1　プログラム

プログラムには Python3 を用いた。

Listing 1: system_report.py

```
#!/usr/bin/env Python3
# -*- coding: utf-8 -*-



## modules -----------------------------------------------------------------
import numpy as np
from scipy import optimize
import csv
import matplotlib.pyplot as plt



## functions ---------------------------------------------------------------
def main():
    #q_1_2()
    #q_2_2()
    #q_4_5()
    #q_4_5_2()
    #q_5_1()
    #q_5_3()
    q_5_4()
    #q_6_1()


def q_1_2():
    q = Q_1_2()
    q.calc()
    q.plot()


def q_2_2():
    q = Q_2_2()
    q.mkElist()
    q.plot_sum_eq()
    q.plot_mlt()


def q_4_5():
    q = Q_4_5()
    q.calc()
    q.print_result()
```

```
    q.plot()


def q_4_5_2():
    q = Q_4_5_2()
    q.calc()
    q.print_result()
    q.plot()


def q_5_1():
    np.random.seed(1)                                    # 適当なseedを選ぶseedの固定
    seeds = np.random.randint(0, 100000000, 100)    # 適当なseedを選ぶ
    AICc_sum = np.zeros(4, dtype=float)
    Q_sum = np.zeros(4, dtype=float)
    for seed in seeds:                                   #各seedについて計算
        q = Q_5_1(seed)
        q.calc()
        #q.print_result()
        #q.plot_train()
        #q.plot_test()
        for i in range(4):
            AICc_sum[i] += q.result_list[i][2]
            Q_sum[i] += q.result_list[i][3]
    AICc_mean = AICc_sum / len(seeds)
    Q_mean = Q_sum / len(seeds)
    AICs_min_idx = np.argmin(AICc_mean)
    Q_min_idx = np.argmin(Q_mean)
    for i in range(4):
        dim = i + 1
        print("Dimension : {}".format(dim))
        print("\t AICc ({} times mean) : {}".format(len(seeds), AICc_mean[i]))
        print("\t Q    ({} times mean) : {}".format(len(seeds), Q_mean[i]))
        print()
    print("valid dimension is {} from AICc value".format(AICs_min_idx + 1))
    print("valid dimension is {} from Q value".format(Q_min_idx + 1))


def q_5_3():
    q = Q_5_3()
    q.calc()
    q.print_result()
    q.plot()


```

```
def q_5_4():
    q = Q_5_4()
    q.calc()
    q.plot()


def q_6_1():
    q = Q_6_1()
    q.calc1()
    q.print_result(1)
    q.calc2()
    q.print_result(2)
    q.plot()



## classes ----------------------------------------------------------------
class Q_1_2:
    def __init__(self):
        self.t_f = 30.0  # s

        self.T_analysis = np.arange(0, self.t_f, 0.001)
        self.X_analysis = np.zeros_like(self.T_analysis)

        self.dt = 1.0    # s
        self.N = int(self.t_f/self.dt)
        self.T_pts = np.arange(0, self.t_f, self.dt)
        self.X_pts = np.zeros((self.N, 2), dtype=float)
        self.X_pts[0, :] = np.array([1, 0], dtype=float)

        self.A_dash = np.array([[0.582, 0.727], [-0.727, 0.364]],
    dtype=float)
        self.B_dash = np.array([0.418, 0.727], dtype=float)

    def u(self, t):
        return np.sin(2 * t)

    def x_analysis(self, t):
        alpha = -3/20
        beta = np.sqrt(391)/20
        first = (83/78) * np.exp(alpha * t) * (np.cos(beta * t) +
    (1249/1560/beta)*np.sin(beta * t))
        second = -(5/78) * (np.cos(2 * t) + 5 * np.sin(2 * t))
        return first + second

    def x_pts(self, k):
        t = k * self.dt
```

```
        x_k = self.X_pts[k, :]
        x_kplus1 = np.dot(self.A_dash, x_k) + self.B_dash * self.u(t)
        return x_kplus1

    def calc(self):
        self.X_analysis = self.x_analysis(self.T_analysis)
