Problema 1

Implementar el algoritmo de *Gram-Schmidt* modificado 8.1 del Trefet-hen (p. 58) para generar la descomposición *QR*.

A continuación un extracto de la implementación en python.

Algorithm 1: Modified Gram-Schmidt (Trefethen pag. 58). De **factorization.py** .

```python
for idx in range(0, n):
    # Normalize ortogonal vector
    R[idx, idx] = np.sqrt(Q[:, idx].dot(Q[:, idx]))
    Q[:, idx] /= R[idx, idx]

    # Substract projection to q_i ortonormal vector
    for jdx in range(idx + 1, n):
        R[idx, jdx] = Q[:, idx].dot(Q[:, jdx])
        Q[:, jdx] -= R[idx, jdx] * Q[:, idx]
```

Problema 2

Implementar el algoritmo que calcula el estimador de mínimos cuadrados en una regresión usando la descomposición *QR*.

Para ajustar un polinomio de grado $p-1$ con una regresión consideramos un sistema de ecuaciones sobredeterminado, con $n > p$, de la forma :

$$\mathbf{X}\boldsymbol{\beta} = \mathbf{y} \tag{1}$$

donde $\mathbf{X}$ es la matrix de Vandermonde :

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{p-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{p-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{p-1} \end{bmatrix}, \qquad \boldsymbol{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{2}$$

Como el sistema es sobredeterminado, no existe una solución exacta. Así que formulamos un problema de *Mínimos Cuadrados* que consiste en encontrar una solución aproximada resolviendo el siguiente problema de minimización :

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}} \left\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\right\|^2 \tag{3}$$

La solución al problema de minimización se calcula de manera analítica derivando la expresión anterior respecto a β . De modo que la solución es, como también vimos en clase,

$$(\mathbf{X}^{\mathrm{T}}\mathbf{X})\hat{\boldsymbol{\beta}} = \mathbf{X}^{\mathrm{T}}\mathbf{y} \tag{4}$$

Encontrando la factorización **QR** de la matriz **X** podemos sustituir en la solución anterior. Obtenemos que, usando que la matriz **Q** es ortonormal,

$$((\mathbf{QR})^{\mathrm{T}}(\mathbf{QR}))\hat{\boldsymbol{\beta}} = (\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}}\mathbf{QR})\hat{\boldsymbol{\beta}} = (\mathbf{R}^{\mathrm{T}}\mathbf{R})\hat{\boldsymbol{\beta}} = (\mathbf{R}^{\mathrm{T}}\mathbf{Q}^{\mathrm{T}})\mathbf{y} \tag{5}$$

Y suponiendo que la matriz $\mathbf{R}^{\mathrm{T}}$ tiene inversa, obtenemos el sistema lineal ,

$$\mathbf{R}\hat{\boldsymbol{\beta}} = \mathbf{Q}^{\mathrm{T}}\mathbf{y} \tag{6}$$

que se puede resolver con el algoritmo de **Backward Substitution** puesto que **R** es una matriz triangular superior. De modo que podemos implementar la solución en python de la siguiente manera :

Algorithm 2: Estimador de mímos cuadrados. De **least_squares.py** .

