Danmarks Tekniske Universitet

02471 Machine Learning for Signal Processing

Assignment 3

AUTHOR

Cheng-Liang Lu - s220034

December 15, 2023

Contents

1 Problem 3.1 Sparse signal representations

1.1 Problem 3.1.1

A multitone signal is sparse in the DCT domain (we can see that by looking at the DCT formula (week 7 slide 22). Then, we can estimate the signal from a few samples by taking this sparsity into account. The goal is to find an acceptable estimation of the original signal in the sparse domain that has as few non-zero components as possible (the smallest components must be put down to zero). This is possible by minimizing the least square error under some constraints which is equivalent to adding a regularization term on the least square error cost function. The LASSO cost function (equation (9.6) in ML) uses the L1 norm in the regularization term. By minimizing this cost function, we can derive equation (9.13) in ML which shows that if an estimated value of the signal using LS estimate is too small compared to lambda it will be put down to zero in the LASSO estimation. Lambda controls the amount of non zero-elements. We can use other norms such as the L0 norm which is the sparsest norm but the L1 norm is the computationally most efficient norm as it is the most sparse true norm, and is convex. In practice, the cost function could be minimized using the gradient descent as we have seen in the IST algorithm.

1.2 Problem 3.1.2

I decided to use the LASSO cost function and IST. Both to solve this problem as compare to each other. I used the build-in function lasso in Python and IST algorithm. I choose $\lambda = 0.01$ as it gives an acceptable number of non-zero components. The estimated signal is displayed figure 1 both in time and DCT domain. The estimated parameters of the multitone signal are given below: (As too much values I only print a and m with first 4 respective)

- K status: 36
- a_i values: [2.84376962e-01 7.38536161e-01 1.36582746e-02 8.04106128e-01]
- m_j values: [4 9 22 24]

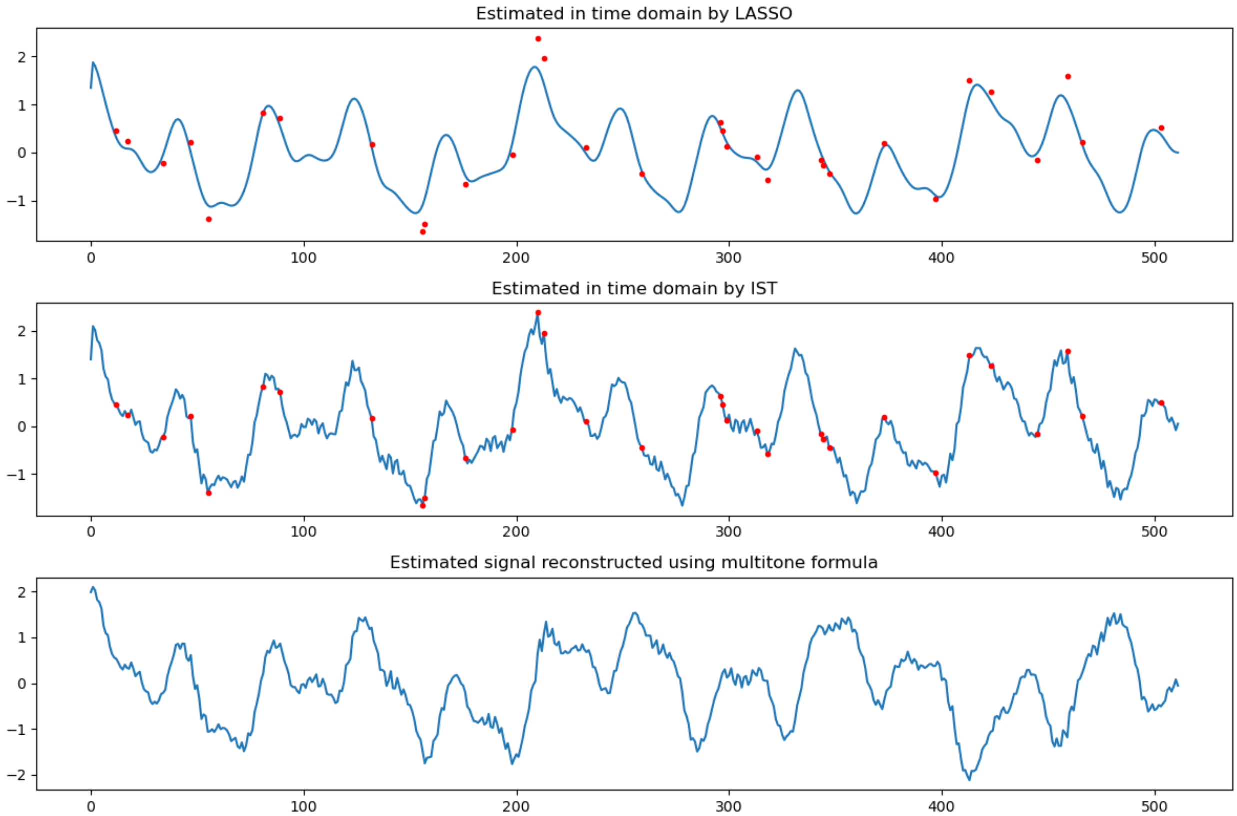

Figure 1: Sampled signal.

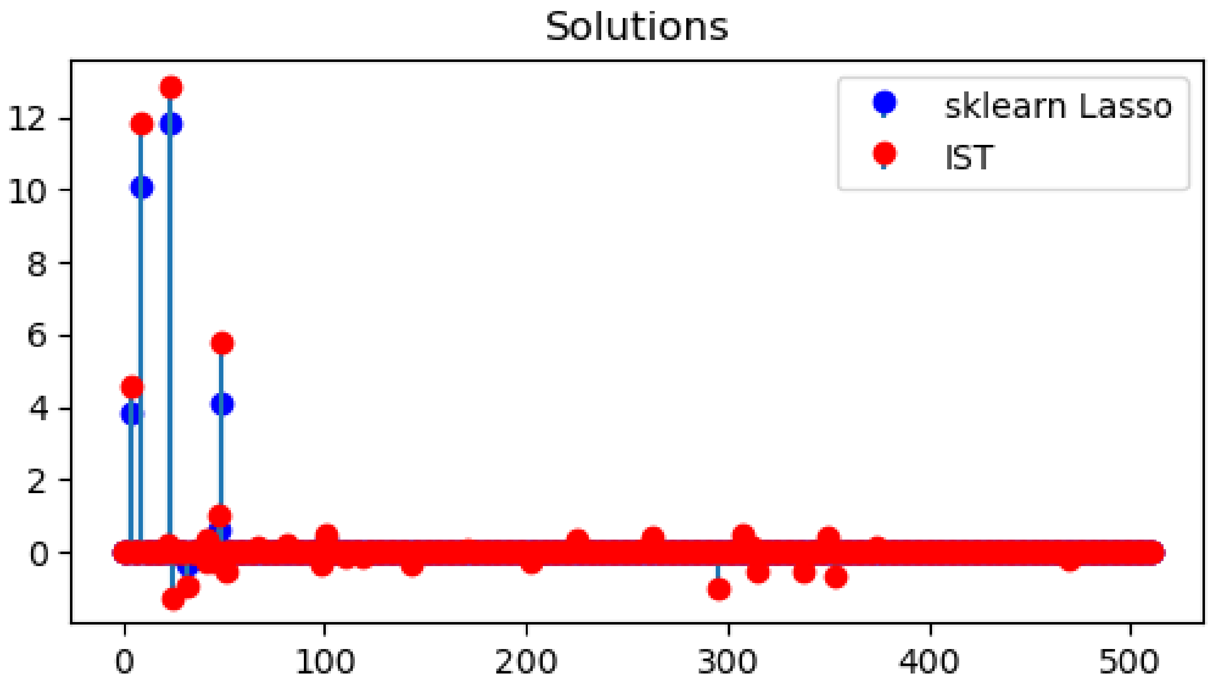

Figure 3: DCT domain.

2 Problem 3.2 Bayesian inference and the EM algorithm

2.1 Problem 3.2.1

In this optimization problem, the two terms can be interpreted as the likelihood, prior, mean square error, and regularization.

Roles of the Objective Terms:

- $\frac{1}{2\sigma_\eta^2}||y - X\theta||_2^2$:
  - **The perspectives of likelihood and mean squared error:**
    - Quantifies the error between the model's predictions $X\theta$ and the observed data y.
    - Appears as the likelihood, minimizing this error encourages in order to maximize the likelihood of observed data y given the model.
- $\frac{1}{2\sigma_\theta^2}||\theta||_2^2$:
  - **Prior Knowledge and Regularization Perspective:**
    - Represents the negative log of the prior probability distribution for model parameters θ.
    - Acts as a regularization term, controlling the size of parameter θ, preventing over-reliance on parameters, avoiding overfitting, and enhancing model generalization.

Influence of Parameters σ_η and σ_θ:

- σ_η**(Standard deviation of observation errors):**
  - A larger σ_η indicates higher noise levels in observed data y, leading to a greater reliance on the regularization term, reducing overfitting.
  - A smaller σ_η indicates lower noise levels in observed data y, resulting in a greater reliance on the MSE term, increasing the fit to observed data.
- σ_θ **(Standard deviation of parameter θ):**
  - A larger σ_θ implies less restriction on θ, allowing wider ranges for parameter values. θ.
  - A smaller σ_θ implies stricter constraints on θ, leading to more restricted parameter values.

These parameter choices balance the model's fit to observed data and control over parameter sizes. Adjusting σ_η and σ_θ values can alter the model's performance to adapt to varying noise levels and complexities.

