Assignment 3

Yiyan Ge

The following is the defined GP class implementation. The choice of covariance function follows the example shown in lecture slides.

```
class GP_RainFall:

    def __init__(self,trn_data,tst_data,h):
        # h is distance between x and x' (bandwidth)
        self.X_trn_raw, self.Y_trn_raw = trn_data[:,:-1], trn_data[:,-1:]
        self.X_tst_raw = tst_data
        self.h = h
        self.K_tsttst = self.covariance(self.ConvertUnit(self.X_tst_raw),self.ConvertUnit(self.X_tst_raw))
        self.K_trntst = self.covariance(self.ConvertUnit(self.X_trn_raw),self.ConvertUnit(self.X_tst_raw))
        self.K_tsttrn = self.covariance(self.ConvertUnit(self.X_tst_raw),self.ConvertUnit(self.X_trn_raw))
        self.K_trntrn = self.covariance(self.ConvertUnit(self.X_trn_raw),self.ConvertUnit(self.X_trn_raw))
        self.I = np.identity(self.K_trntrn.shape[0])


    def ConvertUnit(self,array):
        array_list = []
        for n in array:
            meters = utm.from_latlon(n[0],n[1])
            array_list.append(np.array([meters[0], meters[1]]))
        converted_array = np.vstack(array_list)
        return converted_array

    #Computing Gaussian covariance:
    def covariance(self, X, Z):
        d = spatial.distance_matrix(X,Z)
        K = np.exp(-(d**2) / (2*self.h*self.h))
        return K

    # Make Predictions

    def predict(self, sigma, X_trn_raw, X_tst_raw, Y_trn_pre):
        X_tst = self.ConvertUnit(X_tst_raw)
        X_trn = self.ConvertUnit(X_trn_raw)

        K_tsttrn = self.covariance(X_tst,X_trn)
        K_trntrn = self.covariance(X_trn,X_trn)

        I = np.identity(K_trntrn.shape[0])

        mean = np.mean(Y_trn_pre)
        Y_trn = Y_trn_pre - mean*np.ones(Y_trn_pre.shape)

        pred_mean = np.dot(np.dot(K_tsttrn,inv(K_trntrn + sigma**2*I)),Y_trn)
        pred_Y = pred_mean + mean*np.ones(pred_mean.shape)

        return pred_Y

    def predict_cv(self, k, sigma):
        kf = KFold(len(data), n_folds = k, shuffle=True)
        RMSE = []
        for train_index, test_index in kf:
            X_trn_cv, X_tst_cv = self.X_trn_raw[train_index], self.X_trn_raw[test_index]
            Y_trn_cv, Y_tst_cv = self.Y_trn_raw[train_index], self.Y_trn_raw[test_index]
            Y_pred = self.predict(sigma, X_trn_cv, X_tst_cv, Y_trn_cv)
```

```
            Y_true = Y_tst_cv
            error = sqrt(mean_squared_error(Y_pred, Y_true))
            RMSE.append(error)

        self.RMSE = np.mean(RMSE)
        return self.RMSE

    def simulation(self, sigma):
        m_f = self.predict(sigma, self.X_trn_raw, self.X_tst_raw, self.Y_trn_raw)
        cov = self.K_tsttst - np.dot(np.dot(self.K_tsttrn,
                            inv(self.K_trntrn + sigma**2*self.I)), self.K_trntst)
        L = np.linalg.cholesky(cov + 0.001*np.eye(cov.shape[0])) #gamma~0.001
        u = np.random.normal(0,1,cov.shape[0])
        f_sim = m_f.reshape(-1,) + np.dot(L,u)
        return f_sim, cov
```

Using this, I tested relationship between `RMSE` and `sigma` with fixed `h` as well as the relationship between `RMSE` and `h` with fixed `sigma` using the following code:

```
RMSE1 = []
RMSE2 = []

bandwidth = [x*1000 for x in range(40,90,5)][1:]
sigma = 0.45
k = 5
for h in bandwidth:
    GP = GP_RainFall(trn_data, tst_data, h)
    RMSE1.append(GP.predict_cv(5, sigma))

h = 70000
sigmas = [x for x in frange(0.05, 0.8, 0.05)][1:]
for sigma in sigmas:
    GP = GP_RainFall(trn_data, tst_data, h)
    RMSE2.append(GP.predict_cv(5, sigma))
```

Figure1 is one example of plotted results.

