Report

- Code with Detailed Explanation
  - Calculate kernel function

$$k_{RQ}(x_i, x_j) = \sigma^2 \left(1 + \frac{(x_i - x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

Where $(x_i - x_j)^2$ is RQ distance, l determines the length of wiggles in the function, σ is the average distance of the function away from its mean, α determines the relative weighting of large scale and small scale variations. When $\alpha \to \infty$, rational quadratic kernel will be identical to standard exponential kernel (a.k.a Gaussian kernel).

```
def kernel(X1, X2, a, l, sigma):
    '''
    X1: m points
    X2: n points
    K : (m,n)
    '''
    rqdist2 = np.sum(X1**2, 1).reshape(-1, 1) + np.sum(X2**2, 1) - 2 * X1 @ X2.T
    return sigma**2 * (1 + rqdist2/(2*a*l**2))**(-a)
```

  - Derive negative marginal log-likelihood

$$y_n = f(x_n) + \epsilon_n \ \text{, where } \ \epsilon \sim N(\cdot \mid 0, \beta^{-1}) \ \text{ and } \ \beta = 5$$

$$p(y \mid f) = N(y \mid f, \beta^{-1} I_N)$$

$$p(f) = N(0, K)$$

$$P(y) = \int p(y \mid f) p(f) df = N(y \mid 0, C_N) = \frac{1}{\sqrt{(2\pi)^N} \mid C_N \mid^{1/2}} \exp\left(-\frac{1}{2} y^T C_N^{-1} y\right)$$

where $C_N(x_i, x_j) = k(x_i, x_j) + \beta^{-1}\delta_{ij}$

$$\Rightarrow nll = \frac{1}{2}\log(\mid C_N \mid) + \frac{N}{2}\log(2\pi) + \frac{1}{2} y^T C_N^{-1} y$$

```
def nll(X_train, Y_train, noise):
    Y_train = Y_train.ravel()
    def nll_naive(theta):
        C = kernel(X_train, X_train, a=theta[0], l=theta[1], sigma=theta[2]) + \
            noise*np.identity(N) # noise = 1/beta = 0.2
        return 0.5*np.log(det(C)) + 0.5*N*np.log(2*np.pi) + 0.5*Y_train.T @ inv(C) @ Y_train
    return nll_naive
```

  - Optimize kernel hyperparameters $\theta = (\alpha, l, \sigma)$

Set initial guess $\theta^{(0)} = (\alpha^{(0)}, l^{(0)}, \sigma^{(0)}) = (1,1,1)$, since all parameters are positive, let's set their bounds from 1e−5 to ∞, and apply L-BFGS-B method to optimize θ.

```
from scipy.optimize import minimize
theta = minimize(nll(X_train, Y_train, noise), x0=[1, 1, 1], method='L-BFGS-B',
                     bounds=((1e-5, None),(1e-5, None),(1e-5, None)))
a, l, sigma = theta.x
```

- Prediction

  Given test data, return their means and variances (in covariance.diagonal()).

$$\mu(x_{N+1}) = k^T C_N^{-1} y \ , \ \sigma^2(x_{N+1}) = c + k^T C_N^{-1} k \ , \ C_{N+1} = \begin{bmatrix} C_N & k \\ k^T & c \end{bmatrix}$$

```
def posterior(X_test, X_train, Y_train, noise, a, l, sigma):
    C_N     = kernel(X_train, X_train, a, l, sigma) + noise * np.identity(N)
    C_N_inv = inv(C_N)
    K       = kernel(X_train, X_test, a, l, sigma) # (N,N_test)
    c       = kernel(X_test, X_test, a, l, sigma) + noise * np.identity(X_test.shape[0])
                                                                          # (N_test, N_test)
    mu_new = K.T @ C_N_inv @ Y_train # (N_test,1)
    cov_new = c - K.T @ C_N_inv @ K # (N_test, N_test)
    return mu_new, cov_new
```

- Plot

  Calculate 95% confidence interval, and plot means with 95% confidence interval.

