EE4530 Applied convex optimization: Change Detection in Time Series Model

Thomas Prins (5885221)

***Abstract*— This report studies a change point detection problem for a time-varying autoregressive (AR) model using convex optimization techniques. The AR coefficients are assumed to be piecewise constant over time, changing only at a small number of unknown instants. To exploit this structure, the estimation problem is formulated as a convex optimization problem that combines a global least-squares data fidelity term with regularization on temporal coefficient differences. An ℓ_2-norm regularization term is included to improve numerical stability, while sparsity in coefficient changes is enforced through an ℓ_1-norm constraint.**

The resulting optimization problem is solved using three different approaches: a batch convex optimization method implemented in CVX, subgradient descent, and projected subgradient descent. The first-order methods are developed to handle the non-differentiability introduced by the regularization and constraint terms, and convergence is monitored using a relative objective decrease criterion. All methods are evaluated on a synthetic dataset with known ground-truth coefficients, allowing quantitative comparison in terms of mean squared error, convergence speed, and computational time.

The results show that the CVX-based approach achieves the best overall performance, converging significantly faster and yielding lower estimation error than the first-order methods. While subgradient descent and projected subgradient descent are computationally more demanding and converge more slowly, they provide flexible alternatives for large-scale or constrained settings. The study highlights the trade-offs between solver efficiency, convergence behavior, and implementation complexity in convex optimization-based change detection.

I. Introduction

Change point detection is a fundamental problem in statistical signal processing, with applications spanning econometrics, biomedical signal analysis, fault detection, and adaptive control. The goal is to identify time instants at which the statistical properties of a stochastic process change. These changes may correspond to abrupt variations in model parameters, system dynamics, or underlying generative mechanisms. In many practical scenarios, it is not known a priori whether a change has occurred, how many changes exist, or when they take place.

In this assignment, we study a change point detection problem arising in the estimation of time-varying autoregressive (AR) models. Specifically, we consider a scalar AR process whose coefficients are assumed to be piecewise constant, changing only at a small number of unknown time instants. This structure naturally induces a trade-off between model fidelity and temporal smoothness of the coefficients, which can be effectively captured using convex optimization techniques.

Thomas Prins is a master students at TU Delft, The Netherlands. E-mail addresses: {t.s.prins}@student.tudelft.nl.

The objective of the assignment is twofold. First, the step detection problem is formulated as a convex optimization problem that explicitly promotes piecewise-constant AR coefficients. Second, the resulting estimator is implemented and evaluated on a given dataset consisting of 300 observations, for which the true AR coefficients are available for validation purposes. The performance of the proposed method is assessed in terms of its ability to accurately estimate the AR coefficients and to correctly detect the locations of coefficient changes.

II. Problem Formulation

A. Autoregressive model

We consider a scalar autoregressive model of order three:

$$y(t+3) = a(t)y(t+2) + b(t)y(t+1) + c(t)y(t) + v(t), \quad (1)$$

where $v(t) \sim \mathcal{N}(0, 0.5^2)$ is additive white Gaussian noise. The coefficients $a(t)$, $b(t)$, and $c(t)$ are assumed to be piecewise constant over time.

Define the parameter vector

$$\boldsymbol{W}(t) = \begin{bmatrix} a(t) & b(t) & c(t) \end{bmatrix}^\top, \quad (2)$$

and the regressor

$$\mathbf{x}(t) = \begin{bmatrix} y(t+2) & y(t+1) & y(t) \end{bmatrix}^\top. \quad (3)$$

Then the model can be written compactly as

$$y(t+3) = \mathbf{x}^\top(t)\boldsymbol{W}(t). \quad (4)$$

B. Naive least-squares approach

A straightforward approach would be to estimate $\boldsymbol{W}(t)$ independently at each time step using least squares:

$$\min_{\boldsymbol{W}(t)} \left\| y(t+3) - \mathbf{x}^\top(t)\boldsymbol{W}(t) \right\|_2^2. \quad (5)$$

However, this problem is fundamentally ill-posed: at each time t, only a single scalar measurement is available to estimate three parameters. As a result, the regression matrix is rank-deficient and the solution is non-unique. Moreover, independent estimation ignores the strong temporal structure present in the coefficients.

