EE 263 Midterm

Luis A. Perez

Fitting a model for hourly temperature

Solution:

(a) Let our capture image be $Y = \begin{pmatrix} y_1 & \cdots & y_N \end{pmatrix} \in \mathbb{R}^{2\times N}$. The rotation by angle θ and positive $\alpha > 0$ for face i would correspond to the operation:

$$Y = RF^{(i)}$$

So the tasks consists of finding the i for which the above equation holds. The other complicating aspect is that we don't know R. However, we propose the following algorithm to find a candidate matrix $\mathbb{R}$.

- Take the j-th facial feature from the i-th example, $z_1 = F^i_j \in \mathbb{R}^2$ and the corresponding facial feature of our output face, $z_2 = Y_j \in \mathbb{R}^2$ (these are just the j-th columns of F^i and Y).
- Find a rotation and scaling matrix $\hat{R}$ between these two vectors, z_1, z_2. Note that this is possible as long as these two vectors are non-zero. However, in the case where $z_1 = 0, z_2 \neq 0$, no transformation exists, and if $z_1 \neq 0, z_2 = 0$, the only possible transformation is $\hat{R} = 0$ (the zero-matrix). Lastly, if both are 0, then the trivial must have $\hat{R} = I$.

  As such, for the cases where we know both $z_1 \neq 0, z_2 \neq 0$, we know that z_1 can always be transformed into z_2 by rotating and scaling. In more detail. to find this candidate matrix $\hat{R}$, we need to know (1) the scaling factor α and (2) the angle of rotation θ. We have:

$$\alpha = \left| \frac{||z_2||}{||z_1||} \right|$$

since it is just the ratio of the legths. The angle of rotation is just slightly trickier, but can be computed using the following relation

$$\theta = \cos^{-1}\left(\frac{z_2^T z_1}{||z_1||||z_2||} \right)$$

Note that this actually gives us two candidate angles, the other being $2\pi - \theta$. With this information, we can construct two candidate rotation/scaling matrices $\hat{R}_1$ and $\hat{R}_2$. To select the correct one, we simply try both and find the k such that:

$$Y_j = \hat{R}_k F_j^{(i)}$$

holds.

- Now that we've narrowed it down to a single $\hat{R}$ candidate matrix, we just check if

$$Y = \hat{R} F^{(i)}$$

If the above hold, we are done, and have found that the i-th example face corresponds to the capture face. If it does not hold, we continue tot he next example face.

(b) We apply the above method (see attached code). We find that Y_{rot} corresponds to F_2.

(c) Minmizing ρ correspond to minimizing the objective function:

$$J = \sum_{i=1}^{N} ||y_i - Rx_i^{(k)} - t||^2$$

We focus first on finding the optimal value of t. We can do this by taking the derivative with respect to t and setting equal to 0, which is straight-forward. We have:

$$\begin{aligned} -2\sum_{i=1}^{N}(y_i - Rx_i^{(k)} - t) &= 0 \\ \implies \sum_{i=1}^{N} y_i - R\sum_{i=1}^{N} x_i^{(k)} &= tN \\ \implies t &= \frac{1}{N}\sum_{i=1}^{N} y_i - R\frac{1}{N}\sum_{i=1}^{N} x_i^k \\ &= \bar{y} - R\bar{x} \end{aligned}$$

where:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i^{(k)}$$

$$\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$$

Plugging this optimal value of t into our objective, we have:

$$J = \sum_{i=1}^{N} ||y_i - Rx_i^{(k)} - \bar{y} + R\bar{x}||^2 = \sum_{i=1}^{N} ||(y_i - \bar{y}) - R(x_i^{(k)} - \bar{x})||^2$$

So with this transformation, the problem once again just becomes finding the optimal scaling and rotation matrix R to find the given data (and minimize the objective) where we have $\tilde{y}_i = y_i - \bar{y}$ and $\tilde{x}_i = x_i^{(k)} - \bar{x}$.

As such, focusing on this new problem of estimating R to minimize the above, let us see if we can simplify. We have $R = \alpha R'$ where R' is orthonormal. We have:

$$\begin{aligned} ||\tilde{y}_i - \alpha R'\tilde{x}_i^{(k)}||^2 &= (\tilde{y}_i - \alpha R'\tilde{x}_i)^T(\tilde{y}_i - \alpha R'\tilde{x}_i) \\ &= \tilde{y}_i^T\tilde{y}_i + \alpha^2\tilde{x}_i^T R'^T R'\tilde{x}_i - 2\alpha\tilde{y}_i^T R'\tilde{x}_i \\ &\qquad\qquad \text{(Last term is scalar, so } a^T = a\text{)} \\ &= \tilde{y}_i^T\tilde{y}_i + \alpha^2\tilde{x}_i^T\tilde{x}_i - 2\alpha\tilde{y}_i^T R'\tilde{x}_i \qquad (R'^T R' = I) \end{aligned}$$

Since we're trying to minimize this with respect to R', our minimization can be simplified greatly (the first two terms above don't depend on R') and we can drop scalars. As such, we have our new objective the maximization of:

$$\sum_{i=1}^{N} \tilde{y}_i^T R'\tilde{x}_i$$

We can rewrite the above entirely in matrix notation as:

$$\text{trace}(Y^T R' X) = \text{trace} R' X Y^T$$

Then computing the singular-value decomposition of $S = XY^T = U\Sigma V^T$, we have:

$$\text{trace}(R'XY^T) = \text{trace}(R'U\Sigma V^T) = \text{trace}\Sigma V^T R'U$$

