APPM 4600 Homework 5

4 October 2024

1. The code used in this question is listed at the end of the question.

Iteration over the system converges to the approximate root

$$\begin{bmatrix} x \\ y \end{bmatrix} \approx \begin{bmatrix} 0.5 \\ 0.8660254037844386 \end{bmatrix}.$$

This convergence is numerically of first order ($\alpha \approx 0.98811$), although it converges to the desired tolerance in fewer iterations than lazy Newton.

(b) The system has Jacobian

$$J(x, y) = \begin{bmatrix} 6x & -2y \\ 3y^2 - 3x^2 & 6xy \end{bmatrix}.$$

At $x = y = 1$, this is

$$J(1, 1) = \begin{bmatrix} 6 & -2 \\ 0 & 6 \end{bmatrix},$$

with inverse

$$J(1, 1)^{-1} = \begin{bmatrix} \frac{1}{6} & \frac{1}{18} \\ 0 & \frac{1}{6} \end{bmatrix}.$$

Thus, the choice of the matrix implements lazy newton's method, where the Jacobian is only evaluated at the initial guess and never updated in the iteration.

(c) Newton's method converges to the approximate root

$$\begin{bmatrix} x \\ y \end{bmatrix} \approx \begin{bmatrix} 0.5 \\ 0.8660254037844387 \end{bmatrix},$$

which is similar to the previous iteration. This convergence is of first order (numerically, $\alpha \approx 1.0078$).

(d) The exact solution is simply

$$\begin{bmatrix} x \\ y \end{bmatrix} \approx \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2}. \end{bmatrix}$$

```python
import numpy as np
import math
import time
from numpy.linalg import inv
from numpy.linalg import norm

def compute_order(x, xstar):
    diff1 = np.abs(x[1::] - xstar)
    diff2 = np.abs(x[0:-1]-xstar)
    fit = np.polyfit(np.log(diff2.flatten()), np.log(diff1.flatten()),1)
    print('the order of the equation is')
    print("lambda = " + str(np.exp(fit[1])))
    print("alpha = " + str(fit[0]))

    alpha = fit[0]
    l = np.exp(fit[1])

    return [fit, alpha, l]
```

```python
def compute_order_ndim(x, xstar):
    diff1 = norm(x[1::] - xstar, axis=1)
    diff2 = norm(x[0:-1]-xstar, axis=1)
    fit = np.polyfit(np.log(diff2.flatten()), np.log(diff1.flatten()),1)
    print('the order of the equation is')
    print("lambda = " + str(np.exp(fit[1])))
    print("alpha = " + str(fit[0]))

    alpha = fit[0]
    l = np.exp(fit[1])

    return [fit, alpha, l]


def evalF(xn):
    return np.array([3*xn[0]**2 - xn[1]**2, 3*xn[0]*xn[1]**2 - xn[0]**3 - 1])

def evalJ(x):
    return np.array([[6*x[0], -2*x[1]],
        [3*x[1]**2 - 3**x[0]**2, 6*x[0]*x[1]]])


def Newton(x0,tol,Nmax):

    ''' inputs: x0 = initial guess, tol = tolerance, Nmax = max its'''
    ''' Outputs: xstar= approx root, ier = error message, its = num its'''
    iters = np.array([x0])
    for its in range(Nmax):
       J = evalJ(x0)
       Jinv = inv(J)
       F = evalF(x0)

       x1 = x0 - Jinv.dot(F)

       iters = np.vstack([iters, x1])

       if (norm(x1-x0) < tol):
           xstar = x1
           ier =0
           return[xstar, ier, its, iters]

       x0 = x1

    xstar = x1
    ier = 1
    return[xstar,ier,its,iters]

def question1_1(Nmax):
    xn = np.array([1, 1])
    x = np.zeros((Nmax, 2))
    for n in range(Nmax):
        x[n] = xn
        xn = xn - np.array([[1/6, 1/18], [0, 1/6]]).dot(np.array(
            [3*xn[0]**2 - xn[1]**2,
            3*xn[0]*xn[1]**2 - xn[0]**3 - 1]))

    print("Question 1(a)")
    print(xn[0])
```

```
    print(xn[1])
    print(x)
    compute_order_ndim(x, xn)

def question1_3():
    [xstar, ier, its, iters] = Newton(np.array([1,1]), 1e-10, 100)
    print("Question_1(c)")
    print("xstar=", xstar, "ier=", ier, "its=", its)
    print(xstar[0])
    print(xstar[1])
    print(iters)
    compute_order_ndim(iters[0:-1], xstar)

question1_1(30)
question1_3()
```

2. We have iterative step

$$\begin{aligned} x_{n+1} = f(x_n, y_n) &= \frac{1}{\sqrt{2}}\sqrt{1+(x_n+y_n)^2} - \frac{2}{3}, \\ y_{n+1} = g(x_n, y_n) &= \frac{1}{\sqrt{2}}\sqrt{1+(x_n-y_n)^2} - \frac{2}{3}. \end{aligned} \tag{1}$$

We are looking for some closed rectangular region where the partial derivatives of f and g are continuous and

$$|f_x| \leq \frac{1}{2},\ |f_y| \leq \frac{1}{2},\ |g_x| \leq \frac{1}{2},\ \text{and } |g_y| \leq \frac{1}{2}. \tag{2}$$

The component functions f and g have partials

$$\begin{aligned} f_x(x,y) = f_y(x,y) &= \frac{x+y}{\sqrt{2}\sqrt{(x+y)^2+1}}, \\ g_x(x,y) = -g_y(x,y) &= \frac{x-y}{\sqrt{2}\sqrt{(x-y)^2+1}}, \end{aligned}$$

which are continuous everywhere in $\mathbb{R}^2$.

The largest region over which (2) is satisfied is $D = [-\frac{1}{2}, \frac{1}{2}] \times [-\frac{1}{2}, \frac{1}{2}]$. By theorem 10.6, any fixed point iteration of (1) with initial guess $(x_0, y_0) \in D$ will converge to some fixed point $\mathbf{p} \in D$.

3. (a) We start at some initial guess $\mathbf{x_0} = (x_0, y_0)$, where $f(\mathbf{x_0}) \neq 0$. We wish to find $f(\mathbf{x}) = 0$, so we want to move in the direction of the gradient of f, which is the line given by

$$\mathbf{x} - \mathbf{x_0} = \alpha \nabla f(\mathbf{x_0}), \tag{3}$$

where $\alpha \in \mathbb{R}$. About the guess f has first order Taylor expansion

$$f(\mathbf{x}) \approx f(\mathbf{x_0}) + (\mathbf{x} - \mathbf{x_0}) \cdot \nabla f(\mathbf{x_0}). \tag{4}$$

Substituting (3) into (4), we have

$$\begin{aligned} f(\mathbf{x}) &\approx f(\mathbf{x_0}) + \alpha \nabla f(\mathbf{x_0}) \cdot \nabla f(\mathbf{x_0}) \\ &= f(\mathbf{x_0}) + \alpha \nabla^2 f(\mathbf{x_0}). \end{aligned}$$

We want $f(\mathbf{x}) = 0$, so we have

$$\alpha = -\frac{f(\mathbf{x_0})}{\nabla^2 f(\mathbf{x_0})}.$$

Substituting this back into (4), we have

$$\mathbf{x} = \mathbf{x_0} - \frac{f(\mathbf{x_0})}{\nabla^2 f(\mathbf{x_0})} \nabla f(\mathbf{x_0}),$$

which gives the iterative step

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n - \frac{f(x_n,y_n) f_x(x_n,y_n)}{f_x(x_n,y_n)^2 + f_y(x_n,y_n)^2} \\ x_y - \frac{f(x_n,y_n) f_y(x_n,y_n)}{f_x(x_n,y_n)^2 + f_y(x_n,y_n)^2} \end{pmatrix},$$

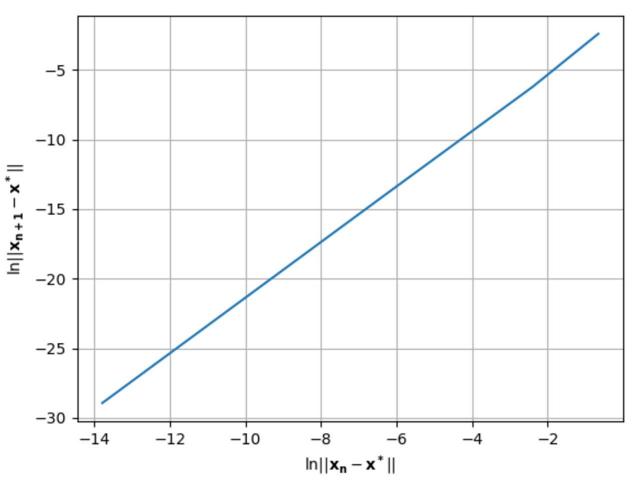
as expected.

4. The code used to answer this question is listed at the end of the question.

We start the iteration at $(1, 1, 1)$ and converge to the point

$$\mathbf{x}^* \approx \begin{pmatrix} 1.09364232 \\ 1.36032838 \\ 1.36032838 \end{pmatrix}.$$

This convergence is indeed second order (numerically, $\alpha \approx 2.013$). This can be seen in the plot of $\ln ||\mathbf{x_{n+1}} - \mathbf{x}^*||$ vs $\ln ||\mathbf{x_n} - \mathbf{x}^*||$ below, where $|| \cdot ||$ is the L^2 norm.



```python
import numpy as np
import math
from numpy.linalg import norm
import matplotlib.pyplot as plt

def f(x):
    return x[0]**2 + 4*x[1]**2+4*x[2]**2 - 16

def grad_f(x):
    return np.array([2*x[0], 8*x[1], 8*x[1]])

def iterate(f, grad_f, x0, tol, Nmax):
    iters = np.array([x0])
    for n in range(Nmax):
        # update step
        laplacian_x0 = np.sum(grad_f(x0)**2)
        x1 = x0 - f(x0) / laplacian_x0 * grad_f(x0)
        # keep track of iterates
        iters = np.vstack([iters, x1])
        # check for bail out
        if norm(x1 - x0) < tol:
            return [x1, 0, iters]

        x0 = x1

    return [x1, 1, iters]

def plot_order(x, xstar):
    diff1 = norm(x[1::] - xstar, axis=1)
    diff2 = norm(x[0:-1]-xstar, axis=1)

    plt.plot(np.log(diff2.flatten()), np.log(diff1.flatten()))
    plt.grid()
    plt.xlabel("$\ln || \mathbf{x_{n}} - \mathbf{x^*} ||$");
    plt.ylabel("$\ln || \mathbf{x_{n+1}} - \mathbf{x^*} ||$");

    fit = np.polyfit(np.log(diff2.flatten()), np.log(diff1.flatten()),1)
    print('the order of the equation is')
    print("lambda = " + str(np.exp(fit[1])))
    print("alpha = " + str(fit[0]))

    alpha = fit[0]
    l = np.exp(fit[1])

    return [fit, alpha, l]


def question3b():
    x0 = np.array([1, 1, 1])
    [xstar, ier, iters] = iterate(f, grad_f, x0, 1e-10, 100)
    print("xstar=", xstar, "ier=", ier)
    plot_order(iters[0:-1], xstar)
    plt.savefig("hw5_3b.png")

question3b()
```