2.2 Problem 3.2.2

As we know the density function,

$$p(z|a, b) = \frac{b^a}{\Gamma(a)} z^{-a-1} exp(-\frac{b}{z})$$

we take log of the density function $\ln p(z|a, b)$,

$$\ln p(z|a, b) = a \ln b + (-a - 1) \ln z - \ln(\Gamma(a)) - \frac{b}{z}$$

Since the question allows us to omit the terms, which are not depend on z, it could be derived as:

$$\ln p(z|a, b) = (-a - 1) \ln z - \frac{b}{z}$$

2.3 Problem 3.2.3

As we know the Bayes theorem: $p(\theta, \sigma_\eta^2|y)$ should be,

$$p(\theta, \sigma_\eta^2|y) = \frac{p(y|\sigma_\eta^2, \theta)p(\sigma_\eta^2, \theta)}{p(y)}$$

As σ_η^2 stands for the variance of observational noise, while θ represents model parameters, and here, we can assume that they are independent. Hence,

$$p(\sigma_\eta^2, \theta) = p(\sigma_\eta^2)\dot{p}(\theta)$$

So it will have this formula for posterior,

$$p(\theta, \sigma_\eta^2|y) = \frac{p(y|\sigma_\eta^2, \theta)p(\sigma_\eta^2)p(\theta)}{p(y)}$$

Then, taking the logarithm of both sides, we get:

$$\ln p(\theta, \sigma_\eta^2|y) = \ln(p(y|\sigma_\eta^2, \theta)p(\sigma_\eta^2)p(\theta)) - \ln p(y)$$

Given the distributions:

- Likelihood:$p(y|\sigma_\eta^2, \theta)$
- Prior on σ_η^2: $p(\sigma_\eta^2)$
- Prior on θ: $p(\theta)$

It can be substituted these into the equation:

$$\ln p(\theta, \sigma_\eta^2|y) = \ln p(y|\sigma_\eta^2, \theta) + \ln p(\sigma_\eta^2) + \ln p(\theta) - \ln p(y)$$

Or even,

$$\ln p(\theta, \sigma_\eta^2|y) \propto \ln p(y|\sigma_\eta^2, \theta) + \ln p(\sigma_\eta^2) + \ln p(\theta)$$

As $p(y)$ is constant and the goal is to minimize $lnp(\theta, \sigma_\theta^2|y)$.

$lnp(sigma_\eta^2|a, b) = -(a+1)ln\theta_\eta^2 - \frac{b}{\theta_\eta^2}$ with the previous question. Use what we derived in ex. week 9:

$$\begin{aligned} lnp(y|\sigma_\eta^2, \theta) &= -\frac{N}{2}ln(2\pi) - \frac{N}{2}ln\sigma_\eta^2 - \frac{1}{2\sigma_\eta^2}||y - \theta^T x||lnp(\theta) \\ &= -\frac{K}{2}ln(2\pi) - \frac{K}{2}ln\sigma_\theta^2 - \frac{1}{2\sigma_\theta^2}||\theta||^2 \quad , \ with\ \theta \in R^k \end{aligned}$$

Then,

$$\begin{aligned} &lnp(y|\sigma_\eta^2, \theta) \\ &= -(\frac{K}{2} + \frac{N}{2})ln(2\pi) - \frac{N}{2}ln(\sigma_\eta^2) - \frac{K}{2}ln(\sigma_\theta^2) - \frac{1}{2\sigma_\theta^2}||y - \theta^T x|| - \frac{K}{2}ln(\sigma_\eta^2)||\theta||^2 \\ &\qquad - (a+1)ln(\sigma_\eta^2) - \frac{b}{\sigma_\eta^2} \end{aligned}$$

3 Problem 3.3 Estimation of ICA solution

3.1 Problem 3.3.1

The findings are presented in Figure 5. The computation of the error involves the normalization of columns in both matrices, A and $\hat{A}$, followed by potential column switches in $\hat{A}$ if deemed necessary. Subsequently, the sum of coefficients resulting from the absolute differences between A and $\hat{A}$ is calculated. An observation of note is the relatively low error, indicating a successful recovery of the sources.

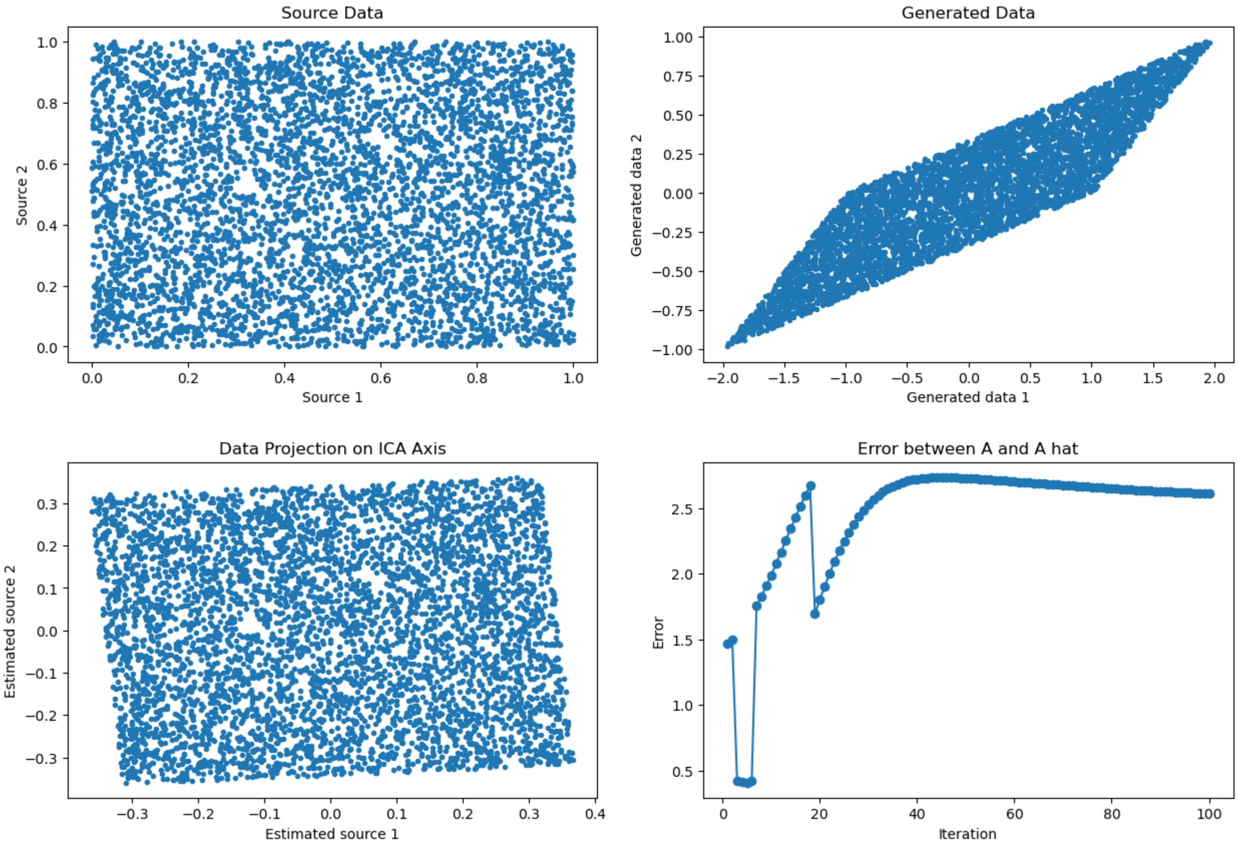

Figure 5: s is drawn from a uniform distribution $U(0, 1)$.

