Math 151B Homework 4

Nick Monozon

11.4.3(a)

Consider the boundary value problem

$$y'' = -e^{-2y},\ 1 \leq x \leq 2,\ y(1) = 0,\ y(2) = \ln 2$$

along with actual solution $y(x) = \ln x$. We will approximation the solution to this BVP using the nonlinear finite difference method for $N = 9$ and tolerance 10^{-4}. Using the Python code nonlinear_finite_diff.py, we obtain convergence in 3 iterations and get the below table as output.

i	x_i	w_i	$y(x_i)$
0	1.0	0.00000000	0.00000000
1	1.1	0.09523436	0.09531018
2	1.2	0.18220299	0.18232156
3	1.3	0.26222554	0.26236426
4	1.4	0.33632929	0.33647224
5	1.5	0.40532953	0.40546511
6	1.6	0.46988413	0.47000363
7	1.7	0.53053154	0.53062825
8	1.8	0.58771808	0.58778666
9	1.9	0.64181777	0.64185389
10	2.0	0.69314718	0.69314718

11.4.3(b)

Consider the boundary value problem

$$y'' = y' \cos x - y \ln y,\ 0 \leq x \leq \pi/2,\ y(0) = 1,\ y(\pi/2) = e$$

together with actual solution $y(x) = e^{\sin x}$. We will approximate the solution to this problem using the nonlinear finite difference method with $N = 9$ and tolerance 10^{-4}. Using nonlinear_finite_diff.py, we get convergence in 3 iterations and obtain the outputted table below.

i	x_i	w_i	$y(x_i)$
0	0.00000000	1.00000000	1.00000000
1	0.15707963	1.16942058	1.16933413
2	0.31415927	1.36244080	1.36208552
3	0.47123890	1.57538813	1.57458304
4	0.62831853	1.80138559	1.79999746
5	0.78539816	2.03011674	2.02811498
6	0.94247780	2.24819259	2.24569937
7	1.09955743	2.44026338	2.43758190
8	1.25663706	2.59083695	2.58844295
9	1.41371669	2.68653080	2.68502044
10	1.57079633	2.71828183	2.71828183

10.1.5(a)

Consider $\boldsymbol{G} : D \subset \mathbb{R}^3 \to \mathbb{R}^3$ where

$$\boldsymbol{G}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} (\cos(x_2 x_3) + 0.5)/3 \\ \frac{1}{25}\sqrt{x_1^2 + 0.3125} - 0.03 \\ -\frac{1}{20}e^{-x_1 x_2} - \frac{10\pi - 3}{60} \end{pmatrix} = \begin{pmatrix} g_1(x_1, x_2, x_3) \\ g_2(x_1, x_2, x_3) \\ g_3(x_1, x_2, x_3) \end{pmatrix}$$

and $D = \left\{(x_1, x_2, x_3)^\top : -1 \le x_i \le 1, i = 1,2,3\right\}$. We want to show that $\boldsymbol{G}$ has a unique fixed point on D by applying Theorem 10.6. Hence, we will first show that $\boldsymbol{G}(\boldsymbol{x}) \in D$ for any $\boldsymbol{x} \in D$; that is, $-1 \le g_i(\boldsymbol{x}) \le 1$ for each $i = 1,2,3$. Using Wolfram Alpha, we find that

$$\begin{aligned} 0.346767 \le g_1(x_1, x_2, x_3) \le \frac{1}{2} &\Rightarrow -1 \le g_1(x_1, x_2, x_3) \le 1 \\ -0.007639 \le g_2(x_1, x_2, x_3) \le 0.015826 &\Rightarrow -1 \le g_2(x_1, x_2, x_3) \le 1 \\ -0.609513 \le g_3(x_1, x_2, x_3) \le -0.491993 &\Rightarrow -1 \le g_3(x_1, x_2, x_3) \le 1 \end{aligned}$$

so we have that $G(\boldsymbol{x}) \in D$ for any $\boldsymbol{x} \in D$. Next, we calculate the partials $\partial g_i / \partial x_j$ for each $i = 1,2,3$ and $j = 1,2,3$ and use Wolfram Alpha to find the minimum and maximum for each on D:

$$\begin{aligned} \frac{\partial g_1}{\partial x_1} &= 0 && (\min = 0, \max = 0) \\ \frac{\partial g_1}{\partial x_2} &= -\frac{1}{3}x_3 \sin(x_2 x_3) && (\min = -0.28049, \max = 0.28049) \\ \frac{\partial g_1}{\partial x_3} &= -\frac{1}{3}x_2 \sin(x_2 x_3) && (\min = -0.28049, \max = 0.28049) \\ \frac{\partial g_2}{\partial x_1} &= \frac{x_1}{25\sqrt{x_1^2 + 0.3125}} && (\min = -0.034914, \max = 0.034914) \\ \frac{\partial g_2}{\partial x_2} &= 0 && (\min = 0, \max = 0) \\ \frac{\partial g_2}{\partial x_3} &= 0 && (\min = 0, \max = 0) \\ \frac{\partial g_3}{\partial x_1} &= \frac{x_2 e^{-x_1 x_2}}{20} && (\min = -0.135914, \max = 0.135914) \\ \frac{\partial g_3}{\partial x_2} &= \frac{x_1 e^{-x_1 x_2}}{20} && (\min = -0.135914, \max = 0.135914) \\ \frac{\partial g_3}{\partial x_3} &= 0 && (\min = 0, \max = 0) \end{aligned}$$