$$\text{95\% confidence interval } = \left[\mu - 1.96\frac{\sigma}{\sqrt{N}}, \mu + 1.96\frac{\sigma}{\sqrt{N}}\right]$$

```
def plot(X_train, Y_train, noise, a, l, sigma):
    fig, (ax) = plt.subplots(1, 1, figsize=(8, 6))
    ax.set_ylim(-6, 6)

    # Show all training data points
    ax.scatter(X_train, Y_train, color='steelblue', marker='.')

    X = np.linspace(-60,60,1000)
    mu, cov = posterior(X.reshape(-1,1), X_train, Y_train, noise, a, l, sigma)
    mu, var = mu.ravel(), cov.diagonal()

    # Draw a line to represent the mean of f in range
    ax.plot(X, mu, color='steelblue')

    # Mark 95% confidence interval of f
    confidence = 1.96*(np.sqrt(var/34))
    ax.fill_between(X, mu-confidence, mu+confidence, color='steelblue', alpha=0.5)
    plt.show()
```

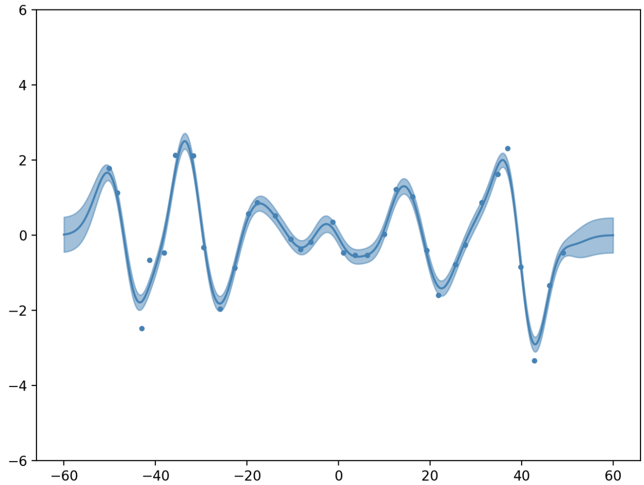
- Experiments Settings and results

  Set distribution of noise: $y_n = f(x_n) + \epsilon_n$, where $\epsilon \sim N(\cdot\,|\,0, \beta^{-1})$ and $\beta = 5$

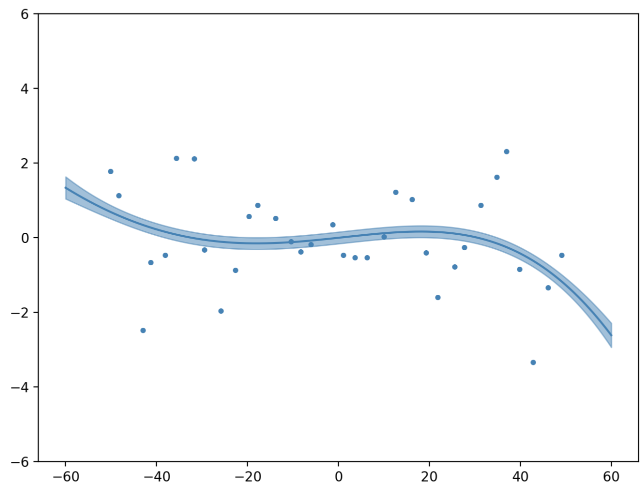
  Set initial guess at $\theta = (1, 1, 1)$, with each bound from 1e−5 to ∞, and then apply L-BFGS-B method to optimize θ.

  We get optimized $\theta = (\alpha, l, \sigma) = (275.5275, 3.3116, 1.3107)$ with negative log-likelihood = 50.6827.

  By prediction of each x in [-60, 60], and plot with mean and 95% confidence interval. (Code explanations are on preceding page)



  Set initial guess at $\theta = (100, 100, 100)$, and we get optimized $\theta = (\alpha, l, \sigma) = (100.7633, 100.6455, 99.9756)$, negative log-likelihood = 153.2907



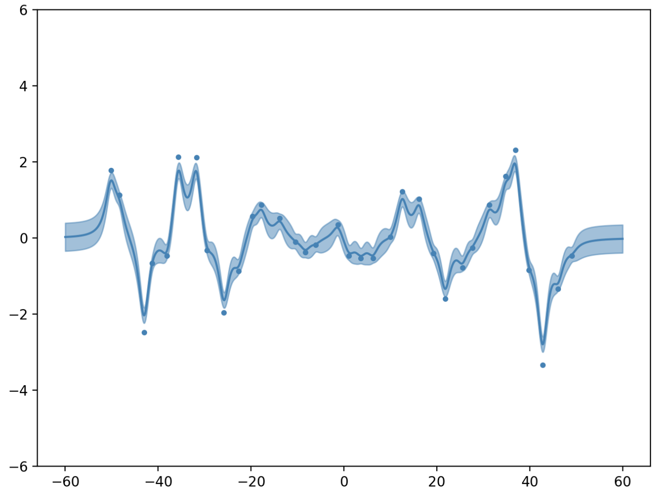
  It's clear that since L-BFGS-B method is gradient-based method, the result is vulnerable to starter value/initial guess. If the initial guess is far away from the global optimum, then it may be easy to get stuck in a local optimum. Thus, we can generate a population of initial guesses to relieve this problem.