        for k in range(0, self.N-1):
            self.X_pts[k+1, :] = self.x_pts(k)
        #print(self.X_analysis)
        #print(self.X_pts)

    def plot(self):
        plt.plot(self.T_analysis, self.X_analysis, label="analyzed solution")
        plt.plot(self.T_pts, self.X_pts[:, 0], label="parsed-time-system
    solution ($\Delta t = 1.0$)")
        plt.xlabel("$t$[s]")
        plt.ylabel("$x$[m]")
        plt.legend()
        plt.savefig("../figures/q_1_2.png")
        plt.show()


class Q_2_2:
    def __init__(self):
        self.n = 1000
        self.N = np.arange(0, self.n, 1)
        self.sum_sq_solution = (91/3) * np.ones(self.n, dtype=float)
        self.mlt_solution = (49/4) * np.ones(self.n, dtype=float)
        self.seeds = [10, 100, 1000]    # seeds for np.random
        self.E_sum_sq_list, self.E_mlt_list = [], []

    def mkrndXY(self, seed):
        np.random.seed(seed)
        x = np.random.randint(1, 7, self.n)
        y = np.random.randint(1, 7, self.n)
        sum_sq = x ** 2 + y ** 2
        mlt = x * y
        # print(sum_sq)
        # print(mlt)
        return sum_sq, mlt

    def calc_E(self, seed):
        sum_sq, mlt = self.mkrndXY(seed)
        E_sum_sq = np.zeros(self.n, dtype=float)
        E_mlt = np.zeros(self.n, dtype=float)
        for i in range(1, self.n):
            E_sum_sq[i] = (E_sum_sq[i - 1] * (i - 1) + sum_sq[i]) / i
```

```
            E_mlt[i] = (E_mlt[i - 1] * (i - 1) + mlt[i]) / i
        # print(E_sum_sq)
        # print(E_mlt)
        return E_sum_sq, E_mlt

    def mkElist(self):
        for seed in self.seeds:
            E_sum_sq, E_mlt = self.calc_E(seed)
            self.E_sum_sq_list.append(E_sum_sq)
            self.E_mlt_list.append(E_mlt)

    def plot_sum_eq(self):
        for E, seed in zip(self.E_sum_sq_list, self.seeds):
            plt.plot(self.N, E, label="seed = {}".format(seed))
        plt.plot(self.N, self.sum_sq_solution)
        plt.xlim([1, self.n])
        plt.ylim([20, 40])
        plt.xlabel("$n$")
        plt.ylabel("$E[X^2 + Y^2]$")
        plt.legend()
        plt.show()

    def plot_mlt(self):
        for E, seed in zip(self.E_mlt_list, self.seeds):
            plt.plot(self.N, E, label="seed = {}".format(seed))
        plt.plot(self.N, self.mlt_solution)
        plt.xlim([1, self.n])
        plt.ylim([5, 17.5])
        plt.xlabel("$n$")
        plt.ylabel("$E[XY]$")
        plt.legend()
        plt.savefig("../figures/q_2_2.png")
        plt.show()


class Q_4_5:
    def __init__(self):
        self.smoking_rate = np.array([18.2, 25.82, 18.24, 28.6, 31.1, 33.6, 40.46, 28.27, 20.1, 27.91, 26.18, 22.12])
        self.death_rate = np.array([17.05, 19.8, 15.98, 22.07, 22.83, 24.55, 27.27, 23.57, 13.58, 22.8, 20.3, 16.59])
        self.a = 0
        self.b = 0
        self.y_curve = lambda x: self.a * x + self.b
        self.e = np.zeros_like(self.smoking_rate)
        self.mu = 0
        self.sigma_sq = 0
```

```
    def least_square(self, x, y):
        c = np.polyfit(x, y, 1)
        self.a = c[0]
        self.b = c[1]
        self.y_curve = np.poly1d(c)
        return self.y_curve

    def calc(self):
        self.least_square(self.smoking_rate, self.death_rate)
        y_pred = self.y_curve(self.smoking_rate)
        self.e = self.death_rate - y_pred
        self.mu = np.average(self.e)
        self.sigma_sq = np.var(self.e)