Next, we take the absolute value of all maximums and minimums above and find the largest value. By inspection, this is $0.28049 \approx 0.281$ (to avoid any potential roundoff error). We hence have that $K/3 = 0.281$, so $K = 0.281(3) = 0.843 < 1$. Hence, since $\boldsymbol{G}(\boldsymbol{x}) \in D$ and

$$\left|\frac{\partial g_i}{\partial x_j}\right| \le \frac{K}{3}$$

for $i, j \in \{1,2,3\}$ and $K = 0.843 < 1$, we have by Theorem 10.6 that $\boldsymbol{G}(\boldsymbol{x})$ has a unique fixed point in D. Finally, using fixed_point_iteration.py with an initial guess of $\boldsymbol{x}^{(0)} = (1,1,1)^\top \in D$, we obtain as output the following table and approximate the fixed point to be $\boldsymbol{x}^{(5)} = (0.5, 0, -0.5235988)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	1.0000000	1.0000000	1.0000000
1	0.3467674	0.0158258	-0.4919927
2	0.4999899	-0.0036866	-0.5233251
3	0.4999994	-0.0000003	-0.5236910
4	0.5000000	0.0000000	-0.5235988

5	0.5000000	0.0000000	-0.5235988

10.1.5(b)

Consider $\boldsymbol{G}: D \subset \mathbb{R}^3 \to \mathbb{R}^3$ where

$$\boldsymbol{G}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1/15(13 - x_2^2 + 4x_3) \\ 1/10(11 + x_3 - x_1^2) \\ 1/25(22 + x_2^3) \end{pmatrix} = \begin{pmatrix} g_1(x_1, x_2, x_3) \\ g_2(x_1, x_2, x_3) \\ g_3(x_1, x_2, x_3) \end{pmatrix}$$

and $D = \left\{(x_1, x_2, x_3)^\top : 0 \le x_i \le 1.5, i = 1,2,3\right\}$. We want to show that $\boldsymbol{G}$ has a unique fixed point on D by applying Theorem 10.6. Hence, we will first show that $\boldsymbol{G}(\boldsymbol{x}) \in D$ for any $\boldsymbol{x} \in D$; that is, $0 \le g_i(\boldsymbol{x}) \le 1.5$ for each $i = 1,2,3$. Using Wolfram Alpha, we find that

$$\begin{aligned} 0.716667 \le g_1(x_1, x_2, x_3) \le 1.26667 &\Rightarrow 0 \le g_1(x_1, x_2, x_3) \le 1.5 \\ 0.875 \le g_2(x_1, x_2, x_3) \le 1.25 &\Rightarrow 0 \le g_2(x_1, x_2, x_3) \le 1.5 \\ 0.88 \le g_3(x_1, x_2, x_3) \le 1.015 &\Rightarrow 0 \le g_3(x_1, x_2, x_3) \le 1.5 \end{aligned}$$

so we have that $\boldsymbol{G}(\boldsymbol{x}) \in D$ for any $\boldsymbol{x} \in D$. Next, we calculate the partials $\partial g_i / \partial x_j$ for each $i = 1,2,3$ and $j = 1,2,3$ and use Wolfram Alpha to find the minimum and maximum for each on D:

$$\begin{aligned} \frac{\partial g_1}{\partial x_1} &= 0 && (\min = 0, \max = 0) \\ \frac{\partial g_1}{\partial x_2} &= -\frac{2}{15}x_2 && (\min = -0.2, \max = 0) \\ \frac{\partial g_1}{\partial x_3} &= \frac{4}{15} && (\min = \frac{4}{15}, \max = \frac{4}{15}) \\ \frac{\partial g_2}{\partial x_1} &= -\frac{1}{5}x_1 && (\min = -0.3, \max = 0) \\ \frac{\partial g_2}{\partial x_2} &= 0 && (\min = 0, \max = 0) \\ \frac{\partial g_2}{\partial x_3} &= \frac{1}{10} && (\min = \frac{1}{10}, \max = \frac{1}{10}) \\ \frac{\partial g_3}{\partial x_1} &= 0 && (\min = 0, \max = 0) \\ \frac{\partial g_3}{\partial x_2} &= \frac{3}{25}x_2^2 && (\min = 0, \max = 0.27) \\ \frac{\partial g_3}{\partial x_3} &= 0 && (\min = 0, \max = 0) \end{aligned}$$

