1. code with detailed explanations

- ## Part1

i. Import package & load data

```
import numpy as np
from matplotlib import pyplot as plt
from scipy.optimize import minimize
```

This homework has a strict limitation of the package, so we just import these tree package in our whole code.

```
def build_data():
    X = []
    Y = []
    f = open("ML_HW05-1/input.data","r")
    each_line = f.readline()
    while each_line!="":
        X_Y = each_line.strip("\n").split(" ")
        X.append(float(X_Y[0]))
        Y.append(float(X_Y[1]))
        each_line = f.readline()
    return X, Y
```

```
X, Y = build_data()
```

Simpliy just use “open()” function to load our data.

ii. rational quadratic kernel

The formula of rational quadratic function is:

$$k(x_a, x_b) = \sigma^2 \left(1 + \frac{\|x_a - x_b\|^2}{2\alpha\ell^2}\right)^{-\alpha}$$

Implementation in our code:

```
def kernel_rational_quadratic(cls, x, y, l=1, a=5, var=1):
    return var * ((1 + abs((x-y)**2) / (2 * a * (l**2))) ** (-a))
```

Note that there are three hyperparameters in rational quadratic kernel function: L, alpha, var. We will optimize these three hyperparameters in part2.

iii. Train and predict

In gaussian prosses, we need to consider:

$$\begin{pmatrix} f \\ f_* \end{pmatrix} \sim \left(\begin{pmatrix} \mu \\ \mu * \end{pmatrix}, \begin{pmatrix} K & K* \\ K*^T & K** \end{pmatrix}\right)$$

We use “calculate_sigma()” function to calculate K:

```
def calculate_sigma(cls, x, cov_f):
    N = len(x)
    sigma = np.ones((N, N))
    for i in range(N):
        for j in range(N):
            sigma[i][j] = cov_f(x[i], x[j])
    return sigma
```

And the calculation of K*:

```
for i in range(self.N):
    sigma_1_2[i] = self.cov_f(self.x[i], x)
```

Note that in above function, “self.x” means the x coordinate of the 34 points in our input data, and “x” represents the new point we want to predict its y coordinate. Also, the function “self.cov_f()” is equal to our rational quadratic kernel function.

Calculate K**:

```
noise = 1/5
cov =  self.cov_f(x, x) + noise
```

The noise is 1/5 because the suggestion in the homework spec.

Finally, we can calculate u* and its variance:

```
m_pre = (sigma_1_2.T * np.mat(self.sigma).I) * np.mat(self.y).T
sigma_pre = cov - (sigma_1_2.T * np.mat(self.sigma).I) * sigma_1_2
```

Above code are the implement of these function:

$$\bar{y}_* = K_* K^{-1} \mathbf{y}, \qquad \mathrm{var}(y_*) = K_{**} - K_* K^{-1} K_*^{\mathrm{T}}.$$

- Part2

The log likelihood of the hyperparameters is:

$$\log p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}) = -\frac{1}{2}\mathbf{y}^{\mathrm{T}} K^{-1}\mathbf{y} - \frac{1}{2}\log|K| - \frac{n}{2}\log 2\pi.$$

The implementation:

```
def Cal_likelihood(args):
    X, Y = x, y
    N = len(X)
    l, a, var = args
    K = np.ones((N, N))
    for i in range(N):
        for j in range(N):
            K[i][j] = rational_quadratic(X[i], X[j], l, a, var)

    return -float((-1/2)*np.mat(Y)*np.mat(K).I*np.mat(Y).T - 1/2*np.log(abs(np.linalg.det(K))) - N/2*np.log(2*np.pi))
```

In above function, X, Y represent the coordinate of the input points (total 34 point), and the function parameters “args”, include three hyperparameters of rational quadratic kernel: L, alpha, variance.

Because in homework spec, we need to “minimize negative log-likelihood”, we put negative in our return value, and use scipy.optimize.minimize, which suggested in homework spec, to optimize our hyperparameters:

In [18]:
```
x0 = np.asarray((1, 1, 1))
```

In [19]:
```
res = minimize(Cal_likelihood, x0, method='SLSQP')
print(res.fun)
print(res.success)
print(res.x)
```

The variable “X0” is the initial guess.

2. experiments settings and results

- ### Part1

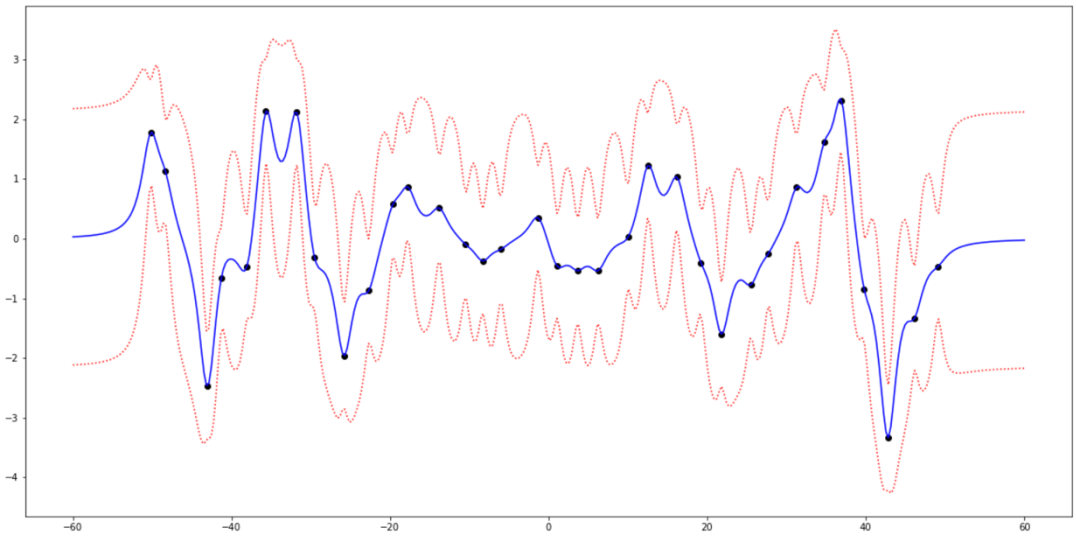
We use the code below to visualize our result:

```
x = np.array(X);
y = np.array(Y)
gaus = GP(x, y)

plt.figure(figsize=(20,10))
x_guess = np.linspace(-60, 60, 400)
y_pred = np.vectorize(gaus.predict)(x_guess)

plt.scatter(x, y, c="black")
plt.plot(x_guess, y_pred[0], c="b")
plt.plot(x_guess, y_pred[0] - 1.96*np.sqrt(y_pred[1]) * 1, "r:")
plt.plot(x_guess, y_pred[0] + 1.96*np.sqrt(y_pred[1]) * 1, "r:")
```

Result (L = alpha = variance = 1):



The black point in the graph represent the 34 input data points, and red line is the 95% confidence interval of our prediction.

- ### Part2

We run the code mention in above to optimize the hyperparameters:

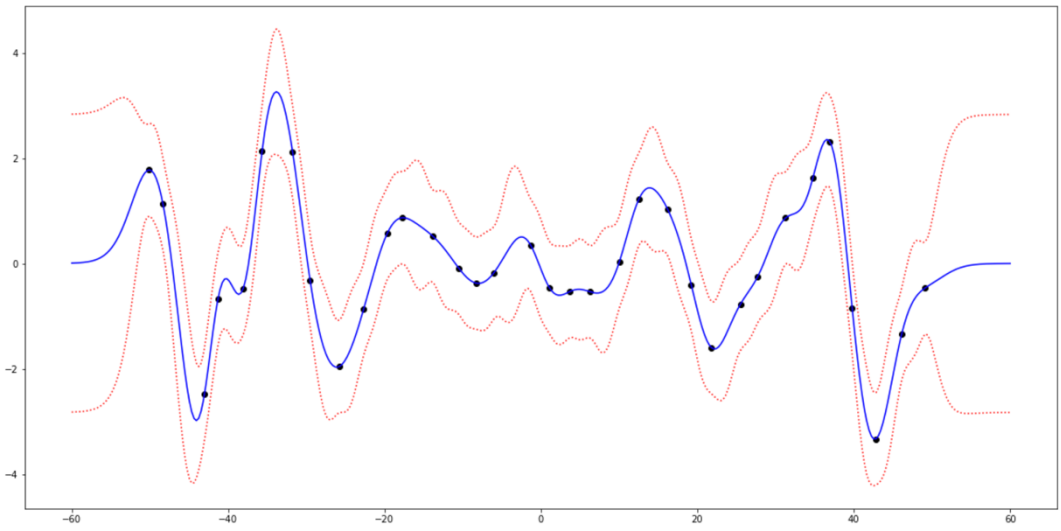
In [18]:
```
x0 = np.asarray((1, 1, 1))
```

In [19]:
```
res = minimize(Cal_likelihood, x0, method='SLSQP')
print(res.fun)
print(res.success)
print(res.x)
```

```
51.98967972798525
True
[2.47623475 8.50867061 1.88337348]
```

The above result mean when L = 2.47623475, alpha = 8.50867061, variance = 1.88337348, the negative log-likelihood will have the minimize value 51.98967972.