IN this last step, let us note that V, R' and U are all orthogonal matrices (by definition of rotation and by SVD), as such, we have $(V^T RU)_{ij} \leq 1$. This means that the only way to maximize the above objective (by manipulating R'), racalling that Σ is a diagonal with non-negative entries, is to have the diaginal entries of $(V^T R'U)_{ii} = 1$, which implies the following:

$$\begin{aligned} & I = V^T R'U \\ \implies\ & R' = VU^T \qquad \text{(Inverse of orthonormal matrices is transpose)} \end{aligned}$$

We know have a formula for computing R' and t. The only variable left to figure out is α. Going back to our simplified objective, we are now trying to minimize:

$$\alpha^2 \sum_{i=1}^{n} \tilde{x}_i^T\tilde{x}_i - 2\alpha \sum_{i=1}^{N} \tilde{y}_i^T R'\tilde{x}_i$$

Taking the derivative with respect to α and setting equal to zero, we have:

$$2\alpha \sum_{i=1}^{N} \tilde{x}_i^T \tilde{x}_i = 2 \sum_{i=1}^{N} \tilde{y}_i^T R' \tilde{x}_i$$
$$\implies \alpha = \frac{\sum_{i=1}^{N} \tilde{y}_i^T R' \tilde{x}_i}{\sum_{i=1}^{N} \tilde{x}_i^T \tilde{x}_i}$$

With the above, we have completed the process of finding all the required parameters.

(d) See the attached code for details. These are the results:

Scale rotation + translation estimates for F_1

```
R = [[ 1.38710025 -0.50402031]
 [ 0.50402031  1.38710025]]
t = [[0.26475362]
 [0.07103614]]
```

Scale rotation + translation estimates for F_2

```
R = [[ 1.36865856 -0.49728905]
 [ 0.49728905  1.36865856]]
t = [[0.27399331]
 [0.04565355]]
```

Scale rotation + translation estimates for F_3

```
R = [[ 1.38049264 -0.49966651]
 [ 0.49966651  1.38049264]]
t = [[0.27343758]
 [0.05007972]]
```

Scale rotation + translation estimates for F_4

```
R = [[ 1.42094548 -0.5162301 ]
 [ 0.5162301   1.42094548]]
t = [[0.25391536]
 [0.10100179]]
```

Scale rotation + translation estimates for F_5

```
R = [[ 1.33411965 -0.48425504]
 [ 0.48425504  1.33411965]]
t = [[0.27703282]
 [0.03803171]]
```

We now have the RMSEs:

- RMSE for F_1 is 1.215783026538849
- RMSE for F_2 is 1.202885795582196
- RMSE for F_3 is 1.2080822140116791
- RMSE for F_4 is 1.2321052401027686
- RMSE for F_5 is 1.1978958153633974

(e) The method used in part (b) actually requires a lower decimal point precission, since it involves no inversion of calculation of matrix decompositions. In fact, the method in part (b) was selected with the explicit knowledge that no noise would be found in the image, and as such, we could take advantage of that.

As for the method in part (d), this also does not rely on matrix inversion (which would throw our condition value quite high for the matrices). Instead, we're taking advange of the fact that what we want is a rotation matrix, and can therefore make use of SVD, which has lower precision requirements compared to matrix inversion.

See attached code for the figure.

Solution:

(a) For the weighed least squares problem, we know that we can actualy rewrite the objective back into matrix form. Our objective is:

$$\sum_{i=1}^{n} w_i(\tilde{a}_i^T x - y_i)^2 = \sum_{i=1}^{n}(\sqrt{w_i}\tilde{a}_i^T x - \sqrt{w_i}y_i)^2$$

The above can be written in the form:

$$||WAx - Wa||_2^2$$

where the matrix $W \in \mathbb{R}^{m\times m}$ is given by:

$$W = \begin{bmatrix} w_1 & 0 & \cdots & 0 \\ 0 & w_2 & \cdots & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \cdots & w_m \end{bmatrix}$$

Where w_i lies in the diagonal. For convenience, we define $\sqrt{W}$ as the matrix W where each diagonal entry has been square-rooted. Since the w_i are given, we can just consider these to be part of our system (eg, consider $\sqrt{W}A$ as $\tilde{A}$ and $\sqrt{W}y$ as $\tilde{y}$, the output of our system).

With the phrasing, as long as $\sqrt{W}A$ is also skinny and full-rank (and given that A is skinny and full rank, the only additional requirement would be that W be full-rank). Since W is a diagonal matrix, full-rank is possible only when $w_i > 0$ for all i. This are the conditions given in our problem statement.

As such, the least squares solution can be computed directly using $\tilde{A}$ and $\tilde{y}$:

$$\begin{aligned}\hat{x} &= (\tilde{A}^T\tilde{A})^{-1}\tilde{A}^T\tilde{y} \\ &= ((\sqrt{W}A)^T(\sqrt{W}A))^{-1}(\sqrt{W}A)^T(\sqrt{W}y) \\ &= (A^TWA)^{-1}A^Ty\end{aligned}$$

which is solve-able as long as $w_i > 0$ for all i and A is skinny and full rank.

