optim_tarea07

March 29, 2022

1 Curso de Optimización (DEMAT)

1.1 Tarea 7

Descripción:	Fechas
Fecha de publicación del documento:	**Marzo 19, 2022**
Fecha límite de entrega de la tarea:	**Marzo 27, 2022**

1.1.1 Indicaciones

- Envie el notebook que contenga los códigos y las pruebas realizadas de cada ejercicio.
- Si se requiren algunos scripts adicionales para poder reproducir las pruebas, agreguelos en un ZIP junto con el notebook.
- Genere un PDF del notebook y envielo por separado.

1.2 Ejercicio 1 (5 puntos)

Programar el método de Gauss-Newton para resolver el problema de mínimos cuadrados no lineales

$$\min_x f(z) = \frac{1}{2}\sum_{j=1}^{m} r_j^2(z),$$

donde $r_j : \mathbb{R}^n \to \mathbb{R}$ para $j = 1, ..., m$. Si definimos la función $R : \mathbb{R}^n \to \mathbb{R}^m$ como

$$R(z) = \begin{pmatrix} r_1(z) \\ \vdots \\ r_m(z) \end{pmatrix},$$

entonces

$$min_z f(z) = \frac{1}{2} R(z)^\top R(z).$$

Dar la función de residuales $R(z)$, la función Jacobiana $J(z)$, un punto inicial z_0, un número máximo de iteraciones N, y una tolerancia $\tau > 0$.

1. Hacer $res = 0$.
2. Para $k = 0, 1, ..., N$:

- Calcular $R_k = R(z_k)$
- Calcular $J_k = J(z_k)$
- Calcular la dirección de descenso p_k resolviendo el sistema

$$J_k^\top J_k p_k = -J_k^\top R_k$$

- Si $\|p_k\| < \tau$, hacer $res = 1$ y terminar el ciclo
- Hacer $z_{k+1} = z_k + p_k$.

3. Devolver $z_k, R_k, k, \|p_k\|$ y res.

---

1. Escriba una función que implementa el algoritmo anterior usando arreglos de Numpy.
2. Leer el archivo **puntos2D__1.npy** que contiene una matriz con dos columnas. La primer columna tiene los valores $x_1, x_2, ..., x_m$ y en la segunda columna los valores $y_1, y_2, ..., y_m$, de modo que cada par (x_i, y_i) es un dato. Queremos ajustar al conjunto de puntos (x_i, y_i) el modelo

$$A \sin(wx + \phi)$$

por lo que la función $R(\mathbf{z}) = R(A, w, \phi)$ está formada por los residuales

$$r_i(z) = r_i(A, w, \phi) = A \sin(wx_i + \phi) - y_i$$

para $i = 1, 2, ..., m$.

Programe la función $R(\mathbf{z})$ con $\mathbf{z} = (A, w, \phi)$ y su Jacobiana $J(\mathbf{z})$.

Nota: Puede programar estas funciones de la forma `funcion(z, paramf)`, donde `paramf` corresponda a la matriz que tiene los puntos (x_i, y_i). También puede pasar el arreglo `paramf` como arumento del algoritmo para que pueda evaluar las funciones.

3. Use el algoritmo con estas funciones $R(\mathbf{z})$ y $J(\mathbf{z})$, el punto inicial $\mathbf{z}_0 = (15, 0.6, 0)$ (esto es $A_0 = 15$, $w_0 = 0.6$ y $\phi_0 = 0$), un número máximo de iteraciones $N = 5000$ y una tolerancia $\tau = \sqrt{\epsilon_m}$ donde ϵ_m es el épsilon máquina.

- Imprima el valor inicial $f(\mathbf{z}_0) = \frac{1}{2} R(\mathbf{z}_0)^\top R(\mathbf{z}_0)$.
- Ejecute el algoritmo e imprima un mensaje que indique si el algoritmo converge dependiendo de la variable res.
- Imprima $\mathbf{z}_k$, $f(\mathbf{z}_k) = \frac{1}{2} R(\mathbf{z}_k)^\top R(\mathbf{z}_k)$, la norma $\|p_k\|$, y el número de iteraciones k realizadas.

