

Codice di laboratorio e domande orale  
Corso di Informatica  
Esame di Calcolo Numerico

Alessandro Amella

22 gennaio 2024

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



## Indice

<b>Esercitazione 1: Epsilon macchina, Numpy, Matplotlib</b>	<b>2</b>
Esercizio 1.1 . . . . .	2
Esercizio 2.1 . . . . .	3
Esercizio 2.2 . . . . .	4
<b>Esercitazione 2: Norme, condizionamento, fattorizzazione LU</b>	<b>5</b>
Esercizio 1.1 . . . . .	5
Esercizio 2.1 . . . . .	7
Esercizio 2.2 . . . . .	9
<b>Esercitazione 3: Ricostruzione immagini con SVD</b>	<b>11</b>
Esercizio 1.1 . . . . .	11
Esercizio 1.2 . . . . .	13
<b>Esercitazione 4: Minimi quadrati con equazioni normali e SVD</b>	<b>15</b>
Esercizio 1.1 . . . . .	15
Esercizio 1.2 . . . . .	16
<b>Esercitazione 5: Metodi iterativi per radici di funzioni</b>	<b>18</b>
Esercizio 1.1 . . . . .	18
Esercizio 1.2 . . . . .	22
<b>Esercitazione 6: Minimizzazione e metodi di discesa del gradiente</b>	<b>24</b>
Esercizio 1.1 . . . . .	24
Esercizio 1.2 . . . . .	26
Esercizio 1.3 . . . . .	29
<b>Esercitazione 7: Deblurring di immagini</b>	<b>31</b>
Esercizio 1.1 Problema test . . . . .	31
Esercizio 1.2 Soluzione naiva . . . . .	34
Esercizio 1.3 Soluzione regolarizzata . . . . .	36
Esercizio 1.4 . . . . .	40
<b>Domanda 1: Fattorizzazione LU con pivoting</b>	<b>47</b>

<b>Domanda 2: Fattorizzazione con Cholesky</b>	<b>47</b>
<b>Domanda 3: Cholesky con matrice di Hilbert</b>	<b>48</b>
<b>Domanda 4: Compressione SVD</b>	<b>49</b>
<b>Domanda 5: Interpolazione polinomiale</b>	<b>51</b>
<b>Domanda 6: Interpolazione polinomiale</b>	<b>52</b>
<b>Domanda 7: Interpolazione polinomiale</b>	<b>53</b>
<b>Domanda 8: Calcolo zero funzione</b>	<b>54</b>
<b>Domanda 9: Calcolo zero funzione</b>	<b>55</b>
<b>Domanda 10: Calcolo zero funzione</b>	<b>57</b>
<b>Domanda 11: Metodo del gradiente</b>	<b>58</b>
<b>Domanda 12: Metodo del gradiente</b>	<b>59</b>
<b>Domanda 13: Deblur</b>	<b>61</b>
<b>Domanda 14: Super Resolution</b>	<b>66</b>

## 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  $\varepsilon$  in (float) default precision with a `while` construct. Compute also the mantissa digits number.

---

```

1 def machine_epsilon(func=float):
2     eps = func(1)
3     while func(1) + func(eps) != func(1):
4         eps_last = eps
5         eps = func(eps) / func(2)
6     return eps_last
7
8 print(machine_epsilon(float)) # 2.220446049250313e-16
9
10 def mantissa_digits(eps):
11     return -math.log10(eps)
12
13 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`.

---

```

1 import numpy as np
2 print(machine_epsilon(np.float16)) # 0.000977

```

---

```
3 print(machine_epsilon(np.float32)) # 1.1920929e-07
4
5 print(np.finfo(float).eps) # 2.220446049250313e-16
6 print(np.finfo(np.float16).eps) # 0.000977
7 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].

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Array [0, 10] con 100 elementi
5 x = np.linspace(0, 10, 100)
6
7 # Calcolo cos(x) e sin(x)
8 y1 = np.cos(x)
9 y2 = np.sin(x)
10
11 # Mostra i grafici
12 plt.plot(x, y1, color='red', label='cos')
13 plt.plot(x, y2, color='blue', label='sin')
14
15 # Aggiungi legenda, titolo e label
16 plt.legend()
17 plt.title('Grafico seno e coseno')
18 plt.xlabel('x')
19 plt.ylabel('y')
20 plt.show()
```

---

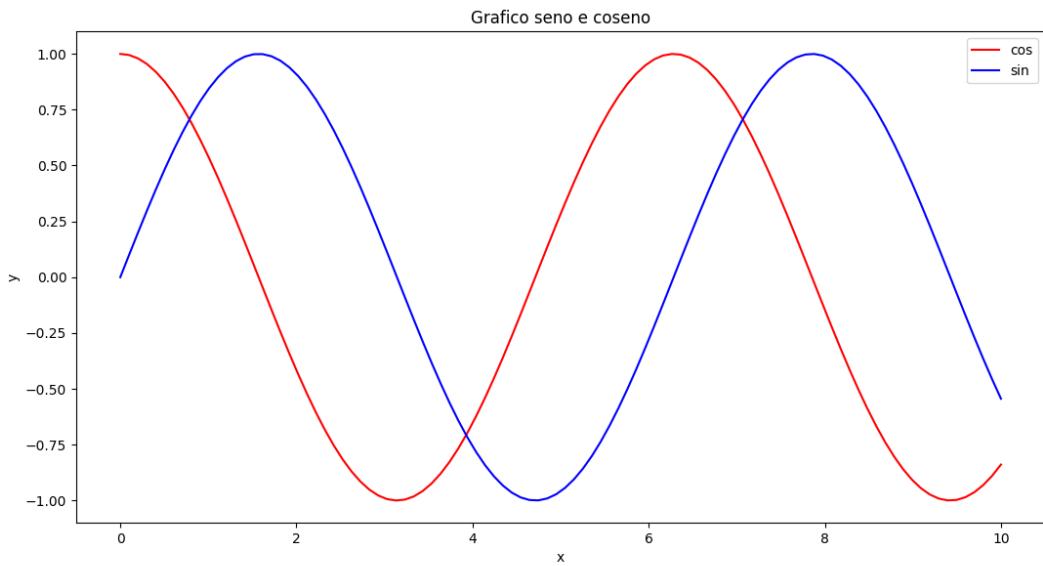


Figura 1: Funzioni  $\sin(x)$  e  $\cos(x)$

## Esercizio 2.2

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

---

```

1 def fib_iter(n):
2     a = 0 # F(0)
3     b = 1 # F(1)
4     for i in range(n):
5         c = a + b
6         a = b
7         b = c
8     return a
9
10 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.

---

```

1 def fib_ratio(k):
2     a = fib_iter(k)
3     b = fib_iter(k - 1)
4     if b == 0:
5         return math.inf
6     return a / b
7
8 print(fib_ratio(10)) # 1.6176470588235294

```

---

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

---

```

1 phi = (1 + math.sqrt(5)) / 2
2 def fib_ratio_convergence(k):
3     return phi - fib_ratio(k)
4
5 print(fib_ratio_convergence(10)) # 0.00038692992636546464

```

---

4. Create a plot of the error (with respect to  $\varphi$ ).

---

```

1 errors = []
2 for n in range(30):
3     r = fib_ratio(n)
4     e = abs(phi - r)
5     errors.append(e)
6 plt.plot(errors)
7 plt.title("Errore approssimazione di Fibonacci")
8 plt.xlabel("n")
9 plt.ylabel("Errore")
10 plt.show()

```

---

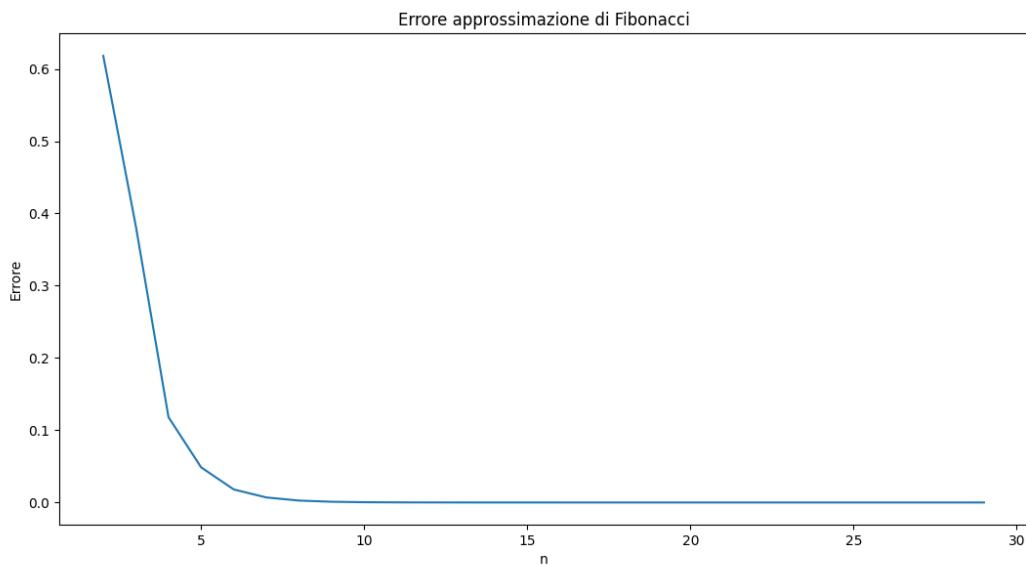


Figura 2: Errore rispetto a  $\varphi$

## 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()`

---

```

1 import numpy as np
2
3 n = 2

```

---

```

4 A = np.array([[1, 2], [0.499, 1.001]])
5
6 print ('Norme di A:')
7 norm1 = np.linalg.norm(A, 1)
8 norm2 = np.linalg.norm(A, 2)
9 normfro = np.linalg.norm(A, 'fro')
10 norminf = np.linalg.norm(A, np.inf)
11
12 print('Norma1 = ', norm1, '\n') # 3.001
13 print('Norma2 = ', norm2, '\n') # 2.500200104037774
14 print('Normafro = ', normfro, '\n') # 2.5002003919686118
15 print('Norma infinito = ', norminf, '\n') # 3.0

```

---

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

```

1 cond1 = np.linalg.cond(A, 1)
2 cond2 = np.linalg.cond(A, 2)
3 cond_fro = np.linalg.cond(A, 'fro')
4 cond_inf = np.linalg.cond(A, np.inf)
5
6 print ('K(A)_1 = ', cond1, '\n') # 3001.0000000001082
7 print ('K(A)_2 = ', cond2, '\n') # 2083.666853410337
8 print ('K(A)_fro = ', cond_fro, '\n') # 2083.6673333334084
9 print ('K(A)_inf = ', cond_inf, '\n') # 3001.0000000001082

```

---

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

```

1 x = np.ones((2,1))
2 b = A @ x
3 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  $A\tilde{x} = \tilde{b}$ .

```

1 btilde = np.array([[3], [1.4985]])
2 xtilde = np.array([[2, 0.5]]).T
3 my_btilde = A @ xtilde
4 print ('A*xtilde = ', btilde) # [[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  $\Delta b$  con  $\Delta x$ .

```

1 deltax = np.linalg.norm(x-xtilde, ord=2)
2 deltab = np.linalg.norm(b-btilde, ord=2)
3
4 print ('delta x = ', deltax) # 1.118033988749895
5 print ('delta b = ', deltab) # 0.0015000000000000568

```

---

## Esercizio 2.1

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

---

```
1 import numpy as np
2
3 A = np.array ([[ 3,-1, 1,-2], [0, 2, 5, -1], [1, 0, -7, 1], [0, 2, 1, 1] ])
4 x = np.ones((4,1))
5 b = A @ x
6
7 condA = np.linalg.cond(A, 2)
8
9 print('x: \n', x , '\n') # [[1.], [1.], [1.], [1.]]
10 print('x.shape: ', x.shape, '\n') # (4, 1)
11 print('b: \n', b , '\n') # [[1.], [6.], [-5.], [4.]]
12 print('b.shape: ', b.shape, '\n') # (4, 1)
13 print('A: \n', A, '\n') # ...
14 print('A.shape: ', A.shape, '\n') # (4, 4)
15 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.

---

```
1 import scipy
2 import scipy.linalg
3 from scipy.linalg import lu_factor as LUdec # pivoting
4 from scipy.linalg import lu as LUfull # partial pivoting
5
6 lu, piv = LUdec(A)
7
8 print('lu',lu,'\\n')
9 print('piv',piv,'\\n')
```

---

---

```
1 lu [[ 3.         -1.          1.         -2.          ]
2      [ 0.          2.          5.         -1.          ]
3      [ 0.33333333  0.16666667 -8.16666667  1.83333333]
4      [ 0.          1.          0.48979592  1.10204082]]
5
6 piv [0 1 2 3]
```

---

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

---

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

---

---

```
1 # IMPLEMENTAZIONE ALTERNATIVA - 1
2 P, L, U = LUfull(A)
```

---

```

3 print ('P*L*U = ', np.matmul(P , np.matmul(L, U)))
4 print ('diff = ', np.linalg.norm(A - np.matmul(P , np.matmul(L, U)), 'fro' ) )
5 invP = np.linalg.inv(P)
6 y = scipy.linalg.solve_triangular(L, invP @ b, lower=True, unit_diagonal=True)
7 my_x = scipy.linalg.solve_triangular(U, y, lower=False)
8 print('\nSoluzione calcolata: ', my_x)
9 print('norm =', scipy.linalg.norm(x-my_x, 'fro'))

```

---

```

1 Soluzione calcolata: [[1.]
2 [1.]
3 [1.]
4 [1.]]

```

---

- Stampare la soluzione calcolata e valutarne la correttezza.

```

1 my_x =
2 [[1.]
3 [1.]
4 [1.]
5 [1.]]
6 norm = 7.021666937153402e-16

```

---

```

1 # IMPLEMENTAZIONE ALTERNATIVA - 1

```

---

```

1 P = [[1. 0. 0. 0.]
2 [0. 1. 0. 0.]
3 [0. 0. 1. 0.]
4 [0. 0. 0. 1.]]
5 L = [[1. 0. 0. 0.]
6 [0. 1. 0. 0.]
7 [0.33333333 0.16666667 1. 0.]
8 [0. 1. 0.48979592 1.]]
9 U = [[3. -1. 1. -2.]
10 [0. 2. 5. -1.]
11 [0. 0. -8.1666667 1.8333333]
12 [0. 0. 0. 1.10204082]]
13 P*L*U = [[3. -1. 1. -2.]
14 [0. 2. 5. -1.]
15 [1. 0. -7. 1.]
16 [0. 2. 1. 1.]]
17 diff = 1.5700924586837752e-16
18
19 Soluzione calcolata: [[1.]
20 [1.]
21 [1.]
22 [1.]]
23 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  $n$ .

---

```

1 n = 5
2 A = scipy.linalg.hilbert(n)
3 x = np.ones((n,1))
4 b = A @ x
5
6 condA = np.linalg.cond(A, 2)
7
8 print('x: \n', x, '\n')
9 print('x.shape: ', x.shape, '\n' )
10 print('b: \n', b, '\n')
11 print('b.shape: ', b.shape, '\n' )
12 print('A: \n', A, '\n')
13 print('A.shape: ', A.shape, '\n' )
14 print('K(A)=', condA, '\n') # 476607.2502425855

```

---

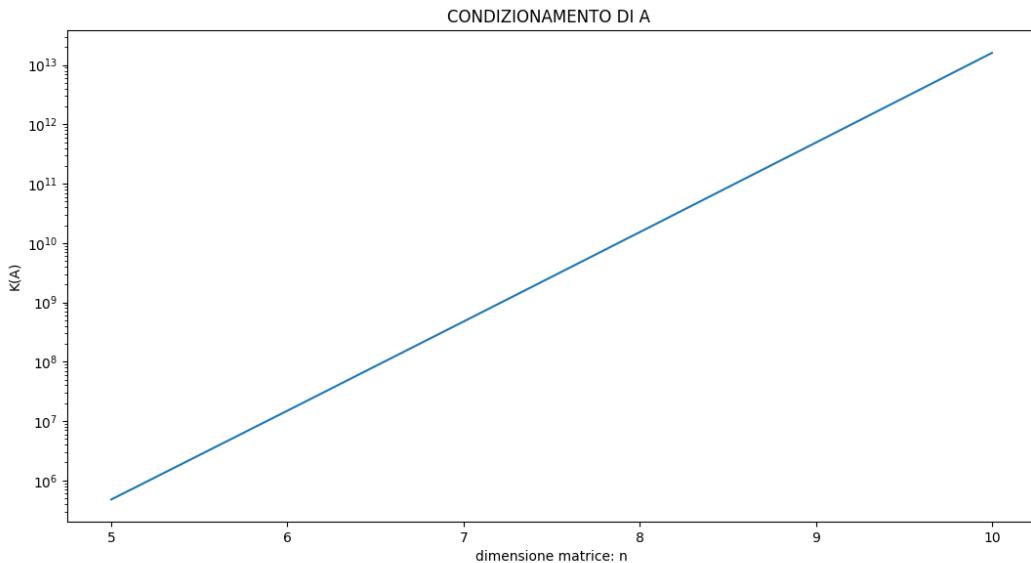


Figura 3:  $K(A)$  al variare di  $n$

- 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.

---

```

1 L = scipy.linalg.cholesky(A, lower=True)
2 print('L:', L, '\n')
3
4 print('L.T*L =', scipy.linalg.norm(A-np.matmul(np.transpose(L),L))) # 0.9734839217207908
5 print('err = ', scipy.linalg.norm(A-np.matmul(np.transpose(L),L), 'fro')) # 0.9734839217207908

```

```

6
7  y = scipy.linalg.solve_triangular(L, b, lower=True)
8  my_x = scipy.linalg.solve_triangular(L.T, y, lower=False)
9  print('my_x = \n ', my_x)
10
11 print('norm =', np.linalg.norm(x-my_x, 'fro')) # 2.6732599660997558e-12
12
13
14 K_A = np.zeros((6,1))
15 Err = np.zeros((6,1))

```

---

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

```

1 for n in np.arange(5,11):
2     # creazione dati e problema test
3     A = scipy.linalg.hilbert(n)
4     x = np.ones((n,1))
5     b = A @ x
6
7     # numero di condizione
8     K_A[n-5] = np.linalg.cond(A, 2)
9
10    # fattorizzazione
11    L = scipy.linalg.cholesky(A, lower=True)
12    y = scipy.linalg.solve_triangular(L, b, lower=True)
13    my_x = scipy.linalg.solve_triangular(L.T, y, lower=False)
14
15    # errore relativo
16    Err[n-5] = np.linalg.norm(x-my_x, 'fro')/np.linalg.norm(x, 'fro')
17
18 xplot = np.arange(5,11)
19
20 # grafico del numero di condizione vs dim
21 plt.semilogy(xplot, K_A)
22 plt.title('CONDIZIONAMENTO DI A ')
23 plt.xlabel('dimensione matrice: n')
24 plt.ylabel('K(A)')
25 plt.show()
26
27
28 # grafico errore in norma 2 in funzione della dimensione del sistema
29 plt.plot(xplot, Err)
30 plt.title('Errore relativo')
31 plt.xlabel('dimensione matrice: n')
32 plt.ylabel('Err= ||my_x-x||/||x||')
33 plt.show()

```

---

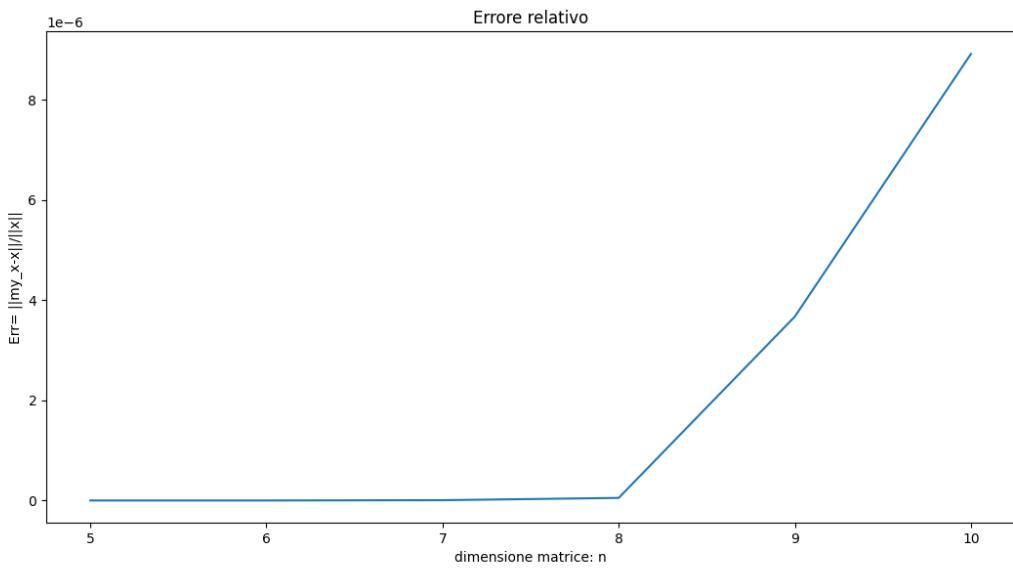


Figura 4:  $K(A)$  al variare di  $n$

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

---

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.linalg
4 from skimage import data
5
6 A = data.camera()
7
8 U, s, Vh = np.linalg.svd(A)
9
10 A_p = np.zeros(A.shape)
11 p_max = 10
12
13 for i in range(p_max):
14     ui = U[:, i]
15     vi = Vh[i, :]
16
17     A_p = A_p + s[i] * np.outer(ui, vi)

```

---

- Visualizzare l'immagine  $A_p$ .

---