(b) We can shose $w_i(x^{(k)}$ as follows:

$$w_i(x^{(k)}) = \frac{|\tilde{a}_i^Tx^{(k)} - y_i|}{(\tilde{a}_i^Tx^{(k)} - y_i)^2}$$

This means that at each iteration, we'll be finding $x^(k+1)$ which minimizes:

$$\begin{aligned}\sum_{i=1}^m w_i(x^{(k)})(\tilde{a}_i^Tx^{(k+1)} - y_i)^2 &= \sum_{i=1}^m \frac{|\tilde{a}_i^Tx^{(k)} - y_i|}{(\tilde{a}_i^Tx^{(k)} - y_i)^2}(\tilde{a}_i^Tx^{(k+1)} - y_i)^2 \\ &= \sum_{i=1}^m \frac{|\tilde{a}_i^Tx - y_i|}{(\tilde{a}_i^Tx - y_i)^2}(\tilde{a}_i^Tx - y_i)^2 \approx \\ &\qquad \text{(At convergence, we should have } x^{(k)} = x^{(k+1)} = x\text{)} \\ &= \sum_{i=1}^m |\tilde{a}_i^Tx - y_i|\end{aligned}$$

As such, with the given weights, the solution will optimize the objective function desired.

(c) We are asked to explicitly define the matrix D. It is not specified in the problem statement, but we assume that $\hat{x} = \begin{bmatrix}\hat{x}_1\end{bmatrix}$ We have:

$$D = \begin{bmatrix} -1 & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & -1 & 1 \end{bmatrix} \in \mathbb{R}^{(n-1)\times n}$$

Essentially, we have $D_{ii} = -1$ and $D_{i(i+1)} = 1$ for $i = 1, \cdots, n-1$.

(d) We follow the lecture note on this one, pretty closely. We basically want to minimize the following:

$$||\hat{x} - x_{\text{noisy}}||_2^2 + \mu||D\hat{x}||_2^2$$

Assume we're given a value for μ. To match the structure presented during lecture, we now that the above is equivalent to the problem:

$$||Ax - y||_2^2 + \mu||Fx - g||_2^2$$

where $x = \hat{x} \in \mathbb{R}^{1000}$, $y = x_{\text{noisy}} \in \mathbb{R}^{1000}$, $A = I \in \not{\mathbb{R}}^{\not{1}\not{0}\not{0}\not{0} \times \not{1}\not{0}\not{0}\not{0}}$ (the identity matrix), $F = D \in \not{\mathbb{R}}^{\not{9}\not{9}\not{9} \times \not{1}\not{0}\not{0}\not{0}}$, and $g = 0 \in \not{\mathbb{R}}^{\not{9}\not{9}\not{9}}$ (the zero vector). As such, we can express the multi-objective system as an ordinarly LS problem where we're minimizing:

$$||\tilde{A}x - \tilde{y}||_2^2$$

where

$$\tilde{A} = \begin{bmatrix} I \\ \sqrt{\mu}D \end{bmatrix} \in \mathbb{R}^{1999 \times 1000}$$

$$\tilde{y} = \begin{bmatrix} x_{\text{noisy}} \\ 0 \end{bmatrix} \in \mathbb{R}^{1999}$$

Assuming $\tilde{A}$ is full-rank, the least squares solution would then be given by:

$$\begin{aligned} \hat{x} &= (\tilde{A}^T\tilde{A})^{-1}\tilde{A}^T\tilde{y} \\ &= (I^TI + \mu D^TD)^{-1}(I^Tx_{\text{noisy}} + \mu D^T0) \\ &= (I + \mu D^TD)^{-1}x_{\text{noisy}} \end{aligned}$$

With the above, we sweep over μ for the both scenarios given. See Figure 1 for a plot of the optimal-trade off curve for both system.

Using the plot as a guide, we chose the following values of μ, which leads to a reasonable balance in losses:

- For the GSU subsystem:

$$\mu = 0.717251449792546$$

- For the BLUE substemt:

$$\mu = 0.9243436791194677$$

Midterm

July 20, 2019

1 Midterm

author: Luis Perez
email: luis0@stanford.edu

2 Imports

[12]:
```
import numpy as np
from scipy import linalg
import seaborn as sns
import math
```

2.1 Problem 1: Optimal correction of facial features used by face recognition algorithms

2.1.1 Part (b)

[6]:
```
"""Loading face_features_data.m"""
# number of features
n = 20
# number of example faces (and feature matrices)
K = 5
# example faces
F1 = np.array([
[0.5,-0.5,0,-0.05,0.05,-0.7,-0.6,-0.5,-0.4,-0.3,-0.2,-0.1,0,0.1,0.2,0.3,0.4,0.
 ↪5,0.6,0.7],
[1.0,1.0,0.1,-0.05,-0.05,-0.8,-0.85,-0.9,-0.9,-0.9,-0.9,-0.9,-0.9,-0.9,-0.9,-0.
 ↪9,-0.9,-0.9,-0.85,-0.8]
])
F2 = np.array([
[0.55,-0.55,0,-0.05,0.05,-0.7,-0.6,-0.5,-0.4,-0.3,-0.2,-0.1,0,0.1,0.2,0.3,0.4,0.
 ↪5,0.6,0.7],
[1.0,1.0,0.2,-0.01,-0.01,-0.8,-0.85,-0.9,-0.9,-0.9,-0.9,-0.89,-0.88,-0.89,-0.
 ↪9,-0.9,-0.9,-0.9,-0.85,-0.8]
])
F3 =np.array([
```

```
[0.5,-0.5,0,-0.05,0.05,-0.7,-0.6,-0.5,-0.4,-0.3,-0.2,-0.1,0,0.1,0.2,0.3,0.4,0.
 ↪5,0.6,0.7],
[1.05,1.03,0.08,-0.05,-0.05,-0.8,-0.85,-0.86,-0.87,-0.88,-0.89,-0.9,-0.91,-0.
 ↪9,-0.89,-0.88,-0.87,-0.86,-0.85,-0.8]
])
F4 = np.array([
[0.6,-0.6,0,-0.05,0.05,-0.7,-0.6,-0.5,-0.4,-0.3,-0.2,-0.1,0,0.1,0.2,0.3,0.4,0.
 ↪5,0.6,0.7],
[.9,.9,0.075,-0.05,-0.05,-0.8,-0.85,-0.875,-0.89,-0.9,-0.9,-0.9,-0.9,-0.9,-0.
 ↪9,-0.9,-0.89,-0.875,-0.85,-0.8]
])
F5 = np.array([
[0.56,-0.56,0,-0.05,0.05,-0.7,-0.6,-0.5,-0.4,-0.3,-0.2,-0.1,0,0.1,0.2,0.3,0.4,0.
 ↪5,0.6,0.7],
[1.1,1.1,0.1,-0.05,-0.05,-0.85,-0.875,-0.9,-0.9,-0.9,-0.9,-0.9,-0.9,-0.9,-0.
 ↪9,-0.9,-0.9,-0.9,-0.875,-0.85]
])
```

