

Codice di laboratorio e domande orale

Alessandro Amella

22 gennaio 2024

Quest'opera è distribuita con licenza Creative Commons “Attribuzione – Non commerciale – Condividi allo stesso modo 4.0 Internazionale”.



Indice

Esercitazione 1: Epsilon macchina, Numpy, Matplotlib	2
Esercizio 1.1	2
Esercizio 2.1	3
Esercizio 2.2	3
Esercitazione 2: Norme, condizionamento, fattorizzazione LU	4
Esercizio 1.1	4
Esercizio 2.1	6
Esercizio 2.2	7
Esercitazione 3: Ricostruzione immagini con SVD	10
Esercizio 1.1	10
Esercizio 1.2	11
Esercitazione 4: Minimi quadrati con equazioni normali e SVD	12
Esercizio 1.1	12
Esercizio 1.2	14
Esercitazione 5: Metodi iterativi per radici di funzioni	16
Esercizio 1.1	16
Esercizio 1.2	19
Esercitazione 6: Minimizzazione e metodi di discesa del gradiente	21
Esercizio 1.1	21
Esercizio 1.2	22
Esercizio 1.3	25
Esercitazione 7: Deblurring di immagini	26
Esercizio 1.1 Problema test	26
Esercizio 1.2 Soluzione naiva	29
Esercizio 1.3 Soluzione regolarizzata	30
Esercizio 1.4	33
Domanda 1: Fattorizzazione LU con pivoting	41
Domanda 2: Fattorizzazione con Cholesky	41
Domanda 3: Cholesky con matrice di Hilbert	42

Domanda 4: Compressione SVD	42
Domanda 5: Interpolazione polinomiale	42
Domanda 6: Interpolazione polinomiale	44
Domanda 7: Interpolazione polinomiale	45
Domanda 8: Calcolo zero funzione	45
Domanda 9: Calcolo zero funzione	47
Domanda 10: Calcolo zero funzione	47
Domanda 11: Metodo del gradiente	48
Domanda 12: Metodo del gradiente	49
Domanda 13: Deblur	50
Domanda 14: Super Resolution	51

Esercitazione 1: Epsilon macchina, Numpy, Matplotlib

Esercizio 1.1

Machine epsilon or machine precision is an upper bound on the relative approximation error due to rounding in floating point arithmetic.

- Write a code to compute the machine precision ε in (float) default precision with a `while` construct. Compute also the mantissa digits number.

```
def machine_epsilon(func=float):
    eps = func(1)
    while func(1) + func(eps) != func(1):
        eps_last = eps
        eps = func(eps) / func(2)
    return eps_last

print(machine_epsilon(float)) # 2.220446049250313e-16

def mantissa_digits(eps):
    return -math.log10(eps)

print(mantissa_digits(machine_epsilon(float))) # 15.653559774527022
```

- Use NumPy and exploit the functions `float16` and `float32` in the `while` statement and see the differences. Check the result of `np.finfo(float).eps`.

```
import numpy as np
print(machine_epsilon(np.float16)) # 0.000977
print(machine_epsilon(np.float32)) # 1.1920929e-07

print(np.finfo(float).eps) # 2.220446049250313e-16
print(np.finfo(np.float16).eps) # 0.000977
print(np.finfo(np.float32).eps) # 1.1920929e-07
```

Esercizio 2.1

Create a figure combining together the cosine and sine curves, on the domain [0, 10].

```
import matplotlib.pyplot as plt
import numpy as np

# Array [0, 10] con 100 elementi
x = np.linspace(0, 10, 100)

# Calcolo cos(x) e sin(x)
y1 = np.cos(x)
y2 = np.sin(x)

# Mostra i grafici
plt.plot(x, y1, color='red', label='cos')
plt.plot(x, y2, color='blue', label='sin')

# Aggiungi legenda, titolo e label
plt.legend()
plt.title('Grafico seno e coseno')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

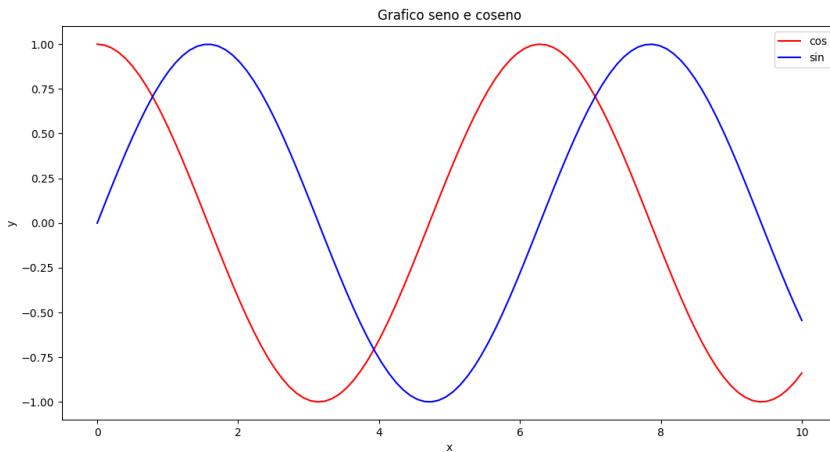


Figura 1: Funzioni $\sin(x)$ e $\cos(x)$

Esercizio 2.2

1. Write a script that, given an input number n , computes the number F_n of the Fibonacci sequence.

```
def fib_iter(n):
    a = 0 # F(0)
    b = 1 # F(1)
    for i in range(n):
        c = a + b
        a = b
        b = c
```

```

    b = c
    return a

print(fib_iter(10)) # 55

```

2. Write a code computing, for a natural number k , the ratio $r_k = \frac{F_{k+1}}{F_k}$, where F_k are the Fibonacci numbers.

```

def fib_ratio(k):
    a = fib_iter(k)
    b = fib_iter(k - 1)
    if b == 0:
        return math.inf
    return a / b

print(fib_ratio(10)) # 1.6176470588235294

```

3. Verify that, for a large k , $\{r_k\}_k$ converges to the value $\phi = \frac{1+\sqrt{5}}{2}$.

```

phi = (1 + math.sqrt(5)) / 2
def fib_ratio_convergence(k):
    return phi - fib_ratio(k)

print(fib_ratio_convergence(10)) # 0.00038692992636546464

```

4. Create a plot of the error (with respect to φ).

```

errors = []
for n in range(30):
    r = fib_ratio(n)
    e = abs(phi - r)
    errors.append(e)
plt.plot(errors)
plt.title("Errore approssimazione di Fibonacci")
plt.xlabel("n")
plt.ylabel("Errore")
plt.show()

```

Esercitazione 2: Norme, condizionamento, fattorizzazione LU

Esercizio 1.1

- Calcolare la norma 1, la norma 2, la norma Frobenius e la norma infinito di A con `numpy.linalg.norm()`

```

import numpy as np

n = 2
A = np.array([[1, 2], [0.499, 1.001]])

print ('Norme di A:')
norm1 = np.linalg.norm(A, 1)
norm2 = np.linalg.norm(A, 2)
normfro = np.linalg.norm(A, 'fro')
norminf = np.linalg.norm(A, np.inf)

```

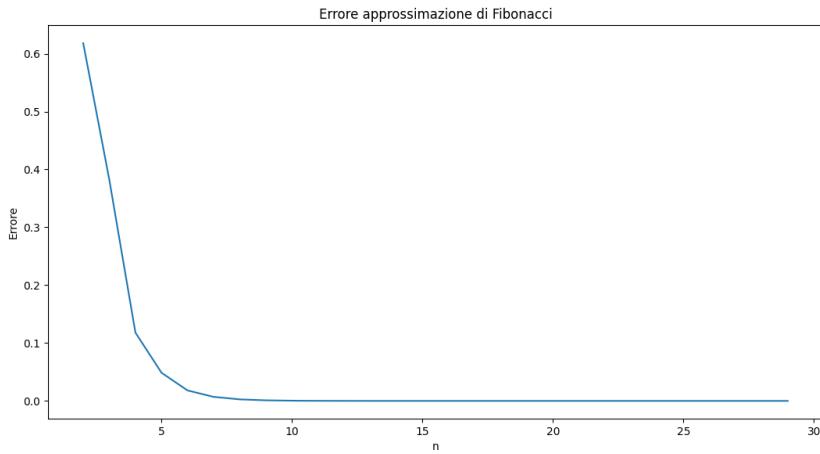


Figura 2: Errore rispetto a φ

```
print('Norma1 = ', norm1, '\n') # 3.001
print('Norma2 = ', norm2, '\n') # 2.500200104037774
print('Normafro = ', normfro, '\n') # 2.5002003919686118
print('Norma infinito = ', norminf, '\n') # 3.0
```

- Calcolare il numero di condizionamento di A con `numpy.linalg.cond()` (guardare l'help della funzione).

```
cond1 = np.linalg.cond(A, 1)
cond2 = np.linalg.cond(A, 2)
condfro = np.linalg.cond(A, 'fro')
condinf = np.linalg.cond(A, np.inf)

print ('K(A)_1 = ', cond1, '\n') # 3001.0000000001082
print ('K(A)_2 = ', cond2, '\n') # 2083.666853410337
print ('K(A)_fro = ', condfro, '\n') # 2083.6673333334084
print ('K(A)_inf = ', condinf, '\n') # 3001.0000000001082
```

- Considerare il vettore colonna $x = (1, 1)^T$ e calcolare il corrispondente termine noto b per il sistema lineare $Ax = b$.

```
x = np.ones((2,1))
b = A @ x
print ('b = ', b) # [[3.    ], [1.5]]
```

- Considerare ora il vettore $\tilde{b} = (3, 1.4985)^T$ e verifica che $\tilde{x} = (2, 0.5)^T$ è soluzione del sistema $A\tilde{x} = \tilde{b}$.

```
btilde = np.array([[3], [1.4985]])
xtilde = np.array([[2, 0.5]]).T
my_btilde = A @ xtilde
print ('A*xtilde = ', btildes) # [[3.    ], [1.4985]]
```

- Calcolare la norma 2 della perturbazione sui termini noti $\Delta b = \|b - \tilde{b}\|_2$ e la norma 2 della perturbazione sulle soluzioni $\Delta x = \|x - \tilde{x}\|_2$. Confrontare Δb con Δx .

```
deltax = np.linalg.norm(x-xtilde, ord=2)
deltab = np.linalg.norm(b-btilde, ord=2)
```

```

print ('delta x = ', deltax) # 1.118033988749895
print ('delta b = ', deltab) # 0.0015000000000000568

```

Esercizio 2.1

- Creare il problema test in cui il vettore della soluzione esatta è $\mathbf{x} = (1, 1, 1)^T$ e il vettore termine noto è $\mathbf{b} = A\mathbf{x}$.

```

import numpy as np

A = np.array ([[ 3,-1, 1,-2], [0, 2, 5, -1], [1, 0, -7, 1], [0, 2, 1, 1] ])
x = np.ones((4,1))
b = A @ x

condA = np.linalg.cond(A, 2)

print('x: \n', x , '\n') # [[1.], [1.], [1.], [1.]]
print('x.shape: ', x.shape, '\n') # (4, 1)
print('b: \n', b , '\n') # [[1.], [6.], [-5.], [4.]]
print('b.shape: ', b.shape, '\n') # (4, 1)
print('A: \n', A, '\n') # ...
print('A.shape: ', A.shape, '\n') # (4, 4)
print('K(A)=', condA, '\n') # 14.208370392921381

```

- Guardare l'help della funzione `scipy.linalg.lu_factor` e `scipy.linalg.lu` e utilizzare una delle sue funzioni per calcolare la fattorizzazione LU di A con pivoting. Verificare la correttezza dell'output.

```

import scipy
import scipy.linalg
from scipy.linalg import lu_factor as LUdec # pivoting
from scipy.linalg import lu as LUfull # partial pivoting

lu, piv = LUdec(A)

print('lu',lu,'\\n')
print('piv',piv,'\\n')

lu [[ 3.         -1.          1.         -2.          ]
 [ 0.          2.          5.         -1.          ]
 [ 0.33333333  0.16666667 -8.16666667  1.83333333]
 [ 0.          1.          0.48979592  1.10204082]]

piv [0 1 2 3]

```

- Risolvere il sistema lineare con la funzione `scipy.linalg.lu_solve` oppure utilizzando la funzione `scipy.linalg.solve_triangular`.

```

my_x = scipy.linalg.lu_solve((lu, piv), b)
print('my_x = \\n', my_x)
print('norm =', scipy.linalg.norm(x-my_x, 'fro'))

# IMPLEMENTAZIONE ALTERNATIVA - 1
P, L, U = LUfull(A)
print ('P*L*U = ', np.matmul(P , np.matmul(L, U)))
print ('diff = ', np.linalg.norm(A - np.matmul(P , np.matmul(L, U)), 'fro' ) )

```

```

invP = np.linalg.inv(P)
y = scipy.linalg.solve_triangular(L, invP @ b, lower=True, unit_diagonal=True)
my_x = scipy.linalg.solve_triangular(U, y, lower=False)
print('\nSoluzione calcolata: ', my_x)
print('norm =', scipy.linalg.norm(x-my_x, 'fro'))

```

```

Soluzione calcolata:  [[1.]
 [1.]
 [1.]
 [1.]]

```

- Stampare la soluzione calcolata e valutarne la correttezza.

```

my_x =
[[1.]
[1.]
[1.]
[1.]]
norm = 7.021666937153402e-16

# IMPLEMENTAZIONE ALTERNATIVA - 1

P = [[1. 0. 0. 0.]
[0. 1. 0. 0.]
[0. 0. 1. 0.]
[0. 0. 0. 1.]]
L = [[1. 0. 0. 0.]
[0. 1. 0. 0.]
[0.33333333 0.16666667 1. 0.]
[0. 1. 0.48979592 1. ]]
U = [[3. -1. 1. -2.]
[0. 2. 5. -1.]
[0. 0. -8.1666667 1.83333333]
[0. 0. 0. 1.10204082]]
P*L*U = [[3. -1. 1. -2.]
[0. 2. 5. -1.]
[1. 0. -7. 1.]
[0. 2. 1. 1.]]
diff = 1.5700924586837752e-16

Soluzione calcolata:  [[1.]
[1.]
[1.]
[1.]]
norm = 5.438959822042073e-16

```

Esercizio 2.2

Si ripeta l'esercizio precedente sulla matrice di Hilbert, che si può generare con la funzione $A = \text{scipy.linalg.hilbert}(n)$ per $n = 5, \dots, 10$. In particolare:

- Calcolare il numero di condizionamento di A e rappresentarlo in un grafico al variare di n .

```

n = 5
A = scipy.linalg.hilbert(n)
x = np.ones((n,1))

```

```

b = A @ x

condA = np.linalg.cond(A, 2)

print('x: \n', x, '\n')
print('x.shape: ', x.shape, '\n')
print('b: \n', b, '\n')
print('b.shape: ', b.shape, '\n')
print('A: \n', A, '\n')
print('A.shape: ', A.shape, '\n')
print('K(A)=', condA, '\n') # 476607.2502425855

```

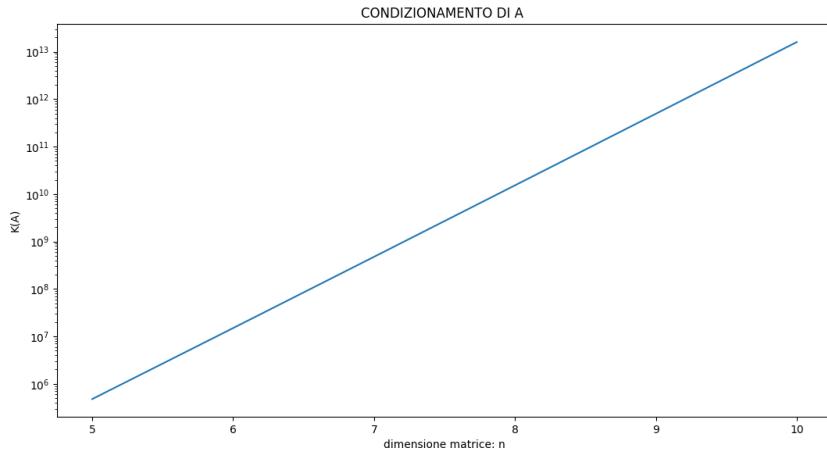


Figura 3: $K(A)$ al variare di n

- Considerare il vettore colonna $x = (1, \dots, 1)^T$, calcola il corrispondente termine noto b per il sistema lineare $Ax = b$ e la relativa soluzione \tilde{x} usando la fattorizzazione di Cholesky come nel caso precedente.