3.2 Problem 3.3.2

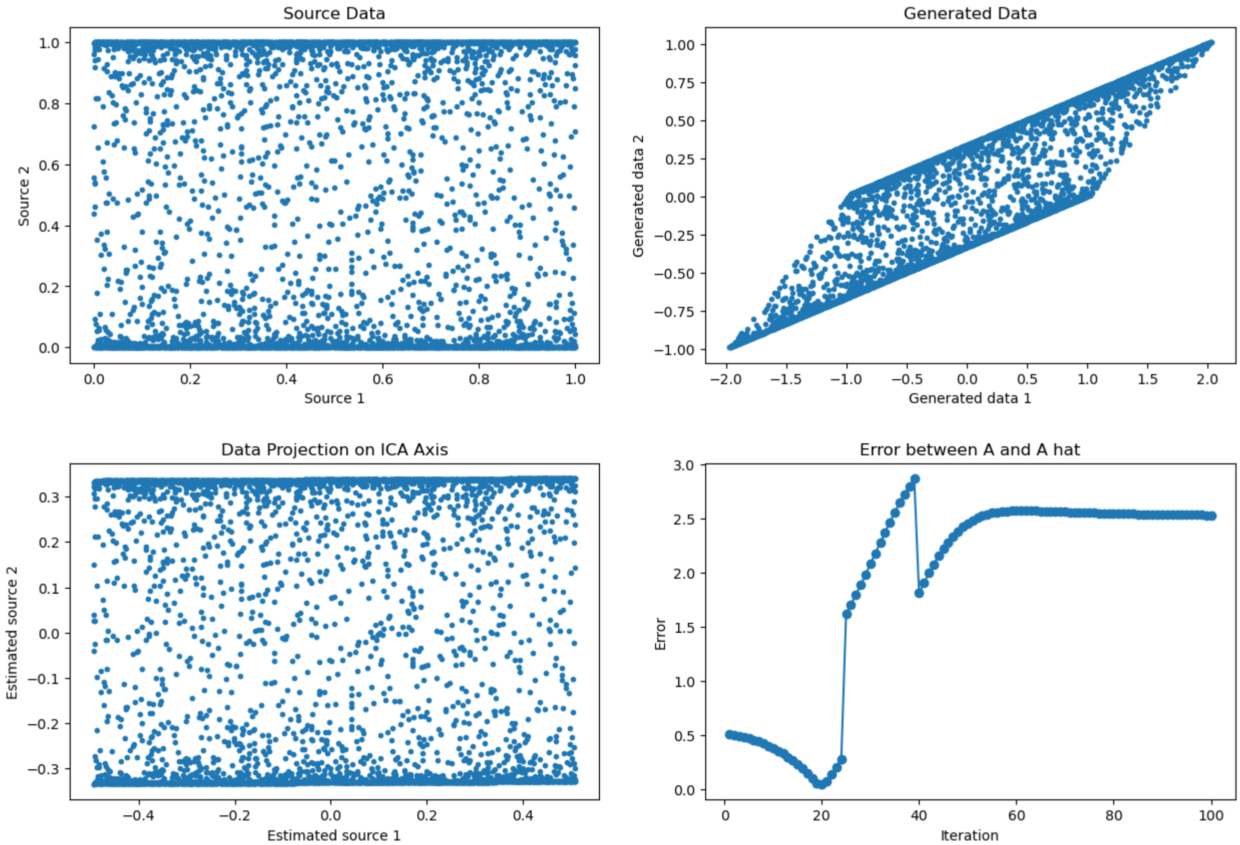

Figure 7: $s1$ is drawn from $U(0;1)$, and $s2$ is drawn from a beta distribution $B(0.1, 0.1)$.

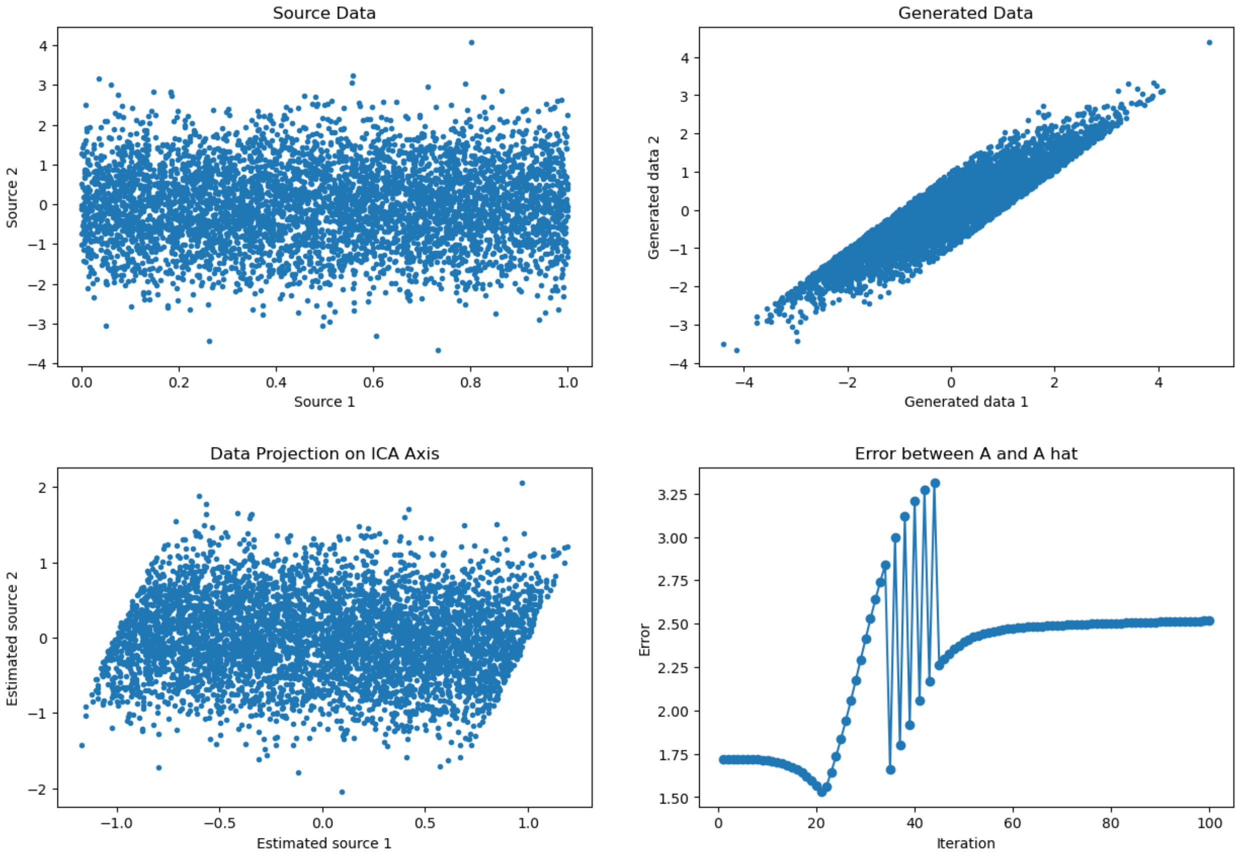

Figure 9: $s1$ is drawn from $U(0,1)$, and $s2$ is drawn from a normal distribution $N(0,1)$.

The results are displayed. We can see that ICA is able to successfully estimate in the first two cases but ICA canât unmix the sources for the last case. Indeed, we know from week 8 that we canât unmix gaussian distributed sources with ICA. This can be shown by calculating the distribution of the observations using a change of variables. This leads to the fact that the observation variables and the source variables have the same distribution under the ICA model. Hence, the ICA model is not able to separate Gaussian sources.

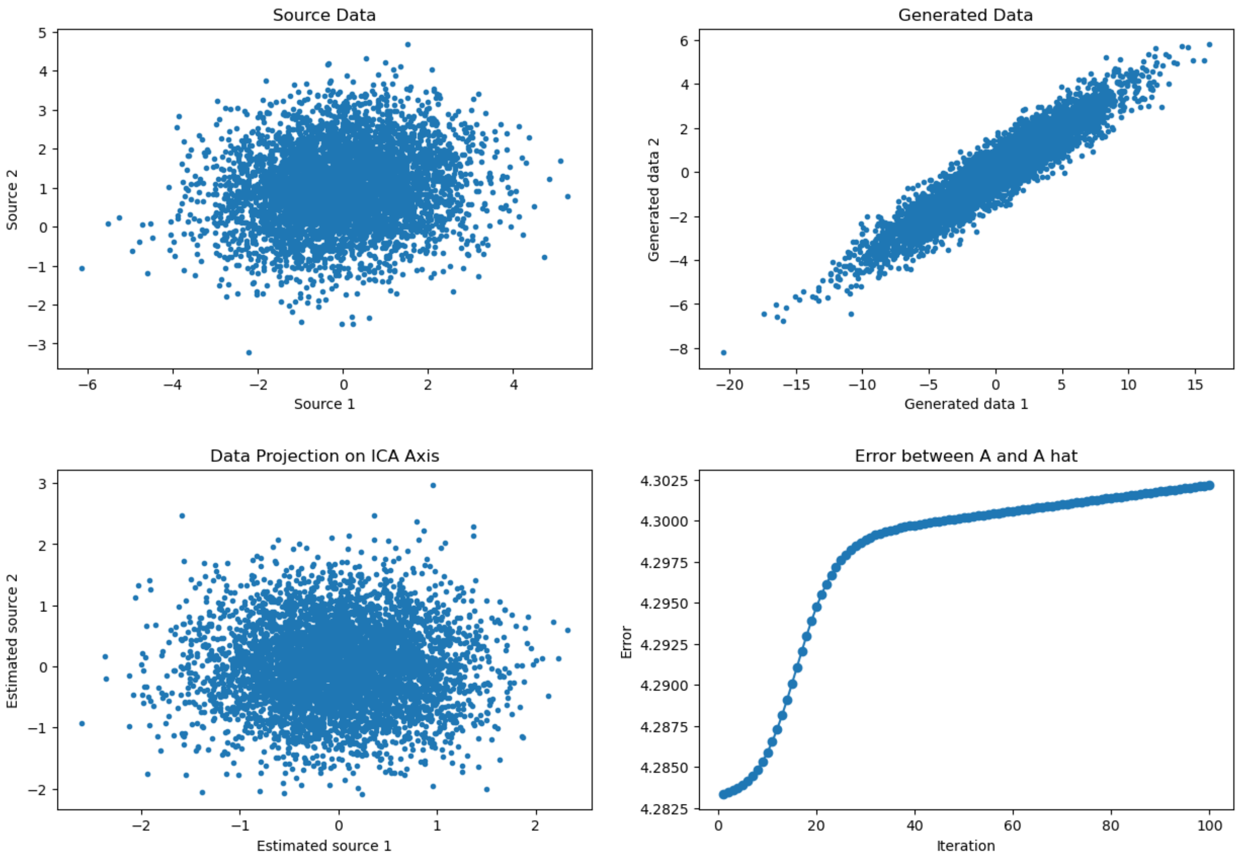

Figure 11: s is drawn from a multivariate normal distribution with $\mu = (0, 1)$, $\Sigma = [[20.25][0.251]]$.

4 Problem 3.4 Hidden Markov Models

4.1 Problem 3.4.1

- The number of the states is $K = 2$
- The initial probabilities matrix is $P_k = \begin{bmatrix} 0.6 \\ 0.4 \end{bmatrix}$
- The transition probabilites matrix is $P_{ij} = \begin{bmatrix} 0.9 & 0.1 \\ 0.35 & 0.65 \end{bmatrix}$ (As $P_{11} + P_{12} = P_{21} + P_{22} = 1$)
- The emission distributions matrix is

$$P(y|k) = \begin{bmatrix} 0.6 & 0.3 & 0.1 \\ 0.1 & 0.6 & 0.3 \end{bmatrix}$$

  The sum along each row should be equal to 1.