4. Genere una gráfica que muestre a los puntos (x_i, y_i) y la gráfica del modelo $z_k[0] \sin(z_k[1]x + z_k[2])$, evaluando esta función en el intervalo

$$x \in [\min x_i, \max x_i]$$

5. De la gr'afica de los datos, e interpretando el parámetro A como la amplitud de la onda, se ve que $A_0 = 15$ es una buena inicialización para este paramétro. Para los otros parámetros también debe se debería usar su interpretación para dar buenos valores iniciales. Repita las pruebas con los puntos iniciales $\mathbf{z}_0 = (15, 1, 0)$ y $\mathbf{z}_0 = (15, 0.6, 1.6)$.

1.2.1 Solución:

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

def GaussNewton(R, J, zk, paramf, maxN, tol):
    res = 0
    for k in range(maxN):
        Rk = R(zk, paramf)
        Jk = J(zk, paramf)
        pk = np.linalg.solve( Jk.T @ Jk, -Jk.T @ Rk )

        if np.linalg.norm(pk) < tol:
            res = 1
            break
        zk = zk + pk
        #print(zk)
    return zk, Rk, k, np.linalg.norm(pk), res
```

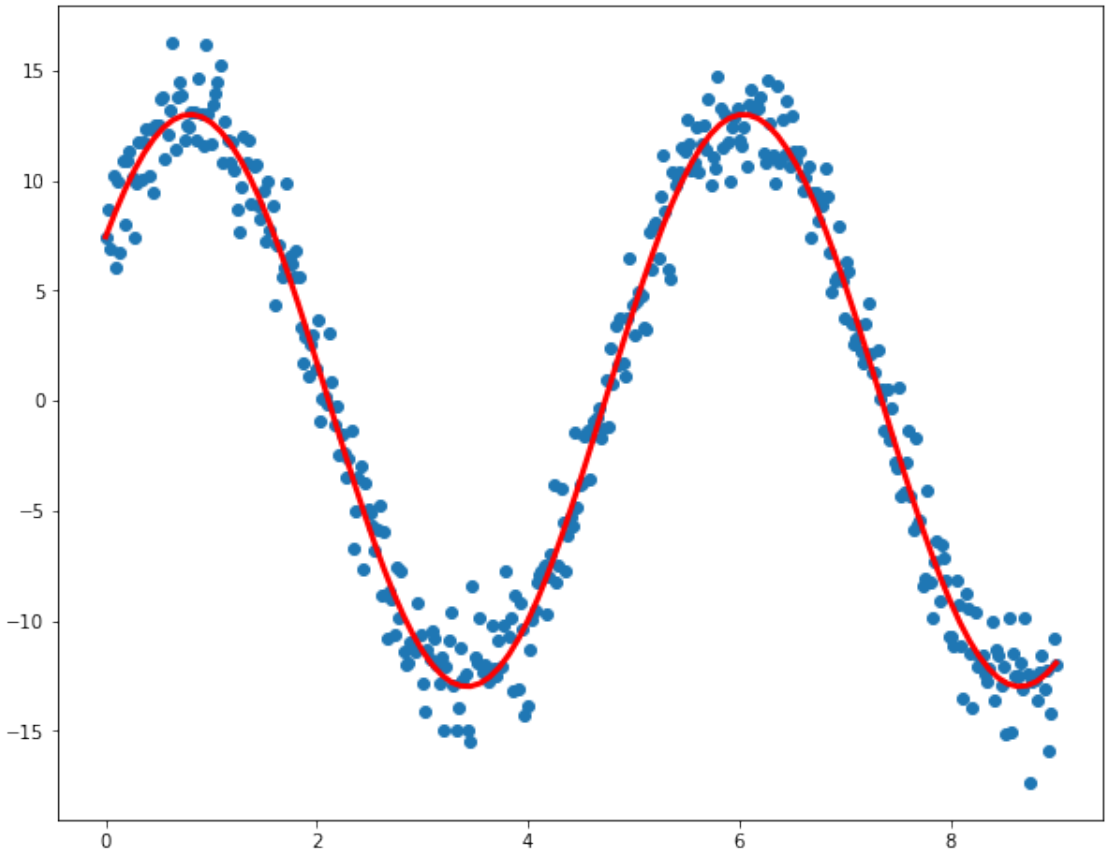
[2]:
```
def residuales(z, paramf):
    # ri(z) = A sin(wx_i + phi) - yi
    A, w, phi = z
    x, y = paramf[:, 0], paramf[:, 1]
    n, _ = paramf.shape