```

1 plt.figure(figsize=(20, 10))
2
3 fig1 = plt.subplot(1, 2, 1)
4 fig1.imshow(A, cmap='gray')
5 plt.title('True image')
6
7 fig2 = plt.subplot(1, 2, 2)
8 fig2.imshow(A_p, cmap='gray')
9 plt.title('Reconstructed image with p = ' + str(p_max))
10
11 plt.show()

```

---

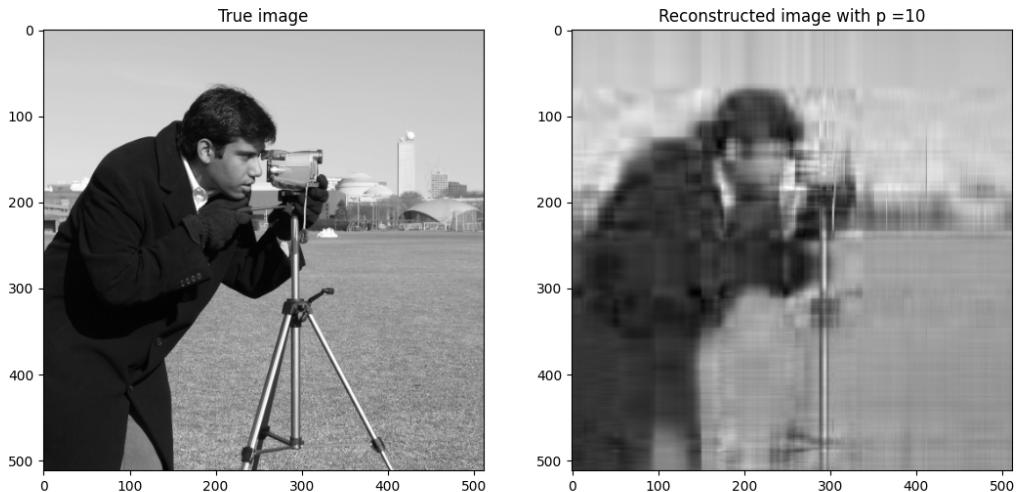


Figura 5:  $A_p$  ricostruita con  $p = 10$

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

---

```

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

```

---

- Calcolare il fattore di compressione

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

---

```

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

```

---

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

---

```

1 p_max = 100
2 A_p = np.zeros(A.shape)
3 err_rel = np.zeros((p_max))
4 c = np.zeros((p_max))
5
6 for i in range(p_max):
7     ui = U[:, i]
8     vi = Vh[i, :]
9
10    A_p = A_p + s[i] * np.outer(ui, vi)
11
12    err_rel[i] = np.linalg.norm(A - A_p) / np.linalg.norm(A)
13    c[i] = 1 / (i + 1) * (min(A.shape) - 1)
14
15 plt.figure(figsize=(10, 5))
16
17 fig1 = plt.subplot(1, 2, 1)
18 fig1.plot(err_rel, 'o-')
19 plt.title('Errore relativo')
20
21 fig2 = plt.subplot(1, 2, 2)
22 fig2.plot(c, 'o-')
23 plt.title('Fattore di compressione')
24
25 plt.show()

```

---

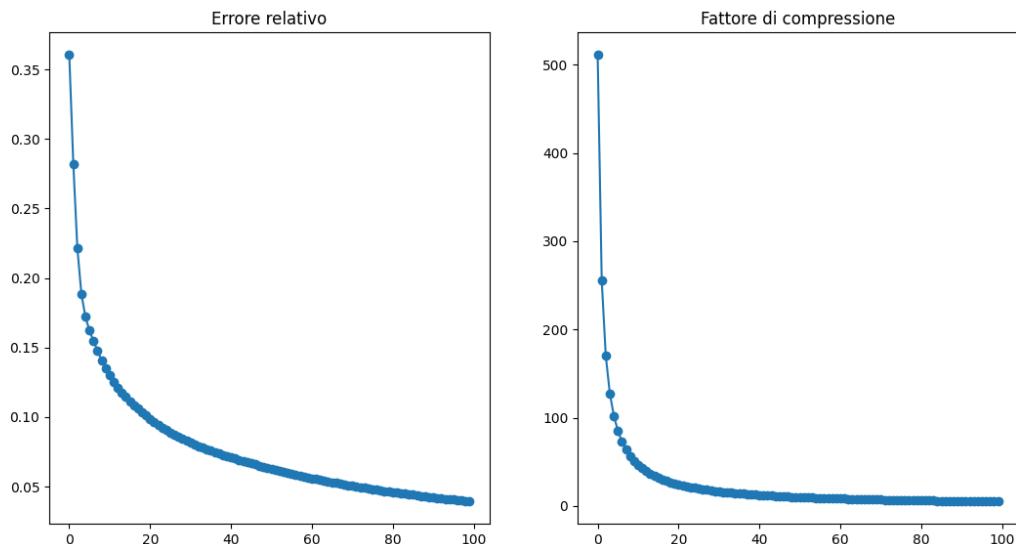


Figura 6: Errore relativo e fattore di compressione al variare di  $p$

## Esercizio 1.2

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

---

```
1 from skimage.io import imread
2 import os
3
4 # uso immagine ./phantom.png
5 img = 'phantom.png'
6 cur_dir = os.path.dirname(os.path.abspath(__file__))
7 A = imread(os.path.join(cur_dir, img), as_gray=True)
```

---

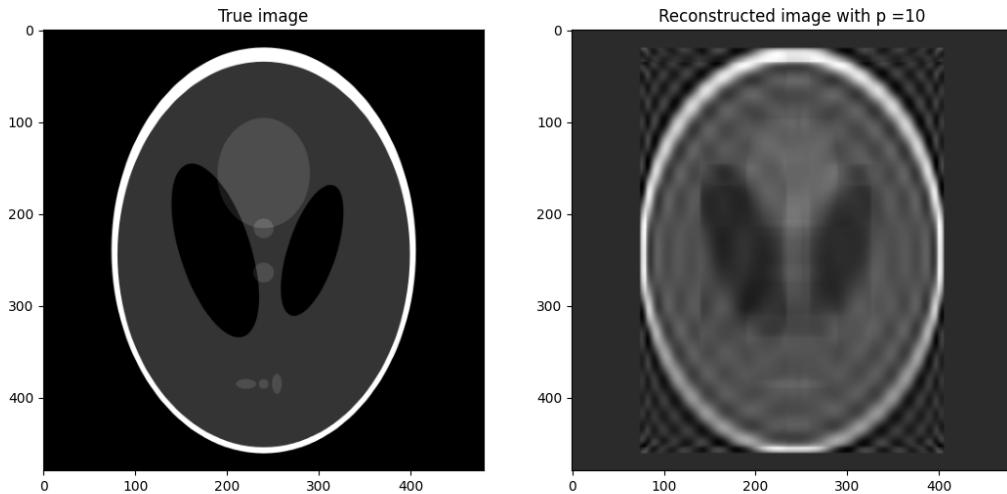


Figura 7:  $A_p$  ricostruita con  $p = 10$

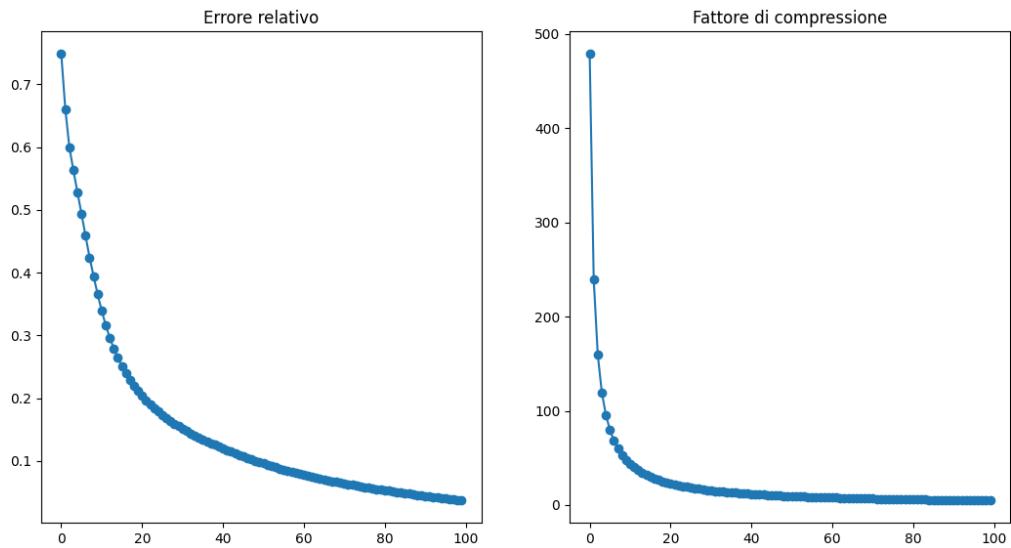


Figura 8: Errore relativo e fattore di compressione al variare di  $p$

## 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  $\alpha$  (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:

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

---

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.linalg
4 from scipy.linalg import lu_factor as LUdec
5
6 m = 100
7 n = 10
8
9 A = np.random.rand(m, n)
10
11 alpha_test = np.full(n, 0.5) # esempio con alpha = 0.5
12 y = A @ alpha_test # y = A*alpha
13
14 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]
15
16 ATA = A.T@A
17 ATy = A.T@y
18
19 lu, piv = LUdec(ATA)

```

```

20 alpha_LU = scipy.linalg.lu_solve((lu,piv), ATy)
21
22 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]
23
24 L = scipy.linalg.cholesky(ATA)
25 x = scipy.linalg.solve_triangular(np.transpose(L), ATy, lower=True)
26 alpha_chol = scipy.linalg.solve_triangular(L, x, lower=False)
27
28 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  $A$ .

```

1 U, s, Vh = scipy.linalg.svd(A)
2
3 alpha_svd = np.zeros(s.shape)
4
5 for i in range(n):
6     ui = U[:, i]
7     vi = Vh[i, :]
8
9     alpha_svd = alpha_svd + (ui.T@y/s[i])*vi
10
11 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  $\alpha$ , soluzione esatta utilizzata per generare il problema test.

```

1 err_rel_LU = np.linalg.norm(alpha_LU - alpha_test) / np.linalg.norm(alpha_test)
2 err_rel_chol = np.linalg.norm(alpha_chol - alpha_test) / np.linalg.norm(alpha_test)
3 err_rel_svd = np.linalg.norm(alpha_svd - alpha_test) / np.linalg.norm(alpha_test)
4
5 print('Errore relativo LU', err_rel_LU) # 6.7143255907214105e-15
6 print('Errore relativo chol', err_rel_chol) # 7.27852596108914e-15
7 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  $(x_i, f(x_i))$ .

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import scipy.linalg
4 from scipy.linalg import lu_factor as LUdec
5

```

```

6  case = 0 # [0, 1, 2]
7  m = 10
8  m_plot = 100
9
10 # Grado polinomio approssimante
11 n = 5
12
13 if case==0:
14     x = np.linspace(-1,1,m)
15     y = np.exp(x/2)
16 elif case==1:
17     x = np.linspace(-1,1,m)
18     y = 1/(1+25*(x**2))
19 elif case==2:
20     x = np.linspace(0,2*np.pi,m)
21     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.

```

1 A = np.zeros((m, n+1))
2
3 for i in range(n+1):
4     A[:, i] = x**i
5
6 U, s, Vh = scipy.linalg.svd(A)
7
8 alpha_svd = np.zeros(n+1)
9
10 for i in range(n+1):
11     ui = U[:, i]
12     vi = Vh[i, :]
13
14     alpha_svd = alpha_svd + (ui.T@y/s[i])*vi
15
16 print('alpha_svd', alpha_svd)
17 # [ 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.

```

1 # for n in [1,2,3,5,7]:
2 x_plot = np.linspace(x[0], x[-1], m_plot)
3 A_plot = np.zeros((m_plot, n+1))
4
5 for i in range(n+1):
6     A_plot[:, i] = x_plot**i
7
8 y_interpolation = A_plot@alpha_svd
9
10 plt.plot(x, y, 'o')
11 plt.plot(x_plot, y_interpolation, 'r')
12 plt.xlabel('x')

```

```

13 plt.ylabel('y')
14 plt.title(f'Interpolazione polinomiale di grado {n}')
15 plt.grid()
16 plt.show()

```

---

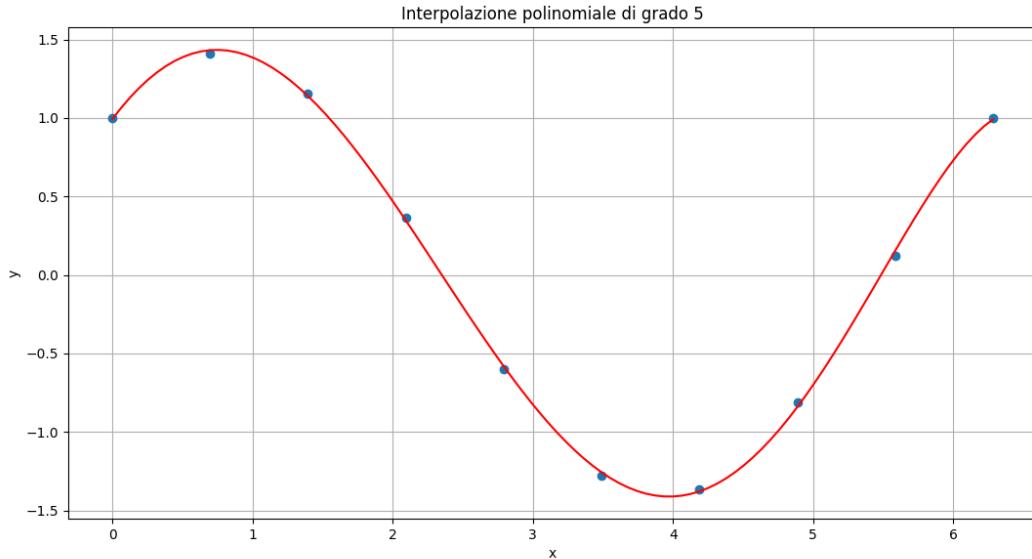


Figura 9: Interpolazione con  $n = 5$  di  $\sin(x) + \cos(x)$

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

```

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

```

---

```

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

```

---

## 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  $[-1, 1]$ .

---

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 f = lambda x: np.exp(x)-x**2
5
6 xTrue = -0.703467
7 fTrue = f(xTrue)
8 print('fTrue = ', fTrue) # 8.035078391532835e-07
9
10 xplot = np.linspace(-1, 1)
11 fplot = f(xplot)
12
13 plt.plot(xplot,fplot)
14 plt.plot(xTrue,fTrue, 'or', label='\$x^*\$')
15
16 plt.legend()
17 plt.grid()
18 plt.show()
```

---

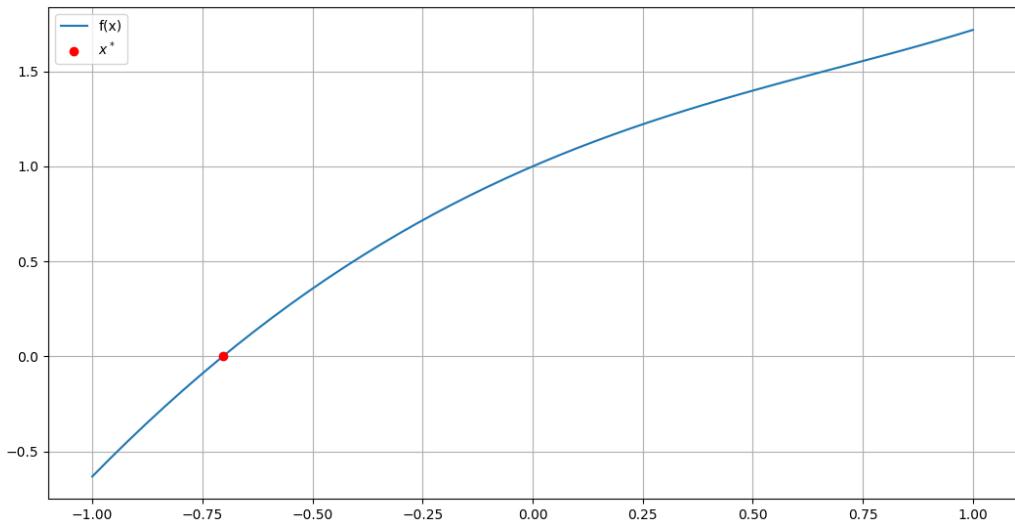


Figura 10: Grafico di  $f(x) = e^x - x^2$  con  $x^* = -0.703467$

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

---

```

1 def succ_app(f, g, tol_f, tol_x, max_it, x_true, x_0=0):
2     i=0
3     err=np.zeros(max_it+1, dtype=np.float64)
4     err[0]=tol_x+1
5     vecErrore=np.zeros(max_it+1, dtype=np.float64)
6     vecErrore[0] = np.abs(x_0-x_true)
7     x=x_0
8
9     while (err[i]>tol_x and i<max_it): # scarto assoluto tra iterati
10         x_new=g(x)
11         err[i+1]=np.abs(x_new-x)
12         vecErrore[i+1]=np.abs(x_new-x_true)
13         i=i+1
14         x=x_new
15     err=err[0:i]
16     vecErrore = vecErrore[0:i]
17     return (x, i, err, vecErrore)
18
19 def newton(f, df, tol_f, tol_x, max_it, x_true, x_0=0):
20     g = lambda x: x-f(x)/df(x)
21     (x, i, err, vecErrore) = succ_app(f, g, tol_f, tol_x, max_it, x_true, x_0)
22     return (x, i, err, vecErrore)
23
24 df = lambda x: np.exp(x)-2*x
25 g1 = lambda x: x-f(x)*np.exp(x/2)
26 g2 = lambda x: x-f(x)*np.exp(-x/2)

```

---

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

---

```

1 tol_x= 10**(-10)
2 tol_f = 10**(-6)
3 max_it=100
4 x_0= 0
5
6 [sol_g1, iter_g1, err_g1, vecErrore_g1]=succ_app(f, g1, tol_f, tol_x, max_it, x_true, x_0)
7 print('Metodo approssimazioni successive g1 \n x =',sol_g1,'\\n iter_new=', iter_g1)
8
9 plt.plot(sol_g1,f(sol_g1), 'o', label='g1') # converge in 23 iter
10
11 [sol_g2, iter_g2, err_g2, vecErrore_g2]=succ_app(f, g2, tol_f, tol_x, max_it, x_true, x_0)
12 print('Metodo approssimazioni successive g2 \n x =',sol_g2,'\\n iter_new=', iter_g2)
13
14 plt.plot(sol_g2,f(sol_g2), 'og', label='g2') # non converge dopo 100 iterate
15
16 [sol_newton, iter_newton, err_newton, vecErrore_newton]=newton(f, df, tol_f, tol_x, max_it, x_true, x_0)
17 print('Metodo Newton \n x =',sol_newton,'\\n iter_new=', iter_newton) # converge 6 it
18
19 plt.plot(sol_newton,f(sol_newton), 'ob', label='Newton')
20 plt.legend()
21 plt.grid()
22 plt.show()

```

---

---

```

1 Metodo approssimazioni successive g1
2 x = -0.7034674225096886
3 iter_new= 23
4 Metodo approssimazioni successive g2
5 x = -0.48775858993453886
6 iter_new= 100
7 Metodo Newton
8 x = -0.7034674224983917
9 iter_new= 6

```

---

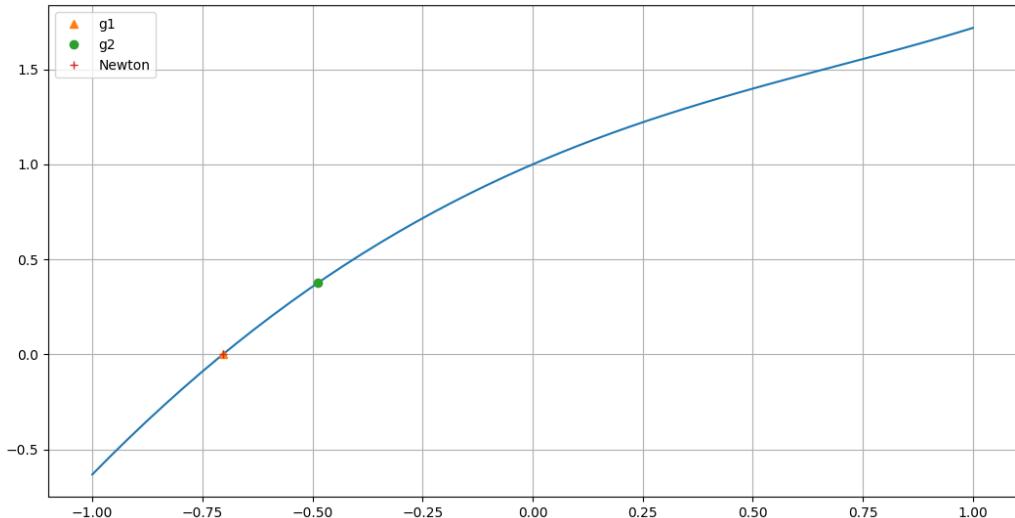


Figura 11: Solo  $g_1$  e Newton convergono alla soluzione esatta

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

---

