RA : TME 1

Axel Foltyn[1] et Petar Calic[2], à Sorbone Université, Campus Pierre et Marie Curie
[1] 3700260, e-mail: `axel.foltyn@gmail.com`
[2] 3971601, e-mail: `petarcalic@gmail.com`

Soumis	11/02/2021
Reçu	....
Validé	....

Regression Labs: Instructions

The objective of regression or function approximation is to create a model from observed data. We have created some data points and stored them in two vectors *x_data* and *y_data*(one dimentional data so we will be able to display it using a 2D graph) We have used 50 data points for the overall project. Hoever we can easily augment the number of data points to have a more acccurate vision of the performance of our learned models. For example we can use 1000 points.

1 The linear least squares method

The simplest method to implement is the method of least squares.
Indeed, this method is only similar to an affine function of the form $f_\theta(X) = \theta^T * X + b$. In order not to have to calculate two values, we make enter b in θ by adding a column of 1 in X.
To find θ we just have to minimize the differences between the affine function and the real results. Moreover, to increase the difference between the theoretical and real results, we raise the difference squared.
The parameters of the linear approximation are learned in the train function and stored in the Theta vector. The first element will be the slope and the second the intercept.

```
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))))

        theta_opt = np.dot(np.dot(np.linalg.inv(np.dot(x.transpose(),x)),np.transpose(x)), y_data)
        self.theta = theta_opt
        print("theta", self.theta)
```

We can compare our train function to the scipy already iplemented function. The theta parameters are the same so we can assume they used the same matrix approach. hoewer its slightly faster so its probably better optimised.

```
def train_from_stats(self, x_data, y_data):
        # Finds the Least Square optimal weights: python provided version
        slope, intercept, r_value, _, _ = stats.linregress(x_data, y_data)
       # print("theta",self.theta)
        #print("slope",slope)
        #print("intercept",intercept)
        self.theta = np.hstack((slope,intercept))
        print("theta2",self.theta)
```

The first method can easily be disrupted by isolated points. To avoid this, a regulation can be added. In our case, the term of regulation will be $\lambda * I$ with λ a coefficient that we will choose and I the identity matrix.
(study q 3) Ridge Regression is a Variation of Linear Regression. LLS doesn't consider which independent variable is more important than others. In ridge regression, you can tune the lambda parameter so that model coefficients change. The term with lambda is often called 'Penalty' since it increases the residus. We will iterate certain values onto the lambda and evaluate the model with a measurement such as the Mean Square Error. Ridge regression performance change with lambda and becomes the same as LLS if lambda is equal to zero (no penalty).

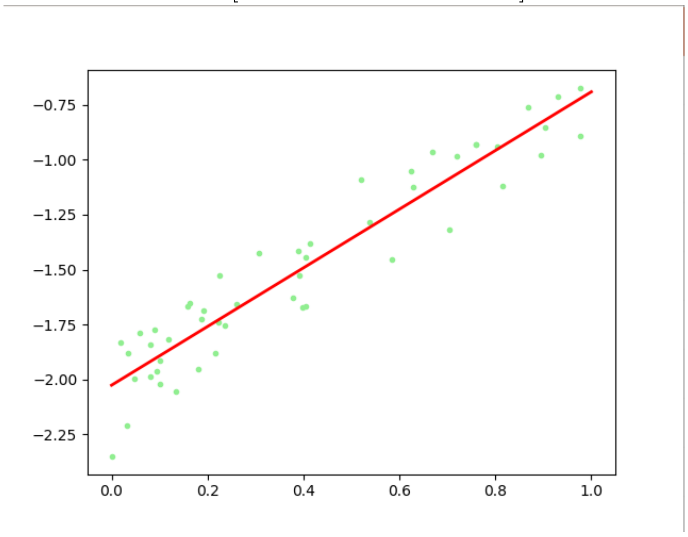
```
def train_regularized(self, x_data, y_data, coef):
        # Finds the regularized Least Square optimal weights
        x_data = np.array([x_data]).transpose()
        y_data = np.array(y_data)
        x = np.hstack((x_data, np.ones((x_data.shape[0], 1))))

        # identite * le coeff
        idc = np.eye((np.dot(x.transpose(),x)).shape[0]) * coef
       # print(idc)

        theta_opt3 = np.dot(np.dot(np.linalg.inv(np.add(np.dot(x.transpose(),x),idc)),np.transpose(x
             )), y_data)
        self.theta = theta_opt3          #update de Theta
        print("theta3", self.theta)
```

