Homework 1

Daniel Helfrich

June 11, 2014

Problem 1. Find roots of $f(x) = x + e^x$. The root is approximately -0.56714329 in all methods.

(a) Bisection method with initial endpoints -2 and 1. This method converges very slowly and the relative error only decreases by a factor of two for each iteration.

```
Bisection Method
Iteration: 0, a: -2.00000000, b: 1.00000000, p: -0.50000000  error = 100.00000000
Iteration: 1, a: -2.00000000, b: -0.50000000, p: -1.25000000  error = 1.50000000
Iteration: 2, a: -1.25000000, b: -0.50000000, p: -0.87500000  error = 0.30000000
Iteration: 3, a: -0.87500000, b: -0.50000000, p: -0.68750000  error = 0.21428571
Iteration: 4, a: -0.68750000, b: -0.50000000, p: -0.59375000  error = 0.13636364
Iteration: 5, a: -0.59375000, b: -0.50000000, p: -0.54687500  error = 0.07894737
Iteration: 6, a: -0.59375000, b: -0.54687500, p: -0.57031250  error = 0.04285714
Iteration: 7, a: -0.57031250, b: -0.54687500, p: -0.55859375  error = 0.02054795
Iteration: 8, a: -0.57031250, b: -0.55859375, p: -0.56445312  error = 0.01048951
Iteration: 9, a: -0.57031250, b: -0.56445312, p: -0.56738281  error = 0.00519031
Iteration: 10, a: -0.56738281, b: -0.56445312, p: -0.56591797  error = 0.00258176
Iteration: 11, a: -0.56738281, b: -0.56591797, p: -0.56665039  error = 0.00129422
Iteration: 12, a: -0.56738281, b: -0.56665039, p: -0.56701660  error = 0.00064627
Iteration: 13, a: -0.56738281, b: -0.56701660, p: -0.56719971  error = 0.00032293
Iteration: 14, a: -0.56719971, b: -0.56701660, p: -0.56710815  error = 0.00016141
Iteration: 15, a: -0.56719971, b: -0.56710815, p: -0.56715393  error = 0.00008072
```

(b) Fixed point iteration method with initial guess of 0.5 and $g(x) = -e^x$ This method converges at about the same rate as bisection. However, it initially goes off in the wrong direction before converging.

```
FixedPt Method
Iteration: 0 x: 0.500000000000000  error: N/A
Iteration: 1 x: -1.648721270700128  error: 4.297442541400256
Iteration: 2 x: -0.192295645547965  error: 0.883366795245926
Iteration: 3 x: -0.825062906255259  error: 3.290595888971710
```

```
Iteration: 4 x: -0.438207425609723  error: 0.468879982014185
Iteration: 5 x: -0.645191939645051  error: 0.472343693736658
Iteration: 6 x: -0.524561848228372  error: 0.186967759521364
Iteration: 7 x: -0.591814612188527  error: 0.128207501531593
Iteration: 8 x: -0.553322307784673  error: 0.065041152433715
Iteration: 9 x: -0.575036186118657  error: 0.039242730734857
Iteration: 10 x: -0.562684507070213  error: 0.021479829177038
Iteration: 11 x: -0.569677705469196  error: 0.012428276078536
Iteration: 12 x: -0.565707733830244  error: 0.006968802887735
Iteration: 13 x: -0.567958041362497  error: 0.003977862414956
Iteration: 14 x: -0.566681398062763  error: 0.002247777488406
Iteration: 15 x: -0.567405310063066  error: 0.001277458555685
Iteration: 16 x: -0.566994707188182  error: 0.000723650039227
Iteration: 17 x: -0.567227564647697  error: 0.000410687183784
Iteration: 18 x: -0.567095496855108  error: 0.000232830350321
Iteration: 19 x: -0.567170396851394  error: 0.000132076513924
Iteration: 20 x: -0.567127917381652  error: 0.000074897191352
```

(c) Newton's method converges very quickly with an initial guess of 0.5, but requires the value of the derivative to be known at every point. It is possible to use Newton's method because we can calculate $f'(x) = 1 + e^x$.

```
Newton's Method
Iteration: 0 x: 0.500000000000000  Error: 1.000100000000000
Iteration: 1 x: -0.311229665600927  Error: 2.606530659712634
Iteration: 2 x: -0.554407072604111  Error: 1.622459331201854
Iteration: 3 x: -0.567113824854529  Error: 0.781343920200064
Iteration: 4 x: -0.567143290252680  Error: 0.022919534901915
Iteration: 5 x: -0.567143290409784  Error: 0.000051956762222
```