```

L = scipy.linalg.cholesky(A, lower=True)
print('L:', L, '\n')

print('L.T*L =', scipy.linalg.norm(A-np.matmul(np.transpose(L),L))) # 0.9734839217207908
print('err = ', scipy.linalg.norm(A-np.matmul(np.transpose(L),L), 'fro')) # 0.9734839217207908

y = scipy.linalg.solve_triangular(L, b, lower=True)
my_x = scipy.linalg.solve_triangular(L.T, y, lower=False)
print('my_x = \n', my_x)

print('norm =', np.linalg.norm(x-my_x, 'fro')) # 2.6732599660997558e-12

```

```

K_A = np.zeros((6,1))
Err = np.zeros((6,1))

```

- Si rappresenti l'errore relativo al variare delle dimensioni della matrice.

```

for n in np.arange(5,11):
    # creazione dati e problema test
    A = scipy.linalg.hilbert(n)

```

```

x = np.ones((n,1))
b = A @ x

# numero di condizione
K_A[n-5] = np.linalg.cond(A, 2)

# fattorizzazione
L = scipy.linalg.cholesky(A, lower=True)
y = scipy.linalg.solve_triangular(L, b, lower=True)
my_x = scipy.linalg.solve_triangular(L.T, y, lower=False)

# errore relativo
Err[n-5] = np.linalg.norm(x-my_x, 'fro')/np.linalg.norm(x, 'fro')

xplot = np.arange(5,11)

# grafico del numero di condizione vs dim
plt.semilogy(xplot, K_A)
plt.title('CONDIZIONAMENTO DI A ')
plt.xlabel('dimensione matrice: n')
plt.ylabel('K(A)')
plt.show()

# grafico errore in norma 2 in funzione della dimensione del sistema
plt.plot(xplot, Err)
plt.title('Errore relativo')
plt.xlabel('dimensione matrice: n')
plt.ylabel('Err= ||my_x-x||/||x||')
plt.show()

```

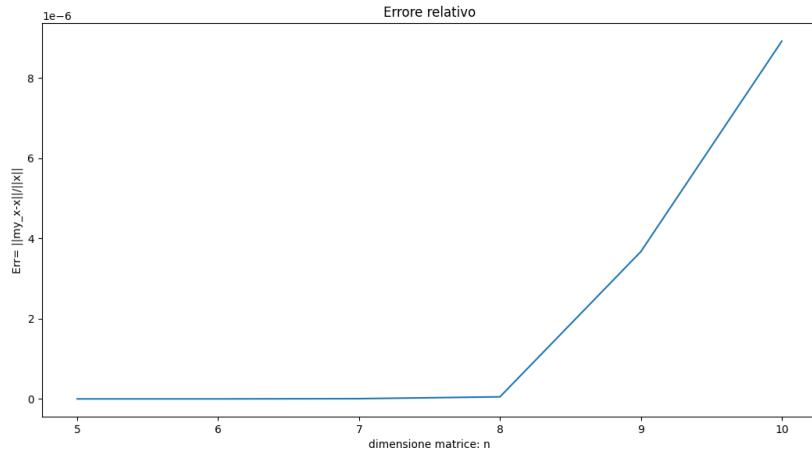


Figura 4: $K(A)$ al variare di n

Esercitazione 3: Ricostruzione immagini con SVD

Esercizio 1.1

Utilizzando la libreria `skimage`, nello specifico il modulo `data`, caricare e visualizzare un'immagine A (diversa dal cameraman) in scala di grigio di dimensione $m \times n$.

- Calcolare la matrice

$$A_p = \sum_{i=1}^p u_i \cdot v_i^T \cdot \sigma_i$$

dove $p \leq \text{rango}(A)$.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg
from skimage import data

A = data.camera()

U, s, Vh = np.linalg.svd(A)

A_p = np.zeros(A.shape)
p_max = 10

for i in range(p_max):
    ui = U[:, i]
    vi = Vh[i, :]

    A_p = A_p + s[i] * np.outer(ui, vi)
```

- Visualizzare l'immagine A_p .

```
plt.figure(figsize=(20, 10))

fig1 = plt.subplot(1, 2, 1)
fig1.imshow(A, cmap='gray')
plt.title('True image')

fig2 = plt.subplot(1, 2, 2)
fig2.imshow(A_p, cmap='gray')
plt.title('Reconstructed image with p = ' + str(p_max))

plt.show()
```

- Calcolare l'errore relativo: $\frac{\|A - A_p\|_2}{\|A\|_2}$.

```
err_rel = np.linalg.norm(A - A_p) / np.linalg.norm(A)
print('L\'errore relativo della ricostruzione di A è', err_rel) # 0.36547368241427036
```

- Calcolare il fattore di compressione

$$c_p = \frac{1}{p \cdot \min(m, n) - 1}.$$

```
c = 1 / p_max * (min(A.shape) - 1)
print('Il fattore di compressione è c=', c) # 47.900000000000006
```

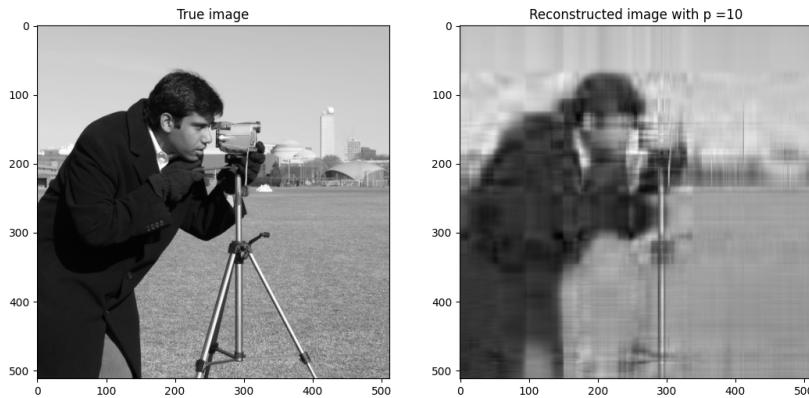


Figura 5: A_p ricostruita con $p = 10$

- Calcolare e plottare l'errore relativo e il fattore di compressione al variare di p .

```

p_max = 100
A_p = np.zeros(A.shape)
err_rel = np.zeros((p_max))
c = np.zeros((p_max))

for i in range(p_max):
    ui = U[:, i]
    vi = Vh[i, :]

    A_p = A_p + s[i] * np.outer(ui, vi)

    err_rel[i] = np.linalg.norm(A - A_p) / np.linalg.norm(A)
    c[i] = 1 / (i + 1) * (min(A.shape) - 1)

plt.figure(figsize=(10, 5))

fig1 = plt.subplot(1, 2, 1)
fig1.plot(err_rel, 'o-')
plt.title('Errore relativo')

fig2 = plt.subplot(1, 2, 2)
fig2.plot(c, 'o-')
plt.title('Fattore di compressione')

plt.show()

```

Esercizio 1.2

Eseguire l'esercizio precedente caricando un'immagine da un file usando la funzione `skimage.io.imread`.

```

from skimage.io import imread
import os

```

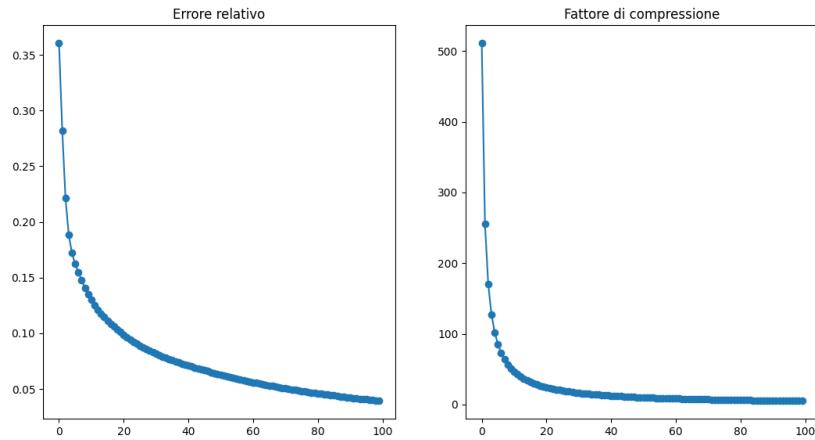


Figura 6: Errore relativo e fattore di compressione al variare di p

```
# uso immagine ./phantom.png
img = 'phantom.png'
cur_dir = os.path.dirname(os.path.abspath(__file__))
A = imread(os.path.join(cur_dir, img), as_gray=True)
```

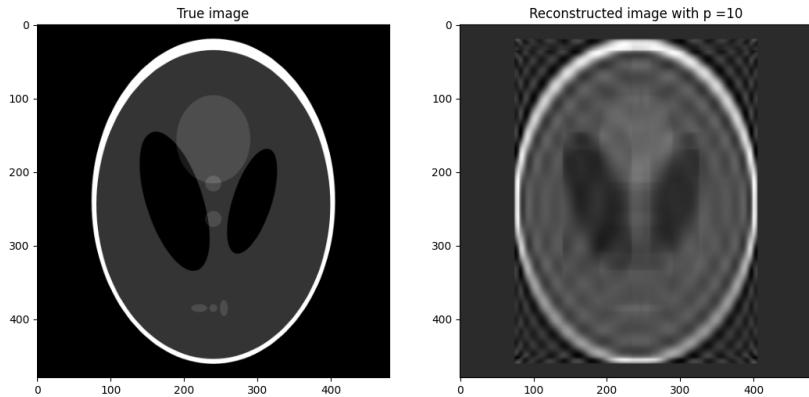


Figura 7: A_p ricostruita con $p = 10$

Esercitazione 4: Minimi quadrati con equazioni normali e SVD

Esercizio 1.1

Assegnata una matrice A di numeri casuali di dimensione $m \times n$ con $m > n$, generata utilizzando la funzione `np.random.rand`, scegliere un vettore α (per esempio con elementi costanti) come soluzione per creare un problema test e calcolare il termine noto $y = A\alpha$.

Definito quindi il problema dei minimi quadrati con la matrice A e il termine noto y calcolato:

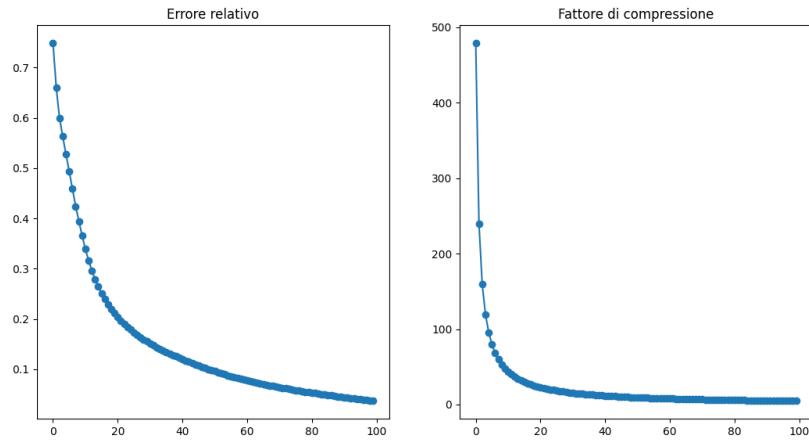


Figura 8: Errore relativo e fattore di compressione al variare di p

- Calcolare la soluzione del problema risolvendo le equazioni normali mediante la fattorizzazione LU e Cholesky.

```

import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg
from scipy.linalg import lu_factor as LUdec

m = 100
n = 10

A = np.random.rand(m, n)

alpha_test = np.full(n, 0.5) # esempio con alpha = 0.5
y = A @ alpha_test # y = A*alpha

print('alpha test', alpha_test) # [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]

ATA = A.T@A
ATy = A.T@y

lu, piv = LUdec(ATA)
alpha_LU = scipy.linalg.lu_solve((lu,piv), ATy)

print('alpha LU', alpha_LU) # [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]

L = scipy.linalg.cholesky(ATA)
x = scipy.linalg.solve_triangular(np.transpose(L), ATy, lower=True)
alpha_chol = scipy.linalg.solve_triangular(L, x, lower=False)

print('alpha chol', alpha_chol) # [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]

```

- Calcolare la soluzione del problema usando la SVD della matrice A .

```
U, s, Vh = scipy.linalg.svd(A)
```

```

alpha_svd = np.zeros(s.shape)

for i in range(n):
    ui = U[:, i]
    vi = Vh[i, :]

    alpha_svd = alpha_svd + (ui.T@y/s[i])*vi

print('alpha SVD', alpha_svd) # [0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]

```

- Calcolare l'errore relativo delle soluzioni trovate, rispetto al vettore α , soluzione esatta utilizzata per generare il problema test.

```

err_rel_LU = np.linalg.norm(alpha_LU - alpha_test) / np.linalg.norm(alpha_test)
err_rel_chol = np.linalg.norm(alpha_chol - alpha_test) / np.linalg.norm(alpha_test)
err_rel_svd = np.linalg.norm(alpha_svd - alpha_test) / np.linalg.norm(alpha_test)

print('Errore relativo LU', err_rel_LU) # 6.7143255907214105e-15
print('Errore relativo chol', err_rel_chol) # 7.27852596108914e-15
print('Errore relativo SVD', err_rel_svd) # 1.5570853182758493e-15

```

Esercizio 1.2

Date le seguenti funzioni:

- $f(x) = \exp\left(\frac{x}{2}\right)$ per $x \in [-1, 1]$
- $f(x) = \frac{1}{1+25x^2}$ per $x \in [-1, 1]$
- $f(x) = \sin(x) + \cos(x)$ per $x \in [0, 2\pi]$

Per ciascuna delle funzioni date, eseguire le seguenti richieste:

- Calcolare $m = 10$ coppie di punti $(x_i, f(x_i))$.

```

import numpy as np
import matplotlib.pyplot as plt
import scipy.linalg
from scipy.linalg import lu_factor as LUdec

case = 0 # [0, 1, 2]
m = 10
m_plot = 100

# Grado polinomio approssimante
n = 5

if case==0:
    x = np.linspace(-1,1,m)
    y = np.exp(x/2)
elif case==1:
    x = np.linspace(-1,1,m)
    y = 1/(1+25*(x**2))
elif case==2:
    x = np.linspace(0,2*np.pi,m)
    y = np.sin(x)+np.cos(x)

```

- Per n fissato, calcolare una soluzione del problema dei minimi quadrati utilizzando un metodo a scelta tra quelli utilizzati nell'esercizio precedente.

```
A = np.zeros((m, n+1))

for i in range(n+1):
    A[:, i] = x**i

U, s, Vh = scipy.linalg.svd(A)

alpha_svd = np.zeros(n+1)

for i in range(n+1):
    ui = U[:, i]
    vi = Vh[i, :]

    alpha_svd = alpha_svd + (ui.T@y/s[i])*vi

print('alpha_svd', alpha_svd)
# [ 0.99315181  1.18619381 -0.77082962 -0.07787167  0.06176099 -0.00551072]
```

- Per ciascun valore di $n \in \{1, 2, 3, 5, 7\}$, creare una figura con il grafico della funzione esatta $f(x)$ insieme a quello del polinomio di approssimazione $p(x)$, evidenziando i m punti noti.

```
# for n in [1,2,3,5,7]:
x_plot = np.linspace(x[0], x[-1], m_plot)
A_plot = np.zeros((m_plot, n+1))

for i in range(n+1):
    A_plot[:, i] = x_plot**i

y_interpolation = A_plot@alpha_svd

plt.plot(x, y, 'o')
plt.plot(x_plot, y_interpolation, 'r')
plt.xlabel('x')
plt.ylabel('y')
plt.title(f'Interpolazione polinomiale di grado {n}')
plt.grid()
plt.show()
```

- Per ciascun valore di $n \in \{1, 2, 3, 5, 7\}$, calcolare e stampare il valore del residuo in norma 2 commesso nei punti x_i .

```
res = np.linalg.norm(y - A@alpha_svd)
print(f'Residual n={n}: ', res)

Residual n=1:  2.8367505921049365
Residual n=2:  1.7952974022498318
Residual n=3:  0.8007707717731807
Residual n=5:  0.06559431617548347
Residual n=7:  0.0016911098620945113
```

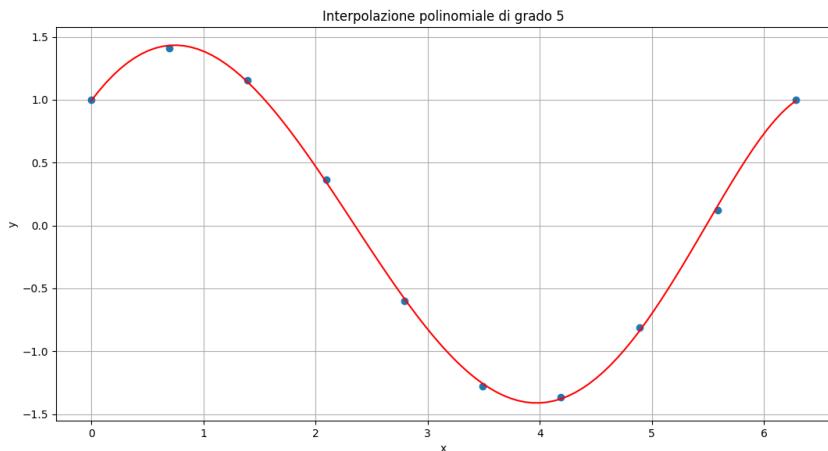


Figura 9: Interpolazione con $n = 5$ di $\sin(x) + \cos(x)$

Esercitazione 5: Metodi iterativi per radici di funzioni

Esercizio 1.1

Scrivere una funzione che implementi il metodo delle approssimazioni successive per il calcolo dello zero di una funzione $f(x)$, prendendo come input una delle seguenti funzioni per l'aggiornamento:

- $g(x) = x - f(x)e^{x/2}$
- $g(x) = x - f(x)e^{-\frac{x}{2}}$

Testare la funzione per trovare lo zero della funzione $f(x) = e^x - x^2$, la cui soluzione è $x^* = -0.703467$.