[7]:
```
# rotated and scaled measurement
Y_rot = np.array([
[-0.159955655589731,-1.33134307892299,-0.149129873451272,-0.0457883892062209,0.
 ↪0607013765513481,-0.148908866497895,-0.00513663237750780,0.138635601742879,0.
 ↪245125367500448,0.351615133258017,0.458104899015586,0.557138171100591,0.
 ↪656171443185597,0.770117702615729,0.884063962045862,0.990553727803431,1.
 ↪09704349356100,1.20353325931857,1.27274055671332,1.34194785410807],
[1.47500480956669,0.654790505584691,0.212979531515138,-0.0479314449385749,0.
 ↪0266334917870611,-1.37387268314000,-1.35255262929315,-1.33123257544630,-1.
 ↪25666763872066,-1.18210270199503,-1.10753776526939,-1.02232385196800,-0.
 ↪937109938666607,-0.873193978516728,-0.809278018366849,-0.734713081641213,-0.
 ↪660148144915577,-0.585583208189941,-0.457773388585520,-0.329963568981100]
])
```

[196]:
```
def getRotationMatrix(x, y):
    """Finds the rotation + scaling matrix such that y = Rx"""
    alpha = linalg.norm(y) / linalg.norm(x)
    d = np.dot(y,x) / (linalg.norm(x) * linalg.norm(y))
    theta = np.arccos(d)
    R = alpha * np.array([
        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]
    ])
    if np.allclose(np.dot(R, x), y):
        return R
    theta = 2*math.pi - theta
    R = alpha * np.array([
        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]
```

```
    ])
    if np.allclose(np.dot(R, x), y):
        return R
    raise ValueError("This is not possible")

def findMatchingFace(candidateFaces, observedFace):
    """Returns index in candidateFaces matching the observedFace"""
    results = []
    for i, face in enumerate(candidateFaces):
        R = getRotationMatrix(face[:, 0], observedFace[:,0])
        if np.allclose(np.dot(R, face), observedFace):
            results.append((i, R))
    assert len(results) == 1
    return results[0]
```

[197]:
```
faces = [F1, F2, F3, F4, F5]
index, R = findMatchingFace(faces, Y_rot)
```

[198]:
```
assert np.allclose(np.dot(R, faces[index]), Y_rot)
```

[199]:
```
print("Y_rot corresponds to $F_%s$." % (index + 1))
```

```
Y_rot corresponds to $F_2$.
```

2.1.2 Part (d)

[213]:
```
Y_noisy = np.array([
[0.6643,-1.0615,0.2158,0.1961,0.3398,-0.3526,-0.1577,0.0293,0.1507,0.2684,0.
 ↪4543,0.5414,0.7290,0.8677,1.0180,1.1605,1.2837,1.3910,1.5616,1.6403],
[1.6783,1.0782,0.1971,0.0118,0.0585,-1.4016,-1.4223,-1.4022,-1.3651,-1.3600,-1.
 ↪2885,-1.2392,-1.1590,-1.1113,-1.0985,-1.0010,-0.9753,-0.8606,-0.7619,-0.6922]
])
```

[237]:
```
# Compute weighed centroids of datasets.
def faceComputeMatrices(face, noisy):
    N = face.shape[1]
    xbar = np.mean(face, axis=1)
    ybar = np.mean(noisy, axis=1)
    xbar.shape = (2,1)
    ybar.shape = (2,1)
    xCentered = face - xbar
    yCentered = noisy - ybar
    S = np.dot(xCentered, yCentered.T)
    u, s, v = np.linalg.svd(S)
    R = np.dot(v, u.T)
    # Verify it is rotation.
    assert np.allclose(np.linalg.det(R), 1)
    den = 0.0
    num = 0.0
```

```
    for i in range(N):
        num += np.dot(np.dot(yCentered[:, i].T, R), xCentered[:, i])
        den += np.dot(xCentered[:, i].T, xCentered[:, i])
    alpha = num / den
    t = ybar - alpha * np.dot(R, xbar)
    return alpha * R, t
```

