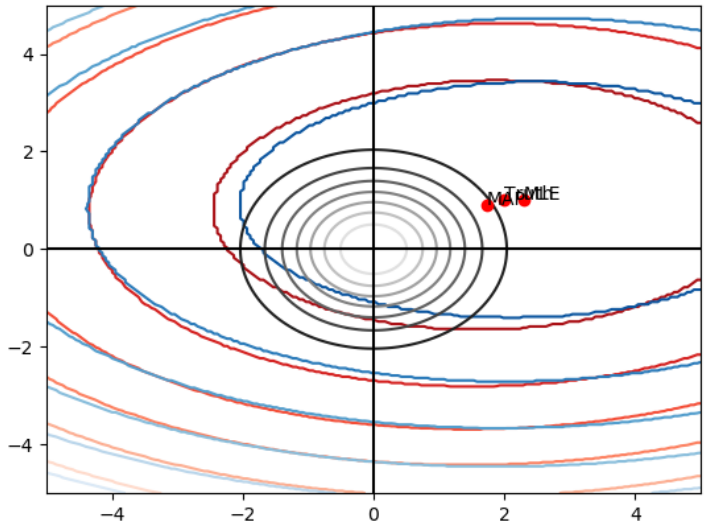
# Appendix

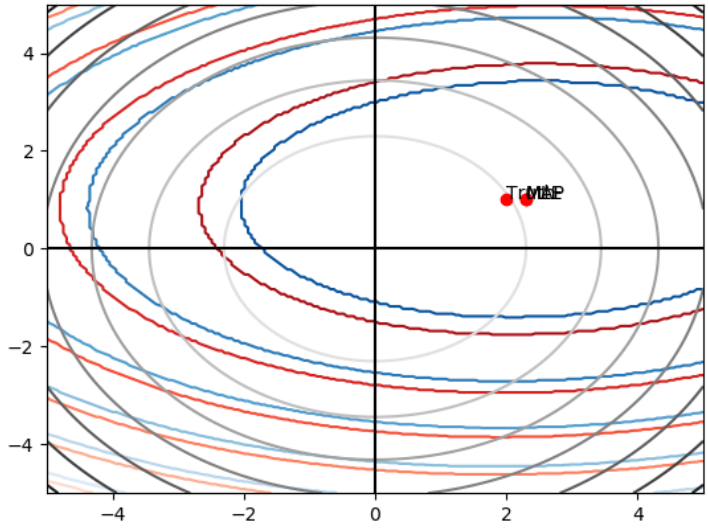
1.

(b)