Scrivere una funzione che implementi il metodo di Newton, ricordando che il metodo di Newton può essere considerato come un caso particolare del metodo delle approssimazioni successive dove la funzione di aggiornamento è $g(x) = x - \frac{f(x)}{f'(x)}$.

- Disegnare il grafico della funzione f nell'intervallo $I = [-1, 1]$ e verificare che x^* sia lo zero di f in $[-1, 1]$.

```
import numpy as np
import matplotlib.pyplot as plt

f = lambda x: np.exp(x)-x**2

xTrue = -0.703467
fTrue = f(xTrue)
print('fTrue = ', fTrue) # 8.035078391532835e-07

xplot = np.linspace(-1, 1)
fplot = f(xplot)

plt.plot(xplot,fplot)
plt.plot(xTrue,fTrue, 'or', label='x*')

plt.legend()
plt.grid()
plt.show()
```

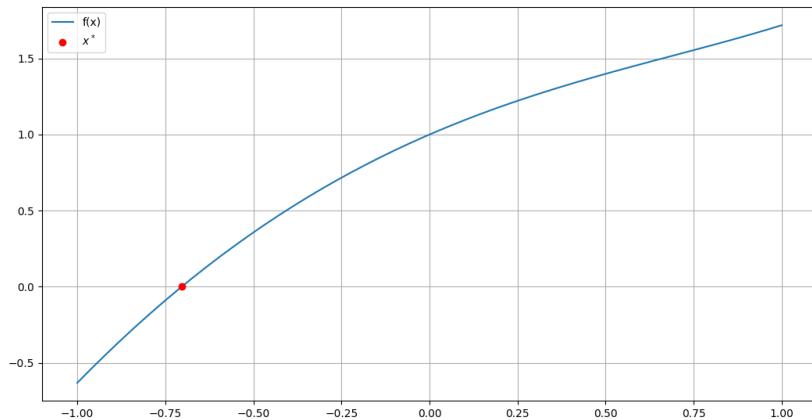


Figura 10: Grafico di $f(x) = e^x - x^2$ con $x^* = -0.703467$

- Calcolare lo zero della funzione utilizzando i metodi precedentemente descritti.

```

def succ_app(f, g, tol_f, tol_x, max_it, x_true, x0=0):
    i=0
    err=np.zeros(max_it+1, dtype=np.float64)
    err[0]=tol_x+1
    vec_errore=np.zeros(max_it+1, dtype=np.float64)
    vec_errore[0] = np.abs(x0-x_true)
    x=x0

    while (err[i]>tol_x and i<max_it): # scarto assoluto tra iterazioni
        x_new=g(x)
        err[i+1]=np.abs(x_new-x)
        vec_errore[i+1]=np.abs(x_new-x_true)
        i=i+1
        x=x_new
    err=err[0:i]
    vec_errore = vec_errore[0:i]
    return (x, i, err, vec_errore)

def newton(f, df, tol_f, tol_x, max_it, x_true, x0=0):
    g = lambda x: x-f(x)/df(x)
    (x, i, err, vec_errore) = succ_app(f, g, tol_f, tol_x, max_it, x_true, x0)
    return (x, i, err, vec_errore)

df = lambda x: np.exp(x)-2*x
g1 = lambda x: x-f(x)*np.exp(x/2)
g2 = lambda x: x-f(x)*np.exp(-x/2)

```

- Confrontare l'accuratezza delle soluzioni trovate e il numero di iterazioni effettuate dai solutori.

```

tol_x= 10**(-10)
tol_f = 10**(-6)
max_it=100
x0= 0

```

```

[sol_g1, iter_g1, err_g1, vecErrore_g1]=succ_app(f, g1, tol_f, tol_x, max_it, x_true,
→ x0)
print('Metodo approssimazioni successive g1 \n x =',sol_g1,'\\n iter_new= ', iter_g1)

plt.plot(sol_g1,f(sol_g1), 'o', label='g1') # converge in 23 iter

[sol_g2, iter_g2, err_g2, vecErrore_g2]=succ_app(f, g2, tol_f, tol_x, max_it, x_true,
→ x0)
print('Metodo approssimazioni successive g2 \n x =',sol_g2,'\\n iter_new= ', iter_g2)

plt.plot(sol_g2,f(sol_g2), 'og', label='g2') # non converge dopo 100 iterate

[sol_newton, iter_newton, err_newton, vecErrore_newton]=newton(f, df, tol_f, tol_x,
→ max_it, x_true, x0)
print('Metodo Newton \n x =',sol_newton,'\\n iter_new= ', iter_newton) # converge 6 it

plt.plot(sol_newton,f(sol_newton), 'ob', label='Newton')
plt.legend()
plt.grid()
plt.show()

Metodo approssimazioni successive g1
x = -0.7034674225096886
iter_new= 23
Metodo approssimazioni successive g2
x = -0.48775858993453886
iter_new= 100
Metodo Newton
x = -0.7034674224983917
iter_new= 6

```

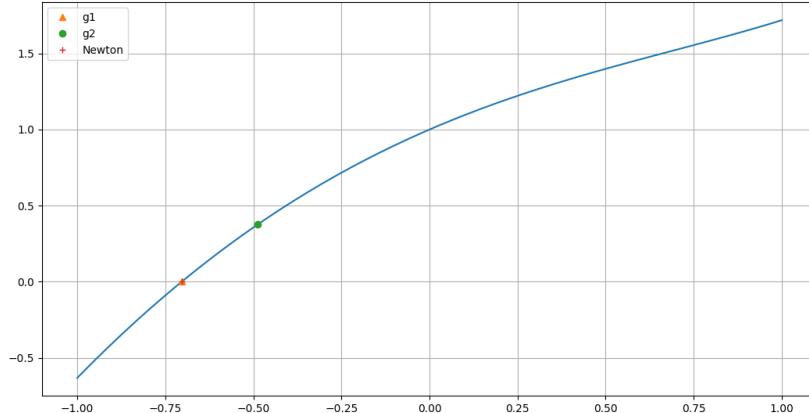


Figura 11: Solo g_1 e Newton convergono alla soluzione esatta

- Modificare le due funzioni in modo da calcolare l'errore $\|x_k - x^*\|_2$ ad ogni iterazione k -esima e graficare.

```

# g1
plt.plot(vecErrore_g1, '.-', color='blue')

```

```

# g2
plt.plot(vecErrore_g2[:20], '.-', color='green')
# Newton
plt.plot(vecErrore_newton, '.-', color='red')

plt.legend( ("g1", "g2", "newton"))
plt.xlabel('iter')
plt.ylabel('errore')
plt.title('Errore vs Iterazioni')
plt.grid()
plt.show()

```

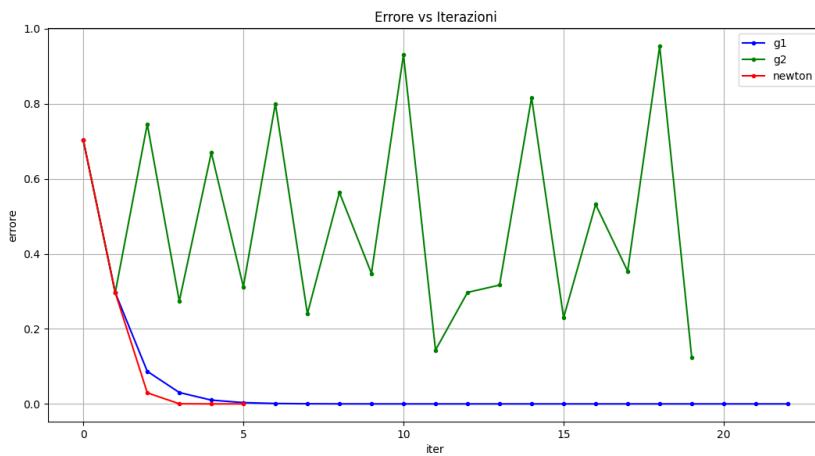


Figura 12: Ovviamente g_2 , in figura limitato a 20 iterazioni, non converge

Esercizio 1.2

Applicare il metodo delle approssimazioni successive e il metodo di Newton a:

- Per la funzione $f(x) = x^3 + 4x \cos(x) - 2$ nell'intervallo $[0, 2]$ con $g(x) = \frac{2-x^3}{4\cos(x)}$, con $x^* \approx 0.5369$

```

f = lambda x: x**3+4*x*np.cos(x)-2
df = lambda x: 3*x**2+4*np.cos(x)-4*x*np.sin(x)
g = lambda x: (2-x**3)/(4*np.cos(x))

```

```

xplot = np.linspace(0, 2)
fplot = f(xplot)

```

```

Metodo approssimazioni successive g
x = 0.5368385515641152
iter_new= 10
Metodo Newton
x = 0.5368385515667755
iter_new= 6

```

- Per la funzione $f(x) = x - x^{\frac{1}{3}} - 2$ nell'intervallo $[3, 5]$ con $g(x) = x^{\frac{1}{3}} + 2$, con $x^* \approx 3.5213$

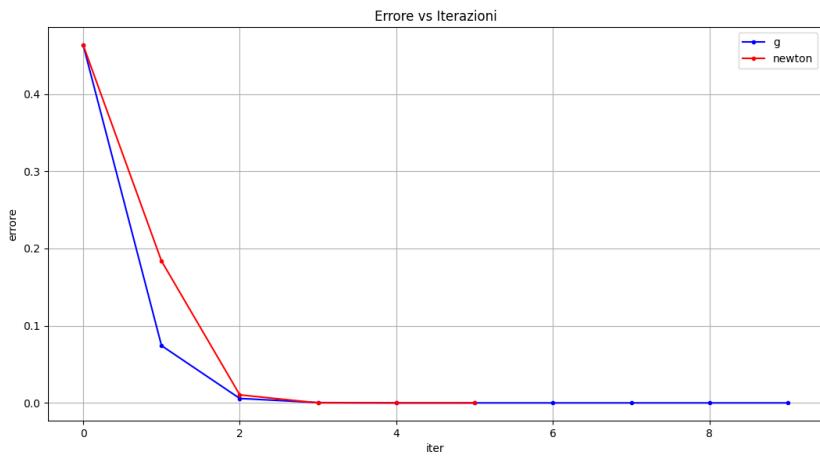


Figura 13: Sia Newton che $g(x) = \frac{2-x^3}{4 \cos(x)}$ convergono

```
f = lambda x: x-x**(-1/3)-2
df = lambda x: 1-1/(3*x**(-2/3))
g = lambda x: x**(-1/3)+2

xplot = np.linspace(3, 5)
fplot = f(xplot)
```

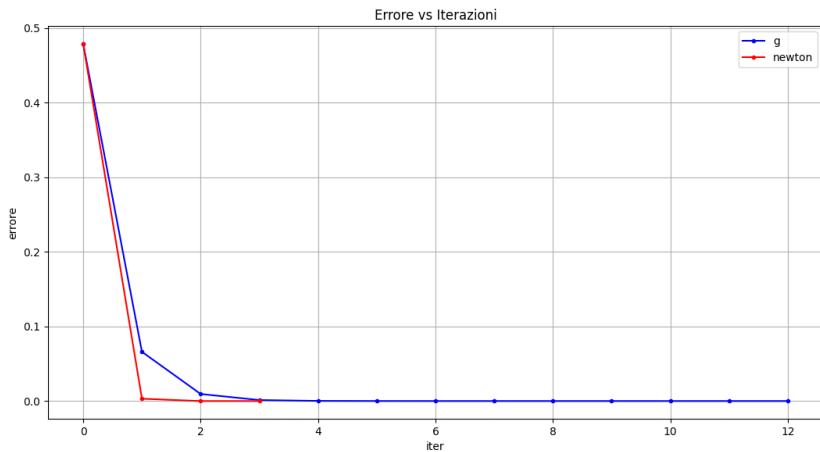


Figura 14: Newton converge più rapidamente di $g(x) = \frac{2-x^3}{4 \cos(x)}$

```
Metodo approssimazioni successive g
x = 3.521379706798214
iter_new= 13
Metodo Newton
x = 3.521379706804568
iter_new= 4
```

Esercitazione 6: Minimizzazione e metodi di discesa del gradiente

Esercizio 1.1

Si consideri $f : \mathbb{R}^n \rightarrow \mathbb{R}$ differenziabile. Scrivere una funzione Python che implementi il metodo di discesa del gradiente per risolvere il problema di minimo:

$$\arg \min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x})$$

Utilizzare una flag per scegliere se utilizzare:

- step size $\alpha > 0$ costante, passato in input;
- step size α_k variabile, calcolato secondo la procedura di backtracking ad ogni iterazione k -esima.

```
def next_step(x,grad): # backtracking procedure for the choice of the steplength
    alpha=1.1
    rho = 0.5
    c1 = 0.25
    p=-grad
    j=0
    jmax=10
    while ((f(x+alpha*p) > f(x)+c1*alpha*np.dot(grad,p)) and j<jmax):
        alpha= rho*alpha
        j+=1
    if (j>jmax):
        return -1
    else:
        print('alpha=',alpha)
        return alpha

def minimize(f,grad_f,x0,step,maxit,tol,xTrue,fixed=True): # funzione che implementa il
→ metodo del gradiente
    #declare x_k and gradient_k vectors
    # x_list only for logging
    x_list=np.zeros((2,maxit+1))

    norm_grad_list=np.zeros(maxit+1)
    function_eval_list=np.zeros(maxit+1)
    error_list=np.zeros(maxit+1)

    #initialize first values
    x_last = x0

    x_list[:,0] = x_last

    k=0

    function_eval_list[k]=f(x_last)
    error_list[k]=np.linalg.norm(x_last-xTrue)
    norm_grad_list[k]=np.linalg.norm(grad_f(x_last))

    while (np.linalg.norm(grad_f(x_last))>tol and k < maxit):
        k=k+1
        grad = grad_f(x_last)#direction is given by gradient of the last iteration
```

```

if fixed:
    # Fixed step
    step = step
else:
    # backtracking step
    step = next_step(x_last,grad)

if(step== -1):
    print('non convergente')
    return (k) #no convergence

x_last=x_last-step*grad

x_list[:,k] = x_last

function_eval_list[k]=f(x_last)
error_list[k]=np.linalg.norm(x_last-xTrue)
norm_grad_list[k]=np.linalg.norm(grad_f(x_last))

function_eval_list = function_eval_list[:k+1]
error_list = error_list[:k+1]
norm_grad_list = norm_grad_list[:k+1]

print('iterations=',k)
print('last guess: x=(%f,%f)'%(x_list[0,k],x_list[1,k]))

return (x_last,norm_grad_list, function_eval_list, error_list, x_list, k)

```

Esercizio 1.2

Si consideri la seguente funzione

$$f(x, y) = 3(x - 2)^2 + (y - 1)^2$$

che ha un minimo globale in $(2, 1)$ dove $f(2, 1) = 0$.