4.2 Problem 3.4.2

According to ML Sec. 16.5,

$$\alpha(x_1) = P(y_1, x_1) = P(y_1|x_1)P(x_1)$$

As we observe a_1 first:

$$\alpha(x_1) = P(y_1 = a_1|x - 1)P(x_1)$$

Then,

$$\begin{bmatrix}\alpha(x_1 = s_1) \\ \alpha(x_1 = s_2)\end{bmatrix} = \begin{bmatrix}P(y_1 = a_1|x_1 = s_1)P(x_1 = s_1) \\ P()y_1 = a_1|x_1 = s_2)P(x_1 = s_2)\end{bmatrix} = \begin{bmatrix}0.6 \cdot 0.6 \\ 0.1 \cdot 0.4\end{bmatrix}$$

$$\begin{bmatrix}\alpha(x_1 = s_1) \\ \alpha(x_1 = s_2)\end{bmatrix} = \begin{bmatrix}0.36 \\ 0.04\end{bmatrix}$$

As we know,

$$\alpha(x_2) = P(y_2|x_2)\Sigma_{x1}\alpha(x_1)P(x_2|x_1)$$

$$= P(y_2|x_2)[\alpha(x_1 = s_1)P(x_2|x_1 = s_1) + \alpha(x_1 = s_2)P(x_2|x_1 = s_2)]$$

Turn into a_2,

$$\begin{bmatrix}\alpha(x_2 = s_1) \\ \alpha(x_2 = s_2)\end{bmatrix} = \big[P(y_2|x_2)[\alpha(x_1 = s_1)P(x_2|x_1 = s_1) + \alpha(x_1 = s_2)P(x_2|x_1 = s_2)]\big]$$

As we then observe a_2:

$$\begin{bmatrix}\alpha(x_2 = s_1) \\ \alpha(x_2 = s_2)\end{bmatrix} = \begin{bmatrix}P(y_2 = a_2|x_2 = s_1)[\alpha(x_1 = s_1)P(x_2 = s_1|x_1 = s_1) + \alpha(x_1 = s_2)P(x_2 = s_1|x_1 = s_2)] \\ P(y_2 = a_2|x_2 = s_2)[\alpha(x_1 = s_1)P(x_2 = s_2|x_1 = s_1) + \alpha(x_1 = s_2)P(x_2 = s_2|x_1 = s_2)]\end{bmatrix}$$

$$\begin{bmatrix}\alpha(x_2 = s_1) \\ \alpha(x_2 = s_2)\end{bmatrix} = \begin{bmatrix}0.3 \cdot [0.36 \cdot 0.9 + 0.04 \cdot 0.35] \\ 0.6 \cdot [0.36 \cdot 0.1 + 0.04 \cdot 0.65]\end{bmatrix} = \begin{bmatrix}0.101 \\ 0.037\end{bmatrix}$$

As we know in the ML,

$$P(x_n|y_{[1:n]}) = \frac{\alpha(x_n)}{P(y_{[1:n]})}$$

With $P(y_{[1:n]}) = \Sigma_{x_n} p(y_{[1:n],x_n}) = \Sigma_{x_n}\alpha(x_n)$ Thus,

$$P(x_2 = s_1|y_{1:2}) = 0.732$$

4.3 Problem 3.4.3

$$\begin{aligned} P(y_3|x_2) &= \Sigma_{x_3} P(y_3, x_3|x_2) \quad \textit{with sum rule.} \\ &= \Sigma_{x_3} P(y_3|x_3, x_2) P(x_3|x_2) \quad \textit{with product rule} \\ &= \Sigma_{x_3} P(y_3|x_3) P(x_3|x_2) \quad \textit{because of the graphical model} \end{aligned}$$

$$\begin{aligned} P(y_3 = a_1|x_2 = s_1) &= p(y_3 = a_1|x_3 = s_1)p(x_3 = s_1|x_2 = s_1) + p(y_3 = a_1|x_3 = s_2)p(x_3 = s_2|x_2 = s_1) \\ &= 0.6 \cdot 0.9 + 0.1 \cdot 0.1 \\ &= 0.55 \end{aligned}$$

$$P(y_3 = a_1|x_2 = s_2) = 0.6 \cdot 0.35 + 0.1 \cdot 0.65 = 0.275$$

4.4 Problem 3.4.4

From ML eq.(16.49) we have: $P(x_n|y) = \frac{\alpha(x_n)\beta(x_n)}{p(y)}$
and from ML eq.(16.47) $\beta(x_2) = P(y_3|x_2)$
Thus we get:

$$\begin{aligned} P(x_2 = s_1|y_{1:3}) &= \frac{\alpha(x_2 = s_1)\beta(x_2 = s_1)}{P(y_{1:3})} \\ &= \frac{\alpha(x_2 = s_1)\beta(x_2 = s_1)}{\Sigma_{x_3}\alpha(x_3)} \end{aligned}$$

$$\beta(x_2 = s_1) = p(y_3 = a_1|x_2 = s_1) = 0.55$$

As the third observation with $Q3$ is a_1 Then $\alpha(x_3) = p(y_3|x_3)\Sigma_{x_2} p(x_3|x_2)\alpha(x_2)$

$$\begin{aligned} &\begin{bmatrix} \alpha(x_3 = s_1) \\ \alpha(x_3 = s_2) \end{bmatrix} \\ &= \begin{bmatrix} p(y_3 = a_1|x_3 = s_1)[p(x_3 = s_1|x_2 = s_1)\alpha(x_2 = s_1) + p(x_3 = s_1|x_2 = s_2)\alpha(x_2 = s_2)] \\ p(y_3 = a_1|x_3 = s_2)[p(x_3 = s_2|x_2 = s_1)\alpha(x_2 = s_1) + p(x_3 = s_2|x_2 = s_2)\alpha(x_2 = s_2)] \end{bmatrix} \\ &= \begin{bmatrix} 0.062 \\ 0.003 \end{bmatrix} \end{aligned}$$

$$\Sigma_{x_3}\alpha(x_3) = \alpha(x_3 = s_1) + \alpha(x_3 = s_2) = 0.062 + 0.03 = 0.065$$

Thus $P(x_2 = s_1|y_{1:3}) = \frac{0.101 \cdot 0.55}{0.065} = 0.855$

5 Problem 3.5 Kalman Filter

According to the question ask, the state vector:

$$x_n = \begin{bmatrix} p_n \\ v_n \end{bmatrix}$$

which the p_n is the position of the time n and the v_n is the velocity of the object at time n. Here, I use

$$F_n = \begin{bmatrix} 1 & dt \\ 0 & 1 \end{bmatrix}$$

which indicates that the velocity is constant without noise. Besides, I also use $dt = 0.1$. As the observation is only in the position, $H_n = [1 \quad 0]$. About η_n and v_n, I select them such as white noise with a standard deviation of 0.21. The result of the estimation on simulated data is displayed figure 13. The Kalman Filter seems to be able to predict approximately the position of the moving object correctly in the condition.

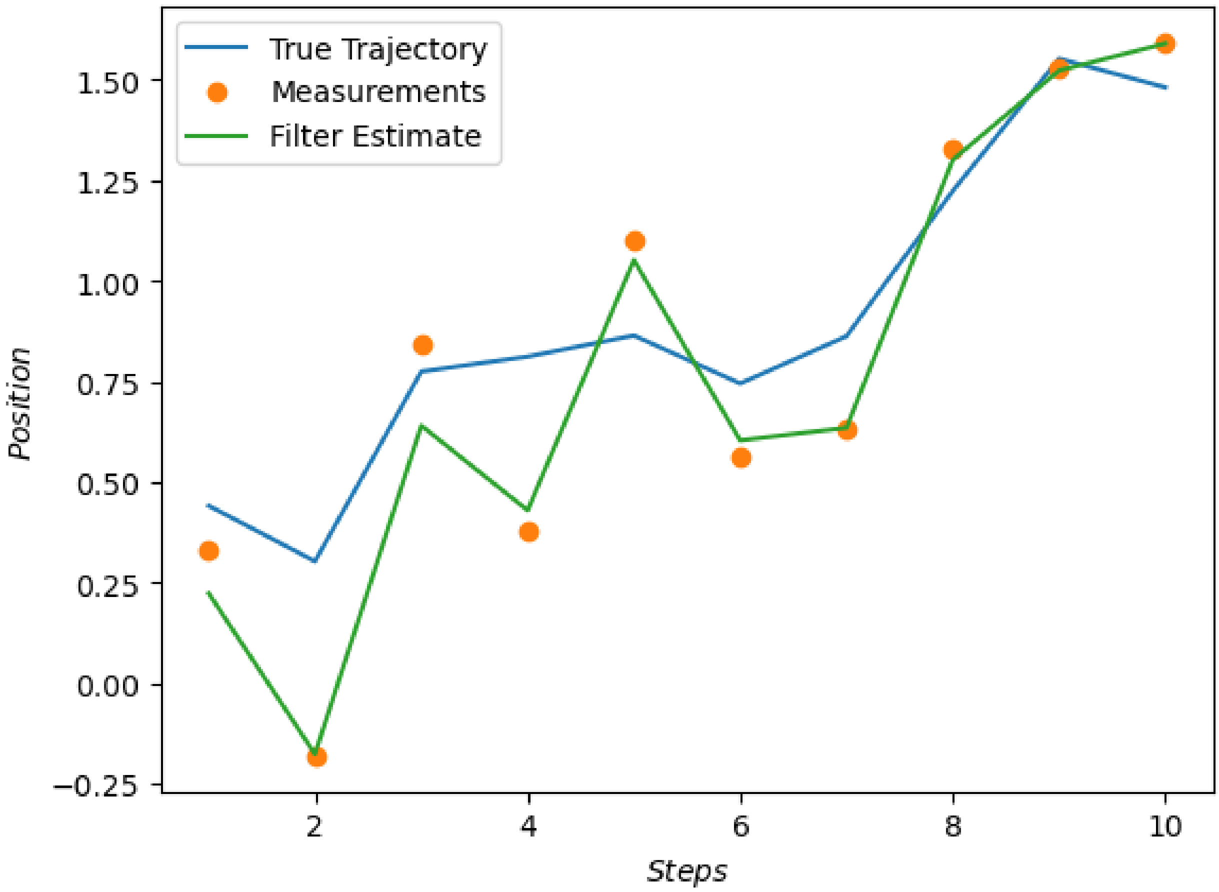

Figure 13: Performance of Kalman Filter for estimating the position of a moving object.