    R = A * np.sin(w * x + phi) - y
    return R

def jacobiana(z, paramf):
    A, w, phi = z
    x = paramf[:, 0]
    n, _ = paramf.shape

    J = np.zeros((n, 3))
    s = np.sin(w * x  + phi)
    c = np.cos(w * x  + phi)

    # Parcial con respecto a A
    J[:, 0] = s
    # Parcial con respecto a w
    J[:, 1] = A * c * x
```

```
    # Parcial con respecto a phi
    J[:, 2] = A * c
    return J

def test(residuales, jacobiana, z0, pts, maxN, tol):
    print('='*40, '\n\n')
    zk, Rk, k, pk_norm, res = GaussNewton(residuales, jacobiana, z0, pts, maxN,␣
 ↪tol)

    R0 = residuales(z0, pts)

    print(f'Valor inicial z0: {z0}')
    print(f'f(z0): {R0.T @ R0 / 2}')

    if res == 0:
        print('El algoritmo NO convergio')
    else:
        print('El algoritno SI convergio')

    print(f'zk = {zk}')
    print(f'f(zk) = {Rk.T @ Rk / 2}')
    print(f'||pk|| = {pk_norm}')
    print(f'Número de iteraciones = {k}')

    x, y = pts[:, 0], pts[:, 1]
    x_min = np.min(x)
    x_max = np.max(x)

    X = np.linspace(x_min, x_max, 100).T
    Y = np.zeros((100,1))
    Y = residuales(zk, np.column_stack((X, Y)))

    plt.figure(figsize = (10,8))
    plt.scatter(x,y)
    plt.plot(X, Y, color='r', linewidth=3)
    plt.show()

# Datos
pts = np.load('puntos2D_1.npy')

# Test 1
Z0 = np.array([15, 0.6, 0.0])
MAX_N = 5000
EPS_M = np.finfo(float).eps
TOL = EPS_M ** (1/2)
test(residuales, jacobiana, Z0, pts, MAX_N, TOL)
```

```python
# Test 2
Z0 = np.array([15, 1.0, 0.0])
test(residuales, jacobiana, Z0, pts, MAX_N, TOL)

# Test 3
Z0 = np.array([15, 0.6, 1.6])
test(residuales, jacobiana, Z0, pts, MAX_N, TOL)
```

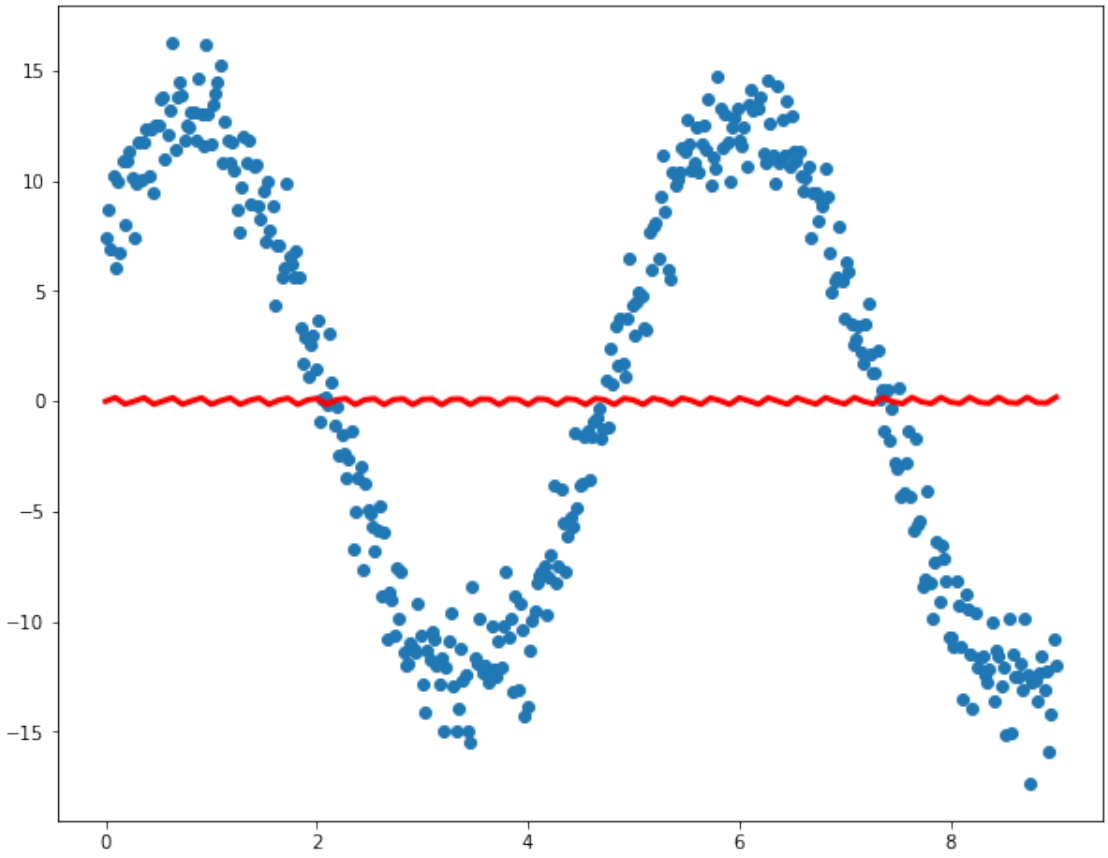
==========