For our display of different coefficients, we wanted to be able to put several curves on the same graph, so we modified the plot method like this:

```
def plot(self, x_data, y_data, label=None):
        xs = np.linspace(0.0, 1.0, 1000)
        z = self.f(xs)
        if (label is not None):
            plt.plot(x_data, y_data, 'o', markersize=3, color='lightgreen')
            plt.plot(xs, z, lw=2, label = label)
            plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
        else:
            plt.plot(x_data, y_data, 'o', markersize=3, color='lightgreen')
            plt.plot(xs, z, lw=2, color='red')
            plt.show()
```

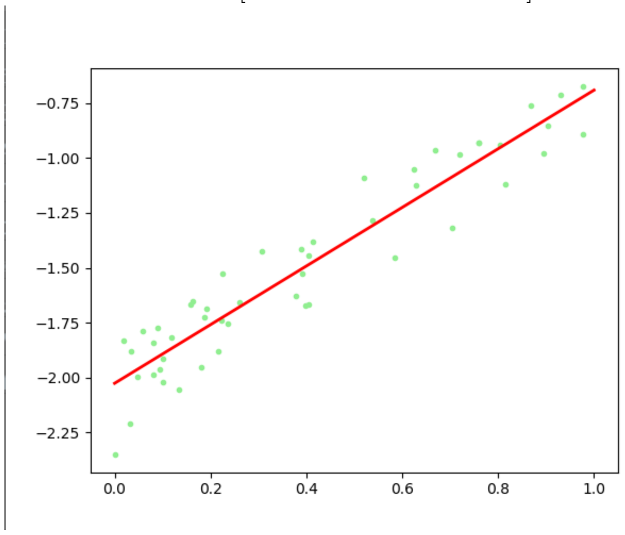
Here are the function tests.

theta [1.33455743 -2.02597371]

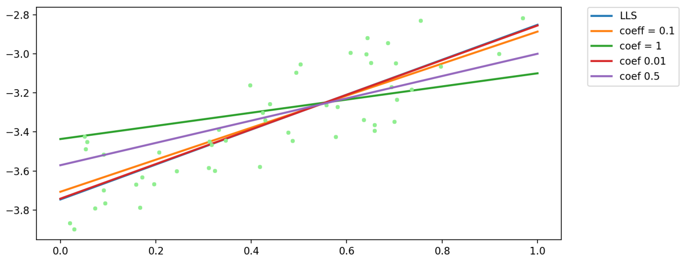


LLS time: 0.0021664809999999424

theta2 [1.33455743 -2.02597371]



LLS from scipy stats: 0.001045998000000048
theta3 [1.28963307 -2.00418647]



regularized LLS (coeff 0.5): 0.0015347860000001212

As we can see, the higher the coefficient, the further the curve moves away from the optimal value. the ideal value of the coefficient is small around 0.1.

2 Radial Basis Function Networks

For the calculation of more complex curves the least squares method is not salifiable. We have therefore switched to the RBFN method. For reasons of time we preferred to go through a Gaussian projection of our set of points. This brings us back to a least square calculation.

