Inverse Problems Exercises: 2024s s07 (non-sc)

https://www.umm.uni-heidelberg.de/miism/

Notes

- Please **DO NOT** change the name of the `.ipynb` file.
- Please **DO NOT** import extra packages to solve the tasks.
- Please put the `.ipynb` file directly into the `.zip` archive without any intermediate folder.

Please provide your personal information

- full name (Name):

YOUR ANSWER HERE

D04c: Gradient descent

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import fminbound
```

In [ ]:
```
file_gaussian = 'file_gaussian.npz'
with np.load(file_gaussian) as data:
    f_true = data['f_true']
    A_psf = data['A_psf']
    list_gn = data['list_gn']
```

Imaging model

The imaging model can be represented by

$$g = h \otimes f_{\text{true}} = A f_{\text{true}} = \mathcal{F}^{-1}\{\mathcal{F}\{h\}\mathcal{F}\{f_{\text{true}}\}\},$$

$$g' = g + \epsilon.$$

- f_{true} is the input signal
- h is the point spread function (kernel)
- $\otimes$ is the convolution operator
- A is the Toeplitz matrix of h
- $\mathcal{F}$ and $\mathcal{F}^{-1}$ are the Fourier transform operator and inverse Fourier transform operator
- ϵ is the additive Gaussian noise
- g is the filtered signal
- g' is the noisy signal

Mean squared error

Implement the mean squared error (MSE)

$$\text{MSE}(f) = \frac{1}{n} \sum_{i=1}^{n} (f_i - f_{\text{true}i})^2$$

- Given the input signal f
- Given the true signal f_{true}
- Implement the function `mean_squared_error()` (using `numpy.array`)

In [ ]:

```
def mean_squared_error(f, f_true):
    """ Compute the mean squared error comparing to the true signal:

    :param f: Input signal.
    :param f_true: True signal.
    :returns: Mean squared error.
    """
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:

```
# This cell contains hidden tests.
```

Difference matrix

Implement the difference matrix D_{diff}

$$D_{\text{diff}} = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 & -1 \\ -1 & 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & -1 & 1 & 0 & \dots & 0 & 0 \\ & & & \dots & & & \\ 0 & 0 & 0 & 0 & \dots & -1 & 1 \end{bmatrix}$$

- Given the size n_{diff}
- Implement the function `get_diff_matrix()` (using `numpy.array`)

In [ ]:
```
def get_diff_matrix(n):
    """ Compute a matrix to calculate the difference along a vector of the size
    between two neighboring elements.

    :param n: Size of the target vector.
    :returns: Matrix with shape (n, n), which calculates the difference.
    """
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell contains hidden tests.
```

Tikhonov regularization

Implement the objective function with Tikhonov regularization

$$L(f) = \|Af - g'\|_2^2 + \lambda\|D'f\|_2^2$$

- Given the input signal f
- Given the system matrix A
- Given the measurement g'
- Given the regularization matrix D'
- Given the regularization parameter λ
- Implement the function `objective_tikhonov()` (using `numpy.array`)

Implement the closed form solution of the regularized objective function

$$\tilde{f} = (A^T A + \lambda D'^T D')^{-1} A^T g' = A_\lambda^{PI} g'$$

- Given the system matrix A
- Given the measurement g'
- Given the regularization matrix D'
- Given the regularization parameter λ
- Implement the function `solution_tikhonov()` (using `numpy.array`)

In [ ]:
```python
def objective_tikhonov(f, A, g, D, lb):
    """ Compute the objective function with Tikhonov regularization.

    :param f: Current estimate of the signal.
    :param A: 2D matrix of the linear problem.
    :param g: Observed signal.
    :param D: 2D matrix in the regularization term.
    :param lb: Regularization parameter.
    :returns: Objective function value.
    """
    # YOUR CODE HERE
    raise NotImplementedError()

def solution_tikhonov(A, g, D, lb):
    """ Compute the estimate of the true signal with Tikhonov regularization.

    Use a regularization term to suppress noise.

    :param A: 2d matrix A of the linear problem.
    :param g: Observed signal.
    :param D: 2D matrix in the regularization term.
    :param lb: Regularization parameter.
    :returns: Estimate of the true signal.
    """
# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```python
# This cell contains hidden tests.
```

In [ ]:
```python
# This cell contains hidden tests.
```

Gradient magnitude solution

The gradient magnitude solution is the solution with $D' = D_{\text{diff}}$

- Calculate the closed form solution for the noisy signals in `list_gn`
- Return the outputs with λ of 0.1, 0.01, 0.001, respectively
- Save the solutions in the variable `list_f_closed` (as `list` of `numpy.array`)
- Save the corresponding objective values in the variable `list_L_closed` (as `list` of scalars)

