MA 544 Programming Assignment 2

Bring your questions on the discussion board for Module 3 for helpful hints on these question.

In [ ]:
```
import numpy as np
import sys
```

Question 1: Iterative Methods for Linear Systems

Consider the following linear system

$$\begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 6 \\ 25 \\ -11 \\ 15 \end{pmatrix}$$

Jacobi Method

> Initialize the iterative solution vector $x^{(0)}$ randomly, or with the zero vector,
>
> for k=0:maxIteration, update every element until convergece
>
> > for i=1:n

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right).$$

In [ ]:
```python
# You can modify this code to answer the following
'''
Jacobi's iteration method for solving the system of equations Ax=b.
p0 is the initialization for the iteration.
'''
def jacobi(A, b, p0, tol, maxIter=100):
    n=len(A)
    p = p0

    for k in range(maxIter):
        p_old = p.copy() # In python assignment is not the same as copy

        # Update every component of iterant p
        for i in range(n):
            sumi = b[i];
            for j in range(n):
                if i==j: # Diagonal elements are not included in Jacobi
                    continue;
                sumi = sumi - A[i,j] * p_old[j]
            p[i] = sumi/A[i,i]

        rel_error = np.linalg.norm(p-p_old)/n # Actually 'n' should be replace by n
        # print("Relative error in iteration", k+1,":",rel_error)
        if rel_error<tol:
            print("TOLERANCE MET BEFORE MAX-ITERATION")
            break
    return p;
```

In [ ]:
```python
# Example System
A1 = np.array([[10, -1, 2, 0],
               [-1, 11, -1, 3],
               [2, -1, 10, -1],
               [0, 3, -1, 8]],dtype=float)
b = np.array([6, 25, -11, 15],dtype=float)
```

In [ ]:
```python
# Solved by using Jacobi Method
x = jacobi(A1,b, np.array([0,0,0,0],dtype=float),0.0000001, 100)
print("The solution is: ",x)
```

```
TOLERANCE MET BEFORE MAX-ITERATION
The solution is:  [ 1.00000003  1.99999996 -0.99999997  0.99999995]
```

(A) **Modify** the code for Jacobi Method to implement the Gauss-Siedel Iteration in Python. Solve the above system by using this method. Exact answer is (1,2,-1,1). Stopping criteria could be a relative error $\delta < 0.000001$.

In [ ]:
```python
# Your Code here
def gauss_siedel(A, b, p0, tol, maxIter=100):
    n=len(A)
    p = p0

    for k in range(maxIter):
        # Even though we don't use p_old for updating,
        # still obtain it for error_tolerance checking
        p_old = p.copy() # In python assignment is not the same as copy

        # Update every component of iterant p
        for i in range(n):
            sumi = b[i];
            for j in range(n):
                if i==j: # Diagonal elements are not included
                    continue;
                # For j = 1 to i - 1, new p[j] is used
                # For J = i + 1 to n, old p[j] is used
                # Therefore, just directly access and modify p
                # Since the first i - 1 elements of p will
                # be the new p[j]s anyway
                sumi = sumi - A[i,j] * p[j]
            p[i] = sumi/A[i,i]

        rel_error = np.linalg.norm(p-p_old)/n # Actually 'n' should be replace by n
        # print("Relative error in iteration", k+1,":",rel_error)
        if rel_error<tol:
            print("TOLERANCE MET BEFORE MAX-ITERATION")
            break
    return p;

# Solved by using Jacobi Method
x = gauss_siedel(A1,b, np.array([0,0,0,0],dtype=float),0.0000001, 100)
print("The solution is: ",x)
```

```
TOLERANCE MET BEFORE MAX-ITERATION
The solution is:  [ 1.  2. -1.  1.]
```

(B) Successive overrelaxation (SOR) is another iterative method for solving linear systems. It picks up the next iteration from a weighted sum of the current iteration and the next iteration by Gauss-seidel. **Modify** the code for Jacobi Method to implement the SOR method in Python and solve the above system again with $\omega = 1.5$. Display the solution of the above system by this method.