    def print_result(self):
        print("a        : {}".format(self.a))
        print("b        : {}".format(self.b))
        print("mean     : {}".format(self.mu))
        print("variance : {}".format(self.sigma_sq))

    def plot(self):
        x_min = int(np.min(self.smoking_rate) - 3)
        x_max = int(np.max(self.smoking_rate) + 3)
        x = np.arange(x_min, x_max, 1)
        plt.plot(x, self.y_curve(x), label="1D")
        plt.plot(self.smoking_rate, self.death_rate, ".", label="real data")
        plt.xlabel("smoking rate [%]")
        plt.ylabel("death rate [%]")
        plt.savefig("../figures/q_4_5.png")
        plt.show()


class Q_4_5_2:
    def __init__(self):
        self.Time = np.arange(0, 13900, 280)
        self.Temp = np.array([12.8, 13.3, 14.1, 14.3, 14.7, 14.9, 15.6, 16.1, 16.2, 15.4,
                              14.4, 13.0, 12.0, 11.0, 10.0, 9.4, 9.6, 9.8, 10.7, 11.1,
                              12.1, 12.2, 13.1, 13.5, 14.2, 14.6, 14.5, 15.1, 15.4, 16.0,
                              15.6, 14.2, 13.2, 12.0, 10.6, 9.8, 8.9, 9.5, 9.5, 10.4,
                              10.7, 11.7, 12.2, 12.4, 13.1, 14.0, 14.3, 14.5, 15.0, 15.6])
        self.params = np.array([13.0, 3.0, 2*np.pi/6000.0, 0.1], dtype=float)
```

```
        self.e = np.zeros_like(self.Time)
        self.mu = 0
        self.sigma_sq = 0

    def T_model(self, param, t):
        T_0 = param[0]
        a = param[1]
        omega = param[2]
        theta = param[3]
        T_pred = T_0 + a * np.sin(omega * t + theta)
        return T_pred

    def residual_func(self, param, t, T_observed):
        T_pred = self.T_model(param, t)
        return T_observed - T_pred

    def least_square(self):
        result = optimize.leastsq(self.residual_func, self.params,
    args=(self.Time, self.Temp))
        self.params = result[0]
        return self.params

    def calc(self):
        self.least_square()
        self.e = self.residual_func(self.params, self.Time, self.Temp)
        self.mu = np.average(self.e)
        self.sigma_sq = np.var(self.e)

    def print_result(self):
        print("T0       : {}".format(self.params[0]))
        print("a        : {}".format(self.params[1]))
        print("omega    : {}".format(self.params[2]))
        print("theta    : {}".format(self.params[3]))
        print("mean     : {}".format(self.mu))
        print("variance : {}".format(self.sigma_sq))

    def plot(self):
        t = np.arange(0, 13900, 280)
        plt.plot(t, self.T_model(self.params, t), label="model")
        plt.plot(self.Time, self.Temp, ".", label="real data")
        plt.xlabel("Time [s]")
        plt.ylabel("Temperature [deg]")
        plt.savefig("../figures/q_4_5_2.png")
        plt.show()


class Q_5_1:
```

```python
    def __init__(self, seed=10):
        self.year, self.price = self.load_data()
        self.sample = int(0.3 * len(self.year))     # use 30% of the data for training
        self.year_train, self.year_test, self.price_train, self.price_test = \
            self.parse_data(self.year, self.price, self.sample, seed)
        self.n_dim = 4      # max dimension for fitting

        self.params_list = []
        self.result_list = []
        for i in range(self.n_dim):
            dim = i + 1
            params = np.ones(dim+1, dtype=float)    # list for storing parameters (like a_0, a_1, ...)
