Machine Learning 2023/2024 (2nd semester)

Master in Electrical and Computer Engineering

Department of Electrical and Computer Engineering

A. Pedro Aguiar (pedro.aguiar@fe.up.pt), **Aníbal Matos** (anibal@fe.up.pt), **Andry Pinto** (amgp@fe.up.pt), **Daniel Campos** (dfcampos@fe.up.pt), **Maria Inês Pereira** (up201505461@edu.fe.up.pt)

FEUP, Feb. 2024

Notebook #02: Fundamentals

3D and contour plots in Python

Matplotlib is a Python library for creating visualizations. Online doucmentation is available at https://matplotlib.org.

Example:

Let us produce some plots of the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x) = x_1^2 + 2x_2^2 + x_1 x_2$, where $x = (x_1, x_2)^\top$.

First, we plot the 3D surface defined by the function: points $(x_1, x_2, f(x))$ in $\mathbb{R}^3$.

Plot 3D surface - example

```
#@title Plot 3D surface - example
import numpy as np
import matplotlib.pyplot as plt

# sample x_1 and x_2 variables
```

```
MIN = -3
MAX = 3
STEP = 0.2
x1 = np.arange(MIN,MAX+STEP,STEP)
x2 = np.arange(MIN,MAX+STEP,STEP)

# generate x_1 and x_2 coordinates of a rectangular 2D grid
# these matrices will be used to compute function values
X1, X2 = np.meshgrid(x1, x2)
print(X1)

# generate an array with the values of the function
F = X1**2 + 2*X2**2 + X1*X2

# now plot the 3D surface
plot_size = 8
fig, ax = plt.subplots(figsize=(plot_size,plot_size))
ax = plt.axes(projection='3d')
ax.plot_surface(X1, X2, F, rstride=2, cstride=2, cmap='viridis', edgecolor='none
ax.set_title('surface');

plt.show()
```

```
[[-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
   2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
   1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
   1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
   2.60000000e+00  2.80000000e+00  3.00000000e+00]
 [-3.00000000e+00 -2.80000000e+00 -2.60000000e+00 -2.40000000e+00
  -2.20000000e+00 -2.00000000e+00 -1.80000000e+00 -1.60000000e+00
  -1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
```

Now compute the gradient of f, $\nabla f : \mathbb{R}^2 \to \mathbb{R}^2$,

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right)^\top = (2x_1 + x_2, x_1 + 4x_2)^\top$$

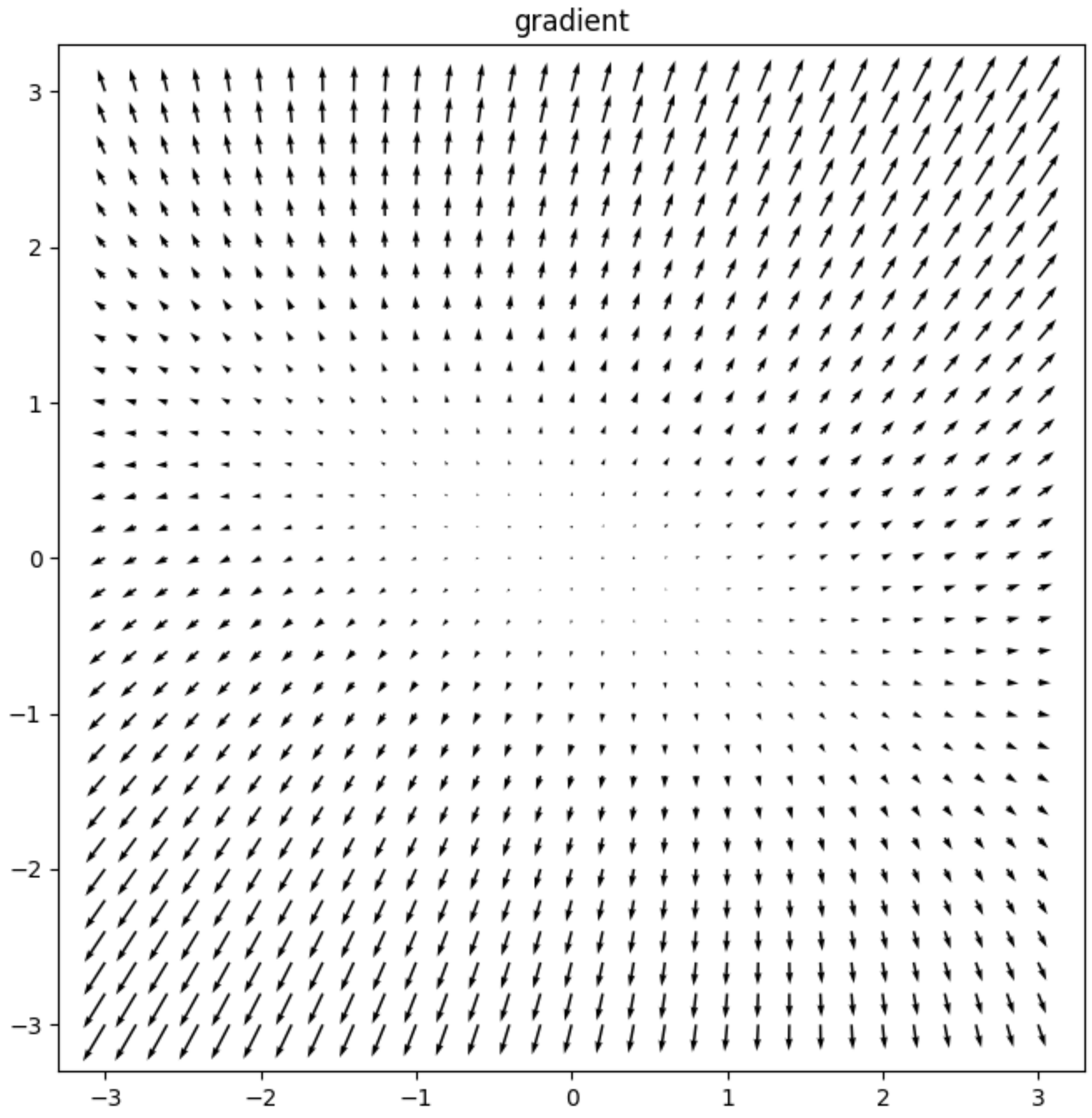
and plot at each point an arrow proportional to the gradient.

```
  -6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
```

Plot gradient field - example

```
#@title Plot gradient field - example

# compute gradient
dFdX1 = 2*X1+X2
dFdX2 = X1+4*X2

scale_factor = 0.1
plot_size = 8
```

```
fig, ax = plt.subplots(figsize=(plot_size,plot_size))
ax.quiver(X1,X2,scale_factor*dFdX1,scale_factor*dFdX2)

ax.set_aspect('equal')
ax.set_title('gradient')

plt.show()
```



```
-6.00000000e-01 -4.00000000e-01 -2.00000000e-01  2.66453526e-15
 2.00000000e-01  4.00000000e-01  6.00000000e-01  8.00000000e-01
 1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
 1.80000000e+00  2.00000000e+00  2.20000000e+00  2.40000000e+00
 2.60000000e+00  2.80000000e+00  3.00000000e+00]
```