Next, we take the absolute value of all maximums and minimums above and find the largest value. By inspection, this is 0.27. We hence have that $K/3 = 0.27$, so $K = 0.27(3) = 0.81 < 1$. Hence, since $\boldsymbol{G}(\boldsymbol{x}) \in D$ and

$$\left|\frac{\partial g_i}{\partial x_j}\right| \le \frac{K}{3}$$

for $i, j \in \{1,2,3\}$ and $K = 0.81 < 1$, we have by Theorem 10.6 that $\boldsymbol{G}(\boldsymbol{x})$ has a unique fixed point in D. Leveraging the Python code fixed_point_iteration.py with an initial guess of $\boldsymbol{x}^{(0)} = (1,1,1)^\top \in D$, we obtain as output the following table and approximate the fixed point to be $\boldsymbol{x}^{(9)} = (1.0364011, 1.0857072, 0.9311911)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$
0	1.0000000	1.0000000	1.0000000
1	1.0666667	1.1000000	0.9200000
2	1.0313333	1.0782222	0.9332400
3	1.0380265	1.0869592	0.9301401

4	1.0359387	1.0852641	0.9313688
5	1.0365118	1.0858200	0.9311289
6	1.0363674	1.0856772	0.9312075
7	1.0364090	1.0857150	0.9311873
8	1.0363981	1.0857044	0.9311926
9	1.0364011	1.0857072	0.9311911

10.1.11

Consider the function $\boldsymbol{F} : \mathbb{R}^3 \to \mathbb{R}^3$ defined by

$$\boldsymbol{F}(x_1, x_2, x_3) = (x_1 + 2x_3, x_1 \cos x_2, x_2^2 + x_3)^\top = \big(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x})\big)^\top.$$

We want to show that $\boldsymbol{F}$ is continuous on $\mathbb{R}^3$. Hence, we will show that each component function $f_i(\boldsymbol{x})$ of $\boldsymbol{F}$ is continuous for $i = 1, 2, 3$. Let's start by showing that $f_1(\boldsymbol{x}) = x_1 + 2x_3$ is continuous. Note that x_1 is continuous for any $\boldsymbol{x} \in \mathbb{R}^3$ and $2x_3$ is also continuous for any $\boldsymbol{x} \in \mathbb{R}^3$. The sum of two continuous functions on $\mathbb{R}^3$ is also continuous, so we have that $f_1(\boldsymbol{x}) = x_1 + 2x_3$ is continuous on $\mathbb{R}^3$. Next, we'll show that $f_2(\boldsymbol{x})$ is continuous on $\mathbb{R}^3$. We have that x_1 is continuous on $\mathbb{R}^3$ for any $\boldsymbol{x} \in \mathbb{R}^3$ and furthermore that $\cos x_2$ is continuous on $\mathbb{R}^3$ for any $\boldsymbol{x} \in \mathbb{R}^3$. Since the product of continuous functions is continuous, we hence have that $f_2(\boldsymbol{x}) = x_1 \cos x_2$ is also continuous on $\mathbb{R}^3$. Lastly, we will show the continuity of $f_3(\boldsymbol{x})$ on $\mathbb{R}^3$. The functions x_2^2 and x_3 are both continuous on $\mathbb{R}^3$ for any choice of $\boldsymbol{x} \in \mathbb{R}^3$, so it follows that the sum $f_3(\boldsymbol{x}) = x_2^2 + x_3$ is also continuous on $\mathbb{R}^3$. By Definition 10.3, since we have shown that $f_i(\boldsymbol{x})$ is continuous on $\mathbb{R}^3$ for $i \in \{1, 2, 3\}$, $\boldsymbol{F}$ is continuous on $\mathbb{R}^3$, i.e., $\boldsymbol{F} \in \mathcal{C}(\mathbb{R}^3)$.

10.2.7(a)

Consider the system of nonlinear equations

$$\begin{cases} 3x_1^2 - x_2^2 & = 0 \\ 3x_1 x_2^2 - x_1^3 - 1 & = 0 \end{cases}$$

along with initial guess $\boldsymbol{x}^{(0)} = (1, 1)^\top$. We will iterate until $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\| < 10^{-6}$ using Newton's Method for Nonlinear Systems. Using the Python code newtons_method_systems.py, we get the table of iterative approximations below and obtain $\boldsymbol{x}^{(5)} = (0.5000000, 0.8660254)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$
0	1.0000000	1.0000000
1	0.6111111	0.8333333
2	0.5036591	0.8524944
3	0.4999641	0.8660456
4	0.5000000	0.8660254
5	0.5000000	0.8660254

10.2.7(b)