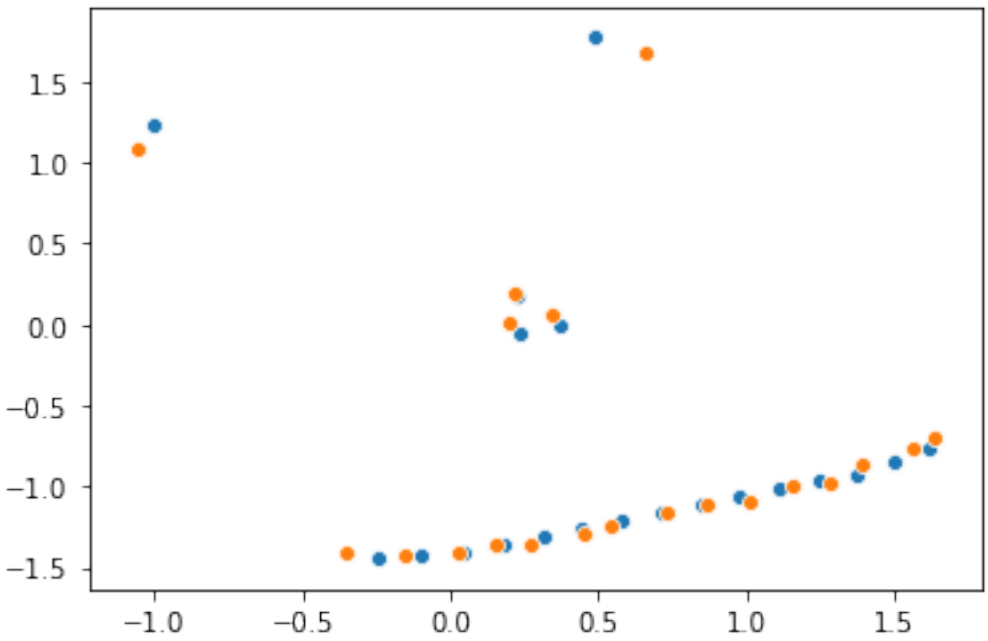
[243]:
```
for i, face in enumerate(faces):
    R, t = faceComputeMatrices(face, Y_noisy)
    print("Scale rotation + translation estimates for $F_%s$" % (i+1))
    print(R)
    print(t)
    # Compute the RMSE error.
    _, N = Y_noisy.shape
    error = 0.0
    for j in range(N):
        error += (1/N) * linalg.norm(Y_noisy[:, j] - np.dot(R, face[:, j] - t))
    print("RMSE for $F_%s$ is %s" % (i + 1, np.sqrt(error)))
```

```
Scale rotation + translation estimates for $F_1$
[[ 1.38710025 -0.50402031]
 [ 0.50402031  1.38710025]]
[[0.26475362]
 [0.07103614]]
RMSE for $F_1$ is 1.215783026538849
Scale rotation + translation estimates for $F_2$
[[ 1.36865856 -0.49728905]
 [ 0.49728905  1.36865856]]
[[0.27399331]
 [0.04565355]]
RMSE for $F_2$ is 1.202885795582196
Scale rotation + translation estimates for $F_3$
[[ 1.38049264 -0.49966651]
 [ 0.49966651  1.38049264]]
[[0.27343758]
 [0.05007972]]
RMSE for $F_3$ is 1.2080822140116791
Scale rotation + translation estimates for $F_4$
[[ 1.42094548 -0.5162301 ]
 [ 0.5162301   1.42094548]]
[[0.25391536]
 [0.10100179]]
RMSE for $F_4$ is 1.2321052401027686
Scale rotation + translation estimates for $F_5$
[[ 1.33411965 -0.48425504]
 [ 0.48425504  1.33411965]]
[[0.27703282]
 [0.03803171]]
```

```
RMSE for $F_5$ is 1.1978958153633974
```

[247]:
```
# The closest face is F5. So let's plot that.
R = np.array([[ 1.33411965, -0.48425504],
 [ 0.48425504,  1.33411965]])
t = np.array([[0.27703282],
 [0.03803171]])
translated = np.dot(R, F5) + t
```

[260]:
```
sns.scatterplot(x=translated[0, :], y=translated[1, :])
ax = sns.scatterplot(x=Y_noisy[0, :], y=Y_noisy[1, :])
ax.get_figure().savefig("test")
```



[211]:
```
np.mean(F1, axis=1)
```

[211]:
```
array([-1.66533454e-17, -5.60000000e-01])
```

[209]:
```
# Find
F1Concat = np.concatenate([F1, np.ones((1,20))], axis=0)
np.dot(np.dot(Y_noisy, F1Concat.T), np.linalg.inv(np.dot(F1Concat, F1Concat.T)))
```

[209]:
```
array([[ 1.46558094, -0.48848387,  0.27345403],
       [ 0.53870348,  1.35194456,  0.05134896]])
```

2.2 Problem 2: Fitting the power consumption of a system

2.2.1 Part (d)

[50]:

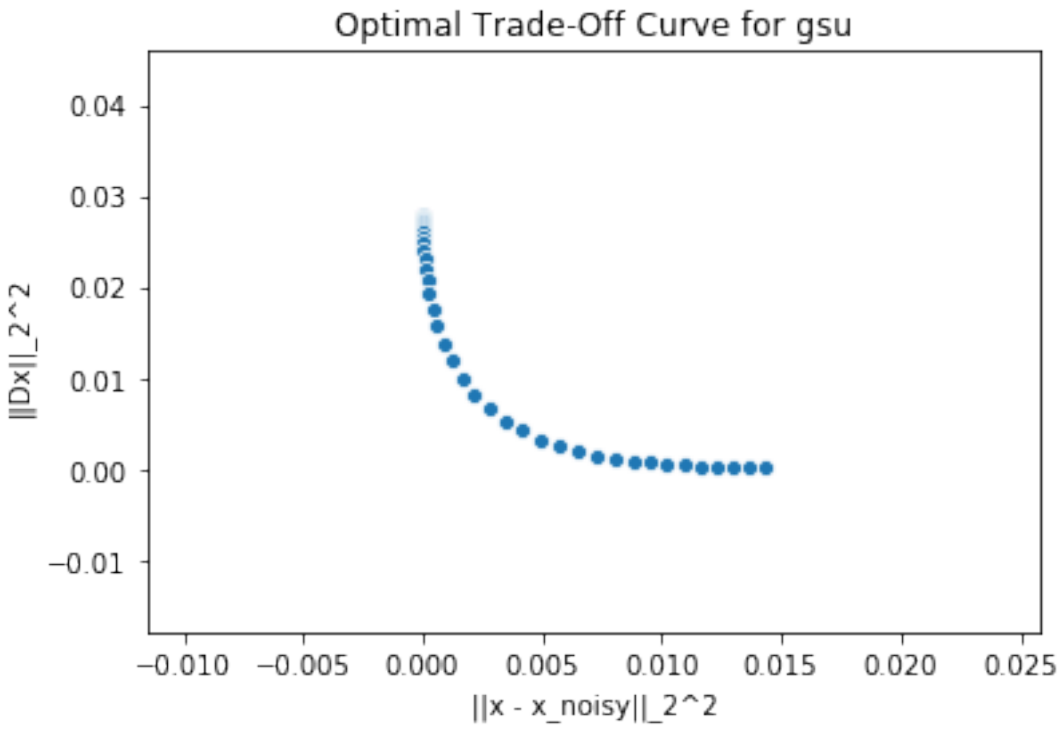
```
n  = 1000
```

[184]:
```
def functorForOptimal(n):
    I = np.identity(n)
    D = -1*np.eye(n-1, n) + np.eye(n-1, n, 1)
    DD = np.dot(D.T, D)
    def findOptimalxHat(mu, noisy):
        assert noisy.shape == (n,1)
        return np.dot(np.linalg.inv(I + mu * DD), noisy)
    return findOptimalxHat, D
```

[185]:
```
def sweepParamAndPlot(noise, name, saveFig=False):
    x = []
    y = []
    losses = []
    J1s = []
    J2s = []
    mus = np.geomspace(1e-4, 25, num=50)
    for mu in mus:
        estimator, D = functorForOptimal(n)
        estimate = estimator(mu, noise)
        J1 = linalg.norm(estimate - noise)**2
        J2 = linalg.norm(np.dot(D, estimate))**2
        J1s.append(J1)
        J2s.append(J2)
        x.append(J1)
        y.append(J2)
    ax = sns.scatterplot(x=x, y=y)
    ax.set_title("Optimal Trade-Off Curve for %s" % name)
    ax.set(xlabel="||x - x_noisy||_2^2", ylabel="||Dx||_2^2")
    if saveFig:
        ax.get_figure().savefig("%s_optimal_trade_off" % name,␣
↪bbox_inches='tight')

    return mus, J1s, J2s
```

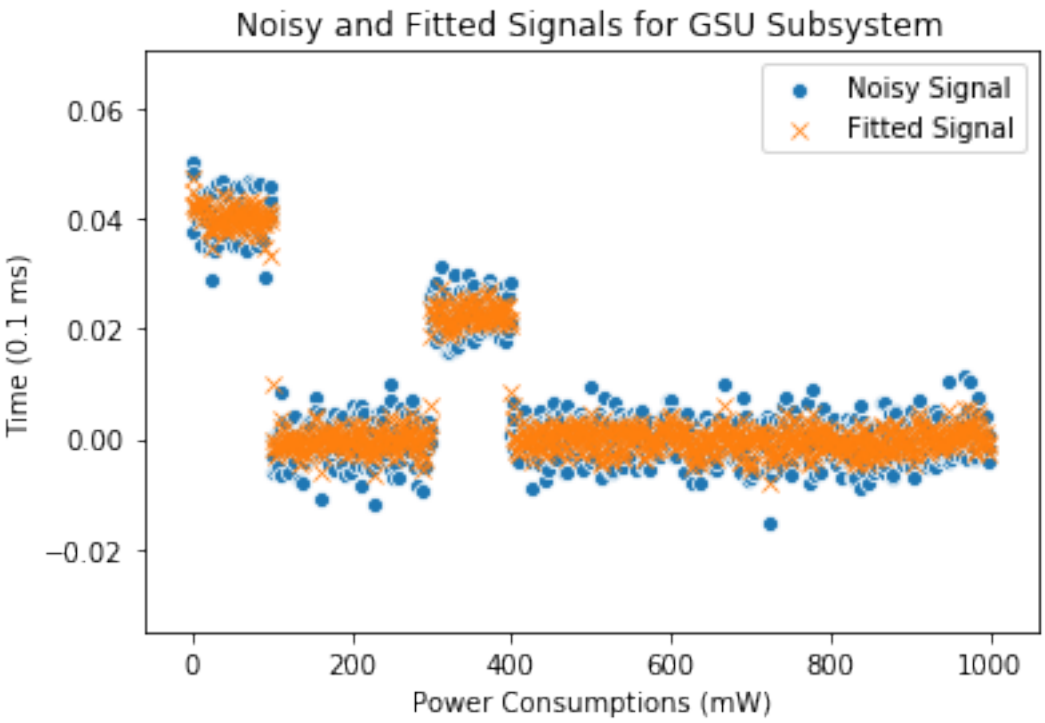
[186]:
```
mus, J1, J2 = sweepParamAndPlot(x_gsu_noisy, "gsu", saveFig=True)
```