In [ ]:

```python
# Your work starts here
def sor(A, b, p0, tol, w=1.5, maxIter=100):
    n=len(A)
    p = p0

    for k in range(maxIter):
        # Even though we don't use p_old for updating,
        # still obtain it for error_tolerance checking
        p_old = p.copy() # In python assignment is not the same as copy

        # Update every component of iterant p
        for i in range(n):
            sumi = b[i];
            for j in range(n):
                if i==j: # Diagonal elements are not included
                    continue;
                # For j = 1 to i - 1, new p[j] is used
                # For J = i + 1 to n, old p[j] is used
                # Therefore, just directly access and modify p
                # Since the first i - 1 elements of p will
                # be the new p[j]s anyway
                sumi = sumi - A[i,j] * p[j]
            p[i] = (1 - w) * p[i] + w * sumi/A[i,i]

        rel_error = np.linalg.norm(p-p_old)/n # Actually 'n' should be replace by n
        # print("Relative error in iteration", k+1,":",rel_error)
        if rel_error<tol:
            print("TOLERANCE MET BEFORE MAX-ITERATION")
            break
    return p;

# Solved by using Jacobi Method
x = sor(A1,b, np.array([0,0,0,0],dtype=float),0.0000001, 1.5, 100)
print("The solution is: ",x)
```

```
TOLERANCE MET BEFORE MAX-ITERATION
The solution is:  [ 1.00000007  2.00000001 -1.00000004  1.00000006]
```

Question 2: Gaussian Elimination with Pivoting

In [ ]:

```
## Gaussian Elimination: Scaled Row Pivoting
## This function is based on the pseudo-code on page-148 in the Text by Kincaid and
def GE_srpp(X, verbose=False):
    '''
    This function returns the P'LU factorization of a square matrix A
    by scaled row partial pivoting.
    In place of returning L and U, elements of modified A are used to hold values o
    '''
    A = np.copy(X)
    m,n = A.shape
    swap=0;

    # The initial ordering of rows
    p = list(range(m))
    if verbose:
        print("permutation vector initialized to: ",p)

    # Scaling vector: absolute maximum elements of each row
    s = np.max(np.abs(A), axis=1)

    # Start the k-1 passes of Guassian Elimination on A
    for k in range(m-1):
        if verbose:
            print("Scaling Vector: ",s)
        # Find the pivot element and interchange the rows
        pivot_index = k + np.argmax(np.abs(A[p[k:], k])/s[p[k:]])

        # Interchange elements in the permutation vector if needed
        if pivot_index !=k:
            temp = p[k]
            p[k]=p[pivot_index]
            p[pivot_index] = temp
            swap+=1;

        if verbose:
            print("\nPivot Element: {0:.4f} \n".format(A[p[k],k]))
        if np.abs(A[p[k],k]) < 10**(-20):
             sys.exit("ERROR!! Provided matrix is singular or there is a zero pivot
        # Check the new order of rows
        if verbose:
            print("permutation vector: ",p)
        # For the k-th pivot row Perform the Gaussian elimination on the following
        for i in range(k+1, m):
            # Find the multiplier
            z = A[p[i],k]/A[p[k],k]
            #Save the multiplier z in A itself. You can save this in L also
            A[p[i],k] = z
            #Elimination operation: Changes all elements in a row simultaneously
            A[p[i],k+1:] = A[p[i],k+1:] - z*A[p[k],k+1:]

        if verbose:
            print("\n After PASS {}=========: \n".format(k+1), A)
    return A, p, swap
```

LU Decomposition Example

The above code could be used or modified for a umber of purposes. Here is how it coudl be sed for $PA = LU$ decomposition.

In [ ]:
```
A2 = np.array([[5, 4, 7, 6, 9],
               [7, 8, 9, 9, 8],
               [2, 3, 5, 9, 8],
               [3, 1, 7, 5, 6],
               [9, 1, 3, 7, 3]], dtype=float)
```

