



University of Colorado
Boulder

HW6

Jesse Hettleman
APPM 4600

COLLEGE OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF APPLIED MATH

October 11, 2024

1 Problem 1

In this problem, we will use two quasi Newton methods—Lazy Newton (Chord) and Broyden—in addition to the original Newton method to solve the following system:

$$f(x, y) = x^2 + y^2 - 4 = 0 \quad (1)$$

$$g(x, y) = e^x + y - 1 \quad (2)$$

Using the three initial guesses $[(1, 1), (1, -1), (0, 0)]$, we find that in general Newton has the shortest run time, although Broyden is quite close. For the first initial guess of $(1, 1)$, both Newton and Broyden converged to the root $(-1.81626407, 0.8373678)$, whereas Lazy Newton reached the maximum number of iterations and failed to converge. For this initial guess, Newton took 0.000316 seconds to converge on average, while Broyden took 0.000468 seconds on average. Newton took 7 iterations, while Broyden took 12. So even though Broyden took more iterations, it was more efficient per iteration.

For the initial guess of $(1, -1)$ we see similar results. This time all three methods converged to the root $(1.00416874, -1.72963729)$. On average, this took Newton 0.000213 seconds, 0.000485 seconds for Lazy Newton, and 0.000224 for Broyden. It also took 5, 36, and 6 iterations respectively.

The final initial guess $(0, 0)$ did not converge for any of these methods due to the fact that the Jacobian matrix evaluated at this point is not invertible.

In conclusion, due to the fact that the Broyden method is more efficient per iteration, I would argue that its performance is better than the Newton method. This would be especially true if the system of equations was larger and it was more difficult to compute the inverse of the Jacobian each iteration.

All code for this problem can be found in 3

2 Problem 2

In this problem, we are asked to approximate solutions to the following nonlinear system using Newton's method, Steepest descent method, and a hybrid of the two:

$$x + \cos(xyz) - 1 = 0 \quad (3)$$

$$(1 - x)^{1/4} + y + 0.05z^2 - 0.15z - 1 = 0 \quad (4)$$

$$-x^2 - 0.1y^2 + 0.01y + z - 1 = 0 \quad (5)$$

Using $(-0.5, 1.5, 0.5)$ as the initial guess results in the following performance metrics for each method:

Figure 1: Estimates, Average Runtimes, and Iterations by Method

```
Newton Method:

[-4.04339648e-17  1.00000000e-01  1.00000000e+00]
Newton: the error message reads: 0
Newton: took this many seconds: 0.00015019893646240235
Newton: number of iterations is: 4

Steepest Descent Method:

[8.08086373e-05  1.00021341e-01  1.00001962e+00]
Steepest: g evaluated at this point is: 6.9064621420546366e-09
Steepest: the error message reads: 0
Steepest: took this many seconds: 0.0009596967697143555
Steepest: number of iterations is: 7

Hybrid Method:

[-4.15791533e-17  1.00000000e-01  1.00000000e+00]
Hybrid: the error message reads: 0
Hybrid: took this many seconds: 0.00032196044921875
Hybrid: number of iterations is: 5
```

Evidently, Newton's method took the least amount of time to run on average, as well as the fewest iterations. However, this may just be a circumstantial result based on the specific initial guess. If the initial guess was outside of the basin of convergence, I hypothesize that the hybrid method would take the least amount of time and/or iterations because the steepest descent method could help get the guess closer to the basin of convergence before applying the Newton method.

Further, I hypothesize that we see the steepest descent method taking the longest because it can only have linear convergence at best. This is analogous to how the bisection method can also only be linear at best. On the other hand, the Newton method has quadratic convergence, which is why the Newton and Hybrid methods take fewer iterations and less time overall.

3 Appendix

3.1 Code

```
1 import numpy as np
2 import math
3 import time
4 from numpy.linalg import inv
5 from numpy.linalg import norm
6
7 # -----Question 1-----
8
9 def evalF1(x):
10
```

```

11     F = np.zeros(2)
12
13     F[0] = (x[0]**2) + (x[1]**2) - 4
14     F[1] = math.e**x[0] + x[1] - 1
15
16     return F
17
18 def evalJ1(x):
19
20     J = np.array([[2*x[0], 2*x[1]],
21                   [math.e**x[0], 1]])
22     return J
23
24
25 def Newton1(x0,tol,Nmax):
26
27     ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
28     ''' Outputs: xstar= approx root, ier = error message, its = num ...
29         its'''
29
30     for its in range(Nmax):
31         J = evalJ1(x0)
32         Jinv = inv(J)
33         F = evalF1(x0)
34
35         x1 = x0 - Jinv.dot(F)
36
37         if (norm(x1-x0) < tol):
38             xstar = x1
39             ier =0
40             return[xstar, ier, its]
41
42         x0 = x1
43
44     xstar = x1
45     ier = 1
46     return[xstar,ier,its]
47
48 def LazyNewton1(x0,tol,Nmax):
49
50     ''' Lazy Newton = use only the inverse of the Jacobian for ...
51         initial guess'''
51     ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
52     ''' Outputs: xstar= approx root, ier = error message, its = num ...
53         its'''
53
54     J = evalJ1(x0)
55     Jinv = inv(J)
56     for its in range(Nmax):
57
58         F = evalF1(x0)
59         x1 = x0 - Jinv.dot(F)
60
61         if (norm(x1-x0) < tol):
62             xstar = x1
63             ier =0
64             return[xstar, ier,its]
65