```

1 # g1
2 plt.plot(vecErrore_g1, '.-', color='blue')
3 # g2
4 plt.plot(vecErrore_g2[:20], '.-', color='green')
5 # Newton
6 plt.plot(vecErrore_newton, '.-', color='red')
7
8 plt.legend( ("g1", "g2", "newton"))
9 plt.xlabel('iter')
10 plt.ylabel('errore')
11 plt.title('Errore vs Iterazioni')
12 plt.grid()
13 plt.show()

```

---

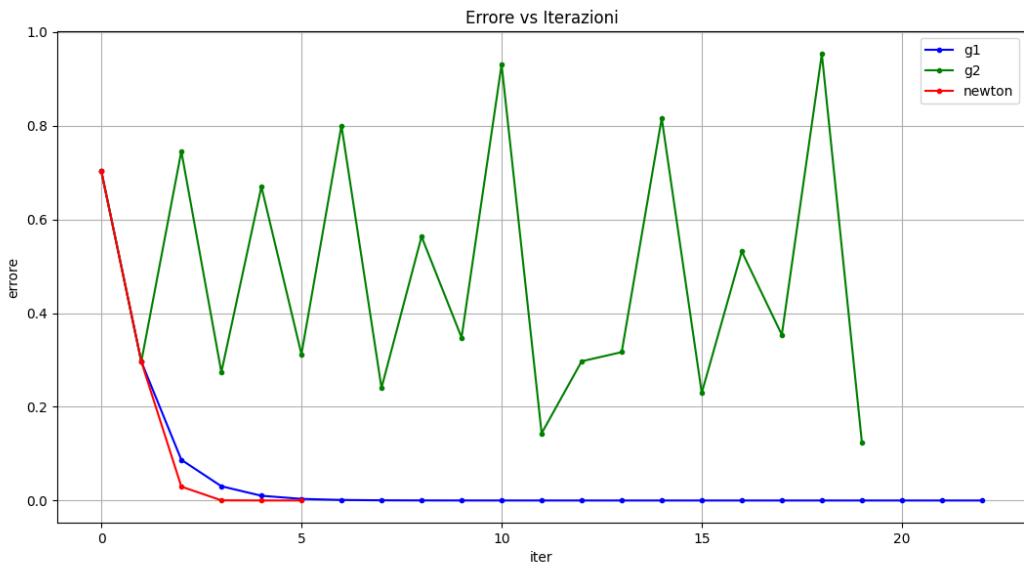


Figura 12: Ovviamente  $g_2$ , in figura limitato a 20 iterazioni, non converge

### 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  $x^* \approx 0.5369$

---

```

1 f = lambda x: x**3+4*x*np.cos(x)-2
2 df = lambda x: 3*x**2+4*np.cos(x)-4*x*np.sin(x)
3 g = lambda x: (2-x**3)/(4*np.cos(x))
4
5 xplot = np.linspace(0, 2)
6 fplot = f(xplot)

```

---

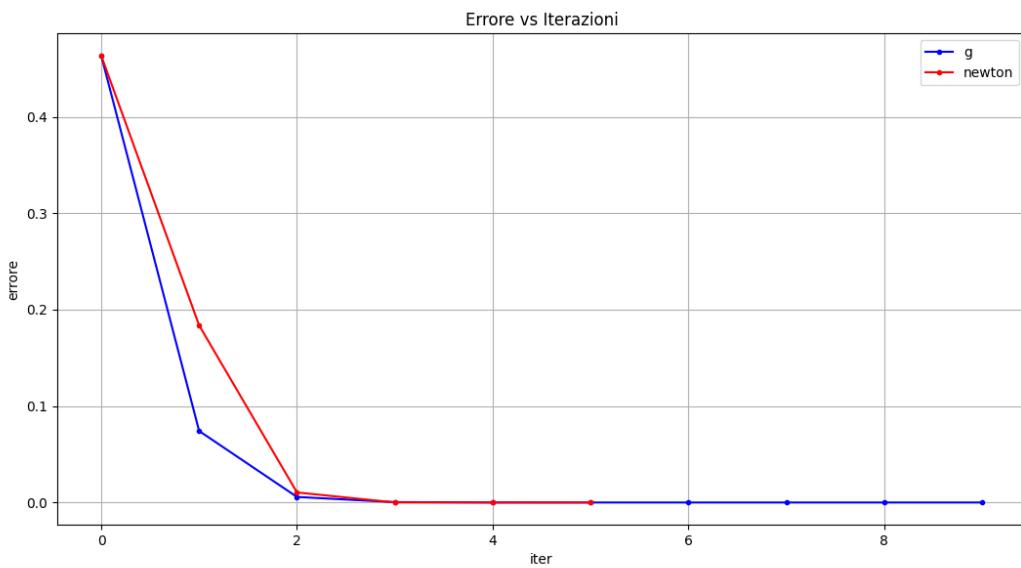


Figura 13: Sia Newton che  $g(x) = \frac{2-x^3}{4\cos(x)}$  convergono

---

```

1 Metodo approssimazioni successive g
2 x = 0.5368385515641152
3 iter_new= 10
4 Metodo Newton
5 x = 0.5368385515667755
6 iter_new= 6

```

---

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

---

```

1 f = lambda x: x-x**(1/3)-2
2 df = lambda x: 1-1/(3*x**(2/3))
3 g = lambda x: x***(1/3)+2
4
5 xplot = np.linspace(3, 5)
6 fplot = f(xplot)

```

---

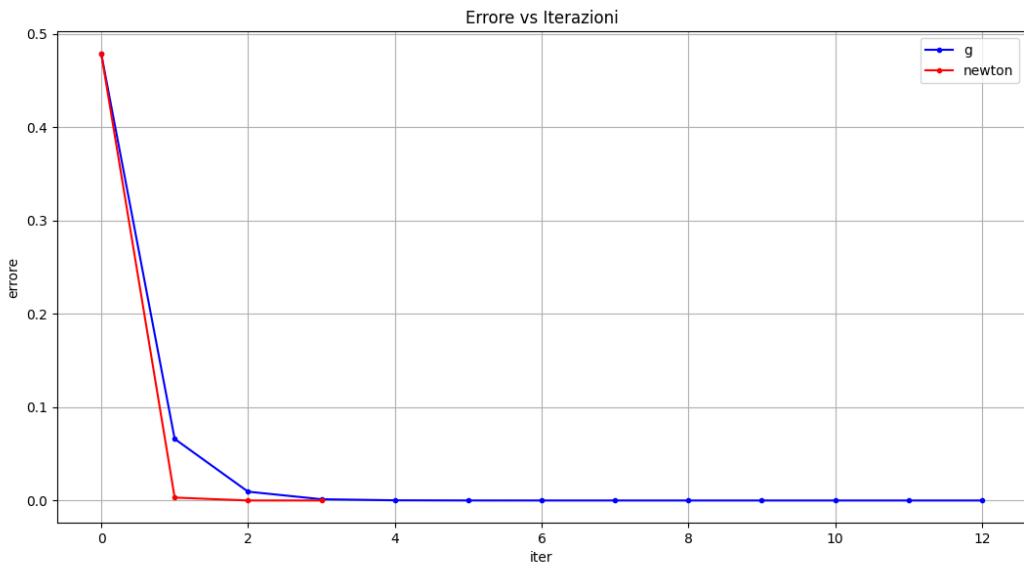


Figura 14: Newton converge più rapidamente di  $g(x) = \frac{2-x^3}{4\cos(x)}$

---

```

1 Metodo approssimazioni successive g
2 x = 3.521379706798214
3 iter_new= 13
4 Metodo Newton
5 x = 3.521379706804568
6 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  $\alpha_k$  variabile, calcolato secondo la procedura di backtracking ad ogni iterazione  $k$ -esima.

---

```

1 def next_step(x,grad): # backtracking procedure for the choice of the steplength
2     alpha=1.1
3     rho = 0.5
4     c1 = 0.25
5     p=-grad
6     j=0
7     jmax=10
8     while ((f(x+alpha*p) > f(x)+c1*alpha*np.dot(grad,p)) and j<jmax ):

```

---

```

9     alpha= rho*alpha
10    j+=1
11    if (j>jmax):
12        return -1
13    else:
14        print('alpha=',alpha)
15    return alpha
16
17
18 def minimize(f,grad_f,x0,step,maxit,tol,xTrue,fixed=True): # funzione che implementa il metodo del gra
19     #declare x_k and gradient_k vectors
20     # x_list only for logging
21     x_list=np.zeros((2,maxit+1))
22
23     norm_grad_list=np.zeros(maxit+1)
24     function_eval_list=np.zeros(maxit+1)
25     error_list=np.zeros(maxit+1)
26
27     #initialize first values
28     x_last = x0
29
30     x_list[:,0] = x_last
31
32     k=0
33
34     function_eval_list[k]=f(x_last)
35     error_list[k]=np.linalg.norm(x_last-xTrue)
36     norm_grad_list[k]=np.linalg.norm(grad_f(x_last))
37
38     while (np.linalg.norm(grad_f(x_last))>tol and k < maxit):
39         k=k+1
40         grad = grad_f(x_last)#direction is given by gradient of the last iteration
41
42
43         if fixed:
44             # Fixed step
45             step = step
46         else:
47             # backtracking step
48             step = next_step(x_last,grad)
49
50         if(step== -1):
51             print('non convergente')
52             return (k) #no convergence
53
54         x_last=x_last-step*grad
55
56         x_list[:,k] = x_last
57
58         function_eval_list[k]=f(x_last)
59         error_list[k]=np.linalg.norm(x_last-xTrue)
60         norm_grad_list[k]=np.linalg.norm(grad_f(x_last))
61
62     function_eval_list = function_eval_list[:k+1]

```

---

```

63     error_list = error_list[:k+1]
64     norm_grad_list = norm_grad_list[:k+1]
65
66     print('iterations=' ,k)
67     print('last guess: x=(%f,%f)'%(x_list[0,k],x_list[1,k]))
68
69     return (x_last,norm_grad_list, function_eval_list, error_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  $[-1.5, 3.5] \times [-1, 5]$ .

---

```

1  def f(vec):
2      x, y = vec
3      fout = 3*(x-2)**2 + (y-1)**2
4      return fout
5
6  def grad_f(vec):
7      x, y = vec
8      dfdx = 6*(x-2)
9      dfdy = 2*(y-1)
10     return np.array([dfdx,dfdy])
11
12 x = np.linspace(-1.5,3.5)
13 y = np.linspace(-1,5)
14
15 X, Y = np.meshgrid(x, y)
16 vec = np.array([X,Y])
17 Z=f(vec)
18
19 fig = plt.figure(figsize=(15, 8))
20
21 ax = plt.axes(projection='3d')
22 ax.set_title('$f(x)=(x-1)^2 + (y-2)^2$')
23 #ax.view_init(elev=50., azim=30)
24 s = ax.plot_surface(X, Y, Z, cmap='viridis')
25 fig.colorbar(s)
26 plt.show()

```

---

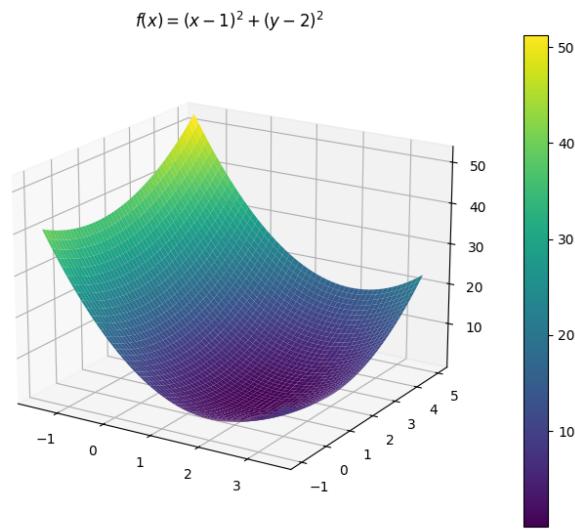


Figura 15: Superficie di  $f(x, y)$

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

---

```

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

```

---

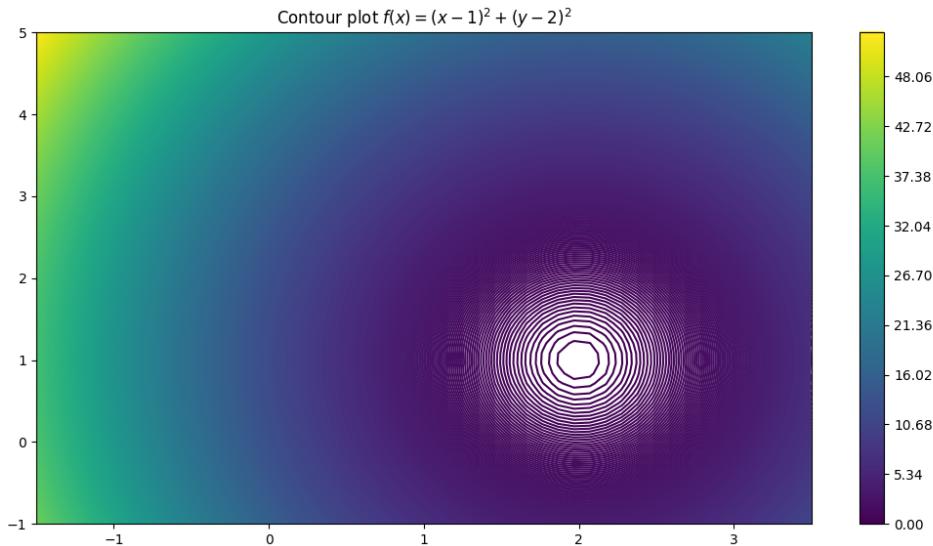


Figura 16: Curve di livello di  $f(x, y)$  a livello 1000

- 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  $(3, 5)$ .

---

```

1 step = 0.001
2 maxitS=1000
3 tol=1.e-5
4 x0 = np.array([-0.5,1])
5 xTrue = np.array([1,1])
6 (x_last,norm_grad_listf, function_eval_listf, error_listf, xlist, k)= minimize(f,grad_f,x0,step, m
7 plt.plot(function_eval_listf,'*-')
8 (x_last,norm_grad_list, function_eval_list, error_list, xlist, k)= minimize(f,grad_f,x0,step,maxi
9 plt.plot(function_eval_list,'*-')
10 plt.legend(['fixed', 'backtracking'])
11 plt.show()

```

---

```

1 last guess: x=(2.000001,1.000003)

```

---

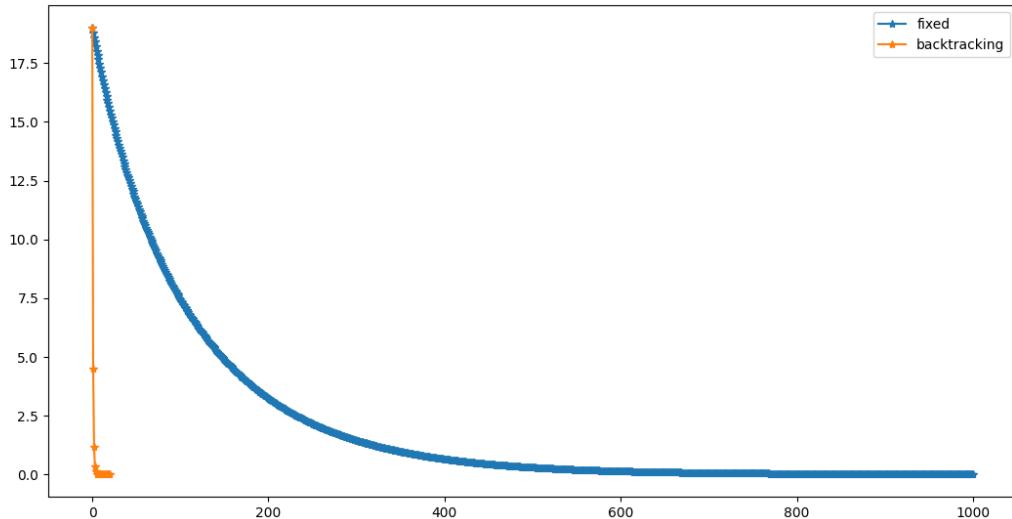


Figura 17: Discesa del gradiente con  $\alpha$  fisso e  $\alpha_k$  da backtracking

- Si analizzino i risultati in entrambi i casi.

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

---

```

1 fig, (ax1, ax2, ax3) = plt.subplots(1, 3)
2 ax1.semilogy(norm_grad_listf)
3 ax1.semilogy(norm_grad_list)
4 ax1.set_title('$\|\nabla f(x_k)\|$')
5 ax2.semilogy(function_eval_listf)
6 ax2.semilogy(function_eval_list)

```

---

```

7  ax2.set_title('f(x_k)')
8  ax3.semilogy(error_listf)
9  ax3.semilogy(error_list)
10 ax3.set_title('||x_k-x^*||')
11 fig.tight_layout()
12 fig.legend(['fixed', 'backtracking'], loc='lower center', ncol=4)
13 plt.show()

```

---

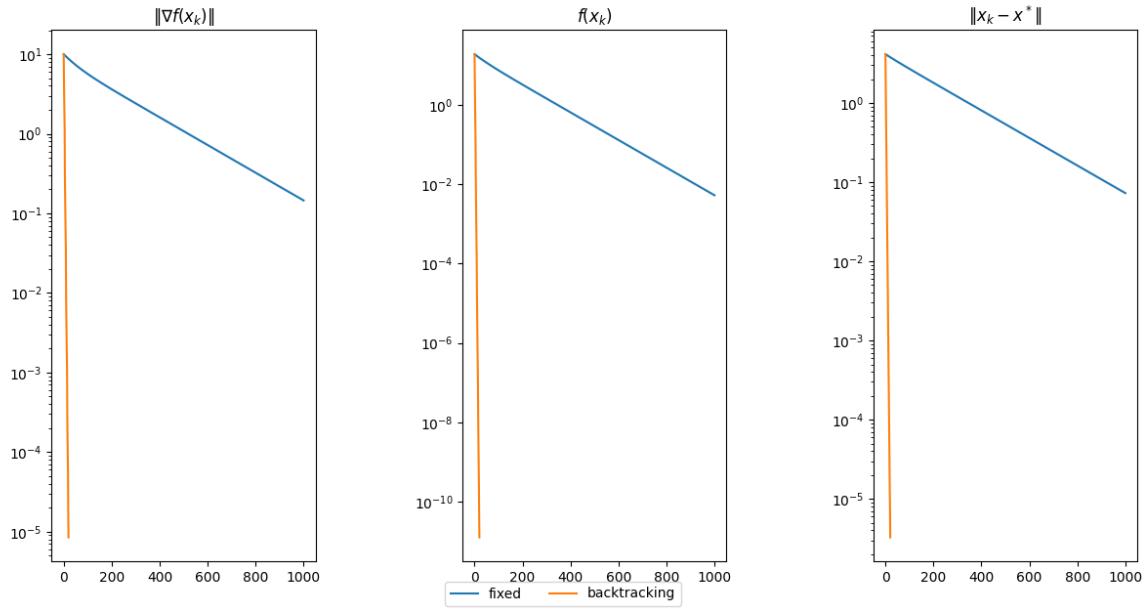


Figura 18: Confronto tra lunghezza del passo  $\alpha$  fissa e  $\alpha_k$  da backtracking

### 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 eseguono le richieste dell'esercizio precedente nel dominio  $[-2, 2] \times [-1, 3]$ , usando come punto iniziale  $(-0.5, 1)$ .

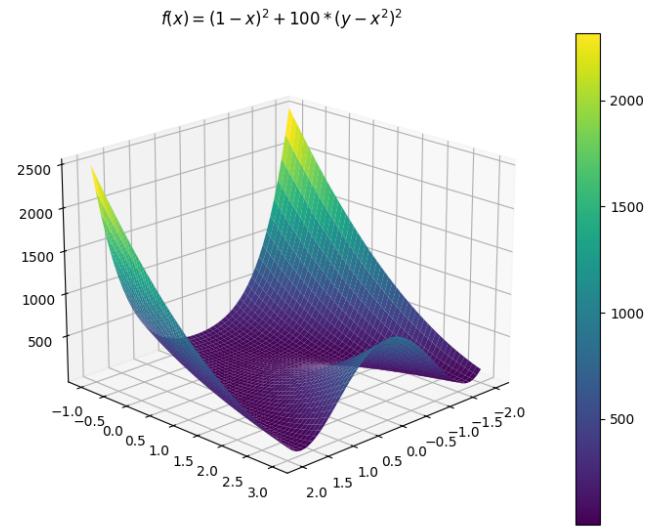


Figura 19: Superficie di  $f(x) = (1 - x)^2 + 100 * (y - x^2)^2$

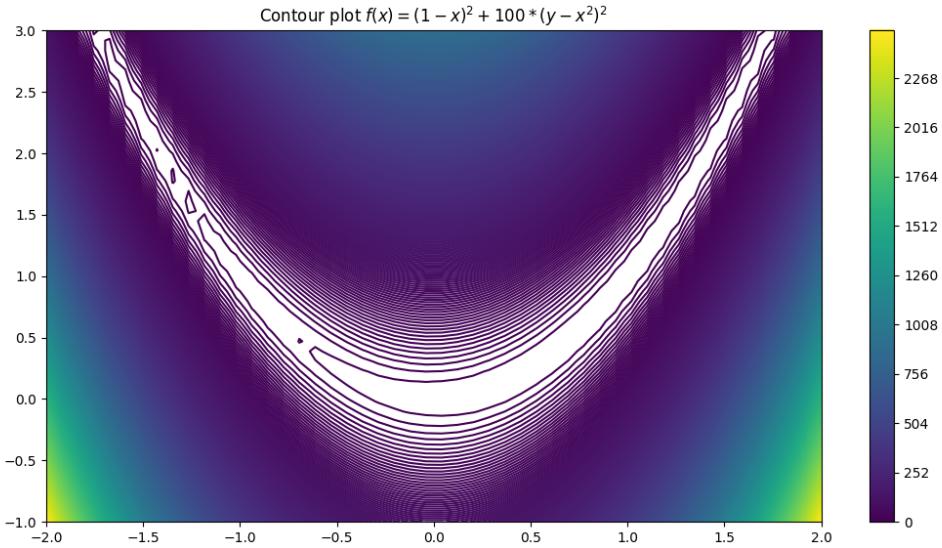


Figura 20: Curve di livello di  $f(x, y)$  a livello 1000

In questo caso, sia la scelta di mantenere fissa la lunghezza del passo  $\alpha$  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  $\alpha$  fisso si ottiene:

---

1 last guess: x=(0.482604, 0.230313)

---

in tal punto, la funzione vale 0.26837130770971884.

- Mentre con l'algoritmo di backtracking si ottiene:

---

```
1 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  $\alpha$  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  $\sigma$ .