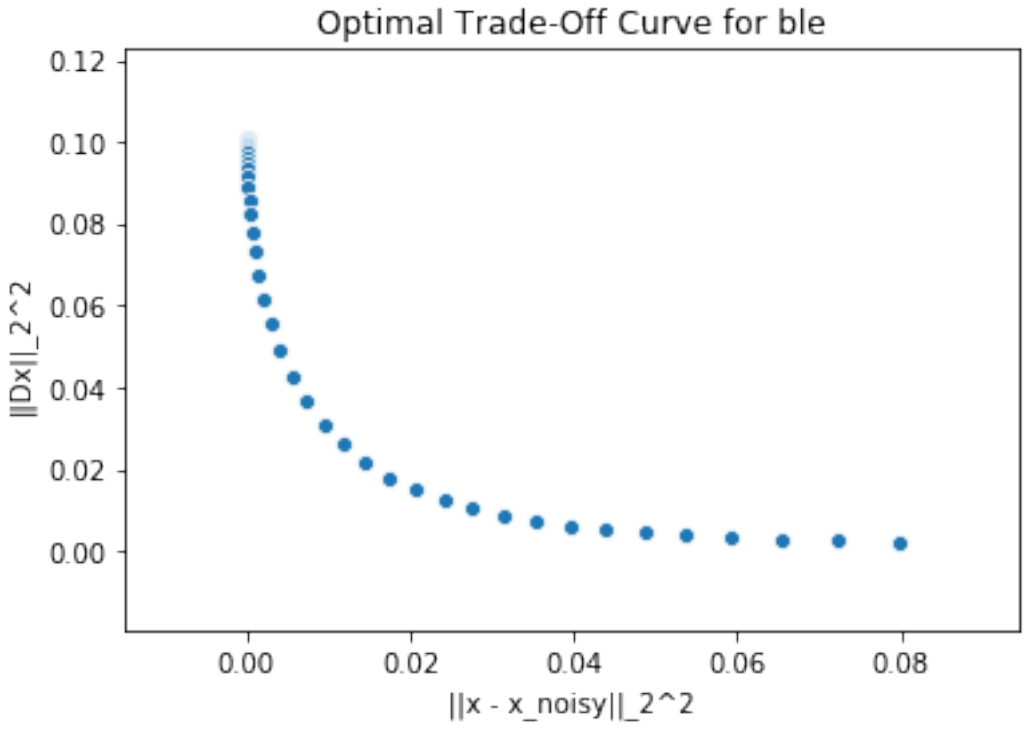
[187]:
```
# A good candidate for mu is around the point (0.004, 0.004).
# which occurs when mu = 0.717251449792546
INDEX = 35
J1[35], J2[35], mus[35]
```

[187]:
```
(0.004151260483021891, 0.004218590509784454, 0.717251449792546)
```

[188]:
```
# Using the selected mu, plot both signal and estimate.
x_gsu_estimate = functorForOptimal(n)[0](mus[35], x_gsu_noisy)
ax = sns.scatterplot(x=range(len(x_gsu_noisy)), y=x_gsu_noisy.flatten(),␣
 ↪marker='o', label="Noisy Signal")
ax = sns.scatterplot(x=range(len(x_gsu_estimate)), y=x_gsu_estimate.flatten(),␣
 ↪marker='x', label="Fitted Signal")
ax.set_title("Noisy and Fitted Signals for GSU Subsystem")
ax.set(xlabel="Power Consumptions (mW)", ylabel="Time (0.1 ms)")
ax.get_figure().savefig("gsu_subsystem_plot")
```



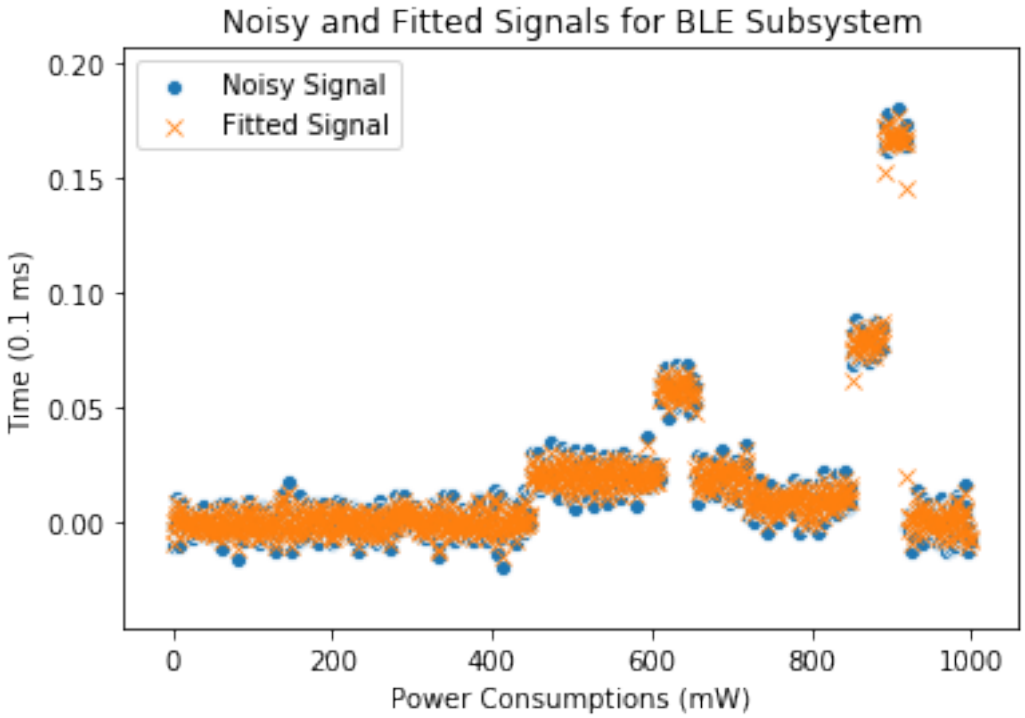
[189]:
```
mus, J1s, J2s = sweepParamAndPlot(x_ble_noisy, "ble", saveFig=True)
```



