COMP0114 Inverse Problems in Imaging. Coursework 1

L.C.

2025/01/22

1 Week 1: Solving Underdetermined Problems

1.1 Part a: Definition of Objective Function

Define a function of two variables, x and p, to compute:

$$\Phi = \sum_i |x_i|^p$$

subject to the constraint $x_1 + 2x_2 = 5$

Implementation is shown below:

```
def phi(x, p):
    # Compute the p-norm of vector x
    return np.sum(np.abs(x) ** p)
```

1.2 Part b: Computing Solutions for Different p Values

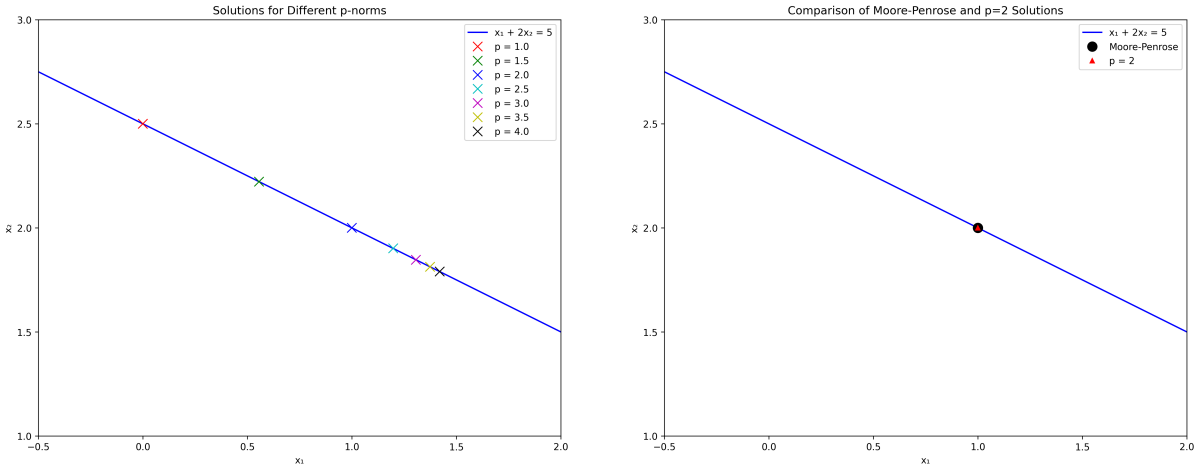
Use `scipy.optimize.minimize` to solve the optimization problem for $p = 1, 1.5, 2, 2.5, 3, 3.5, 4$:

```
def solve_optimization(p_values):
    # Solve the optimization problem for multiple p values

    # Define constraint matrix and vector
    A = np.array([[1, 2]])
    b = np.array([5])
    # Define equality constraint
    constraint = {'type': 'eq', 'fun': lambda x: A @ x - b}
    solutions = {}
    for p in p_values:
        result = optimize.minimize(phi,x0=np.zeros(2),args=(p,),
            constraints=[constraint],method='SLSQP',options={'ftol': 1e-8})
        solutions[p] = {'x': result.x,'phi': result.fun,'success': result.success}

    return solutions
```

```
p = 1.0: x = [0.0000, 2.5000], Φ = 2.5000
p = 1.5: x = [0.5556, 2.2222], Φ = 3.7268
p = 2.0: x = [1.0000, 2.0000], Φ = 5.0000
p = 2.5: x = [1.1977, 1.9012], Φ = 6.5535
p = 3.0: x = [1.3060, 1.8470], Φ = 8.5284
p = 3.5: x = [1.3740, 1.8130], Φ = 11.0646
p = 4.0: x = [1.4205, 1.7897], Φ = 14.3321
```

1.3 Part c: Visualization of Solutions

Plot the solutions obtained for different p values along with the constraint line:



(a) Solutions Plot

(b) Moore-Penrose Comparison

Figure 1: Analysis Comparison

1.4 Part d: Moore-Penrose Solution

The Moore-Penrose generalized inverse is defined as:

$$A^{\dagger} := A^T (AA^T)^{-1}$$

Solution and explanation with $p = 2$ are shown below:

```
Moore-Penrose solution:
x_MP = [1.0000, 2.0000]
```

```
Difference between Moore-Penrose and p=2 solutions: 0.00000000
```

The Moore-Penrose solution corresponds to the case when $p = 2$, which can be proven through the following steps:

1. For $p = 2$, our optimization problem is:

$$\text{minimize } |x_1|^2 + |x_2|^2 \text{ subject to } x_1 + 2x_2 = 5$$

2. Using Lagrange multipliers:

$$L(x_1, x_2, \lambda) = x_1^2 + x_2^2 + \lambda(x_1 + 2x_2 - 5)$$

3. The first-order conditions give:

$$\begin{aligned} 2x_1 + \lambda &= 0 \\ 2x_2 + 2\lambda &= 0 \\ x_1 + 2x_2 - 5 &= 0 \end{aligned}$$

4. Solving these equations yields exactly the Moore-Penrose solution.

2 Week 2: Singular Value Decomposition

2.1 Part a: Setting up the Spatial Grid

Create a spatial grid on the interval $[-1, 1]$ with n equally spaced points. The step size is given by $\delta n = 2/(n-1)$:

```
def create_spatial_grid(n):
    # Create a spatial grid on [-1,1] with n points
    x = np.linspace(-1, 1, n)
    dn = 2 / (n-1)
    return x, dn
```

2.2 Part b: Gaussian Function

Create a vector of values for the Gaussian function centered at $\mu = 0$ with $\sigma = 0.2$:

$$G(x) = \frac{\delta n}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

Evaluate it along the grid set in part a). The result of evaluation is shown below as a plot:

```
def gaussian_function(x, delta_n, sigma=0.2, mu=0):
    # Compute Gaussian function values
    prefactor = delta_n / (np.sqrt(2 * np.pi) * sigma)
    exponent = -((x - mu) ** 2) / (2 * sigma ** 2)
    return prefactor * np.exp(exponent)
```

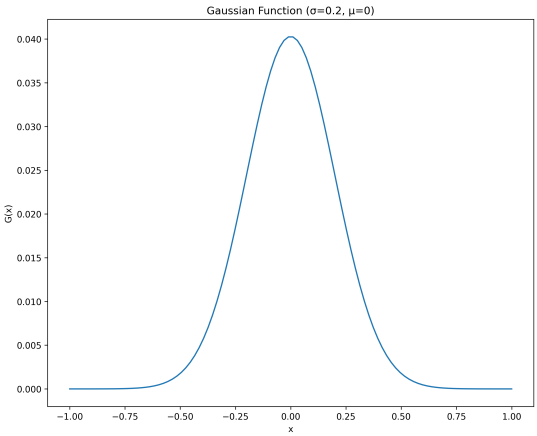


Figure 2: Gaussian Function

2.3 Part c: Convolution Matrix

Create the convolution matrix A of size n × n with entries:

$$A_{i,j} = G(x_i - x_j) = \frac{\delta n}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x_i - x_j)^2}{2\sigma^2}\right)$$

```
def create_convolution_matrix(n, sigma=0.2):
    # Create convolution matrix A
    x, dn = create_spatial_grid(n)
    X, Y = np.meshgrid(x, x)
    diff = X - Y
    return gaussian_function(diff, dn, sigma)
```

2.4 Part d: Matrix Visualization

Visualize the convolution matrix A for $n = 100$ as a 2D image:

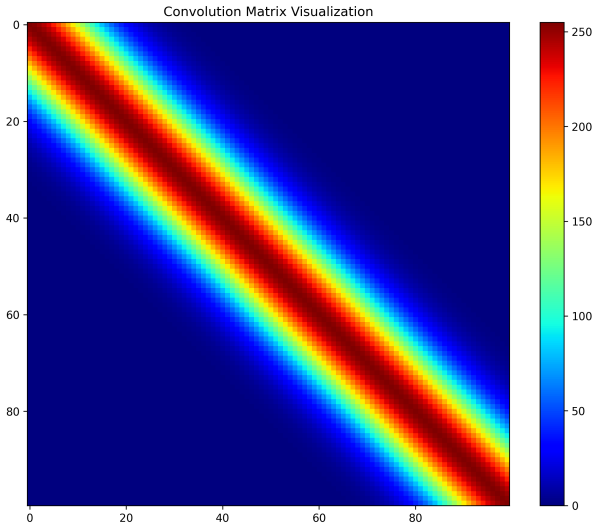


Figure 3: Convolution Visualization

2.5 Part e: SVD Computation and Verification

Compute the SVD of matrix A using `numpy.linalg.svd` functions. Then, verify the equation $A = UWV^T$ is satisfied by computing the error:

```
def compute_svd(A):
    # Compute SVD components of matrix A
    U, s, VT = np.linalg.svd(A, full_matrices=True)
    # Create diagonal matrix W
    W = np.diag(s)
    return U, W, VT

def verify_svd(A, U, W, VT, tol=1e-9):
    # Verify SVD decomposition A = UWV^T within tolerance
    A_reconstructed = U @ W @ VT
    error = np.linalg.norm(A - A_reconstructed)
    print(f"SVD reconstruction error: {error:.8e}")
    is_valid = np.allclose(A, A_reconstructed, atol=tol)
    print(f"A = UWV^T: {is_valid}")
```

```
SVD reconstruction error: 3.21722242e-15
A = UWV^T: True
```

2.6 Part f: Pseudoinverse Computation

Compute the pseudoinverse $A^\dagger$ of A by using the formula $A^\dagger = VW^\dagger U^T$ as given in lectures.

- Create a method for constructing $W^\dagger$.
- For the case $n = 10$, check that this has the property $WW^\dagger = W^\dagger W = Id_n$ where Id_n is the $n \times n$ Identity matrix.
- Check also that $AA^\dagger = A^\dagger A = Id_n$

```
def compute_pseudoinverse(U, W, VT, n):
    # Compute and verify pseudoinverse properties
```

```
    # Create identity matrix
    I = np.eye(n)
    # Compute W†
    W_diag = np.diag(W)
    W_pinv = np.zeros_like(W)
    np.fill_diagonal(W_pinv, 1/W_diag)
    # Compute WW† =? W†W =? I
    WWi = W @ W_pinv
    WiW = W_pinv @ W

    # Compute A†
    A_pinv = VT.T @ W_pinv @ U.T
    # Compute AA† =? A†A =? I
    A = U @ W @ VT
    AAi = A @ A_pinv
    AiA = A_pinv @ A
    return A_pinv
```

```
Verification of W† properties:
||WW† - I|| = 1.57009246e-16
||W†W - I|| = 1.57009246e-16
WW† = I: True
W†W = I: True

Verification of A† properties:
||AA† - I|| = 6.00072254e-15
||A†A - I|| = 1.06933827e-14
AA† = I: True
A†A = I: True
```

Results show that for $n = 10$, all properties are satisfied within numerical precision ($error \approx 10^{-15}$).

2.7 Part g: Analysis for Different n Values

Repeat the last two steps. Analysis for n = 20 and n = 100, including plotting first/last 9 columns of V and singular values:

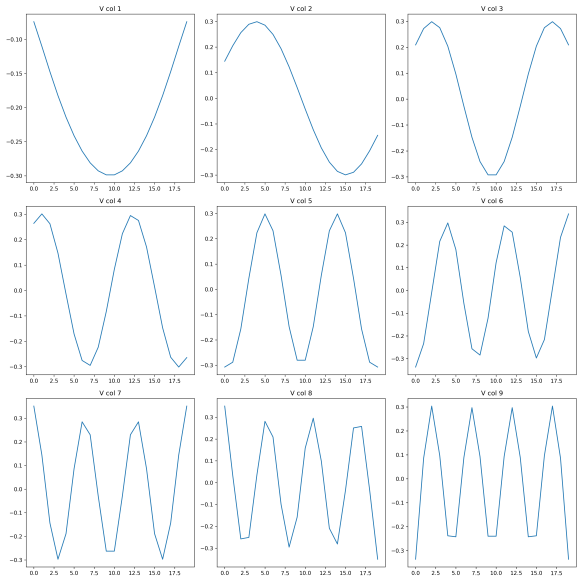
Table 1: Comparison of W† and A† Properties for Different n

Property	**n = 20**	**n = 100**
Verification of W† properties:		
$\|WW^\dagger - I\|$	1.92296269e-16	9.15513360e-16
$\|W^\dagger W - I\|$	1.92296269e-16	9.15513360e-16
$WW^\dagger = I$	True	True
$W^\dagger W = I$	True	True
Verification of A† properties:		
$\|AA^\dagger - I\|$	2.96632057e-10	1.64578354e+01
$\|A^\dagger A - I\|$	5.32289994e-10	1.45391549e+01
$AA^\dagger = I$	True	False
$A^\dagger A = I$	True	False

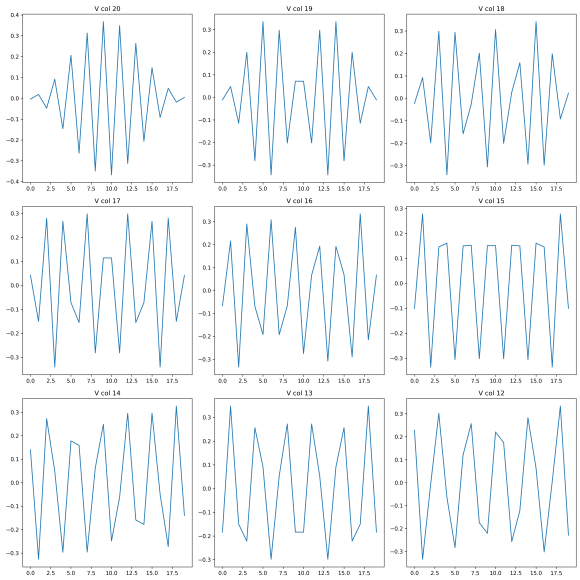
For $n = 20$: The SVD reconstruction error remains small ($\approx 1.84e-15$), while the pseudoinverse properties are still well-satisfied but with slightly larger errors ($\approx 3.65e-10$). The matrix remains well-conditioned.

Plots are shown below:

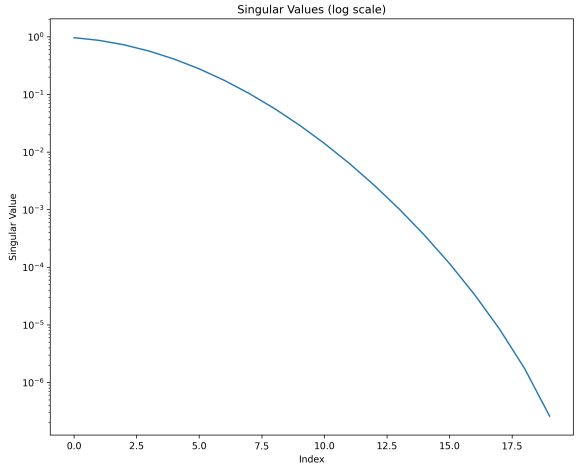
For $n = 100$:



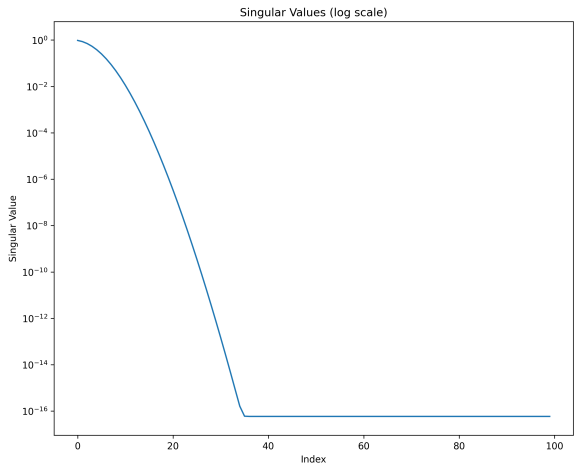
(a) First 9 columns ($n = 20$)



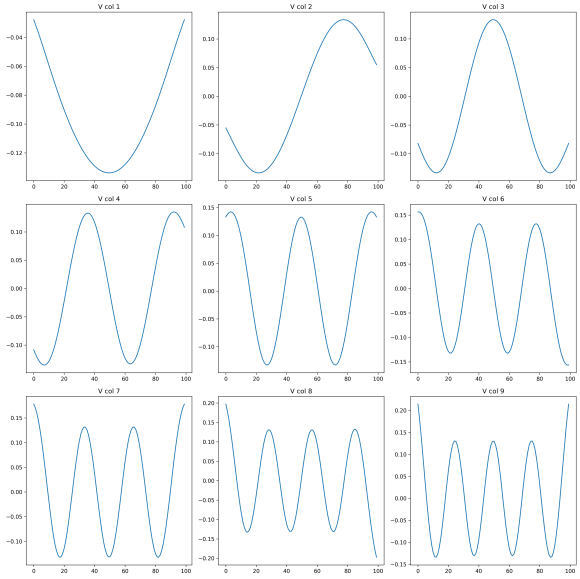
(b) Last 9 columns ($n = 20$)



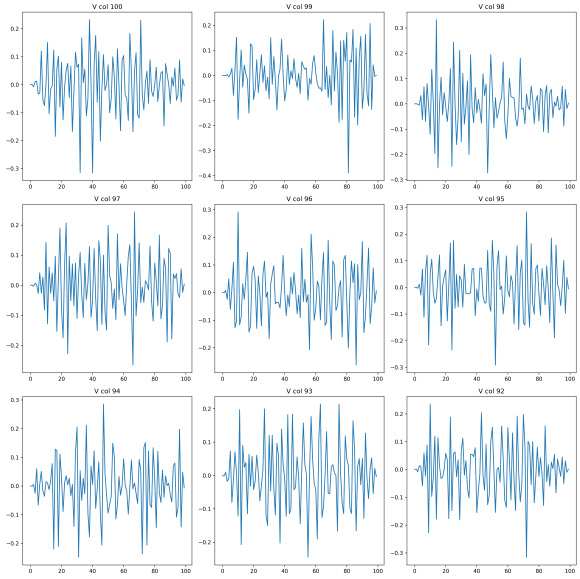
(c) Singular values ($n = 20$)



(d) Singular values ($n = 100$)



(e) First 9 columns ($n = 100$)



(f) Last 9 columns ($n = 100$)

Figure 4: Analysis of V matrices and singular values for $n = 20$ and $n = 100$

- First 9 columns of V show smooth oscillatory patterns, and the number of oscillations increases with column index, which represent low-frequency components of the decomposition.
- Last 9 columns of V exhibit high-frequency oscillations and more irregular patterns, which correspond to the smallest singular values. These components are more susceptible to numerical errors.
- Singular values (on logarithmic scale) show exponential decay, dropping from 1 to 10^{-16} over the indices, which indicate the ill-conditioning of the matrix. The condition number (ratio of largest to smallest singular value) is very large.

The observations confirm that while the SVD itself remains stable, the pseudoinverse computation becomes challenging for larger matrices due to the exponential decay of singular values.

3 Week 3: Convolutions and Fourier Transform

3.1 Part a: Step Function Implementation

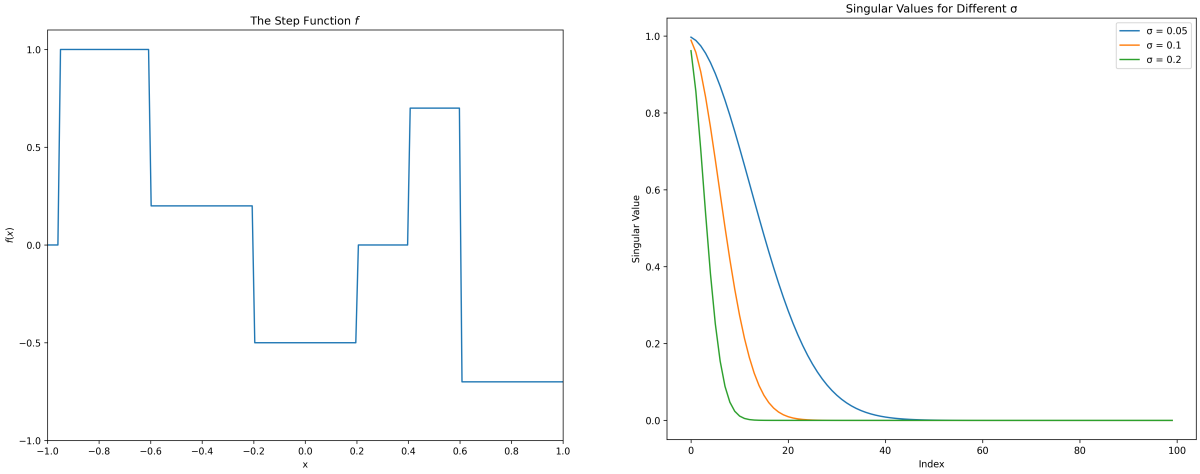
First, I implement the step function $f(x)$ defined by:

$$f(x) = \chi_{(-0.95,-0.6]}(x) + 0.2\chi_{(-0.6,-0.2]}(x) - 0.5\chi_{(-0.2,0.2]}(x) + 0.7\chi_{(0.4,0.6]}(x) - 0.7\chi_{(0.6,1]}(x)$$

where $\chi(x)$ is the characteristic function:

$$\chi_{(a,b]}(x) = \begin{cases} 1 & \text{for } a < x \leq b \\ 0 & \text{otherwise} \end{cases}$$

While $n = 1000$, the step function on $[-1, 1]$ is shown below:



(a) Step Function

(b) Singular Values

Figure 5: Step Function and Its Singular Values Analysis

3.2 Part b: Computing Matrix A and Singular Values

Compute the convolution matrix A for different values of σ and plot its singular values:

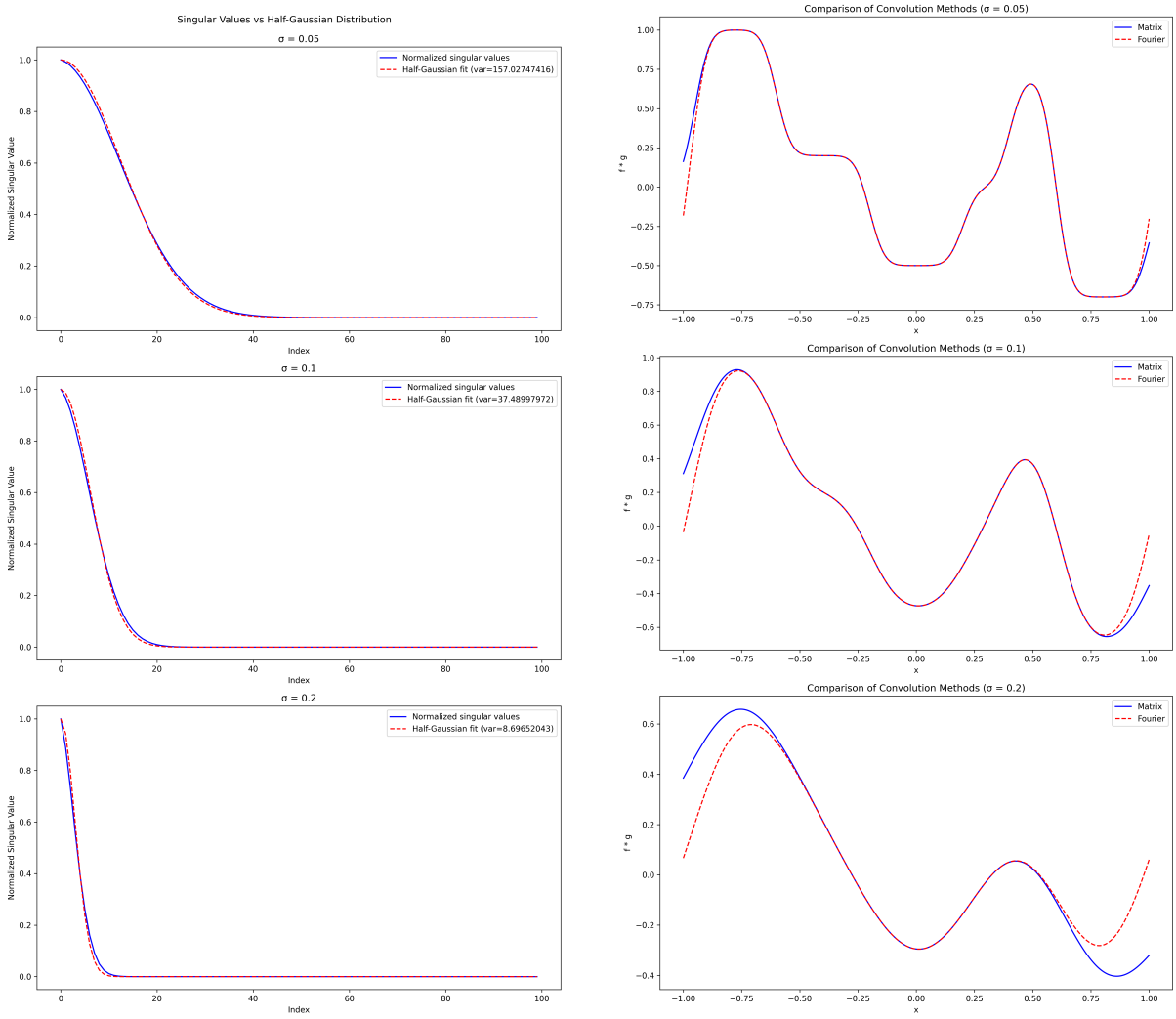
3.3 Part c: Verification of Gaussian Behavior

I verify that the singular values follow a Gaussian distribution and determine their variance:

```python
def half_gaussian(x, sigma):
    # Half Gaussian function for fitting
    return np.exp(-x**2 / (2*sigma**2))
```

Table 2: Analysis of Different σ Values

σ	Variance	Maximum Difference	Analysis
0.05	157.027	0.344	Highest variance, lowest difference
0.10	37.490	0.347	Moderate variance and difference
0.20	8.697	0.381	Lowest variance, highest difference



(a) Singular Value Analysis

(b) Comparison of Methods for Different σ

Figure 6: Singular Value Analysis and Methods Comparison

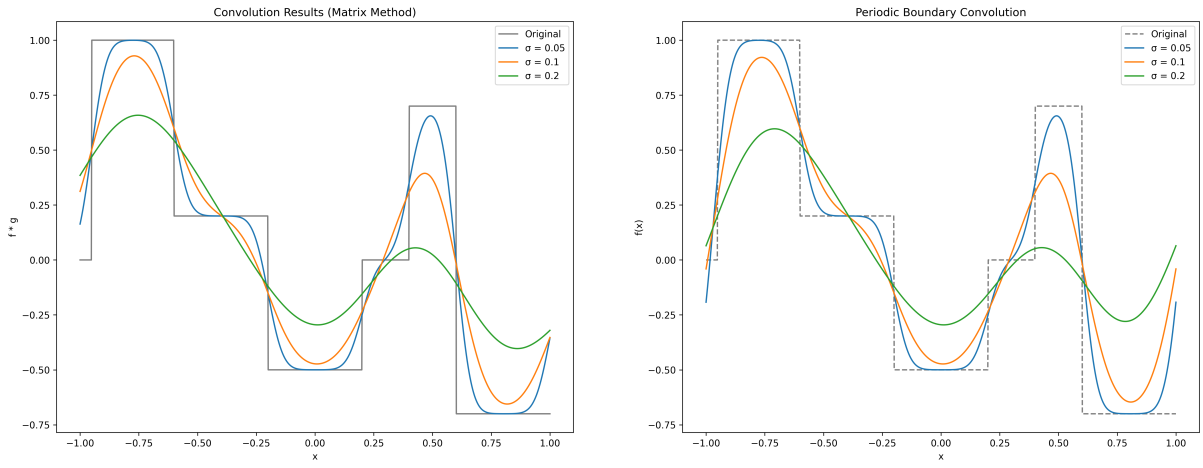
3.4 Part d: Matrix Convolution

Perform convolution using matrix multiplication:

```python
def matrix_convolution(f, sigma, n=1000):
    # Perform convolution using matrix multiplication
    x = np.linspace(-1, 1, n)
```

```
    f_values = f(x)
    A = create_convolution_matrix(n, sigma)
    return x, A @ f_values
```

The result is as below:



(a) Matrix Convolution

(b) Periodic Boundary Conditions

Figure 7: Convolution Results w/wo Periodic Boundary Conditions

3.5 Part e: Fourier Space Convolution

Implement convolution using Fourier transform:

```
def fourier_convolution(f, sigma, n=1000):
    # Perform convolution using Fourier transform
    x = np.linspace(-1, 1, n)
    dx = x[1] - x[0]

    # Compute Fourier transform of f
    f_values = f(x)
    F_f = np.fft.fftshift(np.fft.fft(np.fft.fftshift(f_values)))

    # Create Gaussian in frequency space
    freq = np.fft.fftshift(np.fft.fftfreq(n, dx))
    G_f = np.exp(-2*(np.pi*freq*sigma)**2)

    # Multiply in Fourier space and inverse transform
    conv = np.fft.ifftshift(np.fft.ifft(np.fft.ifftshift(F_f * G_f)))
    return x, conv.real
```

Comparing the two convolution methods(Fourier Space vs Matrix Convolution), I observe several differences:

1. Edge Behavior: Matrix convolution shows smooth decay at boundaries, while Fourier method exhibits oscillations near boundaries (-1 and 1). Differences most pronounced for $\sigma = 0.2$.
2. Amplitude:

- For $\sigma = 0.05$: nearly identical except at edges
- For $\sigma = 0.1$: slight amplitude differences near peaks
- For $\sigma = 0.2$: noticeable amplitude differences, especially at leftmost peak

3. Smoothness: Matrix method produces consistently smooth results, while Fourier method shows some high-frequency artifacts at discontinuities.

3.6 Part f: Periodic Boundary Conditions

using periodic boundary conditions, and repeat the convolution steps:

```
def create_periodic_convolution_matrix(n, sigma=0.2):
    # Create convolution matrix with periodic boundary conditions
    x = np.linspace(-1, 1, n)
    dx = x[1] - x[0]
    A = np.zeros((n, n))

    for i in range(n):
        for j in range(n):
            # Calculate distance with periodic boundary conditions
            dist = min(abs(x[i] - x[j]), 2 - abs(x[i] - x[j]))
            A[i,j] = dx/np.sqrt(2*np.pi*sigma**2) * np.exp(-dist**2/(2*sigma**2))

    return A
```

The periodic boundary conditions provide a more physically meaningful result for cyclic signals, eliminating artificial edge effects seen in the previous methods.