- Plottare la superficie $f(x, y)$ con il comando `plot_surface` nel dominio $[-1.5, 3.5] \times [-1, 5]$.

```

def f(vec):
    x, y = vec
    fout = 3*(x-2)**2 + (y-1)**2
    return fout

def grad_f(vec):
    x, y = vec
    dfdx = 6*(x-2)
    dfdy = 2*(y-1)
    return np.array([dfdx,dfdy])

x = np.linspace(-1.5,3.5)
y = np.linspace(-1,5)

X, Y = np.meshgrid(x, y)
vec = np.array([X,Y])

```

```

Z=f(vec)

fig = plt.figure(figsize=(15, 8))

ax = plt.axes(projection='3d')
ax.set_title('f(x)=(x-1)^2 + (y-2)^2')
#ax.view_init(elev=50., azim=30)
s = ax.plot_surface(X, Y, Z, cmap='viridis')
fig.colorbar(s)
plt.show()

```

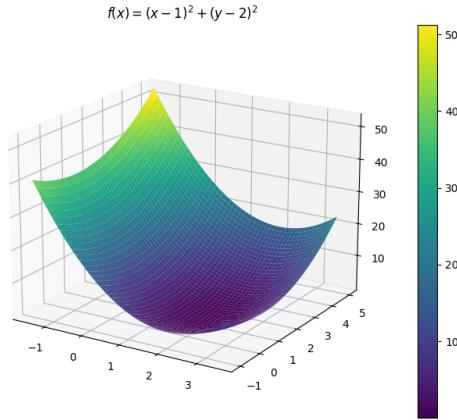


Figura 15: Superficie di $f(x, y)$

- Plottare le curve di livello di $f(x, y)$ con il comando `contour` nello stesso dominio.

```

fig = plt.figure(figsize=(8, 5))
contours = plt.contour(X, Y, Z, levels=1000)
plt.title('Contour plot $f(x)=(1-x)^2+100*(y-x^2)^2$')
fig.colorbar(contours)
plt.show()

```

- Determinare il punto di minimo di $f(x, y)$ utilizzando la funzione precedentemente scritta (sia con passo fisso che con passo variabile) usando come punto iniziale $(3, 5)$.

```

step = 0.001
maxitS=1000
tol=1.e-5
x0 = np.array([-0.5,1])
xTrue = np.array([1,1])
(x_last,norm_grad_listf, function_eval_listf, error_listf, xlist, k)= minimize(f,grad_f,x0,step,maxitS,tol)
plt.plot(function_eval_listf,'*-')
(x_last,norm_grad_list, function_eval_list, error_list, xlist, k)= minimize(f,grad_f,x0,step,maxitS,tol)
plt.plot(function_eval_list,'*-')
plt.legend(['fixed', 'backtracking'])
plt.show()

last guess: x=(2.000001,1.000003)

```

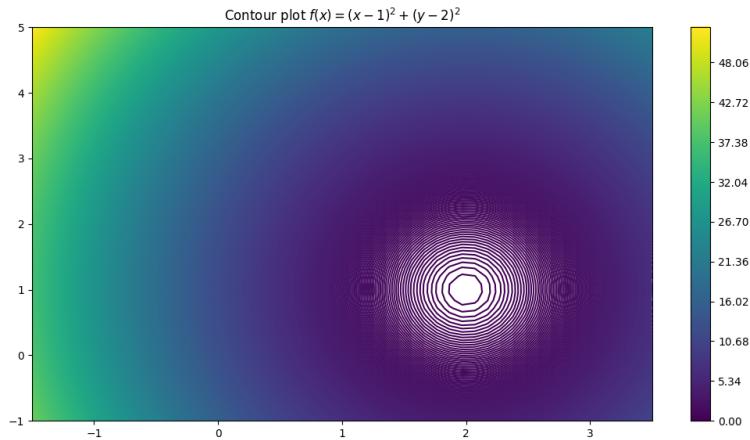


Figura 16: Curve di livello di $f(x, y)$ a livello 1000

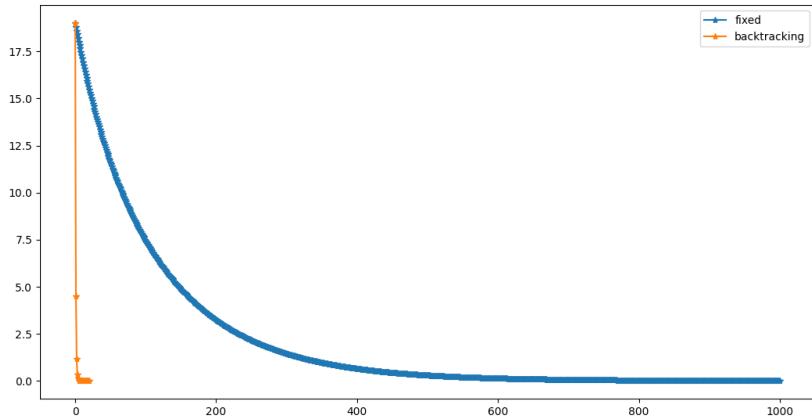


Figura 17: Discesa del gradiente con α fisso e α_k da backtracking

- Si analizzino i risultati in entrambi i casi.

Notiamo come, determinando la lunghezza del passo α_k con l'utilizzo dell'algoritmo di backtracking, la discesa del gradiente termina con successo quasi immediatamente, dopo appena 20 iterazioni. Al contrario, utilizzando α fisso, la discesa non converge al minimo entro la soglia limite di iterazioni.

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
ax1.semilogy(norm_grad_listf)
ax1.semilogy(norm_grad_list)
ax1.set_title('$\|\nabla f(x_k)\|$')
ax2.semilogy(function_eval_listf)
ax2.semilogy(function_eval_list)
ax2.set_title('$f(x_k)$')
ax3.semilogy(error_listf)
ax3.semilogy(error_list)
ax3.set_title('$\|x_k - x^*\|$')
fig.tight_layout()
```

```
fig.legend(['fixed', 'backtracking'], loc='lower center', ncol=4)
plt.show()
```

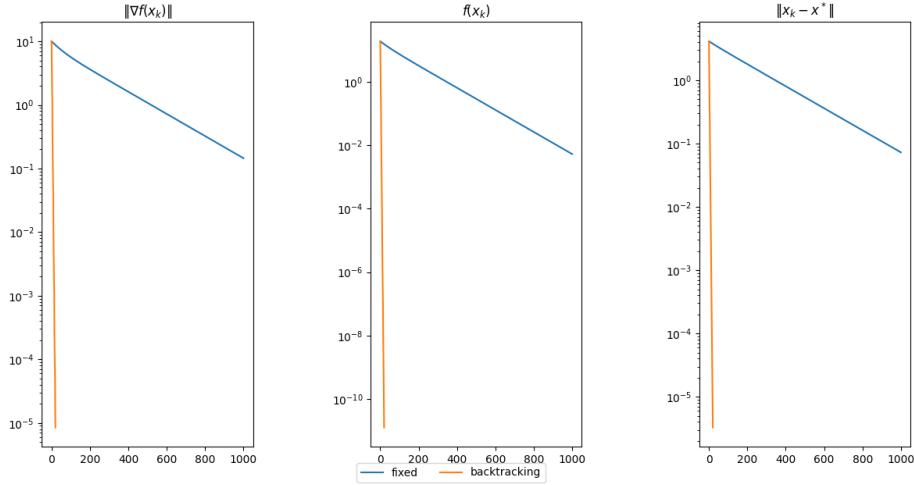


Figura 18: Confronto tra lunghezza del passo α fissa e α_k da backtracking

Esercizio 1.3

Si consideri la seguente funzione detta funzione di Rosenbrock:

$$f(x, y) = 100(y - x^2)^2 + (1 - x)^2$$

che ha un minimo globale in $(1, 1)$ dove $f(1, 1) = 0$. Si eseguano le richieste dell'esercizio precedente nel dominio $[-2, 2] \times [-1, 3]$, usando come punto iniziale $(-0.5, 1)$.

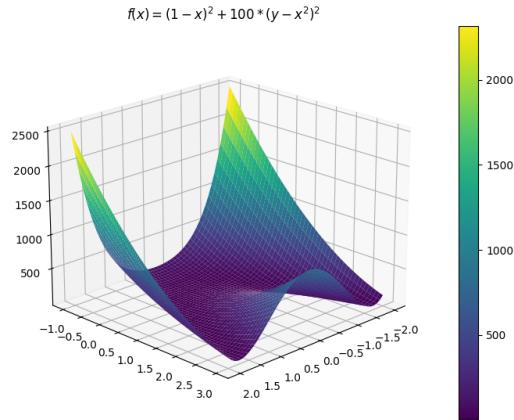


Figura 19: Superficie di $f(x) = (1 - x)^2 + 100 * (y - x^2)^2$

In questo caso, sia la scelta di mantenere fissa la lunghezza del passo α che l'impiego dell'algoritmo di backtracking portano alla terminazione della discesa del gradiente non a causa del raggiungimento di una soluzione convergente, bensì perché si è giunti al limite massimo di iterazioni prestabilito.

- Dopo 1000 iterazioni, con α fisso si ottiene:

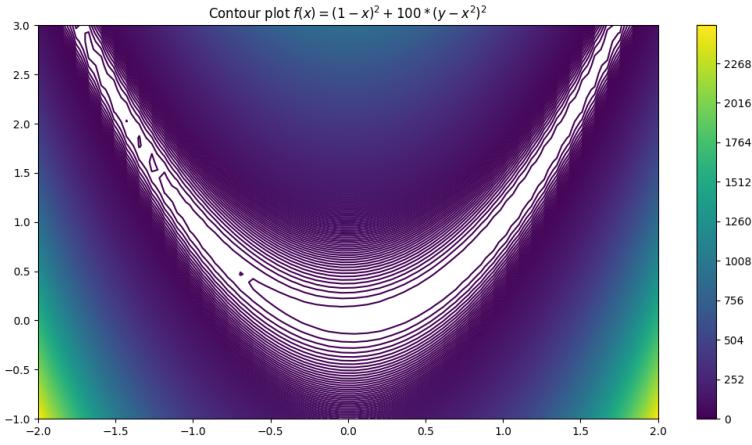


Figura 20: Curve di livello di $f(x, y)$ a livello 1000

```
last guess: x=(0.482604,0.230313)
```

in tal punto, la funzione vale 0.26837130770971884.

- Mentre con l'algoritmo di backtracking si ottiene:

```
last guess: x=(0.992916,0.985849)
```

in tal punto, la funzione vale 0.00005029316752055019.

Si osserva che la versione con backtracking sembra convergere a un valore molto inferiore rispetto alla versione con α costante, che mostra un risultato meno soddisfacente.

Esercitazione 7: Deblurring di immagini

Il problema di deblur consiste nella ricostruzione di un'immagine a partire da un dato acquisito mediante il seguente modello:

$$y = Ax + \eta$$

dove:

- y rappresenta l'immagine corrotta,
- x rappresenta l'immagine originale che vogliamo ricostruire,
- A rappresenta l'operatore che applica il blur Gaussiano,
- $\eta \sim \mathcal{N}(0, \sigma^2)$ rappresenta una realizzazione di rumore additivo con distribuzione Gaussiana di media $\mu = 0$ e deviazione standard σ .

Esercizio 1.1 Problema test

- Caricare l'immagine `camera` dal modulo `skimage.data` rinormalizzandola nel range [0, 1].

```

import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt
from skimage import data, metrics
from scipy import signal
from numpy import fft
from utils import psf_fft, A, AT, gaussian_kernel

# Immagine in floating point con valori tra 0 e 1
X = data.camera() / 255
m, n = X.shape

```

- Applicare un blur di tipo gaussiano con deviazione standard 3 il cui kernel ha dimensioni 24×24 utilizzando la funzione. Utilizzare prima cv2 (open-cv) e poi la trasformata di Fourier.

```

# Genera il filtro di blur
k = gaussian_kernel(24, 3)
plt.imshow(k)
plt.show()

# Blur with openCV
X_blurred = cv.filter2D(X, -1, k)
plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122).imshow(X_blurred, cmap='gray', vmin=0, vmax=1)
plt.title('Blurred')
plt.xticks([]), plt.yticks([])
plt.show()

# Blur with FFT
K = psf_fft(k, 24, X.shape)
plt.imshow(np.abs(K))
plt.show()

```

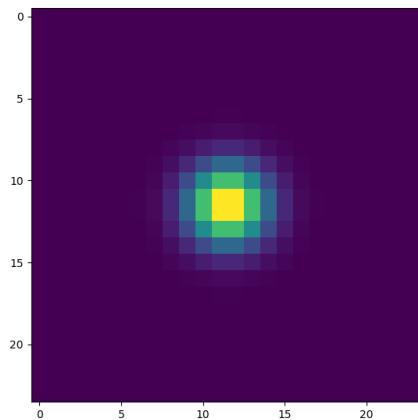


Figura 21: Kernel Gaussiano di dimensione 24×24 con deviazione standard 3

- Aggiungere rumore di tipo gaussiano, con $\sigma = 0.02$, usando la funzione `np.random.normal()`.



Figura 22: Blur con cv2

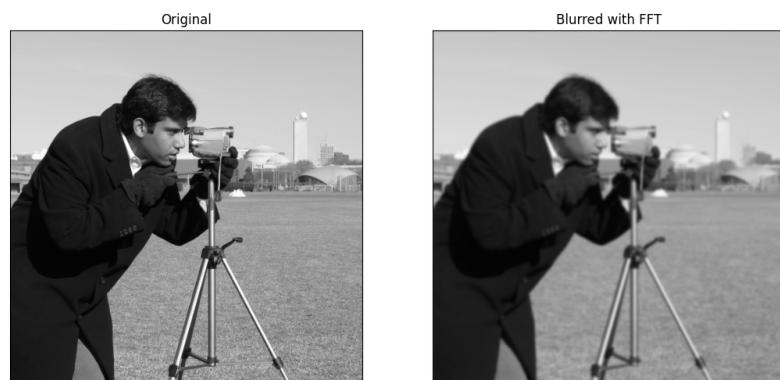


Figura 23: Blur con trasformata di Fourier (FFT)

```
# Genera il rumore
sigma = 0.02
np.random.seed(42)
noise = np.random.normal(size=X.shape) * sigma
```

- Calcolare le metriche Peak Signal Noise Ratio (PSNR) e Mean Squared Error (MSE) tra l'immagine degradata e l'immagine esatta usando le funzioni `peak_signal_noise_ratio` e `mean_squared_error` disponibili nel modulo `skimage.metrics`.

```
# Aggiungi blur e rumore
y = X + noise
PSNR = metrics.peak_signal_noise_ratio(X, y)
ATy = AT(y, K)

# Visualizziamo i risultati
plt.figure(figsize=(30, 10))
```

```

plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
plt.title('Original')
plt.xticks([]), plt.yticks([])
plt.subplot(122).imshow(y, cmap='gray', vmin=0, vmax=1)
plt.title(f'Corrupted (PSNR: {PSNR:.2f})')
plt.xticks([]), plt.yticks([])
plt.show()

```

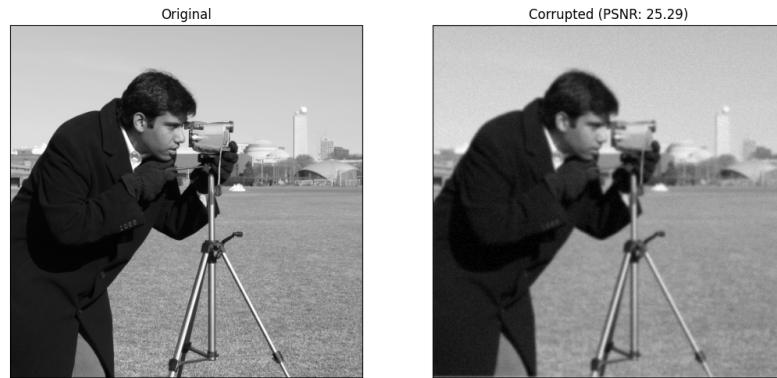


Figura 24: Confronto tra immagine originale e degradata dal rumore

Esercizio 1.2 Soluzione naïve

Soluzione naïve Una possibile ricostruzione dell’immagine originale x partendo dall’immagine corrotta y è la soluzione naïve data dal minimo del seguente problema di ottimizzazione:

$$x^* = \arg \min_x \frac{1}{2} \|Ax - y\|_2^2$$

- Utilizzando il metodo del gradiente coniugato implementato dalla funzione `minimize` della libreria `scipy`, calcolare la soluzione naïve.