This motivates a global estimation strategy that couples the parameters across time.

III. Convex Optimization Formulation

Let

$$\boldsymbol{W} = \begin{bmatrix} \boldsymbol{W}(1)^\top & \boldsymbol{W}(2)^\top & \cdots & \boldsymbol{W}(T-3)^\top \end{bmatrix}^\top$$

denote the stacked parameter vector.

A. Data fidelity term

A global least-squares objective is formed by summing the squared prediction errors:

$$f_{\mathrm{LS}}(\boldsymbol{W}) = \sum_{t=1}^{T-3} \left\| y(t+3) - \mathbf{x}^\top(t)\boldsymbol{W}(t) \right\|_2^2 . \tag{6}$$

B. Regularization for change detection

To promote piecewise-constant coefficients, we penalize temporal differences between successive parameter vectors:

$$\Delta \boldsymbol{W}(t) = \boldsymbol{W}(t) - \boldsymbol{W}(t-1).$$

In this work, two complementary mechanisms are used to control temporal variation.

First, an ℓ_2-norm regularization term is included in the objective function:

$$f_{\mathrm{REG}}(\boldsymbol{W}) = \sum_{t=2}^{T-3} \left\| \Delta \boldsymbol{W}(t) \right\|_2 . \tag{7}$$

This term promotes smoothness in the evolution of the AR coefficients and improves numerical stability of the optimization problem.

Second, sparsity in the number of coefficient changes is enforced through an ℓ_1-norm constraint:

$$\sum_{t=2}^{T-3} \left\| \Delta \boldsymbol{W}(t) \right\|_1 \le C, \tag{8}$$

where $C > 0$ controls the maximum total variation allowed in the parameter sequence. This constraint encourages most temporal differences to be exactly zero, resulting in piecewise-constant coefficient trajectories and explicit change point detection.

C. Final convex problem

The resulting convex optimization problem is

$$\begin{aligned} \min_{\boldsymbol{W}} \quad & \sum_{t=1}^{T-3} \left\| y(t+3) - \mathbf{x}^\top(t)\boldsymbol{W}(t) \right\|_2^2 \\ & + \lambda \sum_{t=2}^{T-3} \left\| \boldsymbol{W}(t) - \boldsymbol{W}(t-1) \right\|_1 , \\ \text{s.t.} \quad & \sum_{t=0}^{T-3} \left\| W(t) \cdot W(t-1) \right\|_1 \le C \end{aligned} \tag{9}$$

where $\lambda > 0$ controls the trade-off between data fidelity and change sparsity. This problem is convex and can be efficiently solved using standard solvers.

IV. First-Order Optimization Methods

A. (Projected) Subgradient Descent

Due to the presence of non-differentiable terms in the optimization problem, first-order methods based on subgradients are employed. In particular, both subgradient descent and projected subgradient descent are used to solve the convex problem.

1) Subgradient descent: For a convex objective function $f(\boldsymbol{W})$, the generic subgradient descent iteration is given by

$$\boldsymbol{W}^{k+1} = \boldsymbol{W}^k - \alpha_k g^k, \tag{10}$$

where $g^k \in \partial f(\boldsymbol{W}^k)$ denotes a subgradient of the objective function at iteration k, and $\alpha_k > 0$ is the step size.

In the considered problem, the objective function consists of two parts:

- A smooth least-squares data fidelity term.
- A non-smooth ℓ_2-norm regularization term on temporal parameter differences.