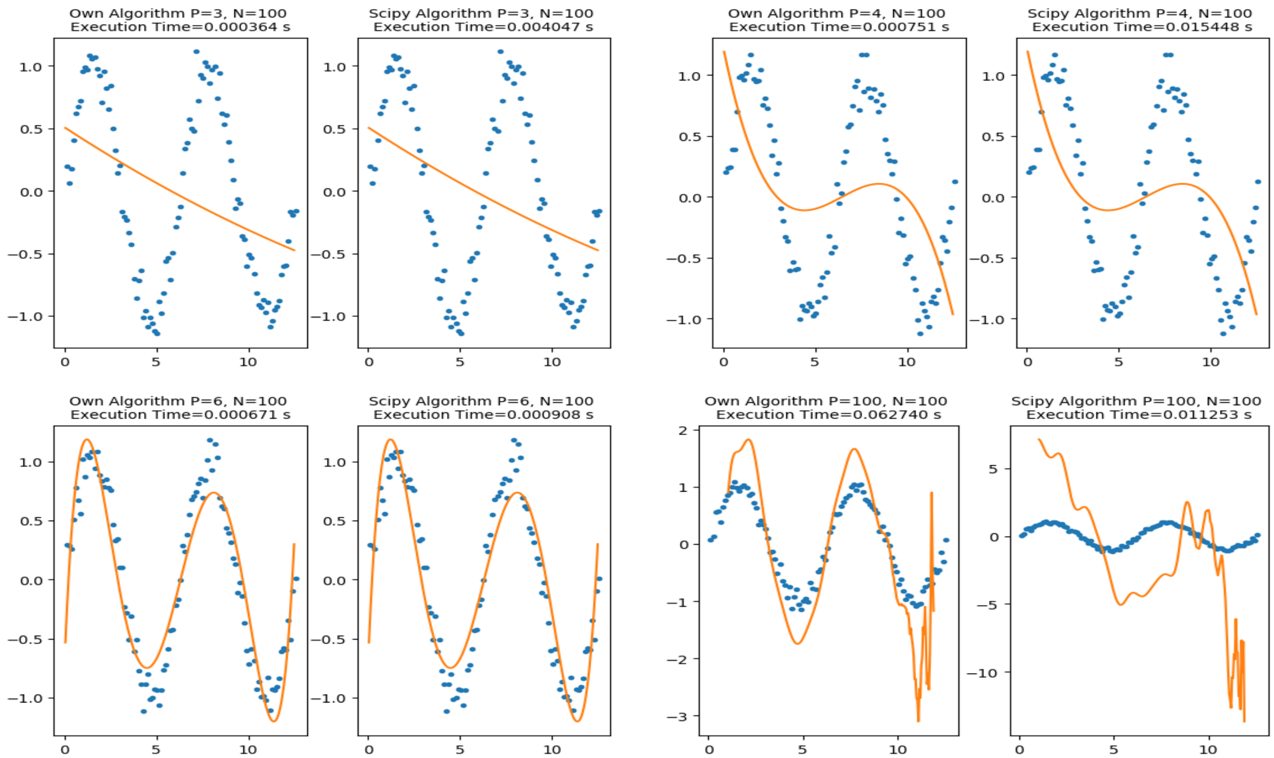
```python
def least_squares_estimator(X, Y):
    # Get QR decomposition of data matrix
    (Q, R) = qr_factorization(X)

    # Transform y vector
    Y_prime = Q.T @ Y.T

    # Solve system R * beta = y_prime
    beta = backward_substitution(R, Y_prime.T)

    return beta
```