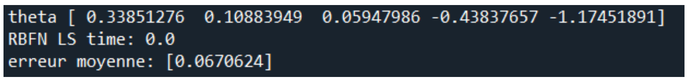
```
def train_ls(self, x_data, y_data):
        x = np.array(x_data)
        y = np.array(y_data)
        X = self.phi_output(x)
        X = X.transpose()
        #id au train du line

        theta_opt = np.dot(np.dot(np.linalg.inv(np.dot(X.transpose(),X)),np.transpose(X)), y)
        self.theta = theta_opt
        print("theta", self.theta)
```

RBFN is used to project from an input space into a feature space, and then we perform the standard linear least square calculation in this projected space. The projection is done using the feature funcitons(here we take take Gaussian function) that takes data points as centers of the gaussians and based on it, adjusts the weights of the function approximation. Here is the trainig code.

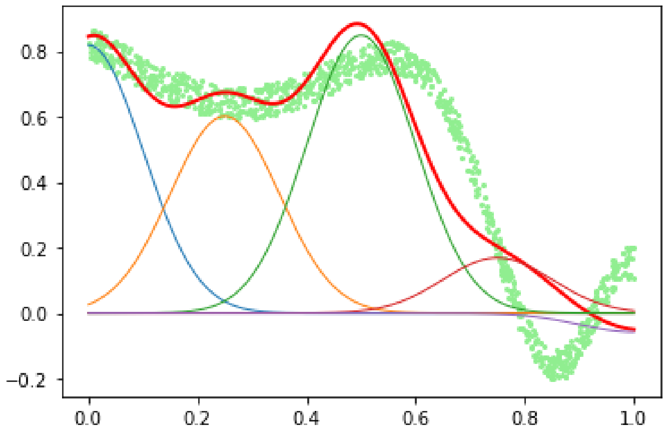
As we can see the complexity of this method is very High: We have to do a inverse of a matrix of size $(numb\_features * numb\_features)$, while the complexity of an inversion is $O(n^3)$. Therefor we will implement a iterative method that will be able to take points one by one and do an approximation which will come in handy if we want to add more points.

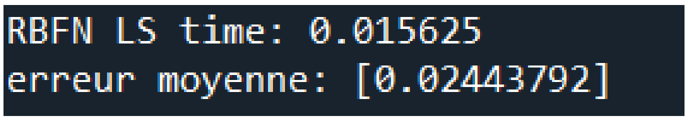
we have added a funciton that calculates des mean error since we augmented the number of data points to 1000 x_data points. This way we will easily be abble to compare algorithnms.

Here is a first tests with the training being done the 1000 data points. The number of features on the first graph is 5 and the second 20.

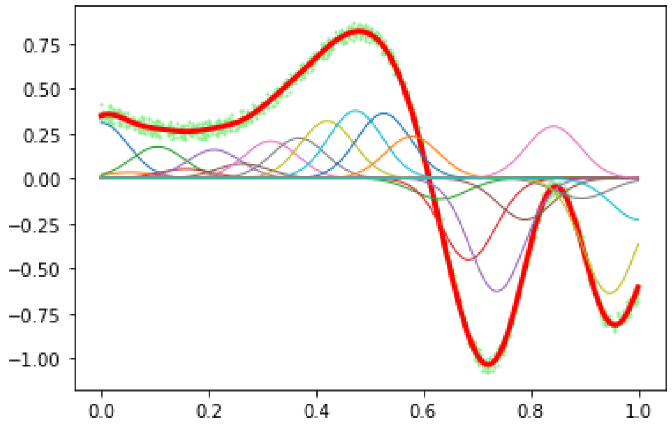


```
theta [ 0.33851276  0.10883949  0.05947986 -0.43837657 -1.17451891]
RBFN LS time: 0.0
erreur moyenne: [0.0670624]
```



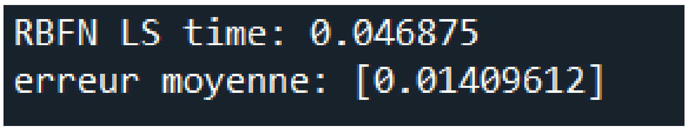


```
RBFN LS time: 0.015625
erreur moyenne: [0.02443792]
```