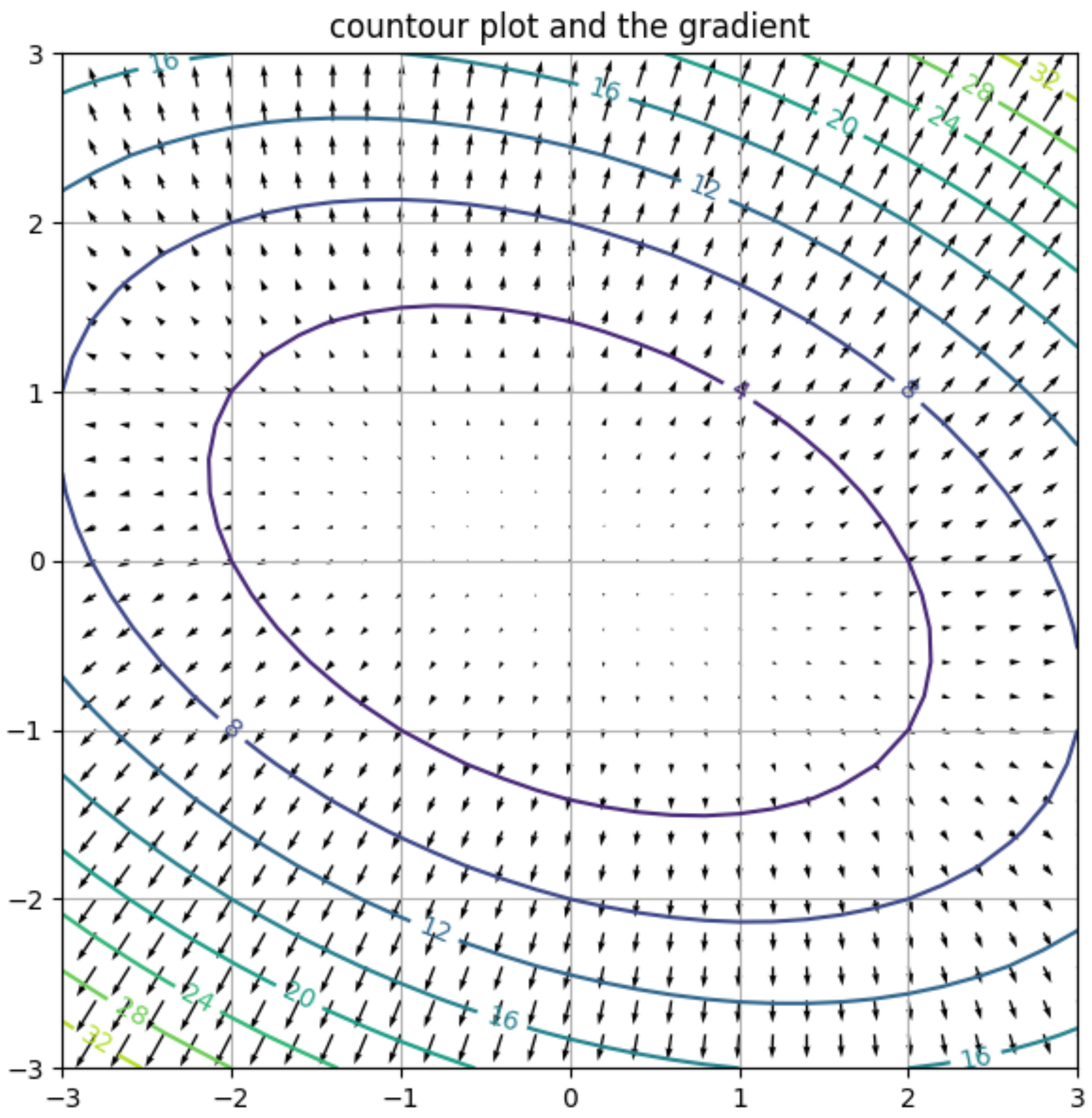
Let's now include some level curves...

As you can see the gradient vectors are perpendicular to the line tangent to the corresponding level curves!

```
      1.00000000e+00  1.20000000e+00  1.40000000e+00  1.60000000e+00
```

Plot level curves - example

```
#@title Plot level curves - example

plot_size = 7
fig, ax = plt.subplots(figsize=(plot_size,plot_size))

ax.quiver(X1,X2,scale_factor*dFdX1,scale_factor*dFdX2)

cs = ax.contour(X1,X2,F,10)

ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot and the gradient')
plt.grid()
plt.show()
```



```
-1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
```

Activity 1

Plot the surface of the Probability Density Function (PDF) of the 2D Gaussian distribution with mean $\mu = (2, 1)^\top$ and covariance $\Sigma = \begin{bmatrix} 1 & -1 \\ -1 & 1.5 \end{bmatrix}$.

Also obtain a contour plot of this PDF.

Note: The PDF of a d dimensional multivariate Gaussian distribution is given by

$$p(x \ \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d \ \Sigma}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1} (x-\mu) \right)$$

```
-1.40000000e+00 -1.20000000e+00 -1.00000000e+00 -8.00000000e-01
```

Plot 2D Gaussian PDF surface and contour lines

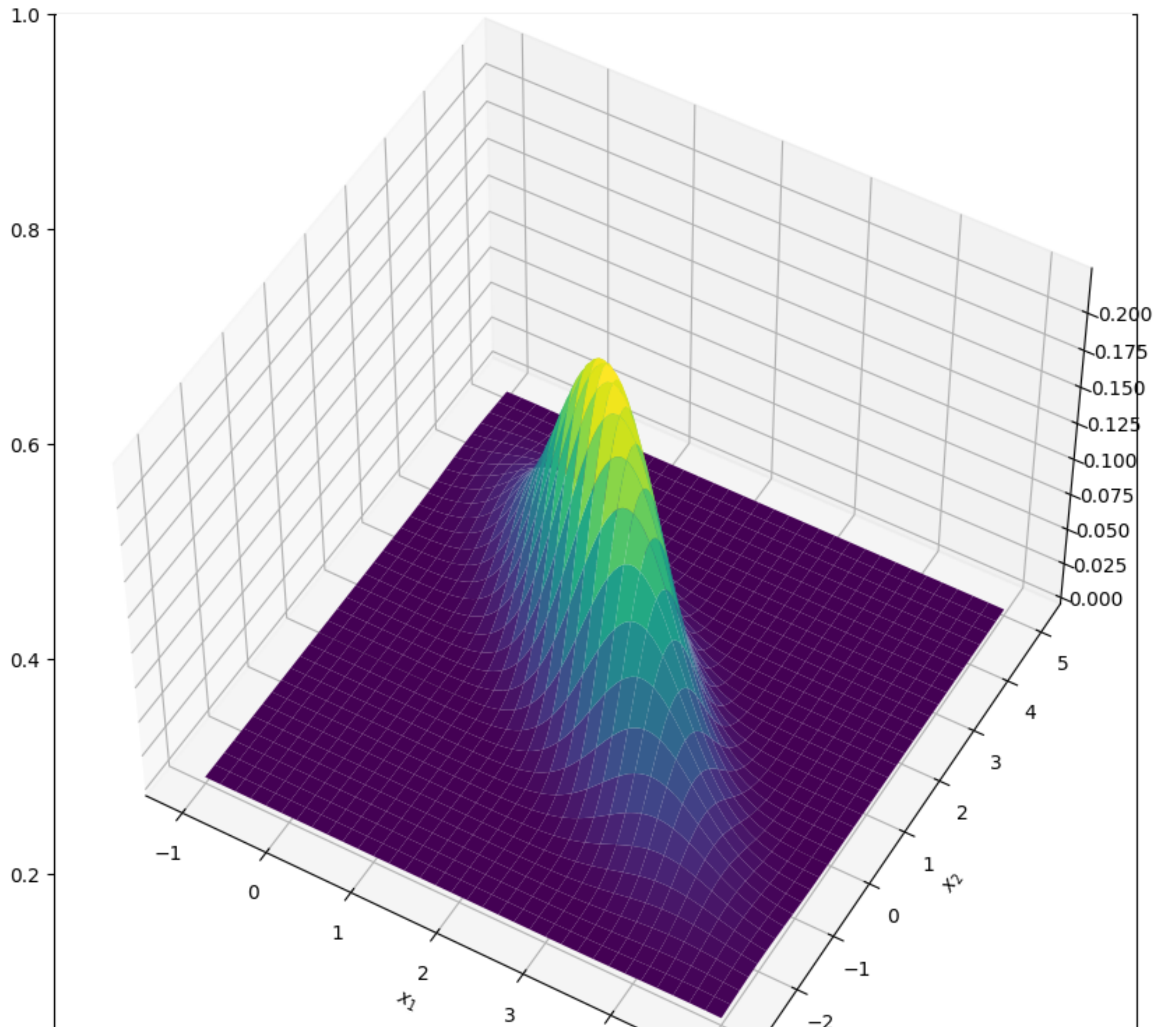
```
#@title Plot 2D Gaussian PDF surface and contour lines
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

def gaussian2D(X1,X2,mu,Sigma) :
    ''' return 2D array with values of PDF '''
    # Compute determinant and inverse of covariance matrix
    Sigma_det = np.linalg.det(Sigma)
    Sigma_inv = np.linalg.inv(Sigma)

    #F = ... (to complete..., we have to expand the vectorial form)

    c = np.sqrt((2*np.pi)**2 * Sigma_det)

    return np.exp(-F/2) / c


# Our 2-dimensional distribution will be over variables X1 and X2
N = 100
x1 = np.linspace(-1, 5, N)
x2 = np.linspace(-3, 5, N)
X1, X2 = np.meshgrid(x1, x2)


# Mean vector and covariance matrix
mu = np.array([2, 1])
Sigma = np.array([[ 1 , -1], [-1,  1.5]])


# compute PDF
Z=gaussian2D(X1,X2,mu,Sigma)


# plot using subplots
fig, ax = plt.subplots(figsize=(10,10))
ax = plt.axes(projection='3d')

ax.plot_surface(X1, X2, Z, rstride=3, cstride=3, cmap=cm.viridis)
ax.view_init(50,-60)
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.show()

fig, ax = plt.subplots(figsize=(10,10))
```

