# Gaussian Processes for Detection of Muscle Co-contraction

Oct. 2020

**Chenkuan Liu**

# 1 Introduction

In this project, we are using Gaussian Processes to detect muscle co-contraction when an individual is tracing a curve. Each tracing is counted as one trial. The data we have are five trials from an individual and, numerically, each trial is composed of the 3D locations of 50 markers located in different parts of the body through a given time range.

To have better observation, we will specifically choose the marker located on the right arm of the individual (the $8^{th}$ marker in our data). The five trials are divided into four training trials and one testing trial. We will use both global approach and sliding-window approach to fit the Gaussian kernel from the training trials and make predictions on the 3D locations of the marker in testing trial through time. The comparison of kernel parameters in the sliding-window approach will also provide information on potential muscle co-contraction during the curve tracing.

# 2 Method

## 2.1 Global Window Approach

In global window approach, we use the data from training trials through the entire time period to fit a single kernel, and then use this kernel to make predictions on the testing trial.

After importing and combining the dataframe from the four training trials, we first filter out

the time entries that contain negative $c$ values, which indicate that the locations traced at these specific time entries are invalid. Then, we randomly selected half of the entries as our training data, which will make our computation faster.

After that, we use `RBF()` and `GaussianProcessRegressor()` in Python's `sklearn` library to set our initial kernel and fit the training data. The detailed algorithm will be demonstrated in the *Algorithm* section below. After training, we will use the testing time entries to predict the corresponding 3D location (namely the mean value) and the variance at each location. We will also compute the mean squared error as well as correlation in each axis to evaluate our result.

### 2.2 Sliding Window Approach

In sliding window approach, most of the steps are identical to those in global window approach. The only major difference is that, at here, we divide the entire time range into different "chunks" and each chunk is viewed as a window. We fit the kernel parameters and predict the locations in each window separately and combine our results together to form our overall prediction. Note that, to better detect the change of kernel parameters, the windows are allowed to overlap. In the overlapped region, we will get more than one predictions from different kernel parameters. To better formulate our results, we will take the mean values and mean variances from the multiple predictions inside the overlapped regions.

## 3 Algorithm

### 3.1 Data Import and Preprocessing

I am using the tracing data from the second subject CJ. The marker ID is 8 as mentioned above, which is located on the right upper arm close to the elbow. There are five trials in total; the first four are used as training trials and the last one is used as the testing trial. After importing the five corresponding `csv` files into five `pandas` dataframes `df1`, `df2`, `df3`, `df4` and `df5`, the code is:

```
data_1 = df1.loc[:,['elapsed_time','8_x','8_y','8_z','8_c']]
data_2 = df2.loc[:,['elapsed_time','8_x','8_y','8_z','8_c']]
data_3 = df3.loc[:,['elapsed_time','8_x','8_y','8_z','8_c']]
data_4 = df4.loc[:,['elapsed_time','8_x','8_y','8_z','8_c']]

data_5 = df5.loc[:,['elapsed_time','8_x','8_y','8_z','8_c']]

data_1 = data_1[data_1['8_c'] > 0]
data_2 = data_2[data_2['8_c'] > 0]
data_3 = data_3[data_3['8_c'] > 0]
data_4 = data_4[data_4['8_c'] > 0]

data_5 = data_5[data_5['8_c'] > 0]

data = pd.concat([data_1, data_2, data_3, data_4])
data = data.sample(frac=0.50, random_state=0)

X_dt = np.array(data['elapsed_time']).reshape(-1,1)
y_dt = data.loc[:, ['8_x','8_y','8_z']]
```

We separate `data_5` into a single line as it is our testing data. After we filter out the row entries with invalid $c$ values, we concatenate the dataframes together, randomly select half of the rows, and reshape into training inputs `X_dt` and training outputs `y_dt`.

This part is the same for both global window method and sliding window method. The only difference I made in sliding window method is that I kept all the data from the first four trials for training without randomly selecting row entries. This is because, by partitioning into many chunks, the sliding window method by itself has much shorter running time than global window method.