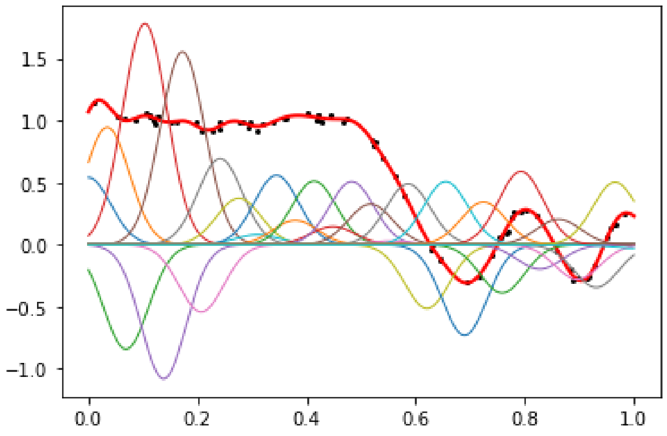


As we can se the approximation is much better with more features. Hoever the calculation takes more time, and will increase faster and faster as the number of features augments for the reasons explained above.

If we increase a lot the number of features we can observe some overfitting. Here is an example with 75 *x_data* so that the phenomenon is more visible.



```
RBFN LS time: 0.046875
erreur moyenne: [0.01409612]
```



The number of features is 30. We can clearly see that the mean distance between the points and the funciton approximation is very small but thats not a good thig in this case. We can see that the function is not approximating but passing throw points. The function is then becoming specialised in this particuliar data set. We could show the bad performance of the algorithm on a smaller test batch.

3 Gradient Descent

We can also, instead of returning to a method of least squares, try to approximate the curve according to our errors at each point.

```
 def train_gd(self, x, y, alpha):
        X = self.phi_output(x)

        X = X.transpose()
        X = X[0]

            #une seule iteration ici
        self.theta = self.theta + np.dot(alpha*(y - np.dot(np.transpose(X) , self.theta)), X)
```

To calculate the mean error of our models we have added a calculation method.

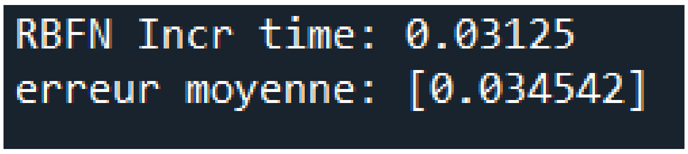
```
def calcerr(self, x_data ,y_data , batchsize):
        sumerr = 0
        for i in range(batchsize):
            sumerr = sumerr + abs(y_data[i] - self.f(x_data[i]))
        return sumerr/ batchsize
```

We have also modified *approx_rbfn_iterative(self)*.

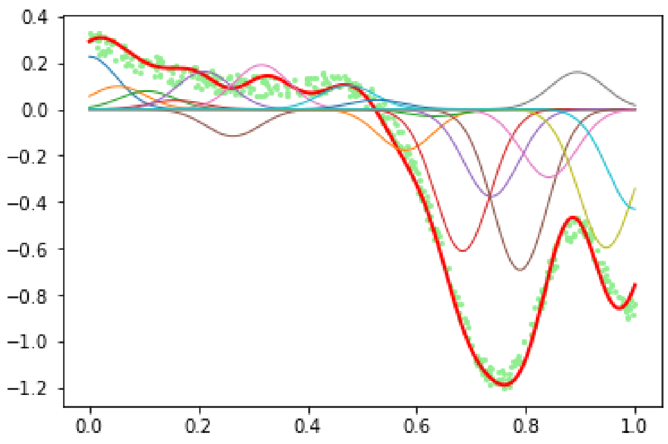
```
def approx_rbfn_iterative_aux(self, max_iter=50, nb_features=10, plot = False):
        model = RBFN(nb_features=nb_features)     #3,5,10..,    40 suraprentissage
        start = time.process_time()
        # Generate a batch of data and store it
        self.reset_batch()
        g = SampleGenerator()

        for i in range(max_iter):
            #print(i+1)
            # Draw a random sample on the interval [0,1]
            x = np.random.random()
            y = g.generate_non_linear_samples(x)
            self.x_data.append(x)
            self.y_data.append(y)

            # Comment the ones you don't want to use
            model.train_gd(x, y, alpha=0.5)
            #model.train_rls(x, y)
            #model.train_rls_sherman_morrison(x, y)

        print("RBFN with max iter=", max_iter,"and nb_features=",nb_features,"Incr time:", time.
             process_time() - start)
        err = model.calcerr(self.x_data, self.y_data, max_iter)
        print("erreur moyenne :", err)
        if plot:
            model.plot(self.x_data, self.y_data)

    def approx_rbfn_iterative(self):
        L_iter = [5,10,15,30,50] #testons plusieurs valeurs 5,10,50   mais pas plus!
        L_feature = [3, 5, 10, 13, 15, 20, 30, 40] #3,5,10..,    40 suraprentissage
        for iter in L_iter:
            for feature in L_feature:
                self.approx_rbfn_iterative_aux(iter, feature)
```