2) Least-squares gradient: The gradient of the least-squares term with respect to the parameter vector $\boldsymbol{W}(t)$ is given by

$$\frac{\partial f_{\mathrm{LS}}}{\partial \boldsymbol{W}(t)} = -2\,\mathbf{x}(t)\big(y(t+3) - \mathbf{x}^\top(t)\boldsymbol{W}(t)\big). \tag{11}$$

3) ℓ_2-regularization subgradient: The regularization term penalizes temporal differences

$$\Delta \boldsymbol{W}(t) = \boldsymbol{W}(t) - \boldsymbol{W}(t-1).$$

Since the ℓ_2 norm is non-differentiable at the origin, its subgradient is given by

$$\frac{\partial f_{\mathrm{REG}}}{\partial \boldsymbol{W}(t)} = \begin{cases} \lambda \dfrac{\Delta \boldsymbol{W}(t)}{\|\Delta \boldsymbol{W}(t)\|_2}, & \Delta \boldsymbol{W}(t) \neq \mathbf{0}, \\ \mathbf{0}, & \Delta \boldsymbol{W}(t) = \mathbf{0}. \end{cases} \tag{12}$$

The total subgradient used in the update step is the sum of the least-squares gradient and the regularization subgradient.

4) Projected subgradient descent: In addition to the regularization term in the objective, the optimization problem includes an ℓ_1-norm constraint on the temporal differences:

$$\sum_t \|\Delta \boldsymbol{W}(t)\|_1 \le C.$$

To handle this constraint, projected subgradient descent is employed. After each subgradient step, the iterate is projected onto the feasible set defined by the constraint:

$$\boldsymbol{W}^{k+1} = \Pi_{\mathcal{C}} \left(\boldsymbol{W}^k - \alpha_k g^k \right), \tag{13}$$

where $\Pi_{\mathcal{C}}(\cdot)$ denotes the Euclidean projection onto the convex set

$$\mathcal{C} = \left\{ \boldsymbol{W} \;\middle|\; \sum_t \|\Delta \boldsymbol{W}(t)\|_1 \le C \right\}.$$

This projection enforces sparsity in the temporal differences of the parameter vectors, thereby promoting piecewise-constant behavior.

B. Step Size Selection

To guarantee convergence of the subgradient method for convex functions with bounded subgradients, the step size sequence $\{\alpha_k\}$ must satisfy

$$\alpha_k > 0, \quad \sum_{k=0}^{\infty} \alpha_k = \infty, \quad \sum_{k=0}^{\infty} \alpha_k^2 < \infty. \tag{14}$$

A commonly used diminishing step size is

$$\alpha_k = \frac{\alpha_0}{\sqrt{k+1}}. \tag{15}$$

Alternatively, if the optimal objective value $f^\star$ is known, the Polyak step size may be used:

$$\alpha_k = \frac{f(\boldsymbol{W}^k) - f^\star}{\|g^k\|_2^2}, \tag{16}$$

which ensures monotonic decrease of the objective function, but requires prior knowledge of $f^\star$.

V. Results

In this section, the performance of three different optimization approaches is evaluated:

1) A batch convex optimization approach using CVX.
2) Subgradient descent.
3) Projected subgradient descent.

All methods are evaluated in terms of convergence behavior, computational time, and the quality of the estimated coefficients. The regularization parameter λ is tuned in steps of 0.1 for each method to achieve the best tracking performance. Furthermore, parameters such as tolerance levels and maximum number of iterations have also been adjusted but will not be specifically mentioned.

A. Stopping criterion

For the iterative optimization methods, convergence is determined using a relative objective decrease criterion. Let f^k denote the objective value at iteration k. The algorithm is terminated when

$$\frac{\left|f^k - f^{k-1}\right|}{\max\left(1, \left|f^{k-1}\right|\right)} < \varepsilon, \tag{17}$$

where $\varepsilon > 0$ is a predefined tolerance. This stopping condition ensures scale-invariant convergence detection and prevents premature termination when the objective value is close to zero.

The same criterion is used for both subgradient descent and projected subgradient descent. For the CVX-based approach, the solver's internal stopping criteria are used.

B. CVX-based optimization

The convex optimization problem described in Equation 9 is solved using the SCS solver provided by the CVXPY framework. The solver directly handles the ℓ_1-norm constraint on the temporal parameter differences and optimizes the full batch problem.