```
Valor inicial z0: [15.   0.6  0. ]
f(z0): 45454.05280978729
El algoritno SI convergio
zk = [12.99606648  1.19935917 -5.67317097]
f(zk) = 457.1693612130722
||pk|| = 1.4545189687689 75e-08
Número de iteraciones = 8
```



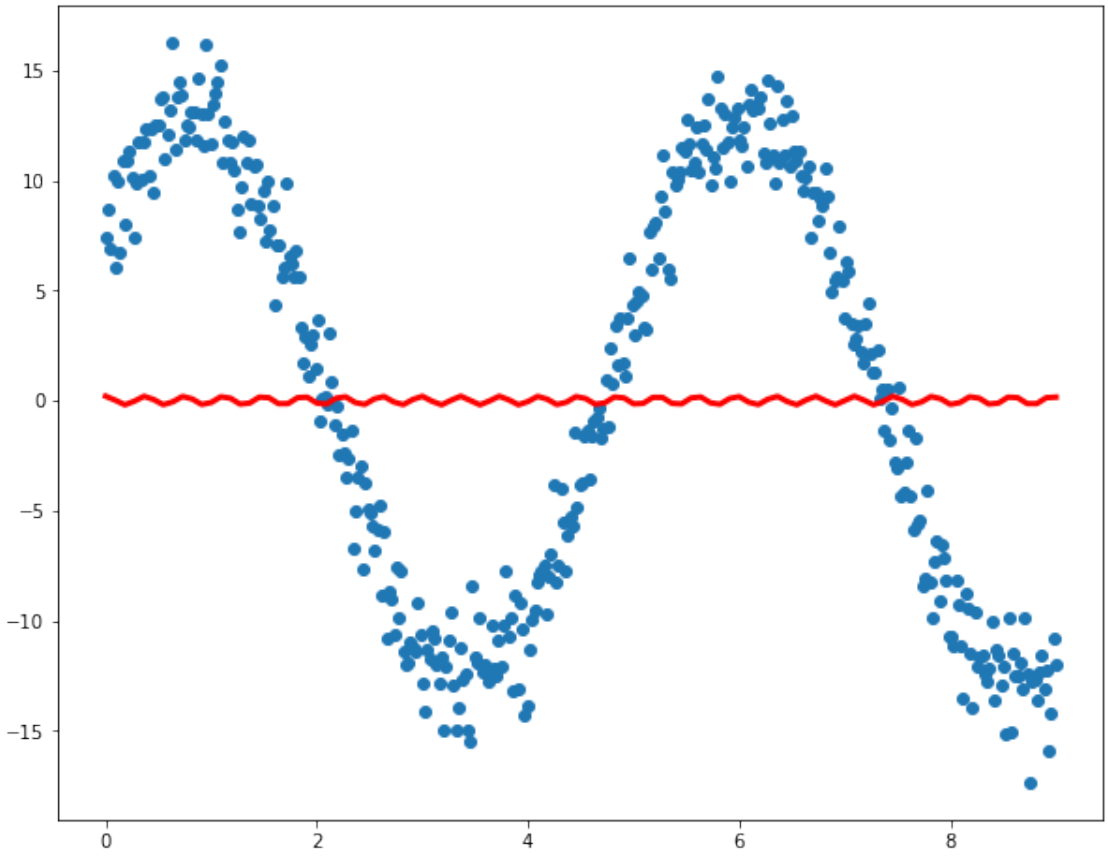
==========