Lets test the gradient descent method with the same parameters as the batch method so we can compare them. We will the set the parameters to $max\_iter = 1000$, features 5 then 20, alpha = 0.3 which seems to give the best results.

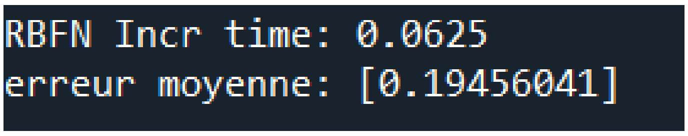
The good thing about this approach is that we do not need to use all the data that is given to as to make a good approximation. In this example we use 400 data points(itermax) and maintain 20 features.



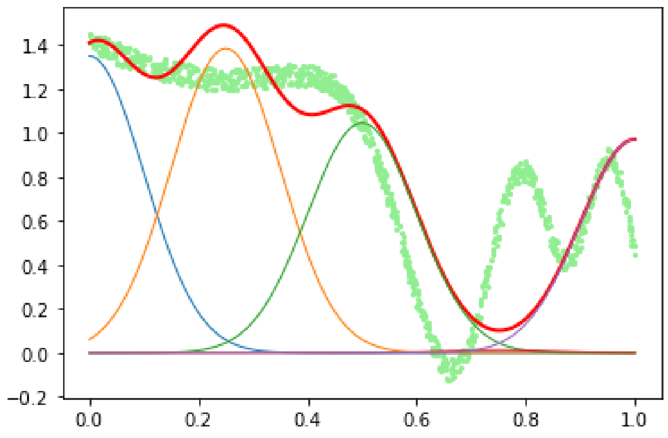
```
RBFN Incr time: 0.03125
erreur moyenne: [0.034542]
```

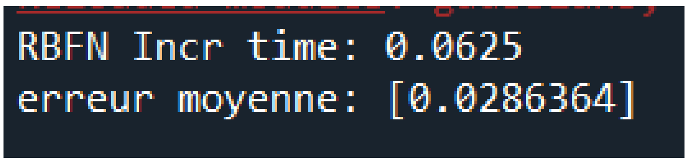


We can see that the speed is divided by two and the precision is getting close to the batch version. The performance of this algorithme will show its potential with a bigger batch of data and bigger number of features. We have tested this idea by increase the size of the batch to 10000. Now the *RBFN_batch* with 60 features returns a good result in 0.15 seconds. The gradient descent approach returns the same precision in half the time with 10 times less data but using 100 features. This is aloved by the iterativ propreties of the algorthm. We can use much more features.