Empirically, the best performance is obtained for a regularization parameter value of

$$\lambda = 0.5.$$

For this configuration, the solver converges after 4475 iterations, with a total computation time of

$$1.82 \text{ seconds}.$$

The estimated coefficients obtained using CVX are shown in Fig. 1. This solution serves as a reference for evaluating the performance of the first-order methods.

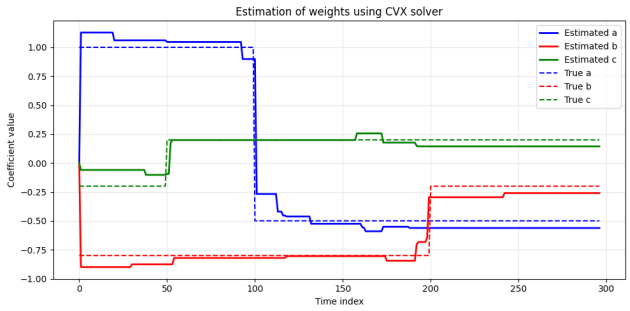

Fig. 1. Results of the CVX solver

C. Subgradient descent

Subgradient descent is implemented using the Polyak step size rule and without explicit constraint projection, since constraints are not possible to add without reworking the problem. The following parameters are varied during testing:

- Regularization parameter λ
- Polyak step size α_k

The best tracking performance is achieved for

$$\lambda = 5.4.$$

Under this configuration, the algorithm converges after 6054 iterations, with a total computation time of

$$48.53 \text{ seconds}.$$

The resulting coefficient estimates are shown in Fig. 2. Compared to the CVX solution, subgradient descent requires a larger number of iterations and significantly more computation time, reflecting the slower convergence of first-order methods.

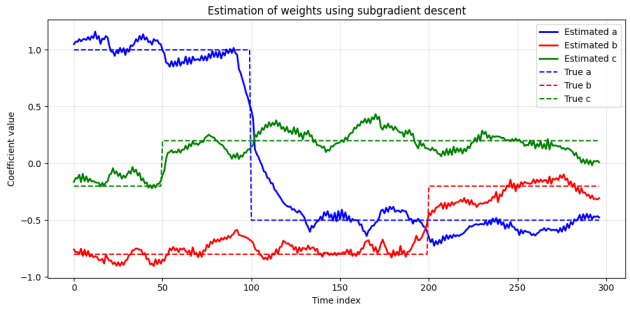

Fig. 2. Results of the subgradient descent algorithm

D. Projected subgradient descent

Projected subgradient descent extends the standard subgradient method by explicitly enforcing the ℓ_1-norm constraint through projection at each iteration. The same parameter set is explored:

- Regularization parameter λ
- Constraint parameter C

- Diminishing step size rule α_k

The best performance is obtained for

$$\lambda = 4.5.$$

In this case, the algorithm requires more than 20000 iterations to converge, with a total computation time of

$$243.08 \text{ seconds.}$$

The estimated coefficients are shown in Fig. 3. While the projection step enforces strict adherence to the constraint, it introduces additional computational overhead and slows down convergence compared to unconstrained subgradient descent.

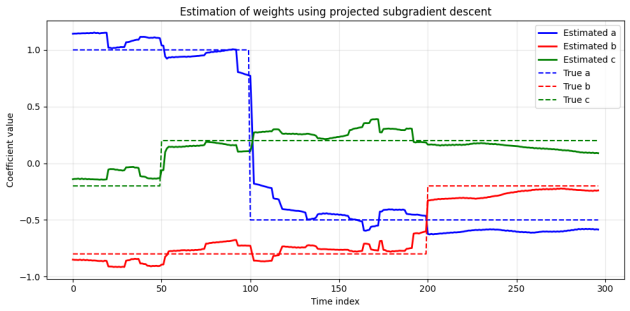

Fig. 3. Results of the projected subgradient descent algorithm

E. Performance comparison

To quantitatively compare the three approaches, the mean squared error (MSE) is evaluated at the optimal λ value for each method. The resulting MSE values are shown in Fig. 4.