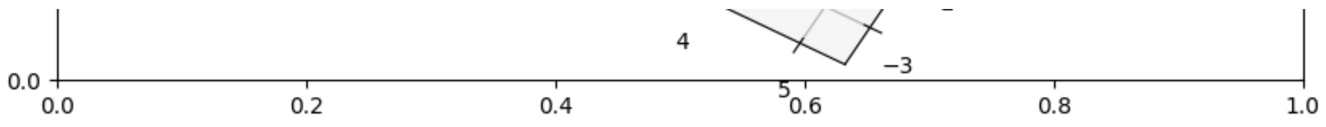
```
cs = ax.contour(X1, X2, Z, 10)
ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot')
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.grid()
plt.show()
```

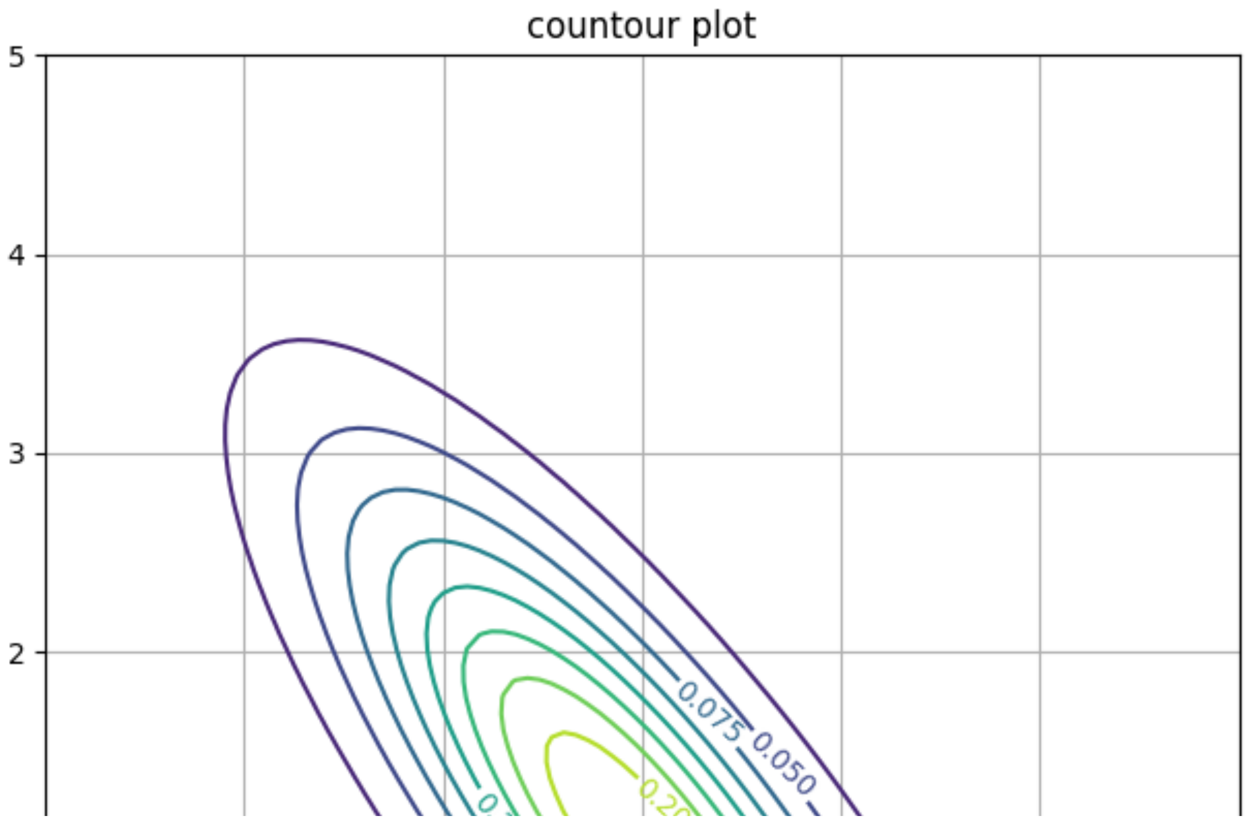
Plot 2D Gaussian PDF surface and contour lines (solution)

```
#@title Plot 2D Gaussian PDF surface and contour lines (solution)

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

def gaussian2D(X1,X2,mu,Sigma) :
    ''' return 2D array with values of PDF '''
    # Compute determinant and inverse of covariance matrix
    Sigma_det = np.linalg.det(Sigma)
    Sigma_inv = np.linalg.inv(Sigma)

    #F = ... (to complete..., we have to expand the vectorial form)
    F = Sigma_inv[0][0]*(X1-mu[0])**2+2*Sigma_inv[0][1]*(X1-mu[0])*(X2-mu[1])+S
    c = np.sqrt((2*np.pi)**2 * Sigma_det)

    return np.exp(-F/2) / c


# Our 2-dimensional distribution will be over variables X1 and X2
N = 100
x1 = np.linspace(-1, 5, N)
x2 = np.linspace(-3, 5, N)
X1, X2 = np.meshgrid(x1, x2)


# Mean vector and covariance matrix
mu = np.array([2, 1])
Sigma = np.array([[ 1 , -1], [-1,  1.5]])


# compute PDF
Z=gaussian2D(X1,X2,mu,Sigma)


# plot using subplots
```

```
fig, ax = plt.subplots(figsize=(10,10))
ax = plt.axes(projection='3d')

ax.plot_surface(X1, X2, Z, rstride=3, cstride=3, cmap=cm.viridis)
ax.view_init(50,-60)
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.show()

fig, ax = plt.subplots(figsize=(10,10))
cs = ax.contour(X1, X2, Z, 10)
ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot')
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.grid()
plt.show()
```







Activity 2

Plot the surface and the contour lines of

$$f(x) = \max\{p(x \;\mu_1, \Sigma_1), p(x \;\mu_2, \Sigma_2)\}$$

where $p(x \;\mu, \Sigma)$ is the Probability Density Function (PDF) of the 2D Gaussian distribution with mean μ and covariance Σ, and

