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
```

```python
    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
```

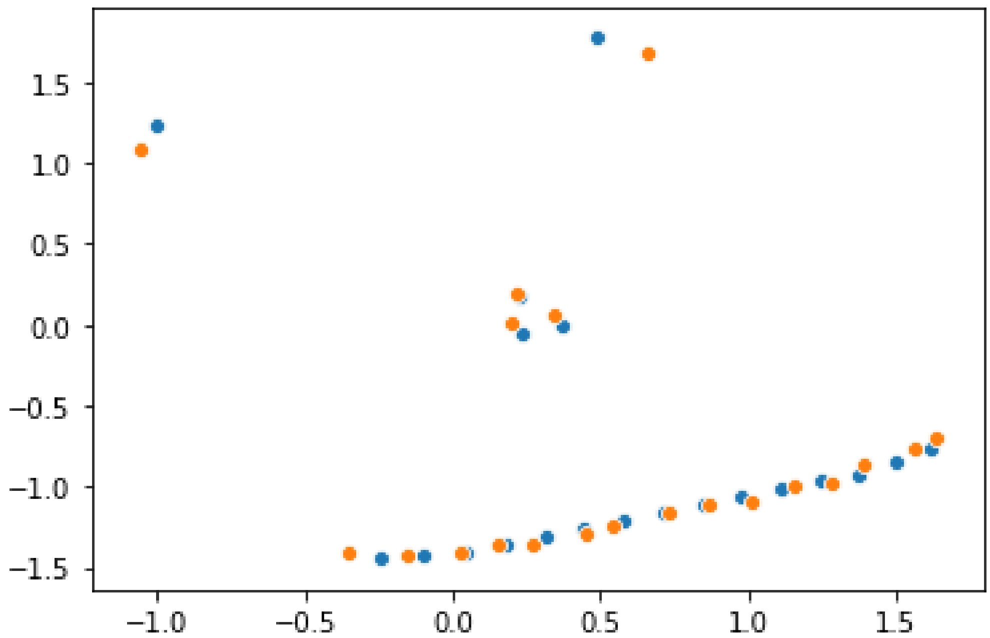
[243]:
```python
for i, face in enumerate(faces):
    R, t = faceComputeMatrices(face, Y_noisy)
    print("Scale rotation + translation estimates for $F_%s$" % (i+1))
    print(R)
    print(t)
    # Compute the RMSE error.
    _, N = Y_noisy.shape
    error = 0.0
    for j in range(N):
        error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