$\sigma_h = 1$

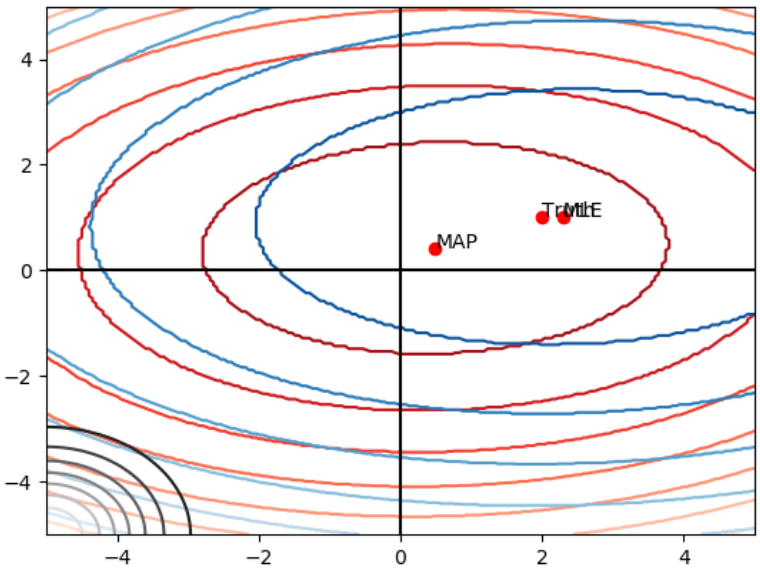



$\sigma_h = 10$

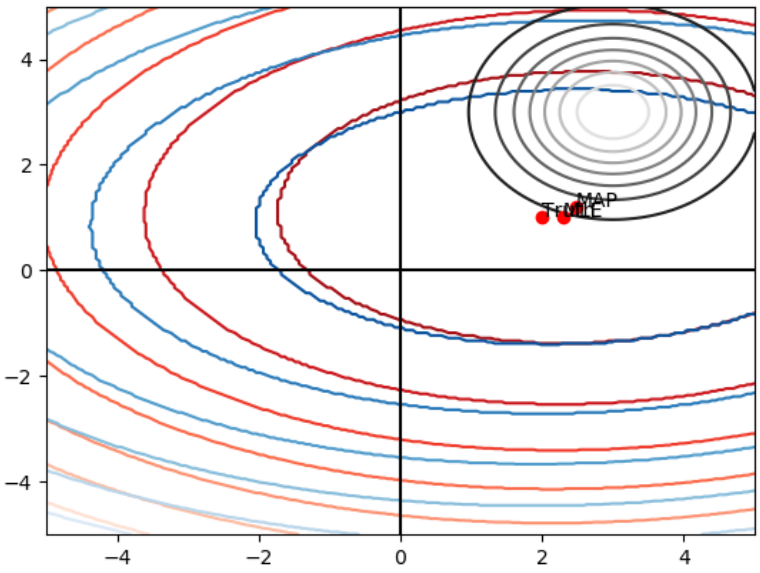


(c)

$\mu_\theta = \begin{matrix} -5 \\ -5 \end{matrix}$:

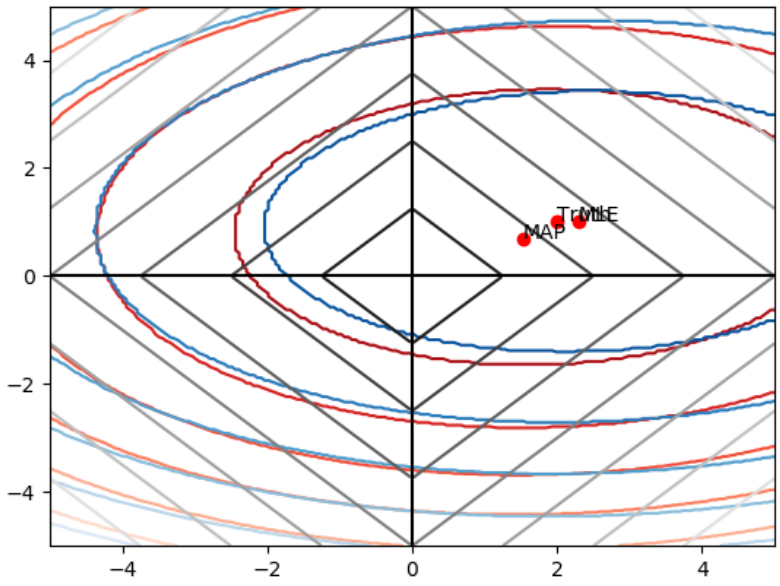



$\mu_\theta = \begin{matrix} 3 \\ 3 \end{matrix}$

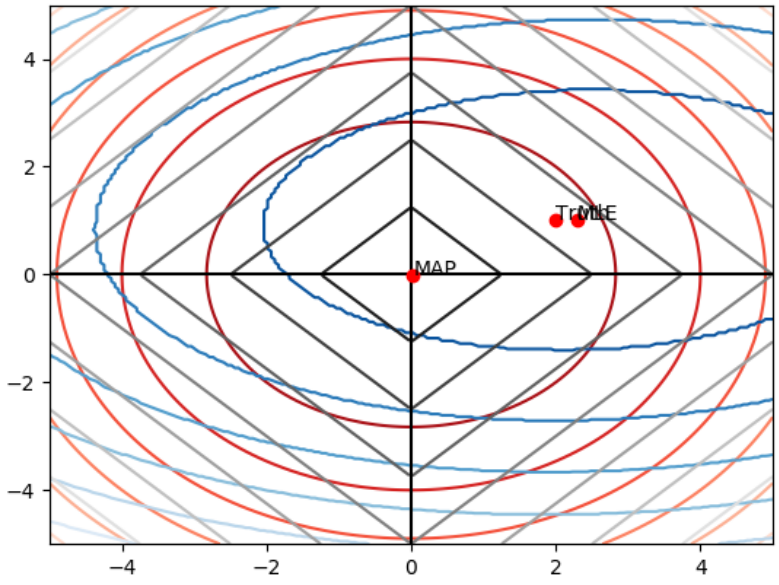


(e)