$$\mu_1 = (2, 1)^\top$$

$$\Sigma_1 = \begin{bmatrix} 1 & -1 \\ -1 & 1.5 \end{bmatrix}$$

$$\mu_1 = (-1, 0)^\top$$

$$\Sigma_1 = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$

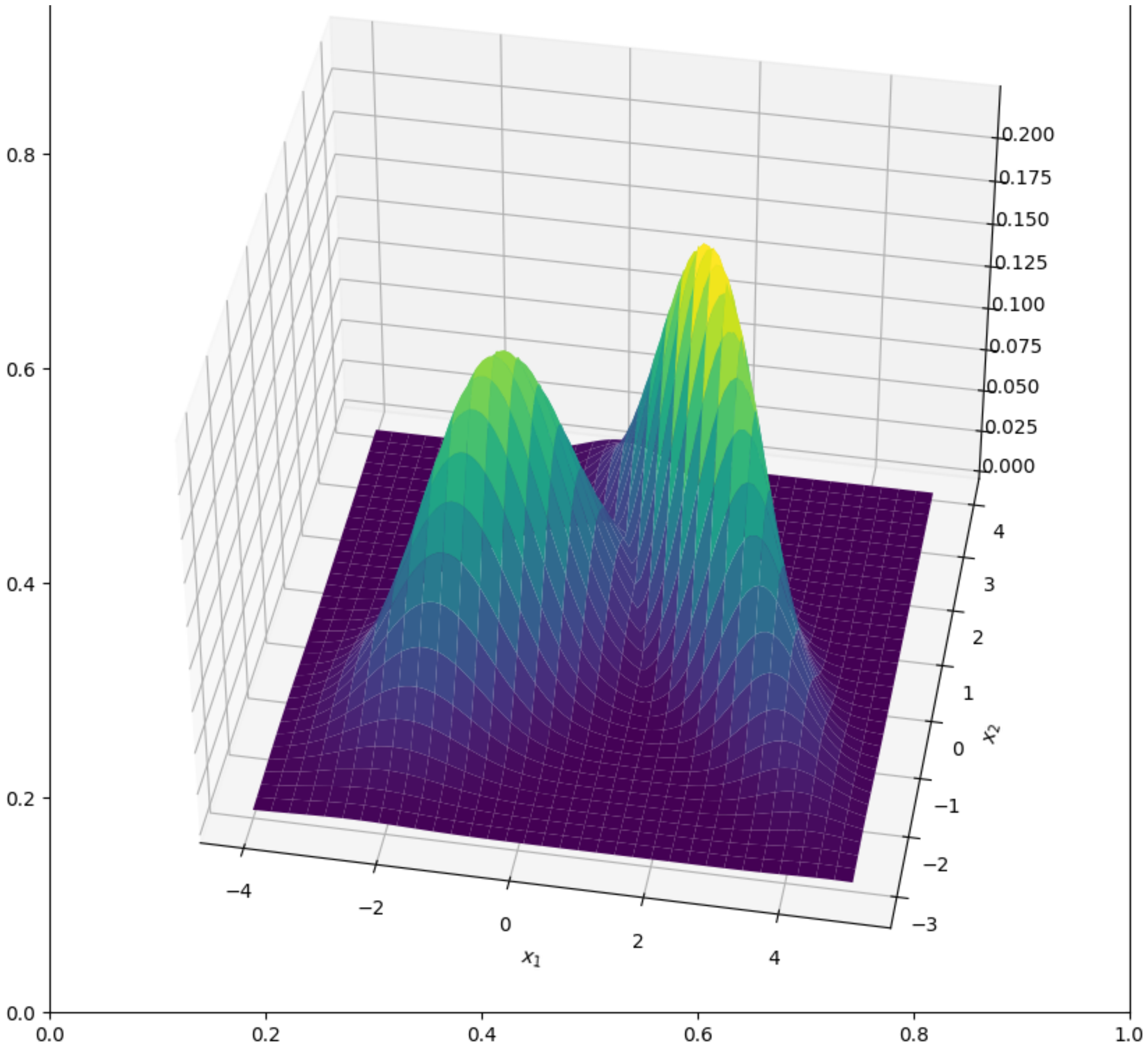
Activity 2 code

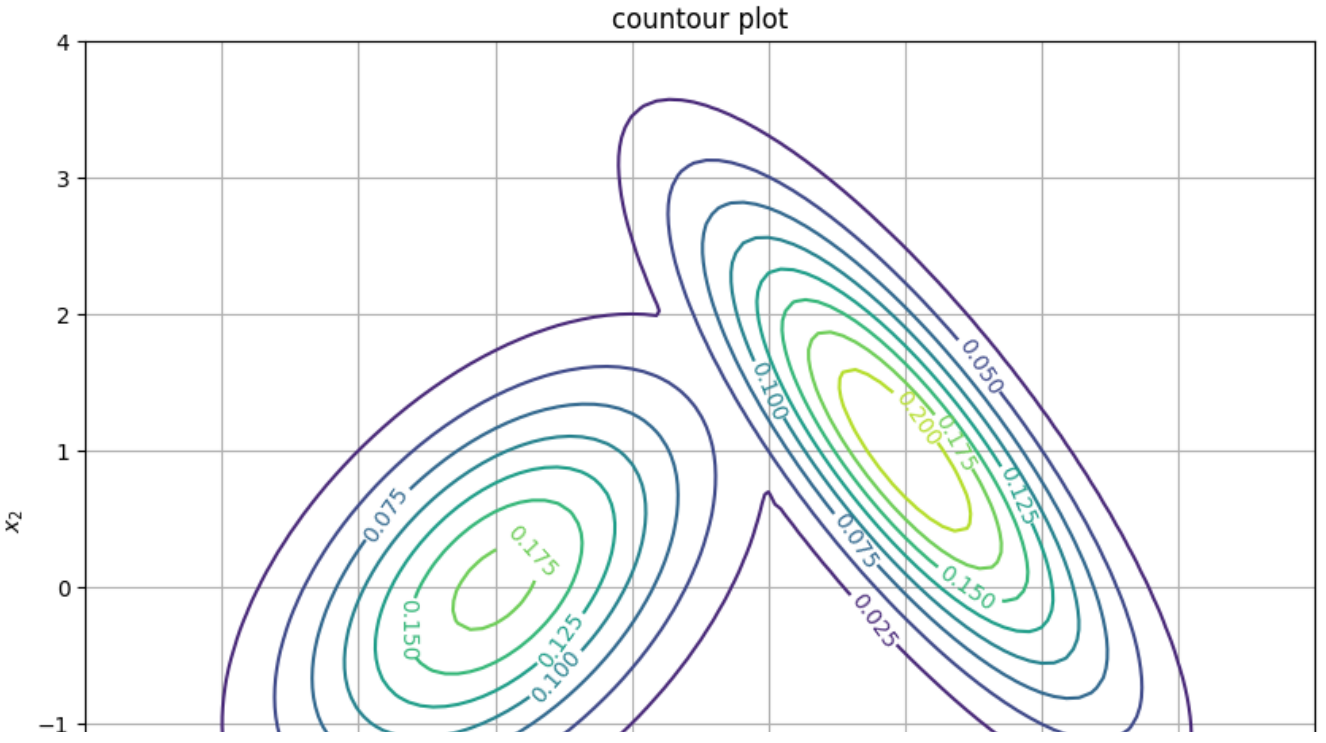
```
#@title Activity 2 code
```

```
# Our 2-dimensional distribution will be over variables X1 and X2
N = 100
x1 = np.linspace(-4, 5, N)
x2 = np.linspace(-3, 4, N)
X1, X2 = np.meshgrid(x1, x2)

# To complete...
# Mean vector and covariance matrix PDF1
# mu1 = ...
# Sigma1 = ...

# compute PDF1
# Z1 = ...

# Mean vector and covariance matrix PDF2
# mu2 = ...
# Sigma2 = ...

# compute PDF2
# Z2 = ...

# compute F
# F = ...

# plot using subplots
fig, ax = plt.subplots(figsize=(10,10))
ax = plt.axes(projection='3d')

ax.plot_surface(X1, X2, F, rstride=3, cstride=3, cmap=cm.viridis)
ax.view_init(60,-80)
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.show()

fig, ax = plt.subplots(figsize=(10,10))
cs = ax.contour(X1, X2, F, 10)
ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot')
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.grid()
plt.show()
```

Activity 2 code (solution)

```
#@title Activity 2 code (solution)

# Our 2-dimensional distribution will be over variables X1 and X2
N = 100
x1 = np.linspace(-4, 5, N)
x2 = np.linspace(-3, 4, N)
X1, X2 = np.meshgrid(x1, x2)


# Mean vector and covariance matrix PDF1
mu1 = np.array([2, 1])
Sigma1 = np.array([[ 1 , -1], [-1,  1.5]])

# compute PDF1
Z1=gaussian2D(X1,X2,mu1,Sigma1)

# Mean vector and covariance matrix PDF2
mu2 = np.array([-1, 0])
Sigma2 = np.array([[ 1 , 0.5], [0.5,  1]])

# compute PDF2
Z2=gaussian2D(X1,X2,mu2,Sigma2)

# compute F
F = np.maximum(Z1,Z2)

# plot using subplots
fig, ax = plt.subplots(figsize=(10,10))
ax = plt.axes(projection='3d')

ax.plot_surface(X1, X2, F, rstride=3, cstride=3, cmap=cm.viridis)
ax.view_init(40,-80)
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.show()

fig, ax = plt.subplots(figsize=(10,10))
cs = ax.contour(X1, X2, F, 10)
ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot')
ax.set_xlabel(r'$x_1$')
ax.set_ylabel(r'$x_2$')
plt.grid()
plt.show()
```


0.0
0.2
0.4
0.6
0.8
1.0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
−4
−2
0
2
4
x_1
−3
−2
−1
0
1
2
3
4
x_2


countour plot
−1
0
1
2
3
4
x_2
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200

Optimization

Activity 3

For each of the following functions $f : \mathbb{R}^2 \to \mathbb{R}$, check that the origin $x = (0, 0)^\top$ is a critical point, that is, the gradient at that point is zero. Also check whether it is a minimum point, a maximum point or a saddle point.

1. $f(x) = x_1^2 + 2x_2^2 - x_1x_2$
2. $f(x) = x_1^2 - x_2^2$
3. $f(x) = x_1^3 - x_2^3$
4. $f(x) = -x_1^2 + x_1x_2 - x_2^2$

A critical point x_0 of a differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is a point where the gradient of f is null, that is, $\nabla f(x_0) = 0$. Such point can be:

- a local maximum, if there exists a neighbourhood $\mathcal{N}$ of x_0 such that $f(x_0) \geq f(x)$ for all $x_0 \in \mathcal{N}$,
- a local miminum, if there exists a neighbourhood $\mathcal{N}$ of x_0 such that $f(x_0) \leq f(x)$ for all $x_0 \in \mathcal{N}$,
- a saddle point, if for any neighbourhood $\mathcal{N}$ of x_0 there are $x, y \in \mathcal{N}$ such that $f(x) < f(x_0) < f(y)$.

