Lasso Regression:

```
Y = wX + b
Y --> Dependent Variable
X --> Independent Variable
w --> weight
b --> bias
```

Gradient Descent:

Gradient Descent is an optimization algorithm used for minimizing the loss function in various machine learning algorithms. It is used for updating the parameters of the learning model.

w = w - α*dw

b = b - α*db

Learning Rate:

Learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function.

if (w_j > 0) :

$$\frac{dJ}{dw} = \frac{-2}{m}\left[\left[\sum_{i=1}^{m} x_j \cdot \left(y^{(i)} - \hat{y}^{(i)} \right)\right] + \lambda\right]$$

else (w_j ≤ 0) :

$$\frac{dJ}{dw} = \frac{-2}{m}\left[\left[\sum_{i=1}^{m} x_j \cdot \left(y^{(i)} - \hat{y}^{(i)} \right)\right] - \lambda\right]$$

Gradient for Bias

$$\frac{dJ}{db} = \frac{-2}{m}\left[\sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)} \right)\right]$$

```python
import numpy as np


#creating the lasso regression model

class Lasso_regression():

    def __init__(self,learning_rate,no_of_iterations,lambda_parameter):

        self.learning_rate = learning_rate
        self.no_of_iterations= no_of_iterations
        self.lambda_parameter = lambda_parameter

    def fit(self,X,Y):

        # m --> no of rows
        # n --> number of columns(features)  # weights are dependent on the number of imput features for a dataset
        self.m , self.n = X.shape

        self.w = np.zeros(self.n)

        self.b = 0

        self.X = X
        self.Y = Y

        #implementing graduent descent algorithm for optimzation

        for i in range(self.no_of_iterations):

            self.update_weights()
```

```
#function to updatw weight and bias value

def update_weights(self,):
    #linear equation

    Y_prediction = self.predict(self.X)
    #gradients = ( dw , db)
    #gradient for weight

    dw = np.zeros(self.n)

    for i in range(self.n):

        if self.w[i]>0:

            dw[i] = (-(2*(self.X[:,i]).dot(self.Y - Y_prediction)) + self.lambda_parameter) / self.m

        else :

            dw[i] = (-(2*(self.X[:,i]).dot(self.Y - Y_prediction)) - self.lambda_parameter) / self.m

# gradient for bias
    db = - 2 * (self.Y - Y_prediction).sum() / self.m

# updating the weights & bias


    self.w = self.w - self.learning_rate*dw
    self.b = self.b - self.learning_rate*db

def predict(self,X):

    return X.dot(self.w) + self.b
```