6 Problem 3.6 Kernel methods

6.1 Problem 3.6.1

We can see in figure 11 and figure 13 that the center location of the chirp is approximately $t = 5$.

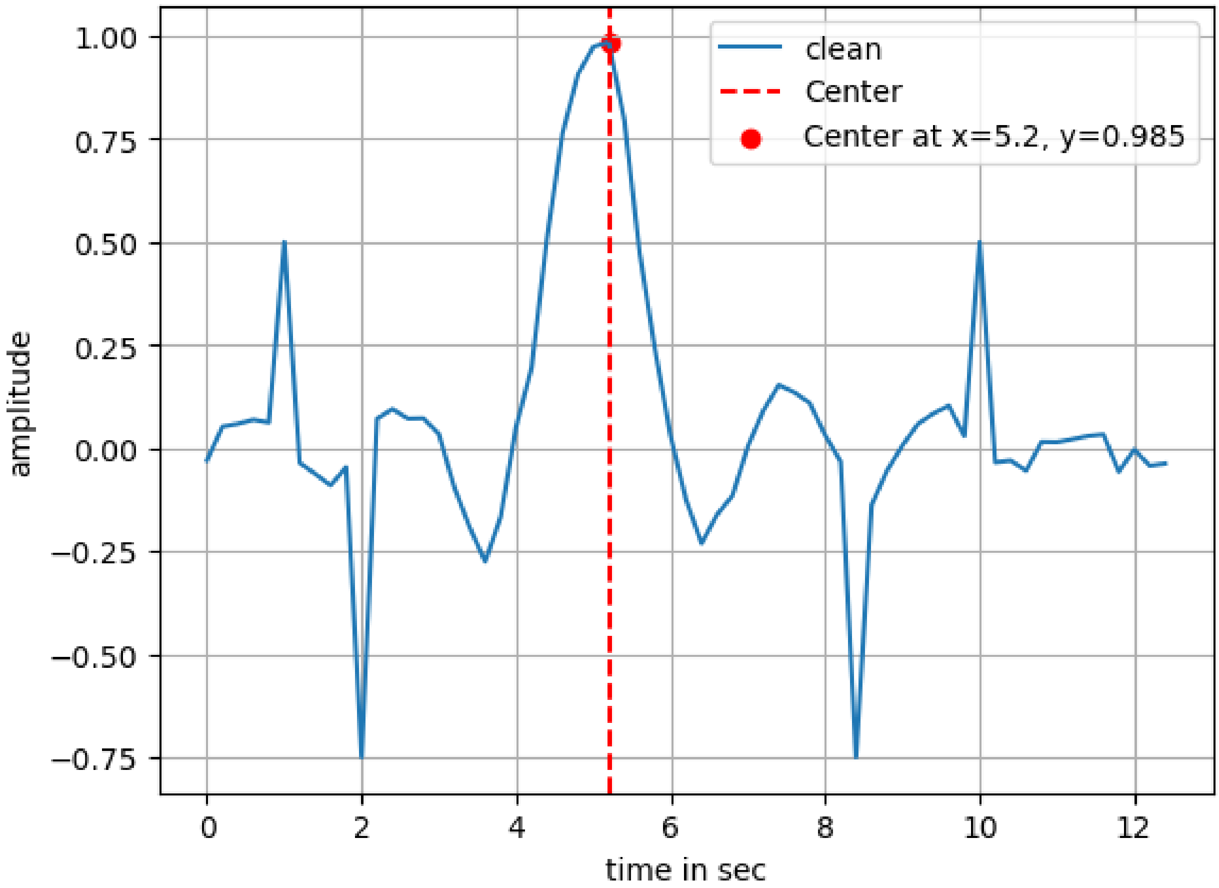

Figure 15: Sampled signal.

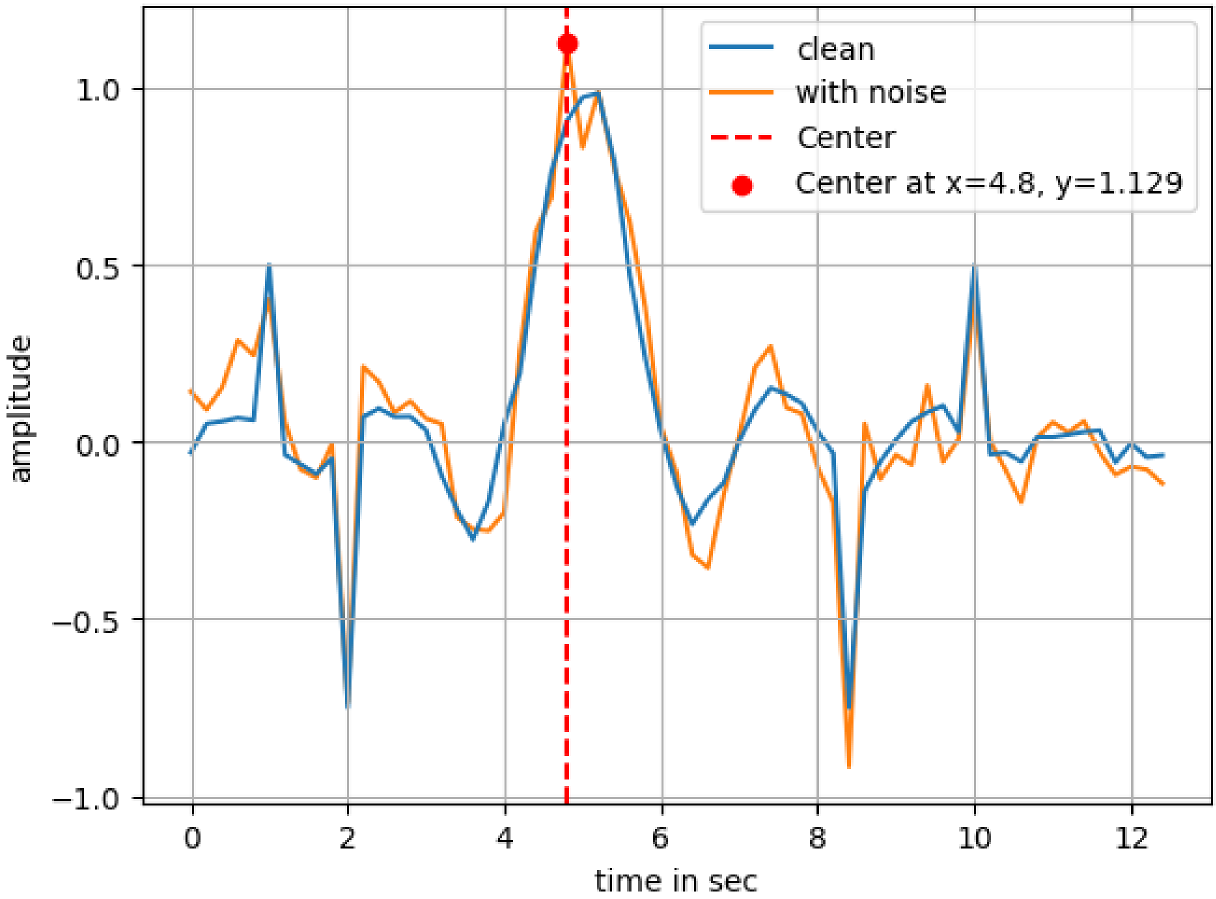

Figure 17: Sampled signal with random white noise.

6.2 Problem 3.6.2

The result of the estimation using Kernel Ridge Regression is available figure 19 and figure 21. I tried different couples of parameters but we can see that the estimation is still not perfect. However, it allows us to get a quite accurate estimation of the center location of the chirp which is approximately t = 5. Then, I calculated SNR using the estimated signal and white noise with a standard deviation of 0.004. Indeed, by looking at the data, the noise seems to have a standard deviation smaller than 0.004. Finally, I get SNR = 34.07.