$P(\theta_i) \sim L(0, 1)$



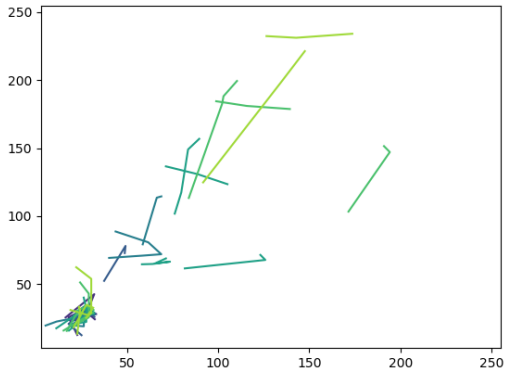

$P(\theta_i) \sim L(0, 0.0001)$



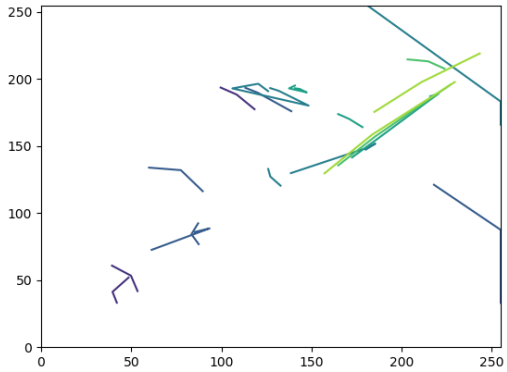
4.

(a)

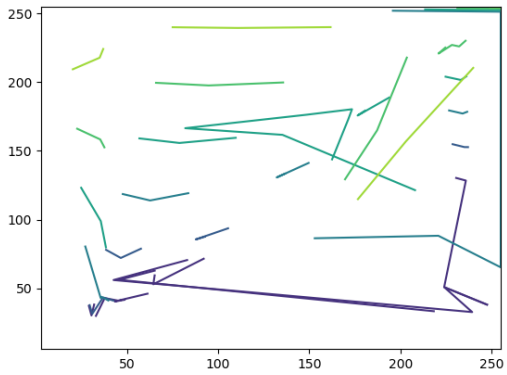
$0^{th}$ image




$10^{th}$ image




$20^{th}$ image




Their corresponding control vectors are:

```
0th image control vector: [ 0. -1.  0.]
10th image control vector: [-1.         -0.45111084 -1.        ]
20th image control vector: [0.         0.         0.37368774]
```

(c)

```
Average squared Euclidien distance for different lambda on training set: [1.256702068937529e-15,
1.25669241758656e-13, 1.2565943984111686e-11, 1.2556154166406199e-09, 1.2459339631725425e-07]
```

(d)

```
Average squared Euclidien distance for different lambda on training set after scale: [3.255747498
915746e-07, 2.910512290768579e-05, 0.0015903814573038663, 0.034773122042375766, 0.254402961467970
3]
```

(e)

```
Average squared Euclidien distance for different lambda on test set: [2.0792664172815383e-16, 2.0
7926090800701e-14, 2.0792067442486828e-12, 2.0786652769260328e-10, 2.073269975904201e-08]
Average squared Euclidien distance for different lambda on test set after scale: [5.4858500599334
95e-08, 5.2638881496879074e-06, 0.0003806734444981973, 0.011336004792241953, 0.13159252242709102]
```