Display the result

- Plot the outputs in `list_f_closed` in the same order of the parameter options in the subplots of `axs`
- Show the cases of the same noisy signal in the same subplot column (outer loop)
- Show the cases with the same λ in the same subplot row (inner loop)
- Plot the corresponding noisy signal in each subplot (after the filter output)
- Plot the input signal `f_true` in each subplot (after the noisy signal)
- Show the legend in each subplot
- Show the case information in the titles to the subplots
- Show the mean squared error of each output comparing to `f_true` in the titles to the subplots
- Show the objective function value of each output in the titles to the subplots

In [ ]:
```
fig, axs = plt.subplots(3, 3, figsize=(15, 15))
fig.suptitle('Gradient magnitude solution (closed form)')

# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell contains hidden tests.
```

Gradient descent technique

Gradient descent is an optimization method to find an f, which minimize the objective function $L(f)$. One iterative update is given by

$$f^{(i+1)} = f^{(i)} - s_i \nabla L(f^{(i)}),$$

where s_i is the optimal step size of the one-dimensional optimization problem

$$s_i = \arg\min_{s \in \mathbb{R}^+} L(f^{(i)} - s \nabla L(f^{(i)})).$$

Implement the iterative gradient descent updates

- Given the objective function $L(f)$
- Given the gradient of the objective function $\nabla L(f)$
- Given the initial value $f^{(0)}$
- Given the number of iterations n
- Estimating the optimal step size s_i in $[0, 10]$ (using `scipy.optimize.fminbound()`)
- Return the final value $f^{(n)}$ as the first output
- Return the history array of objective values $[L(f^{(0)}), \ldots, L(f^{(n)})]$ as the second output
- Implement the function `solve_gradient_descent_ls()` (using `numpy.array`)

In [ ]:
```
def solve_gradient_descent_ls(objective_function, gradient_function, f0, n):
    """
    :param objective_function: objective function of f.
    :param gradient_function: gradient of the objective function of f.
    :param f0: Starting values for initializing the parameters f.
    :param n: Number of iterative gradient updates.
    :returns: Final f and an array of n + 1 objective values in the optimization
    """
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [ ]:
```
# This cell contains hidden tests.
```

In [ ]:
```
# This cell contains hidden tests.
```

Tikhonov regularization with gradient descent

Implement the gradient of the objective function with Tikhonov regularization

$$\nabla L(f) = 2A^T(Af - g') + 2\lambda D'^T D' f$$

- Given the input signal f
- Given the system matrix A
- Given the measurement g'
- Given the regularization matrix D'
- Given the regularization parameter λ
- Implement the function `gradient_tikhonov()` (using `numpy.array`)

The gradient magnitude solution is the solution with $D' = D_{\text{diff}}$

- Calculate the solution by gradient descent for the noisy signals in `list_gn`
- Return the outputs with λ of 0.1, 0.01, 0.001, respectively, with $f^{(0)} = 0$, $n = 20$
- Save the solutions in the variable `list_f_gd` (as `list` of `numpy.array`)
- Save the corresponding objective value history in the variable `list_L_gd` (as `list` of `numpy.array`)

Display the result

- Plot the outputs in `list_f_gd` in the same order of the parameter options in the subplots of `axs`
- Show the cases of the same noisy signal in the same subplot column (outer loop)
- Show the cases with the same λ in the same subplot row (inner loop)
- Plot the corresponding noisy signal in each subplot (after the filter output)
- Plot the input signal `f_true` in each subplot (after the noisy signal)
- Show the legend in each subplot
- Show the case information in the titles to the subplots
- Show the mean squared error of each output comparing to `f_true` in the titles to the subplots
- Show the objective function value of each output in the titles to the subplots

In [ ]:

```python
def gradient_tikhonov(f, A, g, D, lb):
    """ Compute the gradient of the objective function with Tikhonov regularizat

    :param f: Current estimate of the signal.
    :param A: 2D matrix of the linear problem.
    :param g: Observed signal.
    :param D: 2D matrix in the regularization term.
    :param lb: Regularization parameter.
    :returns: Gradient value of the objective function.
    """
    # YOUR CODE HERE
    raise NotImplementedError()

fig, axs = plt.subplots(3, 3, figsize=(15, 15))
fig.suptitle('Gradient magnitude solution (gradient descent)')

# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:

```python
# This cell contains hidden tests.
```