$H(x_0)$, the Hessian (matrix of second order partial derivatives) at a critical point x_0 can be used to classify the critical point according to:

- if all eigenvalues of $H(x_0)$ are positive then it is a local minimum,
- if all eigenvalues of $H(x_0)$ are negative then it is a local maximum,
- if $H(x_0)$ has both negative and positive eigenvalues then it is a saddle point,
- in other cases, this test in inconclusive.

Note: the Hessian of a twice continuously differentiable function is a symmetric matrix and, therefore, its eigenvalues are real.

For $f : \mathbb{R}^2 \to \mathbb{R}$, $(x, y) \mapsto f(x, y)$, the Hessian reduces to

$$H = \begin{bmatrix} f_{xx} & f_{xy} \\ f_{xy} & f_{yy} \end{bmatrix}.$$

Defining $D = \det(H) = f_{xx}f_{yy} - f_{xy}^2$, the conditions above reduce to:

- if $D > 0$ and $f_{xx} > 0$, the point is a local minimum,
- if $D > 0$ and $f_{xx} < 0$, the point is a local maximum,
- if $D < 0$, the point is a saddle point,
- if $D = 0$, the test is inconclusive.

Activity 3 solution (numerical)

```
#@title Activity 3 solution (numerical)
import sympy
import numpy as np

def check_classify_critical_at_origin(func):
  x1, x2 = sympy.symbols('x1 x2')
  grad = sympy.matrices.Matrix([sympy.diff(func,x1),sympy.diff(func,x2)])
  grad = grad.subs({x1:0,x2:0})
  grad = np.array(grad).astype(np.float64)
  if np.linalg.norm(grad)>0 : #not robust to finite precision arithmetic...
    return 'not a critical point'
  Hess = sympy.matrices.Matrix([[sympy.diff(func,x1,x1),sympy.diff(func,x1,x2)]
  Hess = Hess.subs({x1:0,x2:0})
  Hess = np.array(Hess).astype(np.float64)
  (val,vect) = np.linalg.eig(Hess)
  mx = np.max(val)
  mn = np.min(val)
  if mn>0 :
    res = 'minimum'
  elif mx<0 :
    res = 'maximum'
  elif mn<0 and mx>0 :
    res = 'saddle'
  else :
    res = 'inconclusive'
  return res


fx = ['x1**2+2*x2**2-x1*x2', 'x1**2-x2**2', 'x1**3-x2**3', '-x1**2+x1*x2-x2**2'
for f in fx :
  print('f =',f,' at (0,0):', check_classify_critical_at_origin(f))
```

```
    f = x1**2+2*x2**2-x1*x2  at (0,0): minimum
    f = x1**2-x2**2  at (0,0): saddle
    f = x1**3-x2**3  at (0,0): inconclusive
    f = -x1**2+x1*x2-x2**2  at (0,0): maximum
```

Solution

Eq.1

$f(x, y) = x^2 + 2y^2 - xy$

$\nabla f\ (x,\ y) = \left(\frac{\partial\ f}{\partial\ x},\ \frac{\partial\ f}{\partial\ y} \right) = (2x - y, 4y - x)$

$\nabla f(0, 0) = (0, 0)$

so $(0, 0)$ is a critical point

$f_{xx}(x, y) = 2$

$f_{xy}(x, y) = -1$

$f_{yy}(x, y) = 4$

$D(x, y) = f_{xx}(x, y) f_{yy}(x, y) - f_{xy}^2(x, y) = 7$

Since $D > 0$ and $f_{xx} > 0$, (0,0) is a local minimum.

Activity 3 - Eq 1 - plot

```
#@title Activity 3 - Eq 1 - plot

import numpy as np
import matplotlib.pyplot as plt

# sample x_1 and x_2 variables
MIN = -3
MAX = 3
STEP = 0.2
x1 = np.arange(MIN,MAX+STEP,STEP)
x2 = np.arange(MIN,MAX+STEP,STEP)

# generate x_1 and x_2 coordinates of a rectangular 2D grid
# these matrices will be used to compute function values
X1, X2 = np.meshgrid(x1, x2)

# generate an array with the values of the function
F = X1**2 + 2*X2**2 - X1*X2

# now plot the 3D surface
plot_size = 8
fig, ax = plt.subplots(figsize=(plot_size,plot_size))
ax = plt.axes(projection='3d')
ax.plot_surface(X1, X2, F, rstride=2, cstride=2, cmap='viridis', edgecolor='nor
ax.set_title('equation1');

plt.show()
```

Solution

Eq.2

$f(x, y) = x^2 - y^2$

$\nabla f\ (x,\ y) = \left(\frac{\partial\ f}{\partial\ x},\ \frac{\partial\ f}{\partial\ y}\right) = (2x, -2y)$

$\nabla f(0, 0) = (0, 0)$

so $(0, 0)$ is a critical point

$f_{xx}(x, y) = 2$

$f_{xy}(x, y) = 0$

$f_{yy}(x, y) = -2$

$D(x, y) = f_{xx}(x, y) f_{yy}(x, y) - f_{xy}^2(x, y) = -4$

Since $D = -4$, $(0, 0)$ is a saddle.

Activity 3 - Eq 2 - plot

```
#@title Activity 3 - Eq 2 - plot

import numpy as np
import matplotlib.pyplot as plt

# sample x_1 and x_2 variables
MIN = -3
MAX = 3
STEP = 0.2
x1 = np.arange(MIN,MAX+STEP,STEP)
x2 = np.arange(MIN,MAX+STEP,STEP)

# generate x_1 and x_2 coordinates of a rectangular 2D grid
# these matrices will be used to compute function values
X1, X2 = np.meshgrid(x1, x2)

# generate an array with the values of the function
F = X1**2 - X2**2

# now plot the 3D surface
plot_size = 8
fig, ax = plt.subplots(figsize=(plot_size,plot_size))
ax = plt.axes(projection='3d')
ax.plot_surface(X1, X2, F, rstride=2, cstride=2, cmap='viridis', edgecolor='nor
ax.set_title('Equation 2');

plt.show()
```

Solution

Eq.3

$f(x, y) = x^3 - y^3$

$\nabla f\,(x,\ y) = \left(\frac{\partial\, f}{\partial\, x},\ \frac{\partial\, f}{\partial\, y}\right) = \left(3x^2, -3y^2\right)$

$\nabla f(0, 0) = (0, 0)$

so $(0, 0)$ is a critical point

$f_{xx}(x, y) = 6x$

$f_{xy}(x, y) = 0$

$f_{yy}(x, y) = -6y$

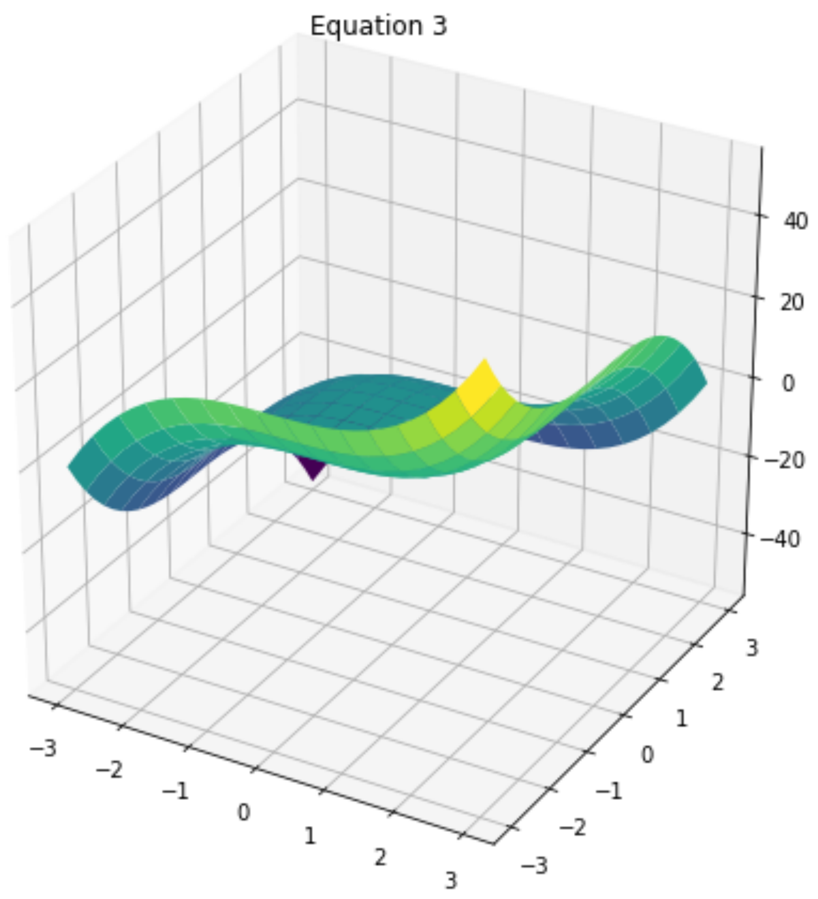
$D(x, y) = f_{xx}(x, y) f_{yy}(x, y) - f_{xy}^2(x, y) = -36xy$

Since $D(0, 0) = 0$, the second order test is inconclusive.