            result = np.zeros(4, dtype=float)       # list for storing mean, variance, AICc, and Q
            self.params_list.append(params)
            self.result_list.append(result)
        self.polynomial_list = [self.poly_1D, self.poly_2D, self.poly_3D, self.poly_4D]
        self.residual_func_list = [self.residual_func_1D, self.residual_func_2D, self.residual_func_3D, self.residual_func_4D]
        self.linestyle_list = ["solid", "dashed", "dashdot", "dotted"]    # line style for each dimensions

    def load_data(self):
        with open("../data/mazda_cars.txt", "r") as f:
            f.readline()    # delete header
            year_list = []
            price_list = []
            for line in f:
                year_str, price_str = line.split("\t")
                year = int(year_str)
                price = int(price_str)
                year_list.append(year)
                price_list.append(price)
        year = np.array(year_list)
        price = np.array(price_list)
        return year, price

    def parse_data(self, x, y, sample, seed=10):
        n = len(x)
        np.random.seed(seed)
        idx_list = np.random.randint(0, n-1, sample)
        x_train = np.zeros(sample, dtype=float)
        y_train = np.zeros(sample, dtype=float)
```

```
        for i, idx in enumerate(idx_list):
            x_train[i] = x[idx]
            y_train[i] = y[idx]
        x_test = np.delete(x, idx_list)
        y_test = np.delete(y, idx_list)
        return x_train, x_test, y_train, y_test

    def poly_1D(self, params, x):
        return params[0] + params[1] * x

    def poly_2D(self, params, x):
        return params[0] + params[1] * x + params[2] * x**2

    def poly_3D(self, params, x):
        return params[0] + params[1] * x + params[2] * x**2 + params[3] * x**3

    def poly_4D(self, params, x):
        return params[0] + params[1] * x + params[2] * x**2 + params[3] * x**3 + params[4] * x**4

    def residual_func_1D(self, params, x, price_known):
        price_pred = self.poly_1D(params, x)
        return price_known - price_pred

    def residual_func_2D(self, params, x, price_known):
        price_pred = self.poly_2D(params, x)
        return price_known - price_pred

    def residual_func_3D(self, params, x, price_known):
        price_pred = self.poly_3D(params, x)
        return price_known - price_pred

    def residual_func_4D(self, params, x, price_known):
        price_pred = self.poly_4D(params, x)
        return price_known - price_pred

    def least_square(self, residual_func, params):
        result = optimize.leastsq(residual_func, params, args=(self.year_train, self.price_train))
        params = result[0]
        return params

    def AICc(self, n, k, Q):
        aicc = n * np.log(Q/n) + (2*k*n)/(n-2*k-1)      # for not so large n
        return aicc

```

```
    def calc_params(self):
        for i in range(self.n_dim):
            func = self.residual_func_list[i]
            params = self.params_list[i]
            params_calced = self.least_square(func, params)
            e = func(params_calced, self.year_train, self.price_train)
            self.result_list[i][0] = np.average(e)   # mean
            self.result_list[i][1] = np.var(e)       # variance
            self.params_list[i] = params_calced

    def calc_train_AICc(self):
        n = len(self.year_train)
        for i in range(self.n_dim):
            func = self.residual_func_list[i]
            params = self.params_list[i]
            e = func(params, self.year_test, self.price_test)
            Q = (np.linalg.norm(e))**2
            k = len(params)
            aicc = self.AICc(n, k, Q)
            self.result_list[i][2] = aicc

    def calc_test_Q(self):
        for i in range(self.n_dim):
            func = self.residual_func_list[i]
            params = self.params_list[i]
            e = func(params, self.year_test, self.price_test)
            Q = (np.linalg.norm(e))**2
            self.result_list[i][3] = Q

    def calc(self):
        self.calc_params()
        self.calc_train_AICc()
        self.calc_test_Q()

    def print_result(self):
        for i in range(self.n_dim):
            dim = i + 1
            params = self.params_list[i]
            result = self.result_list[i]
            print("Dimension : {}".format(dim))
            for j in range(dim + 1):
                print("\t a{}       : {}".format(j, params[j]))
            print("\t mean     : {}".format(result[0]))
            print("\t variance : {}".format(result[1]))
            print("\t AICc     : {}".format(result[2]))
            print("\t Q (test) : {}".format(result[3]))
            print()
```

```
    def plot_train(self):
        plt.plot(self.year_train, self.price_train, ".")
        x = np.arange(self.year.min()-2, self.year.max()+2, 1)
        for i in range(self.n_dim):
            func = self.polynomial_list[i]
            params = self.params_list[i]
            plt.plot(x, func(params, x), linestyle=self.linestyle_list[i],
    label="{}D Polynomial model".format(i+1))
        plt.xlabel("year")
        plt.ylabel("price")
        plt.ylim([0, self.price.max()+5000])
        plt.legend()
        plt.savefig("../figures/q_5_1_train.png")
        plt.show()

    def plot_test(self):
        plt.plot(self.year_test, self.price_test, ".")
        x = np.arange(self.year.min()-2, self.year.max()+2, 1)
        for i in range(self.n_dim):
            func = self.polynomial_list[i]
            params = self.params_list[i]
            plt.plot(x, func(params, x), linestyle=self.linestyle_list[i],
    label="{}D Polynomial model".format(i+1))
        plt.xlabel("year")
        plt.ylabel("price")
        plt.ylim([0, self.price.max()+5000])
        plt.legend()
        plt.savefig("../figures/q_5_1_test.png")
        plt.show()

    def plot_all_point(self):
        plt.plot(self.year, self.price, ".")