Consider the system of nonlinear equations

$$\begin{cases} \ln(x_1^2 + x_2^2) - \sin(x_1 x_2) - \ln 2 - \ln \pi & = 0 \\ e^{x_1 - x_2} + \cos(x_1 x_2) & = 0 \end{cases}$$

along with initial guess $\boldsymbol{x}^{(0)} = (2, 2)^\top$. We will iterate until $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\| < 10^{-6}$ using Newton's Method for Nonlinear Systems. Using the Python code newtons_method_systems.py, we get the table of iterative approximations below and obtain $\boldsymbol{x}^{(6)} = (1.7724539, 1.7724539)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$
0	2.0000000	2.0000000

1	1.9686826	1.4789055
2	1.8300800	1.7090238
3	1.7755575	1.7684117
4	1.7724655	1.7724386
5	1.7724539	1.7724539
6	1.7724539	1.7724539

10.3.5(a)

Consider the system of nonlinear equations

$$\begin{cases} x_1(1-x_1)+4x_2 & = 12 \\ (x_1-2)^2+(2x_2-3)^2 & = 25 \end{cases}$$

for which we want to approximate the solutions of using Broyden's Method. Choosing initial guess $\boldsymbol{x}^{(0)} = (3,4)^\top$ and iterating until $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\| < 10^{-6}$, we use broydens_method.py to obtain as output the following table of iterative approximations and find that $\boldsymbol{x}^{(4)} = (2.5469465, 3.9849975)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$
0	3.0000000	4.0000000
1	2.5520114	3.9856524
2	2.5470104	3.9850062
3	2.5469466	3.9849975
4	2.5469465	3.9849975

10.3.5(b)

Consider the system of nonlinear equations

$$\begin{cases} 5x_1^2 - x_2^2 & = 0 \\ x_2 - 0.25(\sin x_1 + \cos x_2) & = 0 \end{cases}$$

which we want to approximate the solution of using Broyden's Method. Choosing initial guess $\boldsymbol{x}^{(0)} = (0.1, 0.3)^\top$ and iterating until $\|\boldsymbol{x}^{(k)} - \boldsymbol{x}^{(k-1)}\| < 10^{-6}$, we use broydens_method.py to obtain as output the following table of iterative approximations and find that $\boldsymbol{x}^{(4)} = (0.1212419, 0.2711052)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$
0	0.1000000	0.3000000
1	0.1208222	0.2710125
2	0.1212378	0.2711043
3	0.1212419	0.2711052
4	0.1212419	0.2711052

10.3.10

Proof. Let $\boldsymbol{y} \in \mathbb{R}^n$ be a nonzero vector and $\boldsymbol{z} \in \mathbb{R}^n$. We consider the orthogonal projection of $\boldsymbol{z}$ onto $\boldsymbol{y}$, which we will denote as $\boldsymbol{z}_1$. This is given by

$$\boldsymbol{z}_1 = \operatorname{proj}_{\boldsymbol{y}} \boldsymbol{z} = \frac{\boldsymbol{z}^\top \boldsymbol{y}}{\boldsymbol{y}^\top \boldsymbol{y}} \boldsymbol{y} = \underbrace{\frac{\boldsymbol{z}^\top \boldsymbol{y}}{\|\boldsymbol{y}\|_2^2}}_{c} \boldsymbol{y} = c\boldsymbol{y}.$$

Clearly, since $z_1 = cy$ for the constant $c \in \mathbb{R}$ as defined above, z_1 is parallel to y. Now let $z_2 = z - z_1$. We want to show that z_2 is orthogonal to y, which implies that $z_2^\top y = 0$. By substitution and definition of scalar products, it follows that

$$\begin{aligned}
z_2^\top y &= (z - z_1)^\top y \\
&= z^\top y - z_1^\top y \\
&= z^\top y - \left(\frac{z^\top y}{\|y\|_2^2} y\right)^\top \\
&= z^\top y - \frac{z^\top y}{\|y\|_2^2} y^\top y \\
&= z^\top y - \frac{z^\top y}{\|y\|_2^2} \|y\|_2^2 \\
&= z^\top y - z^\top y \\
&= 0
\end{aligned}$$

and hence z_2 and y are orthogonal. Therefore, we have that $z = z_1 + z_2$, where z_1 is parallel to y and z_2 is orthogonal to y. This completes the proof, and we're done. □

10.4.1(a)

Consider the system of nonlinear equations

$$\begin{cases} 4x_1^2 - 20x_1 + \frac{1}{4}x_2^2 + 8 &= 0 \\ \frac{1}{2}x_1 x_2^2 + 2x_1 - 5x_2 + 9 &= 0 \end{cases}$$

for which we want to approximation the solutions of using the method of steepest descent. Choosing initial guess $x^{(0)} = (0,0)^\top$ for a tolerance of 0.05, we use steepest_descent.py to obtain as output the following table of iterative approximations, finding that $x^{(11)} = (0.4934642, 1.9398924)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$	$g\left(x_1^{(k)}, x_2^{(k)}\right)$
0	0.0000000	0.0000000	128.0000000
1	0.4120558	0.1144599	68.3317155
2	0.2130500	0.9990791	29.9003392
3	0.4300149	1.0468405	15.0816216
4	0.3491509	1.5154867	6.6372618
5	0.4571904	1.5337223	3.2594559
6	0.4240333	1.7618942	1.5149149
7	0.4766078	1.7694624	0.7513079
8	0.4616658	1.8797549	0.3683378
9	0.4875924	1.8832549	0.1867730
10	0.4803637	1.9381709	0.0945728
11	0.4934642	1.9398924	0.0487144

