608157	1.98409519
2	22.60016974	23.0599374	23.0713563	23.071363	6.75314E	0.0114257	0.4711933	-7.4336	-3.7194	1.998605423

			8	13	-06	27	89	15064	01028	
3	23.0599374	23.07135638	23.07136313	23.07136313	2.30216E-12	6.75314E-06	0.011425727	-14.8916712	-7.433615064	2.003287912
4	23.07135638	23.07136313	23.07136313	23.07136313	0	2.30216E-12	6.75314E-06	#NUM!	-14.8916712	#NUM!
5	23.07136313	23.07136313	23.07136313	23.07136313	0	0	2.30216E-12	#DIV/0!	#NUM!	#DIV/0!
6	23.07136313	23.07136313	23.07136313	23.07136313	0	0	0	#DIV/0!	#DIV/0!	#DIV/0!

b. Secant using 20.0, 24.0 as lambda_0, lambda_1

iteration (n)	pn-1	pn	pn+1	p	|pn+1-p|	|pn-p|	|pn-1-p|	ln(|pn+1-p|/|pn-p|)	ln(|pn-p|/|pn-1-p|)	alpha
0	20	24	23.21690871	23.07136313	0.145545577	0.928636868	3.071363132	-1.853228498	-1.196158981	1.549316209
1	24	23.21690871	23.06432866	23.07136313	0.007034467	0.145545577	0.928636868	-3.029667389	-1.853228498	1.634805094
2	23.21690871	23.06432866	23.07141614	23.07136313	5.30105E-05	0.007034467	0.145545577	-4.888087971	-3.029667389	1.613407462
3	23.06432866	23.07141614	23.07136315	23.07136313	1.92917E-08	5.30105E-05	0.007034467	-7.918569492	-4.888087971	1.619972787
4	23.07141614	23.07136315	23.07136313	23.07136313	0	1.92917E-08	5.30105E-05	#NUM!	-7.918569492	#NUM!
5	23.07136315	23.07136313	23.07136313	23.07136313	0	0	1.92917E-08	#DIV/0!	#NUM!	#DIV/0!
6	23.07136313	23.07136313	23.07136313	23.07136313	0	0	0	#DIV/0!	#DIV/0!	#DIV/0!

Although excel has its limitations in precision while making calculations, we see in both tables that the ratio we were given approximates alpha; for Newton's method our values for alpha confirm that our algorithm converges with a quadratically with an order of two, and for secant method our program converges super-linearly with an order equivalent to about 1.6. For both methods we see that absolute errors of pn+1-p converge to 0 faster than the absolute error of pn-1-p, and the convergence of the absolute error of pn-p falls between the two. This is expected since our newest approximations are intended to be more accurate than our initial approximations. Furthermore we see that the natural logarithms of our given ratios increase in magnitude but remain negative; this is because these ratios demonstrate that the error in newer approximations are always smaller than previous approximations, and the ratios approach 0 as our iterations increase. The natural log demonstrates how e^x approaches 0 as x approaches negative infinity.