(d) The secant method is used below with initial guesses of 0.45 and 0.5. It is only slightly slower than Newton's because we are only using an estimate of the derivative instead of the true value.

```
Secant Method
Iteration: 1 x: 0.450000000000000  Error: 1.000100000000000
Iteration: 2 x: -0.323838794375666  Error: 0.100000000000000
Iteration: 3 x: -0.514828004847278  Error: 1.719641765279258
Iteration: 4 x: -0.564738242652279  Error: 0.589766309005142
Iteration: 5 x: -0.567120333934678  Error: 0.096945460105277
Iteration: 6 x: -0.567143280415685  Error: 0.004218044932129
Iteration: 7 x: -0.567143290409742 f(x): 0.000000000000065
```

(e) The false position method takes a very large number of iterations. Note that one of the endpoints never changes. Below is the data from the starting points -3 and 3. However, the advantage of this method is that it guarantees that the root will be trapped. This can be sped up using the halving method. It probably converges so slow because $b = 3$ is so far away from the root.

```
        FalsePosition Method
Iteration: 1 a: -3.000000000000000 b: 3.000  Error: 1.000100000000000
Iteration: 2 a: -2.320116467223012 b: 3.000  Error: 0.226627844258996
Iteration: 3 a: -1.853038542277555 b: 3.000  Error: 0.201316585414572
Iteration: 4 a: -1.520855268030006 b: 3.000  Error: 0.179264093362713
Iteration: 5 a: -1.279438214481248 b: 3.000  Error: 0.158737690971391
Iteration: 6 a: -1.101549578848593 b: 3.000  Error: 0.139036519012198
Iteration: 7 a: -0.969295437620997 b: 3.000  Error: 0.120061905307827
Iteration: 8 a: -0.870388608718669 b: 3.000  Error: 0.102039919990834
Iteration: 9 a: -0.796128647573648 b: 3.000  Error: 0.085318167541671
Iteration: 10 a: -0.740223960586585 b: 3.000  Error: 0.070220669934995
Iteration: 11 a: -0.698059392892777 b: 3.000  Error: 0.056961906043131
Iteration: 12 a: -0.666216462577894 b: 3.000  Error: 0.045616362503089
Iteration: 13 a: -0.642146175527107 b: 3.000  Error: 0.036129829271478
Iteration: 14 a: -0.623939118822374 b: 3.000  Error: 0.028353445677361
Iteration: 15 a: -0.610160380546520 b: 3.000  Error: 0.022083465934722
Iteration: 16 a: -0.599729182233137 b: 3.000  Error: 0.017095830286523
Iteration: 17 a: -0.591830156116159 b: 3.000  Error: 0.013170988424417
Iteration: 18 a: -0.585847444476609 b: 3.000  Error: 0.010108832031829
Iteration: 19 a: -0.581315483340207 b: 3.000  Error: 0.007735735948206
Iteration: 20 a: -0.577882102995773 b: 3.000  Error: 0.005906225522682
Iteration: 21 a: -0.575280785054713 b: 3.000  Error: 0.004501468253775
Iteration: 22 a: -0.573309761163517 b: 3.000  Error: 0.003426194551254
Iteration: 23 a: -0.571816242527321 b: 3.000  Error: 0.002605081471429
Iteration: 24 a: -0.570684507567663 b: 3.000  Error: 0.001979193446229
Iteration: 25 a: -0.569826896457791 b: 3.000  Error: 0.001502776224867
Iteration: 26 a: -0.569176999023878 b: 3.000  Error: 0.001140517300872
Iteration: 27 a: -0.568684499436638 b: 3.000  Error: 0.000865283713299
Iteration: 28 a: -0.568311273399447 b: 3.000  Error: 0.000656297186861
Iteration: 29 a: -0.568028432771683 b: 3.000  Error: 0.000497686111471
Iteration: 30 a: -0.567814087194806 b: 3.000  Error: 0.000377350084099
Iteration: 31 a: -0.567651648520310 b: 3.000  Error: 0.000286077218160
Iteration: 32 a: -0.567528546268483 b: 3.000  Error: 0.000216862317141
Iteration: 33 a: -0.567435254396620 b: 3.000  Error: 0.000164382694890
Iteration: 34 a: -0.567364553883840 b: 3.000  Error: 0.000124596616499
Iteration: 35 a: -0.567310973966706 b: 3.000  Error: 0.000094436490203
```

