inverseScattering_aux

May 2, 2024

[ ]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import hankel2
from typing import Any
from numpy import dtype, ndarray
import numpy.random as rnd
```

0.1 Forward Scattering

[ ]:
```python
def circle_data(a: float, N: int, cx: float = 0.0, cy: float = 0.0) -> np.
 ↪ndarray:
    """
    Calculate coordinates, lengths, angles, and normals for points on a circle.

    Args:
        a (float): Radius of the circle.
        N (int): Number of points to generate.
        cx (float): x-coordinate of the center of the circle. Default is 0.0.
        cy (float): y-coordinate of the center of the circle. Default is 0.0.

    Returns:
        numpy.ndarray: Array containing coordinates (x, y), lengths, angles,
                       and normals for each point.
    """
    data = np.zeros((N, 2))
    # Calculate coordinates xn, yn
    for i in range(N):
        theta = (2 * np.pi / N) * i
        xn = cx + a * np.cos(theta)
        yn = cy + a * np.sin(theta)
        data[i, 0] = xn
        data[i, 1] = yn
    # Duplicate first row at the end for circularity
    circular_data = np.vstack((data[1:], data[0]))
    data = np.hstack((data, circular_data))
    return data
```

```python
def kite_data(a: float, N: int, cx: float = 0.0, cy: float = 0.0) -> np.ndarray:
    """
    Generate coordinates for points on a kite shape.

    Args:
        a (float): Scaling factor for the kite.
        N (int): Number of points to generate.
        cx (float): x-coordinate of the center of the kite. Default is 0.0.
        cy (float): y-coordinate of the center of the kite. Default is 0.0.

    Returns:
        numpy.ndarray: Array containing coordinates (x, y) of the kite points.
    """
    data = np.zeros((N, 2))
    for i in range(N):
        theta = (2 * np.pi / N) * i
        x = cx + a * (np.cos(theta) + 0.65 * np.cos(2*theta) - 0.65)
        y = cy + a * np.sin(theta)
        data[i, 0] = x
        data[i, 1] = y

    # Duplicate first row at the end for circularity
    circular_data = np.vstack((data[1:], data[0]))
    data = np.hstack((data, circular_data))
    return data

def rectangle_data(a: float, b: float, N: int, cx: float = 0.0, cy: float = 0.
↪0) -> np.ndarray:
    """
    Generate coordinates for points on a rectangle.

    Args:
        a (float): Length of the rectangle along the x-axis.
        b (float): Length of the rectangle along the y-axis.
        N (int): Number of points to generate.
        cx (float): x-coordinate of the center of the rectangle. Default is 0.0.
        cy (float): y-coordinate of the center of the rectangle. Default is 0.0.

    Returns:
        numpy.ndarray: Array containing coordinates (x, y) of the rectangle␣
↪points.
    """
    N = 4*(N//4)
    data = np.zeros((N, 2))
    # Points on the top side
    data[:N//4, 0] = np.linspace(cx - a/2, cx + a/2, N//4)
    data[:N//4, 1] = cy + b/2
```

```python
    # Points on the right side
    data[N//4:N//2, 0] = cx + a/2
    data[N//4:N//2, 1] = np.linspace(cy + b/2, cy - b/2, N//4)
    # Points on the bottom side
    data[N//2:3*N//4, 0] = np.linspace(cx + a/2, cx - a/2, N//4)
    data[N//2:3*N//4, 1] = cy - b/2
    # Points on the left side
    data[3*N//4:, 0] = cx - a/2
    data[3*N//4:, 1] = np.linspace(cy - b/2, cy + b/2, N//4)

    return data

def data_ellypse(rx: float, ry: float, N: int) -> np.ndarray:
    """
    Generate coordinates for points on an ellipse.

    Args:
        rx (float): The radius of the ellipse along the x-axis.
        ry (float): The radius of the ellipse along the y-axis.
        N (int): The number of points to generate.

    Returns:
        np.ndarray: Array containing coordinates of points on the ellipse.
    """
    phi_stop = + np.pi - 2 * np.pi / N
    phi = np.linspace(- np.pi, phi_stop, N)
    data = np.zeros((N, 2))
    data[:, 0] = rx * np.cos(phi)
    data[:, 1] = ry * np.sin(phi)
    circular_data = np.vstack((data[1:], data[0]))
    data = np.hstack((data, circular_data))
    return data

def is_inside_kite(x: np.ndarray, y: np.ndarray, data: np.ndarray) -> bool:
    """
    Check if the given point (x, y) is inside the geometry defined by data.

    Args:
        x (np.ndarray): Array of x-coordinates of the points.
        y (np.ndarray): Array of y-coordinates of the points.
        data (np.ndarray): Array defining the geometry.

    Returns:
        bool: True if the point is inside the geometry, False otherwise.
    """
    n = len(data)
    inside = False
```

```python
    for i in range(n):
        j = (i + 1) % n
        if ((data[i, 1] > y) != (data[j, 1] > y)) and \
                (x < (data[j, 0] - data[i, 0]) * (y - data[i, 1]) / (data[j, 1]␣
↪- data[i, 1]) + data[i, 0]):
            inside = not inside
    return inside

def is_inside_rectangle(x: np.ndarray, y: np.ndarray, data: np.ndarray) -> bool:
    """
    Check if points (x, y) are inside a rectangle defined by data.

    Parameters:
        x (np.ndarray): X-coordinates of the points.
        y (np.ndarray): Y-coordinates of the points.
        data (np.ndarray): Vertices of the rectangle (2x2 array).

    Returns:
        bool: True if points are inside the rectangle, False otherwise.
    """
    x_min, x_max = data[:, 0].min(), data[:, 0].max()
    y_min, y_max = data[:, 1].min(), data[:, 1].max()

    inside = np.logical_and(x >= x_min, x <= x_max)
    inside = np.logical_and(inside, y >= y_min)
    inside = np.logical_and(inside, y <= y_max)

    return np.all(inside)
```

[ ]:

```python
def calculate_current_distribution_aux_sources(data_contour: np.ndarray,␣
↪data_aux: np.ndarray, phi_i: float, source_type: str = "plane_wave",␣
↪x_source: float = None, y_source: float = None) -> np.ndarray:
    """
    Calculate the current distribution for auxiliary sources.

    Args:
        data_contour: Contour data.
        data_aux: Auxiliary sources data.
        phi_i: Angle of incidence in degrees.
        source_type: Type of source, either "plane_wave" or "point_source"␣
↪(default is "plane_wave").
        x_source: x-coordinate of the point source (required if source_type is␣
↪"point_source").
        y_source: y-coordinate of the point source (required if source_type is␣
↪"point_source").

    Returns:
```

```
        ndarray: Current distribution for auxiliary sources.
    """
    k = 2 * np.pi  # Wavenumber
    phi_i = np.deg2rad(phi_i)  # Angle of incidence in radians
    M = len(data_contour)  # Number of segments
    N = len(data_aux)  # Number of auxiliary sources
    Z = np.zeros((M, N), dtype=complex)  # Impedance matrix
    V = np.zeros((M, 1), dtype=complex)  # Excitation vector

    # Calculate impedance matrix
    for l in range(N):
        xm, ym = data_aux[l, 0], data_aux[l, 1]
        for m in range(M):
            xl, yl = data_contour[m, 0], data_contour[m, 1]
            rml = np.sqrt((xm - xl)**2 + (ym - yl)**2)
            Z[m, l] += hankel2(0, k * rml)

    # Calculate excitation vector
    if source_type == "plane_wave":
        for i in range(M):
            xm, ym = data_contour[i, 0], data_contour[i, 1]
            V[i] = np.exp(1j * k * (xm * np.cos(phi_i) + ym * np.sin(phi_i)))
    elif source_type == "point_source":
        for i in range(M):
            xm, ym = data_contour[i, 0], data_contour[i, 1]
            r_source = np.sqrt((xm - x_source)**2 + (ym - y_source)**2)
            V[i] = np.exp(1j * k * r_source)
    else:
        raise ValueError("Invalid source_type. Must be either 'plane_wave' or␣
↪'point_source'.")