### Esercizio 1.1 Problema test

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

---

```
1 import cv2 as cv
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from skimage import data, metrics
5 from scipy import signal
6 from numpy import fft
7 from utils import psf_fft, A, AT, gaussian_kernel
8
9 # Immagine in floating point con valori tra 0 e 1
10 X = data.camera() / 255
11 m, n = X.shape
```

---

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

---

```
1 # Genera il filtro di blur
2 k = gaussian_kernel(24, 3)
3 plt.imshow(k)
4 plt.show()
5
6 # Blur with openCV
7 X_blurred = cv.filter2D(X, -1, k)
```

---

```

8 plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
9 plt.title('Original')
10 plt.xticks([]), plt.yticks([])
11 plt.subplot(122).imshow(X_blurred, cmap='gray', vmin=0, vmax=1)
12 plt.title('Blurred')
13 plt.xticks([]), plt.yticks([])
14 plt.show()
15
16 # Blur with FFT
17 K = psf_fft(k, 24, X.shape)
18 plt.imshow(np.abs(K))
19 plt.show()

```

---

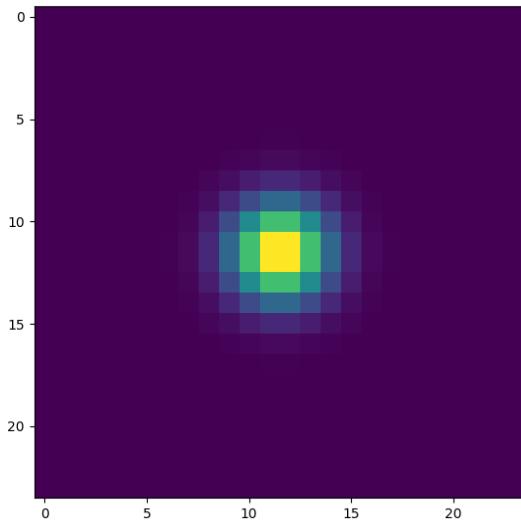


Figura 21: Kernel Gaussiano di dimensione  $24 \times 24$  con deviazione standard 3



Figura 22: Blur con cv2

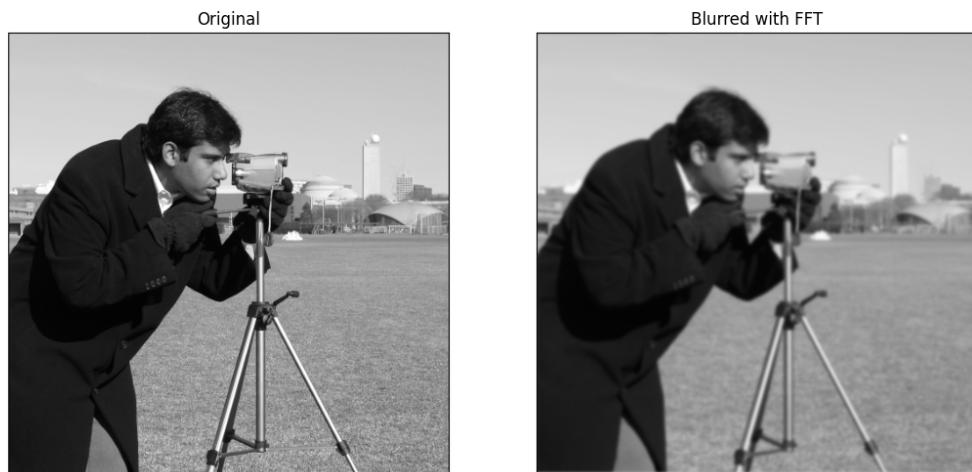


Figura 23: Blur con trasformata di Fourier (FFT)

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

---

```

1 # Genera il rumore
2 sigma = 0.02
3 np.random.seed(42)
4 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`.

---

```

1 # Aggiungi blur e rumore
2 y = X + noise
3 PSNR = metrics.peak_signal_noise_ratio(X, y)
4 ATy = AT(y, K)
5
6 # Visualizziamo i risultati
7 plt.figure(figsize=(30, 10))
8 plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
9 plt.title('Original')
10 plt.xticks([]), plt.yticks([])
11 plt.subplot(122).imshow(y, cmap='gray', vmin=0, vmax=1)
12 plt.title(f'Corrupted (PSNR: {PSNR:.2f})')
13 plt.xticks([]), plt.yticks([])
14 plt.show()

```

---

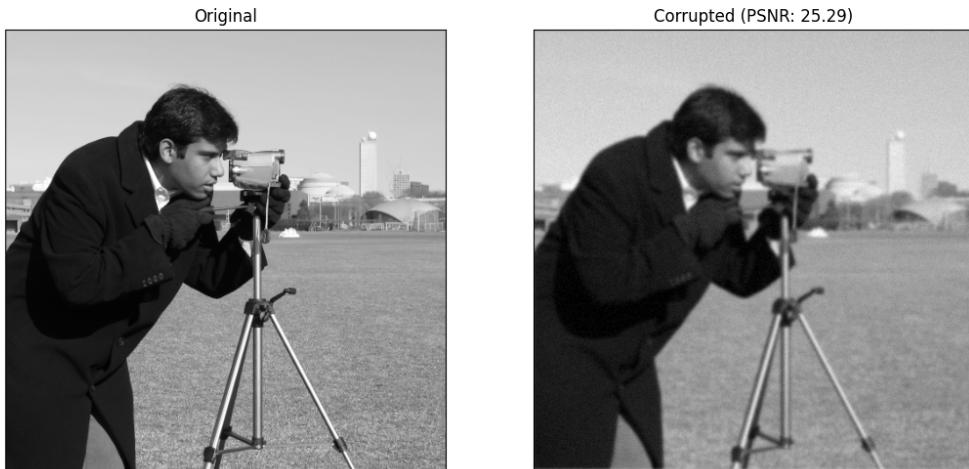


Figura 24: Confronto tra immagine originale e degradata dal rumore

### Esercizio 1.2 Soluzione naive

**Soluzione naive** 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.

---

```

1 # Soluzione naïve
2 from scipy.optimize import minimize
3
4 # Funzione da minimizzare
5 def f(x):
6     x = x.reshape((m, n))
7     Ax = A(x, K)
8     return 0.5 * np.sum(np.square(Ax - y))
9
10 # Gradiente della funzione da minimizzare
11 def df(x):
12     x = x.reshape((m, n))
13     ATAx = AT(A(x,K),K)
14     d = ATAx - ATy
15     return d.reshape(m * n)
16
17 # Minimizzazione della funzione
18 x0 = y.reshape(m*n)
19 max_iter = 25
20 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.

---

```

1 # Per ogni iterazione calcola il PSNR rispetto all'originale
2 PSNR = np.zeros(max_iter + 1)
3 for k, x_k in enumerate(res.allvecs):
4     PSNR[k] = metrics.peak_signal_noise_ratio(X, x_k.reshape(X.shape))
5
6 # Risultato della minimizzazione
7 X_res = res.x.reshape((m, n))
8
9 # PSNR dell'immagine corrotta rispetto all'oginale
10 starting_PSNR = np.full(PSNR.shape[0], metrics.peak_signal_noise_ratio(X, y))
11
12 # Visualizziamo i risultati
13 ax2 = plt.subplot(1, 2, 1)
14 ax2.plot(PSNR, label="Soluzione naïve")
15 ax2.plot(starting_PSNR, label="Immagine corrotta")
16 plt.legend()
17 plt.title('PSNR per iterazione')
18 plt.ylabel("PSNR")
19 plt.xlabel('itr')
20 plt.subplot(1, 2, 2).imshow(X_res, cmap='gray', vmin=0, vmax=1)
21 plt.title('Immagine Ricostruita')
22 plt.xticks([]), plt.yticks([])
23 plt.show()

```

---

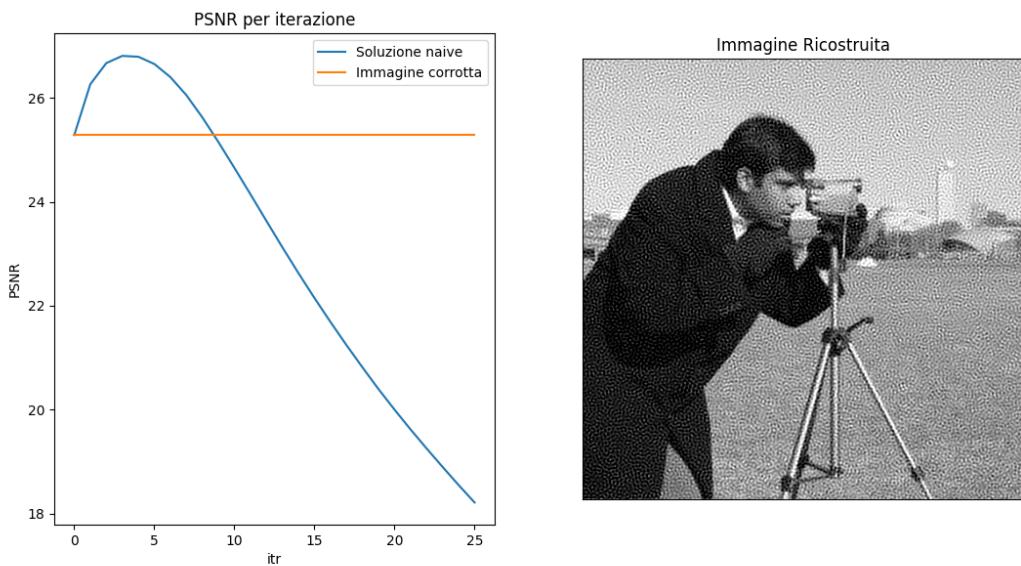


Figura 25: Notiamo come l'immagine presenti un effetto sgradevole dovuto alla mancanza di regolarizzazione

### 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.

---

```

1 def f(x, L):
2     nsq = np.sum(np.square(x))
3     x = x.reshape(m, n)
4     Ax = A(x, K)
5     return 0.5 * np.sum(np.square(Ax - y)) + 0.5 * L * nsq
6
7 # Gradiente della funzione da minimizzare
8 def df(x, L):
9     Lx = L * x
10    x = x.reshape(m, n)
11    ATAx = AT(A(x,K),K)
12    d = ATAx - ATy
13    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.

---

```

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

```

---

```

5
6 # Ricostruzione per diversi valori del parametro di regolarizzazione
7 for i, L in enumerate(lambdas):
8     # Esegui la minimizzazione con al massimo 50 iterazioni
9     max_iter = 50
10    res = minimize(f, x0, (L), method='CG', jac=df, options={'maxiter':max_iter})
11
12    # Aggiungi la ricostruzione nella lista images
13    X_curr = res.x.reshape(X.shape)
14    images.append(X_curr)
15
16    # Stampa il PSNR per il valore di lambda attuale
17    PSNR = metrics.peak_signal_noise_ratio(X, X_curr)
18    PSNRs.append(PSNR)
19    print(f'PSNR: {PSNR:.2f} (\u03bb = {L:.2f})')
20
21 # Visualizziamo i risultati
22 plt.plot(lambdas,PSNRs)
23 plt.title('PSNR per $\lambda$')
24 plt.ylabel("PSNR")
25 plt.xlabel('$\lambda$')
26 plt.show()

```

---

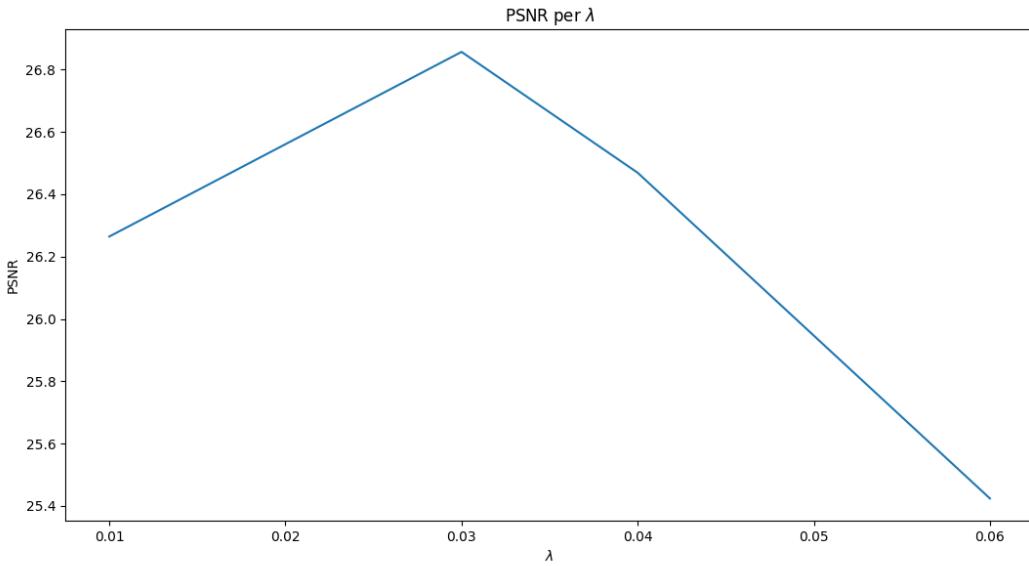


Figura 26: PSNR rispetto al parametro di regolarizzazione  $\lambda$

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

---

```

1 plt.figure(figsize=(30, 10))
2
3 (nrows, ncols) = ((len(lambdas) + 2) // 3, (len(lambdas) + 2) // 2)
4

```

```

5 plt.subplot(nrows, ncols, 1).imshow(X, cmap='gray', vmin=0, vmax=1)
6 plt.title("Originale")
7 plt.xticks([]), plt.yticks([])
8 plt.subplot(nrows, ncols, 2).imshow(y, cmap='gray', vmin=0, vmax=1)
9 plt.title("Corrotta")
10 plt.xticks([]), plt.yticks([])
11
12
13 for i, L in enumerate(lambdas):
14     plt.subplot(nrows, ncols, i + 3).imshow(images[i], cmap='gray', vmin=0, vmax=1)
15     plt.title(f"Ricostruzione ($\lambda$ = {L:.2f})")
16 plt.show()

```

---

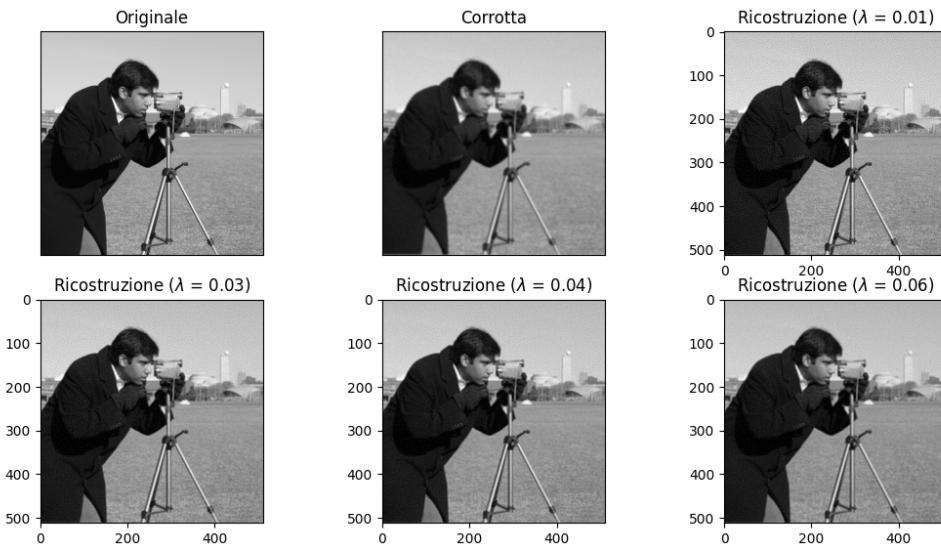


Figura 27: Ricostruzione dell’immagine al variare di  $\lambda$

- Scegliere  $\lambda$  con il metodo di discrepanza.