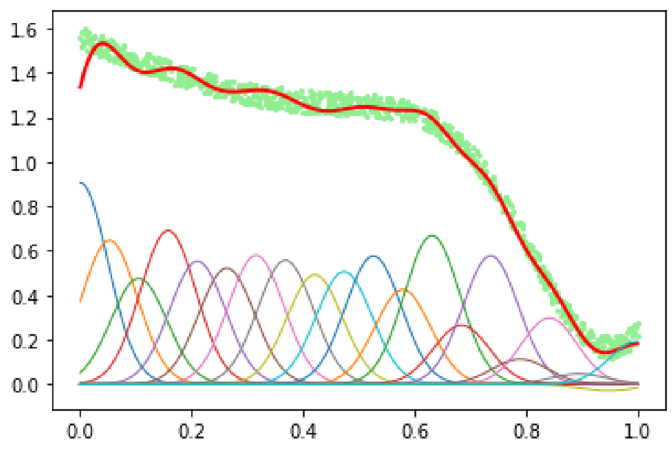


```
RBFN Incr time: 0.0625
erreur moyenne: [0.19456041]
```





```
RBFN Incr time: 0.0625
erreur moyenne: [0.0286364]
```

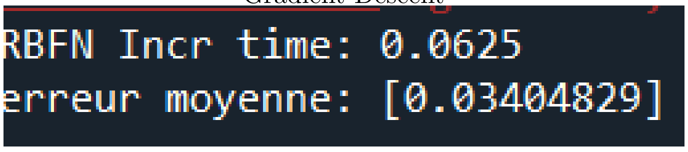


However, not all of these results are sufficient. Indeed, the results should be averaged and also not tested for error with the training base, but with a test base (for example with cross validation).

3.1 Recursive least squares

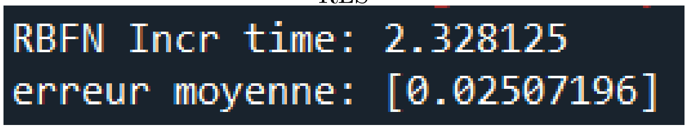
(study q 6 + 7 + 8) Now lets compare the recursive least squares with the gradient descent method. Here are the results for the following parameters: Batch size: 1000, Max_iter: 1000, Features: 40

Gradient Descent



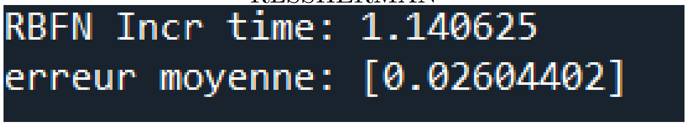
```
RBFN Incr time: 0.0625
erreur moyenne: [0.03404829]
```

RLS



```
RBFN Incr time: 2.328125
erreur moyenne: [0.02507196]
```

RLSSHERMAN



```
RBFN Incr time: 1.140625
erreur moyenne: [0.02604402]
```

The fastest algorithm is the Gradient Descent. The most precise is the RLS. A good compromise of speed and precision would be the Sherman-Morison algorithm. We have run the test 10 times and observed the same phenomen.

(Study question 8)We can conclude from all previous tests about the Radial Basis Function Network algorithms, that the batch method is quick to implement, and has very precise results when we increase the number of features. The problem lies in the size of the features matrix which if too big slows the algorithm very much ($O(n^3)$ complexity). We need a lot of features when the data are too sparse. A big disadvatage also is that the batch size is fixed. if we have more values to add to the batch we will be obliged to calculate again the whole matrixes. This is where the gradient descetn algorithm becomes useful. since its iteratif we can add data one by one and update the learned function.