## 3.2 Kernel Setup, Fitting, and Prediction

### 3.2.1 Global Window

The steps for global window are straightforward. The code is:

```
kernel = 1.0 * RBF(length_scale=10, length_scale_bounds=(1e-1, 1e6))
gpr = GaussianProcessRegressor(kernel=kernel, alpha=1e-3, optimizer='
    fmin_l_bfgs_b', n_restarts_optimizer=20, random_state=0)
```

```
gpr.fit(X_dt, y_dt)

X_pred = np.array(data_5['elapsed_time']).reshape(-1,1)
y_pred, cov = gpr.predict(X_pred, return_cov=True)
variances = np.diag(cov)
```

In the first line we use `RBF()` in `sklearn.kernel` to set up the kernel parameters. The initial $\sigma^2$ and initial length scale are set to 1.0 and 10 respectively. The length scale bounds are set between 0.1 to $10^6$ to limit the influence of overly small or large values. In the second line, we use the `GaussianProcessRegressor()` function to set up the Gaussian regressor. The kernel used comes from above, the `alpha` denotes the noise level to prevent potential numerical issue during fitting, and `n_restarts_optimizer` denotes the number of iterations during kernel parameter optimization. Detailed explanations of the two functions can be found inside `scikit-learn.org/stable/modules/gaussian_process.html#gaussian-process`. In the third line, we use `gpr.fit()` to fit the training input `X_dt` and training output `y_dt`.

Then, we reshape `data_5` to make predictions on its time values. Note that `return_cov` is set to `True` so that we can get the covariance matrix and use its diagonal entries to obtain variance at each predicted value.

### 3.2.2 Sliding Window

The functions called in sliding window method are the same as those in global window method, but it is slightly more complicated as we need to divide entire time range into each window and fit separately. After that, we also need to average the time values inside overlapped regions to get the predicted values and variances.

The algorithm can be divided into four steps: window setup, kernel setup, fitting, and prediction. The detailed code are displayed below:

#### a. Window Setup

```
# window setup
window = []
test = []
i = 0
while i + 1 <= 17.5:
```

```
    window.append(data.loc[(data['elapsed_time'] >= i) & (data['elapsed_time'] < i + 1)])
    test.append(data_5.loc[(data_5['elapsed_time'] >= i) & (data_5['elapsed_time'] < i + 1)])
    i += 0.5
```

Through a general observation of the datasets, we can find that all trials end the tracing before 17.5 second. Therefore, I am dividing the entire time range into 34 windows. The first window is $[0, 1)$, the second window is $[0.5, 1.5)$, and so on. Therefore, the $i^{th}$ window is $[0.5(i-1), 0.5(i+1))$, and the last window is $[16.5, 17.5)$. We perform the division on both training trials and testing trial. Note that, in this way, all time values inside $[0.5, 16.5)$ will have two sets of corresponding kernel parameters and two prediction results.

### b. Kernel Setup

```
# kernel setup
gpr = []
kernel = 1.0 * RBF(length_scale=10, length_scale_bounds=(1e-1, 1e6))
for j in range(len(window)):
    gpr.append(GaussianProcessRegressor(kernel=kernel, alpha=1e-3, optimizer='fmin_l_bfgs_b', n_restarts_optimizer=20, random_state=0))
```

The kernel setup is identical to that in global method. The difference is that we initialize 34 Gaussian regressors and append them into a list.

### c. Gaussian Fitting

```
# fitting
k1 = []
k2 = []
for j in range(len(window)):
    train = window[j]
    X = np.array(train['elapsed_time']).reshape(-1, 1)
    y = np.array(train.loc[:, ['8_x', '8_y', '8_z']])
    gpr[j].fit(X, y)
    print("kernel", j, "params:", gpr[j].kernel_.get_params())
    k1.append(gpr[j].kernel_.get_params()['k1__constant_value'])
    k2.append(gpr[j].kernel_.get_params()['k2__length_scale'])
```

We use the $j^{th}$ Gaussian regressor to fit the $j^{th}$ kernel. The following `print` commands

can display the optimized hyperparameters in each kernel. `k1` and `k2` are used to store $\sigma^2$ and length scale values respectively.

### d. Prediction

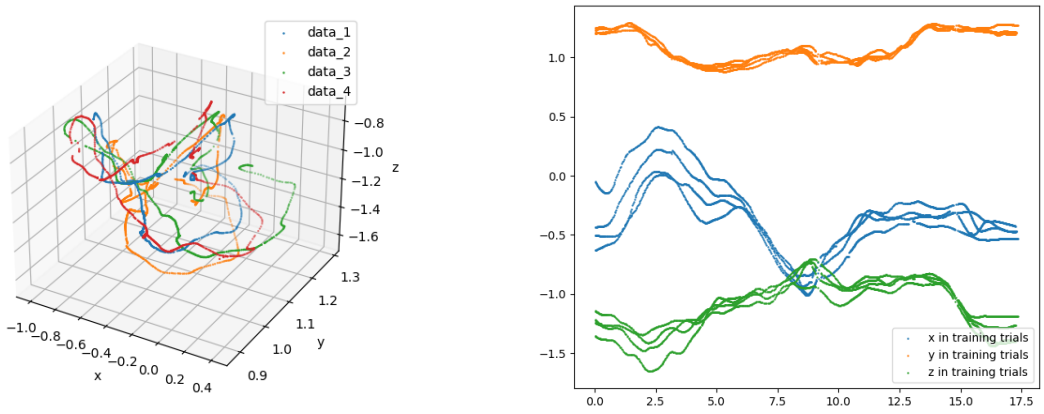
```
# predicting
X_inputs = []
y_predictions = []
y_covs = []
outputs = []
covs = []
for j in range(len(window)):
    pred = test[j]
    X_test = np.array(pred['elapsed_time'])
    X_inputs.append(X_test)

    y_test = np.array(pred.loc[:, ['8_x', '8_y', '8_z']])
    y_pred, cov = gpr[j].predict(X_test.reshape(-1, 1), return_cov=True)

    y_predictions.append(y_pred)
    y_covs.append(cov)

    outputs.append(np.insert(y_pred, 0, X_test, axis=1))
    covs.append(np.insert(np.diag(cov).reshape(-1, 1), 0, X_test, axis=1))
```

We predict each window separately to get the mean values `y_pred` and covariance matrix `cov`. We append the means into the list `outputs` and the diagonal entries of the covariance matrices into `covs` respectively.

Note that our sliding window method is now complete. We can then reorganize the two lists into different chunks, and reformat the inside multidimensional arrays so that for the time values below 0.5 or greater than 17.0 (including 17.0), the mean values and variances remain unchanged; for the time values in between, we take the average results from the two associated windows.

# 4 Results

## 4.1 Training Data




(a) 3d scatter plot

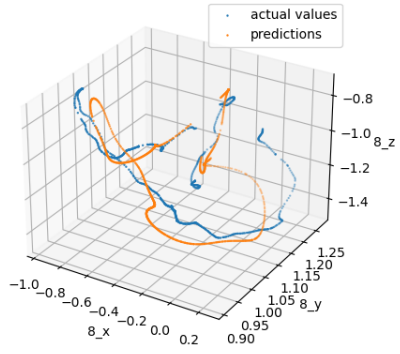
(b) 2d scatter plot

Figure 1: Scatter plots of the traces of four training trials

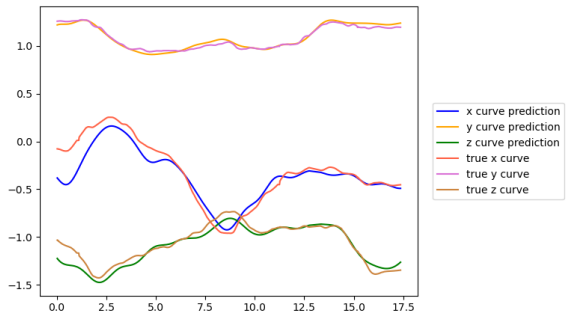
Figure 1 shows the tracing points of the 4 training trials displayed in both 3d and 2d. In (a), the four trials are showed in blue, orange, green and red color in order; in (b), the 4 blue curves show the change of $x$ coordinate value over time in the 4 trials, and the orange and green curves show the change of $y$ and $z$ coordinate values over time respectively.

## 4.2 Global Window Results

The predicted values over time in both 3d and 2d, the variance over time, and the mean squared error and correlation in each axis are displayed below. For the prediction, we can observe that, though the predictions of $x$ coordinate and $z$ coordinate are slightly inaccurate at the beginning, the performance is decent overall. Since the time values of almost all the data are between 0.1 and 17.4, we can see that the variance rises up correspondingly at the beginning and the end, while remains very small in between.

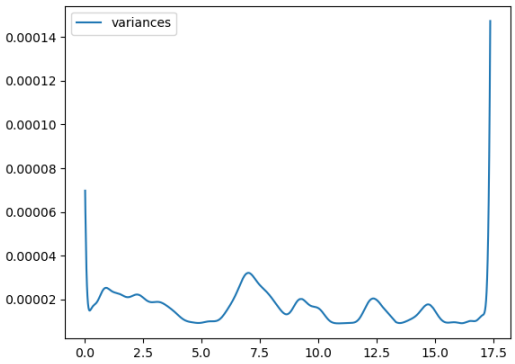



(a) results in 3d, blue is actual values and orange is predicted values




(b) results in 2d, top two are $y$ locations (real and predicted), middle two are $x$, bottom two are $z$

Figure 2: Comparison of predicting curve and actual trace in global approach




(a) variance over time

```
sigma^2: 0.3969664398003677
length scale: 1.0003389367457016
MSE in x: 0.01268234352008708
MSE in y: 0.0007044338011299857
MSE in z: 0.004141555838527904
MSE total: 0.017528333159744957
correlation in x:
 [[1.        0.9498338]
 [0.9498338 1.       ]]
correlation in y:
 [[1.         0.98375644]
 [0.98375644 1.        ]]
correlation in z:
 [[1.         0.94989161]
 [0.94989161 1.        ]]
```

(b) MSE and correlations

Figure 3: Performance Measure

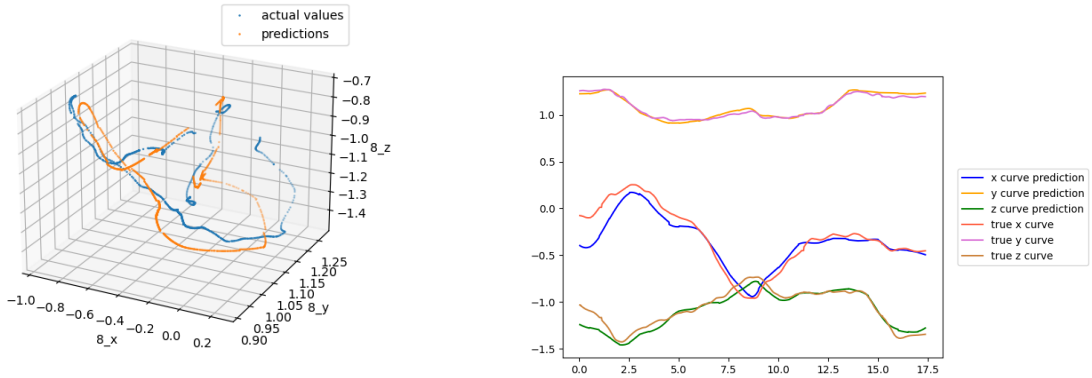
Figure 3(b) also displays the kernel parameters, mean squared errors in each axis and in general, and the correlation between predicted values and actual values over time in each axis. We can see that prediction over $x$ axis has the highest MSE, and the total MSE of the three axes is approximately 0.0175. The correlation between predicted values and actual

values on each axis is above 0.94. They indicate that the prediction is decent overall. Also, $\sigma^2$ is approximately 0.4, and length scale can be rounded to 1.0. Further comparison with the kernel parameters in sliding window method on the next section will show us that the $\sigma^2$ and length scale obtained here are less than those obtained in any of the sliding window.

## 4.3 Sliding Window Results

From Figure 4, we can observe that the predicting traces in sliding window approach are very similar to those in global window approach. In Figure 5(b), we can see that the mean squared errors are slightly lower in each axis and the correlations are slightly higher in each axis comparing to those in global approach. The change of variance depicted in Figure 5(a) shows many spikes and V-shaped curve segments across the time. This is because the variances are measured from different kernels. For each V-shaped segment, the start and end locations have higher variances compared to those in the middle.

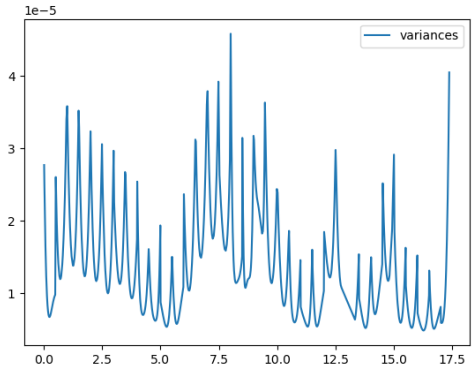
Figure 6 on next page displays the change of kernel parameters over the local windows. In (a), we can see that the multiple kernels lead to the oscillations of $\sigma^2$ between 0.5 and 1.4 over time. In (b), we can observe that while the majority of length scales are below 10, there are a few spikes in between, with the highest one reaching 70. These oscillations and spikes confirm the occurrences of muscle co-contraction during the curve tracing.




(a) results in 3d

(b) results in 2d

Figure 4: Comparison of predicting curve and actual trace in sliding window approach

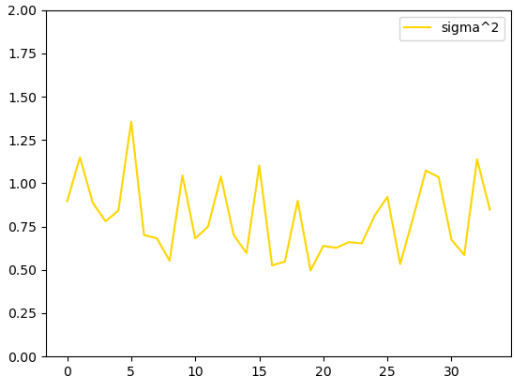



(a) variance over time

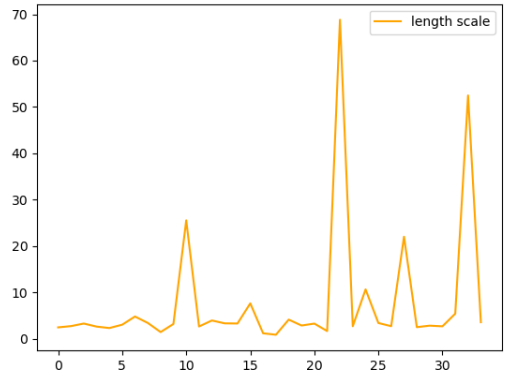
```
MSE in x: 0.011277214413382853
MSE in y: 0.0006852189111647383
MSE in z: 0.003895123209908242
MSE total: 0.01585755653445584
Correlation in x:
 [[1.         0.95623078]
 [0.95623078 1.        ]]
Correlation in y:
 [[1.         0.98430966]
 [0.98430966 1.        ]]
Correlation in z:
 [[1.         0.95319663]
 [0.95319663 1.        ]]
```

(b) MSE and correlations

Figure 5: Performance Measure




(a) $\sigma^2$ over each window




(b) length scale over each window

Figure 6: The change of kernel parameters

# 5 Summary

In general, the prediction results of global window approach and sliding window approach are nearly the same. The mean squared errors are slightly lower and the correlation in each axis are slightly higher in sliding window approach, but the difference is too small that we cannot draw the conclusion that sliding window has significant better performance. However, computationally, it is true that sliding window approach takes much less time than global window approach.

The variance over time follows the same pattern in two approaches: the middle region of the window has lower variance while the regions close to the left and right boundaries get higher. The only difference is that we have only one segment in global window approach while 34 segments in sliding window approach.

For the kernel parameters, $\sigma^2$ and length scale measure the vertical and horizontal variation of the function respectively. The oscillations of kernel parameters through different windows, especially the spikes occurring in the changes of length scale in Figure 6(b), indicate that the muscles are contracting during those specific time intervals. Therefore, it means that a Gaussian Process is able to detect the state of muscle co-contraction as revealed in the kernel parameters used to fit the tracing data.

# 6 References

- `numpy`, `pandas`, `matplotlib` libraries
- `sklearn.metrics` library for evaluation of mean squared errors
- `sklearn.gaussian_process` library, especially `sklearn.gaussian_process.GaussianProcessRegressor` and `sklearn.gaussian_process.kernels.RBF`
- *Machine Learning A Probabilistic Perspective* by Kevin P. Murphy, 2012, Chapter 15.1, 15.2, for mathematical and statistical theories behind