(f) The false position with halving method allows the stationary endpoint to provide a better estimate for the slope, which results in much better convergence This is the output with starting range -3,3.

```
FalsePositionWithHalving Method:
Iteration: 1 a: -3.000000000000000 b: 3.000  Error: 1.000100000000000
f(b) halved
Iteration: 2 a: -2.320116467223012 b: 3.000  Error: 0.226627844258996
Iteration: 3 a: -1.461355281201203 b: 3.000  Error: 0.370137102233353
Iteration: 4 a: -1.031911727865324 b: 3.000  Error: 0.293866631106219
Iteration: 5 a: -0.808976186952113 b: 3.000  Error: 0.216041290057232
Iteration: 6 a: -0.692637246193324 b: 3.000  Error: 0.143810093096938
Iteration: 7 a: -0.632101361437690 b: 3.000  Error: 0.087399118497213
Iteration: 8 a: -0.600711080401952 b: 3.000  Error: 0.049660201592260
Iteration: 9 a: -0.584473164295216 b: 3.000  Error: 0.027031157966773
Iteration: 10 a: -0.576085406877686 b: 3.000  Error: 0.014350970976819
Iteration: 11 a: -0.571756087262574 b: 3.000  Error: 0.007515065584765
Iteration: 12 a: -0.569522458048753 b: 3.000  Error: 0.003906612038911
Iteration: 13 a: -0.568370313878057 b: 3.000  Error: 0.002023000418006
Iteration: 14 a: -0.567776086323584 b: 3.000  Error: 0.001045493650817
Iteration: 15 a: -0.567469626919079 b: 3.000  Error: 0.000539753983809
Iteration: 16 a: -0.567311582273079 b: 3.000  Error: 0.000278507674249
Iteration: 17 a: -0.567230078120004 b: 3.000  Error: 0.000143667352514
Iteration: 18 a: -0.567188046499171 b: 3.000  Error: 0.000074099774420
```

Problem 1. Bottom line: Bisection method is reliable, but slow.

(a) We need to find the roots of $x - \tan x$. The bisection method is very slow. Accuracy increases only by double each time. It was necessary to graph the function in order to find and interval that contained a root, but not a discontinuity. The bisection method also locates the discontinuities.

```
Bisection Method
Iteration: 0, a: 4.30000000, b: 4.60000000, p: 4.45000000  error = 100.00000000
Iteration: 1, a: 4.45000000, b: 4.60000000, p: 4.52500000  error = 0.01685393
Iteration: 2, a: 4.45000000, b: 4.52500000, p: 4.48750000  error = 0.00828729
Iteration: 3, a: 4.48750000, b: 4.52500000, p: 4.50625000  error = 0.00417827
Iteration: 4, a: 4.48750000, b: 4.50625000, p: 4.49687500  error = 0.00208044
Iteration: 5, a: 4.48750000, b: 4.49687500, p: 4.49218750  error = 0.00104239
Iteration: 6, a: 4.49218750, b: 4.49687500, p: 4.49453125  error = 0.00052174
Iteration: 7, a: 4.49218750, b: 4.49453125, p: 4.49335937  error = 0.00026073
Iteration: 8, a: 4.49335937, b: 4.49453125, p: 4.49394531  error = 0.00013040
Iteration: 9, a: 4.49335937, b: 4.49394531, p: 4.49365234  error = 0.00006519
```

(b) We approximate $25^(1/3)$ by finding the roots of $f(x) = x^3 - 25$. We choose starting conditions at approximately $x = 3$. This method also is slow.

```
Bisection Method
Iteration: 0, a: 2.50000000, b: 3.00000000, p: 2.75000000  error = 100.00000000
Iteration: 1, a: 2.75000000, b: 3.00000000, p: 2.87500000  error = 0.04545455
Iteration: 2, a: 2.87500000, b: 3.00000000, p: 2.93750000  error = 0.02173913
Iteration: 3, a: 2.87500000, b: 2.93750000, p: 2.90625000  error = 0.01063830
Iteration: 4, a: 2.90625000, b: 2.93750000, p: 2.92187500  error = 0.00537634
Iteration: 5, a: 2.92187500, b: 2.93750000, p: 2.92968750  error = 0.00267380
Iteration: 6, a: 2.92187500, b: 2.92968750, p: 2.92578125  error = 0.00133333
Iteration: 7, a: 2.92187500, b: 2.92578125, p: 2.92382812  error = 0.00066756
Iteration: 8, a: 2.92382812, b: 2.92578125, p: 2.92480469  error = 0.00033400
Iteration: 9, a: 2.92382812, b: 2.92480469, p: 2.92431641  error = 0.00016694
Iteration: 10, a: 2.92382812, b: 2.92431641, p: 2.92407227  error = 0.00008349
```

Problem 3. We need to solve $2 * \sin(\pi x) + x = 0$ for x in such a way that it's derivative is likely to be less than 1. The obvious choice of $g(x) = -2\sin(\pi x)$ doesn't converge. The next obvious choice for g is $\sin^{-1}(-x/2)/\pi$.