    # Solve the least squares problem
    I, residuals, _, _ = np.linalg.lstsq(Z, -V, rcond=0.01)

    return I

def calculate_current_distribution_aux_sources_point_source(data_contour: np.
↪ndarray, data_aux: np.ndarray, x_source: float, y_source: float) -> np.
↪ndarray:
    """
    Calculate the current distribution for auxiliary sources.

    Args:
        data_contour: Contour data.
        data_aux: Auxiliary sources data.
        x_source: x-coordinate of the point source (required if source_type is␣
↪"point_source").
```

```
        y_source: y-coordinate of the point source (required if source_type is␣
↪"point_source").

    Returns:
        ndarray: Current distribution for auxiliary sources.
    """
    k = 2 * np.pi  # Wavenumber
    M = len(data_contour)  # Number of segments
    N = len(data_aux)  # Number of auxiliary sources
    Z = np.zeros((M, N), dtype=complex)  # Impedance matrix
    V = np.zeros((M, 1), dtype=complex)  # Excitation vector

    # Calculate impedance matrix
    for l in range(N):
        xm, ym = data_aux[l, 0], data_aux[l, 1]
        for m in range(M):
            xl, yl = data_contour[m, 0], data_contour[m, 1]
            rml = np.sqrt((xm - xl)**2 + (ym - yl)**2)
            Z[m, l] += hankel2(0, k * rml)

    for i in range(M):
        xm, ym = data_contour[i, 0], data_contour[i, 1]
        r_source = np.sqrt((xm - x_source)**2 + (ym - y_source)**2)
        V[i] = hankel2(0, k*r_source)

    # Solve the least squares problem
    I, residuals, _, _ = np.linalg.lstsq(Z, -V, rcond=0.01)

    return I
```

[ ]:
```
def scattered_field(data: np.ndarray, I: np.ndarray, xn_grid: np.ndarray,␣
↪yn_grid: np.ndarray) -> ndarray[Any, dtype[Any]]:
    """
    Calculate the scattered field at observation points.

    Args:
        data: Auxiliary sources data.
        I: Current distribution.
        xn_grid: x-coordinates of observation points.
        yn_grid: y-coordinates of observation points.

    Returns:
        ndarray: Scattered field at observation points.
    """
    k = 2 * np.pi  # Wavenumber
    N = len(data)
    M = len(xn_grid)
```

```
    e_scat = np.zeros((M, M), dtype=complex)  # Initialize scattered field␣
↪vector

    for i in range(M):
        xn = xn_grid[i]
        yn = yn_grid[i]
        for j in range(N):
            xm = data[j, 0]  # x-coordinate of the point on the contour
            ym = data[j, 1]  # y-coordinate of the point on the contour
            r = np.sqrt((xn - xm)**2 + (yn - ym)**2)
            e_scat[i] += hankel2(0, k * r) * I[j,0]
    return e_scat

def incident_field(xn_grid: np.ndarray, yn_grid: np.ndarray, phi_inc: float) ->␣
↪ndarray[Any, dtype[Any]]:
    """
    Calculate the incident field at observation points.

    Args:
        xn_grid: x-coordinates of observation points.
        yn_grid: y-coordinates of observation points.
        phi_inc: Angle of incidence.

    Returns:
        ndarray: Incident field at observation points.
    """
    k = 2 * np.pi
    N = len(xn_grid)
    phi_inc = np.deg2rad(phi_inc)  # Angle of incidence in radians
    e_inc = np.zeros((N, N), dtype=complex)  # Initialize incident field vector

    for i in range(N):
        xn = xn_grid[i]
        yn = yn_grid[i]
        e_inc[i] = np.exp(1j * k * (xn * np.cos(phi_inc) + yn * np.
↪sin(phi_inc)))
    return e_inc

def scattered_field_obs(data: np.ndarray, I: np.ndarray, xn_grid: np.ndarray,␣
↪yn_grid: np.ndarray) -> ndarray[Any, dtype[Any]]:
    """
    Calculate the scattered field at observation points.

    Args:
        data: Auxiliary sources data.
        I: Current distribution.
        xn_grid: x-coordinates of observation points.
```

```
        yn_grid: y-coordinates of observation points.

    Returns:
        ndarray: Scattered field at observation points.
    """
    k = 2 * np.pi  # Wavenumber
    N = len(xn_grid)
    M = len(data)
    e_scat = np.zeros((N,1), dtype=complex)  # Initialize scattered field vector

    for i in range(N):
        xn = xn_grid[i]
        yn = yn_grid[i]
        for j in range(M):
            xm = data[j, 0]  # x-coordinate of the point on the contour
            ym = data[j, 1]  # y-coordinate of the point on the contour
            r = np.sqrt((xn - xm)**2 + (yn - ym)**2)
            e_scat[i,0] += hankel2(0, k * r) * I[j,0]
    return e_scat

def incident_field_obs(xn_grid: np.ndarray, yn_grid: np.ndarray, phi_inc:
 ↪float) -> ndarray[Any, dtype[Any]]:
    """
    Calculate the incident field at observation points.

    Args:
        xn_grid: x-coordinates of observation points.
        yn_grid: y-coordinates of observation points.
        phi_inc: Angle of incidence.

    Returns:
        ndarray: Incident field at observation points.
    """
    k = 2 * np.pi
    N = len(xn_grid)
    phi_inc = np.deg2rad(phi_inc)  # Angle of incidence in radians
    e_inc = np.zeros((N, 1), dtype=complex)  # Initialize incident field vector

    for i in range(N):
        xn = xn_grid[i]
        yn = yn_grid[i]
        e_inc[i] = np.exp(1j * k * (xn * np.cos(phi_inc) + yn * np.
 ↪sin(phi_inc)))
    return e_inc

# Define an incident field which is generated by a point source
```

```python
def incident_field_point_source(xn_grid: np.ndarray, yn_grid: np.ndarray,␣
↪x_source: float, y_source: float) -> ndarray[Any, dtype[Any]]:
    """
    Calculate the incident field at observation points.

    Args:
        xn_grid: x-coordinates of observation points.
        yn_grid: y-coordinates of observation points.
        x_source: x-coordinate of the point source.
        y_source: y-coordinate of the point source.

    Returns:
        ndarray: Incident field at observation points.
    """
    k = 2 * np.pi  # Wavenumber
    N = len(xn_grid)
    e_inc = np.zeros((N, N), dtype=complex)  # Initialize incident field vector

    for i in range(N):
        xn = xn_grid[i]
        yn = yn_grid[i]
        r = np.sqrt((xn - x_source)**2 + (yn - y_source)**2)
        e_inc[i] = hankel2(0, k * r)
    return e_inc
```

0.2 Inverse scattering

[ ]:
```python
def cv_inverse_scattering(data_obs: np.ndarray, data_aux: np.ndarray,␣
↪scattered_field: np.ndarray, percentage: float = 1e-10) -> np.ndarray:
    """
    Computes the coefficient vector (cv) for inverse scattering.