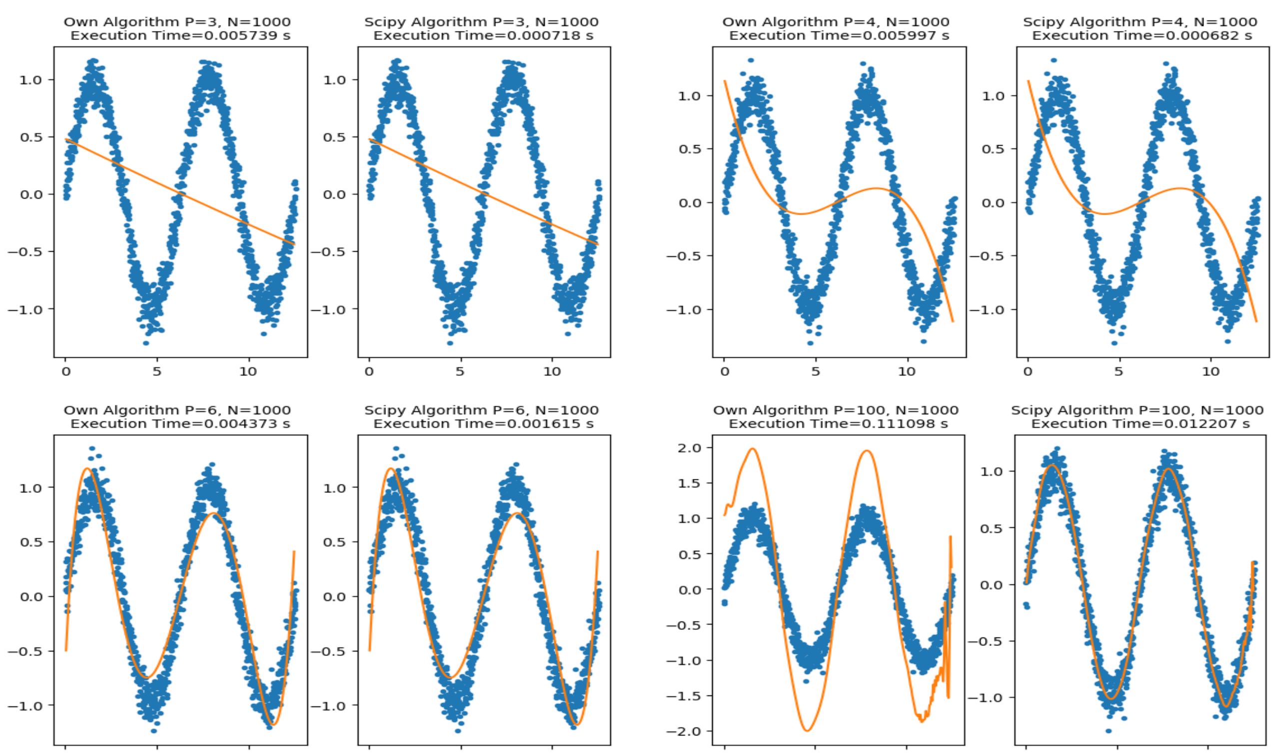
Problema 3

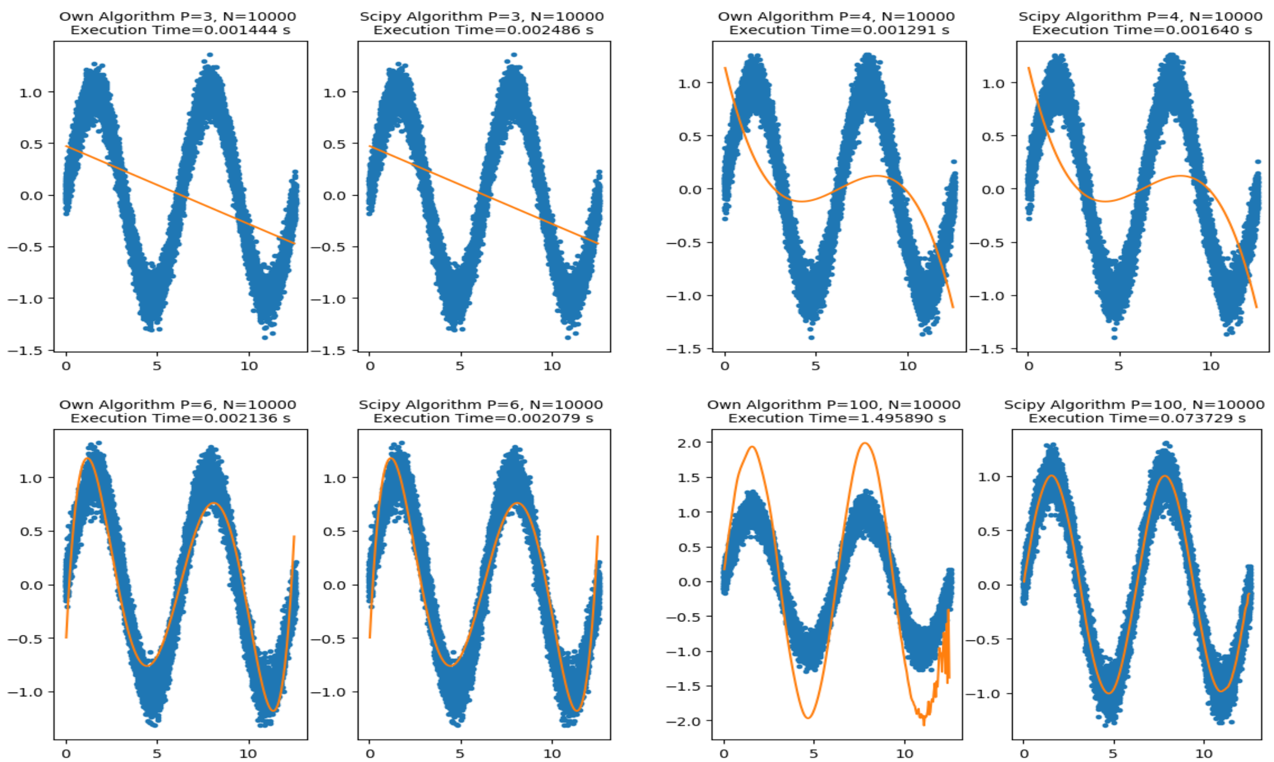
Generar Y compuesto de $y_i = sen(x_i) + \epsilon_i$ donde ϵ_i $N(0, \sigma)$ con $sigma = 0{,}11$ para $x_i = \frac{4\pi i}{n}$ para $i = 1, ..., n$. Hacer un ajuste de mínimos cuadrados a Y, con descomposición QR, ajustando un polinomio de grado $p - 1$.