```

1 # ||A_L - y|| <= tau * ||noise||
2 tau = 1.01
3 L = tau * np.linalg.norm(noise) / np.linalg.norm(ATy)
4 print('L', L) # 0.034985654993768586
5
6 max_iter = 50
7 res = minimize(f, x0, (L), method='CG', jac=df, options={'maxiter':max_iter})
8 X_curr = res.x.reshape(X.shape)
9
10 PSNR = metrics.peak_signal_noise_ratio(X, X_curr)
11 print('PSNR', PSNR) # 26.68424503233115
12
13 plt.subplot(121).imshow(X_blurred, cmap='gray', vmin=0, vmax=1)
14 plt.title('Corrotta')
15 plt.xticks([]), plt.yticks([])

```

```

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

```

---

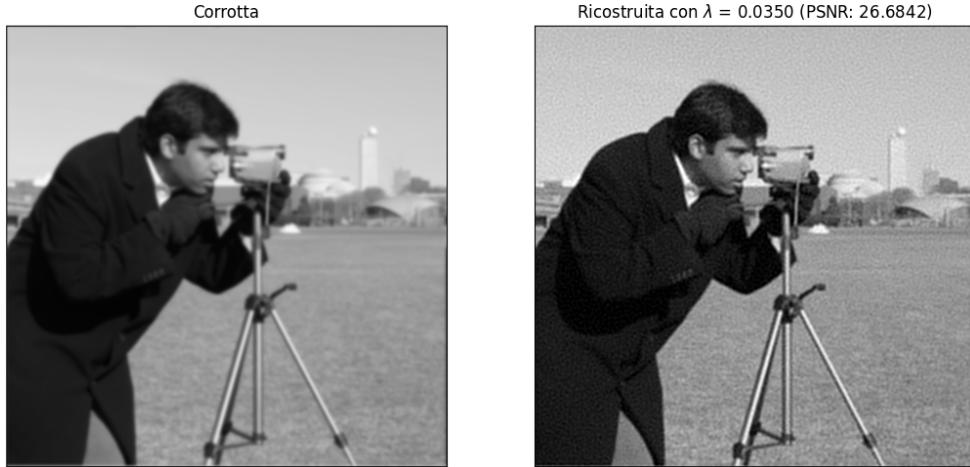


Figura 28: Con  $\lambda$  secondo il principio della discrepanza di Morozov si arriva a un buon PSNR

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

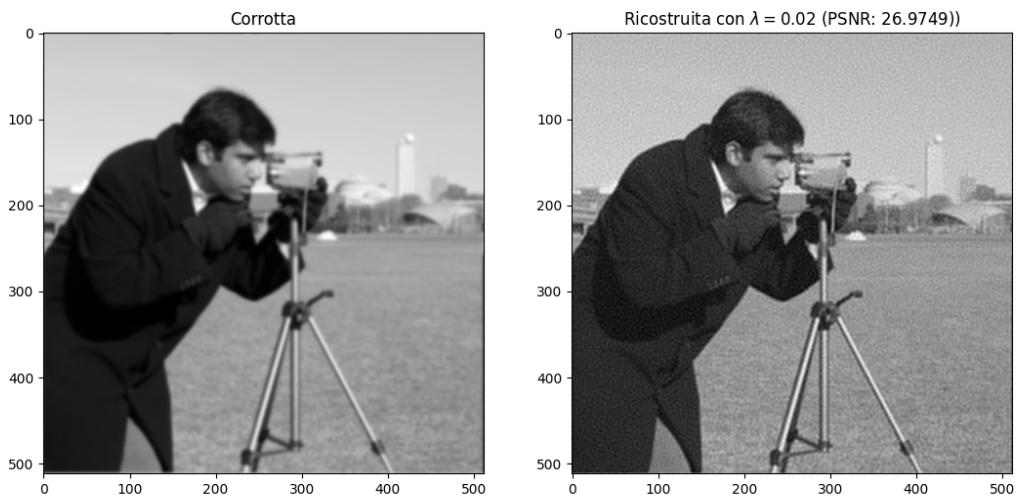


Figura 29: Con  $\lambda = 0.02$  otteniamo ancora un miglior PSNR

## Esercizio 1.4

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

---

```

1 S=2
2 X_d = X[::S,::S]
3 y_d = y[::S,::S]
4 K_d = K[::S,::S]
5 ATy_d = AT(y_d, K_d)
6
7 # Funzione da minimizzare
8 def f(x, L):
9     nsq = np.sum(np.square(x))
10    x = x.reshape((m//S, n//S))
11    Ax = A(x, K_d)
12    return 0.5 * np.sum(np.square(Ax - y_d)) + 0.5 * L * nsq
13
14 # Gradiente della funzione da minimizzare
15 def df(x, L):
16     Lx = L * x
17     x = x.reshape(m//S, n//S)
18     ATAx = AT(A(x,K_d),K_d)
19     d = ATAx - ATy_d
20     return d.reshape(m//S * n//S) + Lx
21
22 x0 = y_d.reshape(m//S*n//S)
23 lambdas = [0.03, 0.04, 0.06, 0.08]

```

---

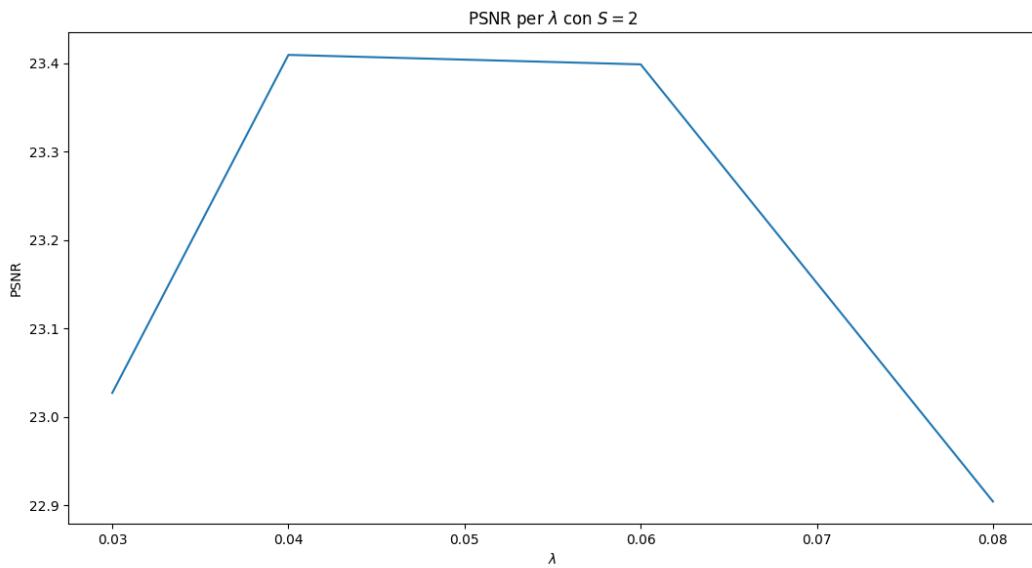


Figura 30: PSNR su  $\lambda$  con downsampling  $S = 2$

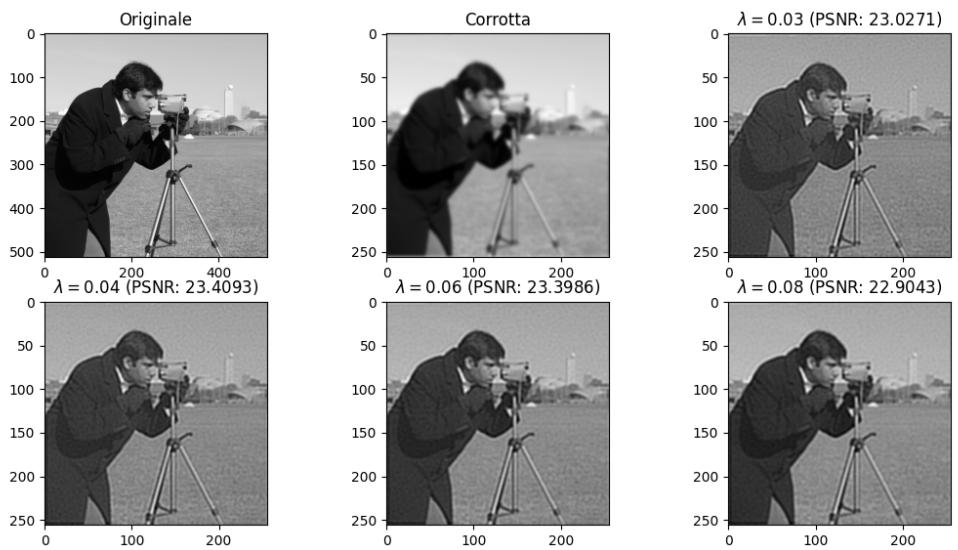


Figura 31: Ricostruzione al variare di  $\lambda$  con  $S = 2$

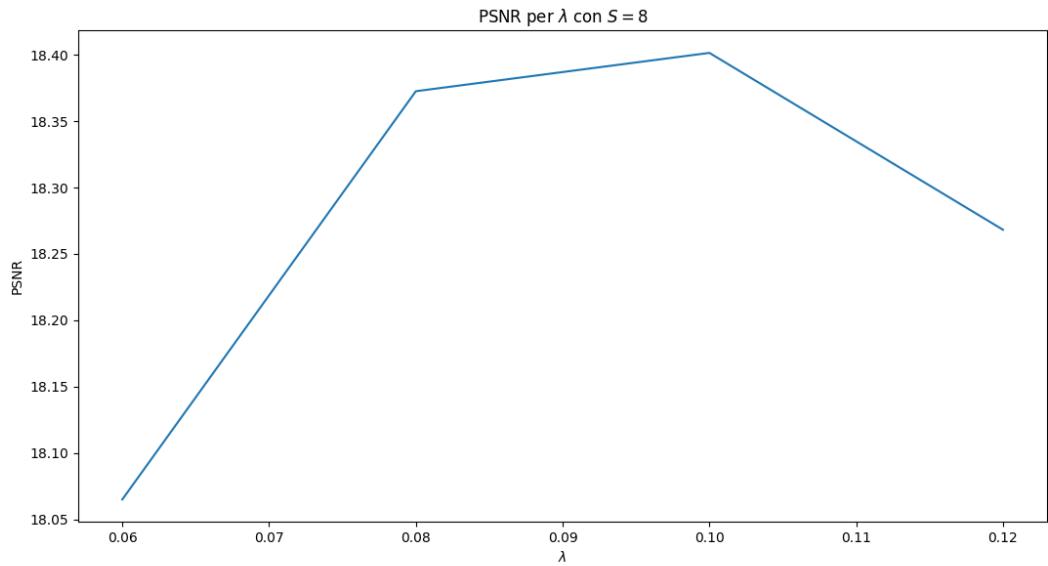


Figura 32: PSNR su  $\lambda$  con downsampling  $S = 8$

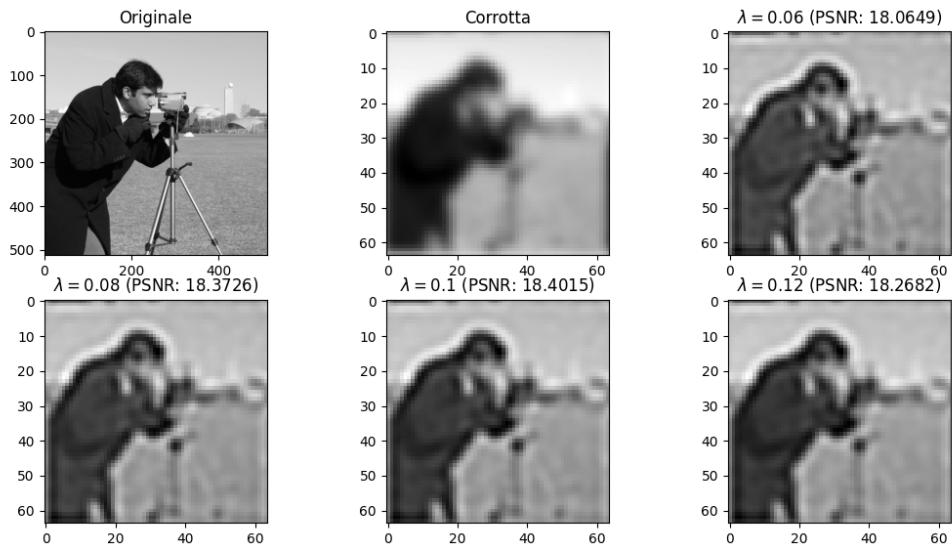


Figura 33: Ricostruzione al variare di  $\lambda$  con  $S = 8$

- 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).

---

```

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

```

---

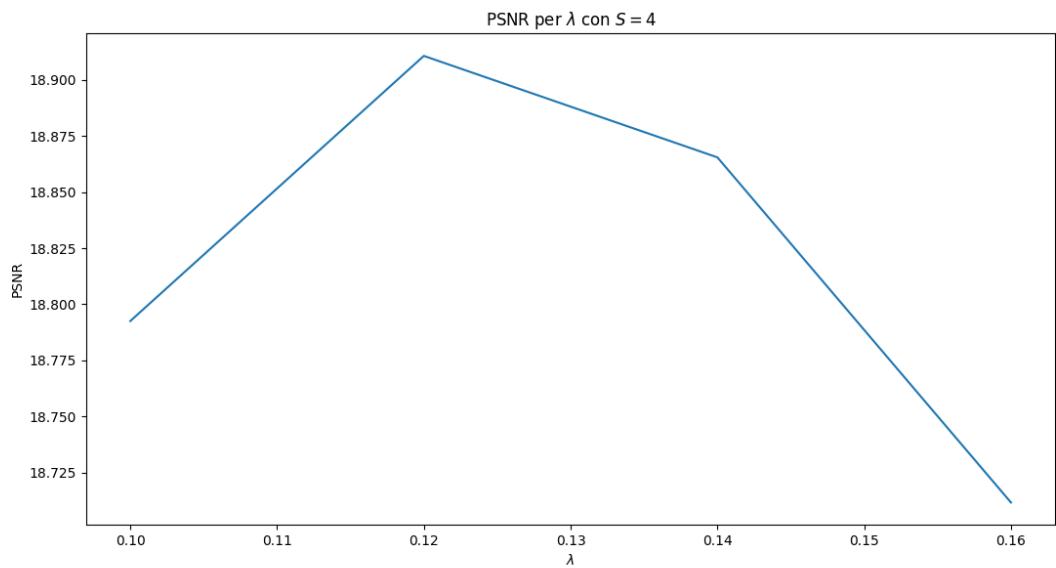


Figura 34: Sabbia su microscopio: PSNR su  $\lambda$  con downsampling  $S = 4$

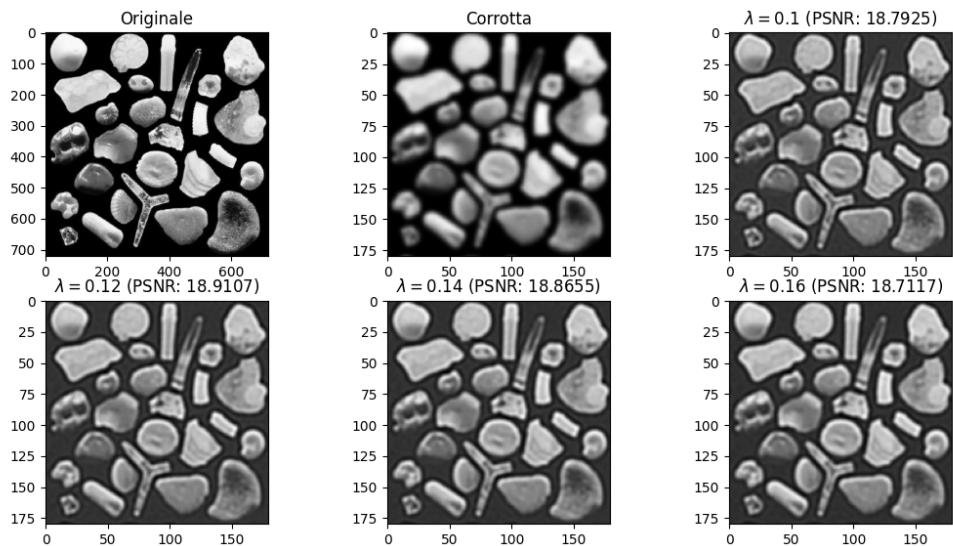


Figura 35: Ricostruzione al variare di  $\lambda$  con  $S = 4$

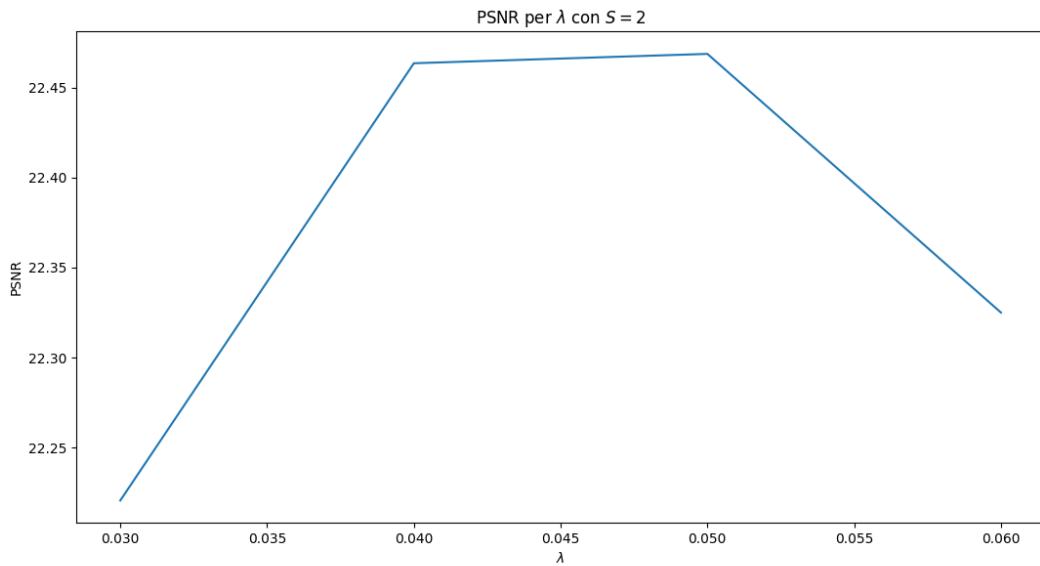


Figura 36: Modena: PSNR su  $\lambda$  con downsampling  $S = 2$

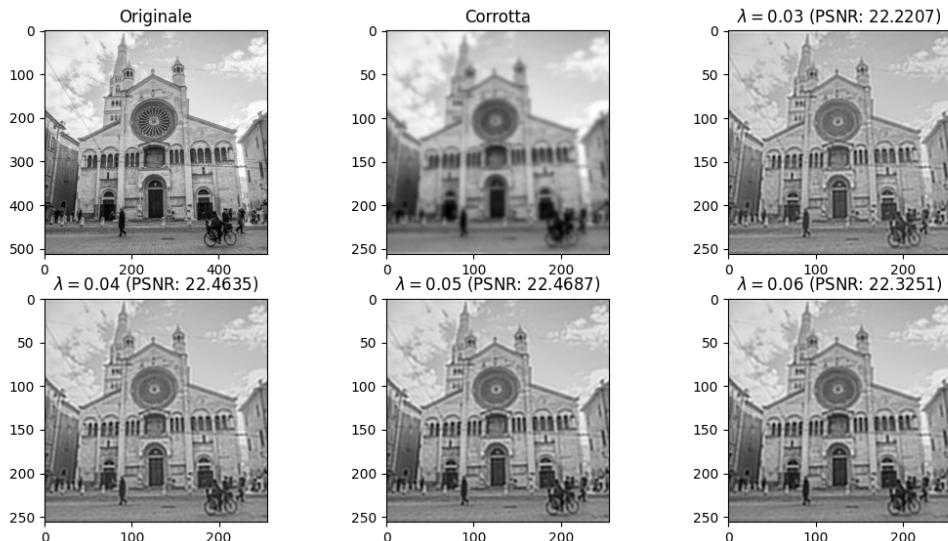


Figura 37: Ricostruzione al variare di  $\lambda$  con  $S = 2$

- Degradare le nuove immagini applicando, mediante le funzioni `gaussian_kernel()`, `psf_fft()`, l’operatore di blur con parametri:
  - $\sigma = 0.5$ , dimensione del kernel  $7 \times 7$  e  $9 \times 9$ .

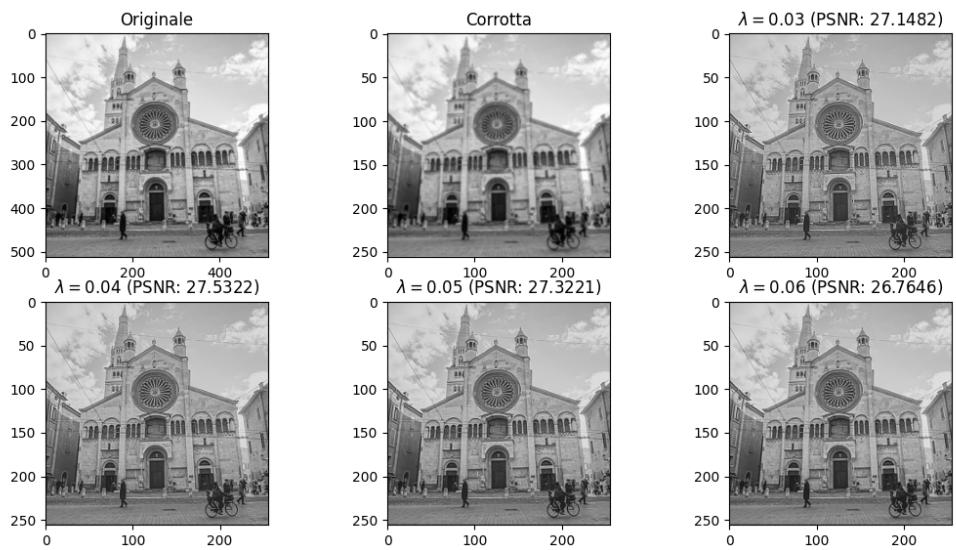


Figura 38: Modena con  $\sigma = 0.5$ , dimensione del kernel  $7 \times 7$ ,  $S = 2$

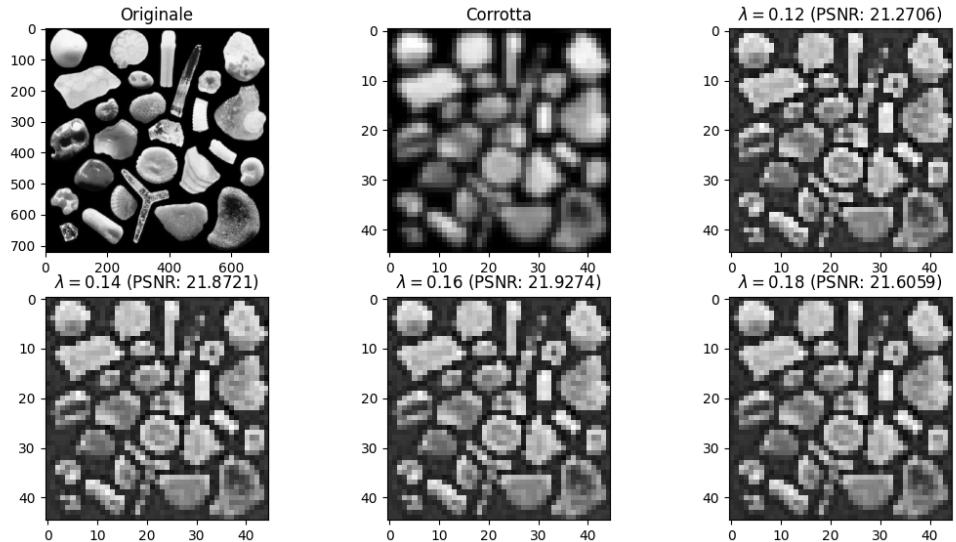


Figura 39: Sabbia con  $\sigma = 0.5$ , dimensione del kernel  $9 \times 9$ ,  $S = 16$

–  $\sigma = 1.3$ , dimensione del kernel  $5 \times 5$ .

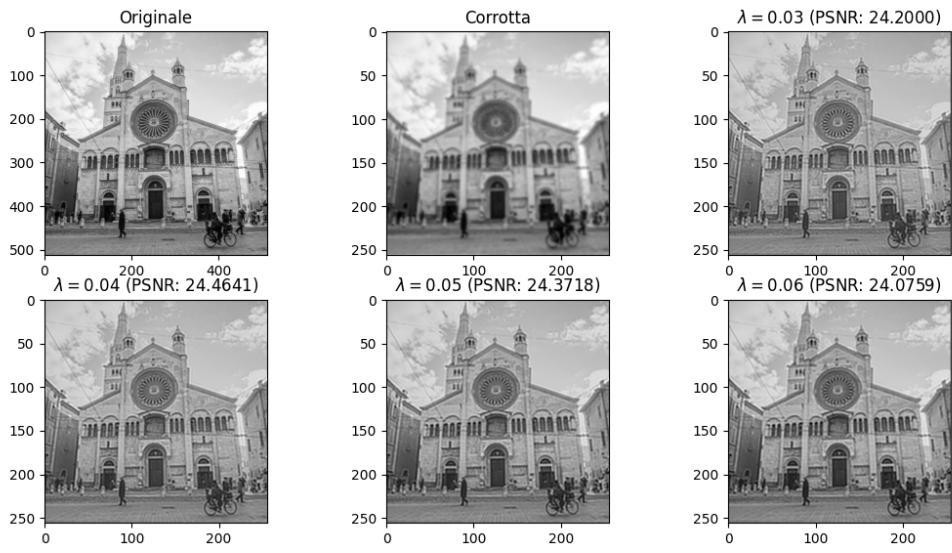


Figura 40: Modena con  $\sigma = 1.3$ , dimensione del kernel  $5 \times 5$ ,  $S = 2$

- Aggiungere rumore gaussiano con deviazione standard nell’intervallo  $(0, 0.05]$ .

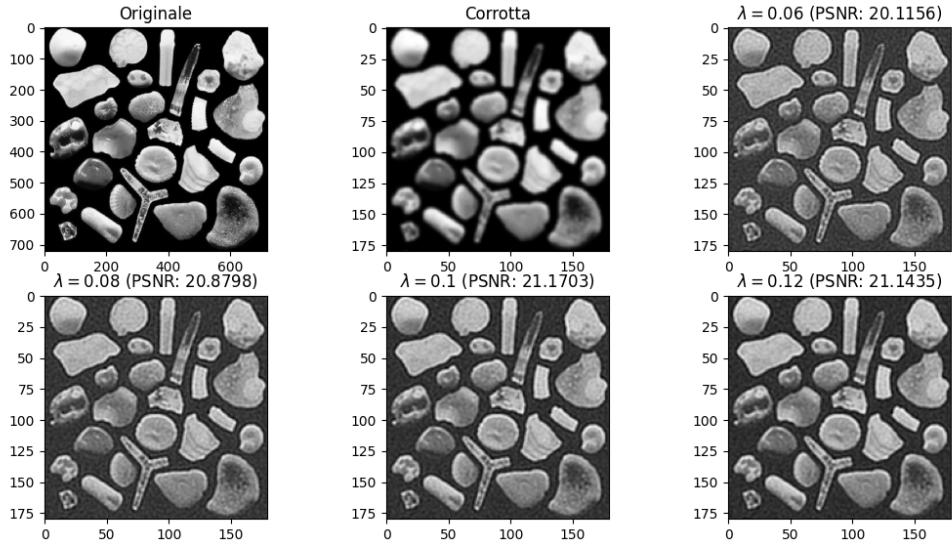


Figura 41: Sabbia con  $\sigma = 1.3$ , dimensione del kernel  $5 \times 5$ ,  $S = 4$  e rumore gaussiano con deviazione standard nell’intervallo  $(0, 0.05]$

## 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  $\rho$  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:  $O\left(\frac{2n^3}{3}\right)$

---

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

---

Per il codice, si veda l'Esercizio 2.1 dell'Esercitazione 2.

## Domanda 2: Fattorizzazione con Cholesky

Considero la matrice  $A$  come una matrice tridiagonale simmetrica definita positiva, avente sulla diagonale principale elementi uguali a 9 ed elementi nella sopra e sottodiagonale uguali a -4, con  $n$  variabile.

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

Posso fattorizzare  $A$  con Cholesky ottenendo:

$$A = LL^T$$

---