10.4.1(b)

Consider the system of nonlinear equations

$$\begin{cases} 3x_1^2 - x_2^2 &= 0 \\ 3x_1 x_2^2 - x_1^3 - 1 &= 0 \end{cases}$$

for which we want to approximation the solutions of using the method of steepest descent. Choosing initial guess $x^{(0)} = (1,1)^\top$ for a tolerance of 0.05, we use steepest_descent.py to obtain as output the following table of iterative approximations, finding that $x^{(3)} = (0.4980017, 0.8649830)^\top$.

k	$x_1^{(k)}$	$x_2^{(k)}$	$g(x_1^{(k)}, x_2^{(k)})$
0	1.0000000	1.0000000	5.0000000
1	0.3687279	0.8947880	0.1813149
2	0.5024076	0.8526487	0.0018779
3	0.4980017	0.8649830	0.0000499

10.5.2(a)

Consider the nonlinear system

$$f_1(x_1,x_2) = x_1^2 - x_2^2 + 2x_2 = 0, \ \ f_2(x_1,x_2) = 2x_1 + x_2^2 - 6 = 0$$

with two known solutions $(0.625204094, 2.179355825)^\top$ and $(2.109511920, -1.334532188)^\top$. Using the continuation method and the Runge-Kutta method of order 4 with $N = 1$ (implemented in runge_kutta_continuation.py) for $\boldsymbol{x}(0) = (0,0)^\top$, we obtain $\boldsymbol{x}(1) = (2.30398796, -2.00109948)^\top$.

10.5.2(b)

Recall the same nonlinear system and solutions from 10.5.2(a). Using the continuation method and the Runge-Kutta method of order 4 with $N = 1$ (again using runge_kutta_continuation.py) this time with $\boldsymbol{x}(0) = (1,1)^\top$, we get as output $\boldsymbol{x}(1) = (0.59709702, 2.25796842)^\top$.

fixed_point_iteration.py

```
# Imports
import numpy as np
from numpy.linalg import norm
import pandas as pd
import math

# Suppressing scientific notation in output
pd.set_option('display.float_format', '{:.7f}'.format)

# Component functions
g1 = lambda x1, x2, x3: (13 - x2**2 + 4*x3)/15
g2 = lambda x1, x2, x3: (11 + x3 - x1**2)/10
g3 = lambda x1, x2, x3: (22 + x2**3)/25

# Tolerance
TOL = 10**(-5)

# Initial guess
x = np.matrix([[0.5], [0.5], [0.5]])

# Arrays for approximations for each iteration
x1k = np.array(x[0,0])
x2k = np.array(x[1,0])
x3k = np.array(x[2,0])

# Defining l-max norm for a 2D vector
def norm(x):
  return max(abs(x[0,0]), abs(x[1,0]), abs(x[2,0]))

# Defining vector-valued function
def G(x1, x2, x3):
  return np.matrix([[g1(x1,x2,x3)], [g2(x1,x2,x3)], [g3(x1,x2,x3)]])

# Starting iteration
k = 1

while True:

  y = G(x[0,0], x[1,0], x[2,0])
```

```
    if norm(y-x, ord=np.inf) < TOL:

        # Appending approximations to array for output
        x1k = np.append(x1k, y[0,0])
        x2k = np.append(x2k, y[1,0])
        x3k = np.append(x3k, y[2,0])

        print(f'Fixed point iteration converged to within tolerance after {k} iterations.')
        break

    x = y

    # Iteration counter
    k += 1

    # Appending approximations to array for output
    x1k = np.append(x1k, x[0,0])
    x2k = np.append(x2k, x[1,0])
    x3k = np.append(x3k, x[2,0])

# Output
df = pd.DataFrame({'x_1^(k)': x1k, 'x_2^(k)': x2k, 'x_3^(k)': x3k})
print(df)
```