Optimization history

Display the result

- Plot the arrays in `list_L_gd` as solid lines in the same order of the parameter options in the subplots of `axs`
- Plot the values in `list_L_closed` as horizontal dash lines in the same order of the parameter options in the subplots of `axs`
- Show the cases of the same noisy signal in the same subplots
- Make the subplots with log scaling on the y axis
- Show the legend in each subplot
- Show the case information in the titles to the subplots

In [ ]:

```python
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
fig.suptitle('Gradient magnitude solution (gradient descent)')

# YOUR CODE HERE
raise NotImplementedError()
```

Total variation

The objective function with total variation is

$$L(f) = \|Af - g\|_2^2 + \lambda \|\nabla f\|_1$$

The gradient of the objective function with total variation is

$$\nabla L(f) \approx 2A^T(Af - g) + \lambda \nabla \sum_{j=1}^{n} \sqrt{(f_j - f_{j-1})^2 + \beta^2} = 2A^T(Af - g) + \lambda \begin{bmatrix} r_1 \\ \dots \\ r_i \\ \dots \\ r_n \end{bmatrix},$$

where $1 \gg \beta^2 > 0$ and

$$r_i = \frac{f_i - f_{i-1}}{\sqrt{(f_i - f_{i-1})^2 + \beta^2}} - \frac{f_{i+1} - f_i}{\sqrt{(f_{i+1} - f_i)^2 + \beta^2}}$$

with $f_{-1} = 0$ and $f_n = 0$.

- Given the input signal f
- Given the system matrix A
- Given the measurement g'
- Given the regularization parameter λ
- Implement the objective function `objective_tv()` (using `numpy.array`)
    - Note, ∇f can be calculated by $D_{\text{diff}} f$
- Implement the gradient of the objective function with $\beta^2 = 0.001$ `gradient_tv()` (using `numpy.array`)

In [ ]:
```
def objective_tv(f, A, g, lb):
    """
    :param f: Current estimate of the signal.
    :param A: 2d Matrix A of the linear problem.
    :param g: Observed signal.
    :param lb: Regularization strength of TV.
    :returns: Objective function value.
    """
    # YOUR CODE HERE
    raise NotImplementedError()

def gradient_tv(f, A, g, lb):
    """
    :param f: Current estimate of the signal.
    :param A: 2d Matrix A of the linear problem.
    :param g: Observed signal.
    :param lb: Regularization strength of TV.
    :returns: Gradient value of the objective function.
    """
    # YOUR CODE HERE
    raise NotImplementedError()
```

In [ ]:
```
# This cell contains hidden tests.
```

Total variation with gradient descent

Solve the objective function with total variation by gradient descent

- Calculate the solution by gradient descent for the noisy signals in `list_gn`
- Return the outputs with λ of 0.1, 0.01, 0.001, respectively, with $f^{(0)} = 0$, $n = 50$
- Save the solutions in the variable `list_f_tv` (as `list` of `numpy.array`)
- Save the corresponding objective value history in the variable `list_L_tv` (as `list` of `numpy.array`)

Display the result

- Plot the outputs in `list_f_tv` in the same order of the parameter options in the subplots of `axs`
- Show the cases of the same noisy signal in the same subplot column (outer loop)
- Show the cases with the same λ in the same subplot row (inner loop)
- Plot the corresponding noisy signal in each subplot (after the filter output)
- Plot the input signal `f_true` in each subplot (after the noisy signal)
- Show the legend in each subplot
- Show the case information in the titles to the subplots
- Show the mean squared error of each output comparing to `f_true` in the titles to the subplots
- Show the objective function value of each output in the titles to the subplots

In [ ]:
```
fig, axs = plt.subplots(3, 3, figsize=(15, 15))
fig.suptitle('Total variation solution (gradient descent)')

# YOUR CODE HERE
raise NotImplementedError()
```

In [ ]:
```
# This cell contains hidden tests.
```

Optimization history

Display the result

- Plot the arrays in `list_L_tv` as solid lines in the same order of the parameter options in the subplots of `axs`
- Show the cases of the same noisy signal in the same subplots
- Make the subplots with log scaling on the y axis
- Show the legend in each subplot
- Show the case information in the titles to the subplots

In [ ]:
```
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
fig.suptitle('Total variation solution (gradient descent)')

# YOUR CODE HERE
raise NotImplementedError()
```

Question: Convergence

- Is the gradient descent method convergent to the global solution?
- Where does the objective function with Tikhonov regularization convergent to?

YOUR ANSWER HERE

Total Variation

Total Variation (Gradient Magnitude - based regularization) produces a series of effects. Could you replicate the following properties by varying the regularization parameter and letting the method converge until there is nearly no further change of the objective function:

- sharper boundaries/edges: how would you measure that
- staircasing: how would you measure that
- sparsity in the signal: how would you measure that

YOUR ANSWER HERE