```
1 import numpy as np
2 from scipy.linalg import cholesky
3
4 n = 5
5 A = np.diag(9 * np.ones(n)) + np.diag(-4 * np.ones(n-1), 1) + np.diag(-4 * np.ones(n-1), -1)
6 b = np.ones(n)
7
8 K_A = np.linalg.cond(A)
```

```

9 L = cholesky(A, lower=True)
10 x = np.linalg.solve(L, b)

```

---

```

1 A
2 [[ 9. -4.  0.  0.  0.]
3 [-4.  9. -4.  0.  0.]
4 [ 0. -4.  9. -4.  0.]
5 [ 0.  0. -4.  9. -4.]
6 [ 0.  0.  0. -4.  9.]]
7
8 K_A = 7.6881108528775615
9
10 L
11 [[ 3.          0.          0.          0.          0.        ]
12 [-1.33333333  2.68741925  0.          0.          0.        ]
13 [ 0.         -1.48841682  2.60472943  0.          0.        ]
14 [ 0.          0.         -1.53566814  2.57715412  0.        ]
15 [ 0.          0.          0.         -1.55209965  2.5672917 ]]
16
17 x = [0.33333333 0.53748385 0.69105066 0.79980645 0.87305206]

```

---

Possiamo risolvere il sistema tramite sostituzioni successive.

### Domanda 3: Cholesky con matrice di Hilbert

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

$$A = \begin{bmatrix} 1/1 & 1/2 & 1/3 & 1/4 & 1/5 \\ 1/2 & 1/3 & 1/4 & 1/5 & 1/6 \\ 1/3 & 1/4 & 1/5 & 1/6 & 1/7 \\ 1/4 & 1/5 & 1/6 & 1/7 & 1/8 \\ 1/5 & 1/6 & 1/7 & 1/8 & 1/9 \end{bmatrix}$$

```

1 import numpy as np
2 from scipy.linalg import cholesky
3 from scipy.linalg import hilbert
4
5 n = 5
6 A = hilbert(n)
7
8 b = np.ones(n)
9
10 K_A = np.linalg.cond(A)
11
12 L = cholesky(A, lower=True)
13 x = np.linalg.solve(L, b)

```

---

```

1 A
2 [[1.          0.5          0.33333333  0.25          0.2        ]
3 [0.5          0.33333333  0.25          0.2          0.16666667]
4 [0.33333333  0.25          0.2          0.16666667  0.14285714]
5 [0.25          0.2          0.16666667  0.14285714  0.125        ]]

```

---

```

6 [0.2      0.16666667 0.14285714 0.125      0.11111111]
7
8 K_A = 476607.2502425855
9
10 L
11 [[1.      0.      0.      0.      0.      ]]
12 [0.5      0.28867513 0.      0.      0.      ]
13 [0.33333333 0.28867513 0.0745356 0.      0.      ]
14 [0.25     0.25980762 0.1118034 0.01889822 0.      ]
15 [0.2      0.23094011 0.12777531 0.03779645 0.0047619 ]]
16
17 x = [1.      1.73205081 2.23606798 2.64575131 3.      ]

```

---

Osservo che la matrice  $A$  è notevolmente mal condizionata.

## 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:

- $\sigma_i$  rappresenta il  $i$ -esimo valore singolare nella matrice  $\Sigma$ . 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.

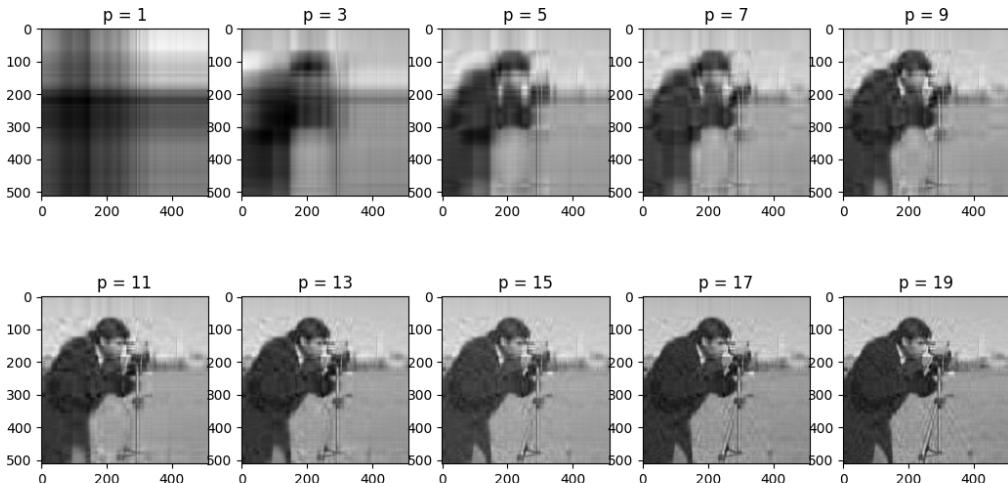


Figura 42: Compressione con SVD rispetto a  $p$

---

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from skimage import data
4
5 A = data.camera()
6 U, s, Vh = np.linalg.svd(A)
7 p_max = 20
8 plt.figure(figsize=(p_max, p_max // 2))
9 for i in range(1, p_max + 1, 2):
10     A_p = np.dot(U[:, :i], np.dot(np.diag(s[:i]), Vh[:i, :]))
11
12     plt.subplot(2, p_max // 4, (i + 1) // 2)
13     plt.imshow(A_p, cmap='gray')
14     plt.title('p = {}'.format(i))
15
16 plt.show()

```

---

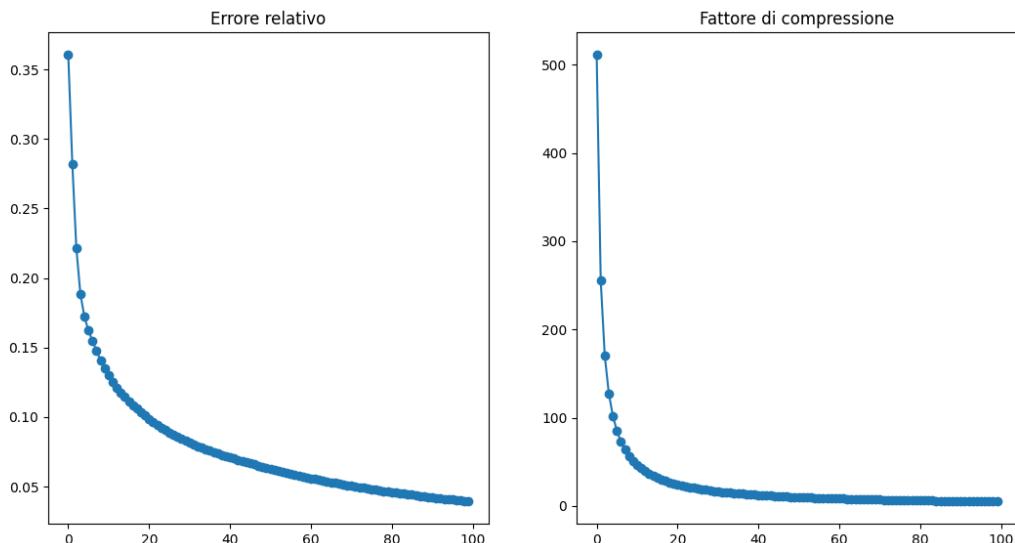


Figura 43: Errore relativo e compressione al variare di  $p$

---

```

1 A = data.camera()
2 U, s, Vh = np.linalg.svd(A)
3 p_max = 100
4 A_p = np.zeros(A.shape)
5 err_rel = np.zeros((p_max))
6 c = np.zeros((p_max))
7 for i in range(p_max):
8     ui = U[:, :i+1]
9     vi = Vh[:i+1, :]
10
11    A_p = np.dot(ui, np.dot(np.diag(s[:i+1]), vi))

```

---

```

12
13     err_rel[i] = np.linalg.norm(A - A_p) / np.linalg.norm(A)
14     c[i] = 1 / (i + 1) * (min(A.shape) - 1)
15
16 plt.figure(figsize=(10, 5))
17
18 fig1 = plt.subplot(1, 2, 1)
19 fig1.plot(err_rel, 'o-')
20 plt.title('Errore relativo')
21
22 fig2 = plt.subplot(1, 2, 2)
23 fig2.plot(c, 'o-')
24 plt.title('Fattore di compressione')
25
26 plt.show()

```

---

## 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:

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

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:

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

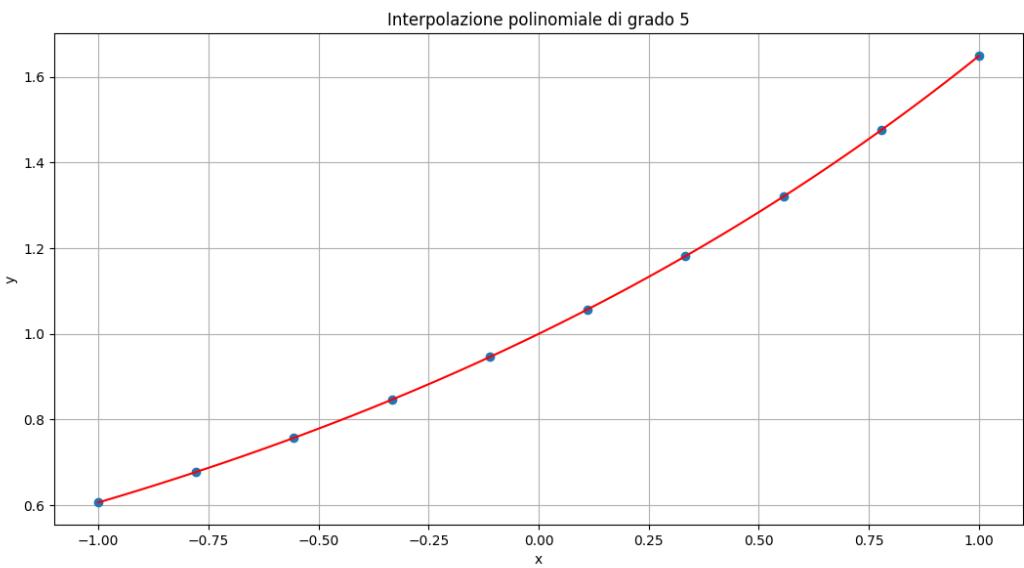


Figura 44: Polinomio interpolatore di  $f(x) = \exp\left(\frac{x}{2}\right)$

Per il codice, si veda l'Esercizio 1.2 dell'Esercitazione 4.  
Osservo che con grado 5 ottengo un'ottima interpolazione polinomiale.

## Domanda 6: Interpolazione polinomiale

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

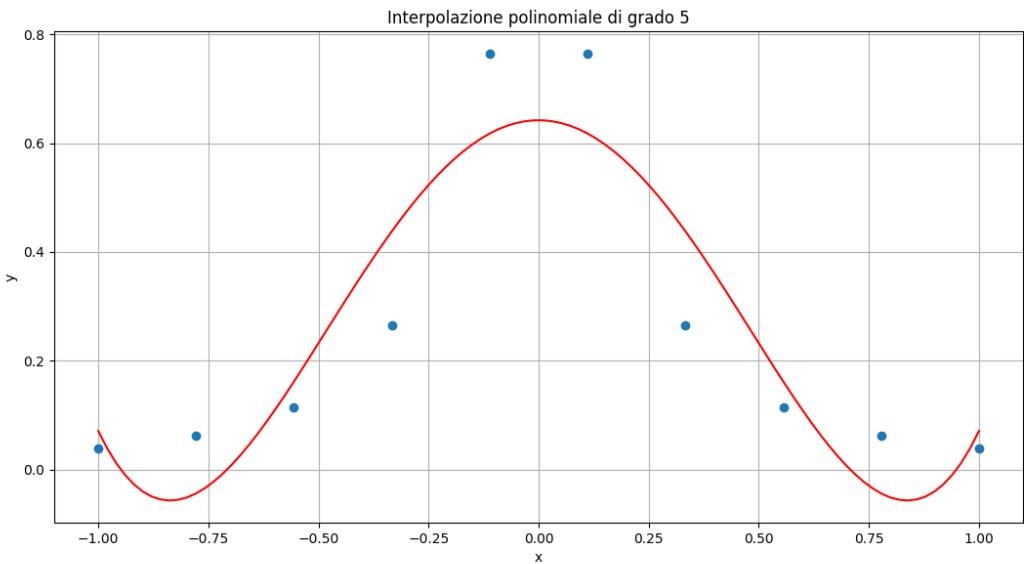


Figura 45: Polinomio interpolatore di  $f(x) = \frac{1}{1+25x^2}$  con  $n = 5$

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

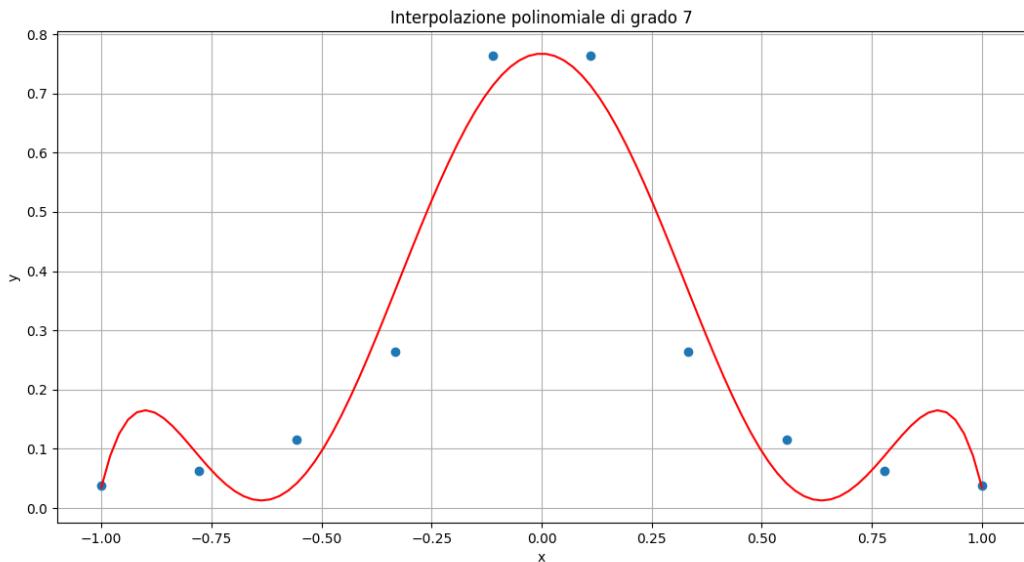


Figura 46: Polinomio interpolatore di  $f(x) = \frac{1}{1+25x^2}$  con  $n = 7$

Per il codice, si veda l'Esercizio 1.2 dell'Esercitazione 4.

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 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

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

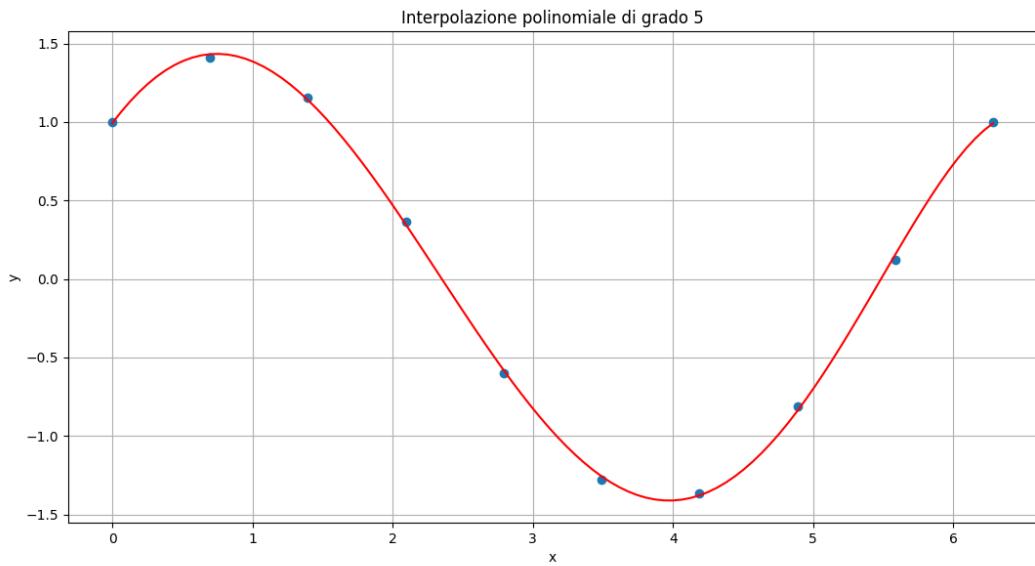


Figura 47: Polinomio interpolatore di  $f(x) = \sin(x) + \cos(x)$

Per il codice, si veda l'Esercizio 1.2 dell'Esercitazione 4.