Consideramos los 12 casos : p = 3,4,6,100 y n = 100,1000,10000, graficamos el ajuste de un polinomio dado por nuestro algoritmo y lo comparamos con el ajuste obtenido al obtener la factorización **QR** con la librería **scipy**. A continuación vemos las gráficas de dichos ajustes.

Nota : Es interesante comparar el ajuste obtenido entre nuestro algoritmo implementando el proceso de Gram-Schmidt modificado y la descomposición **QR** obtenida con la libreria **scipy**. Es claro notar la diferencia en el ajuste en los casos $P = 100, N = 1000$ y $P = 100, N = 10000$.


Own Algorithm P=3, N=100
Execution Time=0.000364 s
Scipy Algorithm P=3, N=100
Execution Time=0.004047 s
Own Algorithm P=4, N=100
Execution Time=0.000751 s
Scipy Algorithm P=4, N=100
Execution Time=0.015448 s
Own Algorithm P=6, N=100
Execution Time=0.000671 s
Scipy Algorithm P=6, N=100
Execution Time=0.000908 s
Own Algorithm P=100, N=100
Execution Time=0.062740 s
Scipy Algorithm P=100, N=100
Execution Time=0.011253 s


Own Algorithm P=3, N=1000
Execution Time=0.005739 s
Scipy Algorithm P=3, N=1000
Execution Time=0.000718 s
Own Algorithm P=4, N=1000
Execution Time=0.005997 s
Scipy Algorithm P=4, N=1000
Execution Time=0.000682 s
Own Algorithm P=6, N=1000
Execution Time=0.004373 s
Scipy Algorithm P=6, N=1000
Execution Time=0.001615 s
Own Algorithm P=100, N=1000
Execution Time=0.111098 s
Scipy Algorithm P=100, N=1000
Execution Time=0.012207 s



Esta diferencia surge ya que el proceso de *Gram-Schmidt Modificado* sufre de algunos problemas de estabilidad (mucho que menos que el proceso clasico) y en la librería **scipy** se implementa una versión optimizada del algoritmo que utiliza reflecciones de Householder [1] que es numericamente un poco más estable [2].

Otra cosa que se debe mencionar es el caso $P = 100, N = 100$ que claramente presenta un ajuste muy malo a los datos generados. Esto se debe a que al considerar la solución de mínimos cuadrados esperamos que trabajar con un sistema sobredeterminado, cuando tenemos muchos más datos (el número de ecuaciones, N) que el grado del polinomio a ajustar (el número de incognitas, P). Procuramos graficar el polinomio en el rango $(1, 12)$ para evitar que en el eje de graficación y el polinomio se salga de un rango considerable e.g $(-3, 3)$ y podamos ver un poco más a detalle el ajuste.

En la figura 1 graficamos los ajustes polinomiales con el mismo tamaño de muestra para diferentes grados del polinomio.

En la figura 2 gráficamos los tiempos de ejecución (medido con **timeit**) de todo el algoritmo (no solo la factorización QR) que ajusta un polinomio de grado $p - 1$ usando mínimos cuadrados. Hacemos la comparación entre le algoritmo que usa nuestra implementación de la factorización QR con el que utiliza el modulo de **scipy** para calcularla.

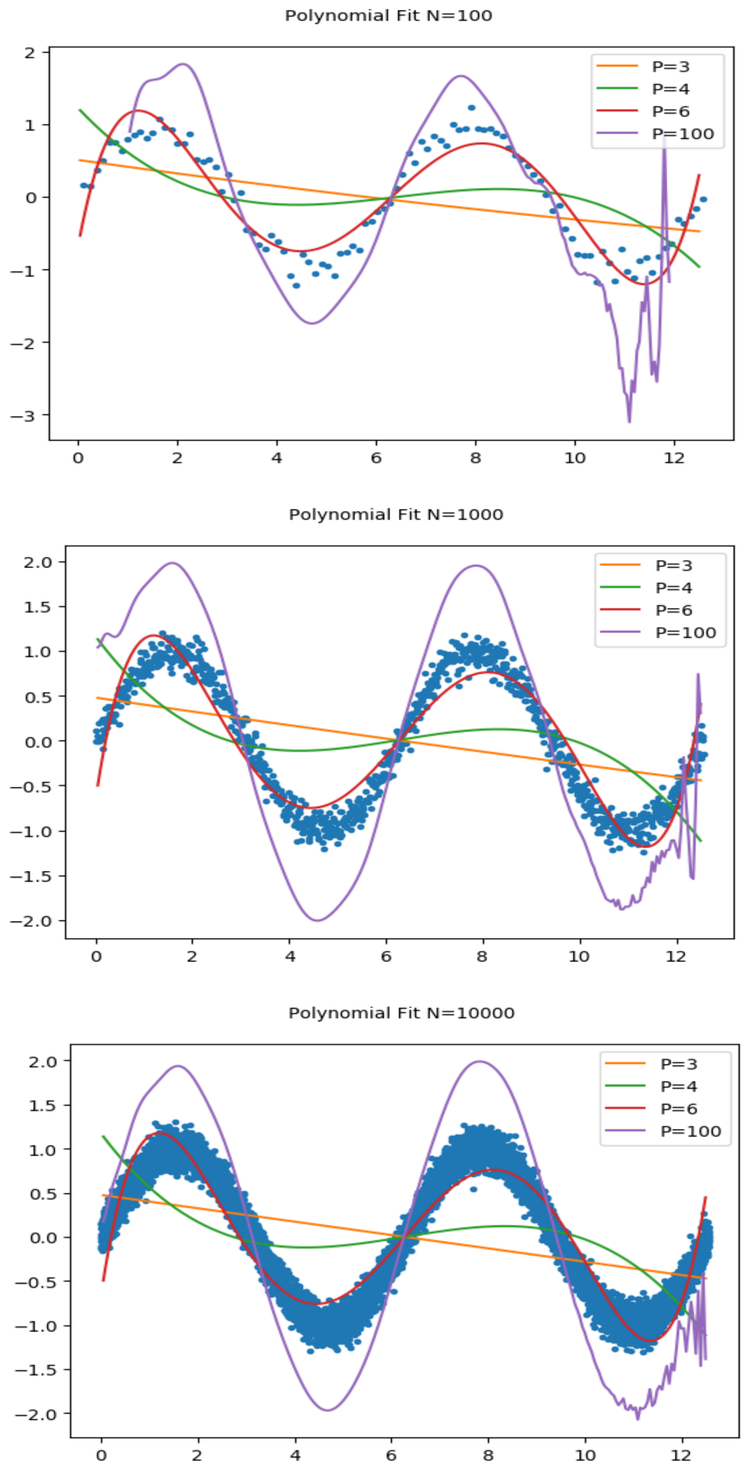

Polynomial Fit N=100
P=3
P=4
P=6
P=100
Polynomial Fit N=1000
P=3
P=4
P=6
P=100
Polynomial Fit N=10000
P=3
P=4
P=6
P=100

Figura 1:

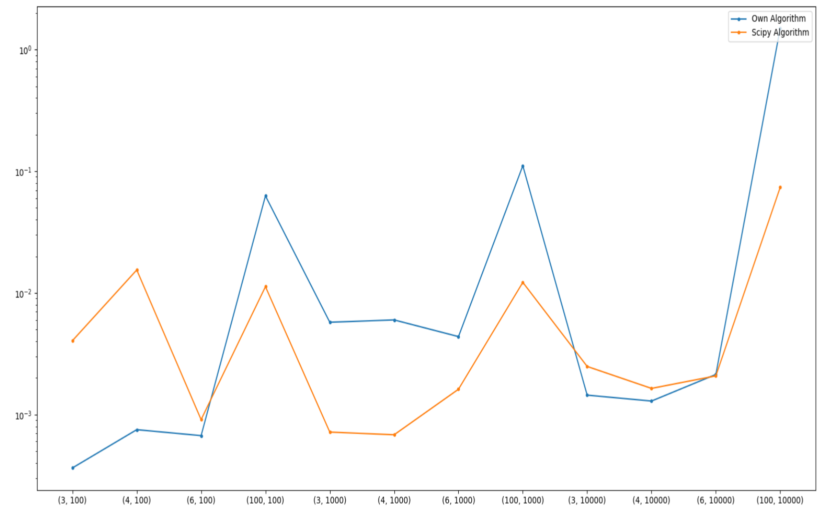


Figura 2: Comparación de los tiempos de ejecucińánea (en escala logaritmica) de los 12 casos de la forma (P, N).

Problema 4

Hacer p = 0.1n, o sea, diez veces más observaciones que coeficientes en la regresión, ¿Cuál es la máxima que puede manejar su computadora?

El máximo valor que la computadora puede manejar es $N = 2810$, osea que $P = 281$. Esto se debe a que al calcular la matriz de Vandermonde ocurre un overflow en la representación en punto de flotante cuando tenemos que calcular los elementos de las últimas columnas e.g. $x_1^{p-1}, x_2^{p-1}, x_3^{p-1}, ..., x_n^{p-1}$, pues se vuelven productos entre valores muy pequeños o muy grandes.

El archivo de python **compare.py** contiene la implementación de todas las comparaciones y gráficas de esta tarea. Ejecutandolo podemos verficar que es posible manejar el caso $N = 2810$ y ver su ajuste. En el primer caso que se encontró un overflow fue cuando $N = 2820$, mencionado como warning de python.

En la figura 3 vemos el tiempo de ejecución de los ajustes con $\mathbf{p = 0.1n}$.

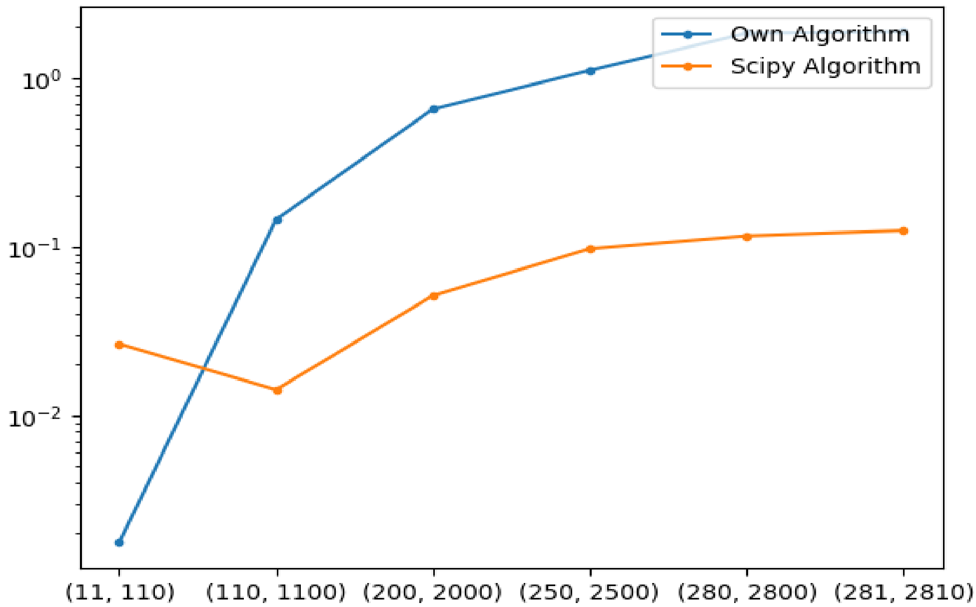


Figura 3: Comparación de los tiempos de ejecución de casos de la forma (P, N).

Referencias

[1] scipy.linalg.qr, numpy.linalg.qr, Scipy Documentation
`https://docs.scipy.org/doc/scipy-0.16.1/reference/generated/scipy.linalg.qr.html`
`https://docs.scipy.org/doc/numpy-1.12.0/reference/generated/numpy.linalg.qr.html`

[2] QR decomposition, Wikipedia
`https://en.wikipedia.org/wiki/QR_decomposition`