```

# Soluzione naïve
from scipy.optimize import minimize

# Funzione da minimizzare
def f(x):
    x = x.reshape((m, n))
    Ax = A(x, K)
    return 0.5 * np.sum(np.square(Ax - y))

# Gradiente della funzione da minimizzare
def df(x):
    x = x.reshape((m, n))
    ATAx = AT(A(x,K),K)
    d = ATAx - ATy
    return d.reshape(m * n)

# Minimizzazione della funzione

```

```

x0 = y.reshape(m*n)
max_iter = 25
res = minimize(f, x0, method='CG', jac=df, options={'maxiter':max_iter, 'return_all':True})

• Analizza l'andamento del PSNR e dell'MSE al variare del numero di iterazioni.

# Per ogni iterazione calcola il PSNR rispetto all'originale
PSNR = np.zeros(max_iter + 1)
for k, x_k in enumerate(res.allvecs):
    PSNR[k] = metrics.peak_signal_noise_ratio(X, x_k.reshape(X.shape))

# Risultato della minimizzazione
X_res = res.x.reshape((m, n))

# PSNR dell'immagine corrotta rispetto all'oginale
starting_PSNR = np.full(PSNR.shape[0], metrics.peak_signal_noise_ratio(X, y))

# Visualizziamo i risultati
ax2 = plt.subplot(1, 2, 1)
ax2.plot(PSNR, label="Soluzione naive")
ax2.plot(starting_PSNR, label="Immagine corrotta")
plt.legend()
plt.title('PSNR per iterazione')
plt.ylabel("PSNR")
plt.xlabel('itr')
plt.subplot(1, 2, 2).imshow(X_res, cmap='gray', vmin=0, vmax=1)
plt.title('Immagine Ricostruita')
plt.xticks([]), plt.yticks([])
plt.show()

```

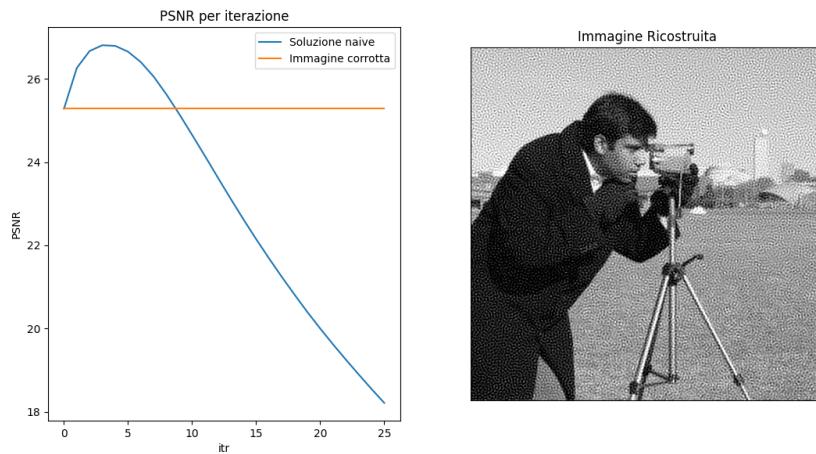


Figura 25: Notiamo come l'immagine presenti un effetto sgradevole dovuto alla mancanza di regolarizzazione

Esercizio 1.3 Soluzione regolarizzata

Si consideri il seguente problema regolarizzato secondo Tikhonov:

$$x^* = \arg \min_x \frac{1}{2} \|Ax - y\|_2^2 + \lambda \|x\|_2^2$$

- Utilizzando sia il metodo del gradiente che il metodo del gradiente coniugato, calcolare la soluzione del problema regolarizzato.

```
def f(x, L):
    nsq = np.sum(np.square(x))
    x = x.reshape(m, n)
    Ax = A(x, K)
    return 0.5 * np.sum(np.square(Ax - y)) + 0.5 * L * nsq

# Gradiente della funzione da minimizzare
def df(x, L):
    Lx = L * x
    x = x.reshape(m, n)
    ATAx = AT(A(x,K),K)
    d = ATAx - ATy
    return d.reshape(m * n) + Lx
```

- Analizzare l'andamento del PSNR (Peak Signal-to-Noise Ratio) e dell'MSE (Mean Squared Error) al variare del numero di iterazioni.

```
x0 = y.reshape(m*n)
lambdas = [0.01, 0.03, 0.04, 0.06]
PSNRs = []
images = []

# Ricostruzione per diversi valori del parametro di regolarizzazione
for i, L in enumerate(lambdas):
    # Esegui la minimizzazione con al massimo 50 iterazioni
    max_iter = 50
    res = minimize(f, x0, (L), method='CG', jac=df, options={'maxiter':max_iter})

    # Aggiungi la ricostruzione nella lista images
    X_curr = res.x.reshape(X.shape)
    images.append(X_curr)

    # Stampa il PSNR per il valore di lambda attuale
    PSNR = metrics.peak_signal_noise_ratio(X, X_curr)
    PSNRs.append(PSNR)
    print(f'PSNR: {PSNR:.2f} (\u03bb = {L:.2f})')

# Visualizziamo i risultati
plt.plot(lambdas, PSNRs)
plt.title('PSNR per $\lambda$')
plt.ylabel("PSNR")
plt.xlabel('$\lambda$')
plt.show()
```

- Facendo variare il parametro di regolarizzazione λ , analizzare come questo influenza le prestazioni del metodo analizzando le immagini.

```
plt.figure(figsize=(30, 10))

(nrows, ncols) = ((len(lambdas) + 2) // 3, (len(lambdas) + 2) // 2)

plt.subplot(nrows, ncols, 1).imshow(X, cmap='gray', vmin=0, vmax=1)
plt.title("Originale")
```

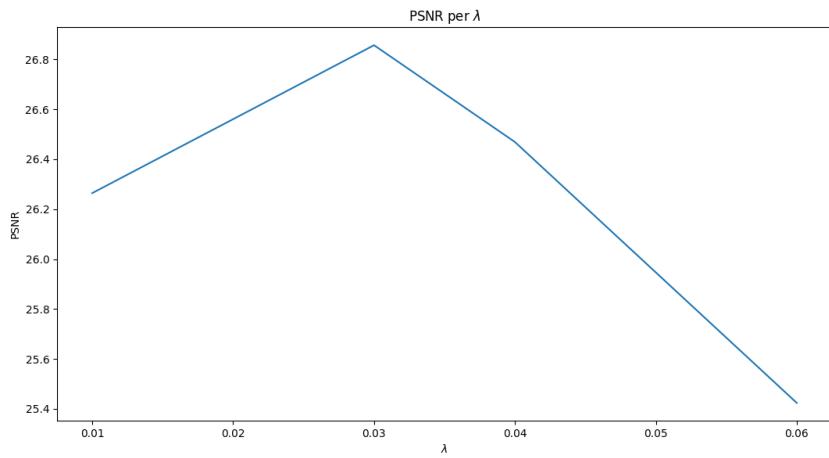


Figura 26: PSNR rispetto al parametro di regolarizzazione λ

```

plt.xticks([], plt.yticks([])
plt.subplot(nrows, ncols, 2).imshow(y, cmap='gray', vmin=0, vmax=1)
plt.title("Corrotta")
plt.xticks([], plt.yticks([])

for i, L in enumerate(lambdas):
    plt.subplot(nrows, ncols, i + 3).imshow(images[i], cmap='gray', vmin=0, vmax=1)
    plt.title(f"Ricostruzione ($\lambda$ = {L:.2f})")
plt.show()

```

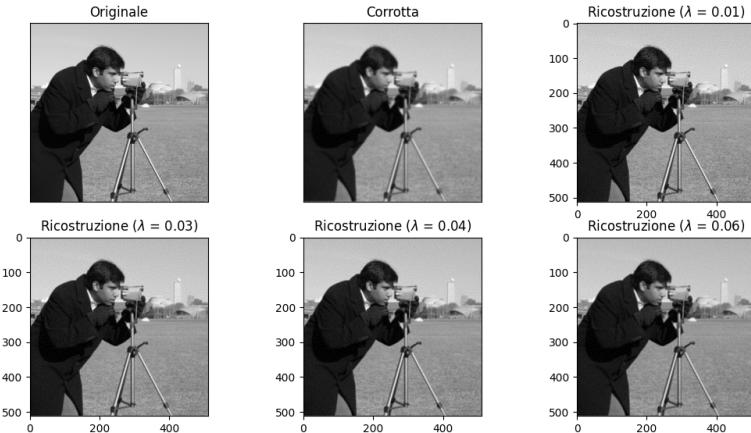


Figura 27: Ricostruzione dell'immagine al variare di λ

- Scegliere λ con il metodo di discrepanza.

```

# ||A_L - y|| <= tau * ||noise||
tau = 1.01

```

```

L = tau * np.linalg.norm(noise) / np.linalg.norm(ATy)
print('L', L) # 0.034985654993768586

max_iter = 50
res = minimize(f, x0, (L), method='CG', jac=df, options={'maxiter':max_iter})
X_curr = res.x.reshape(X.shape)

PSNR = metrics.peak_signal_noise_ratio(X, X_curr)
print('PSNR', PSNR) # 26.68424503233115

plt.subplot(121).imshow(X_blurred, cmap='gray', vmin=0, vmax=1)
plt.title('Corrotta')
plt.xticks([]), plt.yticks([])
plt.subplot(122).imshow(X_curr, cmap='gray', vmin=0, vmax=1)
plt.title(f'Ricostruita con $\lambda$ = {L:.4f} (PSNR: {PSNR:.4f})')
plt.xticks([]), plt.yticks([])
plt.show()

```



Figura 28: Con λ secondo il principio della discrepanza di Morozov si arriva a un buon PSNR

- Scegliere λ attraverso test sperimentali come il valore che massimizza il PSNR. Confrontare il valore ottenuto con quello della massima discrepanza.

Esercizio 1.4

- Ripetere i punti precedenti utilizzando anche l'operatore downsampling con i seguenti fattori di scaling $sf = 2, 4, 8, 16$.

```

S=2
X_d = X[::S,::S]
y_d = y[::S,::S]
K_d = K[::S,::S]
ATy_d = AT(y_d, K_d)

```

```

# Funzione da minimizzare
def f(x, L):
    nsq = np.sum(np.square(x))

```

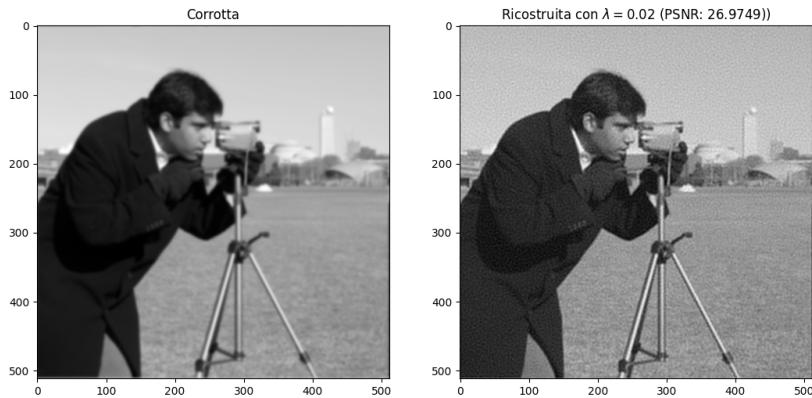


Figura 29: Con $\lambda = 0.02$ otteniamo ancora un miglior PSNR

```

x = x.reshape((m//S, n//S))
Ax = A(x, K_d)
return 0.5 * np.sum(np.square(Ax - y_d)) + 0.5 * L * nsq

# Gradiente della funzione da minimizzare
def df(x, L):
    Lx = L * x
    x = x.reshape(m//S, n//S)
    ATAx = AT(A(x,K_d),K_d)
    d = ATAx - ATy_d
    return d.reshape(m//S * n//S) + Lx

x0 = y_d.reshape(m//S*n//S)
lambdas = [0.03, 0.04, 0.06, 0.08]

```

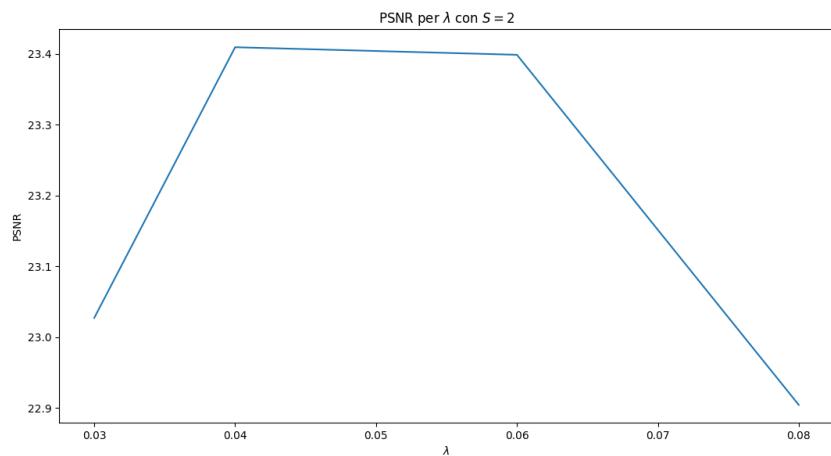


Figura 30: PSNR su λ con downsampling $S = 2$

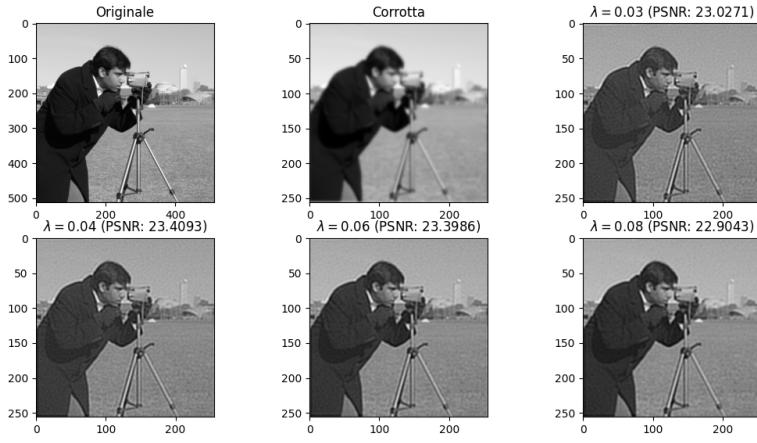


Figura 31: Ricostruzione al variare di λ con $S = 2$

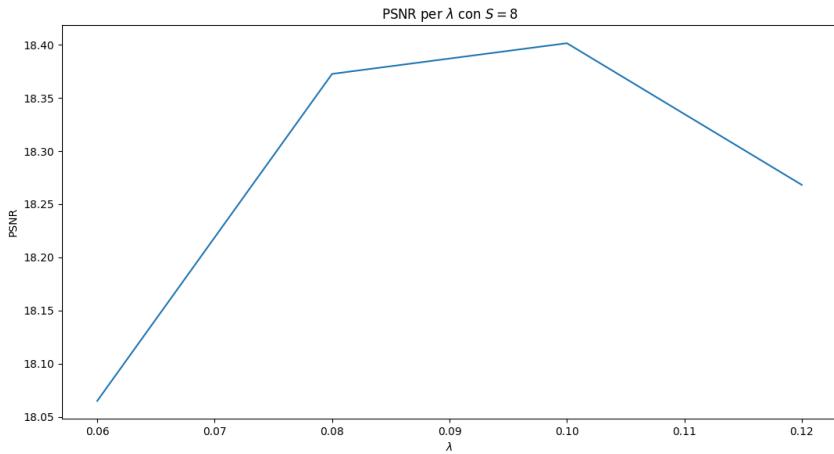


Figura 32: PSNR su λ con downsampling $S = 8$

- Testare i punti precedenti su due immagini in scala di grigio con caratteristiche differenti (per esempio, un’immagine tipo fotografico e una ottenuta con uno strumento differente, microscopio o altro).