### 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  $g(x) = x - \frac{f(x)}{f'(x)}$ .

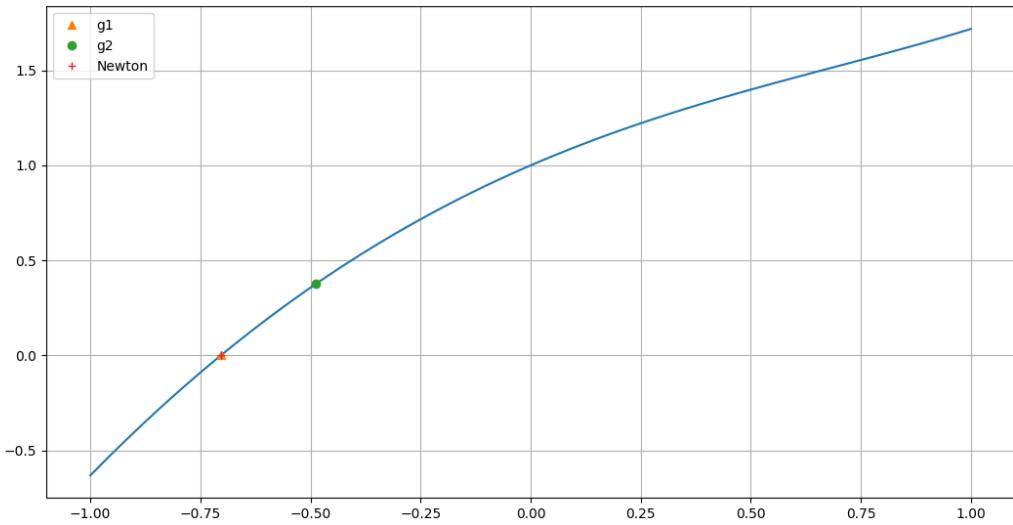


Figura 48: Solo  $g_1$  e Newton convergono alla soluzione esatta

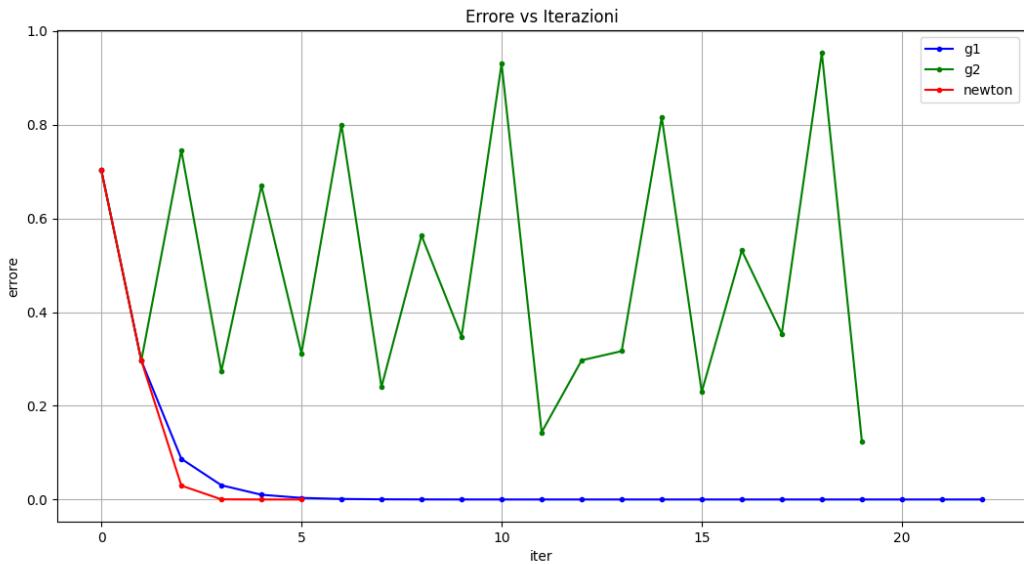


Figura 49:  $g_2$ , in figura limitato a 20 iterazioni, non converge

Per il codice, si veda l'Esercizio 1.1 dell'Esercitazione 5.

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

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

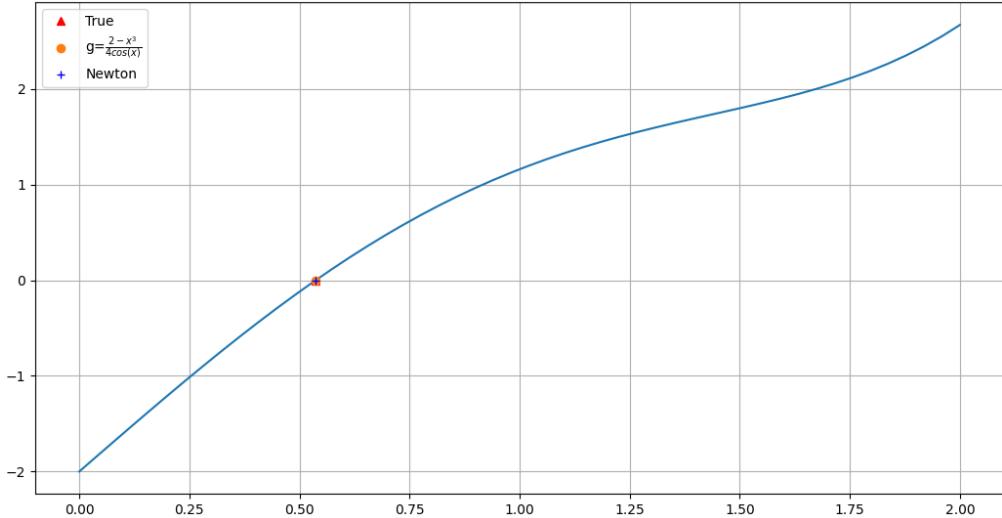


Figura 50: Sia Newton che  $g(x) = \frac{2-x^3}{4\cos(x)}$  convergono

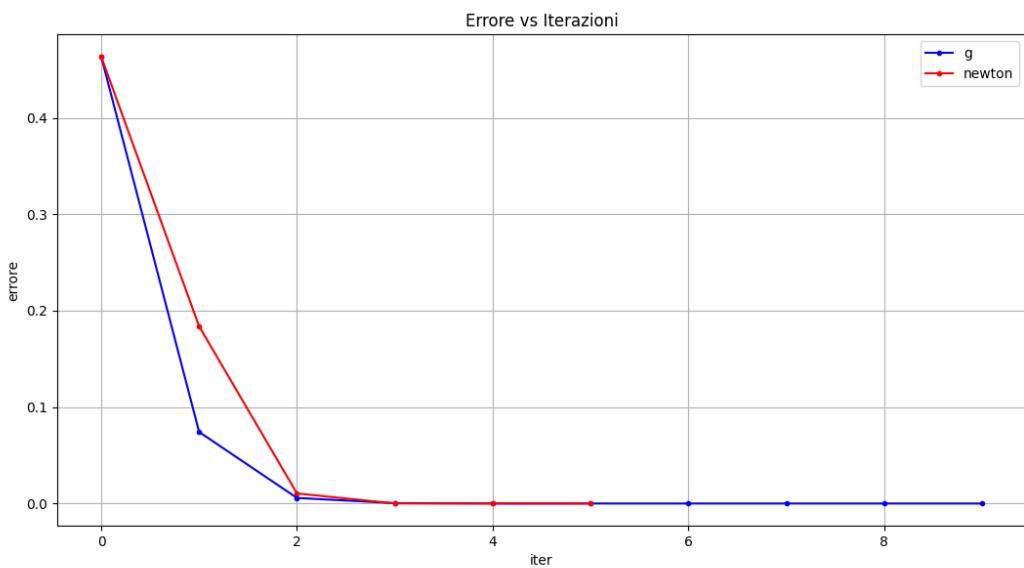


Figura 51: Confronto tra Newton e  $g(x) = \frac{2-x^3}{4\cos(x)}$

Per il codice, si veda l'Esercizio 1.2 dell'Esercitazione 5.  
 $g$  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
- $g(x) = x^{\frac{1}{3}} + 2$

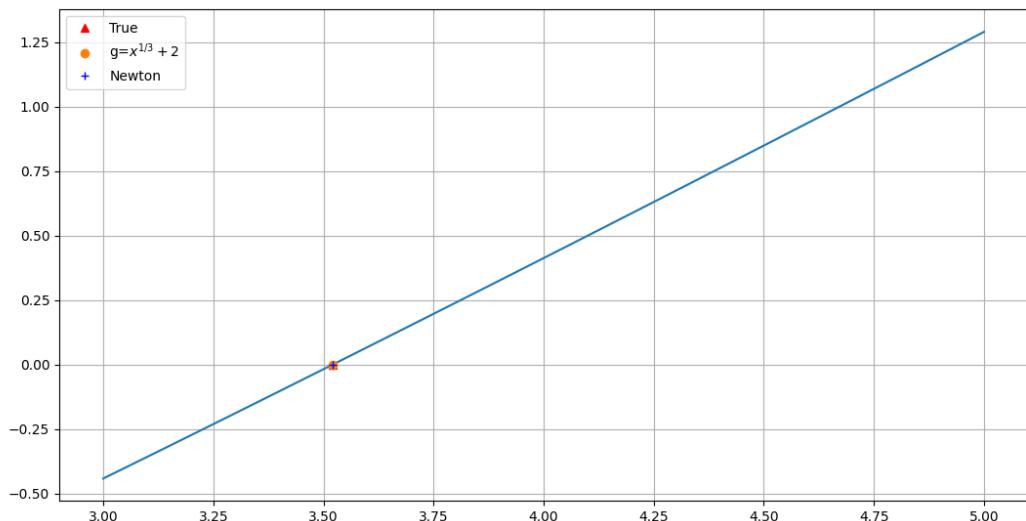


Figura 52: Sia Newton che  $g(x) = x^{\frac{1}{3}} + 2$  convergono

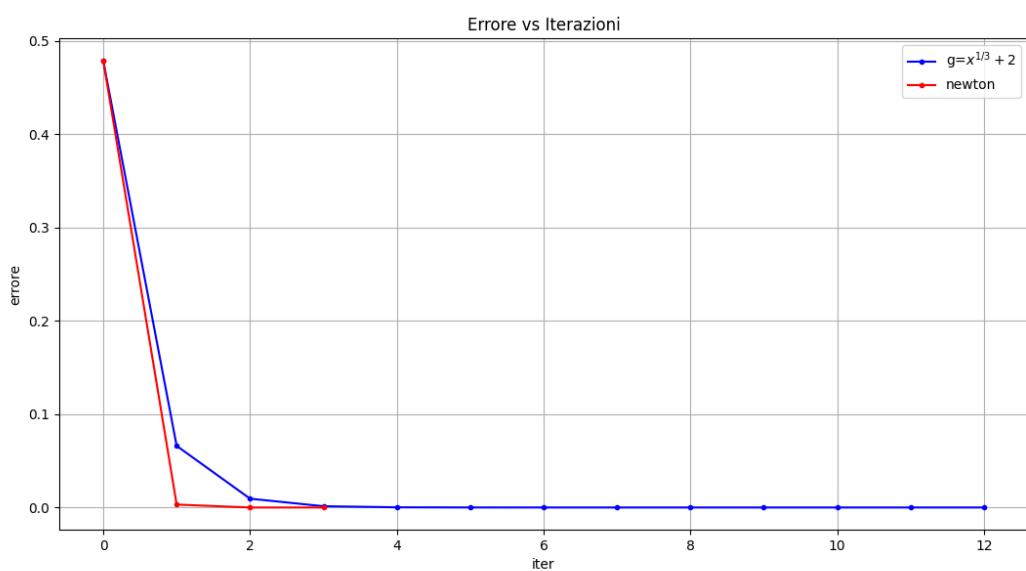


Figura 53: Newton converge più rapidamente di  $g(x) = x^{\frac{1}{3}} + 2$

Per il codice, si veda l'Esercizio 1.2 dell'Esercitazione 5.

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  $\alpha_k$  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  $\alpha_k$ : il primo è semplicemente tenendo un valore costante, il secondo è tramite un algoritmo che sfrutta il backtracking.

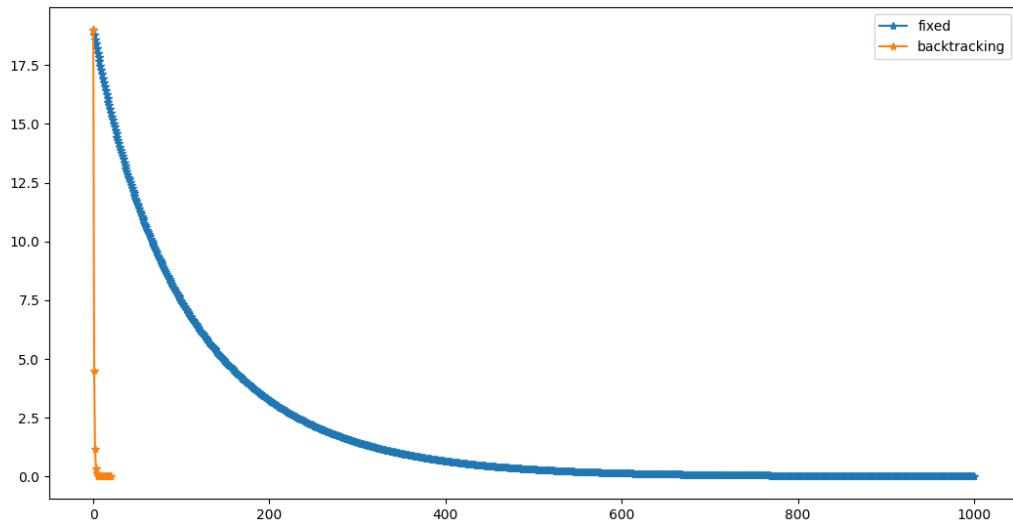


Figura 54: Valore della funzione a confronto con  $\alpha_k$  fisso e scelto con algoritmo di backtracking

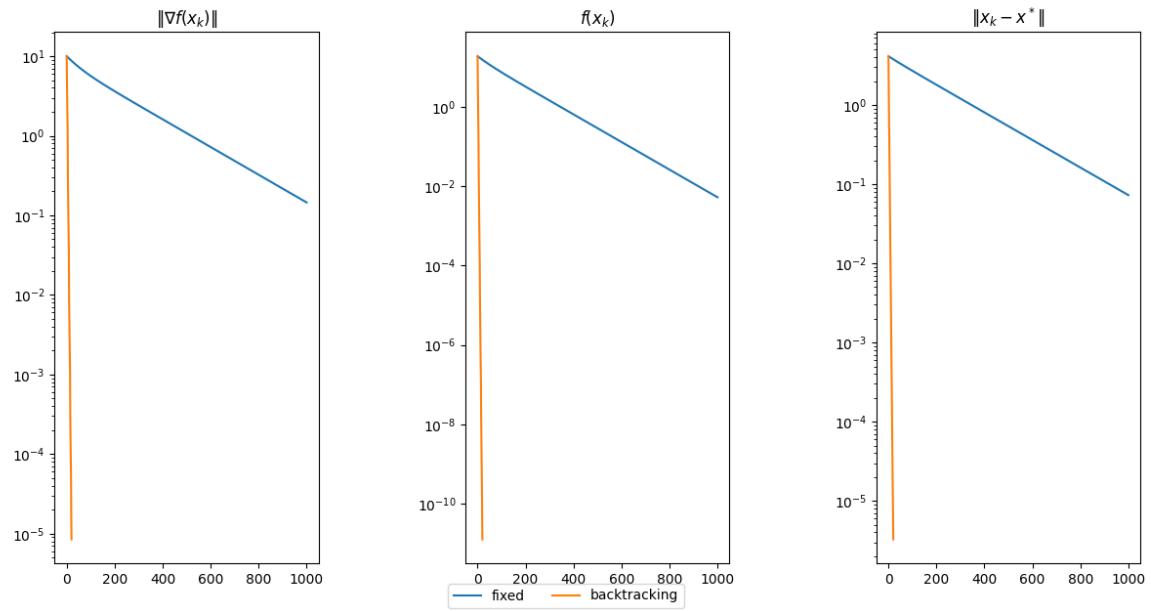


Figura 55: Confronto tra norma del gradiente, valore della funzione ed errore

Per il codice, si veda l’Esercizio 1.2 dell’Esercitazione 6.

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

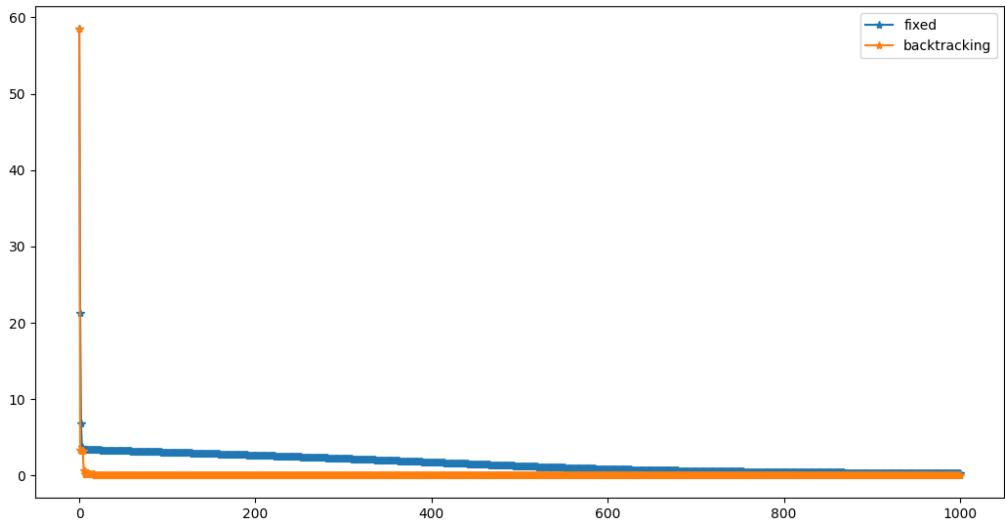


Figura 56: Valore della funzione a confronto con  $\alpha_k$  fisso e scelto con algoritmo di backtracking

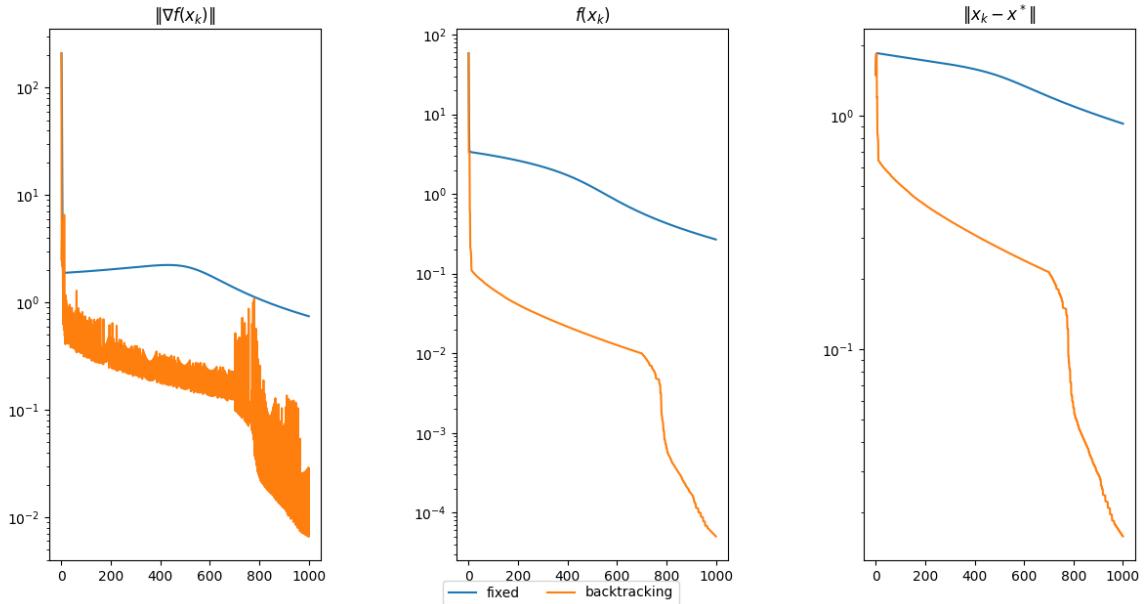


Figura 57: Confronto tra norma del gradiente, valore della funzione ed errore

Per il codice, si veda l’Esercizio 1.3 dell’Esercitazione 6.

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  $\alpha_k$  costante.