(f)

```
Condition number of training data without standardization: 52711693.12866252
Condition number of training data with standardization: 444.7259317110044
Condition number of test data without standardization: 28927142.279349737
Condition number of test data with standardization: 39339.65804431199
```

```
import numpy as np
import matplotlib.pyplot as plt

'''
Hyperparameter Configurations
'''
# Prior weights are assumed to be i.i.d, so they all have the same prior_std
prior_mean = [0, 0]
prior_std = 0.0001
noise_mean = 0.0
noise_std = 1.0
# Can either by Gaussian or Laplacian Prior
options = ["Gaussian", "Laplacian"]
option = options[1]

# Plot and Randomness Configurations
fig, ax = plt.subplots()
ax.axhline(y=0, color='k')
ax.axvline(x=0, color='k')
lower_range = -5
upper_range = 5
# DO NOT Change the Seed and Num Points for Easy and Consistent Grading
np.random.seed(7)
num_data_points = 10


'''
Point Estimate Functions
'''
def MLE_Point_Est(X, Y):
    '''
    Input: X - nx2; Y - nx1
    Return: w - 2x1
    Returns the OLS Solution
    '''
    return np.linalg.solve(X.T@X, X.T@Y)

def Gaussian_MAP_Point_Est(X, Y, noise_std, prior_std, prior_mean):
    '''
    Input: X - nx2; Y - nx1, Prior and Noise Staistics (All scalars)
    Return: w - 2x1
    Returns the Ridge Regression Solution
    '''
    lambda_ = (noise_std/prior_std)**2
    return np.linalg.solve(X.T@X+lambda_*np.identity(X.shape[1]),
        X.T@Y+lambda_*np.array(prior_mean).reshape((2,1)))

def LASSO_Point_Est(w1, w2, map_contour):
    '''
    This part of the code is filled out. Since in most cases there is no
     closed form for the LASSO equation, we directly find the center
     through a linear search.
    DON'T DO THIS.
    '''
    min_candidates = np.where(map_contour == np.amin(map_contour))
```

```
    processed_points = list(zip(min_candidates[0], min_candidates[1]))

    x_ind = processed_points[0][0]
    y_ind = processed_points[0][1]
    return np.array([w1[x_ind][y_ind], w2[x_ind][y_ind]]).reshape((-1,1))

'''
Functions for Contour Plotting
'''

def Gaussian_Prior(w1, w2, prior_mean):
    '''
    Returns the Gaussian Prior or L2 norm squared for weights
    '''
    return
     (1/(2*np.pi*(prior_std**2)))*np.exp(-0.5*((((w1-prior_mean
     [0])/prior_std)**2)+(((w2-prior_mean[1])/prior_std)**2)))

def Laplacian_Prior(w1, w2, prior_mean):
    '''
    Returns the Laplacian Prior for weights
    '''
    return
     ((np.abs(w1-prior_mean[0])+np.abs(w2-prior_mean
     [1]))/prior_std)*(2*(noise_std**2))

def Gaussian_MLE_Contour(w1,w2, X, Y):
    '''
    Returns the MLE estimate value
    '''
    x = np.array([0]*200)
    y = np.array([0]*200)
    Z, z = np.meshgrid(x,y)
    for i in range(200):
        for j in range(200):
            Z[i][j] = np.sum(((Y-X@(np.array([w1[i][j],
                w2[i][j]]).reshape((2,1))))**2)/(2*(noise_std**2))) +
                 num_data_points*np.log(np.sqrt(2*np.pi)*noise_std)
    return Z

def MAP_Contour(w1, w2, X, Y, noise_std, prior_mean, prior_std,
 option="Gaussian"):
    '''
        Hint 1: Use the methods above to compute the MAP. Ideally one line
         of code.
        Hint 2: Use two for loops to go through the (x,y) coordinates for
         w1, w2
    '''
    x = np.array([0]*200)
    y = np.array([0]*200)
    Z, z = np.meshgrid(x,y)
    for i in range(200):
        for j in range(200):
            if option == "Laplacian":