We set the hyperparameters, and visualize the result again (L = 2.47623475, alpha = 8.50867061, variance = 1.88337348):



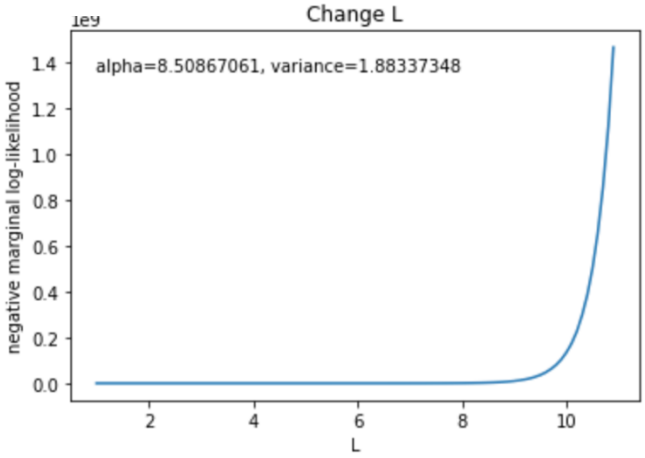
Obviously, the curve in part2 is smoothly than part1. More important, the 95% confidence interval is narrower in part two, that's mean the regression result is more certainly.

3. observations and discussion

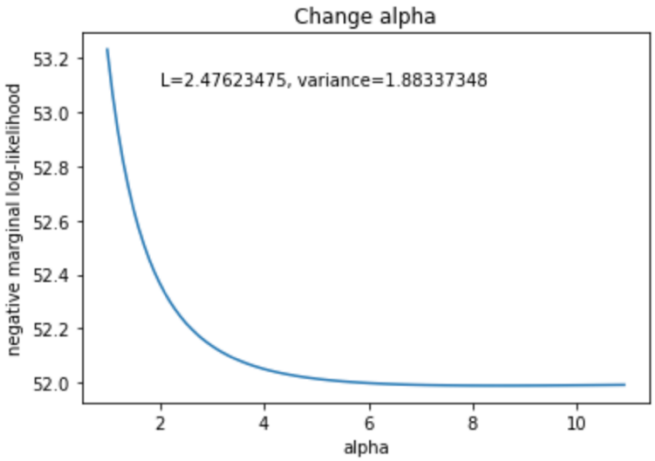
- hyperparameters

To discuss how the change of hyperparameters influent the regression result, in following section, we fix one hyperparameters in each experiment, and observe its negative marginal log-likelihood.

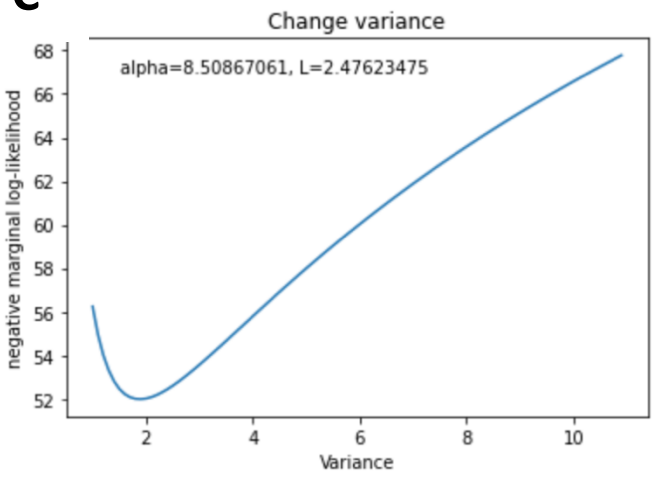
A



B



C



According to graph A, the negative marginal log-likelihood will get extremely higher when L increase. Note that the y-coordinate in graph A is much higher than other's graph. For example, the y-coordinate is "1466783361.0671484" when L = 11.

In contrast, graph B and C show the curve of negative marginal log-likelihood is more smoothly when we change alpha or variance. The

negative marginal log-likelihood is about 52~68 when alpha and variance is in the range of 1~11. Also, we can observe that the curve in graph C has a smallest value when variance is about 1.8. This confirms the result that when variance = 1.88337348, we can get the minimize negative marginal log-likelihood.

● Overfitting

According to above work, the curve of visualize result is very steep. This represents our Gaussian process model is overfitting. Once there is a new data appear, our model can't predict the new point properly. The reason of overfitting is because our data set is too small, we just use 34 pints to build the model. If we use more data to build Gaussian process, the regression will be more accurate.