```
from skimage.io import imread
import os
img = 'sand_microscope.png'
cur_dir = os.path.dirname(os.path.abspath(__file__))
X = imread(os.path.join(cur_dir, img), as_gray=True)
```

- Degradare le nuove immagini applicando, mediante le funzioni `gaussian_kernel()`, `psf_fft()`, l’operatore di blur con parametri:
 - $\sigma = 0.5$, dimensione del kernel 7×7 e 9×9 .
 - $\sigma = 1.3$, dimensione del kernel 5×5 .
- Aggiungere rumore gaussiano con deviazione standard nell’intervallo $(0, 0.05]$.

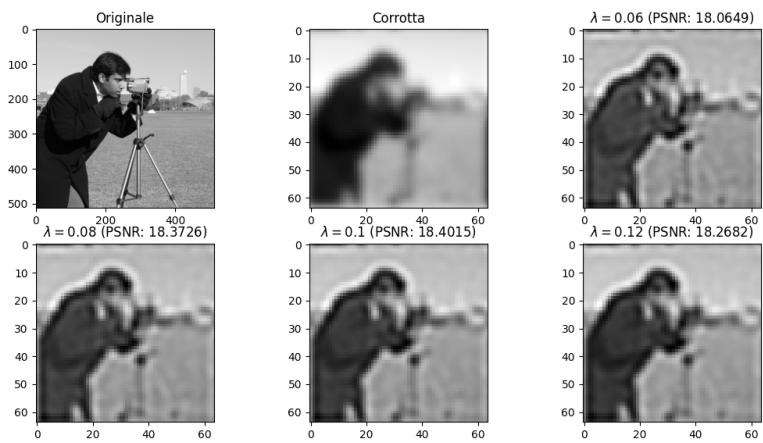


Figura 33: Ricostruzione al variare di λ con $S = 8$

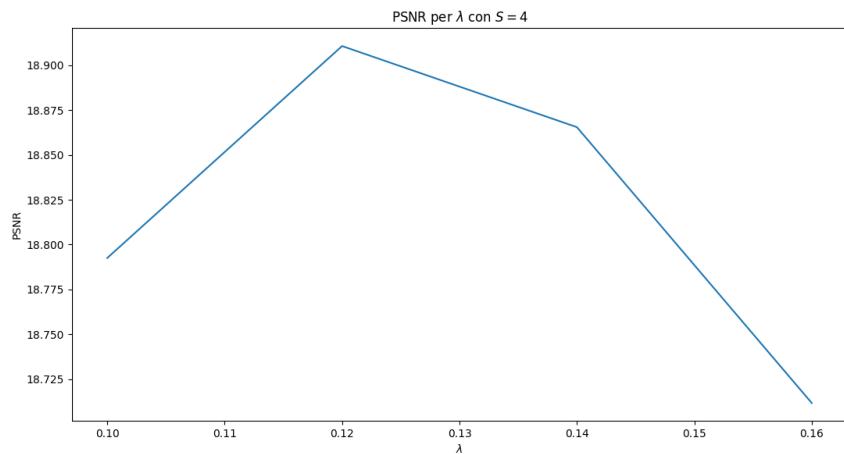


Figura 34: Sabbia su microscopio: PSNR su λ con downsampling $S = 4$

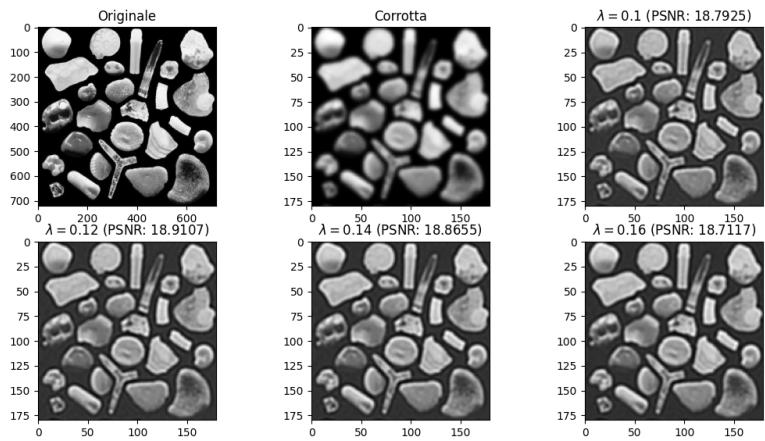


Figura 35: Ricostruzione al variare di λ con $S = 4$

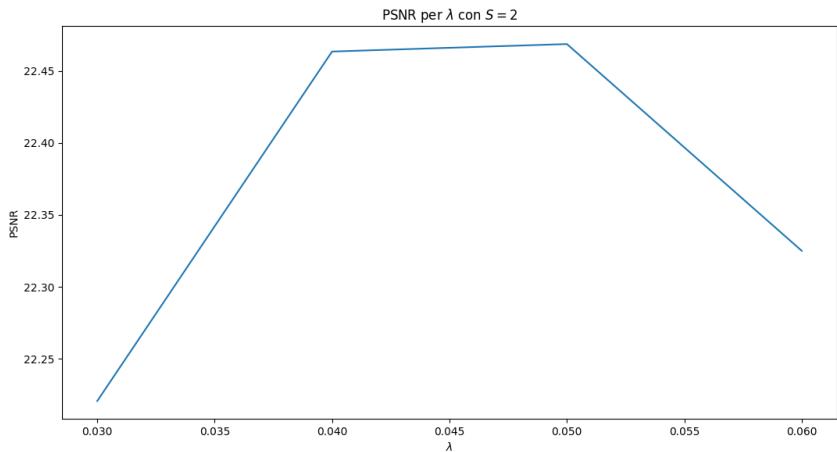


Figura 36: Modena: PSNR su λ con downsampling $S = 2$

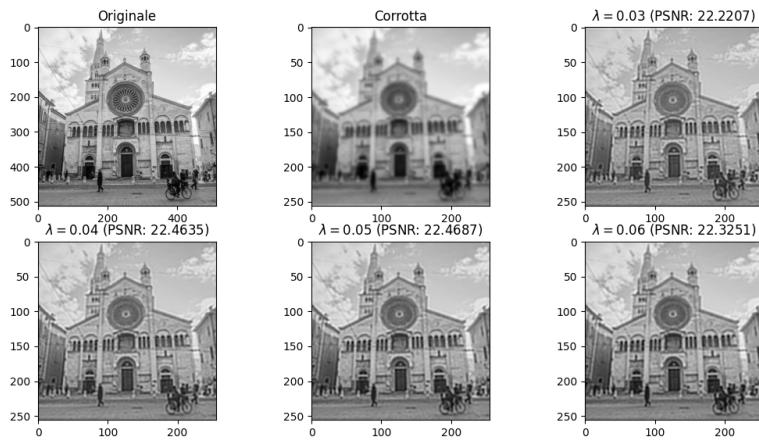


Figura 37: Ricostruzione al variare di λ con $S = 2$

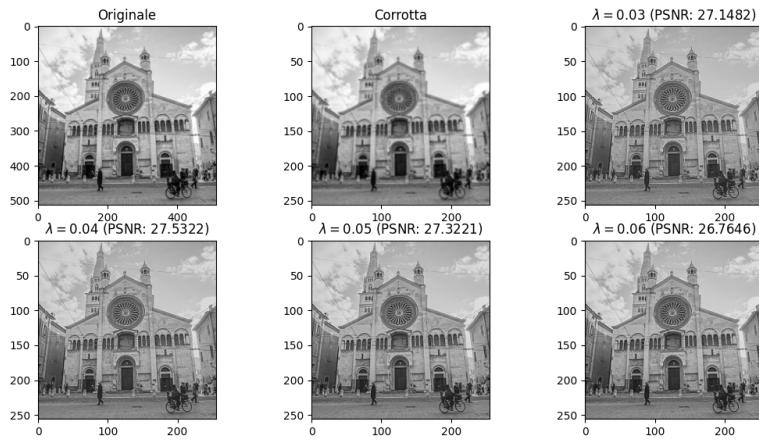


Figura 38: Modena con $\sigma = 0.5$, dimensione del kernel 7×7 , $S = 2$

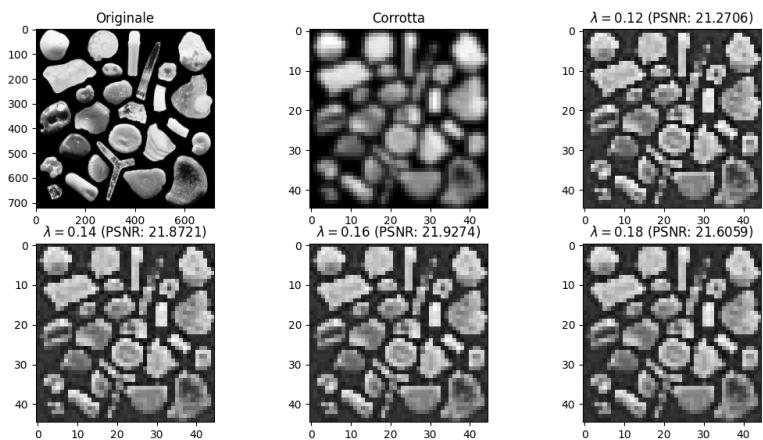


Figura 39: Sabbia con $\sigma = 0.5$, dimensione del kernel 9×9 , $S = 16$

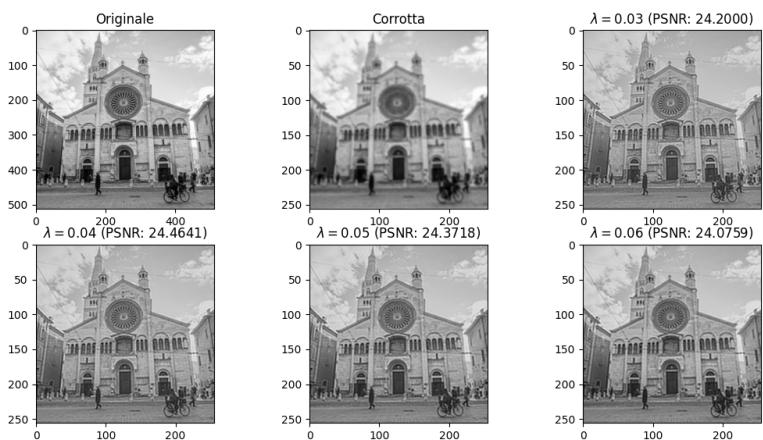


Figura 40: Modena con $\sigma = 1.3$, dimensione del kernel 5×5 , $S = 2$

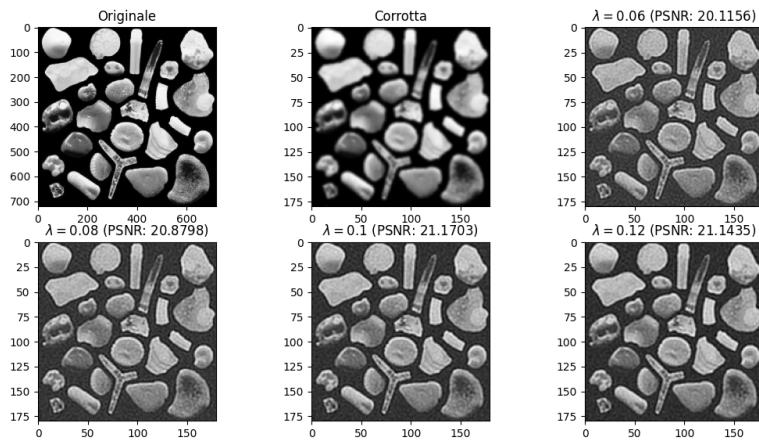


Figura 41: Sabbia con $\sigma = 1.3$, dimensione del kernel 5×5 , $S = 4$ e rumore gaussiano con deviazione standard nell’intervallo $(0, 0, 05]$

Domanda 1: Fattorizzazione LU con pivoting

Considero:

$$A\mathbf{x} = b$$

Con soluzione esatta $\bar{\mathbf{x}} = (1, \dots, 1)^T$.

Il numero di condizionamento di una matrice, il quale misura la sensibilità del sistema in base alla perturbazione sui dati in ingresso, è definito come il prodotto delle sue norme euclidee:

$$K(A) = \|A\| \cdot \|A^{-1}\|$$

dove:

$$\|A\|_2 = \sqrt{\rho(A^T A)}$$

Con ρ il raggio spettrale, ossia $\max_i(|\lambda_i|)$.

Possiamo risolvere il sistema tramite fattorizzazione LU con pivoting, il quale scomponere la matrice A in una matrice triangolare inferiore L , una triangolare superiore U e una matrice di permutazione P .

Di conseguenza risolviamo:

$$\begin{cases} Ly = Pb \\ Ux = y \end{cases}$$

Complessità computazionale: $O\left(\frac{2n^3}{3}\right)$