[194]:
```
# A good candidate for mu is around the point (0.004, 0.004).
# which occurs when 0.9243436791194677
INDEX = 36
J1s[INDEX], J2s[INDEX], mus[INDEX]
```

[194]:
```
(0.017538795477413233, 0.01803654944679504, 0.9243436791194677)
```

[195]:
```
# Using the selected mu, plot both signal and estimate.
x_ble_estimate = functorForOptimal(n)[0](mus[30], x_ble_noisy)
ax = sns.scatterplot(x=range(len(x_ble_noisy)), y=x_ble_noisy.flatten(),␣
 ↪marker='o', label="Noisy Signal")
ax = sns.scatterplot(x=range(len(x_ble_estimate)), y=x_ble_estimate.flatten(),␣
 ↪marker='x', label="Fitted Signal")
ax.set_title("Noisy and Fitted Signals for BLE Subsystem")
ax.set(xlabel="Power Consumptions (mW)", ylabel="Time (0.1 ms)")
ax.get_figure().savefig("ble_subsystem_plot")
```



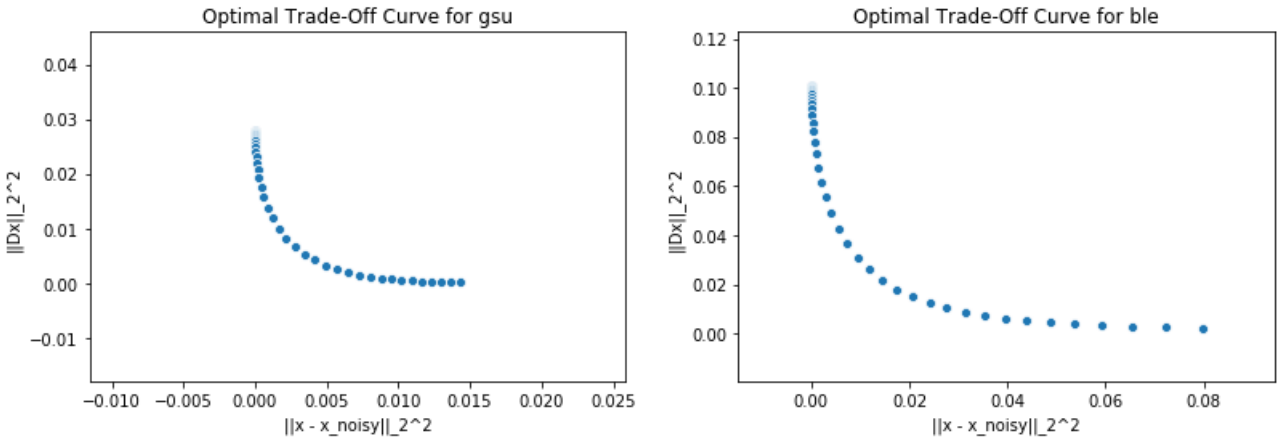


Figure 1: Plot of optimal trade-off curve for each of our subsytems. The plot are both generated using a geometric (logspace) sweep over $\mu \in [1e-4, 25)$.

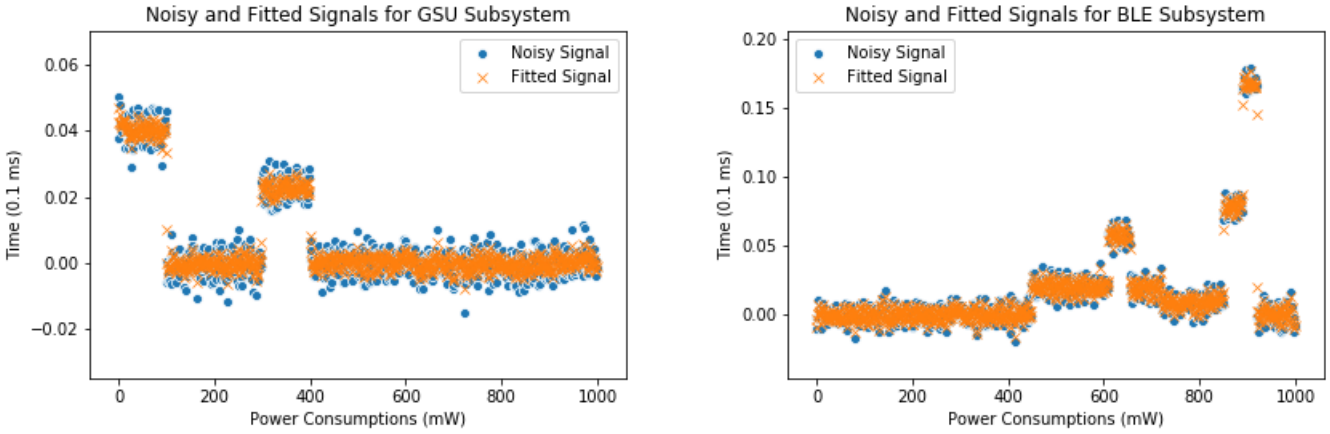


Figure 2: Plot of the received signal as well as the fitted signal for GSU and BLE subsystems..