Activity 3 - Eq 3 - plot

```
#@title Activity 3 - Eq 3 - plot

import numpy as np
import matplotlib.pyplot as plt

# sample x_1 and x_2 variables
MIN = -3
MAX = 3
STEP = 0.2
x1 = np.arange(MIN,MAX+STEP,STEP)
x2 = np.arange(MIN,MAX+STEP,STEP)

# generate x_1 and x_2 coordinates of a rectangular 2D grid
# these matrices will be used to compute function values
X1, X2 = np.meshgrid(x1, x2)

# generate an array with the values of the function
F = X1**3 - X2**3
```

```
# now plot the 3D surface
plot_size = 8
fig, ax = plt.subplots(figsize=(plot_size,plot_size))
ax = plt.axes(projection='3d')
ax.plot_surface(X1, X2, F, rstride=2, cstride=2, cmap='viridis', edgecolor='nor
ax.set_title('Equation 3');

plt.show()
```



Solution

Eq. 4

$f(x, y) = -x^2 + xy - y^2$

$\nabla f\ (x,\ y) = \left(\frac{\partial\ f}{\partial\ x},\ \frac{\partial\ f}{\partial\ y}\right) = (2x + y, x - 2y)$

$\nabla f(0, 0) = (0, 0)$

so $(0, 0)$ is a critical point

$f_{xx}(x, y) = -2$

$f_{xy}(x, y) = 1$

$f_{yy}(x, y) = -2$

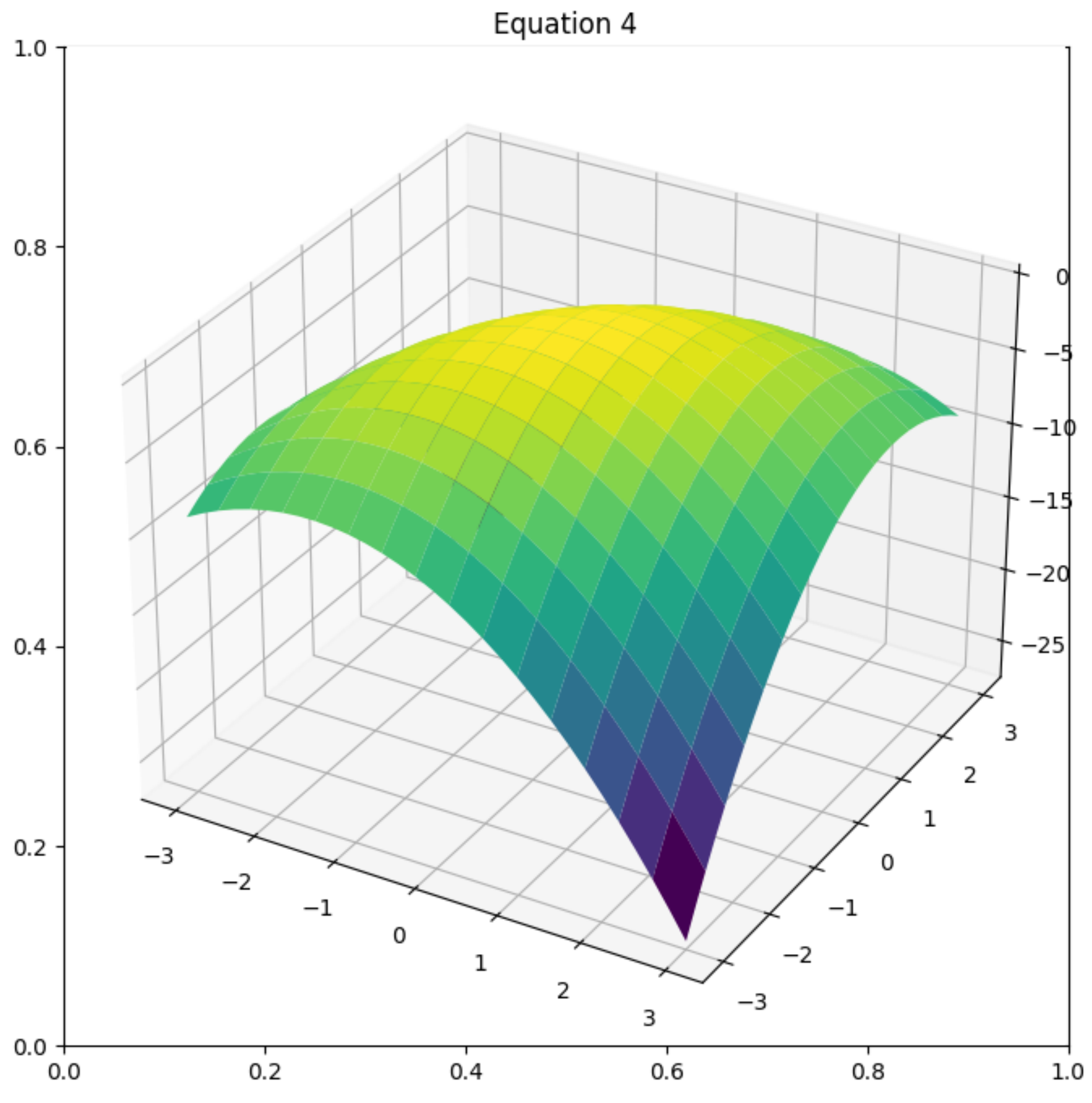
$D(x, y) = f_{xx}(x, y) f_{yy}(x, y) - f_{xy}^2(x, y) = 3$

Since $D > 0$ and $f_{xx} < 0$, $(0, 0)$ is a local maximum.

Activity 3 - Eq 4 - plot

```
#@title Activity 3 - Eq 4 - plot

import numpy as np
import matplotlib.pyplot as plt

# sample x_1 and x_2 variables
MIN = -3
MAX = 3
STEP = 0.2
x1 = np.arange(MIN,MAX+STEP,STEP)
x2 = np.arange(MIN,MAX+STEP,STEP)

# generate x_1 and x_2 coordinates of a rectangular 2D grid
# these matrices will be used to compute function values
X1, X2 = np.meshgrid(x1, x2)

# generate an array with the values of the function
F = -X1**2 + X1*X2- X2**2
```

```
# now plot the 3D surface
plot_size = 8
fig, ax = plt.subplots(figsize=(plot_size,plot_size))
ax = plt.axes(projection='3d')
ax.plot_surface(X1, X2, F, rstride=2, cstride=2, cmap='viridis', edgecolor='nor
ax.set_title('Equation 4');

plt.show()
```



Optimization methods

Consider the optimization problem

$$\min_{\theta \in \mathbb{R}^d} J(\theta)$$

A numerical (approximate) solution of such problem can be obtained using a **grid search**: discretizing the domain (in each dimension) and find the minimum value of J on that grid. Finer grids lead to more accurate results, but the search takes longer!

As the gradient of a function points towards the direction of maximum increase of the function, a simple iterative method can be implemented to seek the minimum. At a given point, θ_k, compute the gradient $\nabla J(\theta_k)$ and move to new point θ_{k+1} in the direction of $-\nabla J(\theta_k)$, that is, make

$$\theta_{k+1} = \theta_k - \gamma \nabla J(\theta_k),$$

where γ known as the step size or learning rate is a positive parameter. Note that for γ small enough $J(\theta_{k+1}) < J(\theta_k)$. This is the **gradient descent** method. Convergence of the method depends on the function J and also on the step γ, that can be different at each iteration.

Activity 4

Consider the function $J : \mathbb{R}^2 \to \mathbb{R}$, defined by

$$J(\theta) = \theta_1^2 + \theta_2^2 + 3(\theta_1 - 1)^2 + (\theta_2 - 1)^2 + \theta_1\theta_2$$

4.1 Grid search

Implement a grid search method to estimate the minimum of the function. Consider different grid spacings and compare resutls.