```
import numpy as np
n = np.random.randint(10, 1001)
random_matrix = np.random.rand(n, n)
```

Domanda 2: Fattorizzazione con Cholesky

Considero:

$$A = \begin{bmatrix} 9 & -4 & 0 & \cdots & 0 \\ -4 & 9 & -4 & \cdots & 0 \\ 0 & -4 & 9 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & -4 \\ 0 & 0 & \cdots & -4 & 9 \end{bmatrix}$$

Posso fattorizzare A con Cholesky ottenendo:

$$A = LL^T$$

Per $n = 3$:

$$K(A) = \|A\| \cdot \|A^{-1}\| = 4.384150540629856$$

$$L = \begin{bmatrix} 3 & 0 & 0 \\ -1.33333333 & 2.68741925 & 0 \\ 0 & -1.48841682 & 2.60472943 \end{bmatrix}$$

Possiamo risolvere il sistema tramite sostituzioni successive.

Domanda 3: Cholesky con matrice di Hilbert

Considero la matrice di Hilbert con $n = 5$:

$$A = \begin{bmatrix} 1 & 0.5 & 0.33333333 & 0.25 & 0.2 \\ 0.5 & 0.33333333 & 0.25 & 0.2 & 0.16666667 \\ 0.33333333 & 0.25 & 0.2 & 0.16666667 & 0.14285714 \\ 0.25 & 0.2 & 0.16666667 & 0.14285714 & 0.125 \\ 0.2 & 0.16666667 & 0.14285714 & 0.125 & 0.11111111 \end{bmatrix}$$

Osservo che la matrice A è notevolmente mal condizionata:

$$K(A) = 476607.2502425855$$

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0.5 & 0.28867513 & 0 & 0 & 0 \\ 0.33333333 & 0.28867513 & 0.0745356 & 0 & 0 \\ 0.25 & 0.25980762 & 0.1118034 & 0.01889822 & 0 \\ 0.2 & 0.23094011 & 0.12777531 & 0.03779645 & 0.0047619 \end{bmatrix}$$

Domanda 4: Compressione SVD

È possibile approssimare le immagini rappresentate come matrici scomponendole in una somma di **diadi**, ossia matrici di rango 1, mediante decomposizione in valori singolari (SVD).

$$A = U\Sigma V^T \Rightarrow A_p = \sum_{i=1}^p \sigma_i u_i v_i^T$$

Dove:

- σ_i rappresenta il i -esimo valore singolare nella matrice Σ . Ho che $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p > \sigma_{p+1} = \dots = \sigma_n = 0$.
- \mathbf{u}_i è la i -esima colonna di U .
- \mathbf{v}_i è la i -esima colonna di V .

Maggiore è $p < k$, con k rango di A , migliore sarà la riproduzione, ma maggiori saranno i dati utilizzati.

Domanda 5: Interpolazione polinomiale

Possiamo approssimare i m dati equispaziati (ottenuti campionando la funzione $f(x) = \exp\left(\frac{x}{2}\right)$) mediante un polinomio interpolatore di Lagrange. Il polinomio interpolatore di Lagrange è una forma polinomiale che passa attraverso un insieme di dati e può essere utilizzato per approssimare una funzione nei punti di campionamento.

Il polinomio interpolatore di Lagrange è definito come:

$$\Pi_n(x) = \sum_{k=0}^n y_k \varphi_k(x_k)$$

dove x_k sono i punti di campionamento e $\varphi_k(x)$ sono i polinomi di Lagrange definiti come:

$$\varphi_k(x) = \prod_{\substack{j=0 \\ j \neq k}}^n \frac{x - x_j}{x_k - x_j}$$

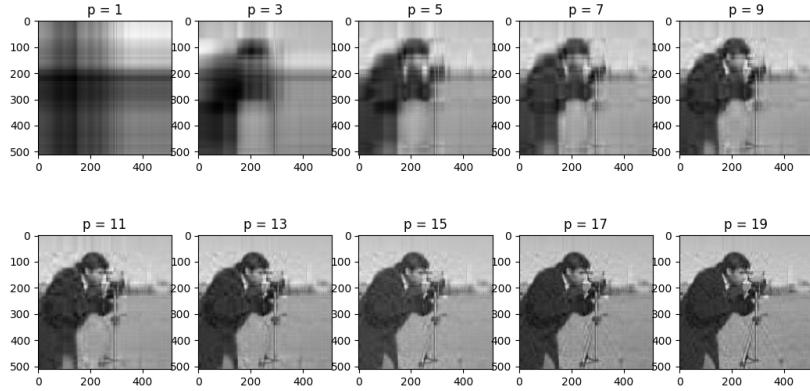


Figura 42: Compressione con SVD rispetto a p

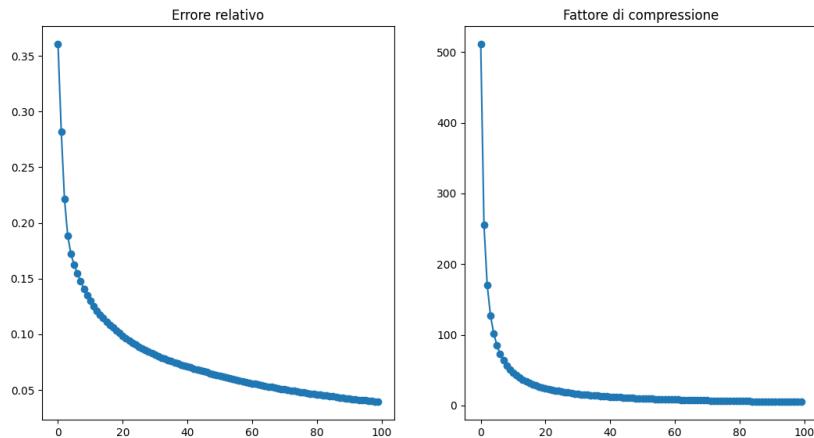


Figura 43: Errore relativo e compressione al variare di p

Utilizzando questo polinomio interpolatore, possiamo approssimare la funzione $f(x) = \exp\left(\frac{x}{2}\right)$ nei dati campionati.

Si può risolvere il problema dei minimi quadrati, che consiste nella minimizzazione del vettore residuo r , ossia $\min \|Ax - b\| = \min \|r\|$, mediante equazioni normali o SVD.

$$A = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}, \quad b = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{bmatrix}$$

- Con equazioni normali:

$$A^T A x = A^T b$$

Se $A^T A$ ha rango massimo, posso ad esempio fattorizzare con Cholesky. Altrimenti impongo condizione di norma minima e risolvo con SVD.

- Con SVD:

$$x = \sum_{i=1}^n \frac{u_i^T \cdot b}{\sigma_i} v_i$$

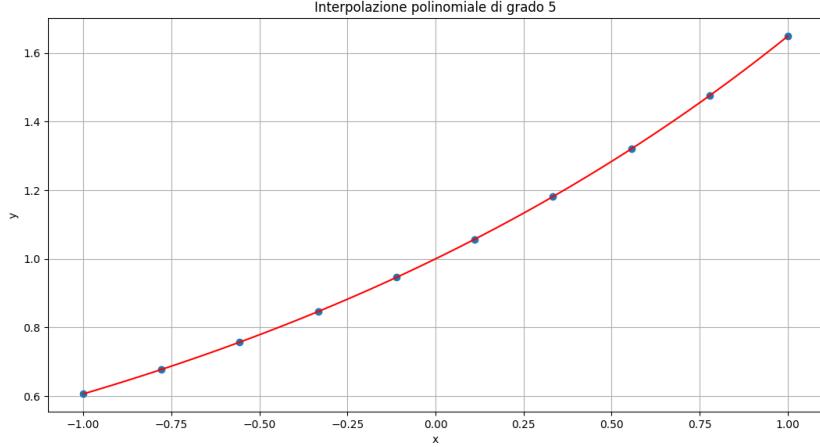


Figura 44: Polinomio interpolatore di $f(x) = \exp\left(\frac{x}{2}\right)$

Osservo che con grado 5 ottengo un'ottima interpolazione polinomiale.

Domanda 6: Interpolazione polinomiale

$$f(x) = \frac{1}{1 + 25x^2}$$

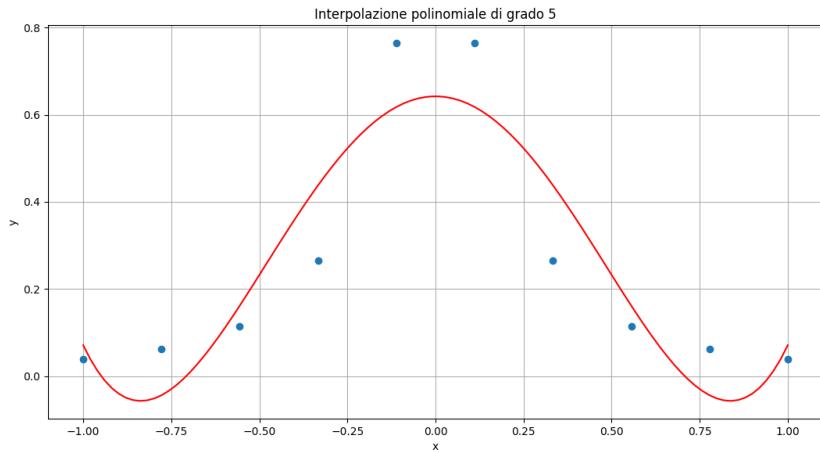


Figura 45: Polinomio interpolatore di $f(x) = \frac{1}{1+25x^2}$ con $n = 5$

Osservo che l'interpolazione ottenuta è poco soddisfacente: aumento il grado a $n = 7$.

Sebbene l'interpolazione sia leggermente migliorata, è chiaro vedere il fenomeno di Runge: l'aumento del grado del polinomio interpolatore non sempre porta a un miglioramento della qualità dell'interpolazione, e

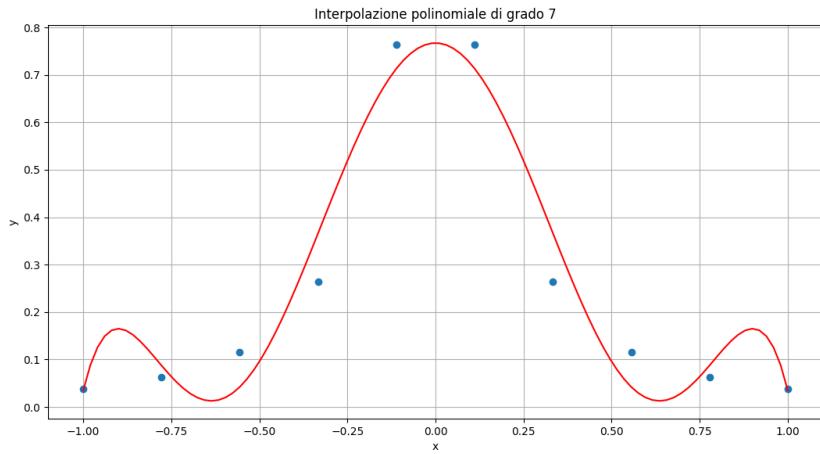


Figura 46: Polinomio interpolatore di $f(x) = \frac{1}{1+25x^2}$ con $n = 7$

in alcuni casi può causare oscillazioni indesiderate. Questo è particolarmente evidente nei punti di estremo interpolazione.

Il fenomeno di Runge suggerisce che, in certi casi, l'uso di polinomi di grado elevato per l'interpolazione può portare a risultati indesiderati. Una strategia per mitigare questo problema è l'uso l'interpolazione con i nodi di Chebyshev-Gauss-Lobatto: $x' = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{\pi i}{n}\right)$

Domanda 7: Interpolazione polinomiale

$$f(x) = \sin(x) + \cos(x)$$

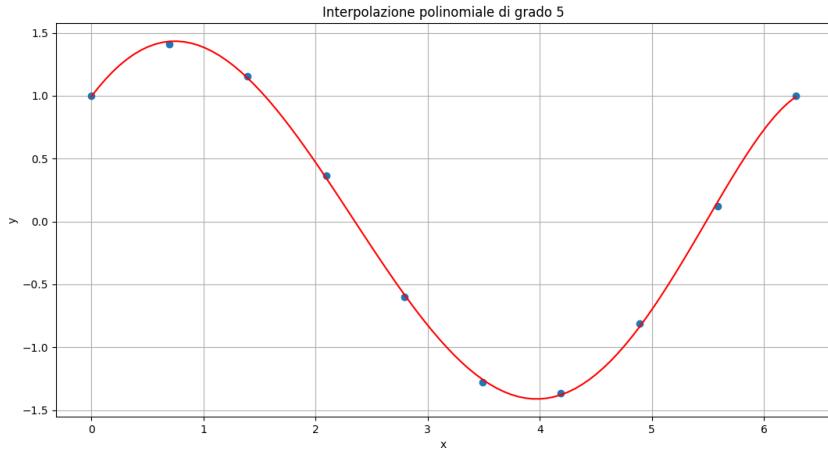


Figura 47: Polinomio interpolatore di $f(x) = \sin(x) + \cos(x)$

Domanda 8: Calcolo zero funzione

Discutere dei risultati per calcolare lo zero di $f(x) = e^x - x^2$ in $I = [-1, 1]$ con:

- Metodo di Newton
- $g(x) = x - f(x)e^{x/2}$
- $g(x) = x - f(x)e^{-x/2}$

Metodo di Approssimazioni Successive: Il metodo di approssimazioni successive è una tecnica iterativa basata sulla costruzione di una sequenza di punti x_k in modo che $x_{k+1} = g(x_k)$. L'obiettivo è trovare un punto fisso della funzione $g(x)$ tale che $x^* = g(x^*)$. Nel nostro caso, $g(x)$ è legato alla funzione $f(x)$ attraverso le due varianti $g(x) = x - f(x)e^{x/2}$ e $g(x) = x - f(x)e^{-x/2}$.