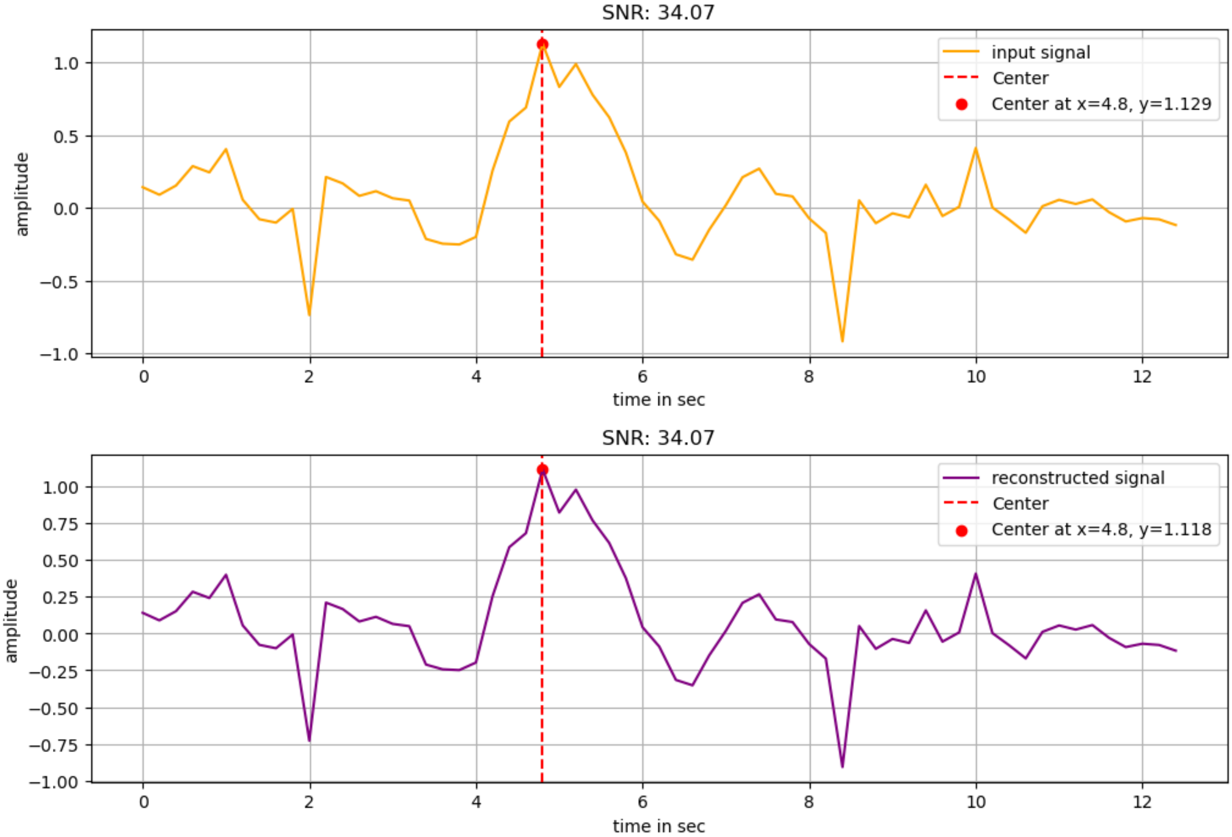

Figure 19: Reconstruct signal using Kernel Ridge Regression.
Parameters : $\sigma = 4e-3, C = 1e-2$.

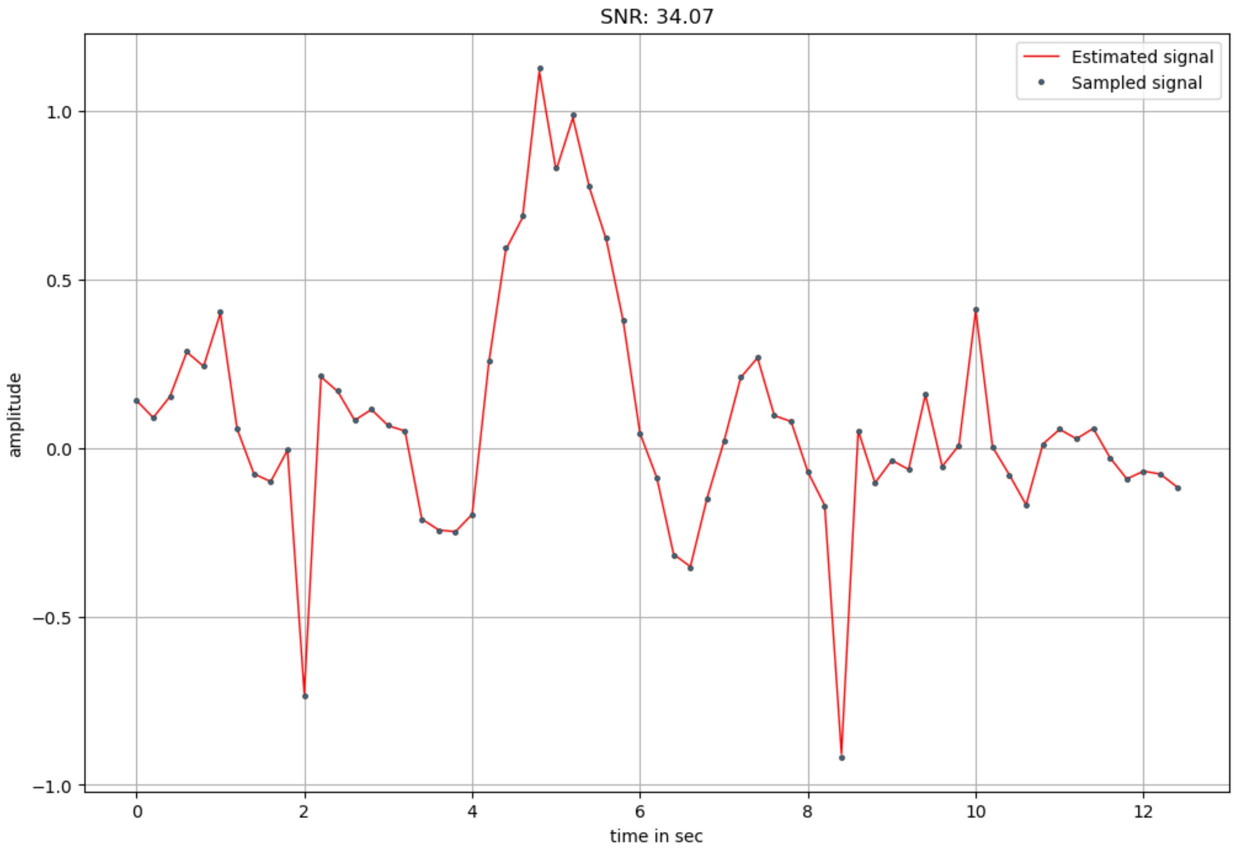

Figure 21: Estimated signal using Kernel Ridge Regression.
Parameters : $\sigma = 4e-3, C = 1e-2$.

6.3 Problem 3.6.3

The result of the estimation using SVR is available figure 23. We can see that SVR seems to perform better than Kernel Ridge Regression. It also allows us to get a quite accurate estimation of the center location of the chirp which is approximately t = 5. Then, I calculated SNR using the estimated signal and white noise with a standard deviation of 0.1. Indeed, by looking at the data, I decided that the standard deviation of the noise is actually smaller than 0.1. I get SNR = 47.3. For detecting the outliers, I set all points that are outside of a tube (around the estimate) of a radius 10 times bigger than ϵ as outsiders. However, the effect seems not very work on it.

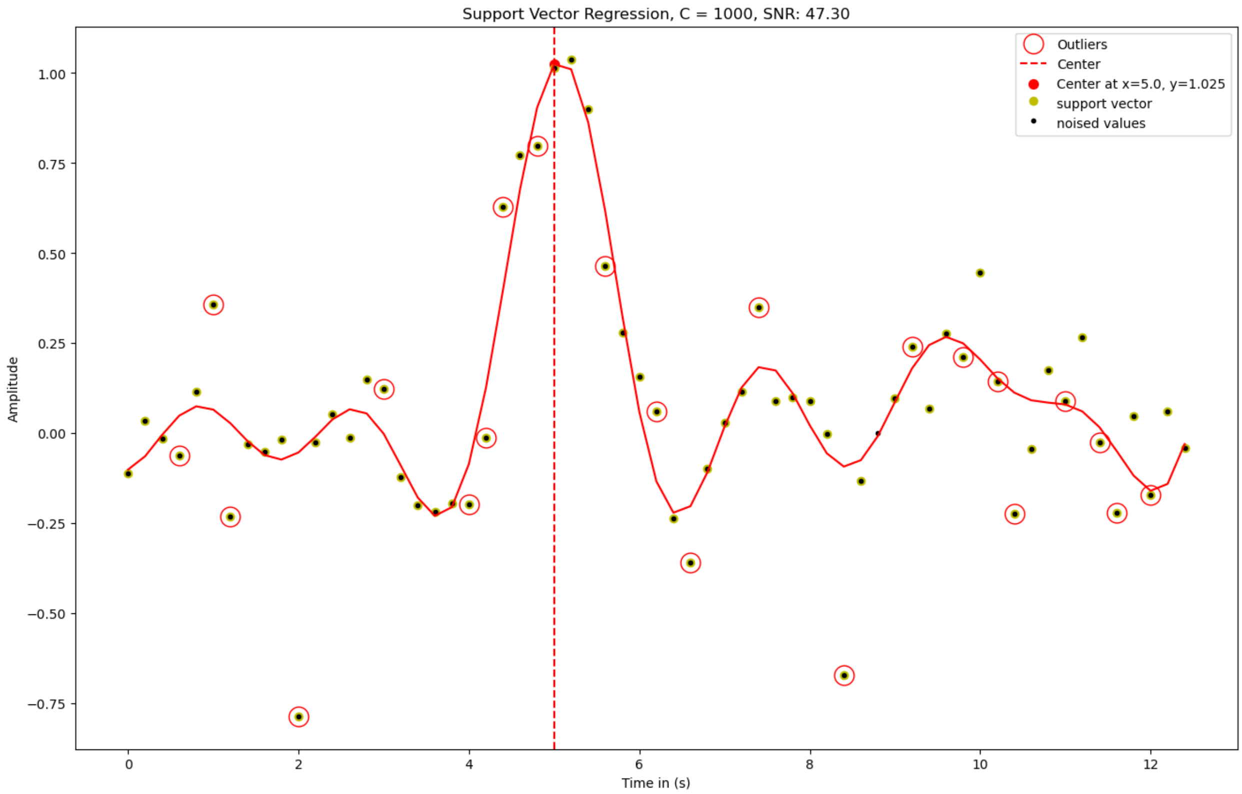

Figure 23: Estimated signal using support vector regression (SVR). Parameters : $\epsilon = 0.01$, $\sigma_{kernel} = 2$, $C = 1e3$