## 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.

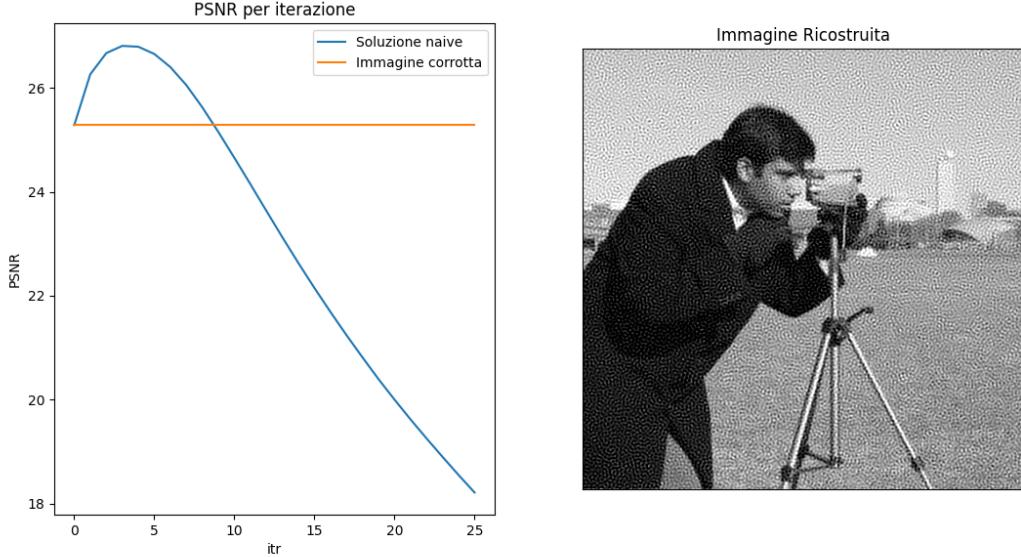


Figura 58: Ricostruzione naïve

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

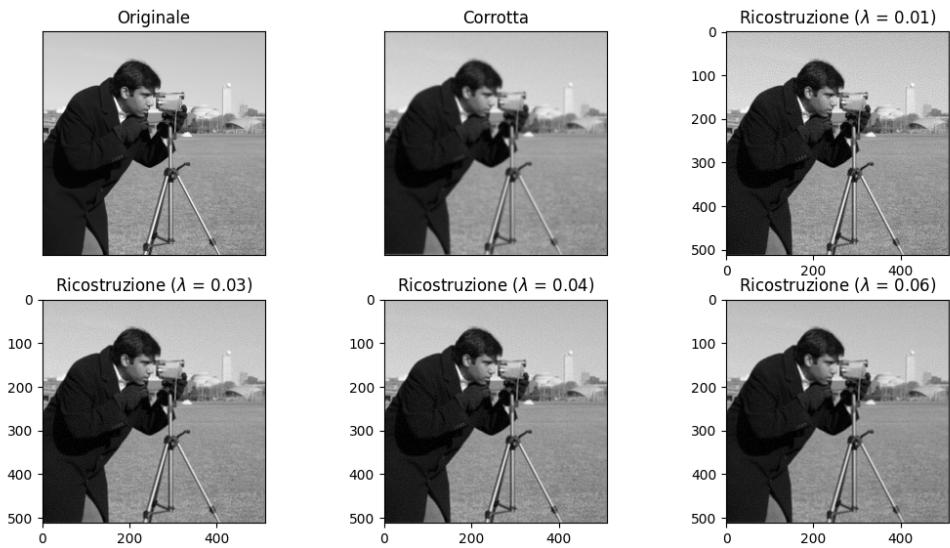


Figura 59: Ricostruzione con regolarizzazione di Tikhonov al variare di  $\lambda$

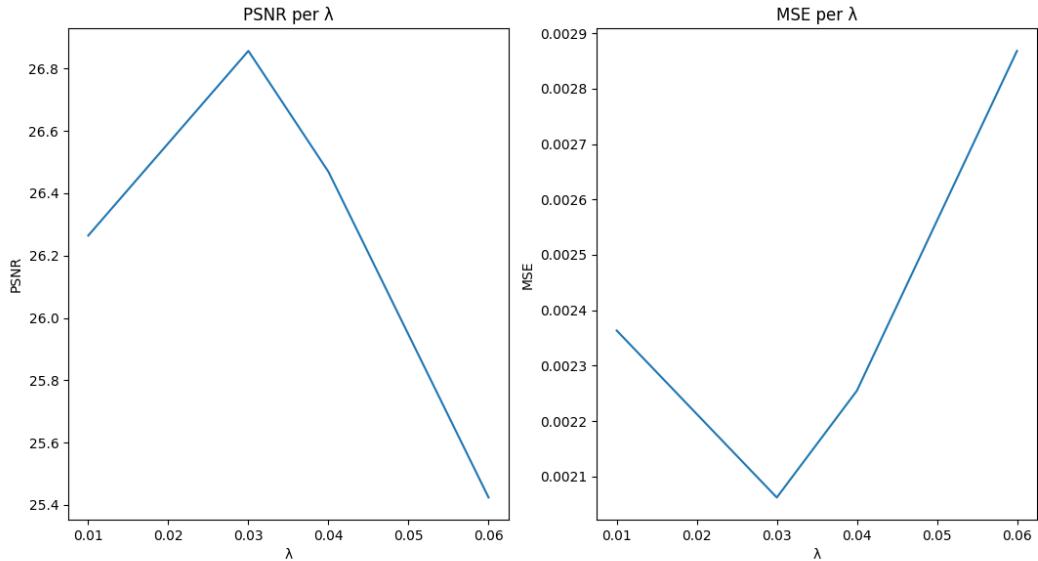


Figura 60: MSNR e MSE con i precedenti valori  $\lambda$

Si può vedere come la scelta del parametro  $\lambda$  giochi un ruolo fondamentale nella qualità della ricostruzione. Una corretta selezione di  $\lambda$  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.

---

```

1 import cv2 as cv
2 import numpy as np
3 import matplotlib.pyplot as plt

```

```

4  from skimage import data, metrics
5  from scipy import signal
6  from numpy import fft
7  from utils import psf_fft, A, AT, gaussian_kernel
8
9  # Immagine in floating point con valori tra 0 e 1
10 X = data.camera() / 255
11 m, n = X.shape
12
13 # Genera il filtro di blur
14 k = gaussian_kernel(24, 3)
15 plt.imshow(k)
16 plt.show()
17
18 # Blur with openCV
19 X_blurred = cv.filter2D(X, -1, k)
20 plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
21 plt.title('Original')
22 plt.xticks([]), plt.yticks([])
23 plt.subplot(122).imshow(X_blurred, cmap='gray', vmin=0, vmax=1)
24 plt.title('Blurred with OpenCV')
25 plt.xticks([]), plt.yticks([])
26 plt.show()
27
28 # Blur with FFT
29 K = psf_fft(k, 24, X.shape)
30 plt.imshow(np.abs(K))
31 plt.show()
32
33 X_blurred = A(X, K)
34
35 # show X_blurred
36 plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
37 plt.title('Original')
38 plt.xticks([]), plt.yticks([])
39 plt.subplot(122).imshow(X_blurred, cmap='gray', vmin=0, vmax=1)
40 plt.title('Blurred with FFT')
41 plt.xticks([]), plt.yticks([])
42 plt.show()
43
44 # Genera il rumore
45 sigma = 0.02
46 np.random.seed(42)
47 noise = np.random.normal(size=X.shape) * sigma
48
49 # Aggiungi blur e rumore
50 y = X_blurred + noise
51 PSNR = metrics.peak_signal_noise_ratio(X, y)
52 ATy = AT(y, K)
53
54 # Visualizziamo i risultati
55 plt.figure(figsize=(30, 10))
56 plt.subplot(121).imshow(X, cmap='gray', vmin=0, vmax=1)
57 plt.title('Original')

```

```

58 plt.xticks([]), plt.yticks([])
59 plt.subplot(122).imshow(y, cmap='gray', vmin=0, vmax=1)
60 plt.title(f'Corrupted (PSNR: {PSNR:.2f})')
61 plt.xticks([]), plt.yticks([])
62 plt.show()
63
64 # Soluzione naive
65 from scipy.optimize import minimize
66
67 # Funzione da minimizzare
68 def f(x):
69     x = x.reshape((m, n))
70     Ax = A(x, K)
71     return 0.5 * np.sum(np.square(Ax - y))
72
73 # Gradiente della funzione da minimizzare
74 def df(x):
75     x = x.reshape((m, n))
76     ATAx = AT(A(x,K),K)
77     d = ATAx - ATy
78     return d.reshape(m * n)
79
80 # Minimizzazione della funzione
81 x0 = y.reshape(m*n)
82 max_iter = 25
83 res = minimize(f, x0, method='CG', jac=df, options={'maxiter':max_iter, 'return_all':True})
84
85 # Per ogni iterazione calcola il PSNR rispetto all'originale
86 PSNR = np.zeros(max_iter + 1)
87 for k, x_k in enumerate(res.allvecs):
88     PSNR[k] = metrics.peak_signal_noise_ratio(X, x_k.reshape(X.shape))
89
90 # Risultato della minimizzazione
91 X_res = res.x.reshape((m, n))
92
93 # PSNR dell'immagine corrotta rispetto all'originale
94 starting_PSNR = np.full(PSNR.shape[0], metrics.peak_signal_noise_ratio(X, y))
95
96 # Visualizziamo i risultati
97 ax2 = plt.subplot(1, 2, 1)
98 ax2.plot(PSNR, label="Soluzione naive")
99 ax2.plot(starting_PSNR, label="Immagine corrotta")
100 plt.legend()
101 plt.title('PSNR per iterazione')
102 plt.ylabel("PSNR")
103 plt.xlabel('itr')
104 plt.subplot(1, 2, 2).imshow(X_res, cmap='gray', vmin=0, vmax=1)
105 plt.title('Immagine Ricostruita')
106 plt.xticks([]), plt.yticks([])
107 plt.show()
108
109 # Regolarizzazione
110 # Funzione da minimizzare
111 def f(x, L):

```

```

112     nsq = np.sum(np.square(x))
113     x  = x.reshape(m, n)
114     Ax = A(x, K)
115     return 0.5 * np.sum(np.square(Ax - y)) + 0.5 * L * nsq
116
117 # Gradiente della funzione da minimizzare
118 def df(x, L):
119     Lx = L * x
120     x = x.reshape(m, n)
121     ATAx = AT(A(x,K),K)
122     d = ATAx - ATy
123     return d.reshape(m * n) + Lx
124
125 x0 = y.reshape(m*n)
126 lambdas = [0.01, 0.03, 0.04, 0.06]
127 PSNRs = []
128 MSEs = []
129 images = []
130
131 # Ricostruzione per diversi valori del parametro di regolarizzazione
132 for i, L in enumerate(lambdas):
133     # Esegui la minimizzazione con al massimo 50 iterazioni
134     max_iter = 50
135     res = minimize(f, x0, (L), method='CG', jac=df, options={'maxiter':max_iter})
136
137     # Aggiungi la ricostruzione nella lista images
138     X_curr = res.x.reshape(X.shape)
139     images.append(X_curr)
140
141     # Stampa il PSNR per il valore di lambda attuale
142     PSNR = metrics.peak_signal_noise_ratio(X, X_curr)
143     PSNRs.append(PSNR)
144
145     # Stampa il MSE per il valore di lambda attuale
146     MSE = metrics.mean_squared_error(X, X_curr)
147     MSEs.append(MSE)
148     print(f'PSNR: {PSNR:.2f} (\u03bb = {L:.2f})')
149
150
151 # Visualizziamo i risultati
152 # PSNR e MSE per i diversi valori di lambda
153 ax1 = plt.subplot(1, 2, 1)
154 ax1.plot(lambdas, PSNRs)
155 plt.title('PSNR per \u03bb')
156 plt.ylabel("PSNR")
157 plt.xlabel('\u03bb')
158 ax2 = plt.subplot(1, 2, 2)
159 ax2.plot(lambdas, MSEs)
160 plt.title('MSE per \u03bb')
161 plt.ylabel("MSE")
162 plt.xlabel('\u03bb')
163 plt.show()
164
165

```

```

166
167 plt.figure(figsize=(30, 10))
168
169 (nrows, ncols) = ((len(lambdas) + 2) // 3, (len(lambdas) + 2) // 2)
170
171 plt.subplot(nrows, ncols, 1).imshow(X, cmap='gray', vmin=0, vmax=1)
172 plt.title("Originale")
173 plt.xticks([]), plt.yticks([])
174 plt.subplot(nrows, ncols, 2).imshow(y, cmap='gray', vmin=0, vmax=1)
175 plt.title("Corrotta")
176 plt.xticks([]), plt.yticks([])
177
178
179 for i, L in enumerate(lambdas):
180     plt.subplot(nrows, ncols, i + 3).imshow(images[i], cmap='gray', vmin=0, vmax=1)
181     plt.title(f"Ricostruzione ($\lambda$ = {L:.2f})")
182 plt.show()

```

---

## 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:

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

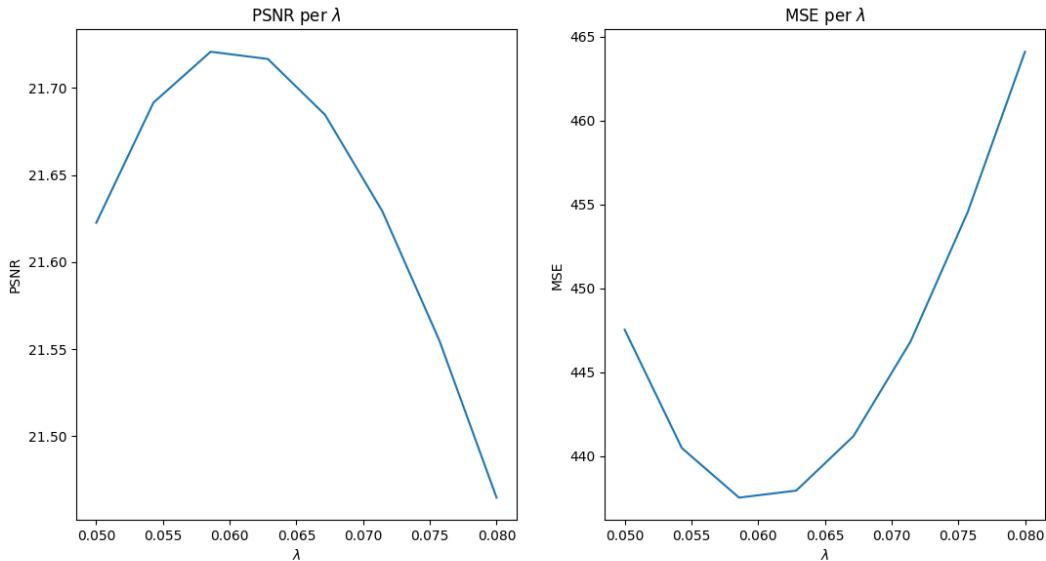


Figura 61: Ricostruzione con regolarizzazione e operatore downsampling dal fattore di scaling  $S = 8$

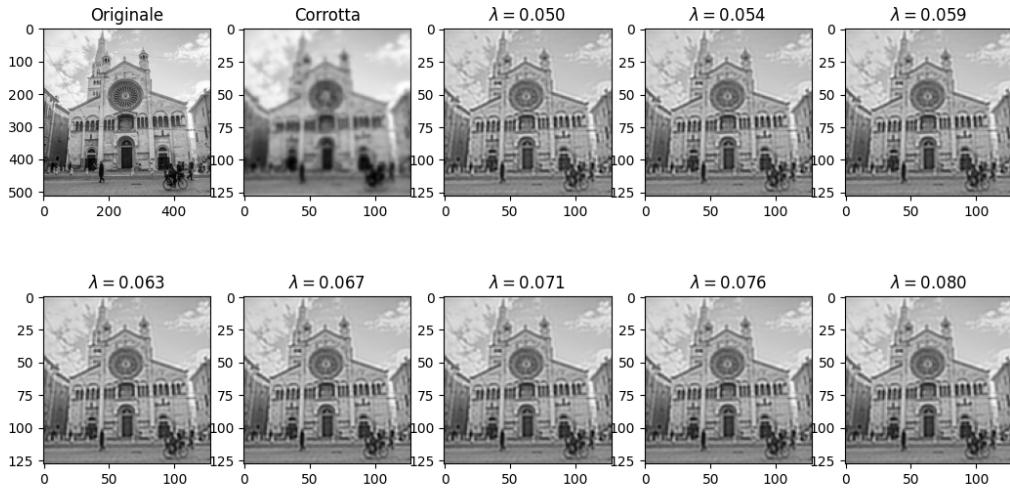


Figura 62: MSNR e MSE con i precedenti valori  $\lambda$

---

```

1 import cv2 as cv
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from skimage import data, metrics
5 from scipy import signal
6 from numpy import fft
7 from utils import psf_fft, A, AT, gaussian_kernel
8 from scipy.optimize import minimize
9 from skimage.io import imread
10 import os
11 img = 'modena.png'
12 cur_dir = os.path.dirname(os.path.abspath(__file__))
13 X = imread(os.path.join(cur_dir, img), as_gray=True)
14 m, n = X.shape
15
16 # Genera il filtro di blur
17 k = gaussian_kernel(5, 1.3)
18 # Blur with FFT
19 K = psf_fft(k, 5, X.shape)
20 X_blurred = A(X, K)
21
22 # rumore gaussiano con deviazione standard nell' intervallo (0, 0, 05].
23 sigma = 0.05
24 noise = np.random.normal(0, sigma, X.shape)
25
26 # Aggiungi blur e rumore
27 y = X_blurred + noise
28 PSNR = metrics.peak_signal_noise_ratio(X, y)
29 ATy = AT(y, K)
30

```

```

31 # Downsampling con fattore di sottocampionamento S
32 S=4
33 X_d = X[:,::S,::S]
34 y_d = y[:,::S,::S]
35 K_d = K[:,::S,::S]
36 ATy_d = AT(y_d, K_d)
37
38 # Funzione da minimizzare
39 def f(x, L):
40     nsq = np.sum(np.square(x))
41     x = x.reshape((m//S, n//S))
42     Ax = A(x, K_d)
43     return 0.5 * np.sum(np.square(Ax - y_d)) + 0.5 * L * nsq
44
45 # Gradiente della funzione da minimizzare
46 def df(x, L):
47     Lx = L * x
48     x = x.reshape(m//S, n//S)
49     ATAx = AT(A(x,K_d),K_d)
50     d = ATAx - ATy_d
51     return d.reshape(m//S * n//S) + Lx
52
53 x0 = y_d.reshape(m//S*n//S)
54 # lambdas = [0.05, 0.06, 0.07, 0.08]
55 lambdas = np.linspace(0.05, 0.08, 8)
56 PSNRs = []
57 MSEs = []
58 images = []
59
60 # Ricostruzione per diversi valori del parametro di regolarizzazione
61 for i, L in enumerate(lambdas):
62     # Esegui la minimizzazione con al massimo 50 iterazioni
63     max_iter = 50
64     res = minimize(f, x0, (L), method='CG', jac=df, options={'maxiter':max_iter})
65
66     # Aggiungi la ricostruzione nella lista images
67     X_curr = res.x.reshape(X_d.shape)
68     images.append(X_curr)
69
70     # Stampa il PSNR per il valore di lambda attuale
71     PSNR = metrics.peak_signal_noise_ratio(X_d, X_curr)
72     PSNRs.append(PSNR)
73
74     # Stampa il MSE per il valore di lambda attuale
75     MSE = metrics.mean_squared_error(X_d, X_curr)
76     MSEs.append(MSE)
77
78     print(f'PSNR for lambda={L}: {PSNR}')
79
80 # Stampa il PSNR ed MSE per il valore di lambda attuale
81 ax1 = plt.subplot(1, 2, 1)
82 ax1.plot(lambdas, PSNRs)
83 plt.title('PSNR per $\lambda$')
84 plt.ylabel("PSNR")

```

```

85 plt.xlabel('$\lambda$')
86 ax2 = plt.subplot(1, 2, 2)
87 ax2.plot(lambdas, MSEs)
88 plt.title('MSE per $\lambda$')
89 plt.ylabel("MSE")
90 plt.xlabel('$\lambda$')
91 plt.show()
92
93 # Stampa originale e le ricostruzioni
94 plt.figure()
95 plt.subplot(2, 5, 1)
96 plt.imshow(X, cmap='gray')
97 plt.title('Originale')
98 plt.subplot(2, 5, 2)
99 y_d = A(X_d, K_d)
100 plt.imshow(y_d, cmap='gray')
101 plt.title('Corrotta')
102 for i, X_curr in enumerate(images):
103     plt.subplot(2, 5, i + 3)
104     plt.imshow(X_curr, cmap='gray')
105     plt.title(f'$\lambda = {lambdas[i]:.3f}$')
106 plt.show()

```

---

Per altre immagini, si veda l'Esercizio 1.4 dell'Esercitazione 7.