        plt.xlabel("year")
        plt.ylabel("price")
        plt.ylim([0, self.price.max()+5000])
        plt.legend()
        plt.savefig("../figures/q_5_1_all.png")
        plt.show()


class Q_5_3:
    def __init__(self):
        self.x = np.array([0.032, 0.034, 0.214, 0.263, 0.275, 0.275, 0.45,
    0.5,
                           0.5, 0.63, 0.8, 0.9, 0.9, 0.9, 0.9, 1.0,
```

```
                             1.1, 1.1, 1.4, 1.7, 2.0, 2.0, 2.0, 2.0],
dtype=float)
        self.y = np.array([170, 290, -130, -70, -185, -220, 200, 290,
                           270, 200, 300, -30, 650, 150, 500, 920,
                           450, 500, 500, 960, 500, 850, 800, 1090],
dtype=float)
        self.polynomial_list = [self.poly_1D_noconst, self.poly_1D]
        self.residual_func_list = [self.residual_func_1D_noconst,
self.residual_func_1D]
        poly_1D_noconst_params = np.array([0], dtype=float)
        poly_1D_params = np.array([0, 0], dtype=float)
        self.params_list = [poly_1D_noconst_params, poly_1D_params]
        poly_1D_noconst_result = np.array([0, 0, 0], dtype=float)
        poly_1D_result = np.array([0, 0, 0], dtype=float)
        self.result_list = [poly_1D_noconst_result, poly_1D_result]

    def poly_1D_noconst(self, params, x):
        return params[0] * x

    def poly_1D(self, params, x):
        return params[0] + params[1] * x

    def residual_func_1D_noconst(self, params, x, y_known):
        y_pred = self.poly_1D_noconst(params, x)
        return y_known - y_pred

    def residual_func_1D(self, params, x, y_known):
        y_pred = self.poly_1D(params, x)
        return y_known - y_pred

    def least_square(self, residual_func, params):
        result = optimize.leastsq(residual_func, params, args=(self.x,
self.y))
        params = result[0]
        return params

    def AICc(self, n, k, Q):
        aicc = n * np.log(Q/n) + (2*k*n)/(n-2*k-1)
        return aicc

    def calc_AICc(self):
        n = len(self.x)
        for i in range(2):
            func = self.residual_func_list[i]
            params = self.params_list[i]
            e = func(params, self.x, self.y)
            Q = (np.linalg.norm(e))**2
```

```
            k = len(params)
            aicc = self.AICc(n, k, Q)
            self.result_list[i][2] = aicc

    def calc(self):
        for i in range(2):
            func = self.residual_func_list[i]
            params = self.params_list[i]
            params_calced = self.least_square(func, params)
            e = func(params_calced, self.x, self.y)
            self.result_list[i][0] = np.average(e)
            self.result_list[i][1] = np.var(e)
            self.params_list[i] = params_calced
        self.calc_AICc()

    def print_result(self):
        title_list = ["without const.", "with const."]
        for i in range(2):
            params = self.params_list[i]
            result = self.result_list[i]
            print(title_list[i])
            for j, param in enumerate(params):
                print("   a{}       : {}".format(j, param))
            print("   mean     : {}".format(result[0]))
            print("   variance : {}".format(result[1]))
            print("   AICc     : {}".format(result[2]))
            print()

    def plot(self):
        title_list = ["without const.", "with const."]