4.1 code

```
#@title 4.1 code

import numpy as np

def J_func(theta1,theta2) :
#  return (to complete)

theta1_min = 0
theta2_min = 0
J_min = float('inf')

XMIN = -3
XMAX = 3
DX = 0.01
YMIN = -3
YMAX = 3
DY = DX

for theta1 in np.arange(XMIN,XMAX+DX,DX) :
  for theta2 in np.arange(YMIN,YMAX+DY,DY) :
    J = J_func(theta1,theta2)
    # to complete...

print(f'min: {J_min} at ({theta1_min},{theta2_min})')
```

4.1 code (solution)

```
#@title 4.1 code (solution)

import numpy as np

def J_func(theta1,theta2) :
#  return (to complete)
  return theta1**2 + theta2**2 + 3*(theta1-1)**2 + (theta2-1)**2 + theta1*theta

theta1_min = 0
theta2_min = 0
J_min = float('inf')

XMIN = -3
XMAX = 3
DX = 0.01
YMIN = -3
YMAX = 3
DY = DX

for theta1 in np.arange(XMIN,XMAX+DX,DX) :
  for theta2 in np.arange(YMIN,YMAX+DY,DY) :
    J = J_func(theta1,theta2)
    if J < J_min :
      J_min = J
      theta1_min = theta1
      theta2_min = theta2

print(f'min: {J_min} at ({theta1_min},{theta2_min})')
```

```
min: 1.5484000000000007 at (0.7099999999999209,0.3199999999992923)
```

4.2 Gradient descent

Implement a gradient descent method to estimate the minimum of the function. Stop the method after a given number of iterations. Show the results along the iterations and produce a contour plot of J with the points along the iterations. Consider different step sizes and compare results.

Also determine the gradient of J and obtain the minimum by solving $\nabla J = 0$.

4.2 code

```
#@title 4.2 code

import numpy as np
import matplotlib.pyplot as plt

# cost function
def J_func(theta1,theta2) :
#  return (to complete)

# gradient
def J_grad(theta1,theta2) :
#  return (to complete)

# step size
gamma = 0.1

# number of iterations
MAX_ITER = 10

# collect points along iterations
points = np.zeros((MAX_ITER+1,2))

# initial point
theta1_0,theta2_0 = 0,0

###########################
# gradient descent method

points[0] = [theta1_0,theta2_0]

for i in range(MAX_ITER) :
  print(i,points[i],J_func(points[i][0],points[i][1]))
  #points[i+1][0] = ... (to complete)
  #points[i+1][1] = ... (to complete)

print(i+1,points[i+1],J_func(points[i+1][0],points[i+1][1]))

# draw contour lines
MIN = -0.5
MAX = 1.5
STEP = 0.1
theta1 = np.arange(MIN,MAX+STEP,STEP)
theta2 = np.arange(MIN,MAX+STEP,STEP)
Theta1, Theta2 = np.meshgrid(theta1, theta2)
J = Theta1**2 + Theta2**2 +3*(Theta1-1)**2 + (Theta2-1)**2

fig, ax = plt.subplots(figsize=(7,7))
```

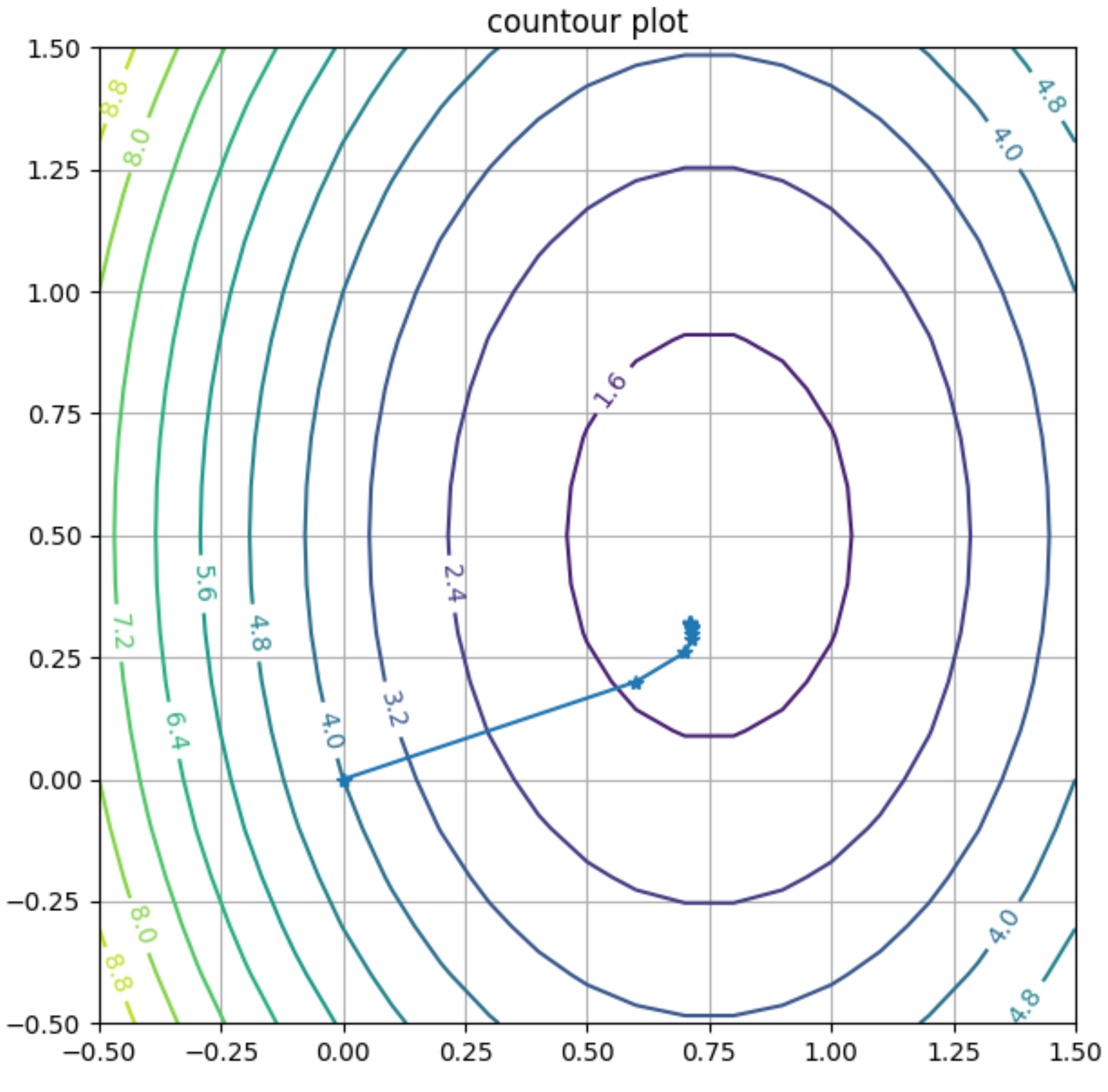
```
cs = ax.contour(Theta1,Theta2,J,10)
ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot')
plt.grid()

# draw sequence of solutions
plt.plot(points[:,0],points[:,1],'-*')

plt.show()
```

4.2 code (solution)

```
#@title 4.2 code (solution)

import numpy as np
import matplotlib.pyplot as plt

# cost function
def J_func(theta1,theta2) :
  return theta1**2 + theta2**2 + 3*(theta1-1)**2 + (theta2-1)**2 + theta1*theta

# gradient
def J_grad(theta1,theta2) :
  return 2*theta1 + 6*(theta1-1)+theta2, 2*theta2 + 2*(theta2-1)+theta1

# step size
gamma = 0.1

# number of iterations
MAX_ITER = 10

# collect points along iterations
points = np.zeros((MAX_ITER+1,2))

# initial point
theta1_0,theta2_0 = 0,0

###########################
# gradient descent method

points[0] = [theta1_0,theta2_0]

for i in range(MAX_ITER) :
  print(i,points[i],J_func(points[i][0],points[i][1]))
  dtheta1,dtheta2 = J_grad(points[i][0],points[i][1])
```

```
    points[i+1][0] = points[i][0] - gamma*dtheta1
    points[i+1][1] = points[i][1] - gamma*dtheta2

  print(i+1,points[i+1],J_func(points[i+1][0],points[i+1][1]))

  # draw contour lines
  MIN = -0.5
  MAX = 1.5
  STEP = 0.1
  theta1 = np.arange(MIN,MAX+STEP,STEP)
  theta2 = np.arange(MIN,MAX+STEP,STEP)
  Theta1, Theta2 = np.meshgrid(theta1, theta2)
  J = Theta1**2 + Theta2**2 +3*(Theta1-1)**2 + (Theta2-1)**2

  fig, ax = plt.subplots(figsize=(7,7))

  cs = ax.contour(Theta1,Theta2,J,10)
  ax.clabel(cs, inline=True, fontsize=10)
  ax.set_aspect('equal')
  ax.set_title('countour plot')
  plt.grid()

  # draw sequence of solutions
  plt.plot(points[:,0],points[:,1],'-*')

  plt.show()
```

```
0 [0. 0.] 4.0
1 [0.6 0.2] 1.6400000000000001
2 [0.7  0.26] 1.5572
3 [0.714 0.286] 1.5509799999999998
4 [0.7142 0.3002] 1.54936948
5 [0.71282 0.3087 ] 1.5487683236
6 [0.711694 0.313938] 1.548535325204
7 [0.710945  0.3171934] 1.54844473987012
8 [0.71046966 0.31922154] 1.5484095133104818
9 [0.71017178 0.32048596] 1.5483958142379743
10 [0.70998576 0.3212744 ] 1.548390486868892
```



4.3 Grid search ++

Modify the code in 2.1 to make an iterative refinement of the grid search. Start with a coarse grid in a given area and at each iteration reduce the search area and consider a finer grid. Stop when the spacing between grid points is less than a given value.