```
Valor inicial z0: [15.  1.  0.]
f(z0): 40807.16289819636
El algoritmo NO convergio
zk = [ -0.1700118   23.21783424 -78.63302966]
f(zk) = 18652.968054878926
||pk|| = 2.032077004816657
Número de iteraciones = 4999
```



==========

```
Valor inicial z0: [15.   0.6  1.6]
f(z0): 37048.62007346928
El algoritno SI convergio
zk = [  -0.19679603   52.28163675 -183.75409605]
f(zk) = 18651.98377600729
||pk|| = 1.153143876973915e-08
Número de iteraciones = 54
```



1.3 _

1.4 Ejercicio 2 (5 puntos)

Programar el método de Levenberg-Marquart para mínimos cuadrados.

Dar la función de residuales $R(z)$, la función Jacobiana $J(z)$, un punto inicial z_0, un número máximo de iteraciones N, $\mu_{ref} > 0$ y la tolerancia $\tau > 0$.

1. Hacer $res = 0$ y construir la matriz identidad I de tamaño igual a la dimensión de z_0.
2. Calcular $R_0 = R(z_0)$
3. Calcular $J_0 = J(z_0)$
4. Calcular $f_0 = 0.5 R_0^\top R_0$
5. Calcular $\mathbf{A} = J_0^\top J_0$ y $\mathbf{g} = J_0^\top R_0$

6. Calcular $\mu = \min\{\mu_{ref}, \max a_{ii}\}$, donde a_{ii} son los elementos de la diagonal de la matriz $\mathbf{A}$.
7. Para $k = 0, 1, ..., N$:

- Calcular $\mathbf{p}_k$ resolviendo el sistema

$$(\mathbf{A} + \mu\mathbf{I})\mathbf{p}_k = -\mathbf{g}$$

- Si $\|\mathbf{p}_k\| < \tau$, hacer $res = 1$ y terminar el ciclo.
- Calcular $\mathbf{z}_{k+1} = \mathbf{z}_k + \mathbf{p}_k$