```

```

66         x0 = x1
67
68     xstar = x1
69     ier = 1
70     return[xstar,ier,its]
71
72 def Broyden1(x0,tol,Nmax):
73     '''tol = desired accuracy
74     Nmax = max number of iterations'''
75
76     '''Sherman-Morrison
77      $(A+xy^T)^{-1} = A^{-1} - 1/p * (A^{-1}xy^TA^{-1})$ 
78     where  $p = 1+y^TA^{-1}Ax$ '''
79
80     '''In Newton
81      $x_{k+1} = x_k - (G(x_k))^{-1} * F(x_k)$ '''
82
83
84     '''In Broyden
85      $x = [F(x_k) - F(x_{k-1}) - \hat{G}_{k-1}(x_k - x_{k-1})$ 
86      $y = x_k - x_{k-1} / ||x_k - x_{k-1}||^2$ '''
87
88     ''' implemented as in equation (10.16) on page 650 of text'''
89
90     '''initialize with 1 newton step'''
91
92     A0 = evalJ1(x0)
93
94     v = evalF1(x0)
95     A = np.linalg.inv(A0)
96
97     s = -A.dot(v)
98     xk = x0+s
99     for its in range(Nmax):
100         '''(save v from previous step)'''
101         w = v
102         ''' create new v'''
103         v = evalF1(xk)
104         ''' $y_k = F(x_k) - F(x_{k-1})$ '''
105         y = v-w;
106         ''' $-A_{k-1}^{-1}y_k$ '''
107         z = -A.dot(y)
108         '''  $p = s_k^t A_{k-1}^{-1}y_k$ '''
109         p = -np.dot(s,z)
110         u = np.dot(s,A)
111         '''  $A = A_k^{-1}$  via Morrison formula'''
112         tmp = s+z
113         tmp2 = np.outer(tmp,u)
114         A = A+1./p*tmp2
115         '''  $-A_k^{-1}F(x_k)$ '''
116         s = -A.dot(v)
117         xk = xk+s
118         if (norm(s)<tol):
119             alpha = xk
120             ier = 0
121             return[alpha,ier,its]
122     alpha = xk
123     ier = 1

```

```

124     return[alpha,ier,its]
125
126 def question1():
127
128     print('Question 1:\n')
129
130     f = lambda x,y: 3*(x**2) - (y**2)
131     g = lambda x,y: 3*x*(y**2) - (x**3) - 1
132
133     fs = np.array([f,g])
134
135     x0s = np.array([[1, 1],[1, -1],[0, 0]])
136
137     Nmax = 100
138     tol = 1e-10
139
140     for x0 in x0s:
141
142         print('\nMethods for initial guess:',x0,'\n')
143
144         if x0[0] == 0 and x0[1] == 0:
145             print("ERROR: Initial Guess Causes Non-Invertible ...
146                 Jacobian\n")
147             break
148
149         t = time.time()
150         for j in range(50):
151             [xstar,ier,its] = Newton1(x0,tol,Nmax)
152             elapsed = time.time()-t
153             print(xstar)
154             print('Newton: the error message reads:',ier)
155             print('Newton: took this many seconds:',elapsed/50)
156             print('Netwon: number of iterations is:',its)
157
158         t = time.time()
159         for j in range(20):
160             [xstar,ier,its] = LazyNewton1(x0,tol,Nmax)
161             elapsed = time.time()-t
162             print(xstar)
163             print('Lazy Newton: the error message reads:',ier)
164             print('Lazy Newton: took this many seconds:',elapsed/20)
165             print('Lazy Newton: number of iterations is:',its)
166
167         t = time.time()
168         for j in range(20):
169             [xstar,ier,its] = Broyden1(x0, tol,Nmax)
170             elapsed = time.time()-t
171             print(xstar)
172             print('Broyden: the error message reads:',ier)
173             print('Broyden: took this many seconds:',elapsed/20)
174             print('Broyden: number of iterations is:',its)
175
176     question1()
177
178     # -----Question 2-----
179
180     def evalF2(x):

```

```

181     F = np.zeros(3)
182
183     F[0] = x[0] + math.cos(x[0]*x[1]*x[2]) - 1
184     F[1] = (1-x[0])**2 + x[1] + 0.05*(x[2]**2) - 0.15*x[2] - 1
185     F[2] = -(x[0]**2) - 0.1*(x[1]**2) + 0.01*x[1] + x[2] - 1
186     return F
187
188 def evalJ2(x):
189
190
191     J = np.array([[1 - x[1]*x[2]*math.sin(x[0]*x[1]*x[2]), ...
192                  (-1)*x[0]*x[2]*math.sin(x[0]*x[1]*x[2]), ...
193                  (-1)*x[0]*x[1]*math.sin(x[0]*x[1]*x[2])],
194                  [(-1/4)*(1-x[0])**(-3/4), 1, 0.1*x[2] - 0.15],
195                  [-2*x[0], -0.2*x[1] + 0.01, 1]])
196
197     return J
198
199 def Newton2(x0,tol,Nmax):
200
201     ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
202     ''' Outputs: xstar= approx root, ier = error message, its = num ...
203         its'''
204
205     for its in range(Nmax):
206         J = evalJ2(x0)
207         Jinv = inv(J)
208         F = evalF2(x0)
209
210         x1 = x0 - Jinv.dot(F)
211
212         if (norm(x1-x0) < tol):
213             xstar = x1
214             ier = 0
215             return[xstar, ier, its]
216
217         x0 = x1
218
219     xstar = x1
220     ier = 1
221     return[xstar,ier,its]
222
223 def evalg(x):
224
225     F = evalF2(x)
226     g = F[0]**2 + F[1]**2 + F[2]**2
227     return g
228
229 def eval_gradg(x):
230     F = evalF2(x)
231     J = evalJ2(x)
232
233     gradg = np.transpose(J).dot(F)
234     return gradg
235
236 def SteepestDescent(x,tol,Nmax):
237
238     count = 0

```

```

236
237     for its in range(Nmax):
238         g1 = evalg(x)
239         z = eval_gradg(x)
240         z0 = norm(z)
241
242         if z0 == 0:
243             print("zero gradient")
244             z = z/z0
245             alpha1 = 0
246             alpha3 = 1
247             dif_vec = x - alpha3*z
248             g3 = evalg(dif_vec)
249
250             while g3>=g1:
251                 alpha3 = alpha3/2
252                 dif_vec = x - alpha3*z
253                 g3 = evalg(dif_vec)
254
255             if alpha3<tol:
256                 print("no likely improvement")
257                 ier = 0
258                 count += 1
259                 return [x,g1,ier,count]
260
261             alpha2 = alpha3/2
262             dif_vec = x - alpha2*z
263             g2 = evalg(dif_vec)
264
265             h1 = (g2 - g1)/alpha2
266             h2 = (g3-g2)/(alpha3-alpha2)
267             h3 = (h2-h1)/alpha3
268
269             alpha0 = 0.5*(alpha2 - h1/h3)
270             dif_vec = x - alpha0*z
271             g0 = evalg(dif_vec)
272
273             if g0<=g3:
274                 alpha = alpha0
275                 gval = g0
276
277             else:
278                 alpha = alpha3
279                 gval =g3
280
281             x = x - alpha*z
282
283             if abs(gval - g1)<tol:
284                 ier = 0
285                 count += 1
286                 return [x,gval,ier,count]
287
288             count += 1
289
290     print('max iterations exceeded')
291     ier = 1
292     return [x,g1,ier,count]
293

```



```

294 def Hybrid(x0,tol,Nmax):
295
296     steepest_return = SteepestDescent(x0,5e-2,Nmax)
297     steepest_guess = steepest_return[0]
298
299     newton_return = Newton2(steepest_guess,tol,Nmax)
300     its = steepest_return[3] + newton_return[2]
301
302     return [newton_return[0],newton_return[1],its]
303
304 def question2():
305
306     print('Question 2:\n')
307
308     x0 = np.array([-0.5, 1.5, .5])
309
310     Nmax = 100
311     tol = 1e-6
312
313     print("\nNewton Method:\n")
314
315     t = time.time()
316     for j in range(50):
317         [xstar,ier,its] = Newton2(x0,tol,Nmax)
318     elapsed = time.time()-t
319     print(xstar)
320     print('Newton: the error message reads:',ier)
321     print('Newton: took this many seconds:',elapsed/50)
322     print('Netwon: number of iterations is:',its)
323
324     print("\nSteepest Descent Method:\n")
325
326     t = time.time()
327     for j in range(50):
328         [xstar,gval,ier,its] = SteepestDescent(x0,tol,Nmax)
329     elapsed = time.time()-t
330     print(xstar)
331     print("Steepest: g evaluated at this point is:", gval)
332     print("Steepest: the error message reads:", ier )
333     print('Steepest: took this many seconds:',elapsed/50)
334     print('Steepest: number of iterations is:',its)
335
336     print("\nHybrid Method:\n")
337
338     t = time.time()
339     for j in range(50):
340         [xstar,ier,its] = Hybrid(x0,tol,Nmax)
341     elapsed = time.time()-t
342     print(xstar)
343     print('Hybrid: the error message reads:',ier)
344     print('Hybrid: took this many seconds:',elapsed/50)
345     print('Hybrid: number of iterations is:',its)
346
347     print('\n')
348
349
350
351 question2()

```

NOTE: ALL CODE FOR THIS ASSIGNMENT IS ORIGINAL, WRITTEN BY JESSE HETTLER-MAN