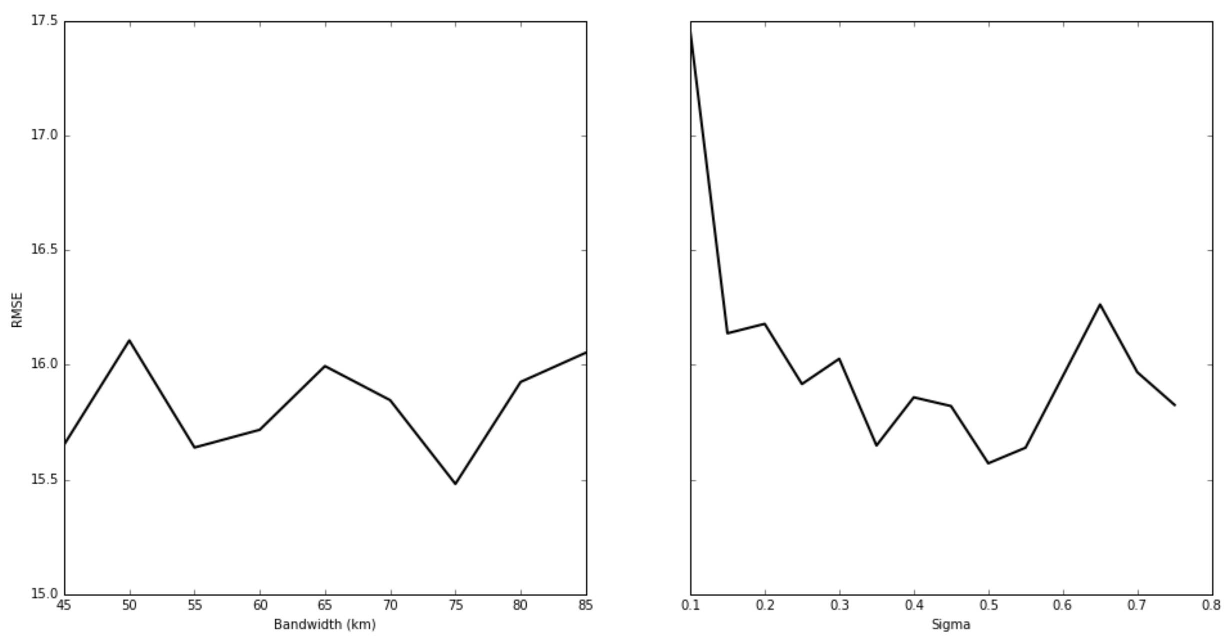

Figure 1: relationship between `RMSE` and `sigma` with fixed `h` & the relationship between `RMSE` and `h` with fixed `sigma`

With many rounds of experiments with different pair of `sigma` and `h`, I chose `h` = 70000 `sigma` = 0.4 as the pair that very likely minimizes `RMSE`.

Figure2 is the overlay on Google Earth.

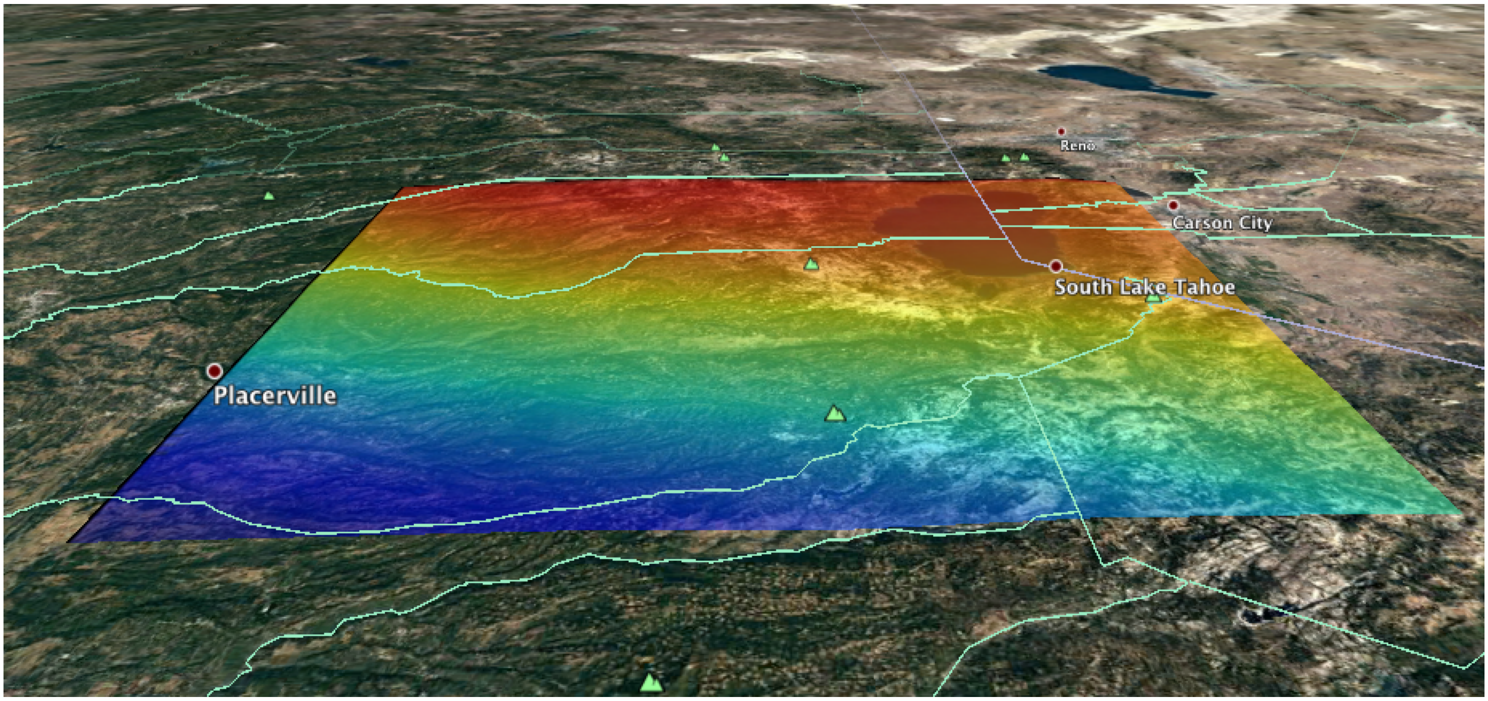

Figure 2: overlay