4 Locally Weighted Least Squares

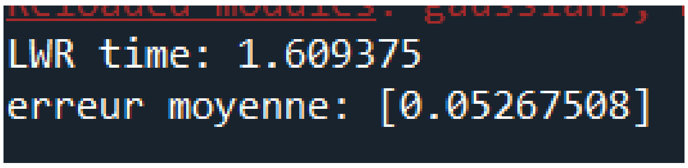
Is the second method to approximate a non linear function. The general idea is that we split the non linear function in parts and the use the LLS on each part and soften the curve by associating it with a Gaussian. This is what LWLS (Locally Weighted Least Squares) does.

```
def train_lwls(self, x_data, y_data) -> None:
        """
        Locally weighted least square function
        This code is specific to the 1D case
        :param x_data: a vector of x values
        :param y_data: a vector of y values
        :return: nothing (set the self.theta vector)
        """

        for k in range(self.nb_features):
            a = np.zeros(shape=(2, 2))
            b = np.zeros((2, 1))

            for i in range(len(x_data)):
                w = np.matrix(bar(x_data[i]))
                phi = self.phi_output(x_data[i])[k]
                ww = np.dot(w, w.transpose())

                a = a + phi[0] * ww
                b = b + y_data[i] * phi[0] * w

            result = np.linalg.pinv(a) * b
            for i in range(2):
                self.theta[i, k] = result[i, 0]
```

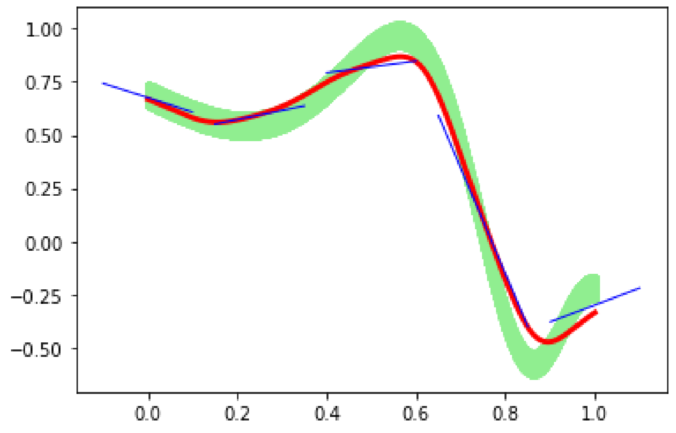
RBFN is faster than LWLS, This is normal because RBFN performs one LS computation in a projected space LWR performs many LS in a local domain. However the advantage of LWLS is that whatever the complexity of the curve we will have a correct efficiency.

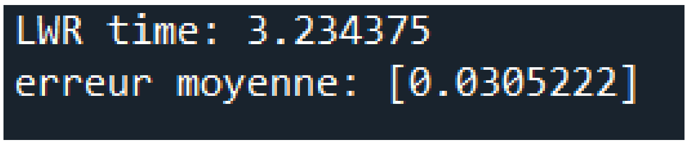
Whe have tested the LWR algorithm with 5 features, and 40 which leaded to good results with resonable calculation time.

(study q 10) The LWR mehtod is without a doubt much slower than the RBFN method. But it seems to make better approximations and reacts better to incresing of number of features. Also we can observe that the LWR algorithm much less vulnerable to overfitting.

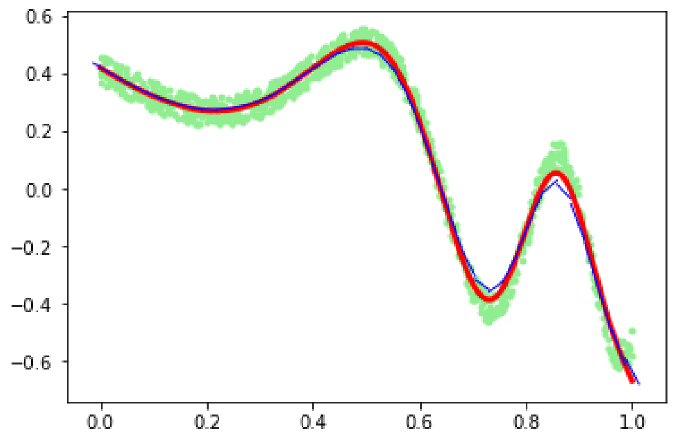


```
LWR time: 1.609375
erreur moyenne: [0.05267508]
```





```
LWR time: 3.234375
erreur moyenne: [0.0305222]
```


0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8
1.0