

# MATH 417 502

## Homework 3

Keegan Smith

September 9, 2024

### Problem 1

a.) Our system of equations can be re-written as:

$$\begin{aligned}x_1 + 2x_2 + 3x_3 - \lambda x_1 &= 0 \\4x_1 + 5x_2 + 6x_3 - \lambda x_2 &= 0 \\7x_1 + 8x_2 + 10x_3 - \lambda x_3 &= 0 \\x_1^2 + x_2^2 + x_3^2 - 1 &= 0\end{aligned}$$

The jacobian of this system is:

$$\begin{bmatrix}1 - \lambda & 2 & 3 & -\lambda x_1 \\4 & 5 - \lambda & 6 & -x_2 \\7 & 8 & 10 - \lambda & -x_3 \\2x_1 & 2x_2 & 2x_3 & 0\end{bmatrix}$$

Thus the Newton iteration looks like:

$$x^{n+1} = x^n - \begin{bmatrix}1 - \lambda & 2 & 3 & -\lambda x_1^n \\4 & 5 - \lambda & 6 & -x_2^n \\7 & 8 & 10 - \lambda & -x_3^n \\2x_1^n & 2x_2^n & 2x_3^n & 0\end{bmatrix}^{-1} \begin{bmatrix}x_1^n + 2x_2^n + 3x_3^n - \lambda x_1^n \\4x_1^n + 5x_2^n + 6x_3^n - \lambda x_2^n \\7x_1^n + 8x_2^n + 10x_3^n - \lambda x_3^n \\(x_1^n)^2 + (x_2^n)^2 + (x_3^n)^2 - 1\end{bmatrix}$$

Essentially we will pick an initial vector  $x_0$  and plug this value into our equation above to get the next vector  $x^{n+1}$ . We continuously do this until we are pretty close to 0. We repeat the process for multiple  $x_0$  until we find all of our solutions.

b.) My program found the following solutions to the system of equations:

Solution 0: [-0.22464024 -0.59734221 -1.06500369 16.65147157]

Solution 1: [ 1.04516341 -0.32819325 -0.41221205 -1.18226152]

Solution 2: [ 0.39884723 -1.03663168 0.58667158 0.12967112]  
where the first 3 values are eigenvectors  $x$  and the last value is the corresponding eigenvalue  $\lambda$ . Thus we have the eigenvectors:

$$\begin{bmatrix} -0.22464024 \\ -0.59734221 \\ -1.06500369 \end{bmatrix}, \begin{bmatrix} 1.04516341 \\ -0.32819325 \\ -0.41221205 \end{bmatrix}, \begin{bmatrix} 0.39884723 \\ -1.03663168 \\ 0.58667158 \end{bmatrix}$$

And their respective eigenvalues:

16.6515, -1.1823, 0.1297

my code to accomplish this is below:

```

1 import numpy as np
2 import random
3 from concurrent.futures import ThreadPoolExecutor,
  as_completed
4 import threading
5 import copy
6 NUM_ITER = 500
7 global_solutions = []
8 lock = threading.Lock()
9 def compute_jacobian_matrix(x_vector):
10     my_jacobian = np.array([[0, 2, 3, 0], [4, 0, 6, 0],
11                             [7, 8, 0, 0], [0, 0, 0, 0]])
12
13     my_jacobian[0][0] = 1 - x_vector[3][0]
14     my_jacobian[0][3] = -x_vector[0][0]
15     my_jacobian[1][1] = 5 - x_vector[3][0]
16     my_jacobian[1][3] = -x_vector[1][0]
17     my_jacobian[2][2] = 10 - x_vector[3][0]
18     my_jacobian[2][3] = -x_vector[2][0]
19     my_jacobian[3][0] = 2 * x_vector[0][0]
20     my_jacobian[3][1] = 2 * x_vector[1][0]
21     my_jacobian[3][2] = 2 * x_vector[2][0]
22     return my_jacobian
23 def compute_f(x_vector):
24     my_f = np.array([[0], [0], [0], [0]])
25
26     my_f[0][0] = x_vector[0][0] + 2 * x_vector[1][0] + 3
27     * x_vector[2][0] - x_vector[3][0] * x_vector
28     [0][0]
29     my_f[1][0] = 4 * x_vector[0][0] + 5 * x_vector[1][0]
30     + 6 * x_vector[2][0] - x_vector[3][0] *
31     x_vector[1][0]
32     my_f[2][0] = 7 * x_vector[0][0] + 8 * x_vector[1][0]
33     + 10 * x_vector[2][0] - x_vector[3][0] *
34     x_vector[2][0]
35     my_f[3][0] = x_vector[0][0]**2 + x_vector[1][0]**2 +
36     x_vector[2][0]**2 - 1

```

```
29     return my_f
30 def iterate(initial_x):
31     for i in range(0, NUM_ITER):
32         jacobian = compute_jacobian_matrix(initial_x)
33         my_f = compute_f(initial_x)
34         initial_x = initial_x + np.linalg.solve(jacobian
35             , -my_f)
36     return initial_x
37 def perform_iteration(i, start, end):
38     #print("got here")
39     j = random.uniform(start, end)
40     k = random.uniform(start, end)
41     l = random.uniform(start, end)
42     m = random.uniform(start, end)
43     try:
44         result = tuple(iterate(np.array([[j], [k], [l],
45             [m]])).flatten())
46
47         return result
48     except Exception as e:
49         return None
50 if __name__ == "__main__":
51     start = -10
52     end = 10
53     num_attempts = 10000
54     with ThreadPoolExecutor() as executor:
55         futures = [executor.submit(perform_iteration, i,
56             start, end) for i in range(num_attempts)]
57         for future in as_completed(futures):
58             result = future.result()
59             if(result):
60                 with lock:
61                     global_solutions.append(result)
62 while(True):
63     result = []
64     for i in range(0, 3):
65         result.append(np.array(global_solutions[
66             random.randint(0, len(global_solutions))
67             ]))
68     result = np.array(result)
69     eigenvectors = []
70     for i in range(0, len(result)):
71         eigenvectors.append(result[i][:3])
72     determinant = np.linalg.det(eigenvectors)
73     if(not (determinant <= 10**(-2) and determinant
74         >= -10**(-2))):
75         for i in range(0, len(result)):
76             print(f"Solution {i}: ", result[i])
77     print("determinant was: ", determinant)
78     break;
```

```
73 | print("determinant was approx. 0, trying again")
```

## Problem 2

To find the minimum of  $f(x)$  we will need to find all of the potential local minima, e.g where  $\nabla f(x) = 0$ . Thus we will analytically solve for  $\nabla f(x)$  and then use newton iteration to solve for  $\nabla f(x) = 0$ . We will pick the local minima which makes  $f(x)$  the smallest.

$$\nabla f(x) = \begin{bmatrix} x_1^3 + x_2x_3 - 2x_1x_2 \\ x_2 + x_1x_3 - x_1^2 \\ x_3 + x_1x_2 \end{bmatrix} = 0$$

To use Newton's method, we will need to calculate the Jacobian of  $\nabla f(x)$ :

$$Jf(x) = \begin{bmatrix} 3x_1^2 - 2x_2 & x_3 - 2x_1 & x_2 \\ x_3 - 2x_1 & 1 & x_1 \\ x_2 & x_1 & 1 \end{bmatrix}$$

Recall Newton's scheme:

$$x^{n+1} = x^n - Jf(x^n)^{-1}f(x^n)$$

$$Jf(x^n)(x^{n+1} - x^n) = -f(x^n)$$

Implemented in python:

```
1 import numpy as np
2 import random
3 from multiprocessing import Process, Manager
4 NUM_ITER = 500
5 def compute_jacobian_matrix(x_vector):
6     my_jacobian = np.array([[0, 0, 0], [0, 1, 0], [0, 0,
7         1]])
8     x_1 = x_vector[0][0]
9     x_2 = x_vector[1][0]
10    x_3 = x_vector[2][0]
11    my_jacobian[0][0] = 3 * x_1**2 - 2*x_2
12    my_jacobian[0][1] = x_3 - 2 * x_1
13    my_jacobian[0][2] = x_2
14    my_jacobian[1][0] = x_3 - 2 * x_1
15    my_jacobian[1][2] = x_1
16    my_jacobian[2][0] = x_2
17    my_jacobian[2][1] = x_1
18    return my_jacobian
19 def compute_f(x_vector):
```

```

19 my_f = np.array([[0], [0], [0]])
20 x_1 = x_vector[0][0]
21 x_2 = x_vector[1][0]
22 x_3 = x_vector[2][0]
23 my_f[0][0] = x_1**3 + x_2 * x_3 - 2 * x_1 * x_2
24 my_f[1][0] = x_2 + x_1 * x_3 - x_1**2
25 my_f[2][0] = x_3 + x_1 * x_2
26 return my_f
27 def iterate(initial_x):
28     for i in range(0, NUM_ITER):
29         jacobian = compute_jacobian_matrix(initial_x)
30         my_f = compute_f(initial_x)
31         initial_x = initial_x + np.linalg.solve(jacobian, -
            my_f)
32     return initial_x
33 def perform_iteration(i, start, end, batch_size, lock,
    global_solutions):
34     results = []
35     for i in range(0, batch_size):
36         j = random.uniform(start, end)
37         k = random.uniform(start, end)
38         l = random.uniform(start, end)
39         try:
40             result = tuple(iterate(np.array([[j], [k], [l]]))
                .flatten())
41
42             results.append(result)
43         except Exception as e:
44             continue
45     with lock:
46         global_solutions.extend(results)
47 def f(x_tuple):
48     x_1 = x_tuple[0]
49     x_2 = x_tuple[1]
50     x_3 = x_tuple[2]
51     return 1/4 * x_1**4 + 1/2 * x_2**2 + 1/2 * x_3**2 + x_1*
        x_2*x_3 - (x_1)**2 * x_2
52 if __name__ == "__main__":
53     start = -1
54     end = 2
55     num_cores = 192
56     batch_size = 20000
57     num_attempts = num_cores
58     processes = []
59     manager = Manager()
60     global_solutions = manager.list()
61     lock = manager.Lock()
62     for i in range(0, num_cores):
63         p = Process(target = perform_iteration, args=(i,
            start, end, batch_size, lock, global_solutions))

```

```

64     p.start()
65     processes.append(p)
66 for process in processes:
67     process.join()
68     smallest = 999999999
69     smallest_sol = None
70 for solution in global_solutions:
71     result = f(solution)
72     if(result < smallest):
73         smallest_sol = solution
74         smallest = result
75
76 print(f"smallest value obtained was {smallest} at {
    smallest_sol}")

```

I ran this on the LAUNCH cluster with 192 cores and achieved a minimum of -3.0595292600760775 with vector:

$$\begin{bmatrix} 1.7945995784244926 \\ 1.9367735174859102 \\ -0.3935949441846148 \end{bmatrix}$$

### Problem 3

a.) The distance from a point  $x$  from  $\hat{x}$  can be given by:

$$\begin{aligned} f(x) &= \sqrt{(x_1 - 0)^2 + (x_2 - \frac{1}{2})^2} \\ &= \sqrt{x_1^2 + (x_2 - \frac{1}{2})^2} \end{aligned}$$

Since we are just minimizing  $f(x)$ , and  $f(x)$  is non-negative, finding an  $x$  which minimizes  $f(x)^2$  will be equivalent to the  $x$  which minimizes  $f(x)$ . Thus for simplicity we will write:

$$f(x) = x_1^2 + (x_2 - \frac{1}{2})^2$$

We are minimizing  $f(x)$  with constraints, thus we will use lagrange multipliers:

$$\begin{aligned} \nabla f(x) - \lambda \nabla g(x) &= 0 \\ g(x) &= 0 \end{aligned}$$

We calculate  $\nabla f(x)$ :

$$\nabla f(x) = \begin{bmatrix} 2x_1 \\ 2x_2 - 1 \end{bmatrix}$$

$\nabla g(x)$ :

$$\nabla g(x) = \begin{bmatrix} 3x_1^2 - 1 \\ -2x_2 \end{bmatrix}$$

plugging these into the lagrange system we had above, we get:

$$\begin{bmatrix} 2x_1 - \lambda(3x_1^2 - 1) \\ 2x_2 - 1 - \lambda(-2x_2) \\ x_1^3 - x_1 + \frac{1}{2} - x_2^2 \end{bmatrix} = 0$$

three equations, three unknowns.

- b.) Now we need to setup the newton iteration. We will start by finding the jacobian of the above system:

$$J = \begin{bmatrix} 2 - 6\lambda x_1 & 0 & -3x_1^2 + 1 \\ 0 & 2 + 2\lambda & 2x_2 \\ 3x_1^2 - 1 & -2x_2 & 0 \end{bmatrix}$$

Thus newton's method looks like:

$$x_{n+1} = x_n - \begin{bmatrix} 2 - 6\lambda x_{n1} & 0 & -3x_{n1}^2 + 1 \\ 0 & 2 + 2\lambda & 2x_{n2} \\ 3x_{n1}^2 - 1 & -2x_{n2} & 0 \end{bmatrix}^{-1} \begin{bmatrix} 2x_{n1} - \lambda(3x_{n1}^2 - 1) \\ 2x_{n2} - 1 - \lambda(-2x_{n2}) \\ x_{n1}^3 - x_{n1} + \frac{1}{2} - x_{n2}^2 \end{bmatrix}$$

- c.) I ran this on the cluster and obtained 0.16673076880672957 at vector:

$$\begin{bmatrix} 0.10116225679070456 \\ 0.6325343241122641 \\ -0.06750779368628224 \end{bmatrix}$$

Code is below:

```

1 import numpy as np
2 import random
3 from multiprocessing import Process, Manager
4 NUM_ITER = 1000
5 vector_ranges = [
6     [-10, 10],
7     [-10, 10],
8     [-10, 10]
9 ]
10 def compute_jacobian_matrix(x_vector):
11     my_jacobian = np.array([[0, 0, 0], [0, 0, 0], [0, 0,
12     0]])
13     x_1 = x_vector[0][0]
```

```

13     x_2 = x_vector[1][0]
14     x_3 = x_vector[2][0]
15     my_jacobian[0][0] = 2 - 6 * x_1 * x_3
16     my_jacobian[0][2] = -3 * x_2**2 + 1
17     my_jacobian[1][1] = 2 + 2 * x_3
18     my_jacobian[1][2] = 2 * x_2
19     my_jacobian[2][0] = 3 * x_1**2 - 1
20     my_jacobian[2][1] = -2 * x_2
21     return my_jacobian
22 def compute_f(x_vector):
23     my_f = np.array([[0], [0], [0]])
24     x_1 = x_vector[0][0]
25     x_2 = x_vector[1][0]
26     x_3 = x_vector[2][0]
27     my_f[0][0] = 2 * x_1 - x_3 * (3*x_1**2 - 1)
28     my_f[1][0] = 2 * x_2 - 1 - x_3 * (-2 * x_2)
29     my_f[2][0] = x_1**3 - x_1 + 1/2 - x_2**2
30     return my_f
31 def iterate(initial_x):
32     for i in range(0, NUM_ITER):
33         jacobian = compute_jacobian_matrix(initial_x)
34         my_f = compute_f(initial_x)
35         initial_x = initial_x + np.linalg.solve(jacobian
36             , -my_f)
37     return initial_x
38 def perform_iteration(i, start, end, batch_size, lock,
39     global_solutions):
40     results = []
41     for i in range(0, batch_size):
42         j = random.uniform(vector_ranges[0][0],
43             vector_ranges[0][1])
44         k = random.uniform(vector_ranges[1][0],
45             vector_ranges[1][1])
46         l = random.uniform(vector_ranges[2][0],
47             vector_ranges[2][1])
48         try:
49             result = tuple(iterate(np.array([[j], [k], [
50                 l]])).flatten())
51             results.append(result)
52         except Exception as e:
53             continue
54     with lock:
55         global_solutions.extend(results)
56 def f(x_tuple):
57     x_1 = x_tuple[0]
58     x_2 = x_tuple[1]
59     return (x_1**2 + (x_2 - 1/2)**2)**(1/2)
60 def g(x_tuple):
61     x_1 = x_tuple[0]

```



```
57     x_2 = x_tuple[1]
58     return x_1**3 - x_1 + 1/2 - x_2**2
59 if __name__ == "__main__":
60     start = -100
61     end = 100
62     num_cores = 192
63     batch_size = 50000
64     num_attempts = num_cores
65     processes = []
66     manager = Manager()
67     global_solutions = manager.list()
68     lock = manager.Lock()
69     for i in range(0, num_cores):
70         p = Process(target = perform_iteration, args=(i,
71             start, end, batch_size, lock,
72             global_solutions))
73         p.start()
74         processes.append(p)
75     for process in processes:
76         process.join()
77     smallest = 999999999
78     smallest_sol = None
79     for solution in global_solutions:
80         result = f(solution)
81         constraint_value = g(solution)
82         if(constraint_value > 10**(-3) or
83             constraint_value < -10**(-3)):
84             continue
85         if(result < smallest):
86             smallest_sol = solution
87             smallest = result
88
89     print(f"smallest value obtained was {smallest} at {
90         smallest_sol}")
```