        plt.plot(self.x, self.y, ".", label="real data")
        x = np.arange(self.x.min()-1, self.x.max()+1, 1)
        for i in range(2):
            func = self.polynomial_list[i]
            params = self.params_list[i]
            plt.plot(x, func(params, x), label="1D Polynomial model
    ({})".format(title_list[i]))
        plt.xlabel("distance")
        plt.ylabel("recession velocity")
        plt.legend()
        plt.savefig("../figures/q_5_3.png")
        plt.show()


class Q_5_4:
    def __init__(self):
        self.month = np.arange(0, 12*15+10, 1)
```

```
        self.temp = self.load_data_from_npy()
        self.n = 12*15
        self.temp_train = self.temp[:self.n]
        self.temp_test = self.temp[self.n:]
        self.month_train = self.month[:self.n]
        self.month_test = self.month[self.n:]
        self.m_list = [1, 2, 12, 24]
        self.temp_pred_list = []

    def load_data_from_csv(self):
        temp_list = []
        with open("../data/TokyoTemperatureSince2001.csv", "r") as f:
            reader = csv.reader(f)
            for row in reader:
                temp_str = row[1]
                temp = float(temp_str)
                temp_list.append(temp)
        temp = np.array(temp_list, dtype=float)
        np.save("../data/q_5_4.npy", temp)
        return temp

    def load_data_from_npy(self):
        temp = np.load("../data/q_5_4.npy")
        return temp

    def coeff(self, m, i, j):
        x_i = self.temp_train[m+1-i: self.n-i].copy()
        x_j = self.temp_train[m+1-j: self.n-j].copy()
        c = np.dot(x_i, x_j)
        return c

    def calc_a(self, m):
        C_ij = np.zeros((m, m), dtype=float)
        C_i0 = np.zeros(m, dtype=float)
        for idx_i in range(m):
            i = idx_i + 1
            C_i0[idx_i] = self.coeff(m, i, 0)
            for idx_j in range(m):
                j = idx_j + 1
                C_ij[idx_i, idx_j] = self.coeff(m, i, j)
        A = np.linalg.solve(C_ij, C_i0)
        return A

    def calc(self):
        for m in self.m_list:
            A_m = self.calc_a(m)
            temp_future = np.zeros(self.n-m, dtype=float)
```

```
            for t in range(m+1, self.n):
                temp_past = self.temp[t-1:t-m-1:-1]
                temp_future[t-(m+1)] = np.dot(A_m, temp_past)
                print(A_m)
                print(temp_past)
            self.temp_pred_list.append(temp_future)
            print(self.temp_pred_list)

    def plot(self):
        for i in range(len(self.m_list)):
            plt.plot(self.month_train, self.temp_train, marker="o",
    label="real data")
            m = self.m_list[i]
            temp_pred = self.temp_pred_list[i]
            months = self.month_train[m:]
            plt.plot(months, temp_pred, label="m = {}".format(m))
            plt.xlabel("Months from Jan. 2001")
            plt.ylabel("Average Temperature")
            plt.legend()
            plt.savefig("../figures/q_5_4_{}.png".format(m))
            plt.show()


class Q_6_1:
    def __init__(self):
        self.x = np.arange(-2, 3, 1)
        self.y = np.array([-1.8623, 0.6339, -2.2588, 2.0622, 2.7188])
        self.params1 = [0, 0, 1]
        self.params2 = [0, 0, 1]

    def poly_1D(self, params, x):
        return params[0] * x

    def residual_func_1D(self, params, x, y_known):
        y_pred = self.poly_1D(params, x)
        return y_known - y_pred

    def least_square(self, residual_func, params):
        result = optimize.leastsq(residual_func, params, args=(self.x,
    self.y))
        params = result[0]
        return params

    def calc1(self):
        params_calced = self.least_square(self.residual_func_1D,
    self.params1)
        self.params1 = params_calced
```

```
            e = self.residual_func_1D(self.params1, self.x, self.y)
            self.params1[-2] = np.average(e)
            self.params1[-1] = np.var(e)

    def print_result(self, i):
        if i == 1:
            params = self.params1
        else:
            params = self.params2
        print("a        : {}".format(params[0]))
        print("mean     : {}".format(params[1]))
        print("variance : {}".format(params[2]))
        print()

    def estimate_a(self, x, y):
        mu = 1
        var = 0.09
        A = (np.linalg.norm(x))**2 + 1/var
        B = np.dot(x.T, y) - (np.linalg.norm(x))**2
        a = mu + A**(-1) * B
        return a

    def calc2(self):
        a = self.estimate_a(self.x, self.y)
        self.params2[0] = a
        y_pred = self.poly_1D([a], self.x)
        e = self.y - y_pred
        self.params2[-2] = np.average(e)
        self.params2[-1] = np.var(e)

    def plot(self):
        plt.plot(self.x, self.y, ".")
        x = np.arange(self.x.min()-1, self.x.max()+2, 1)
        plt.plot(x, self.poly_1D(self.params1, x), linestyle="dashed",
    label="MLE")
        plt.plot(x, self.poly_1D(self.params2, x), linestyle="dashdot",
    label="MAP Estimation")
        plt.xlabel("$x$")
        plt.ylabel("$y$")
        plt.legend()
        plt.savefig("../figures/q_6_1.png")
        plt.show()



## execution -----------------------------------------------------------
if __name__ == "__main__":
```

```
    main()
```