```

```
                Z[i][j] = np.linalg.norm(Y-X@(np.array([w1[i][j],
                 w2[i]
                 [j]])
                 .reshape((2,1))))**2+(2*(noise_std**2)/prior_std)*((np
                 .abs(w1[i][j]-prior_mean[0])+np.abs(w2[i][j]-prior_mean
                 [1])))
            Z[i][j] = np.sum(((Y-X@(np.array([w1[i][j],
             w2[i][j]]).reshape((2,1))))**2)/(2*(noise_std**2))) +
             (np.array([w1[i][j],
                w2[i][j]]) - np.array(prior_mean))@((np.array([w1[i][j],
                 w2[i][j]]) -
                 np.array(prior_mean)).reshape(2,1))/(2*(prior_std**2))
    return Z

'''
Plotting Functions
'''
def plot_point(w, txt):
    plt.plot(w[0][0], w[1][0], 'ro')
    ax.annotate(txt, (w[0][0], w[1][0]))

# Generating Data and Labels with Random Gaussian Noise
w = np.array([[2.0], [1.0]])
Xf= np.random.uniform(-1, 1, num_data_points)
X1 = np.array([1.0]*num_data_points)
X = np.array([Xf, X1]).T
Y = X.dot(w).reshape((-1, 1)) + np.random.normal(noise_mean, noise_std,
 num_data_points).reshape((-1,1))

w1 = np.linspace(lower_range, upper_range, 200)
w2= np.linspace(lower_range, upper_range, 200)
w1, w2 = np.meshgrid(w1, w2)

prior = Gaussian_Prior(w1, w2, prior_mean) if option=="Gaussian" else
 Laplacian_Prior(w1,w2, prior_mean)
mle_contour = Gaussian_MLE_Contour(w1,w2, X, Y)
map_contour = MAP_Contour(w1, w2, X, Y, noise_std, prior_mean, prior_std,
 option)

w_mle = MLE_Point_Est(X,Y)
w_map = Gaussian_MAP_Point_Est(X,Y, noise_std, prior_std, prior_mean) if
 option=="Gaussian" else LASSO_Point_Est(w1, w2, map_contour)

plot_point(w, "Truth")
plot_point(w_mle, "MLE")
plot_point(w_map, "MAP")

# Choose which contours to plot [prior, mle, or map]
plt.contour(w1, w2, map_contour, 7, cmap='Reds_r')
plt.contour(w1, w2, mle_contour, 7, cmap='Blues_r')
plt.contour(w1, w2, prior, 7, cmap='gray')

plt.show()
```

```
import pickle
import matplotlib.pyplot as plt
import numpy as np

class HW3_Sol(object):

    def __init__(self):
        pass

    def load_data(self):
        self.x_train = pickle.load(open('x_train.p','rb'),
         encoding='latin1')
        self.y_train = pickle.load(open('y_train.p','rb'),
         encoding='latin1')
        self.x_test = pickle.load(open('x_test.p','rb'), encoding='latin1')
        self.y_test = pickle.load(open('y_test.p','rb'), encoding='latin1')

if __name__ == '__main__':

    hw3_sol = HW3_Sol()

    hw3_sol.load_data()

    # Your solution goes here
    #4(a)
    hw3_sol.x_train.astype(float)
    hw3_sol.y_train.astype(float)
    hw3_sol.x_test.astype(float)
    hw3_sol.y_test.astype(float)
    #Only choose one to plot each time
    #plot 0th image
    plt.contour(hw3_sol.x_train[0][0], hw3_sol.x_train[0][1],
     hw3_sol.x_train[0][2])
    #plot 10th image
    plt.contour(hw3_sol.x_train[10][0], hw3_sol.x_train[10][1],
     hw3_sol.x_train[10][2])
    #plot 20th image
    plt.contour(hw3_sol.x_train[20][0], hw3_sol.x_train[20][1],
     hw3_sol.x_train[20][2])
    print('0th image control vector:', hw3_sol.y_train[0])
    print('10th image control vector:', hw3_sol.y_train[10])
    print('20th image control vector:', hw3_sol.y_train[20])
    plt.show()

    #4(b)
    n = hw3_sol.x_train.shape[0]
    X = np.zeros((n,2700))
    for i in range(n):
        X[i] = hw3_sol.x_train[i].flatten()
    U = hw3_sol.y_train.reshape((n, 3))
    #PI = np.linalg.solve(X.T@X, X.T@U)