In [ ]:
```
newA,p,swaps = GE_srpp(A2, verbose=True)
print("Modified A after Gaussian elimination:\n",newA)
U=np.triu(newA[p,:])
L=np.tril(newA[p,:], -1)+np.eye(5)
P=np.eye(5)[p,:]
print("\n Upper triangular, U:\n ", U)
print("\n Lower triangular, L:\n", L)
print("\n The Permutation Matrix, P:\n",P)
print("Sanity check: Norm of LU-PA (must be close to zero)=",np.linalg.norm(P@A2-L@
```

```
permutation vector initialized to:  [0, 1, 2, 3, 4]
Scaling Vector:  [9. 9. 9. 7. 9.]

Pivot Element: 9.0000

permutation vector:  [4, 1, 2, 3, 0]

 After PASS 1=========:
 [[0.55555556 3.44444444 5.33333333 2.11111111 7.33333333]
 [0.77777778 7.22222222 6.66666667 3.55555556 5.66666667]
 [0.22222222 2.77777778 4.33333333 7.44444444 7.33333333]
 [0.33333333 0.66666667 6.         2.66666667 5.        ]
 [9.         1.         3.         7.         3.        ]]
Scaling Vector:  [9. 9. 9. 7. 9.]

Pivot Element: 7.2222

permutation vector:  [4, 1, 2, 3, 0]

 After PASS 2=========:
 [[0.55555556 0.47692308 2.15384615 0.41538462 4.63076923]
 [0.77777778 7.22222222 6.66666667 3.55555556 5.66666667]
 [0.22222222 0.38461538 1.76923077 6.07692308 5.15384615]
 [0.33333333 0.09230769 5.38461538 2.33846154 4.47692308]
 [9.         1.         3.         7.         3.        ]]
Scaling Vector:  [9. 9. 9. 7. 9.]

Pivot Element: 5.3846

permutation vector:  [4, 1, 3, 2, 0]

 After PASS 3=========:
 [[ 0.55555556  0.47692308  0.4        -0.52        2.84      ]
 [ 0.77777778  7.22222222  6.66666667  3.55555556  5.66666667]
 [ 0.22222222  0.38461538  0.32857143  5.30857143  3.68285714]
 [ 0.33333333  0.09230769  5.38461538  2.33846154  4.47692308]
 [ 9.          1.          3.          7.          3.        ]]
Scaling Vector:  [9. 9. 9. 7. 9.]

Pivot Element: 5.3086

permutation vector:  [4, 1, 3, 2, 0]

 After PASS 4=========:
 [[ 0.55555556  0.47692308  0.4        -0.09795479  3.2007535 ]
 [ 0.77777778  7.22222222  6.66666667  3.55555556  5.66666667]
 [ 0.22222222  0.38461538  0.32857143  5.30857143  3.68285714]
 [ 0.33333333  0.09230769  5.38461538  2.33846154  4.47692308]
 [ 9.          1.          3.          7.          3.        ]]
Modified A after Gaussian elimination:
 [[ 0.55555556  0.47692308  0.4        -0.09795479  3.2007535 ]
 [ 0.77777778  7.22222222  6.66666667  3.55555556  5.66666667]
 [ 0.22222222  0.38461538  0.32857143  5.30857143  3.68285714]
 [ 0.33333333  0.09230769  5.38461538  2.33846154  4.47692308]
 [ 9.          1.          3.          7.          3.        ]]
```

```
 Upper triangular, U:
  [[9.         1.         3.         7.         3.        ]
 [0.         7.22222222 6.66666667 3.55555556 5.66666667]
 [0.         0.         5.38461538 2.33846154 4.47692308]
 [0.         0.         0.         5.30857143 3.68285714]
 [0.         0.         0.         0.         3.2007535 ]]

 Lower triangular, L:
 [[ 1.          0.          0.          0.          0.        ]
 [ 0.77777778  1.          0.          0.          0.        ]
 [ 0.33333333  0.09230769  1.          0.          0.        ]
 [ 0.22222222  0.38461538  0.32857143  1.          0.        ]
 [ 0.55555556  0.47692308  0.4        -0.09795479  1.        ]]

 The Permutation Matrix, P:
 [[0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0.]]
Sanity check: Norm of LU-PA (must be close to zero)= 0.0
```

(A) Modify the code for Gaussian elimination to write a function that solves a linear system $A\mathbf{x} = \mathbf{b}$. Test this on the following system. Display the verbose output and the solution.

$$\begin{aligned} 3x - 5y + z &= 0 \\ x + 2y + 3z &= 1 \\ -2x + 3y - 4z &= 3 \end{aligned}$$