    Args:
        data_obs (ndarray): Array containing coordinates of observable points.
        data_aux (ndarray): Array containing coordinates of auxiliary sources.
        scattered_field (ndarray): Scattered field.
        percentage (float): Percentage of the maximum singular value to␣
↪determine J.

    Returns:
        ndarray: Coefficient vector.
        ndarray: Truncated impedance matrix.
    """
    M = len(data_obs)  # Number of observable points
    N = len(data_aux)  # Number of auxiliary sources
    k = 2 * np.pi  # Wavenumber
```

```
    Z = np.zeros((M, N), dtype=complex)

    for l in range(N):
        xm = data_aux[l, 0]
        ym = data_aux[l, 1]
        for m in range(M):
            xl = data_obs[m, 0]
            yl = data_obs[m, 1]
            rml = np.sqrt((xm - xl)**2 + (ym - yl)**2)
            Z[m,l] += hankel2(0, k * rml)
    # Solve the least squares problem
    cv, residuals, _, _ = np.linalg.lstsq(Z, scattered_field, rcond=1e-3)

    return cv, Z
```

[ ]:
```
def inverse_scattered_field(data: np.ndarray, I: np.ndarray, xn_grid: np.
 ↪ndarray, yn_grid: np.ndarray) -> ndarray[Any, dtype[Any]]:
    """
    Calculates the inverse scattered field.

    Args:
        data (ndarray): Array containing coordinates of points on the contour.
        I (ndarray): Incident field vector.
        xn_grid (ndarray): x-coordinates grid.
        yn_grid (ndarray): y-coordinates grid.

    Returns:
        ndarray: Inverse scattered field vector.
    """
    k = 2 * np.pi  # Wavenumber
    N = len(data)
    M = len(xn_grid)
    e_inv_scat = np.zeros((M, M), dtype=complex)  # Initialize scattered field␣
 ↪vector
    for i in range(M):
        xn = xn_grid[i]
        yn = yn_grid[i]
        for j in range(N):
            xm = data[j, 0]  # x-coordinate of the point on the contour
            ym = data[j, 1]  # y-coordinate of the point on the contour
            r = np.sqrt((xn - xm)**2 + (yn - ym)**2)
            e_inv_scat[i] += hankel2(0, k * r) * I[j]
    return e_inv_scat

def inverse_scattered_field_obs(data: np.ndarray, I: np.ndarray, xn_grid: np.
 ↪ndarray, yn_grid: np.ndarray) -> ndarray[Any, dtype[Any]]:
    """
```

```
    Calculate the scattered field at observation points.

    Args:
        data: Auxiliary sources data.
        I: Current distribution.
        xn_grid: x-coordinates of observation points.
        yn_grid: y-coordinates of observation points.

    Returns:
        ndarray: Scattered field at observation points.
    """
    k = 2 * np.pi  # Wavenumber
    M = len(xn_grid)
    N = len(data)
    e_inv_scat = np.zeros((M,1), dtype=complex)  # Initialize scattered field␣
↪vector

    for i in range(M):
        xn = xn_grid[i]
        yn = yn_grid[i]
        for j in range(N):
            xm = data[j, 0]  # x-coordinate of the point on the contour
            ym = data[j, 1]  # y-coordinate of the point on the contour
            r = np.sqrt((xn - xm)**2 + (yn - ym)**2)
            e_inv_scat[i] += hankel2(0, k * r) * I[j]
    return e_inv_scat
```

[ ]:
```
def add_noise(x, percentage):
    """Add noise to data.

    The noise is implemented as a complex number with a fixed magnitude
    and random phase.

    Parameters
    ----------
    x : array_like
        Data to receive noise.
    percentage : float
        Noise level in percentage.

    Returns
    -------
    xd : array_like
        Corrupted data with noise.
    """
    # Generate random phase
    phase = np.reshape(2 * np.pi * rnd.rand(x.size), x.shape)
```

```
    # Calculate noise magnitude
    mod = percentage / 100 * np.abs(x)
    # Add noise to the data
    xd = x + mod * np.cos(phase) + 1j * mod * np.sin(phase)
    return xd

def moving_average(data, window_size):
    cumsum = np.cumsum(data)
    return (cumsum[window_size:] - cumsum[:-window_size]) / window_size
```

0.2.1 Circle

[ ]:
```
a_contour = 3
a_obs = a_contour + 3
a_aux = a_contour * 0.8
a_aux_forw = a_contour * 0.9

M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_contour = circle_data(a_contour, M_points)
data_obs = circle_data(a_obs, M)
data_aux = circle_data(a_aux, N)
# Take only the points on the right side of the circle
# data_obs = data_obs[data_obs[:, 0] > 0]

x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [0, 180]
J = len(phi_i)

grid_size = 300  # Size of the grid
x_min, x_max = -6, 6  # X-coordinate range
y_min, y_max = -6, 6  # Y-coordinate range

x_grid, y_grid = np.meshgrid(np.linspace(x_min, x_max, grid_size),
                             np.linspace(y_min, y_max, grid_size))

e_inv_scat = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_inc = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_total = np.zeros((grid_size, grid_size), dtype=complex)

for j in range(J):
```

```
    I = calculate_current_distribution_aux_sources(data_contour, data_aux_forw,␣
 ↪phi_i[j])
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field(data_aux, cv, x_grid, y_grid)
    e_inv_inc = incident_field(x_grid, y_grid, phi_i[j])
    e_inv_total += e_inv_inc + e_inv_scat

e_inv_scat_log = np.log10(np.abs(e_inv_total))
indices = np.where(x_grid[grid_size//2] > a_aux)[0]
x_filtered = x_grid[grid_size//2, indices]
e_filtered = e_inv_scat_log[grid_size//2, indices]

# Find the first minimum
min_index = None
min_value = np.inf
for i in indices:
    if e_inv_scat_log[grid_size//2, i] < min_value:
        min_value = e_inv_scat_log[grid_size//2, i]
        min_index = i

x_coordinate = x_grid[grid_size//2, min_index]

print("First minimum index:", min_index)
print("First minimum value:", min_value)
print("X-coordinate of the minimum value:", x_coordinate)
```

[ ]:
```
# Add noise to the scattered field and calculate SNR for only one incident angle
percentage = np.linspace(1, 20, 100)
phi_i = 0
snr_list_circle = []
for i in range(len(percentage)):
    I = calculate_current_distribution_aux_sources(data_contour, data_aux_forw,␣
 ↪phi_i)
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_contour[:,0],␣
 ↪data_contour[:,1])

    e_scat_noisy = add_noise(e_scat, percentage[i])
    cv_noisy, Z_noisy = cv_inverse_scattering(data_obs, data_aux, e_scat_noisy)
    e_inv_scat_noisy = inverse_scattered_field_obs(data_aux, cv_noisy,␣
 ↪data_contour[:,0], data_contour[:,1])
    snr = 10 * np.log10(np.sum(np.abs(e_inv_scat)**2) / np.sum(np.
 ↪abs(e_inv_scat - e_inv_scat_noisy)**2))
    snr_list_circle.append(snr)
# Denoise the SNR values using a moving average filter
```