    #4(c)
    LAMBDA = [0.1, 1.0, 10.0, 100.0, 1000.0]
    errors = []
```

```
for i in range(5):
    PI = np.linalg.solve(X.T@X+LAMBDA[i]*np.identity(X.shape[1]), X.T@U)
    residual = X@PI - U
    error = np.zeros((n,1))
    for j in range(n):
        error[j] = residual[j][0]**2 + residual[j][1]**2 +
         residual[j][2]**2
    error = np.mean(error)
    errors += [error]
print('Average squared Euclidien distance for different lambda on
 training set:', errors)

#4(d)
scaled_errors = []
scaled_X = X*2/255-1
for i in range(5):
    PI_scaled =
     np.linalg.solve(scaled_X.T@scaled_X+LAMBDA
     [i]*np.identity(scaled_X.shape[1]), scaled_X.T@U)
    residual_scaled = scaled_X@PI_scaled - U
    scaled_error = np.zeros((n,1))
    for j in range(n):
        scaled_error[j] = residual_scaled[j][0]**2 +
         residual_scaled[j][1]**2 + residual_scaled[j][2]**2
    scaled_error = np.mean(scaled_error)
    scaled_errors += [scaled_error]
print('Average squared Euclidien distance for different lambda on
 training set after scale:', scaled_errors)

#4(e)
n_test = hw3_sol.x_test.shape[0]
X_test = np.zeros((n_test,2700))
for i in range(n_test):
    X_test[i] = hw3_sol.x_test[i].flatten()
U_test = hw3_sol.y_test.reshape((n_test, 3))
X_test_scaled = X_test*2/255-1
errors_test = []
scaled_errors_test = []
for i in range(5):
    PI =
     np.linalg.solve(X_test.T@X_test+LAMBDA[i]*np.identity(X_test.shape
     [1]), X_test.T@U_test)
    PI_scaled =
     np.linalg.solve(X_test_scaled.T@X_test_scaled+LAMBDA
     [i]*np.identity(X_test_scaled.shape[1]), X_test_scaled.T@U_test)
    residual_test = X_test@PI - U_test
    residual_scaled_test = X_test_scaled@PI_scaled - U_test
    error_test = np.zeros((n_test, 1))
    scaled_error_test = np.zeros((n_test, 1))
    for j in range(n_test):
        error_test[j] = residual_test[j][0]**2 + residual_test[j][1]**2
         + residual_test[j][2]**2
        scaled_error_test[j] = residual_scaled_test[j][0]**2 +
         residual_scaled_test[j][1]**2 + residual_scaled_test[j][2]**2
    error_test = np.mean(error_test)
```

```
    scaled_error_test = np.mean(scaled_error_test)
    errors_test += [error_test]
    scaled_errors_test += [scaled_error_test]
print('Average squared Euclidien distance for different lambda on test
 set:', errors_test)
print('Average squared Euclidien distance for different lambda on test
 set after scale:', scaled_errors_test)

#4(f)
u_train, s_train, v_train = np.linalg.svd(X.T@X +
 100*np.identity(X.shape[1]))
u_train_scaled, s_train_scaled, v_train_scaled =
 np.linalg.svd(scaled_X.T@scaled_X + 100*np.identity(scaled_X.shape[1]))
u_test, s_test, v_test = np.linalg.svd(X_test.T@X_test +
 100*np.identity(X_test.shape[1]))
u_test_scaled, s_test_scaled, v_test_scaled =
 np.linalg.svd(X_test_scaled.T@X_test_scaled +
 np.identity(X_test_scaled.shape[1]))
k_train = np.amax(s_train) / np.amin(s_train_scaled)
k_train_scaled = np.amax(s_train_scaled) / np.amin(s_train_scaled)
k_test = np.amax(s_test) / np.amin(s_test)
k_test_scaled = np.amax(s_test_scaled) / np.amin(s_test_scaled)
print('Condition number of training data without standardization:',
 k_train)
print('Condition number of training data with standardization:',
 k_train_scaled)
print('Condition number of test data without standardization:', k_test)
print('Condition number of test data with standardization:',
 k_test_scaled)
```