- Calcular $\mathbf{R}_{k+1} = \mathbf{R}(\mathbf{z}_{k+1})$
- Calcular $f_{k+1} = 0.5\mathbf{R}_{k+1}^\top \mathbf{R}_{k+1}$
- Calcular el parámetro ρ (ver las notas de la clase 16)

$$\rho = (f_k - f_{k+1})/(q_k(\mathbf{x}_k) - q_k(\mathbf{x}_{k+1})) = (f_k - f_{k+1})/(-\mathbf{p}_k^\top \mathbf{g} + 0.5\mu_k \mathbf{p}_k^\top \mathbf{p}_k)$$

- Si $\rho < 0.25$, hacer $\mu = 2\mu$.
- Si $\rho > 0.75$, hacer $\mu = \mu/3$.
- Calcular $\mathbf{J}_{k+1} = \mathbf{J}(\mathbf{z}_{k+1})$
- Calcular $\mathbf{A} = \mathbf{J}_{k+1}^\top \mathbf{J}_{k+1}$ y $\mathbf{g} = \mathbf{J}_{k+1}^\top \mathbf{R}_{k+1}$.

8. Devolver el punto $\mathbf{z}_k$, f_k, k y res.

---

1. Escriba una función que implementa el algoritmo anterior usando arreglos de Numpy.
2. Aplique este algoritmo para resolver el problema del Ejercicio 1, imprimiendo la misma información y generando la gráfica correspondiente, usando $\tau = \sqrt{\epsilon_m}, N = 5000, \mu_{ref} = 0.001$ y los tres puntos iniciales

$$\mathbf{z}_0 = (15, 0.6, 0)$$
$$\mathbf{z}_0 = (15, 1.0, 0)$$
$$\mathbf{z}_0 = (15, 0.6, 1.6)$$

1.4.1 Solución:

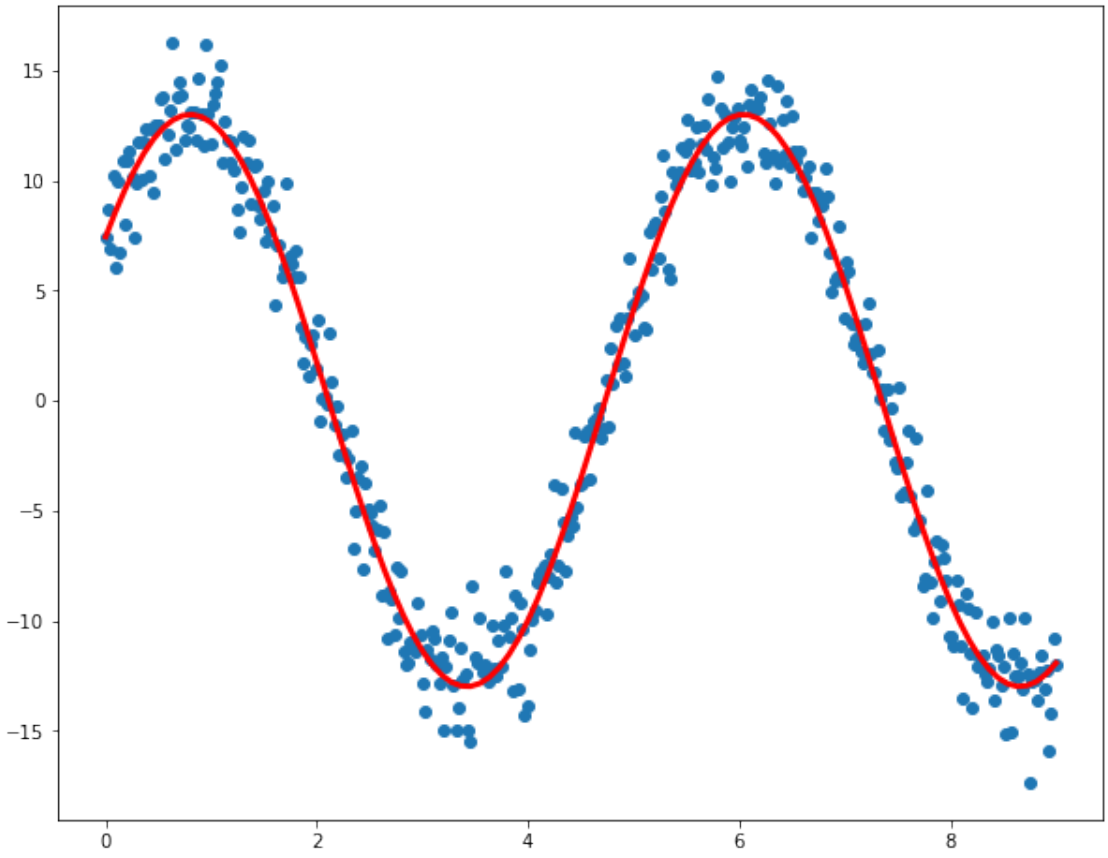
[3]:
```
def LevenbergMarquart(R, J, zk, paramf, mu_ref, maxN, tol):
    res = 0
    Rk = R(zk, paramf)
    Jk = J(zk, paramf)
    fk = Rk.T @ Rk / 2
    A = Jk.T @ Jk
    g = Jk.T @ Rk
    mu = min(mu_ref, np.amax(np.diag(A)))