7 Problem 3.6 Kernel methods

8 code section

8.1 3.1.2

```
Problem_3_1_2.py > ...
Run Cell | Run Below | Debug Cell
#%%
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import Lasso
from scipy.fft import dct, idct
from scipy.io import loadmat

Run Cell | Run Above | Debug Cell
#%%
#Load data from problem3_1.mat
data = loadmat('/Users/luchengliang/ML_sp/Problem Set 3/probl
n = data['n'].flatten()
x = data['x'].flatten()

Run Cell | Run Above | Debug Cell
#%%
# Signal setup
N = 2**5  # number of observations to make
l = 2**9  # signal length

B = np.zeros((N, l))
for i in range(N):
    B[i, n[i]] = 1

# Since it is sparse in the IDCT domain, i.e. B*x = B*Phi*X =
# where X sparse,  BF = B*Phi; and Phi is the DCT matrix, Phi
# Equivalently, since IDCT = transpose of DCT using idct we c
BF = idct(B, norm='ortho')

lambda_ = 0.005
model = Lasso(lambda_, fit_intercept=False)
model.fit(BF, x)
solsB = model.coef_

# create IST solution
nsteps = 100000
```

```
    t_[:, k] = np.sign(t_tilde)*np.maximum(abs(t_tilde) - lambda_*mu, 0)
solsIST = t_[:, -1]

# Get K, ai, and mi
sols = solsIST*np.sqrt(2 / len(solsIST))  # normalize according to DCT specification
a = sols[np.nonzero(sols)]  # get ai values
k = np.count_nonzero(sols)  # get number of non-zero components in DCT domain
m = np.where(sols)[0]  # get positions of non-zero components
print("a_i values:", a)
print("K status:", k)
print("m_j values:", m)

# Reconstruct the multitone signal for verification
t = np.arange(0, l)  # time axis
s_multitone = np.zeros(l)
for i in range(k):
    s_multitone += a[i] * np.cos((np.pi * (2 * m[i] - 1) * t) / (2 * l))
# `s_multitone` now contains the reconstructed multitone signal

# plot solutions
fig, ax= plt.subplots(1, 1, figsize=(6, 3))
ax.stem(solsB, markerfmt='bo', label='sklearn Lasso', basefmt=' ')
ax.stem(solsIST, markerfmt='ro', label='IST', basefmt=' ')
ax.legend()
ax.set_title('Solutions')

# Take the inverse IDCT (i.e. the DCT) in order to compute the estimated signal.
x_hat = dct(solsIST, norm='ortho')
b_hat = dct(solsB, norm='ortho')

fig, ax= plt.subplots(3, 1, figsize=(12, 8))
ax[0].plot(b_hat)
ax[0].plot(n, x, 'r.')
ax[0].set_title('Estimated in time domain by LASSO')
ax[1].plot(x_hat)
ax[1].plot(n, x, 'r.')
ax[1].set_title('Estimated in time domain by IST')
ax[2].plot(s_multitone)
ax[2].set_title('Estimated signal reconstructed using multitone formula')
fig.tight_layout()
plt.show()
```

8.2 3.3

```
#%%
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

Run Cell | Run Above | Debug Cell
#%%
def column_switch(A, A_hat):
    # Compute differences and normalize columns
    diff = np.abs(A - A_hat)
    norm_diff = diff / np.linalg.norm(diff, axis=0)

    # Check if columns need to be switched in A_hat
    for i in range(A.shape[1] - 1):
        if np.sum(norm_diff[:, i]) > np.sum(norm_diff[:, i + 1]):
            # Switch columns in A_hat
            A_hat[:, [i, i + 1]] = A_hat[:, [i + 1, i]]

    return A_hat

Run Cell | Run Above | Debug Cell
#%%
def error_cal(A, A_hat):

    #Check if columns need to be switched in A_hat
    A_hat = column_switch(A, A_hat)

    # Normalize columns of A and A_hat
    A_normalized = A / np.linalg.norm(A, axis=0)
    A_hat_normalized = A_hat / np.linalg.norm(A_hat, axis=0)

    # Compute the absolute difference between normalized matrices
    diff = np.abs(A_normalized - A_hat_normalized)

    # Compute the sum of coefficients from the absolute difference
    error = np.sum(diff)
    return error

Run Cell | Run Above | Debug Cell
#%%
# Independent Component Analysis function
def ICA(x, mu, num_components, iters, mode, A):
```

```python
    # Random initialization
    W = np.random.rand(num_components, num_components)
    N = np.size(x, 0)

    if mode=='superGauss':
        phi = lambda u : 2*np.tanh(u)
    elif mode=='subGauss':
        phi = lambda u : u-np.tanh(u)
    else:
        print("Unknown mode")
        return W

    errors = []

    for i in range(iters):
        u = W @ x.T
        dW = (np.eye(num_components) - phi(u) @ u.T/N) @ W
        # Uniform distribution, so take average for E[]
        # Update
        W = W + mu*dW
        A_hat = W.T
        error = error_cal(A, A_hat)
        errors.append(error)

    return W, errors

Run Cell | Run Above | Debug Cell
#%%
def Find_the_Source(s):

    # Mix signals
    A = np.array([[3, 1], [1, 1]])
    x = (A@s).T
    r = s.T

    # calculate ica
    mu = 0.1
    components = 2
    iterations = 100

    # Mean across the first (column) axis
    col_means = np.mean(x, axis=0)
```

```python
    x = x - col_means

    # run ICA
    W, errors = ICA(x, mu, components, iterations, 'subGauss', A)

    # Normalize unmixing matrix
    W = np.divide(W, np.max(W))

    # Compute unmixed signals
    y = (W@x.T).T


    # Plotting
    plt.figure(figsize=(15, 10))

    plt.subplot(2, 2, 1)  # Subplot 1: Source Data
    plt.plot(r[:, 0], r[:, 1], '.')
    plt.xlabel('Source 1')
    plt.ylabel('Source 2')
    plt.title('Source Data')

    plt.subplot(2, 2, 2)  # Subplot 2: Generated Data
    plt.plot(x[:, 0], x[:, 1], '.')
    plt.xlabel('Generated data 1')
    plt.ylabel('Generated data 2')
    plt.title('Generated Data')

    plt.subplot(2, 2, 3)  # Subplot 3: Data Projection on ICA Axis
    plt.plot(y[:, 0], y[:, 1], '.')
    plt.xlabel('Estimated source 1')
    plt.ylabel('Estimated source 2')
    plt.title('Data Projection on ICA Axis')

    iters = np.arange(1, iterations + 1)
    plt.subplot(2, 2, 4)  # Subplot 4: Error by Iteration
    plt.plot(iters, errors, marker='o')
    plt.xlabel('Iteration')
    plt.ylabel('Error')
    plt.title('Error between A and A hat')

    plt.subplots_adjust(hspace=0.3)
    plt.show()
```

```
Run Cell | Run Above | Debug Cell
#%%
# generate data

N = 5000

# Define two non-gaussian uniform components
s1 = np.random.rand(N)
s2 = np.random.rand(N)
s = np.array(([s1, s2]))

# Define one non-gaussian uniform component and one beta component
s1b = np.random.rand(N)
s2b = np.random.beta(0.1, 0.1, size=N)
sb = np.array(([s1b, s2b]))

# Define one non-gaussian uniform component and one gaussian component
s1n = np.random.rand(N)
s2n = np.random.normal(size=N)
sn = np.array(([s1n, s2n]))

#Define multivariate normal distribution with
#μ = (0, 1), Σ = [2 0.25; 0.25 1]
mean = [0, 1]
covariance = [[2, 0.25], [0.25, 1]]
sm_r = np.random.multivariate_normal(mean, covariance, N)
sm = sm_r.T

Find_the_Source(s)
Find_the_Source(sb)
Find_the_Source(sn)
Find_the_Source(sm)

```

8.3 3.5

```python
#%%
import numpy as np
import matplotlib.pyplot as plt

# Set the parameters

q = 1
dt = 0.1
s = 0.21
F = np.array([
    [1, dt],
    [0, 1],
])
Q = q*np.array([
    [s**2, 0],
    [0,    0]
])

H = np.array([1, 0])
R = s**2*np.identity(2)
m0 = np.array([[0], [1]])
P0 = np.identity(2)