```
window_size = 5
snr_list_circle = moving_average(snr_list_circle, window_size)
```

[ ]:
```
a_obs = a_contour + 10
a_aux_circ = np.linspace(0.1, 0.8, 30)*a_contour
a_aux_forw = a_contour * 0.9

M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
data_contour = circle_data(a_contour, M_points)

x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [0, 180]
J = len(phi_i)

mean_error_lst_circle = []
for i in range(len(a_aux_circ)):
    e_inv_scat = np.zeros((M_points, 1), dtype=complex)
    e_inv_inc = np.zeros((M_points, 1), dtype=complex)
    e_inv_total = np.zeros((M_points, 1), dtype=complex)
    data_aux = circle_data(a_aux_circ[i], N)
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_contour,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
        e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_contour[:
 ↪,0], data_contour[:,1])
        e_inv_inc = incident_field_obs(data_contour[:,0], data_contour[:,1],␣
 ↪phi_i[j])
        e_inv_total += e_inv_inc + e_inv_scat
    mean_error = np.mean(np.abs(e_inv_total)**2)
    mean_error_lst_circle.append(mean_error)
mean_error_lst_circle = np.array(mean_error_lst_circle)/np.
 ↪max(mean_error_lst_circle)
```

[ ]:
```
# Condition number depending on the number of auxiliary sources
a_obs = a_contour + 3
a_aux = a_contour * 0.8
a_aux_forw = a_contour * 0.9
```

```
M = 100
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
N = np.arange(10, 250, 10)
phi_i = [0]
J = len(phi_i)

cond_list_circle = []
for i in range(len(N)):
    data_aux = circle_data(a_aux, N[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_contour,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_circle.append(cond)
```

[ ]:

```
# Condition number depending on the number of observation points
a_obs = a_contour + 3
a_aux = a_contour * 0.8
a_aux_forw = a_contour * 0.9

M = np.arange(10, 250, 10)
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)


phi_i = [0]
J = len(phi_i)

cond_list_circle_M = []
for i in range(len(M)):
    data_obs = circle_data(a_obs, M[i])
    data_aux = circle_data(a_aux, N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
```

```
        I = calculate_current_distribution_aux_sources(data_contour,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_circle_M.append(cond)
```

[ ]:
```
# Condition number depending on the radius of the auxiliary curve
a_obs = a_contour + 10
a_aux = np.linspace(0.1, 0.8, 30)*a_contour
a_obs = a_contour + 3
a_aux_forw = a_contour * 0.9

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
phi_i = [0]
J = len(phi_i)

cond_list_circle_auxil = []
for i in range(len(a_aux)):
    data_aux = circle_data(a_aux[i], N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_contour,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_circle_auxil.append(cond)
```

[ ]:
```
# Compute the conditioning number for different observation radii on the circle
a_start = a_contour + 1
a_end = 5 * a_contour
a_obs_circ = np.linspace(a_start, a_end, 100)
a_aux = a_contour*0.8
a_aux_forw = a_contour * 0.9

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)
```

```
phi_i = [0]
J = len(phi_i)

cond_list_obs_circle = []
for i in range(len(a_obs_circ)):
    data_obs = circle_data(a_obs_circ[i], M)
    data_aux = circle_data(a_aux, N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_contour, data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_obs_circle.append(cond)
```

0.2.2 Kite

[ ]:

```
a_kite = 4
a_obs = a_kite + 3
a_aux = a_kite * 0.55
a_aux_forw = a_kite * 0.6

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw, cx = -0.5, cy = 0.0)
data_kite = kite_data(a_kite, M_points)

data_obs = circle_data(a_obs, M)
data_aux = circle_data(a_aux, N, cx = -1.0, cy = 0.0)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [0, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300, 320, 340]
J = len(phi_i)

grid_size = 300  # Size of the grid
x_min, x_max = -6, 6  # X-coordinate range
y_min, y_max = -6, 6  # Y-coordinate range

x_grid, y_grid = np.meshgrid(np.linspace(x_min, x_max, grid_size),
                             np.linspace(y_min, y_max, grid_size))
```

```python
e_inv_scat = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_inc = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_total = np.zeros((grid_size, grid_size), dtype=complex)

e_inv_scat_list = []
e_inv_total_list = []
for j in range(J):
    I = calculate_current_distribution_aux_sources(data_kite, data_aux_forw, ␣
 ↪phi_i[j])
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)

    e_inv_scat = inverse_scattered_field(data_aux, cv, x_grid, y_grid)
    e_inv_inc = incident_field(x_grid, y_grid, phi_i[j])
    e_inv_total = e_inv_inc + e_inv_scat
    e_inv_scat_list.append(e_inv_scat)
    e_inv_total_list.append(e_inv_total)
```

[ ]:
```python
# Add noise to the scattered field and calculate SNR for only one incident angle
percentage = np.linspace(1, 20, 100)
phi_i = 0
snr_list_kite = []
for i in range(len(percentage)):
    I = calculate_current_distribution_aux_sources(data_kite, data_aux_forw, ␣
 ↪phi_i)
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_kite[:,0], ␣
 ↪data_kite[:,1])

    e_scat_noisy = add_noise(e_scat, percentage[i])
    cv_noisy, Z_noisy = cv_inverse_scattering(data_obs, data_aux, e_scat_noisy)
    e_inv_scat_noisy = inverse_scattered_field_obs(data_aux, cv_noisy, ␣
 ↪data_kite[:,0], data_kite[:,1])
    # Calculate SNR
    snr = 10 * np.log10(np.sum(np.abs(e_inv_scat)**2) / np.sum(np.
 ↪abs(e_inv_scat - e_inv_scat_noisy)**2))
    snr_list_kite.append(snr)
# Denoise the SNR values using a moving average filter
window_size = 5
snr_list_kite = moving_average(snr_list_kite, window_size)
```

[ ]:
```python
a_obs = a_kite + 3
a_aux_kite = np.linspace(0.1, 0.6, 30)*a_kite
a_aux_forw = a_kite*0.6
```

```
M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw, cx = -0.5, cy = 0.0)
data_kite = kite_data(a_kite, M_points)
data_obs = circle_data(a_obs, M)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [0, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280,␣
 ↪300, 320, 340]
J = len(phi_i)
mean_error_lst_kite = []

for i in range(len(a_aux_kite)):
    e_inv_scat = np.zeros((M_points, 1), dtype=complex)
    e_inv_inc = np.zeros((M_points, 1), dtype=complex)
    e_inv_total = np.zeros((M_points, 1), dtype=complex)
    data_aux = circle_data(a_aux_kite[i], N, cx = -1.0, cy = 0.0)
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_kite,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
        e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_kite[:,0],␣
 ↪data_kite[:,1])
        e_inv_inc = incident_field_obs(data_kite[:,0], data_kite[:,1], phi_i[j])
        e_inv_total += e_inv_inc + e_inv_scat
    mean_error = np.mean(np.abs(e_inv_total)**2)
    mean_error_lst_kite.append(mean_error)
mean_error_lst_kite = np.array(mean_error_lst_kite)/np.max(mean_error_lst_kite)
```

[ ]:

```
# Condition number depending on the number of auxiliary sources
a_obs = a_kite + 3
a_aux = a_kite * 0.6
a_aux_forw = a_kite * 0.6