    for k in range(maxN):
        pk = np.linalg.solve( A + mu * np.identity(A.shape[0]), -g )
```

```
        if np.linalg.norm(pk) < tol:
            res = 1
            break

        zk = zk + pk
        Rk = R(zk, paramf)
        f_an = fk
        fk = Rk.T @ Rk / 2

        rho = (f_an - fk) / (-pk.T@g + 0.5*mu * pk.T @ pk)

        if rho < 0.25:
            mu = 2 * mu
        elif rho > 0.75:
            mu = mu/3

        Jk = J(zk, paramf)
        A = Jk.T @ Jk
        g = Jk.T @ Rk
    return zk, fk, k, res
```

[4]:
```
def test(residuales, jacobiana, z0, pts, mu_ref, maxN, tol):
    print('='*40, '\n\n')
    zk, fk, k, res = LevenbergMarquart(residuales, jacobiana, z0, pts, mu_ref, ␣
 ↪maxN, tol)

    R0 = residuales(z0, pts)

    print(f'Valor inicial z0: {z0}')
    print(f'f(z0): {R0.T @ R0 / 2}')

    if res == 0:
        print('El algoritmo NO convergio')
        return
    print('El algoritno SI convergio')

    print(f'zk = {zk}')
    print(f'fk = {fk}')
    print(f'Número de iteraciones = {k}')

    x, y = pts[:, 0], pts[:, 1]
    x_min = np.min(x)
    x_max = np.max(x)

    X = np.linspace(x_min, x_max, 100).T
    Y = np.zeros((100,1))
    Y = residuales(zk, np.column_stack((X, Y)))
```

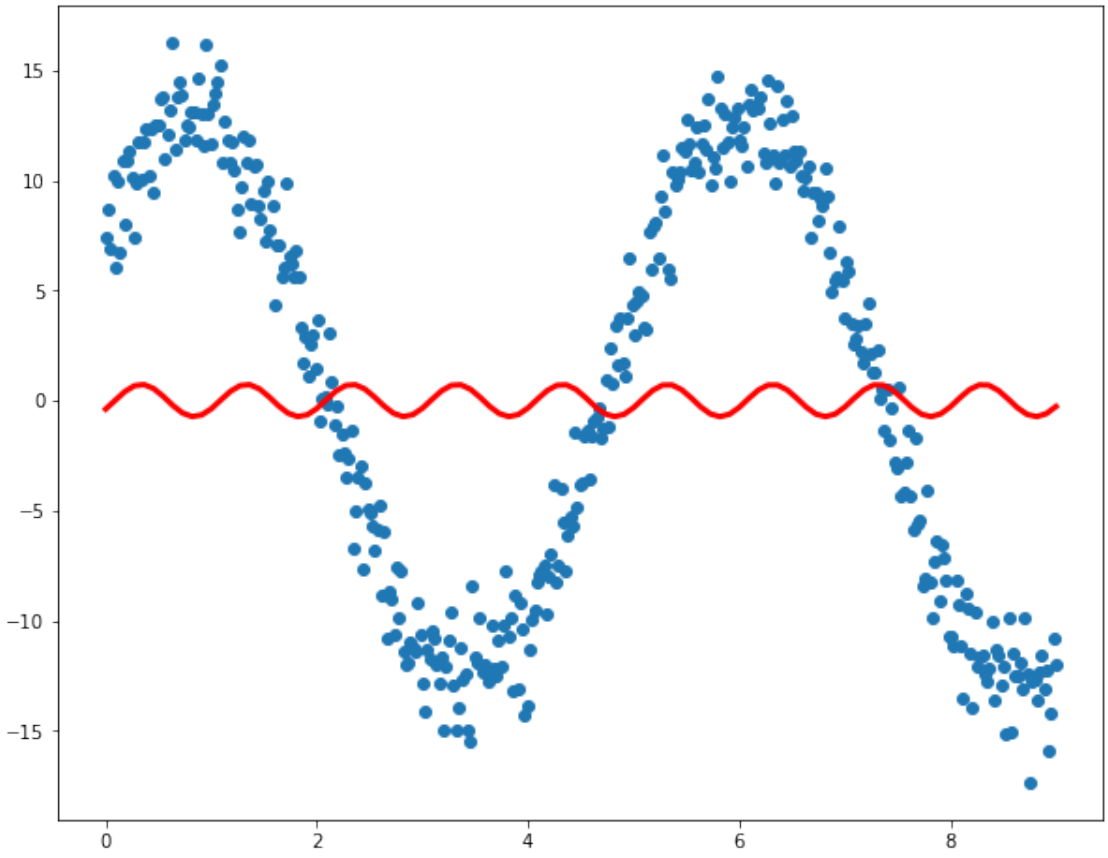
```
    plt.figure(figsize = (10,8))
    plt.scatter(x,y)
    plt.plot(X, Y, color='r', linewidth=3)
    plt.show()

# Datos
pts = np.load('puntos2D_1.npy')

# Test 1
Z0 = np.array([15, 0.6, 0.0])
MU_REF = 0.001
MAX_N = 5000
EPS_M = np.finfo(float).eps
TOL = np.sqrt(EPS_M)
test(residuales, jacobiana, Z0, pts, MU_REF, MAX_N, TOL)

# Test 2
Z0 = np.array([15, 1.0, 0.0])
MU_REF = 0.001
MAX_N = 5000
EPS_M = np.finfo(float).eps
TOL = np.sqrt(EPS_M)
test(residuales, jacobiana, Z0, pts, MU_REF, MAX_N, TOL)