In [ ]:

```
# Your work starts here
## Gaussian Elimination: Scaled Row Pivoting
## This function is based on the pseudo-code on page-148 in the Text by Kincaid and

def GE_srpp_solve(X, b, verbose=False):
    '''
    This function returns the P'LU factorization of a square matrix A
    by scaled row partial pivoting.
    In place of returning L and U, elements of modified A are used to hold values o
    '''
    A = np.copy(X)
    bc = np.copy(b)
    m,n = A.shape
    swap=0;

    # The initial ordering of rows
    p = list(range(m))
    if verbose:
        print("permutation vector initialized to: ",p)

    # Scaling vector: absolute maximum elements of each row
    s = np.max(np.abs(A), axis=1)

    # Start the k-1 passes of Guassian Elimination on A
    for k in range(m-1):
        if verbose:
            print("Scaling Vector: ",s)
        # Find the pivot element and interchange the rows
        pivot_index = k + np.argmax(np.abs(A[p[k:], k])/s[p[k:]])

        # Interchange elements in the permutation vector if needed
        if pivot_index !=k:
            temp = p[k]
            p[k]=p[pivot_index]
            p[pivot_index] = temp
            swap+=1;

        if verbose:
            print("\nPivot Element: {0:.4f} \n".format(A[p[k],k]))
        if np.abs(A[p[k],k]) < 10**(-20):
             sys.exit("ERROR!! Provided matrix is singular or there is a zero pivot
        # Check the new order of rows
        if verbose:
            print("permutation vector: ",p)
        # For the k-th pivot row Perform the Gaussian elimination on the following
        for i in range(k+1, m):
            # Find the multiplier
            z = A[p[i],k]/A[p[k],k]
            #Save the multiplier z in A itself. You can save this in L also
            A[p[i],k] = z
            #Elimination operation: Changes all elements in a row simultaneously
            A[p[i],k+1:] = A[p[i],k+1:] - z*A[p[k],k+1:]
            #Update augmented column
            bc[p[i]] = bc[p[i]] - z*bc[p[k]]

        if verbose:
```

```
            print("\n After PASS {}=========: \n".format(k+1), A)
            print(f"\n b := {bc} ")
    x = np.zeros_like(b)
    # Backsolve for x vector
    for i in range(m-1, -1, -1):
        # Get element we're solving for and
        # solve "equation" for it
        x[i] = bc[p[i]] / A[p[i], i]
        # Subtract out known quantities to get final
        # value for x[i]
        for j in range(m-1, i, -1):
            x[i] -= x[j] * A[p[i], j] / A[p[i], i]
    return A, p, swap, x

A3 = np.array([[3, -5, 1],
               [1, 2, 3],
               [-2, 3, -4]], dtype=float)

b3 = np.array([0, 1, 3], dtype=float)
newA,p,swaps,x = GE_srpp_solve(A3, b3, verbose=True)
print("Modified A after Gaussian elimination:\n",newA)
U=np.triu(newA[p,:])
L=np.tril(newA[p,:], -1)+np.eye(3)
P=np.eye(3)[p,:]
print("\n Upper triangular, U:\n ", U)
print("\n Lower triangular, L:\n", L)
print("\n The Permutation Matrix, P:\n",P)
print("Sanity check: Norm of LU-PA (must be close to zero)=",np.linalg.norm(P@A3-L@
print(f"Solution to Ax=b, {x}")
```

```
permutation vector initialized to:  [0, 1, 2]
Scaling Vector:  [5. 3. 4.]

Pivot Element: 3.0000

permutation vector:  [0, 1, 2]

 After PASS 1=========:
 [[ 3.         -5.          1.        ]
 [ 0.33333333  3.66666667  2.66666667]
 [-0.66666667 -0.33333333 -3.33333333]]

 b := [0. 1. 3.]
Scaling Vector:  [5. 3. 4.]

Pivot Element: 3.6667

permutation vector:  [0, 1, 2]

 After PASS 2=========:
 [[ 3.         -5.          1.        ]
 [ 0.33333333  3.66666667  2.66666667]
 [-0.66666667 -0.09090909 -3.09090909]]

 b := [0.         1.         3.09090909]
Modified A after Gaussian elimination:
```

```
[[ 3.         -5.          1.        ]
 [ 0.33333333  3.66666667  2.66666667]
 [-0.66666667 -0.09090909 -3.09090909]]

Upper triangular, U:
 [[ 3.         -5.          1.        ]
 [ 0.          3.66666667  2.66666667]
 [ 0.          0.         -3.09090909]]

Lower triangular, L:
 [[ 1.          0.          0.        ]
 [ 0.33333333  1.          0.        ]
 [-0.66666667 -0.09090909  1.        ]]

The Permutation Matrix, P:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Sanity check: Norm of LU-PA (must be close to zero)= 0.0
Solution to Ax=b, [ 2.  1. -1.]
```

In [ ]:

```
# Sanity check on known good solution from question 1
newA,p,swaps,x = GE_srpp_solve(A1, b, verbose=True)
print("Modified A after Gaussian elimination:\n",newA)
U=np.triu(newA[p,:])
L=np.tril(newA[p,:], -1)+np.eye(4)
P=np.eye(4)[p,:]
print("\n Upper triangular, U:\n ", U)
print("\n Lower triangular, L:\n", L)
print("\n The Permutation Matrix, P:\n",P)
print("Sanity check: Norm of LU-PA (must be close to zero)=",np.linalg.norm(P@A1-L@
print(f"Solution to Ax=b, {x}")
```

```
permutation vector initialized to:  [0, 1, 2, 3]
Scaling Vector:  [10. 11. 10.  8.]

Pivot Element: 10.0000

permutation vector:  [0, 1, 2, 3]

 After PASS 1=========:
 [[10.  -1.   2.   0. ]
 [-0.1 10.9 -0.8  3. ]
 [ 0.2 -0.8  9.6 -1. ]
 [ 0.   3.  -1.   8. ]]

 b := [  6.   25.6 -12.2  15. ]
Scaling Vector:  [10. 11. 10.  8.]

Pivot Element: 10.9000

permutation vector:  [0, 1, 2, 3]

 After PASS 2=========:
 [[10.         -1.          2.          0.        ]
 [-0.1        10.9        -0.8         3.        ]
 [ 0.2        -0.0733945   9.5412844  -0.77981651]
 [ 0.          0.27522936 -0.77981651  7.17431193]]

 b := [  6.          25.6        -10.32110092   7.95412844]
Scaling Vector:  [10. 11. 10.  8.]

Pivot Element: 9.5413

permutation vector:  [0, 1, 2, 3]

 After PASS 3=========:
 [[10.         -1.          2.          0.        ]
 [-0.1        10.9        -0.8         3.        ]
 [ 0.2        -0.0733945   9.5412844  -0.77981651]
 [ 0.          0.27522936 -0.08173077  7.11057692]]

 b := [  6.          25.6        -10.32110092   7.11057692]
Modified A after Gaussian elimination:
 [[10.         -1.          2.          0.        ]
 [-0.1        10.9        -0.8         3.        ]
 [ 0.2        -0.0733945   9.5412844  -0.77981651]
 [ 0.          0.27522936 -0.08173077  7.11057692]]

 Upper triangular, U:
  [[10.         -1.          2.          0.        ]
 [ 0.         10.9        -0.8         3.        ]
 [ 0.          0.          9.5412844  -0.77981651]
 [ 0.          0.          0.          7.11057692]]

 Lower triangular, L:
 [[ 1.          0.          0.          0.        ]
 [-0.1         1.          0.          0.        ]
 [ 0.2        -0.0733945   1.          0.        ]
```

```
 [ 0.          0.27522936 -0.08173077  1.        ]]

 The Permutation Matrix, P:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
Sanity check: Norm of LU-PA (must be close to zero)= 4.440892098500626e-16
Solution to Ax=b, [ 1.  2. -1.  1.]
```

(B) Modify this code to find the determinant of any square matrix A. Note that

$$PA = LU \Rightarrow \det A = \pm \det U.$$

The sign depends of the number of row-swaps in the elimination process. Use this code to find the determinant of any 10×10 matrix that you randomly generate. Compare your result with the built-in NumPy method.

In [ ]:
```
# Your work starts here
A4 = np.random.randint(low = -10000, high = 10000, size = (10, 10))
A4 = A4.astype(dtype=float)
newA,p,swaps = GE_srpp(A4)
U=np.triu(newA[p,:])
L=np.tril(newA[p,:], -1)+np.eye(10)
P=np.eye(10)[p,:]
print("Sanity check: Norm of LU-PA (must be close to zero)=",np.linalg.norm(P@A4-L@
det = 1
for i in range(10):
    det *= U[i,i]
det *= (-1) ** (swaps % 2)
print(f"Determinant: {det}")
print(f"Sanity check: numpy det: {np.linalg.det(A4)}")
print(f"Checking against 10000 randomly generated arrays")
for _ in range(10000):
    A4 = np.random.randint(low = -10000, high = 10000, size = (10, 10))
    A4 = A4.astype(dtype=float)
    newA,p,swaps = GE_srpp(A4)
    U=np.triu(newA[p,:])
    L=np.tril(newA[p,:], -1)+np.eye(10)
    P=np.eye(10)[p,:]
    det = 1
    for i in range(10):
        det *= U[i,i]
    det *= (-1) ** (swaps % 2)
    # Run a relative error check
    threshold = 0.000001
    truedet = np.linalg.det(A4)
    if (abs(1 - det / truedet) > threshold):
        raise ValueError(f"Determinant not calculated properly. {det} != {truedet}.
print("All Good!")
```

```
Sanity check: Norm of LU-PA (must be close to zero)= 8.71379205123997e-12
Determinant: 8.198492942329563e+40
Sanity check: numpy det: 8.198492942329543e+40
Checking against 10000 randomly generated arrays
All Good!
```

(C) Modify your system-solver to find the inverse of a square matrix. Use this code to display the inverse of the matrix

$$A = \begin{pmatrix} 3 & -5 & 1 \\ 1 & 2 & 3 \\ -2 & 3 & -4 \end{pmatrix}.$$

In [ ]:
```python
# Your work starts here
def inverse(A):
    n, m = A.shape
    if n != m:
        raise ValueError("Not a square matrix")
    bs = np.eye(n)
    sol = np.zeros((n,n))
    for i in range(n):
        newA,p,swaps,x = GE_srpp_solve(A, bs[:,i])
        sol[:,i] = x
    return sol

A = np.array([[3, -5, 1],
              [1, 2, 3],
              [-2, 3, -4]], dtype=float)
iA = inverse(A)
print(f"Inverse = {iA}\n\n")
print(f"A * A^-1 = {A @ iA}\n\n")
print(f"Error = {np.abs(np.eye(3) - A @ iA)}\n\n")
```

```
Inverse = [[ 0.5         0.5         0.5       ]
 [ 0.05882353  0.29411765  0.23529412]
 [-0.20588235 -0.02941176 -0.32352941]]


A * A^-1 = [[ 1.00000000e+00 -2.08166817e-17  0.00000000e+00]
 [-8.32667268e-17  1.00000000e+00  0.00000000e+00]
 [-1.11022302e-16 -8.32667268e-17  1.00000000e+00]]


Error = [[2.22044605e-16 2.08166817e-17 0.00000000e+00]
 [8.32667268e-17 2.22044605e-16 0.00000000e+00]
 [1.11022302e-16 8.32667268e-17 1.11022302e-16]]
```

Questions 3: Gradient Descent

Modify the code provided for gradient descent to find the minimum for a function in two variables. Show the output for the function