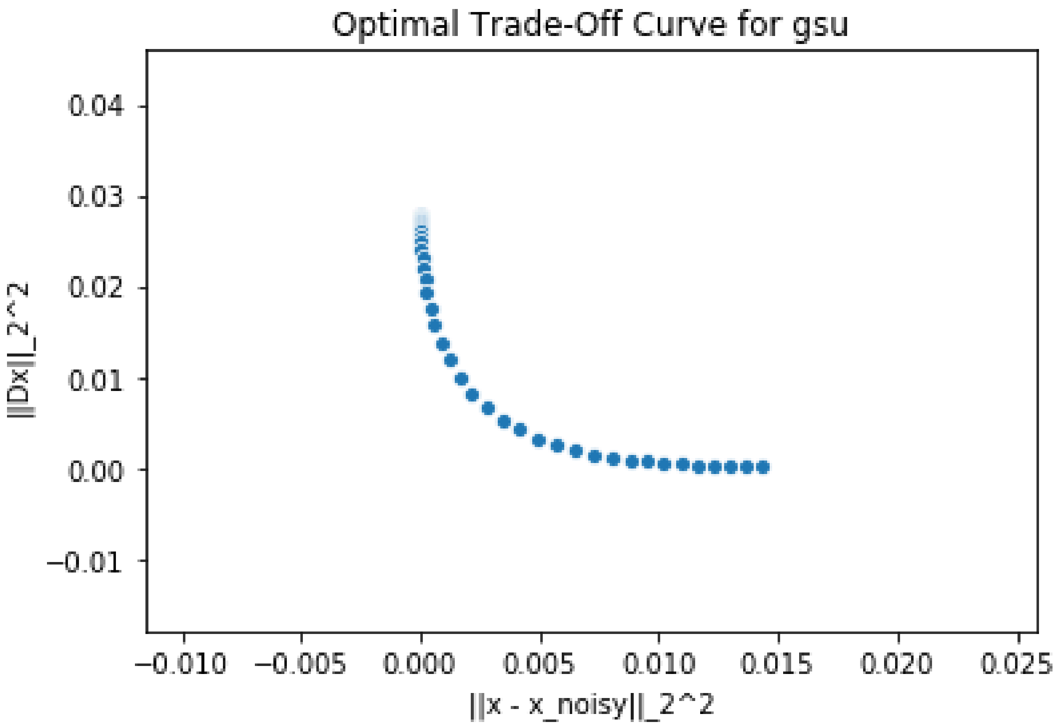
```
n  = 1000
```

[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
```

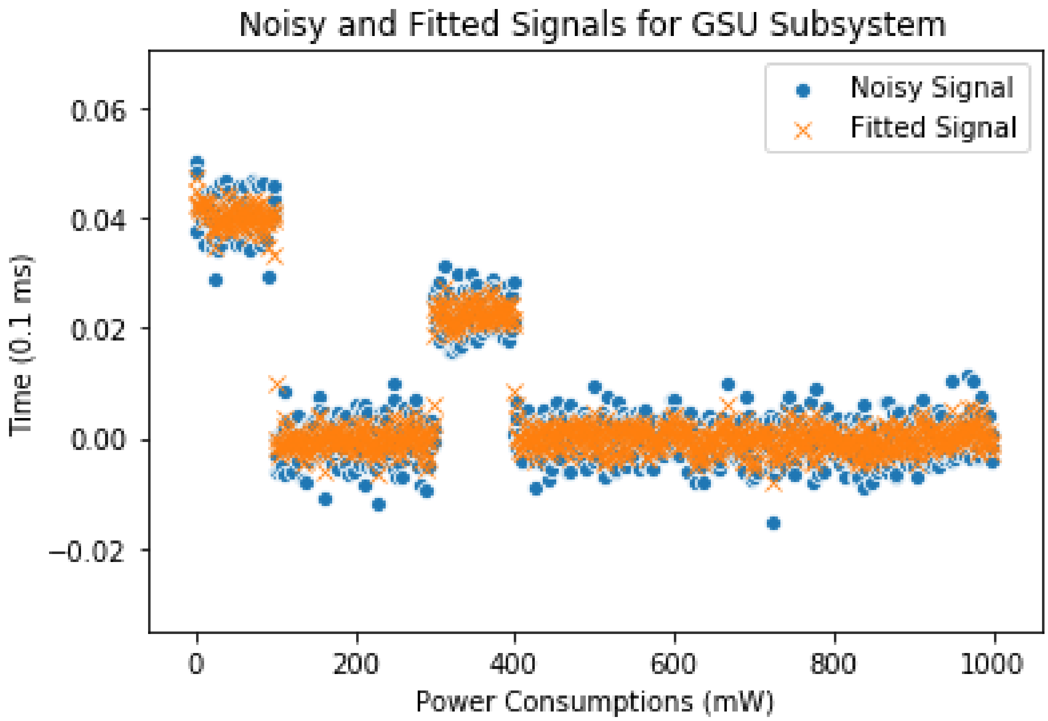
[186]:
```
mus, J1, J2 = sweepParamAndPlot(x_gsu_noisy, "gsu", saveFig=True)
```



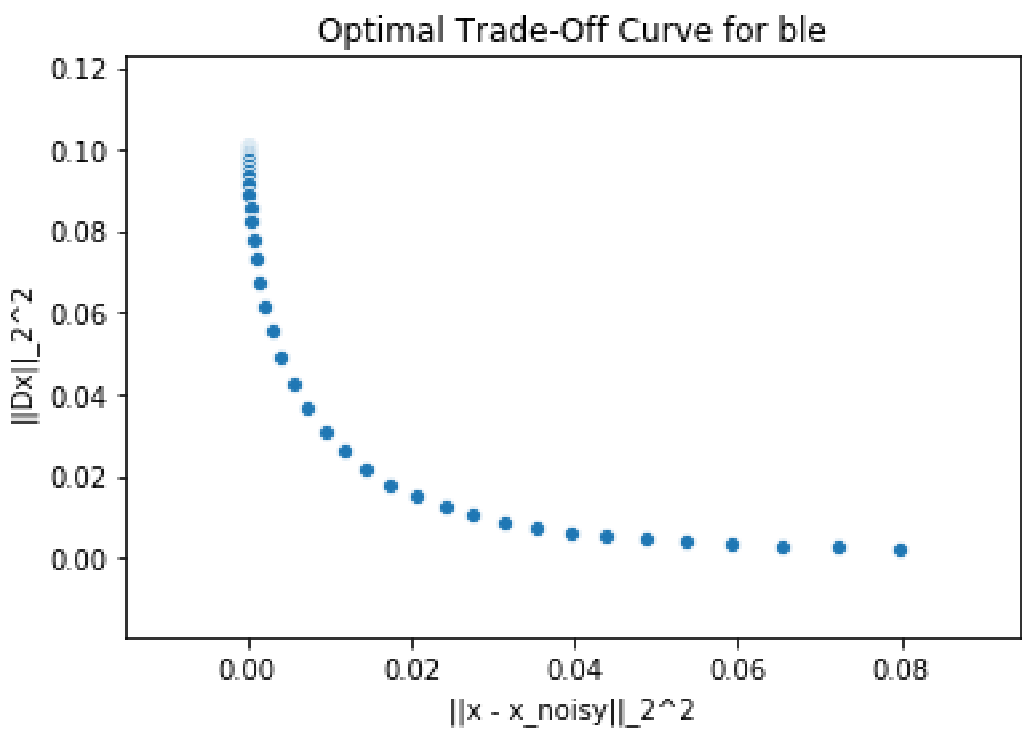
[187]:
```
# A good candidate for mu is around the point (0.004, 0.004).
# which occurs when mu = 0.717251449792546
INDEX = 35
J1[35], J2[35], mus[35]
```

[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")
```