M = 100
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw, cx = -0.5, cy = 0.0)
data_kite = kite_data(a_kite, M_points)
data_obs = circle_data(a_obs, M)
```

```
N = np.arange(10, 250, 10)
phi_i = [0]
J = len(phi_i)

cond_list_kite = []
for i in range(len(N)):
    data_aux = circle_data(a_aux, N[i], cx = -1.0, cy = 0.0)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_kite,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_kite.append(cond)
```

[ ]:
```
a_obs = a_kite + 3
a_aux = a_kite * 0.6
a_aux_forw = a_kite * 0.6

M_points = 400
N_forw = 150
M = np.arange(10, 250, 10)
N = 50
data_aux_forw = circle_data(a_aux_forw, N_forw, cx = -0.5, cy = 0.0)
data_aux = circle_data(a_aux, N, cx = -1.0, cy = 0.0)
data_kite = kite_data(a_kite, M_points)

phi_i = [0]
J = len(phi_i)

cond_list_kite_M = []
for i in range(len(M)):
    data_obs = circle_data(a_obs, M[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_kite,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_kite_M.append(cond)
```

[ ]:
```
a_obs = a_contour + 3
a_aux = np.linspace(0.1, 0.6, 30)*a_kite
a_aux_forw = a_kite*0.6

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw, cx = -0.5, cy = 0.0)
data_obs = circle_data(a_obs, M)
data_kite = kite_data(a_kite, M_points)
phi_i = [0]
J = len(phi_i)

cond_list_kite_auxil = []
for i in range(len(a_aux)):
    data_aux = circle_data(a_aux[i], N, cx = -1.0, cy = 0.0)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_kite,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_kite_auxil.append(cond)
```

[ ]:
```
a_start = a_kite + 1
a_end = 5 * a_kite
a_obs_kite = np.linspace(a_start, a_end, 100)
a_aux = a_kite*0.6
a_aux_forw = a_kite * 0.6

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw, cx = -0.5, cy = 0.0)
data_kite = kite_data(a_kite, M_points)
phi_i = [0]
J = len(phi_i)

cond_list_obs_kite = []
for i in range(len(a_obs_kite)):
    data_obs = circle_data(a_obs_kite[i], M)
    data_aux = circle_data(a_aux, N, cx = -1.0, cy = 0.0)
    x_obs = data_obs[:, 0]
```

```
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_kite,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_obs_kite.append(cond)
```

0.2.3 Rectangle

```
[ ]: a = 7  # Length of the rectangle along the x-axis
b = 6  # Length of the rectangle along the y-axis
cx = 0  # x-coordinate of the center of the rectangle
cy = 0  # y-coordinate of the center of the rectangle
if a < b:
    a_aux = a*0.8/2
    a_aux_forw = a*0.9/2
    a_obs = b + 3
else:
    a_aux = b*0.8/2
    a_aux_forw = b*0.9/2
    a_obs = a + 3

M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_rectangle = rectangle_data(a, b, M_points, cx = 0.0, cy = 0.0)
data_obs = circle_data(a_obs, M)
data_aux = circle_data(a_aux, N, cx = 0.0, cy = 0.0)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [60, 120, 240, 300]
J = len(phi_i)

grid_size = 300  # Size of the grid
x_min, x_max = -6, 6  # X-coordinate range
y_min, y_max = -6, 6  # Y-coordinate range

x_grid, y_grid = np.meshgrid(np.linspace(x_min, x_max, grid_size),
                             np.linspace(y_min, y_max, grid_size))

e_inv_scat = np.zeros((grid_size, grid_size), dtype=complex)
```

```
e_inv_inc = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_total = np.zeros((grid_size, grid_size), dtype=complex)

e_inv_total_list = []
e_inv_scat_list = []
for j in range(J):
    I = calculate_current_distribution_aux_sources(data_rectangle,␣
↪data_aux_forw, phi_i[j])
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)

    e_inv_scat = inverse_scattered_field(data_aux, cv, x_grid, y_grid)
    e_inv_inc = incident_field(x_grid, y_grid, phi_i[j])
    e_inv_total = e_inv_inc + e_inv_scat
    e_inv_total_list.append(e_inv_total)
    e_inv_scat_list.append(e_inv_scat)
```

[ ]:
```
percentage = np.linspace(1, 20, 100)
phi_i = 0
snr_list_rect = []
for i in range(len(percentage)):
    I = calculate_current_distribution_aux_sources(data_rectangle,␣
↪data_aux_forw, phi_i)
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_rectangle[:,0],␣
↪data_rectangle[:,1])

    e_scat_noisy = add_noise(e_scat, percentage[i])
    cv_noisy, Z_noisy = cv_inverse_scattering(data_obs, data_aux, e_scat_noisy)
    e_inv_scat_noisy = inverse_scattered_field_obs(data_aux, cv_noisy,␣
↪data_rectangle[:,0], data_rectangle[:,1])
    # Calculate SNR
    snr = 10 * np.log10(np.sum(np.abs(e_inv_scat)**2) / np.sum(np.
↪abs(e_inv_scat - e_inv_scat_noisy)**2))
    snr_list_rect.append(snr)
# Denoise the SNR values using a moving average filter
window_size = 5
snr_list_rect = moving_average(snr_list_rect, window_size)
```

[ ]:
```
a_obs = a + 3
a_aux_rect = np.linspace(0.1, 0.8, 30)*(b/2)
a_obs = a + 3
if a < b:
    a_aux_forw = a*0.9/2
else:
```

```
    a_aux_forw = b*0.9/2

M = 50
N = 100
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, N)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [0, 60, 120, 240, 300]
J = len(phi_i)

mean_error_lst_rect = []
for i in range(len(a_aux_rect)):
    e_inv_scat = np.zeros((M_points, 1), dtype=complex)
    e_inv_inc = np.zeros((M_points, 1), dtype=complex)
    e_inv_total = np.zeros((M_points, 1), dtype=complex)
    data_aux = circle_data(a_aux_rect[i], N)
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_rectangle,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
        e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_rectangle[:
↪,0], data_rectangle[:,1])
        e_inv_inc = incident_field_obs(data_rectangle[:,0], data_rectangle[:
↪,1], phi_i[j])
        e_inv_total += e_inv_inc + e_inv_scat
    mean_error = np.mean(np.abs(e_inv_total)**2)
    mean_error_lst_rect.append(mean_error)
mean_error_lst_rect = np.array(mean_error_lst_rect)/np.max(mean_error_lst_rect)
```

[ ]:
```
a_obs = a + 3
if a < b:
    a_aux = a*0.8/2
    a_aux_forw = a*0.9/2
else:
    a_aux = b*0.8/2
    a_aux_forw = b*0.9/2

M = 100
M_points = 400
N_forw = 150
```

```
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_rectangle = rectangle_data(a, b, M_points, cx = 0.0, cy = 0.0)
data_obs = circle_data(a_obs, M)
N = np.arange(10, 250, 10)
phi_i = [0]
J = len(phi_i)

cond_list_rect = []
for i in range(len(N)):
    data_aux = circle_data(a_aux, N[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_rectangle,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_rect.append(cond)
```

[ ]:
```
a_obs = a + 3
if a < b:
    a_aux = a*0.8/2
    a_aux_forw = a*0.9/2
else:
    a_aux = b*0.8/2
    a_aux_forw = b*0.9/2

M_points = 400
N_forw = 150
M = np.arange(10, 250, 10)
N = 50
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_aux = circle_data(a_aux, N)
data_rectangle = rectangle_data(a, b, M_points, cx = 0.0, cy = 0.0)

phi_i = [0]
J = len(phi_i)

cond_list_rect_M = []
for i in range(len(M)):
    data_obs = circle_data(a_obs, M[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_rectangle,␣
↪data_aux_forw, phi_i[j])
```

```
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_rect_M.append(cond)
```

[ ]:
```
a_obs = a + 3
a_aux = np.linspace(0.1, 0.8, 30)*(b/2)
if a < b:
    a_aux_forw = a*0.9/2
else:
    a_aux_forw = b*0.9/2

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
data_rectangle = rectangle_data(a, b, M_points, cx = 0.0, cy = 0.0)
phi_i = [0]
J = len(phi_i)

cond_list_rect_auxil = []
for i in range(len(a_aux)):
    data_aux = circle_data(a_aux[i], N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_rectangle,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_rect_auxil.append(cond)
```

[ ]:
```
if a < b:
    a_aux = a*0.8/2
    a_aux_forw = a*0.9/2
    b_start = b + 1
    b_end = 5 * b
    a_obs_rect = np.linspace(b_start, b_end, 100)
else:
    a_start = a + 1
    a_end = 5 * a
    a_obs_rect = np.linspace(a_start, a_end, 100)
    a_aux = b*0.8/2
    a_aux_forw = b*0.9/2
```

```
M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw, cx=0.0, cy=0.0)
data_rectangle = rectangle_data(a, b, M_points, cx = 0.0, cy = 0.0)
phi_i = [0]
J = len(phi_i)

cond_list_obs_rect = []
for i in range(len(a_obs_rect)):
    data_obs = circle_data(a_obs_rect[i], M)
    data_aux = circle_data(a_aux, N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_rectangle,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_obs_rect.append(cond)
```

0.2.4 Ellipse

[ ]:
```
x_ellypse = 3
y_ellypse = 5
a_obs = y_ellypse + 3
a_aux_forw = 0.9 * x_ellypse
if x_ellypse < y_ellypse:
    a_aux_ellypse = 0.8*(x_ellypse)
else:
    a_aux_ellypse = 0.8*(y_ellypse)

M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
data_aux = circle_data(a_aux_ellypse, N)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]
ellypse_data = data_ellypse(x_ellypse, y_ellypse, M_points)
phi_i = [0, 60, 90, 120, 180, 240, 300]
```

```python
J = len(phi_i)

grid_size = 300  # Size of the grid
x_min, x_max = -6, 6  # X-coordinate range
y_min, y_max = -6, 6  # Y-coordinate range

x_grid, y_grid = np.meshgrid(np.linspace(x_min, x_max, grid_size),
                                np.linspace(y_min, y_max, grid_size))

e_inv_scat = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_inc = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_total = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_total_list = []

for j in range(J):
    x_source = 0
    y_source = y_ellypse + 3
    I = calculate_current_distribution_aux_sources_point_source(ellypse_data,
 ↪data_aux_forw, x_source, y_source)
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field(data_aux, cv, x_grid, y_grid)
    e_inv_inc = incident_field(x_grid, y_grid, phi_i[j])
    e_inv_total = e_inv_inc + e_inv_scat
    e_inv_total_list.append(e_inv_total)
```

[ ]:
```python
a_obs = y_ellypse + 3
a_aux_forw = 0.9 * x_ellypse
if x_ellypse < y_ellypse:
    a_aux_ellypse = np.linspace(0.1, 0.8, 30)*(x_ellypse)
else:
    a_aux_ellypse = np.linspace(0.1, 0.8, 30)*(y_ellypse)

M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]
ellypse_data = data_ellypse(x_ellypse, y_ellypse, M_points)
phi_i = [0, 60, 90, 120, 180, 240, 300]
J = len(phi_i)

mean_error_lst_ellypse = []
```

```python
for i in range(len(a_aux_ellypse)):
    e_inv_scat = np.zeros((M_points, 1), dtype=complex)
    e_inv_inc = np.zeros((M_points, 1), dtype=complex)
    e_inv_total = np.zeros((M_points, 1), dtype=complex)
    data_aux = circle_data(a_aux_ellypse[i], N)
    for j in range(J):
        I = calculate_current_distribution_aux_sources(ellypse_data,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
        e_inv_scat = inverse_scattered_field_obs(data_aux, cv, ellypse_data[:
 ↪,0], ellypse_data[:,1])
        e_inv_inc = incident_field_obs(ellypse_data[:,0], ellypse_data[:,1],␣
 ↪phi_i[j])
        e_inv_total += e_inv_inc + e_inv_scat
    mean_error = np.mean(np.abs(e_inv_total)**2)
    mean_error_lst_ellypse.append(mean_error)
mean_error_lst_ellypse = np.array(mean_error_lst_ellypse)/np.
 ↪max(mean_error_lst_ellypse)
```

[ ]:
```python
percentage = np.linspace(1, 20, 100)
phi_i = 0
snr_list_ellypse = []
for i in range(len(percentage)):
    I = calculate_current_distribution_aux_sources(ellypse_data, data_aux_forw,␣
 ↪phi_i)
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field_obs(data_aux, cv, ellypse_data[:,0],␣
 ↪ellypse_data[:,1])

    e_scat_noisy = add_noise(e_scat, percentage[i])
    cv_noisy, Z_noisy = cv_inverse_scattering(data_obs, data_aux, e_scat_noisy)
    e_inv_scat_noisy = inverse_scattered_field_obs(data_aux, cv_noisy,␣
 ↪ellypse_data[:,0], ellypse_data[:,1])
    # Calculate SNR
    snr = 10 * np.log10(np.sum(np.abs(e_inv_scat)**2) / np.sum(np.
 ↪abs(e_inv_scat - e_inv_scat_noisy)**2))
    snr_list_ellypse.append(snr)
# Denoise the SNR values using a moving average filter
window_size = 5
snr_list_ellypse = moving_average(snr_list_ellypse, window_size)
```

[ ]:
```python
a_obs = y_ellypse + 3
a_aux_forw = 0.9 * x_ellypse
if x_ellypse < y_ellypse:
```

```
    a_aux = 0.8 * x_ellypse
else:
    a_aux = 0.8 * y_ellypse

M = 100
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, M)
ellypse_data = data_ellypse(x_ellypse, y_ellypse, M_points)
N = np.arange(10, 250, 10)
phi_i = [0]
J = len(phi_i)

cond_list_ellypse = []
for i in range(len(N)):
    data_aux = circle_data(a_aux, N[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(ellypse_data,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_ellypse.append(cond)
```

[ ]:
```
a_obs = y_ellypse + 3
a_aux_forw = 0.9 * x_ellypse
if x_ellypse < y_ellypse:
    a_aux = 0.8 * x_ellypse
else:
    a_aux = 0.8 * y_ellypse

M_points = 400
N_forw = 150
M = np.arange(10, 250, 10)
N = 50
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_aux = circle_data(a_aux, N)
ellypse_data = data_ellypse(x_ellypse, y_ellypse, M_points)

phi_i = [0]
J = len(phi_i)

cond_list_ellypse_M = []
for i in range(len(M)):
```

```
    data_obs = circle_data(a_obs, M[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(ellypse_data,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_ellypse_M.append(cond)
```

[ ]:
```
a_aux_forw = 0.9 * x_ellypse
if x_ellypse < y_ellypse:
    a_obs = y_ellypse + 3
    a_aux = np.linspace(0.1, 0.8, 30)*(x_ellypse)
else:
    a_obs = x_ellypse + 3
    a_aux = np.linspace(0.1, 0.8, 30)*(y_ellypse)

M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
ellypse_data = data_ellypse(x_ellypse, y_ellypse, M_points)
data_obs = circle_data(a_obs, M)
phi_i = [0]
J = len(phi_i)

cond_list_ellypse_auxil = []
for i in range(len(a_aux)):
    data_aux = circle_data(a_aux[i], N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(ellypse_data,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_ellypse_auxil.append(cond)
```

[ ]:
```
a_aux_forw = 0.9 * x_ellypse
if x_ellypse < y_ellypse:
    a_start = y_ellypse + 1
    a_end = 5 * y_ellypse
```

```python
    a_obs_ellypse = np.linspace(a_start, a_end, 100)
    a_aux = 0.8*(x_ellypse)
else:
    a_start = x_ellypse + 1
    a_end = 5 * x_ellypse
    a_obs_ellypse = np.linspace(a_start, a_end, 100)
    a_aux = 0.8*(y_ellypse)

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)
ellypse_data = data_ellypse(x_ellypse, y_ellypse, M_points)
phi_i = [0]
J = len(phi_i)

cond_list_obs_ellypse = []
for i in range(len(a_obs_ellypse)):
    data_obs = circle_data(a_obs_ellypse[i], M)
    data_aux = circle_data(a_aux, N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(ellypse_data,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_obs_ellypse.append(cond)
```

0.2.5 Square

[ ]:
```python
a_sq = 6  # Length of the rectangle along the x-axis
b_sq = 6  # Length of the rectangle along the y-axis
cx = 0  # x-coordinate of the center of the rectangle
cy = 0  # y-coordinate of the center of the rectangle
a_obs = a_sq + 3
a_aux = a_sq * 0.8/2
a_aux_forw = a_sq * 0.9/2
M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_square = rectangle_data(a_sq, b_sq, M_points, cx = 0.0, cy = 0.0)
```

```
data_obs = circle_data(a_obs, M)
data_aux = circle_data(a_aux, N, cx = 0.0, cy = 0.0)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [60, 120, 240, 300]
J = len(phi_i)

grid_size = 300  # Size of the grid
x_min, x_max = -6, 6  # X-coordinate range
y_min, y_max = -6, 6  # Y-coordinate range

x_grid, y_grid = np.meshgrid(np.linspace(x_min, x_max, grid_size),
                              np.linspace(y_min, y_max, grid_size))

e_inv_scat = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_inc = np.zeros((grid_size, grid_size), dtype=complex)
e_inv_total = np.zeros((grid_size, grid_size), dtype=complex)

e_inv_total_list = []
e_inv_scat_list = []
for j in range(J):
    I = calculate_current_distribution_aux_sources(data_square, data_aux_forw,␣
 ↪phi_i[j])
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)

    e_inv_scat = inverse_scattered_field(data_aux, cv, x_grid, y_grid)
    e_inv_inc = incident_field(x_grid, y_grid, phi_i[j])
    e_inv_total = e_inv_inc + e_inv_scat
    e_inv_total_list.append(e_inv_total)
    e_inv_scat_list.append(e_inv_scat)
```

[ ]:
```
percentage = np.linspace(1, 20, 100)
phi_i = 0
snr_list_sq = []
for i in range(len(percentage)):
    I = calculate_current_distribution_aux_sources(data_square, data_aux_forw,␣
 ↪phi_i)
    e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
    cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_square[:,0],␣
 ↪data_square[:,1])

    e_scat_noisy = add_noise(e_scat, percentage[i])
    cv_noisy, Z_noisy = cv_inverse_scattering(data_obs, data_aux, e_scat_noisy)
```

```
    e_inv_scat_noisy = inverse_scattered_field_obs(data_aux, cv_noisy,␣
↪data_square[:,0], data_square[:,1])
    # Calculate SNR
    snr = 10 * np.log10(np.sum(np.abs(e_inv_scat)**2) / np.sum(np.
↪abs(e_inv_scat - e_inv_scat_noisy)**2))
    snr_list_sq.append(snr)

# Denoise the SNR values using a moving average filter
window_size = 5
snr_list_sq = moving_average(snr_list_sq, window_size)
```

[ ]:
```
a_aux_sq = np.linspace(0.1, 0.8, 30)*(b_sq/2)
a_obs = a_sq + 10
a_aux_forw = a_sq*0.9/2

M = 50
N = 100
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_obs = circle_data(a_obs, N)
x_obs = data_obs[:, 0]
y_obs = data_obs[:, 1]

phi_i = [60, 120, 240, 300]
J = len(phi_i)

mean_error_lst_sq = []
for i in range(len(a_aux_sq)):
    e_inv_scat = np.zeros((M_points, 1), dtype=complex)
    e_inv_inc = np.zeros((M_points, 1), dtype=complex)
    e_inv_total = np.zeros((M_points, 1), dtype=complex)
    data_aux = circle_data(a_aux_sq[i], N)
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_square,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
        e_inv_scat = inverse_scattered_field_obs(data_aux, cv, data_square[:
↪,0], data_square[:,1])
        e_inv_inc = incident_field_obs(data_square[:,0], data_square[:,1],␣
↪phi_i[j])
        e_inv_total += e_inv_inc + e_inv_scat
    mean_error = np.mean(np.abs(e_inv_total)**2)
    mean_error_lst_sq.append(mean_error)
mean_error_lst_sq = np.array(mean_error_lst_sq)/np.max(mean_error_lst_sq)
```

[ ]:
```
a_obs = a_sq + 3
a_aux = a_sq * 0.8/2
a_aux_forw = a_sq * 0.9/2

M = 100
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_square = rectangle_data(a_sq, b_sq, M_points, cx = 0.0, cy = 0.0)
data_obs = circle_data(a_obs, M)
N = np.arange(10, 250, 10)
phi_i = [0]
J = len(phi_i)

cond_list_sq = []
for i in range(len(N)):
    data_aux = circle_data(a_aux, N[i])
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(ellypse_data,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_sq.append(cond)
```

[ ]:
```
a_obs = a_sq + 3
a_aux = a_sq * 0.8/2
a_aux_forw = a_sq * 0.9/2

M_points = 400
N_forw = 150
M = np.arange(10, 250, 10)
N = 50
data_aux_forw = circle_data(a_aux_forw, N_forw)
data_aux = circle_data(a_aux, N)
data_square = rectangle_data(a_sq, b_sq, M_points, cx = 0.0, cy = 0.0)

phi_i = [0]
J = len(phi_i)

cond_list_square_M = []
for i in range(len(M)):
    data_obs = circle_data(a_obs, M[i])
    x_obs = data_obs[:, 0]
```

```
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_square,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_square_M.append(cond)
```

[ ]:
```
a_aux = np.linspace(0.1, 0.8, 30)*(b_sq/2)
a_obs = a_sq + 3
a_aux_forw = a_sq*0.9/2
M = 100
N = 50
M_points = 400
N_forw = 150

data_aux_forw = circle_data(a_aux_forw, N_forw)
data_square = rectangle_data(a_sq, b_sq, M_points, cx = 0.0, cy = 0.0)
data_obs = circle_data(a_obs, M)
phi_i = [0]
J = len(phi_i)

cond_list_sq_auxil = []
for i in range(len(a_aux)):
    data_aux = circle_data(a_aux[i], N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_square,␣
↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_sq_auxil.append(cond)
```

[ ]:
```
a_start = a_sq + 1
a_end = 5 * a_sq
a_obs_sq = np.linspace(a_start, a_end, 100)
a_aux_forw = a_sq*0.9/2
a_aux = a_sq * 0.8/2

M = 100
N = 50
M_points = 400
N_forw = 150
data_aux_forw = circle_data(a_aux_forw, N_forw)
```

```
data_square = rectangle_data(a_sq, b_sq, M_points, cx = 0.0, cy = 0.0)
phi_i = [0]
J = len(phi_i)

cond_list_obs_sq = []
for i in range(len(a_obs_sq)):
    data_obs = circle_data(a_obs_sq[i], M)
    data_aux = circle_data(a_aux, N)
    x_obs = data_obs[:, 0]
    y_obs = data_obs[:, 1]
    for j in range(J):
        I = calculate_current_distribution_aux_sources(data_square,␣
 ↪data_aux_forw, phi_i[j])
        e_scat = scattered_field_obs(data_aux_forw, I, x_obs, y_obs)
        cv, Z = cv_inverse_scattering(data_obs, data_aux, e_scat)
    cond = np.linalg.cond(Z)
    cond_list_obs_sq.append(cond)
```

0.2.6 Evaluation

[ ]:
```
# Plot the SNR for different levels of noise
plt.figure()
plt.plot(percentage[:len(snr_list_circle)], snr_list_circle)
plt.plot(percentage[:len(snr_list_circle)], snr_list_kite)
plt.plot(percentage[:len(snr_list_circle)], snr_list_rect)
plt.plot(percentage[:len(snr_list_circle)], snr_list_ellypse)
plt.plot(percentage[:len(snr_list_circle)], snr_list_sq)
plt.legend([r'Circle', r'Kite', r'Rectangle', r'Ellypse', r'Square'],␣
 ↪fontsize=10)
plt.xlabel(r'Percentage of additve noise (%)', fontsize=15)
plt.ylabel(r'SNR (dB)', fontsize=15)
plt.title(r'SNR as a function of additive noise', fontsize=15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()
```

[ ]:
```
# Plot all mean errors of all geometries
plt.figure()
plt.plot(a_aux_circ/a_contour, mean_error_lst_circle)
plt.plot(a_aux_kite/a_contour, mean_error_lst_kite)
plt.plot(a_aux_rect/a_contour, mean_error_lst_rect)
plt.plot(a_aux_ellypse/a_contour, mean_error_lst_ellypse)
plt.plot(a_aux_sq/a_contour, mean_error_lst_sq)
plt.legend([r'Circle', r'Kite', r'Rectangle', r'Ellypse', r'Square'],␣
 ↪fontsize=10)
plt.xlabel(r'Distance of source points $R_s/R_\Omega$', fontsize=15)
```

```
plt.ylabel(r'MSE at PEC boundary $\langle|E^{t}_{z}/E_0|^2\rangle$',␣
 ↪fontsize=15)
plt.title(r'MSE of the total field along the contour $\partial \Omega$',␣
 ↪fontsize=15)
plt.xlim(0.1, 0.8)
plt.xticks(np.arange(0.1, 0.8, 0.1), fontsize=10)
plt.yticks(fontsize=10)
plt.show()
```

[ ]:
```
# Plot all condition numbers of all geometries
N = np.arange(10, 250, 10)
M = 100
plt.figure()
plt.plot(N, cond_list_circle)
plt.plot(N, cond_list_kite)
plt.plot(N, cond_list_rect)
plt.plot(N, cond_list_ellypse)
plt.plot(N, cond_list_sq)
plt.axvline(x = M, color='b',linestyle="--", label = r'$M^\prime$')
plt.legend([r'Circle', r'Kite', r'Rectangle', r'Ellypse', r'Square',␣
 ↪r'$M^\prime$'], fontsize=10)
plt.title('Conditionning number as a function\nof the number of auxiliary␣
 ↪sources', fontsize=15)
plt.xlabel(r'Number of auxiliary sources ($N^\prime$)', fontsize=15)
plt.ylabel(r'Conditionning number $\log_{10} (\kappa (A))$', fontsize=15)
plt.xticks(np.arange(0, 250, 20), fontsize=10)
plt.yticks(fontsize=10)
plt.yscale('log')
plt.show()
```

[ ]:
```
M = np.arange(10, 250, 10)
N = 50
plt.figure()
plt.plot(M, cond_list_circle_M)
plt.plot(M, cond_list_kite_M)
plt.plot(M, cond_list_rect_M)
plt.plot(M, cond_list_ellypse_M)
plt.plot(M, cond_list_square_M)
plt.axvline(x = N, color='b',linestyle="--", label = r'$N^\prime$')
plt.legend([r'Circle', r'Kite', r'Rectangle', r'Ellypse', r'Square',␣
 ↪r'$N^\prime$'], fontsize=10)
plt.title('Conditionning number as a function\nof the number of observation␣
 ↪points', fontsize=15)
plt.xlabel(r'Number of observation points ($M^\prime$)', fontsize=15)
plt.ylabel(r'Conditionning number $\log_{10} (\kappa (A))$', fontsize=15)
plt.xticks(np.arange(0, 250, 20), fontsize=10)
```

```python
plt.yticks(fontsize=10)
plt.yscale('log')
plt.show()
```

[ ]:
```python
plt.figure()
plt.plot(a_aux_circ/a_contour, cond_list_circle_auxil)
plt.plot(a_aux_kite/a_contour, cond_list_kite_auxil)
plt.plot(a_aux_rect/a_contour, cond_list_rect_auxil)
plt.plot(a_aux_ellypse/a_contour, cond_list_ellypse_auxil)
plt.plot(a_aux_sq/a_contour, cond_list_sq_auxil)
plt.legend([r'Circle', r'Kite', r'Rectangle', r'Ellypse', r'Square'], fontsize=10)
plt.title('Conditionning number as a function\nof the radius of the auxiliary curve', fontsize=15)
plt.xlabel(r'Distance of source points $R_s/R_\Omega$', fontsize=15)
plt.ylabel(r'Conditionning number $\log_{10} (\kappa (A))$', fontsize=15)
plt.xlim(0.1, 0.8)
plt.xticks(np.arange(0.1, 0.8, 0.1), fontsize=10)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.yscale('log')
plt.show()
```

[ ]:
```python
plt.figure()
plt.plot(a_obs_circ, cond_list_obs_circle)
plt.plot(a_obs_kite, cond_list_obs_kite)
plt.plot(a_obs_rect, cond_list_obs_rect)
plt.plot(a_obs_ellypse, cond_list_obs_ellypse)
plt.plot(a_obs_sq, cond_list_obs_sq)
plt.legend([r'Circle', r'Kite', r'Rectangle', r'Ellypse', r'Square'], fontsize=10, bbox_to_anchor=(0.9, 0.6))
plt.title('Conditionning number as a function of the distance\nof observable curve from contour', fontsize=15)
plt.xlabel('Position of observable curve $\Gamma$ from center\nof coordinate system' +  r'($\lambda_{0}$)', fontsize=15)
plt.ylabel(r'Conditionning number $\kappa (A))$', fontsize=15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()
```