$$f(x_1, x_2) = x_1^2 + x_2^2 - 2x_1 + 4x_2 + 8$$

In [ ]:

```
# Your work starts here
# Define a function that finds the approximate value of the derivative function
def derivative_x0(f, x0_value, x1_value):
    '''
    Arguments:
    f: a function provided by using lambda definition
    Return: The derivative of the function f at x_value
    '''
    h=0.01
    # Numerical differentialtion of f at x_value by central difference formula
    df_central_dif = (f(x0_value+h, x1_value) - f(x0_value-h, x1_value)) /(2*h)
    return df_central_dif

def derivative_x1(f, x0_value, x1_value):
    '''
    Arguments:
    f: a function provided by using lambda definition
    Return: The derivative of the function f at x_value
    '''
    h=0.01
    # Numerical differentialtion of f at x_value by central difference formula
    df_central_dif = (f(x0_value, x1_value+h) - f(x0_value, x1_value-h)) /(2*h)
    return df_central_dif

def gradient_descent(f, x00=0.0, x10=0.0, delta=0.000001, max_iter=100, alpha=0.1):
    '''
    This function uses gradient descent method to find a minimum point of a convex
    and the value there.
    Arguments:
    f: The function whose minimum is to be found, passed as an anonymous (lambda) f
    x0: Initialization for the minimum point
    delta: the precision/tolerance of the minimum point.
    max_iter: maximum number of iterations to perform.
    alpha: the learning rate in gradient descent algorithm
    Returns:
    x_min: a point of local minimum for the function
    f(x_min): a local minimum value for the function
    '''
    x0_current, x1_current = x00, x10
    for i in range(max_iter):
        # Find the derivatives or approximate derivatives at x_current
        dfx0 = derivative_x0(f, x0_current, x1_current)
        dfx1 = derivative_x1(f, x0_current, x1_current)
        # The gradient descent step over two variables
        x0_next = x0_current - alpha*dfx0
        x1_next = x1_current - alpha*dfx1
        # Stop iterating once desired accuracy is achieved on both variables
        if(np.abs(x0_next - x0_current)< delta and np.abs(x1_next - x1_current)< de
            break
        x0_current = x0_next
        x1_current = x1_next
        print("\n Iteration {0:d}: {1:.8f}, {2:.8f}".format(i+1,x0_current, x1_curr
    return x0_current, x1_current, f(x0_current, x1_current)
f = lambda x0, x1: x0 ** 2 + x1 ** 2 - 2 * x0 + 4 * x1 + 8
print(gradient_descent(f))
```

```
Iteration 1: 0.20000000, -0.40000000

Iteration 2: 0.36000000, -0.72000000

Iteration 3: 0.48800000, -0.97600000

Iteration 4: 0.59040000, -1.18080000

Iteration 5: 0.67232000, -1.34464000

Iteration 6: 0.73785600, -1.47571200

Iteration 7: 0.79028480, -1.58056960

Iteration 8: 0.83222784, -1.66445568

Iteration 9: 0.86578227, -1.73156454

Iteration 10: 0.89262582, -1.78525164

Iteration 11: 0.91410065, -1.82820131

Iteration 12: 0.93128052, -1.86256105

Iteration 13: 0.94502442, -1.89004884

Iteration 14: 0.95601953, -1.91203907

Iteration 15: 0.96481563, -1.92963126

Iteration 16: 0.97185250, -1.94370500

Iteration 17: 0.97748200, -1.95496400

Iteration 18: 0.98198560, -1.96397120

Iteration 19: 0.98558848, -1.97117696

Iteration 20: 0.98847078, -1.97694157

Iteration 21: 0.99077663, -1.98155326

Iteration 22: 0.99262130, -1.98524260

Iteration 23: 0.99409704, -1.98819408

Iteration 24: 0.99527763, -1.99055527

Iteration 25: 0.99622211, -1.99244421

Iteration 26: 0.99697769, -1.99395537

Iteration 27: 0.99758215, -1.99516430

Iteration 28: 0.99806572, -1.99613144
```

```
Iteration 29: 0.99845257, -1.99690515

Iteration 30: 0.99876206, -1.99752412

Iteration 31: 0.99900965, -1.99801930

Iteration 32: 0.99920772, -1.99841544

Iteration 33: 0.99936617, -1.99873235

Iteration 34: 0.99949294, -1.99898588

Iteration 35: 0.99959435, -1.99918870

Iteration 36: 0.99967548, -1.99935096

Iteration 37: 0.99974039, -1.99948077

Iteration 38: 0.99979231, -1.99958462

Iteration 39: 0.99983385, -1.99966769

Iteration 40: 0.99986708, -1.99973415

Iteration 41: 0.99989366, -1.99978732

Iteration 42: 0.99991493, -1.99982986

Iteration 43: 0.99993194, -1.99986389

Iteration 44: 0.99994555, -1.99989111

Iteration 45: 0.99995644, -1.99991289

Iteration 46: 0.99996516, -1.99993031

Iteration 47: 0.99997212, -1.99994425

Iteration 48: 0.99997770, -1.99995540

Iteration 49: 0.99998216, -1.99996432

Iteration 50: 0.99998573, -1.99997146

Iteration 51: 0.99998858, -1.99997716

Iteration 52: 0.99999087, -1.99998173

Iteration 53: 0.99999269, -1.99998538

Iteration 54: 0.99999415, -1.99998831

Iteration 55: 0.99999532, -1.99999065

Iteration 56: 0.99999626, -1.99999252
```

```
 Iteration 57: 0.99999701, -1.99999401

 Iteration 58: 0.99999761, -1.99999521
(0.9999976054757287, -1.9999952109514307, 3.0000000000286686)
```