newtons_method_systems.py

```
# Imports
import numpy as np
from numpy.linalg import inv
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

# Functions
f1 = lambda x1, x2: np.log(x1**2 + x2**2) - np.sin(x1*x2) - np.log(2) - np.log(math.pi)
f2 = lambda x1, x2: math.exp(x1 - x2) + np.cos(x1*x2)

# Initial guess
x = np.matrix([[2], [2]])
# Tolerance
TOL = 10**(-6)

# Arrays for approximations for each iteration
x1k = np.array(x[0,0])
x2k = np.array(x[1,0])

# Partial derivatives (for Jacobian)
f1_x1 = lambda x1, x2: (2*x1)/(x1**2 + x2**2) - x2*np.cos(x1*x2)
f1_x2 = lambda x1, x2: (2*x2)/(x1**2 + x2**2) - x1*np.cos(x1*x2)
f2_x1 = lambda x1, x2: math.exp(x1-x2) - x2*np.sin(x1*x2)
f2_x2 = lambda x1, x2: -x1*np.sin(x1*x2) - math.exp(x1-x2)

# Defining Jacobian
def Jac(x1, x2, inverse=0):
    mat = np.matrix([[f1_x1(x1,x2), f1_x2(x1,x2)], [f2_x1(x1,x2), f2_x2(x1,x2)]])
    if inverse == 0:
        return mat
    if inverse == 1:
        return inv(mat)

# Defining vector-valued function
def F(x1, x2):
    return np.matrix([[f1(x1,x2)], [f2(x1,x2)]])

# Defining maximum norm for a 2D vector
def norm(x):
```

```
  return max(abs(x[0,0]), abs(x[1,0]))

# Starting iteration
k = 1

while True:
  # Solving n x n linear system for y
  y = np.matmul(Jac(x[0,0], x[1,0], inverse=1), -F(x[0,0], x[1,0]))
  # Updating x
  x = x + y

  # When accuracy tolerance is met
  if norm(y) < TOL:
    print(f'The procedure was successful after {k} iterations.')
    break

  # Next iteration
  k = k + 1

  # Appending approximations to array for output
  x1k = np.append(x1k, x[0,0])
  x2k = np.append(x2k, x[1,0])

# Output
df = pd.DataFrame({'x_1^(k)': x1k, 'x_2^(k)': x2k,})
print(df)
```

nonlinear_finite_diff.py

```
# Imports
import numpy as np
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 8)

# Function (and partials)
f = lambda x, y, yp: yp*np.cos(x) - y*np.log(y)
fy = lambda x, y, yp: -np.log(y) - 1
fyp = lambda x, y, yp: np.cos(x)

# Actual solution
y = lambda x: math.exp(np.sin(x))

# Left endpoint
a = 0
# Right endpoint
b = math.pi/2
# Left endpoint value
alpha = 1
# Right endpoint value
beta = math.exp(1)
# N value
N = 9
# Maximum iterations (for convenience, choose N+1)
M = N+1
# Tolerance
TOL = 10**(-4)

# Initializing arrays (used in approximation computation)
arr_a = np.zeros(N+2)
arr_b = np.zeros(N+2)
arr_c = np.zeros(N+2)
arr_d = np.zeros(N+2)
arr_l = np.zeros(N+2)
arr_u = np.zeros(N+2)
arr_z = np.zeros(N+2)
arr_v = np.zeros(N+2)
```

```python
w      = np.zeros(N+2)

# Defining l-inf norm
def norm(x):
  return max(abs(x))

# Step 1
h = (b-a)/(N+1)
w[0] = alpha
w[-1] = beta

# Timesteps
x_steps = np.arange(a, b+h, h)

# Step 2
for i in range(1, N+1):
  w[i] = alpha + i*((beta-alpha)/(b-a))*h

# Step 3
k = 1

# Step 4
while k <= M:

  # Step 5
  x = a + h
  t = (w[2] - alpha)/(2*h)
  arr_a[1] = 2 + h**2*fy(x, w[1], t)
  arr_b[1] = -1 + (h/2)*fyp(x, w[1], t)
  arr_d[1] = -(2*w[1] - w[2] - alpha + h**2*f(x, w[1], t))

  # Step 6
  for i in range(2, N):
    x = a + i*h
    t = (w[i+1] - w[i-1])/(2*h)
    arr_a[i] = 2 + h**2*fy(x, w[i], t)
    arr_b[i] = -1 + (h/2)*fyp(x, w[i], t)
    arr_c[i] = -1 - (h/2)*fyp(x, w[i], t)
    arr_d[i] = -(2*w[i] - w[i+1] - w[i-1] + h**2*f(x, w[i], t))

  # Step 7
  x = b - h
  t = (beta - w[-3])/(2*h)
  arr_a[-2] = 2 + h**2*fy(x, w[-2], t)
  arr_c[-2] = -1 - (h/2)*fyp(x, w[-2], t)
  arr_d[-2] = -(2*w[-2] - w[-3] - beta + h**2*f(x, w[-2], t))

  # Step 8
  arr_l[1] = arr_a[1]
  arr_u[1] = arr_b[1]/arr_a[1]
  arr_z[1] = arr_d[1]/arr_l[1]

  # Step 9
  for i in range(2, N):
    arr_l[i] = arr_a[i] - arr_c[i]*arr_u[i-1]
    arr_u[i] = arr_b[i]/arr_l[i]
    arr_z[i] = (arr_d[i] - arr_c[i]*arr_z[i-1])/arr_l[i]

  # Step 10
  arr_l[-2] = arr_a[-2] - arr_c[-2]*arr_u[-3]
  arr_z[-2] = (arr_d[-2] - arr_c[-2]*arr_z[-3])/arr_l[-2]

  # Step 11
  arr_v[-2] = arr_z[-2]
  w[-2] = w[-2] + arr_v[-2]

  # Step 12
  for i in range(N-1, 0, -1):
    arr_v[i] = arr_z[i] - arr_u[i]*arr_v[i+1]
    w[i] = w[i] + arr_v[i]
```

```

  # Step 13
  if norm(arr_v) <= TOL:
    # Step 14
    print(f'The procedure was successful after {k} iterations.')
    # Step 15 (output given below)
    break

  # Step 16
  k = k + 1

# Step 17
if k == M:
  print('Maximum number of iterations exceeded.\nThe procedure was not successful.')

yt = np.zeros(N+2)

# Evaluating true values of function
for idx, val in enumerate(x_steps):
  yt[idx] = y(val)

df = pd.DataFrame({'x_i': x_steps, 'w_i': w, 'y(x_i)': yt})
print(df)
```

