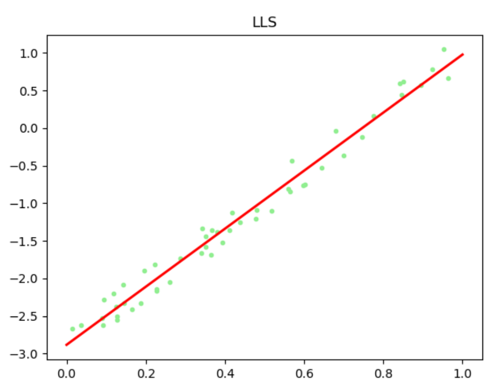
Robotique et Apprentissage : Compte rendu de TME

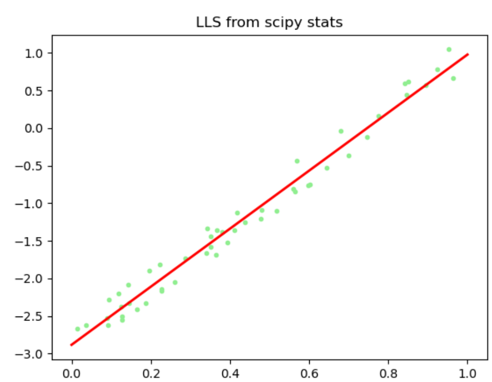
Kadhi Youssef et LIM Vincent

1 Batch Linear Least squares

Question 1

```python
def train(self, x_data, y_data):
    # Finds the Least Square optimal weights
    x_data = np.array([x_data]).transpose()
    y_data = np.array(y_data)
    x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

    self.theta = np.dot(np.dot(np.linalg.inv(np.dot(x.transpose(), x)), x.transpose()), y_data)
    slope, intercept = self.theta

    r_value = np.corrcoef(x_data.transpose(),y_data)[0][1]

    print("slope :", str(slope))
    print("intercept :", str(intercept))
    print("r_value :", str(r_value))

```






```
slope : 3.8621265770103013
intercept : -2.8845353521370325
r_value : 0.9889731365769353
LLS time: about 0.0006
```

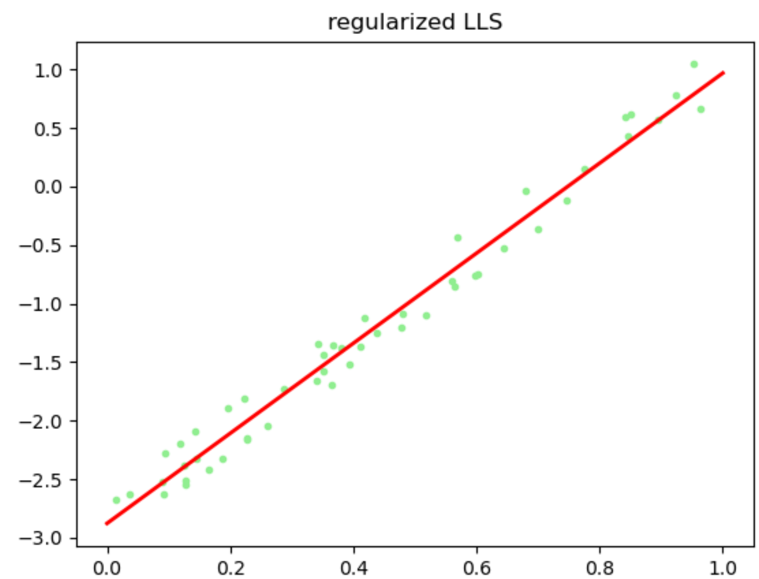
```
slope : 3.8621265770103013
intercept : -2.884535352137033
r_value : 0.988973136576935
LLS from scipy stats: about 0.001
```

We can see that both functions, `train(self, x_data_, y_data)` and `train_from_stats(self, x_data_, y_data)` return the same values for the **slop**, **intercept** and **r_value** variables. However, note that the `train(self ,x_data_, y_data)` function is faster as it requires about 0.0006 ms to finish whereas `train_from_stats(self, x_data_, y_data)` needs about 0.001ms

1.1 Ridge Regression

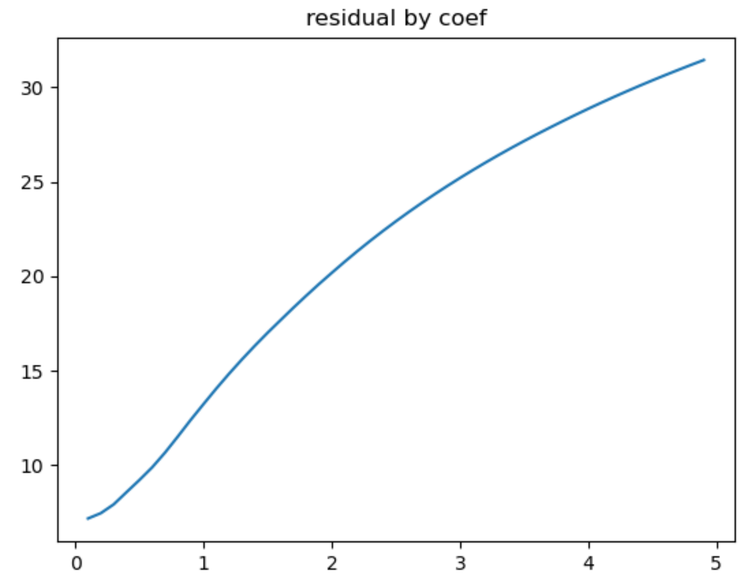
Question 2

```python
def train_regularized(self, x_data, y_data, coef):
    # Finds the regularized Least Square optimal weights
    x_data = np.array([x_data]).transpose()
    y_data = np.array(y_data)
    x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

    self.theta = np.dot(np.dot(np.linalg.inv(coef*np.eye(x.shape[1]) + np.dot(x.transpose(), x)), x.transpose()), y_data)
    slope, intercept = self.theta

    r_value = np.corrcoef(x_data.transpose(),y_data)[0][1]
    print("slope :", str(slope))
    print("intercept :", str(intercept))
    print("r_value :", str(r_value))


```




```
coef : 0.01
slope : 3.8482863165975174
intercept : -2.878082303464065
r_value : 0.9889731365769353
residual 6.9476743382751565
regularized LLS : about 0.015625
```

Question 3



We can see that the residuals increase with the value of the **coef** given to the `train_regularized(self, x_data_, y_data, coef)` function as it should be.
Indeed, with regularized linear least squares, we want to penalize large weights to keep them small and have similar weight with others. These large weights can be due to several reasons such as points being too close to each other or missing points.
With this in mind, the value we are trying to minimize is :

$$\theta^* = arg\min_{\theta} \frac{\lambda}{2}||\theta||^2 + \frac{1}{2}||y - X^T\theta||^2 \text{ (with } \lambda = \text{coef)}$$

In this formula, the first part ($\frac{\lambda}{2}||\theta||^2$) corresponds to the weights and the second part ($\frac{1}{2}||y - X^T\theta||^2$] corresponds to the residual. The higher the λ value is, the more importance we give to the minimisation of the weights against the minimisation of residuals.
This explains why the higher the **coef** value given to the function is, the higher the residual obtained is.

2 Radial Basis Function Networks

2.1 Batch Least Squares

Question 4

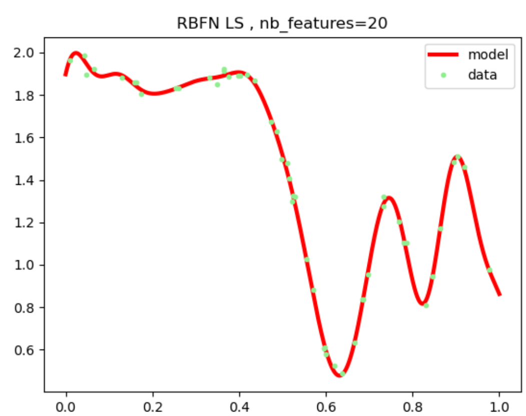
For this question, we have coded two functions `train_ls(self, x_data_, y_data)` and `train_ls_2` (self, x_data_, y_data) corresponding to the 2 approach seen in class.

2.1.1 1st approach

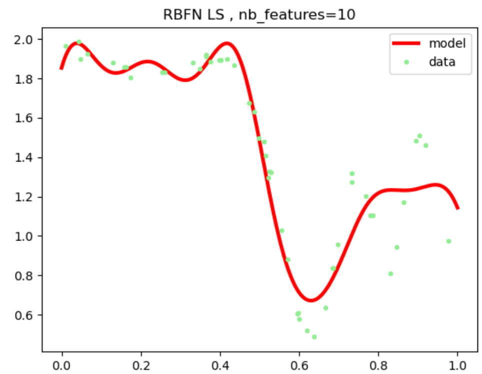
```
def train_ls(self, x_data, y_data):
    x = np.array(x_data)
    y = np.array(y_data)
    X = self.phi_output(x).transpose()

    self.theta = np.dot(np.dot(np.linalg.inv( np.dot(X.transpose(), X) ),
   X.transpose()), y_data)

```

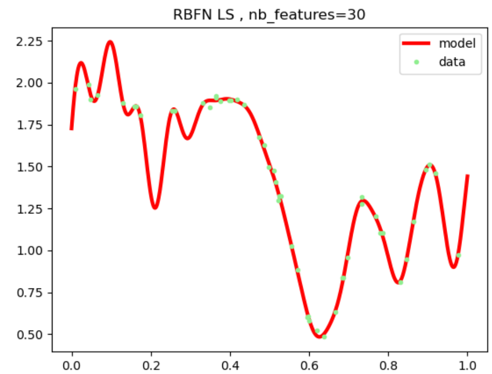
2.1.2 2nd approach

```
def train_ls2(self, x_data, y_data):
    a = np.zeros(shape=(self.nb_features, self.nb_features))
    b = np.zeros(self.nb_features)

    for i in range(len(x_data)):
        b+= np.dot(self.phi_output(x_data[i]), y_data[i])[:,0]
        a+= np.dot(self.phi_output(x_data[i]), self.phi_output(x_data[i]).
   transpose())

    self.theta = np.linalg.solve(a,b)

```



The approximation model obtained seems to be good when the number of features used with Radial Basis Function Networks is not above 20. Indeed, as you can see, from 20 features considered, there is some over fitting in the model obtained. And with the number of features at 10 we can see that the model doesn't fit the data.





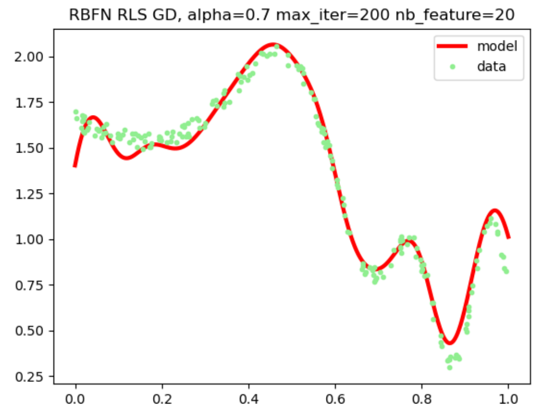
2.2 Gradient Descent

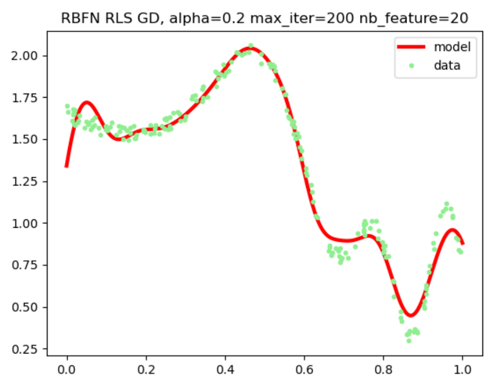
Question 5

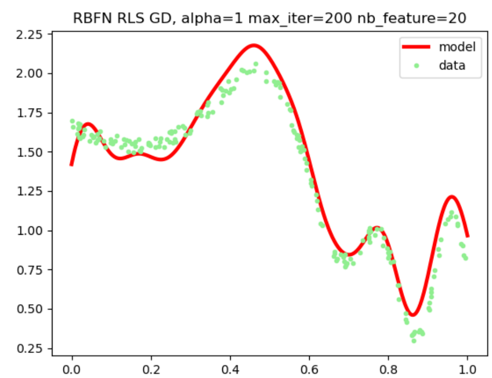
```
def train_gd(self, x, y, alpha):
    phi_x = self.phi_output(x)[:,0]
    self.theta = self.theta  + alpha*np.dot(y-np.dot(phi_x,self.theta),phi_x)

```

We choose the same nbfeature as previously, and we try do find a maxIter to get great results but not to big to reduce computation time. The learning rate shouldn't be too small or too large. Indeed, smaller learning rate will need more iteration as each update will produce a small changes in the weight of each gaussian function. Whereas a learning rate that is to large may cause the model to converge on a solution that is not optimal. After executing the `train_gd(self, x_data_, y_data)` function with different values of **maxIter**, we have found that we can obtain a good approximation model with the following values :

- **maxIter** $= 200$
- **nb_features** $= 20$
- **learning rate** $\alpha = 0.7$

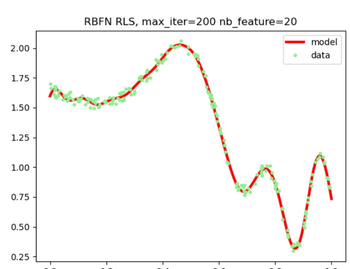


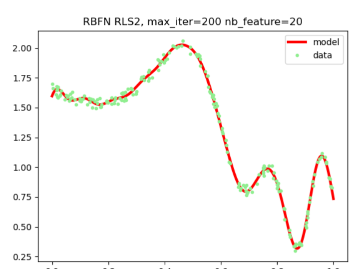


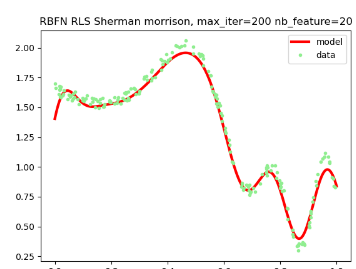


2.3 Recursive Least Squares

Question 6







After executing the different train function with differents values of **maxIter** and **nb_features**, we have found that we can obtain a good approximation model with the following values :

— **maxIter** = 200

— **nb_features** = 20

Question 7

Computation time :

- **train_rls without SM** : 0.05470854599999986
- **train_rl with SM** : 0.02926593699999991
- **train_gd** : 0.007469930000000069

From the images we have seen in previous questions, we can see that the **recursive least squares method** is more accurate without the **Sherman-Morrison** formula, but takes a bit more time to compute. The same can be said between the recursive least squares method with Sherman-Morrison and gradient descent, the latter being faster but less accurate then the former.

Question 8

The batch method execute the training function once, with all data altogether.
Advantages :

- It is computationally faster as we train once on all the data.

Disadvantage :

- A stable error gradient can lead to a local minima.
- Batch method is expensive when the batch is large

The incremental method execute the training function several time. Indeed, a small batch of data is given each time and the model is updated with each batch. The main advantage is that the obtained approximated model is more accurate and the drawback is that it takes more time to approximate a model. **Advantages :**

- It is easier to process since we use a small batch of data everytime instead of everything at once

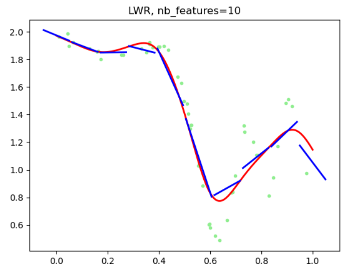
Disadvantage :

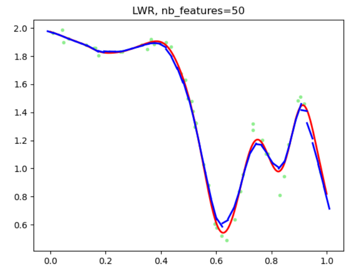
- Since we use the training function several times, the obtained model is more subjected to the noise.
- Using the training function several times require more resources usage.
- Incremental methods converge more slowly to the minima.

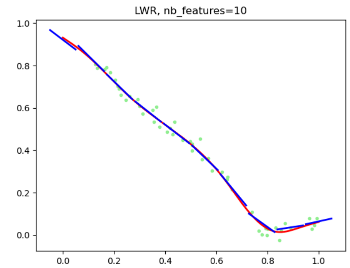
It is better to use incremental methods when there is a lot of data.

3 Locally Weighted Least Squares

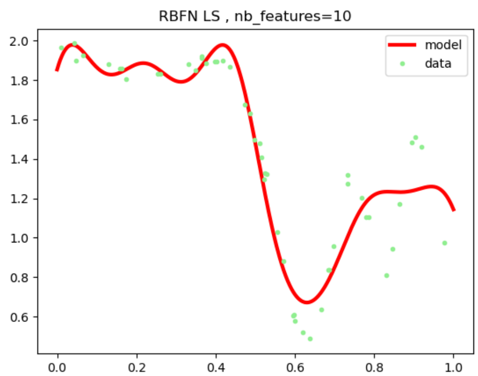
Question 9

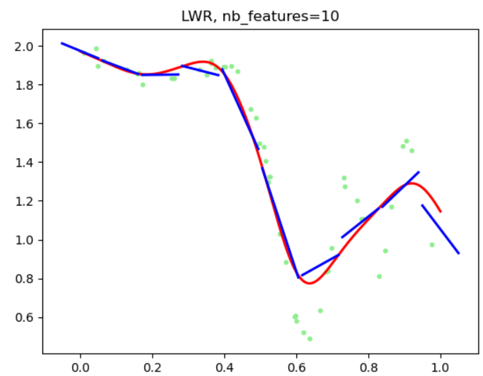


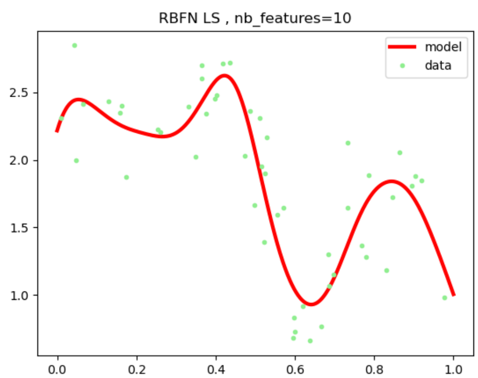


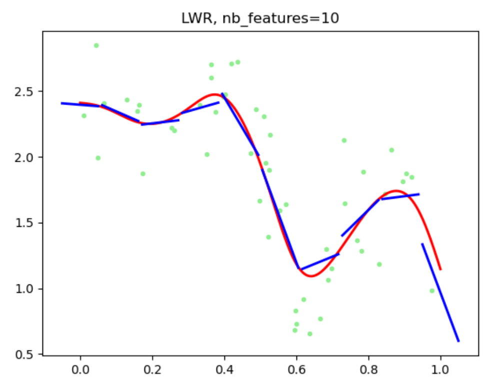


Question 10



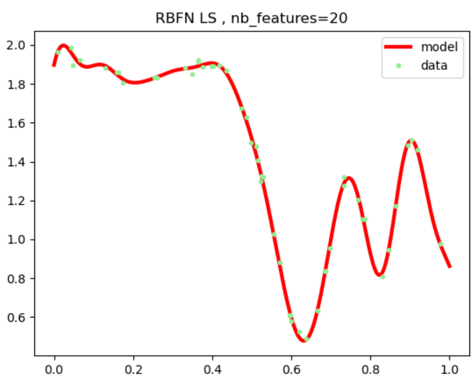


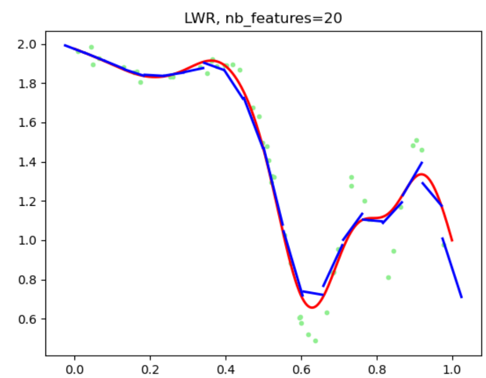


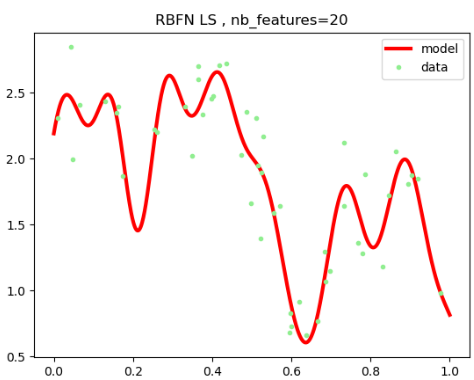


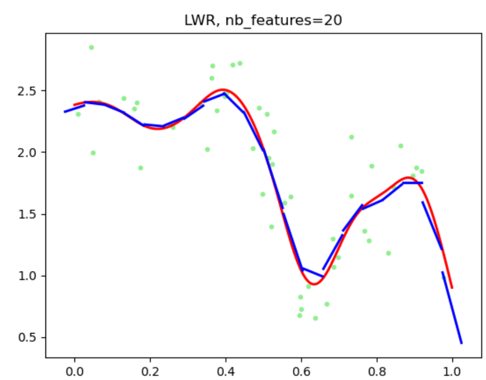
For the 8 figures, the first two have a sigma value of 0.1. The next two have a sigma value of 1. The next two have a sigma value of 0.1. And the last two have a sigma value of 1.
RBFN LS time : 0.0010271490000000005
RBFN LS2 time : 0.004042740000000045
LWR time : 0.043398138000000586

RBFNs functions are faster and are more accurate as seen in the plots.


RBFN LS , nb_features=20
model
data


LWR, nb_features=20


RBFN LS , nb_features=20
model
data


LWR, nb_features=20