- Observations and Discussion

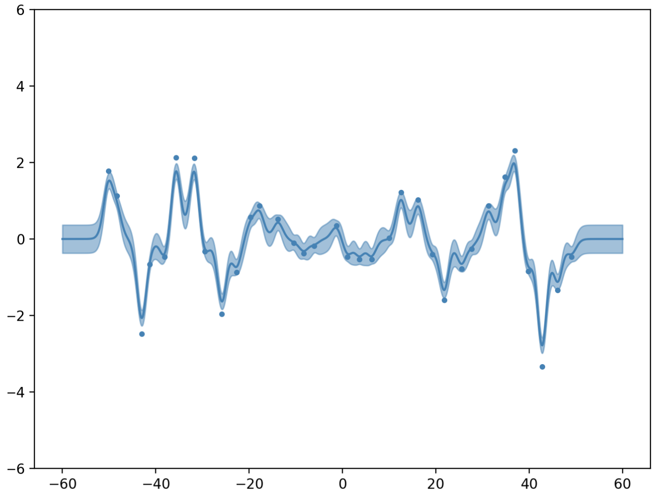
  l determines the length of wiggles in the function, σ is the average distance of the function away from its mean, α determines the relative weighting of large scale and small scale variations. (By hand-tuning hyperparameters, we can see how each hyperparameter affects our model)

  It's clear to observe how l and σ affect our model in the cases below.

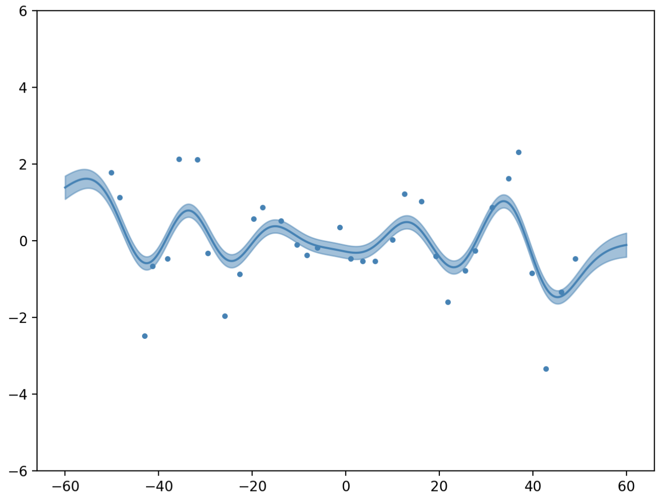
$(\alpha, l, \sigma) = (1, 1, 1)$, nll = 55.9227



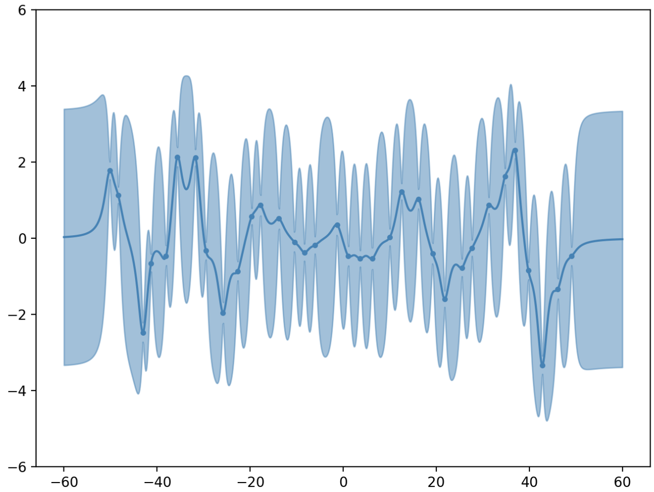
$(\alpha, l, \sigma) = (10,\ 1,\ 1)$, nll = 56.7917