broydens_method.py

```
# Imports
import numpy as np
from numpy.linalg import inv
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

# Functions
f1 = lambda x1, x2: x1*(1-x1) + 4*x2 - 12
f2 = lambda x1, x2: (x1-2)**2 + (2*x2-3)**2 - 25

# Initial guess
x = np.matrix([[3], [4]])
# Tolerance
TOL = 10**(-6)

# Arrays for approximations for each iteration
x1k = np.array(x[0,0])
x2k = np.array(x[1,0])

# Partial derivatives (for step 1 Jacobian)
f1_x1 = lambda x1, x2: 1 - 2*x1
f1_x2 = lambda x1, x2: 4
f2_x1 = lambda x1, x2: 2*(x1-2)
f2_x2 = lambda x1, x2: 4*(2*x2-3)

# Defining Jacobian
def Jac(x1, x2, inverse=0):
  mat = np.matrix([[f1_x1(x1,x2), f1_x2(x1,x2)], [f2_x1(x1,x2), f2_x2(x1,x2)]])
  if inverse == 0:
    return mat
  if inverse == 1:
    return inv(mat)

# Defining vector-valued function
def F(x1, x2):
  return np.matrix([[f1(x1,x2)], [f2(x1,x2)]])

# Defining l-max norm for a 2D vector
def norm(x):
  return max(abs(x[0,0]), abs(x[1,0]))
```

```python

# Step 1
A0 = Jac(x[0,0], x[1,0])
v = F(x[0,0], x[1,0])

# Step 2
A = inv(A0)

# Step 3
s = np.matmul(-A, v)
x = x + s
k = 2

# Step 4
while True:

  # Step 5
  w = v
  v = F(x[0,0], x[1,0])
  y = v - w

  # Step 6
  z = np.matmul(-A, y)

  # Step 7
  p = np.matmul(-s.T, z)

  # Step 8
  ut = np.matmul(s.T, A)

  # Step 9
  A = A + np.multiply(1/p, np.matmul(s+z, ut))

  # Step 10
  s = np.matmul(-A, v)

  # Step 11
  x = x + s

  # Step 12
  if norm(s) < TOL:

    # Appending approximations to array for output
    x1k = np.append(x1k, x[0,0])
    x2k = np.append(x2k, x[1,0])

    print(f'The procedure was successful after {k} iterations.')
    break

  # Step 13 (next iteration)
  k = k + 1

  # Appending approximations to array for output
  x1k = np.append(x1k, x[0,0])
  x2k = np.append(x2k, x[1,0])

# Output
df = pd.DataFrame({'x_1^(k)': x1k, 'x_2^(k)': x2k,})
print(df)
```

runge_kutta_continuation.py

```python
# Imports
import numpy as np
from numpy.linalg import inv, norm
import pandas as pd
import math

# For more decimal places
```

```
pd.set_option("display.precision", 7)

# Functions
f1 = lambda x1, x2: x1**2 - x2**2 + 2*x2
f2 = lambda x1, x2: 2*x1 + x2**2 - 6

# Initial guess
x = np.matrix([[0], [0]])
# Number of iterations
N = 1

# Partial derivatives (for Jacobian)
f1_x1 = lambda x1, x2: 2*x1
f1_x2 = lambda x1, x2: -2*x2 + 2
f2_x1 = lambda x1, x2: 2
f2_x2 = lambda x1, x2: 2*x2

# Defining Jacobian
def Jac(x1, x2):
  mat = np.matrix([[f1_x1(x1,x2), f1_x2(x1,x2)], [f2_x1(x1,x2), f2_x2(x1,x2)]])
  return mat

# Defining vector-valued function
def F(x1, x2):
  return np.matrix([[f1(x1,x2)], [f2(x1,x2)]])

# Step 1
h = 1/N
b = -h*F(x[0,0], x[1,0])

# Step 2
for i in range(1, N+1):

  # Step 3
  A = Jac(x[0,0], x[1,0])
  k1 = np.dot(inv(A), b)

  # Step 4
  A = Jac(x[0,0] + 1/2*k1[0,0], x[1,0] + 1/2*k1[1,0])
  k2 = np.dot(inv(A), b)

  # Step 5
  A = Jac(x[0,0] + 1/2*k2[0,0], x[1,0] + 1/2*k2[1,0])
  k3 = np.dot(inv(A), b)

  # Step 6
  A = Jac(x[0,0] + k3[0,0], x[1,0] + k3[1,0])
  k4 = np.dot(inv(A), b)

  # Step 7
  x = x + (k1 + 2*k2 + 2*k3 + k4)/6

# Output
print(f'The continuation method with fourth-order Runge-Kutta approximates the solution {round(
    float(x[0]), 8), round(float(x[1]), 8)} for N={N}.')
```

steepest_descent.py

```
# Imports
import numpy as np
from numpy.linalg import inv, norm
import pandas as pd
import math

# For more decimal places
pd.set_option("display.precision", 7)

# Initial guess
x = np.matrix([[1], [1]])
```

```
# Tolerance
TOL = 0.05

# Equations in system
f1 = lambda x1, x2: 3*x1**2 - x2**2
f2 = lambda x1, x2: 3*x1*x2**2 - x1**3 - 1

# Combining into g
g = lambda x1, x2: f1(x1,x2)**2 + f2(x1,x2)**2

# Arrays for approximations for each iteration
x1k = np.array(x[0,0])
x2k = np.array(x[1,0])
g_vals = np.array(g(x[0,0], x[1,0]))

# Partial derivatives (for Jacobian)
f1_x1 = lambda x1, x2: 6*x1
f1_x2 = lambda x1, x2: -2*x2
f2_x1 = lambda x1, x2: 3*x2**2 - 3*x1**2
f2_x2 = lambda x1, x2: 6*x1*x2

# Defining Jacobian
def Jac(x1, x2, transpose=0):
  mat = np.matrix([[f1_x1(x1,x2), f1_x2(x1,x2)], [f2_x1(x1,x2), f2_x2(x1,x2)]])
  if transpose == 0:
    return mat
  if transpose == 1:
    return mat.T

# Defining vector-valued function
def F(x1, x2):
  return np.matrix([[f1(x1,x2)], [f2(x1,x2)]])

# Step 1
k = 1

# Step 2
while True:

  # Step 3
  g1 = g(x[0,0], x[1,0])
  z = 2*np.dot(Jac(x[0,0], x[1,0], transpose=1), F(x[0,0], x[1,0]))
  z0 = norm(z)

  # Step 4
  if z0 == 0:
    print('Zero gradient.')
    break

  # Step 5
  z = z/z0
  alpha1 = 0
  alpha3 = 1
  g3 = g(x[0,0] - alpha3*z[0,0], x[1,0] - alpha3*z[1,0])

  # Step 6
  while g3 >= g1:

    # Step 7
    alpha3 = alpha3/2
    g3 = g(x[0,0] - alpha3*z[0,0], x[1,0] - alpha3*z[1,0])

    # Step 8
    if (alpha3 < TOL/2):
      print('No likely improvement.')
      break

  # Step 9
  alpha2 = alpha3/2
  g2 = g(x[0,0] - alpha2*z[0,0], x[1,0]-alpha2*z[1,0])
```

```python

  # Step 10
  h1 = (g2-g1)/alpha2
  h2 = (g3-g2)/(alpha3-alpha2)
  h3 = (h2-h1)/alpha3

  # Step 11
  alpha0 = 1/2*(alpha2 - h1/h3)
  g0 = g(x[0,0]-alpha0*z[0,0], x[1,0]-alpha0*z[1,0])

  # Step 12
  if (g0 <= g3):
    g_min = g0
    alpha = alpha0
  else:
    g_min = g3
    alpha = alpha3

  # Step 13
  x = x - alpha*z

  # Step 14
  if (abs(g_min - g1) < TOL):

    print(f'The procedure was successful after {k} iterations.')
    x1k = np.append(x1k, x[0,0])
    x2k = np.append(x2k, x[1,0])
    g_vals = np.append(g_vals, g_min)

    break

  # Step 15 (iteration)
  k = k + 1

  # Appending approximations to array for output
  x1k = np.append(x1k, x[0,0])
  x2k = np.append(x2k, x[1,0])
  g_vals = np.append(g_vals, g_min)

# Output
df = pd.DataFrame({'x_1^(k)': x1k, 'x_2^(k)': x2k, 'g(x_1^(k), x_2^(k))': g_vals})
print(df)
```