# Test 3
Z0 = np.array([15, 0.6, 1.6])
MU_REF = 0.001
MAX_N = 5000
EPS_M = np.finfo(float).eps
TOL = np.sqrt(EPS_M)
test(residuales, jacobiana, Z0, pts, MU_REF, MAX_N, TOL)
```

```
========================================


Valor inicial z0: [15.   0.6  0. ]
f(z0): 45454.05280978729
El algoritno SI convergio
zk = [12.99606648  1.19935917 -5.67317097]
fk = 457.1693612130723
Número de iteraciones = 8
```



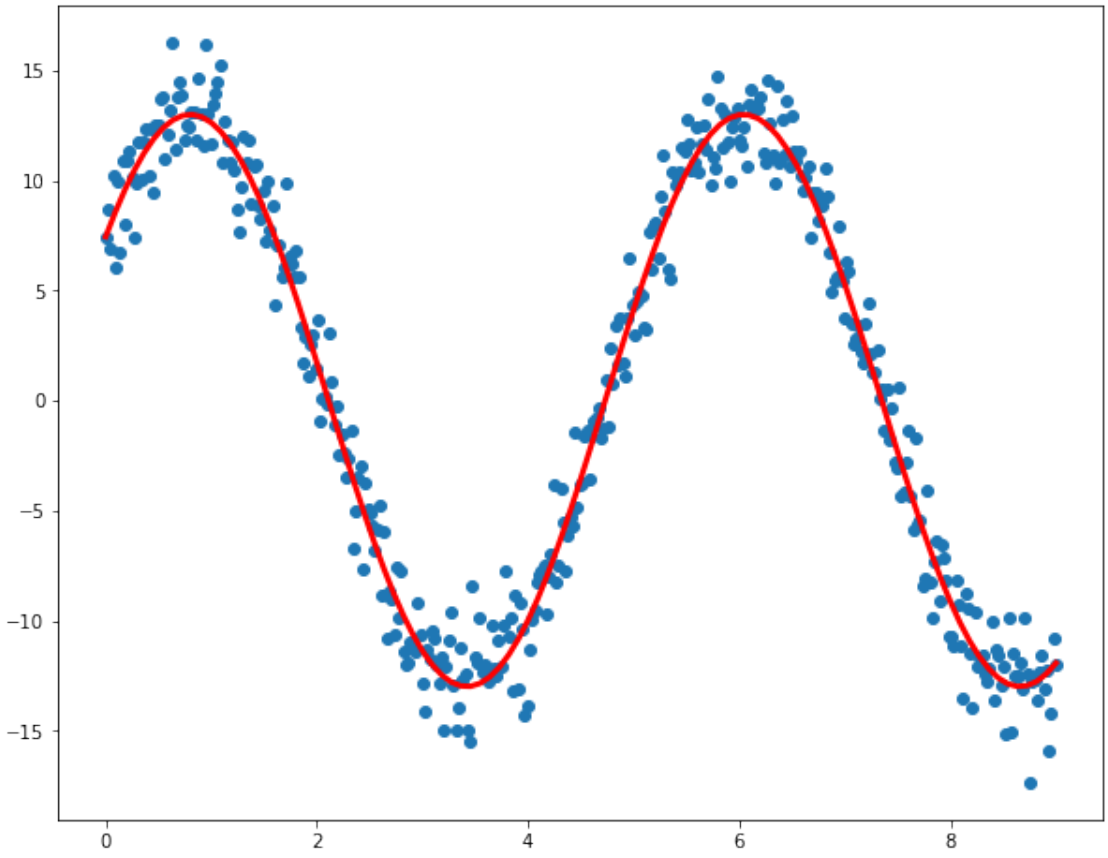
==========

```
Valor inicial z0: [15.  1.  0.]
f(z0): 40807.16289819636
El algoritno SI convergio
zk = [-0.73301615  6.30307377 -3.70002099]
fk = 18602.30622080397
Número de iteraciones = 410
```



==========

```
Valor inicial z0: [15.   0.6  1.6]
f(z0): 37048.62007346928
El algoritno SI convergio
zk = [-12.99606648  -1.19935917   5.67317095]
fk = 457.1693612130712
Número de iteraciones = 17
```


15
10
5
0
−5
−10
−15
0
2
4
6
8

—

[ ]: