MATH 417 502
Homework 3

Keegan Smith

September 9, 2024

Problem 1

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

$$x_1 + 2x_2 + 3x_3 - \lambda x_1 = 0$$
$$4x_1 + 5x_2 + 6x_3 - \lambda x_2 = 0$$
$$7x_1 + 8x_2 + 10x_3 - \lambda x_3 = 0$$
$$x_1^2 + x_2^2 + x_3^2 - 1 = 0$$

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 λ. 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:

```python
import numpy as np
import random
from concurrent.futures import ThreadPoolExecutor,
    as_completed
import threading
import copy
NUM_ITER = 500
global_solutions = []
lock = threading.Lock()
def compute_jacobian_matrix(x_vector):
    my_jacobian = np.array([[0, 2, 3, 0], [4, 0, 6, 0],
        [7, 8, 0, 0], [0, 0 , 0, 0]])

    my_jacobian[0][0] = 1 - x_vector[3][0]
    my_jacobian[0][3] = -x_vector[0][0]
    my_jacobian[1][1] = 5 - x_vector[3][0]
    my_jacobian[1][3] = -x_vector[1][0]
    my_jacobian[2][2] = 10 - x_vector[3][0]
    my_jacobian[2][3] = -x_vector[2][0]
    my_jacobian[3][0] = 2 * x_vector[0][0]
    my_jacobian[3][1] = 2 * x_vector[1][0]
    my_jacobian[3][2] = 2 * x_vector[2][0]
    return my_jacobian
def compute_f(x_vector):
    my_f = np.array([[0], [0], [0], [0]])

    my_f[0][0] = x_vector[0][0] + 2 * x_vector[1][0] + 3
        * x_vector[2][0] - x_vector[3][0] * x_vector
        [0][0]
    my_f[1][0] = 4 * x_vector[0][0] + 5 * x_vector[1][0]
        + 6 * x_vector[2][0] - x_vector[3][0] *
        x_vector[1][0]
    my_f[2][0] = 7 * x_vector[0][0] + 8 * x_vector[1][0]
        + 10 * x_vector[2][0] - x_vector[3][0] *
        x_vector[2][0]
    my_f[3][0] = x_vector[0][0]**2 + x_vector[1][0]**2 +
        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
                , -my_f)
35      return initial_x
36 def perform_iteration(i, start, end):
37      #print("got here")
38      j = random.uniform(start, end)
39      k = random.uniform(start, end)
40      l = random.uniform(start, end)
41      m = random.uniform(start, end)
42      try:
43          result = tuple(iterate(np.array([[j], [k], [l],
                [m]])).flatten())

45          return result
46      except Exception as e:
47          return None
48 if __name__ == "__main__":
49      start = -10
50      end = 10
51      num_attempts = 10000
52      with ThreadPoolExecutor() as executor:
53          futures = [executor.submit(perform_iteration, i,
                start, end) for i in range(num_attempts)]
54          for future in as_completed(futures):
55              result = future.result()
56              if(result):
57                  with lock:
58                      global_solutions.append(result)
59      while(True):
60          result = []
61          for i in range(0, 3):
62              result.append(np.array(global_solutions[
                    random.randint(0, len(global_solutions))
                    ]))
63          result = np.array(result)
64          eigenvectors = []
65          for i in range(0, len(result)):
66              eigenvectors.append(result[i][:3])
67          determinant = np.linalg.det(eigenvectors)
68          if(not (determinant <= 10**(-2) and determinant
                >= -10**(-2))):
69              for i in range(0, len(result)):
70                  print(f"Solution {i}: ", result[i])
71              print("determinant was: ", determinant)
72              break;
```

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

Problem 2

To find the minimum of 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.

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

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:

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

Implemented in python:

```python
import numpy as np
import random
from multiprocessing import Process, Manager
NUM_ITER = 500
def compute_jacobian_matrix(x_vector):
    my_jacobian = np.array([[0, 0, 0], [0, 1, 0], [0, 0,
        1]])
    x_1 = x_vector[0][0]
    x_2 = x_vector[1][0]
    x_3 = x_vector[2][0]
    my_jacobian[0][0] = 3 * x_1**2 - 2*x_2
    my_jacobian[0][1] = x_3 - 2 * x_1
    my_jacobian[0][2] = x_2
    my_jacobian[1][0] = x_3 - 2 * x_1
    my_jacobian[1][2] = x_1
    my_jacobian[2][0] = x_2
    my_jacobian[2][1] = x_1
    return my_jacobian
def compute_f(x_vector):
```

```python
      my_f = np.array([[0], [0], [0]])
      x_1 = x_vector[0][0]
      x_2 = x_vector[1][0]
      x_3 = x_vector[2][0]
      my_f[0][0] = x_1**3 + x_2 * x_3 - 2 * x_1 * x_2
      my_f[1][0] = x_2 + x_1 * x_3 - x_1**2
      my_f[2][0] = x_3 + x_1 * x_2
      return my_f
def iterate(initial_x):
      for i in range(0, NUM_ITER):
          jacobian = compute_jacobian_matrix(initial_x)
          my_f = compute_f(initial_x)
          initial_x = initial_x + np.linalg.solve(jacobian, -
              my_f)
      return initial_x
def perform_iteration(i, start, end, batch_size, lock,
    global_solutions):
      results = []
      for i in range(0, batch_size):
          j = random.uniform(start, end)
          k = random.uniform(start, end)
          l = random.uniform(start, end)
          try:
              result = tuple(iterate(np.array([[j], [k], [l]])
                  ).flatten())

              results.append(result)
          except Exception as e:
              continue
      with lock:
          global_solutions.extend(results)
def f(x_tuple):
      x_1 = x_tuple[0]
      x_2 = x_tuple[1]
      x_3 = x_tuple[2]
      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
if __name__ == "__main__":
      start = -1
      end = 2
      num_cores = 192
      batch_size = 20000
      num_attempts = num_cores
      processes = []
      manager = Manager()
      global_solutions = manager.list()
      lock = manager.Lock()
      for i in range(0, num_cores):
          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:

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

We are trying to minimize this function with respect to $g(x) = 0$, so we
will use Lagrange multipliers and solve the system of equations:

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

We calculate $\nabla f(x)$:

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

$\nabla g(x)$:

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

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

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

three equations, three unknowns.

b.)