Metodo di Newton: Il metodo di Newton è una variante del metodo di approssimazioni successive in cui $g(x) = x - \frac{f(x)}{f'(x)}$.

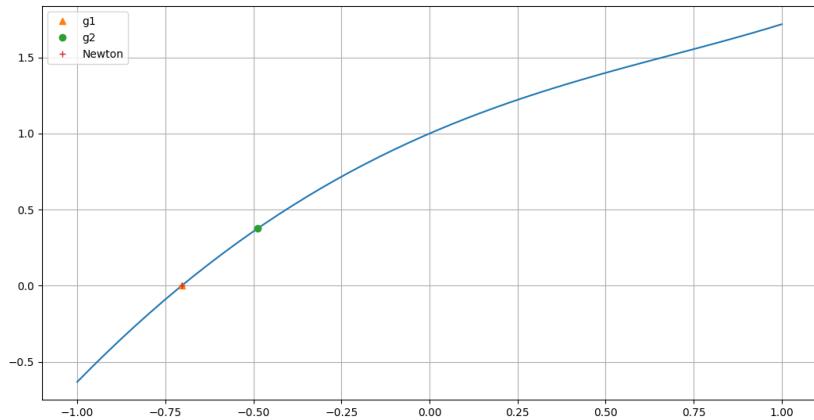


Figura 48: Solo g_1 e Newton convergono alla soluzione esatta

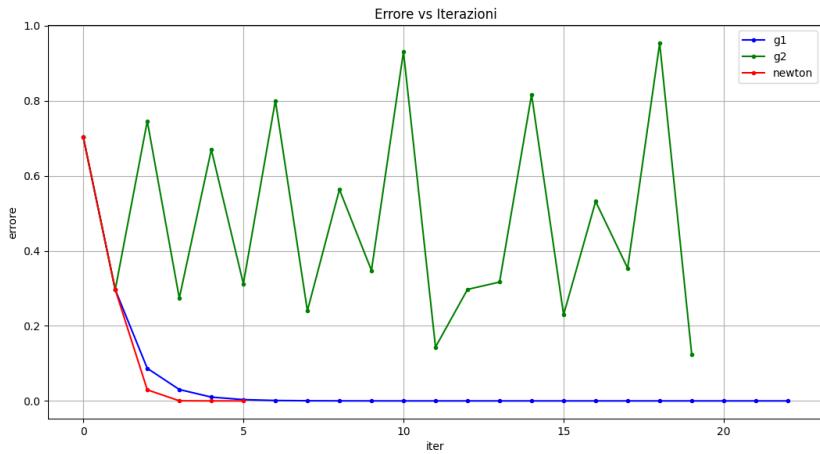


Figura 49: g_2 , in figura limitato a 20 iterazioni, non converge

Notiamo come si converga alla soluzione usando il metodo di Newton e il metodo delle approssimazioni successive con g_1 , mentre ciò non accade con g_2 .

Domanda 9: Calcolo zero funzione

Discutere dei risultati per calcolare lo zero di $f(x) = x^3 + 4x \cos 3(x) - 2$ in $I = [0, 2]$ con:

- Metodo di Newton

- $g(x) = \frac{2-x^3}{4 \cos(x)}$

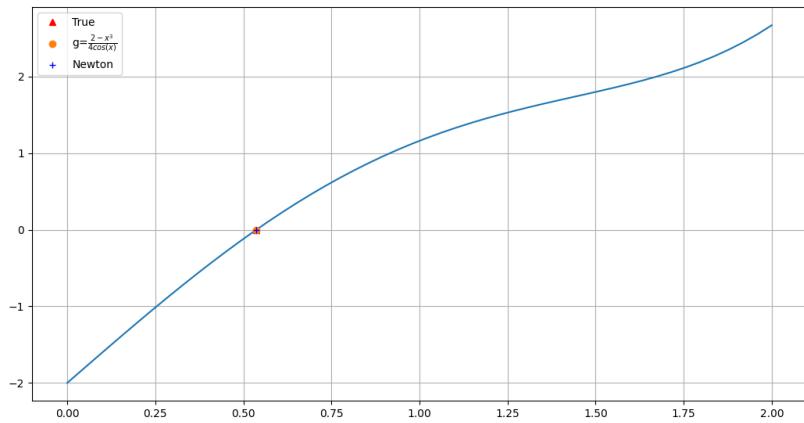


Figura 50: Sia Newton che $g(x) = \frac{2-x^3}{4 \cos(x)}$ convergono

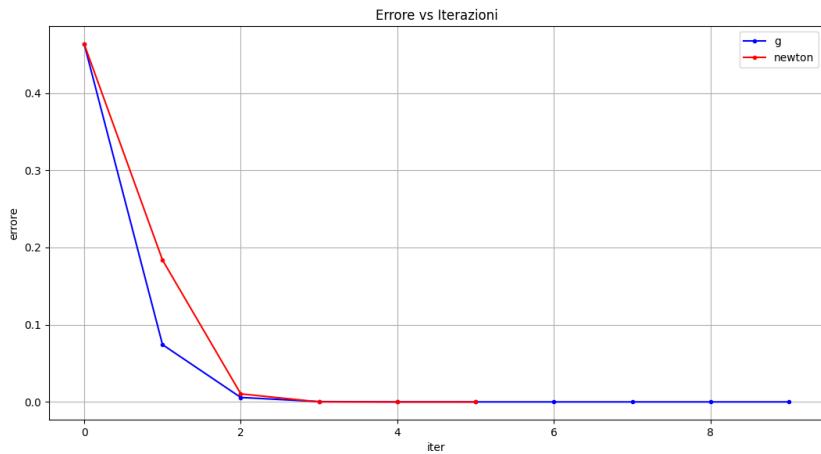


Figura 51: Confronto tra Newton e $g(x) = \frac{2-x^3}{4 \cos(x)}$

g si avvicina inizialmente più velocemente alla soluzione rispetto a Newton, ma impiega un maggior numero di iterazioni per arrivare alla tolleranza prestabilita.

Domanda 10: Calcolo zero funzione

Discutere dei risultati per calcolare lo zero di $f(x) = x - x^{\frac{1}{3}} - 2$ in $I = [3, 5]$ con:

- Metodo di Newton

- $g(x) = x^{\frac{1}{3}} + 2$

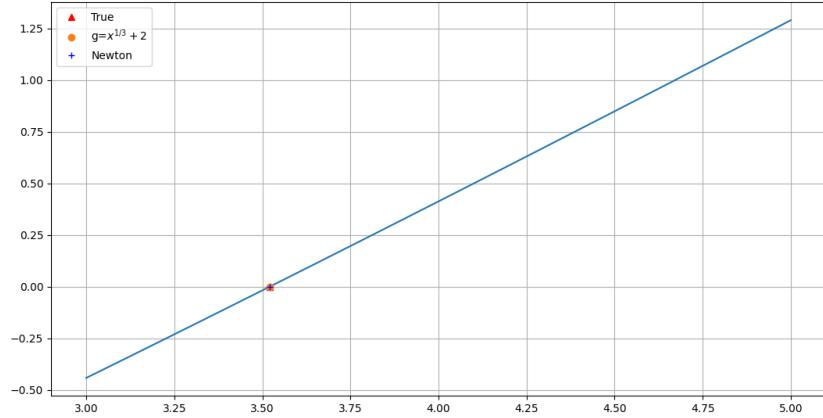


Figura 52: Sia Newton che $g(x) = x^{\frac{1}{3}} + 2$ convergono

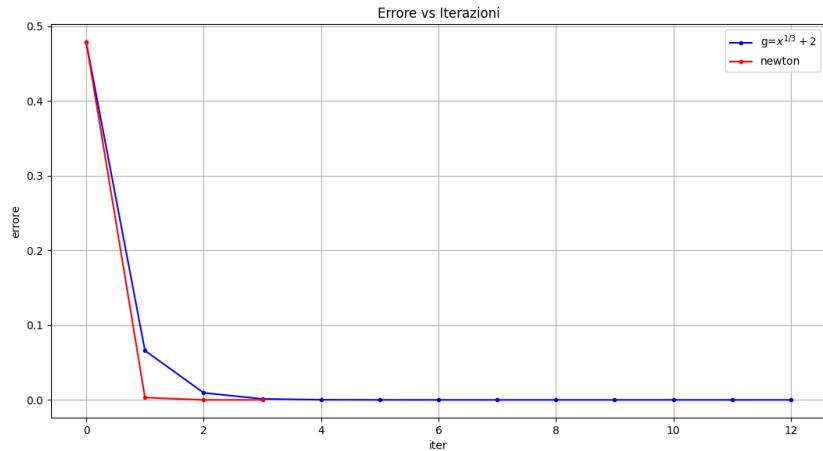


Figura 53: Newton converge più rapidamente di $g(x) = x^{\frac{1}{3}} + 2$

Notiamo come l'utilizzo del metodo di Newton si riveli nuovamente il più veloce: questo in quanto esso ha velocità di convergenza quadratica.

Domanda 11: Metodo del gradiente

Discutere della minimizzazione di $f(x, y) = 3(x - 2)^2 + (y - 1)^2$

Metodo del Gradiente: Il metodo del gradiente è un algoritmo utilizzato per trovare il minimo di una funzione. Ad ogni passo si calcola $x_{k+1} = x_k + \alpha_k p_k$, con α_k lunghezza del passo e p_k direzione di discesa, quest'ultima pari all'antigradiente nel metodo del gradiente $= -\nabla f(x, y)$, e usiamo due modi per ottenere α_k : il primo è semplicemente tenendo un valore costante, il secondo è tramite un algoritmo che sfrutta il backtracking.

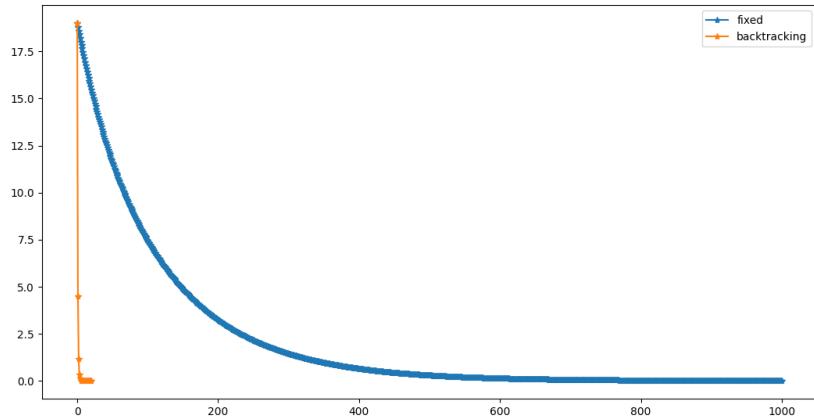


Figura 54: Valore della funzione a confronto con α_k fisso e scelto con algoritmo di backtracking

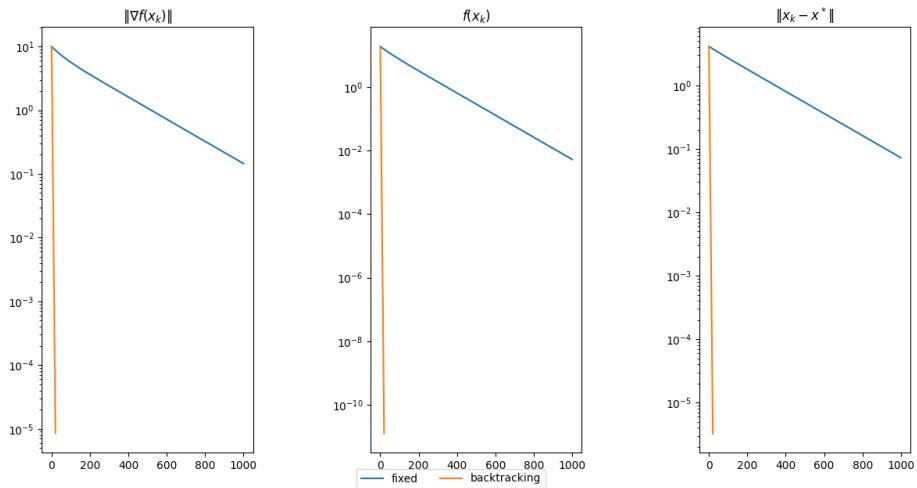


Figura 55: Confronto tra norma del gradiente, valore della funzione ed errore

In questo caso, è evidente come l'impiego dell'algoritmo di backtracking conduca a una soluzione con un numero di iterazioni notevolmente inferiore rispetto all'utilizzo di α_k costante. Ogni metrica mostra una rapida decrescita, e il valore minimo viene raggiunto in soli 20 passaggi, a differenza della lunghezza costante del passo, che si conclude dopo 1000 iterazioni non perché ha raggiunto il minimo, bensì per il limite del numero massimo di iterazioni.

Domanda 12: Metodo del gradiente

Discutere della minimizzazione di $f(x, y) = 100(y - x^2)^2 + (1 - x)^2$

Con la funzione di Rosenbrock, notevolmente difficile da minimizzare, nessuno dei metodi raggiunge la tolleranza desiderata entro 1000 iterazioni, ma il backtracking mostra una decrescita più rapida. La norma dell'errore oscillante con il backtracking indica variazioni nella convergenza, mostrando la flessibilità dell'algoritmo rispetto a α_k costante.

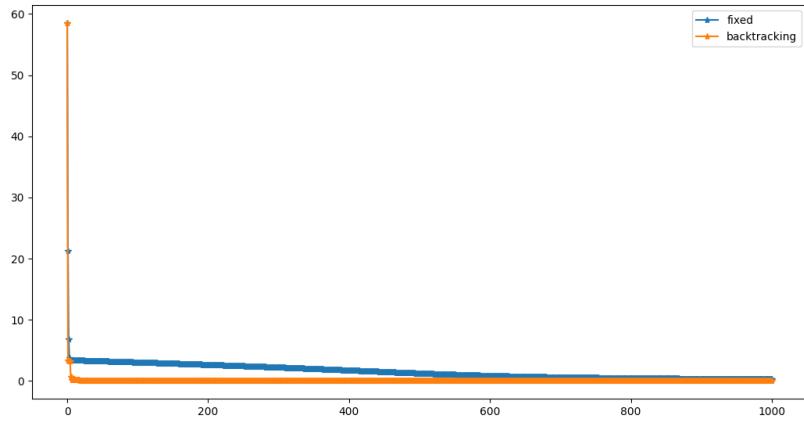


Figura 56: Valore della funzione a confronto con α_k fisso e scelto con algoritmo di backtracking

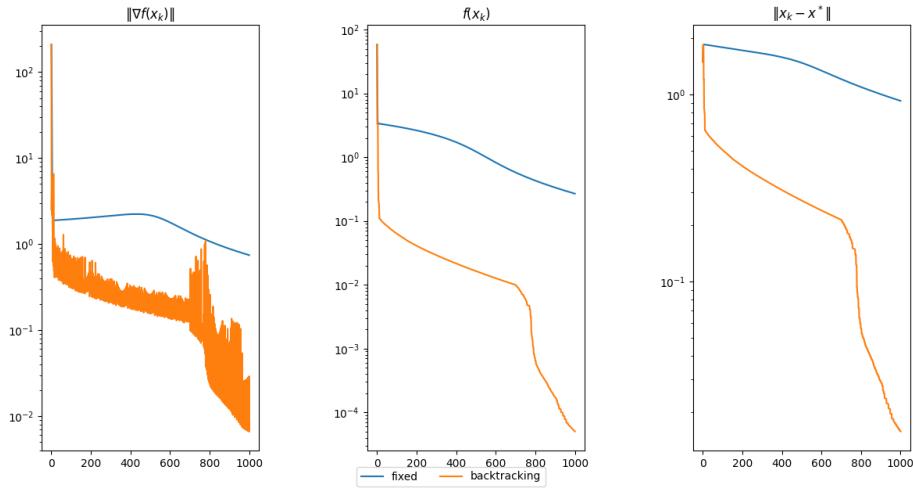


Figura 57: Confronto tra norma del gradiente, valore della funzione ed errore

Domanda 13: Deblur

Vogliamo provare a ricostruire l'immagine originale, data l'immagine degradata da blur e rumore. Questo lo facciamo risolvendo il problema dei minimi quadrati:

$$\min_f \frac{1}{2} \|Af - g\|_2^2$$

con f l'immagine da ricostruire, g immagine osservata corrotta.

Si osserva un effetto indesiderato causato dalla mancanza di regolarizzazione. Per affrontare questo problema, proviamo a risolvere il sistema integrando il metodo di regolarizzazione di Tikhonov, espresso come:

$$\min_f \frac{1}{2} \|Af - g\|_2^2 + \lambda \phi(f)$$

dove $\phi(f)$ rappresenta il termine di regolarizzazione e λ è il parametro di regolarizzazione. λ è il parametro che viene regolato per ottimizzare il PSNR (Peak Signal-to-Noise Ratio) e minimizzare il MSE (Mean Squared Error).

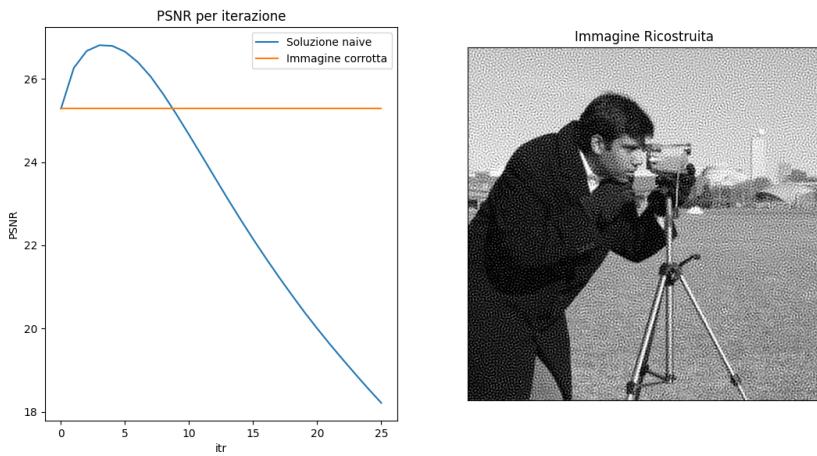


Figura 58: Ricostruzione naïve

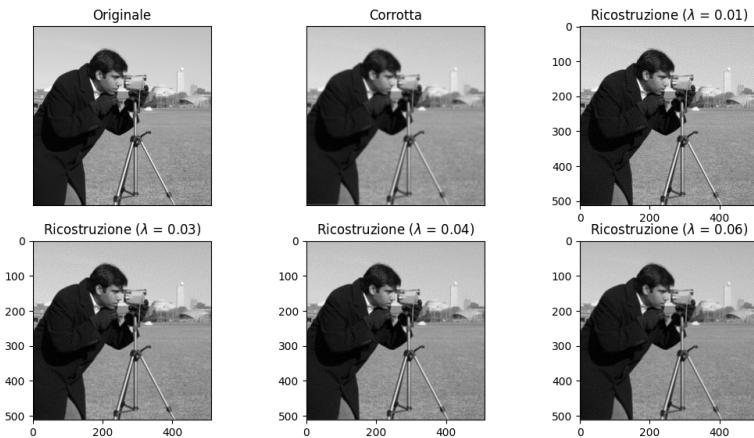


Figura 59: Ricostruzione con regolarizzazione di Tikhonov al variare di λ

Si può vedere come la scelta del parametro λ giochi un ruolo fondamentale nella qualità della ricostruzione. Una corretta selezione di λ porta a un miglioramento del rapporto PSNR e MSE, indicando che la regolarizzazione di Tikhonov può essere efficace nel migliorare la fedeltà della ricostruzione dell'immagine.

Domanda 14: Super Resolution

La super risoluzione è una tecnica utilizzata per migliorare la qualità di immagini a bassa risoluzione, generando un'immagine a risoluzione più alta rispetto a quella acquisita, aumentando dunque il numero di pixel nell'immagine.

Si introduce un operatore di downsampling S , rendendo il problema regolarizzato:

$$\min_f \frac{1}{2} \|SAf - g\|_2^2 + \lambda \phi(f)$$

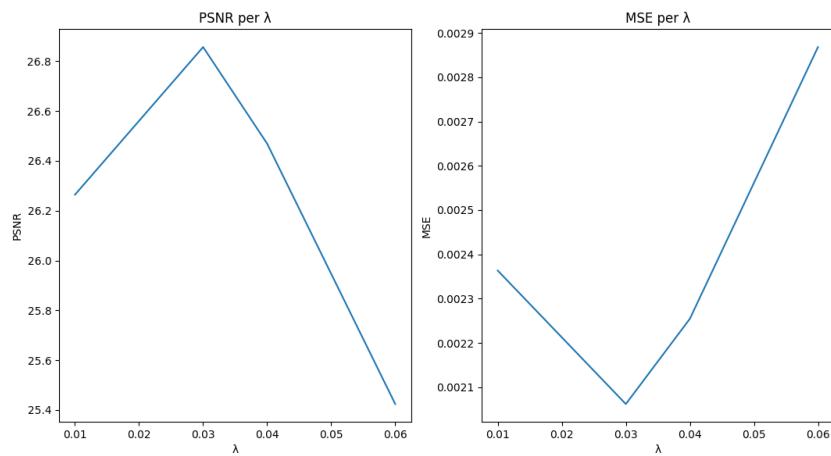


Figura 60: MSNR e MSE con i precedenti valori λ

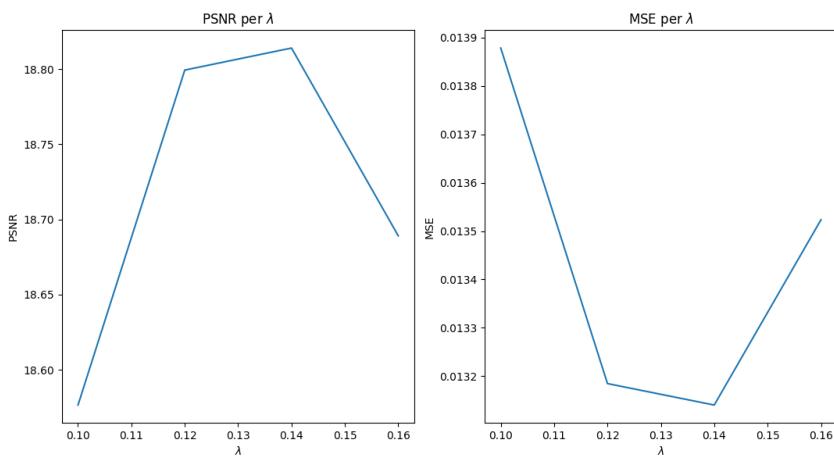


Figura 61: Ricostruzione con regolarizzazione e operatore downsampling dal fattore di scaling $S = 8$

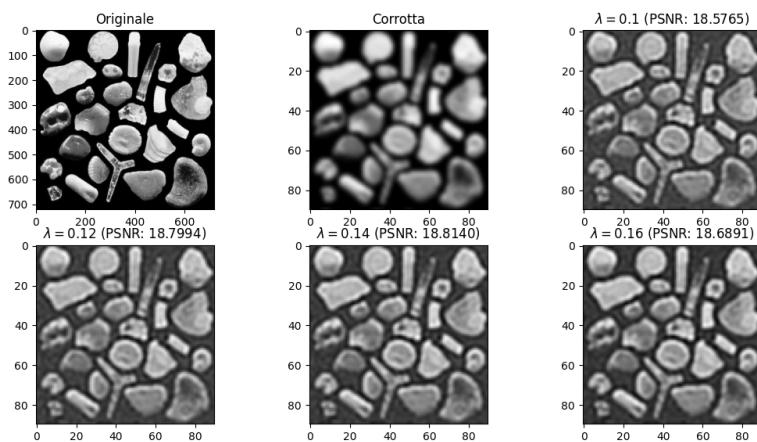


Figura 62: MSNR e MSE con i precedenti valori λ