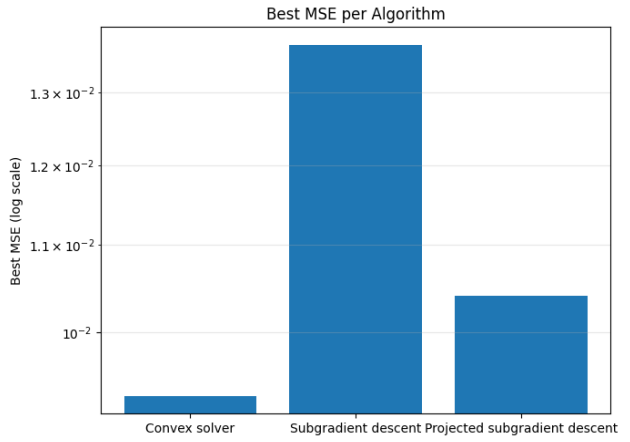

Fig. 4. MSE of the different algorithms at their optimal λ values

VI. Discussion

The results clearly demonstrate that the CVX-based approach outperforms both subgradient descent and projected subgradient descent in terms of mean squared error (MSE), convergence speed, and computational time. This behavior is expected, as CVX employs highly optimized interior-point and operator-splitting methods that are specifically designed for large-scale convex optimization problems. Although these solvers internally rely on first-order concepts, they incorporate advanced acceleration, preconditioning, and adaptive step size strategies that significantly improve convergence.

In contrast, the first-order methods require substantially more iterations to reach convergence. Standard subgradient descent converges more slowly even though Polyak step sizes are used. Projected subgradient descent exhibits the slowest convergence among the three methods, which can be attributed to the additional projection step onto the ℓ_1-norm constraint set. While this projection enforces strict feasibility and promotes sparsity in the detected change points, it introduces significant computational overhead. Furthermore, a simple diminishing step size assures convergence but is not the fastest converging method, resulting in slower progress per iteration.

The difference in convergence behavior between subgradient descent and projected subgradient descent highlights an important trade-off. Unconstrained subgradient descent allows greater flexibility in the updates, but may temporarily violate the desired sparsity structure. Projected subgradient descent strictly enforces the constraint at each iteration, but at the cost of increased computational complexity and reduced convergence speed.

The choice of step size also plays a critical role in the observed performance. Diminishing step size rules provide theoretical convergence guarantees but tend to be conservative in practice. More aggressive or adaptive step size strategies could potentially improve convergence rates, particularly for the projected subgradient method.

VII. Conclusion

In this report, a convex optimization framework for change detection in a time-varying autoregressive model was developed and analyzed. By modeling the AR coefficients as piecewise constant and incorporating both ℓ_2 regularization and an ℓ_1-norm constraint on temporal differences, the formulation enables accurate parameter estimation and effective change point detection.

Three optimization approaches were evaluated: a batch convex optimization method using CVX, subgradient descent, and projected subgradient descent. The CVX-based approach achieved the best performance in terms of estimation accuracy, convergence speed, and computational efficiency. First-order methods, while more flexible and easier to implement, required significantly more iterations and computation time to converge.

Future work could explore the use of more advanced step size strategies, such as Polyak step sizes, for projected subgradient descent. Although convergence guarantees for this combination were not covered in the course material, such an approach may lead to improved practical performance when reference objective values are available.

Overall, this study highlights the trade-offs between solver optimality, computational efficiency, and implementation complexity in convex optimization-based change detection.

References

VIII. Python code

Link to github page: https://github.com/ThomasPrins1/ConvexOptimization.git

```
# −∗− coding: utf−8 −∗−
"""
Created_on_Sun_Jan_11_14:17_2026

@author:■user
@studentID:■5885221
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import expm
from sympy import symbols
from math import isclose
import cvxpy as cp
from collections import Counter
import scipy.io
import math
import time
np.random.seed(19680806)


"""Variables"""
change_Dataset = scipy.io.loadmat('change_detection.mat')
a = np.array((change_Dataset['a']))
b = np.array((change_Dataset['b']))
c = np.array((change_Dataset['c']))
y = np.array((change_Dataset['y'])).reshape(−1, 1)
N = len(y)
variance = 0.5
v = np.random.normal(0, variance∗∗2, (N, 1))
x = np.zeros((3, N))
w_true = np.vstack((a.T, b.T, c.T)) # shape (3, N)
w_single = np.zeros((3, N))
w_CVX = np.zeros((3, N))
w_gradientDescent = np.zeros((3, N))
w_prev = np.zeros((3, 1))
iterMax = 30000
eps = 1e−8

"""Functions"""
def convexSolve_all(x,y,N,Lapda_value):
    #x /= (np.linalg.norm(x, axis=0, keepdims=True) + eps)
    W = cp.Variable((3,N−3))
    Lapda = cp.Parameter(nonneg=True)
    # should be tested with different lapda values!
    Lapda.value = Lapda_value
    const = cp.Constant(0)
    obj = cp.Constant(0)
    # shapes:
    # X is 3,N
    # y is N,1
    # W is 3,N

    for i in range(3,N):
        obj += (cp.square(y[i]−x[:,i].T@W[:,i−3]))
```

```
        if i>4:
            obj += Lapda*(cp.norm(W[:,i-3]-W[:,i-4],2))
            const += (cp.norm(W[:,i-3]-W[:,i-4],1))
    constraints = [const<=3+eps]
    prob = cp.Problem(cp.Minimize(obj), constraints)
    prob.solve(solver=cp.SCS, verbose=True)
    return W.value,prob.solver_stats.num_iters

def objective(W, x, y, Lapda):
    obj = 0
    for i in range(N-3):
        obj += ((y[i]-x[:,i].T@W[:,i])**2)
        if i>=1:
            obj += Lapda*(np.linalg.norm(W[:,i]-W[:,i-1],2))
    return obj

def simpleSubGradientDescent(x,y,N,Lapda,Alpha0=1e-1, tol=5e-7):
    # need the dual function?
    ## Not needed unless there is a distributed problem!
    # need to find subgradient, this is needed due to the norm in the objective function:
    ## comes from ((y[i]-x[:,i].T@W[:,i])**2)'
    x_trim = x[:, 3:] # 3 x (N-3)
    y_trim = y[3:] # length (N-3)
    x_norm = x_trim / (np.linalg.norm(x_trim, axis=0, keepdims=True)+eps) # normalize
    W_out = np.zeros((3, N-3))
    W_simple = np.zeros_like(W_out)
    #initial condition?
    #W_out[:,0] = np.array(([1,-0.2,-0.8])) # something nonzero
    prev_obj = 1000
    k_out = 0
    for k in range(iterMax):
        subgradient = np.zeros_like(W_out)

    # data term
        for j in range(N-3):
            r = x_trim[:,j].T @ W_out[:,j] - y_trim[j]
            subgradient[:,j] += 2 * x_trim[:,j] * r

    # TV term
        for j in range(N-4):
            diff = W_out[:,j+1] - W_out[:,j]
            norm = np.linalg.norm(diff)
            if norm > eps:
                g = Lapda * diff / norm
                subgradient[:,j+1] += g
                subgradient[:,j] -= g

        obj_W = objective(W_out,x_trim,y_trim,Lapda)
        objective_value = max(obj_W,0.0)
        #Alpha_value = Alpha0 / np.sqrt(k + 1) # simple decaying stepsize
        Alpha_value = objective_value/((np.linalg.norm(subgradient)**2)+eps)
        W_out -= Alpha_value * subgradient

        rel_change = abs(obj_W - prev_obj) / max(1.0, abs(prev_obj))
        if rel_change < tol:
            print("convergence■at:■",k)
            k_out=k
            break
```

```
        prev_obj = obj_W
    return W_out,k_out

def project_l1_ball(v, C):
    """
    Project vector v onto {||v||_1 <= C}
    """
    if np.sum(np.abs(v)) <= C:
        return v

    u = np.sort(np.abs(v))[::-1]
    sv = np.cumsum(u)
    rho = np.where(u * np.arange(1, len(u)+1) > (sv - C))[0][-1]
    theta = (sv[rho] - C) / (rho + 1)
    return np.sign(v) * np.maximum(np.abs(v) - theta, 0)

def project_total_change(W, C):
    D = np.diff(W, axis=1)
    D_proj = project_l1_ball(D.flatten(), C)
    D_proj = D_proj.reshape(D.shape)

    W_proj = np.zeros_like(W)
    W_proj[:,0] = W[:,0]
    for t in range(1, W.shape[1]):
        W_proj[:,t] = W_proj[:,t-1] + D_proj[:,t-1]

    return W_proj

def subGradientDescent(x,y,N,Lapda, Alpha0=1e-1, tol=5e-7, C = 30):
    x_trim = x[:, 3:] # 3 x (N-3)
    y_trim = y[3:] # length (N-3)
    W_out = np.zeros((3, N-3))
    #initial condition?
    W_out[:,0] = np.array(([1,-0.2,-0.8])) # something nonzero
    prev_obj = 1000
    k_out = 0
    for k in range(iterMax):
        subgradient = np.zeros_like(W_out)

    # data term
        for j in range(N-3):
            r = x_trim[:,j].T @ W_out[:,j] - y_trim[j]
            subgradient[:,j] += 2 * x_trim[:,j] * r

    # TV term
        for j in range(N-4):
            diff = W_out[:,j+1] - W_out[:,j]
            norm = np.linalg.norm(diff)
            if norm > eps:
                g = Lapda * diff / norm
                subgradient[:,j+1] += g
                subgradient[:,j] -= g

        obj_W = objective(W_out,x_trim,y_trim,Lapda)
        #objective_value = max(obj_W,0.0)
        #Alpha_value = objective_value/((np.linalg.norm(subgradient)**2)+eps)
        Alpha_value = Alpha0 / np.sqrt(k + 1)
```

```
        W_tmp = W_out − Alpha_value∗subgradient
        W_out = project_total_change(W_tmp, C)

        # check if objective changes with new iterations of W
        rel_change = abs(obj_W − prev_obj) / max(1.0, abs(prev_obj))

        if rel_change < tol:
            print("convergence■at:■",k)
            k_out=k
            break
        prev_obj = obj_W
    return W_out,k_out



"""■Main■Code■"""
"""■Question■1:■Creation■of■the■optimization■problem■"""
for i in range(3,N−3):
    x_slice = np.vstack((y[i−1],y[i−2],y[i−3]))
    x[:,i] = x_slice.squeeze()
Max_Lapda = 11
best_mse1 = np.inf
lapda_space1 = np.linspace(0,1,Max_Lapda)
mse_convex = []
for l in lapda_space1:
    start1 = time.time()
    w_temp_CVX,iter_CVX = convexSolve_all(x,y,N,l)
    time_CVX = time.time() − start1
    mse_w_CVX = np.mean((w_temp_CVX − w_true)∗∗2)
    mse_convex.append(mse_w_CVX)
    if mse_w_CVX < best_mse1:
        best_mse1 = mse_w_CVX
        bestTime_CVX = time_CVX
        bestIter_CVX = iter_CVX
        w_CVX = w_temp_CVX
        bestLapda_CVX = l

best_mse2 = np.inf
lapda_space2 = np.linspace(5,6,Max_Lapda)
mse_GD = []
for iter,l in enumerate(lapda_space2):
    print("Lapda:",l,"iter:",iter)
    start2 = time.time()
    w_temp_GD,iter_GD = simpleSubGradientDescent(x,y,N,l)
    time_GD = time.time() − start2
    mse_w_GD = np.mean((w_temp_GD − w_true)∗∗2)
    mse_GD.append(mse_w_GD)
    if mse_w_GD < best_mse2:
        best_mse2 = mse_w_GD
        bestTime_GD = time_GD
        bestIter_GD = iter_GD
        w_gradientDescent = w_temp_GD
        bestLapda_GD = l

best_mse3 = np.inf
lapda_space3 = np.linspace(4,5,Max_Lapda)
mse_IGD = []
for iter,l in enumerate(lapda_space3):
```

```
        print("Lapda:",l,"iter:",iter)
        start3 = time.time()
        w_temp_IGD,iter_IGD = subGradientDescent(x,y,N,l)