```

```
# Simulate data

np.random.seed(1)

steps = 10
X = np.zeros((len(F), steps))
Y = np.zeros((len(H), steps))
x = m0
for k in range(steps):
    x = F@x + s*np.random.randn(len(F), 1)
    y = H@x + s*np.random.randn(1, 1)
    X[:, k] = x[:, 0]
    Y[:, k] = y[:, 0]


# Kalman filter

m = m0
```

```
P = P0
kf_m = np.zeros((len(m), Y.shape[1]))
kf_P = np.zeros(( (property) shape: _Shape e[1]))
for k in range(Y.shape[1]):
    m = F@m
    P = F@P@F.T + Q

    e = Y[:, k].reshape(-1, 1) - H@m
    S = H@P@H.T + R
    K = P@H.T@np.linalg.inv(S)
    m = m + K@e
    P = P - K@S@K.T

    kf_m[:, k] = m[:, 0]
    kf_P[:, :, k] = P

a = np.arange(1,11)
plt.figure()
plt.plot(a, X[0, :], '-')
plt.plot(a, Y[0, :], 'o')
plt.plot(a, kf_m[0, :], '-')
plt.legend(['True Trajectory', 'Measurements', 'Filter Estimate'])
plt.xlabel('$Steps$')
plt.ylabel('$Position$')
'''
plt.figure()
plt.plot(X[0, :], X[1, :], '-')
plt.plot(Y[0, :], Y[1, :], 'o')
plt.plot(kf_m[0, :], kf_m[1, :], '-')
plt.legend(['True Trajectory', 'Measurements', 'Filter Estimate'])
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
'''
```

8.4 3.6

```python
#%%
Run Cell | Run Above | Debug Cell
#%%
import os
from scipy.io import loadmat
import numpy as np
import soundfile as sf
import matplotlib.pyplot as plt
Run Cell | Run Above | Debug Cell
#%%

# Load bladerunner data
data = loadmat('./problem3_6.mat')
t = data['t'].flatten()
y = data['y'].flatten()

# Calculate the center of the signal
max_value = np.max(y)
ind = np.argmax(y)
tmax = t[ind]
max_value_abbrev = round(max_value, 3)

#Initial figure and plot the signal and its center point
fig, ax = plt.subplots()
ax.plot(t, y, label='clean')
ax.axvline(tmax, color='r', linestyle='--', label='Center')
ax.scatter(tmax, max_value, color='red', label=f'Center at x={tmax}, y={max_value_abbrev}')
ax.set_xlabel('time in sec')
ax.set_ylabel('amplitude')
ax.legend()
ax.grid()


#Initial another figure
fig, ax = plt.subplots()
y_original = y
ax.plot(t, y_original, label='clean', zorder=1)

# data parameters
np.random.seed(0)
N = t.size
```

```
percent_outlier = 0.1
snr = 10  # dB

# learning parameters
sigma = 0.004
C = 1e-2

# add white Gaussian noise
noise = np.random.randn(N)
noise *= (np.sum(y**2)/np.sum(noise**2)/10**(snr/10))**0.5
y += noise
ax.plot(t, y, label='with noise', zorder=0)

# Calculate the new center of the signal with noise
max_value_mednoise = np.max(y)
ind_mednoise = np.argmax(y)
tmax_mednoise = t[ind_mednoise]
tmax_mednoise_abbrev = round(tmax_mednoise, 1)
max_value_mednoise_abbrev = round(max_value_mednoise, 3)

# finish figure
ax.axvline(tmax_mednoise, color='r', linestyle='--', label='Center')
ax.scatter(tmax_mednoise, max_value_mednoise, color='red', label=f'Center at x={tmax_mednoise_abbrev}, y={max_value_mednoise_abbrev}')
ax.set_xlabel('time in sec')
ax.set_ylabel('amplitude')
ax.legend()
ax.grid()
Run Cell | Run Above | Debug Cell
#%%
# unbiased L2 Kernel Ridge Regression (KRR-L2)
# build kernel matrix
pair_dist = np.abs(t.reshape(-1, 1) - t.reshape(1, -1))
K = np.exp(-1/(sigma**2)*pair_dist**2)
A = C*np.identity(N) + K
sol = np.linalg.solve(A, y)

# Generate regressor
# NOTE: this loop can be optimized
samples = t[-1]
x = np.arange(0, samples + 0.2, 0.2)
M2 = len(x)
z0 = np.zeros(M2)
for k in range(M2):
    z0[k] = 0
    for j in range(N):
        value = np.exp(-1/(sigma**2)*(t[j] - x[k])**2)
        z0[k] += sol[j]*value

```

```
# Get the center of the chirp
max_value_re = np.max(z0)
ind_re = np.argmax(z0)
tmax_re = x[ind_re]
tmax_re_abbrev = round(tmax_re, 1)
max_value_re_abbrev = round(max_value_re, 3)
# Compute SNR
SNR = np.var(z0) / (0.2 ** 2 * np.var(np.random.rand(1, len(z0))))

# plot
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 8))
ax1.set_xlabel('time in sec')
ax1.set_ylabel('amplitude')
ax1.plot(t, y, label='input signal', color='orange')
ax1.axvline(tmax_mednoise, color='r', linestyle='--', label='Center')
ax1.scatter(tmax_mednoise, max_value_mednoise, color='red', label=f'Center at x={tmax_mednoise_abbrev}, y={max_value_mednoise_abbrev}')
ax1.legend()
ax1.set_title(f'SNR: {SNR:.2f}')
ax1.grid()

ax2.set_xlabel('time in sec')
ax2.set_ylabel('amplitude')
ax2.plot(x, z0, label='reconstructed signal', color='purple')
ax2.axvline(tmax_re, color='r', linestyle='--', label='Center')
ax2.scatter(tmax_re, max_value_re, color='red', label=f'Center at x={tmax_re_abbrev}, y={max_value_re_abbrev}')
ax2.legend()
ax2.set_title(f'SNR: {SNR:.2f}')
ax2.grid()

plt.subplots_adjust(hspace=0.3)
plt.show()

fig, ax = plt.subplots(figsize=(12, 8))
ax.set_xlabel('time in sec')
ax.set_ylabel('amplitude')
ax.plot(x, z0, 'r', linewidth=1, label='Estimated signal')
ax.plot(t, y, '.', markeredgecolor=0.3*np.array([1, 1, 1]), markersize=5, label='Sampled signal')
ax.legend()
ax.set_title(f'SNR: {SNR:.2f}')
ax.grid()
```

```python
#%%
import numpy as np
from scipy.io import loadmat
import librosa
import librosa.display
import matplotlib.pyplot as plt
from random import randrange
from sklearn.svm import SVR
Run Cell | Run Above | Debug Cell
#%%
def awgn(signal, snr):
    x_watts = signal ** 2
    # Set a target SNR
    target_snr_db = snr
    # Calculate signal power and convert to dB
    sig_avg_watts = np.mean(x_watts)
    sig_avg_db = 10 * np.log10(sig_avg_watts)
    # Calculate noise according to [2] then convert to watts
    noise_avg_db = sig_avg_db - target_snr_db
    noise_avg_watts = 10 ** (noise_avg_db / 10)
    # Generate an sample of white noise
    mean_noise = 0
    noise_volts = np.random.normal(mean_noise, np.sqrt(noise_avg_watts), len(x_watts))
    # Noise up the original signal
    y_volts = signal + noise_volts
    return y_volts

#Load the data
data = loadmat('./problem3_6.mat')
t = data['t'].flatten()
y = data['y'].flatten()
x = t

# parameters
N=t.size
snr = 10 #dB
percent_outlier = 0.1

# learning parameters
epsilon=0.01
kernel_type='Gaussian'
```

```python
kernel_params=2
C=1e3

# Add white Gaussian noise
y_noised = awgn(y, snr)

# convert data to proper dimensions in order to fit requirements of the library
x_col = x.reshape(( np.size(x), 1))
y_row = np.copy(y_noised)
t_col = t.reshape(( np.size(t), 1))

t_col = np.around(t_col, decimals=4)
x_col = np.around(x_col, decimals=4)
y_row = np.around(y_row, decimals=4)

# ---------- Support Vectore Regression -----------
gamma = 1/(np.square(kernel_params)) # gamma needs to be calculated in order to use 'Gaussian' kernel, which is not available in the library
regressor = SVR(kernel='rbf', gamma=gamma, C=C, epsilon=epsilon)

regressor.fit(x_col,y_row)
y_pred = regressor.predict(t_col)


# Find outliers using threshold
threshold = 10 * epsilon
outsider = np.zeros(len(t))
for i, sv_index in enumerate(regressor.support_):
    j = np.where(x_col == x_col[sv_index])[0][0]
    if abs(y_row[sv_index] - y_pred[j]) > threshold:
        outsider[i] = 1

outsider = outsider.astype(bool)

# Get center of the chirp
max_value = np.max(y_pred)
ind = np.argmax(y_pred)
t_max = x[ind]
tmax_abbrev = round(t_max, 1)
max_value_abbrev = round(max_value, 3)

SNR = np.var(y_pred) / np.var(0.2 ** 2 * np.random.randn(len(y_pred)))


```

```python
# plot
plt.figure(figsize=(16,10))
plt.stem(x_col[regressor.support_], y_row[regressor.support_], linefmt = 'none', markerfmt='yo', label='support vector', basefmt=" "
            , use_line_collection=True)
plt.stem(x_col, y_row,  linefmt = 'none', markerfmt='k.', label='noised values', basefmt=" ", use_line_collection=True)
plt.plot(t_col, y_pred, color = 'red')
# Plot support vectors and outliers
plt.plot(x_col[outsider], y_row[outsider], 'o', markerfacecolor='none', markersize=15, color='r', label='Outliers')
#plt.plot(x_col[regressor.support_], y_row[regressor.support_], 'o', markerfacecolor='none', markersize=7, color='g', label='Support Vectors')
plt.axvline(t_max, color='r', linestyle='--', label='Center')
plt.scatter(t_max, max_value, color='red', s=50, label=f'Center at x={tmax_abbrev}, y={max_value_abbrev}')
plt.title("Support Vector Regression, C = %d, " % C + f'SNR: {SNR:.2f}')
plt.xlabel("Time in (s)")
plt.ylabel("Amplitude")
plt.legend()
plt.show()
```

List of Figures