$(\alpha, l, \sigma) = (1,\ 10,\ 1)$, nll = 102.8457

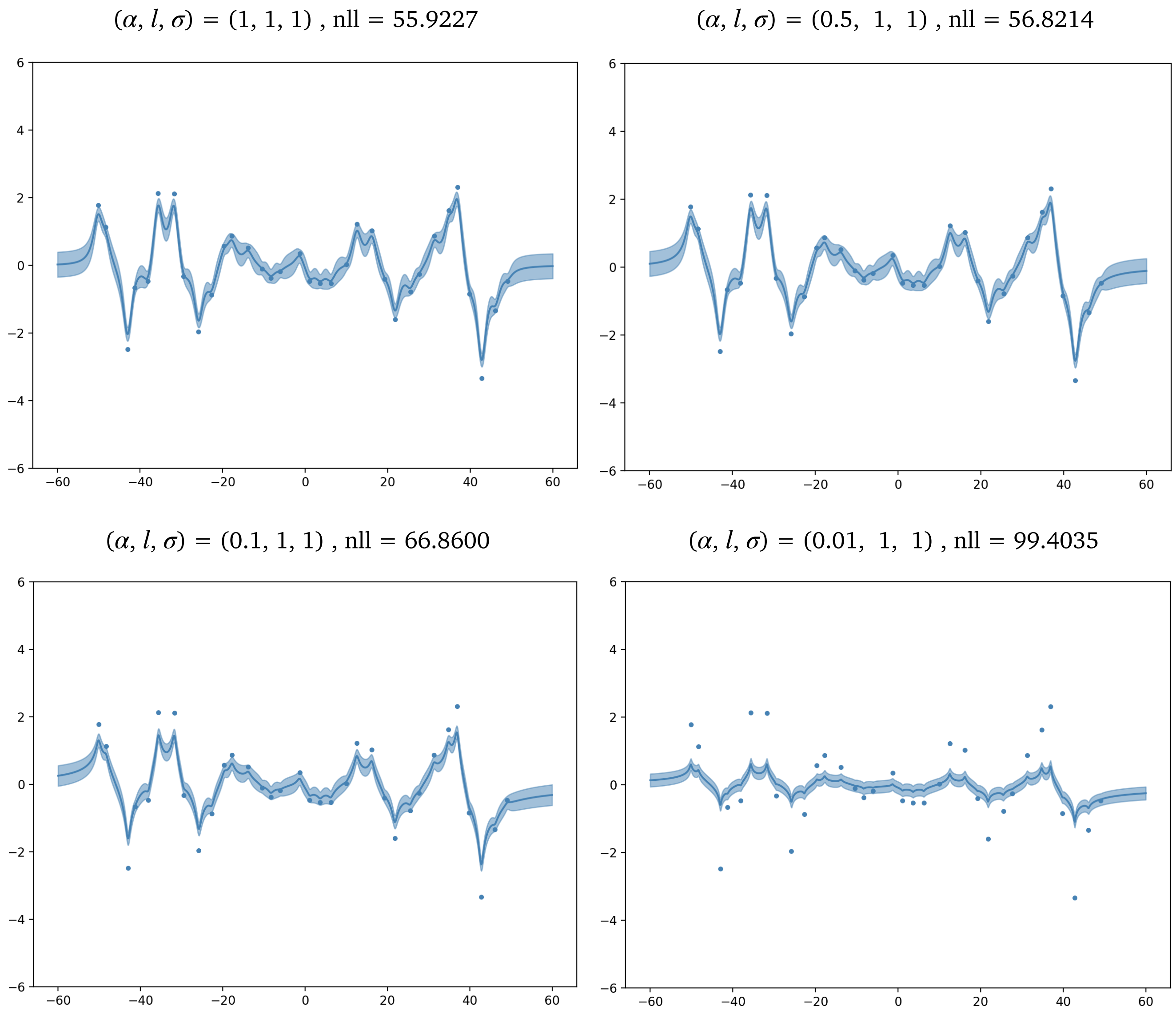


$(\alpha, l, \sigma) = (1,\ 1,\ 10)$, nll = 108.9840



As α increases, the prediction function becomes steeper; as l increases, the length of wiggles become larger, the density of peaks declines, and the prediction function becomes smoother; as σ increases, the variations becomes larger (i.e. further away from mean).

Let's take a further look at α:



It's clear that as α decreases, the prediction function becomes biased/less accurate, yet the number of peaks are the same.