        time_IGD = time.time() - start3
        mse_w_IGD = np.mean((w_temp_IGD - w_true)**2)
        mse_IGD.append(mse_w_IGD)
        if mse_w_IGD < best_mse3:
            best_mse3 = mse_w_IGD
            bestTime_IGD = time_IGD
            bestIter_IGD = iter_IGD
            w_improvedGradientDescent = w_temp_IGD
            bestLapda_IGD = l

print("best value for lapda (CVX)",bestLapda_CVX)
print("best value for lapda (subgradient descent)",bestLapda_GD)
print("best value for lapda (projected subgradient descent)",bestLapda_IGD)
print("time (CVX)",bestTime_CVX)
print("time (subgradient descent)",bestTime_GD)
print("time (projected subgradient descent)",bestTime_IGD)
print("iterations (CVX)",bestIter_CVX)
print("iterations (subgradient descent)",bestIter_GD)
print("iterations (projected subgradient descent)",bestIter_IGD)
""" Plotting """

# Estimated versus true values (all):
plt.figure(figsize=(10,5))

# Estimated weights
plt.plot(w_CVX[0,:], label='Estimated a', linewidth=2, color="blue")
plt.plot(w_CVX[1,:], label='Estimated b', linewidth=2, color="red")
plt.plot(w_CVX[2,:], label='Estimated c', linewidth=2, color="green")

# True weights
plt.plot(w_true[0,:], '--', label='True a', color="blue")
plt.plot(w_true[1,:], '--', label='True b', color="red")
plt.plot(w_true[2,:], '--', label='True c', color="green")

plt.xlabel('Time index')
plt.ylabel('Coefficient value')
plt.title('Estimation of weights using CVX solver')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('CVX.png')
plt.show()

# Estimated versus true values (not CVX):
plt.figure(figsize=(10,5))

# Estimated weights
plt.plot(w_gradientDescent[0,:], label='Estimated a', linewidth=2, color="blue")
plt.plot(w_gradientDescent[1,:], label='Estimated b', linewidth=2, color="red")
plt.plot(w_gradientDescent[2,:], label='Estimated c', linewidth=2, color="green")

# True weights
plt.plot(w_true[0,:], '--', label='True a', color="blue")
plt.plot(w_true[1,:], '--', label='True b', color="red")
plt.plot(w_true[2,:], '--', label='True c', color="green")
```

```
plt.xlabel('Time index')
plt.ylabel('Coefficient value')
plt.title('Estimation of weights using subgradient descent')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('subgradientDescent.png')
plt.show()

# Estimated versus true values (not CVX):
plt.figure(figsize=(10,5))

# Estimated weights
plt.plot(w_improvedGradientDescent[0,:], label='Estimated a', linewidth=2, color="blue")
plt.plot(w_improvedGradientDescent[1,:], label='Estimated b', linewidth=2, color="red")
plt.plot(w_improvedGradientDescent[2,:], label='Estimated c', linewidth=2, color="green")

# True weights
plt.plot(w_true[0,:], '--', label='True a', color="blue")
plt.plot(w_true[1,:], '--', label='True b', color="red")
plt.plot(w_true[2,:], '--', label='True c', color="green")

plt.xlabel('Time index')
plt.ylabel('Coefficient value')
plt.title('Estimation of weights using projected subgradient descent')
plt.legend()
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig('projectedSubgradientDescent.png')
plt.show()

#MSE values:
best_mse_values = [
    np.min(mse_convex),
    np.min(mse_GD),
    np.min(mse_IGD)
]

labels = [
    'Convex solver',
    'Subgradient descent',
    'Projected subgradient descent'
]

plt.figure(figsize=(7,5))
plt.bar(labels, best_mse_values)

plt.yscale('log')
plt.ylabel('Best MSE (log scale)')
plt.title('Best MSE per Algorithm')
plt.grid(axis='y', which='both', alpha=0.3)

plt.tight_layout()
plt.savefig('MSE.png')
plt.show()
```