(solution)

```
#@title (solution)

import numpy as np

def J_func(theta1,theta2) :
#  return (to complete)
  return theta1**2 + theta2**2 + 3*(theta1-1)**2 + (theta2-1)**2 + theta1*theta


tol = 1e-6
div = 10

XMIN = -3
XMAX = 3
DX = 0.01
YMIN = -3
YMAX = 3
DY = DX

while DX>tol :
  theta1_min = 0
  theta2_min = 0
  J_min = float('inf')

  for theta1 in np.arange(XMIN,XMAX+DX,DX) :
    for theta2 in np.arange(YMIN,YMAX+DY,DY) :
      J = J_func(theta1,theta2)
      if J < J_min :
        J_min = J
        theta1_min = theta1
        theta2_min = theta2

  XMIN = theta1_min - DX
  XMAX = theta1_min + DX
  YMIN = theta2_min - DY
  YMAX = theta2_min + DY
  DX /= div
  DY /= div


print(f'min: {J_min} at ({theta1_min},{theta2_min})')
```

```
    min: 1.548387096775 at (0.7096769999999207,0.32258099999992895)
```

4.4 Gradient descent ++

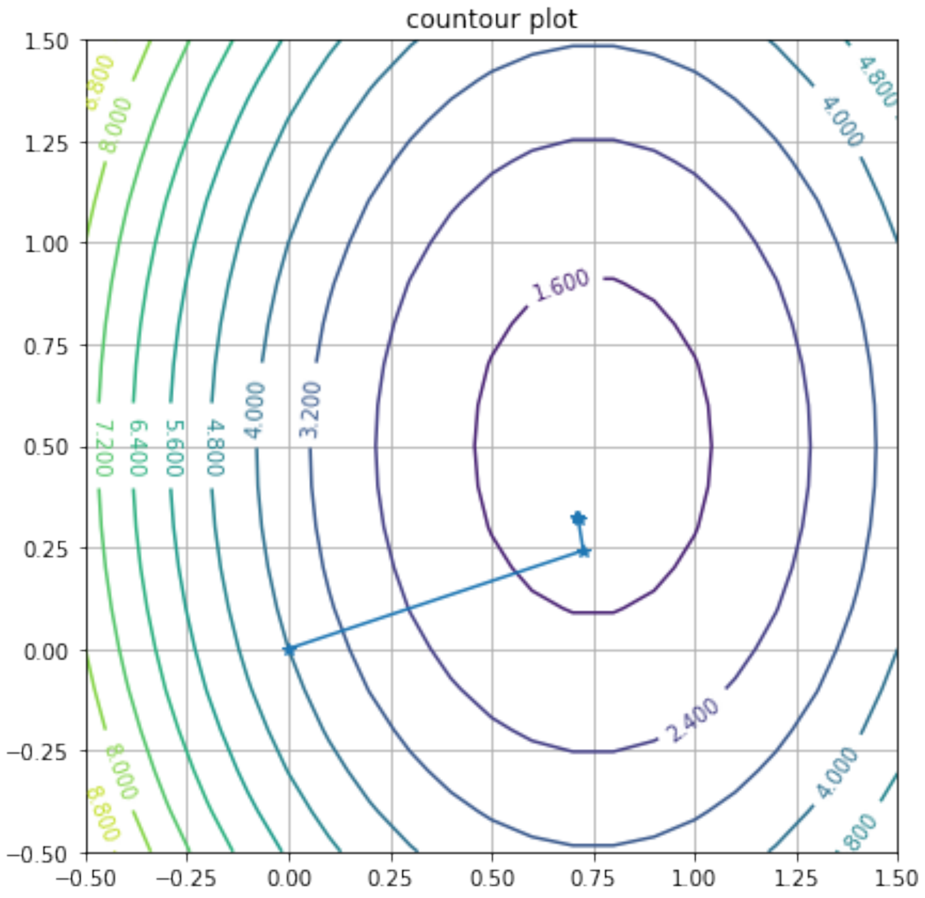
Modify the code in 2.4 to have an adaptive step. In each iteration,if the step is too large (resulting in a function increase) reduce it by a given factor (less than 1) until the reduction in the value on function is obtained. Otherwise increase it by another factor while the function value still decreases.

(solution)

```
#@title (solution)

import numpy as np
import matplotlib.pyplot as plt

# cost function
def J_func(theta1,theta2) :
  return theta1**2 + theta2**2 + 3*(theta1-1)**2 + (theta2-1)**2 + theta1*theta

# gradient
def J_grad(theta1,theta2) :
  return 2*theta1 + 6*(theta1-1)+theta2, 2*theta2 + 2*(theta2-1)+theta1

# step size
gamma = 0.1

# factors
factor_low = 0.8
factor_high = 1.1

# number of iterations
MAX_ITER = 10

# collect points along iterations
points = np.zeros((MAX_ITER+1,3))

# initial point
theta1_0,theta2_0 = 0,0

###########################
# gradient descent method

points[0] = [theta1_0,theta2_0,0]
```

```
for i in range(MAX_ITER) :
  J_base = J_func(points[i][0],points[i][1])
  print(i,points[i],J_base)
  dtheta1,dtheta2 = J_grad(points[i][0],points[i][1])
  t1_base, t2_base = points[i][0],points[i][1]
  t1, t2 = t1_base-gamma*dtheta1, t2_base-gamma*dtheta2
  J_new = J_func(t1,t2)
  if J_new >= J_base :
    while J_new >= J_base :
      gamma *= factor_low
      t1, t2 = t1_base-gamma*dtheta1, t2_base-gamma*dtheta2
      J_new = J_func(t1,t2)
  else :
    while True :
      gamma *= factor_high
      tt1, tt2 = t1_base-gamma*dtheta1, t2_base-gamma*dtheta2
      J_newnew = J_func(tt1,tt2)
      if J_newnew >= J_new : break
      t1, t2, J_new = tt1, tt2, J_newnew

  points[i+1][0] = t1
  points[i+1][1] = t2
  points[i+1][2] = gamma

print(i+1,points[i+1],J_func(points[i+1][0],points[i+1][1]))

# draw contour lines
MIN = -0.5
MAX = 1.5
STEP = 0.1
theta1 = np.arange(MIN,MAX+STEP,STEP)
theta2 = np.arange(MIN,MAX+STEP,STEP)
Theta1, Theta2 = np.meshgrid(theta1, theta2)
J = Theta1**2 + Theta2**2 +3*(Theta1-1)**2 + (Theta2-1)**2

fig, ax = plt.subplots(figsize=(7,7))

cs = ax.contour(Theta1,Theta2,J,10)
ax.clabel(cs, inline=True, fontsize=10)
ax.set_aspect('equal')
ax.set_title('countour plot')
plt.grid()

# draw sequence of solutions
plt.plot(points[:,0],points[:,1],'-*')

plt.show()
```

```
0 [0. 0. 0.] 4.0
1 [0.726  0.242  0.1331] 1.561124
2 [0.71303129 0.32136852 0.28531167] 1.5484309636933695
3 [0.70718381 0.32170967 0.22824934] 1.548415658223582
4 [0.71193593 0.32307403 0.25107427] 1.5484091014061674
5 [0.70820768 0.32222398 0.20085942] 1.5483965158685942
6 [0.710641   0.32280575 0.22094536] 1.5483911290055945
7 [0.70888807 0.32239391 0.24303989] 1.548389806195387
8 [0.71015217 0.32269261 0.19443191] 1.5483880765744984
9 [0.70939194 0.32251322 0.21387511] 1.548387451096624
10 [0.70989481 0.32263196 0.23526262] 1.5483873022340382
```



Activity 5

Repeat activity 4 with the function $J : \mathbb{R}^2 \to \mathbb{R}$, defined by

$$J(x, y) = \frac{2(x^2 + y^2)}{1 + x^2 + y^2} + \frac{(x-2)^2 + (y-1)^2}{1 + (x-2)^2 + (y-1)^2}$$

Note that this function has two minima (a local one and a global one). Check that depending on the starting point, the gradient descent method converges either to the local or the global minimum.