Cómputo Científico para Probabilidad, Estadística y Ciencia de Datos

Ezau Faridh Torres Torres

TAREA 2: Descomposición QR y mínimos cuadrados.

Fecha de entrega: 11/Sep/2024.

NOTA: Los ejercicios se encuentran repartidos en los archivos:

- ejercicio1.py
- ejercicio2.py
- ejercicio3.py
- ejercicio4.py

---

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

En el archivo ejercicio1.py se implementa el algoritmo de Gram-Schmidt modificado para calcular la factorización QR de una matriz A de $m \times n$ en la función *MODIFIED_GRAM_SCHMIDT()*.

Acepta como argumentos a la matriz A a factorizar y un booleano que indica si se sobreescribe la matriz A (*defatul* = *False*). Como salidas tiene a las matrices Q y R tales que $A = QR$ con Q es una matriz ortogonal de tamaño $m \times n$ y R es una matriz triangular superior de tamaño $n \times n$. Además, se incluye un par de ejemplos de uso de la función.

---

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

Dada una regresión lineal:

$$y = X\beta + \epsilon \tag{1}$$

donde y es el vector de valores observados, X es la matriz de diseño (de tamaño $m \times n$), β es el vector de coeficientes de regresión y ϵ es el vector de errores. El objetivo es estimar β que minimiza la función de pérdida de mínimos cuadrados, que es:

$$\min_{\beta} \|y - X\beta\|^2 . \tag{2}$$

Con esto, se usa la descomposición QR de la matriz X, donde $X = QR$, siendo:

- Q es una matriz ortogonal de tamaño $m \times n$.
- R es una matriz triangular superior de tamaño $n \times n$.

$\implies y = QR\beta + \epsilon$ y como $Q^TQ = I_{m\times m} \implies Q^Ty = Q^TQR\beta = R\beta$ y se obtiene que

$$R\beta = Q^Ty. \tag{3}$$

Como R es triangular superior, podemos usar *backward substitution* de la Tarea 1. En el archivo ejercicio2.py se usa la función *MODIFIED_GRAM_SCHMIDT()* del ejercicio 1 para crear la función *LEAST_SQUARES_QR()*, la cual calcula la factorización QR de una matriz X para después resolver el sistema (3) con ayuda de *backward substitution*.

3. Generar $\mathbf{Y}$ compuesto de $y_i = \sin(x_i) + \epsilon_i$ donde $\epsilon_i \sim N(0, \sigma)$ con $\sigma = 0.11$ para $x_i = \frac{4\pi i}{n}$ para $i = 1, \ldots, n$. Hacer un ajuste de mínimos cuadrados a $\mathbf{Y}$, con descomposición QR, ajustando un polinomio de grado $p - 1$.
   - Considerar los 12 casos: $p = 2, 4, 6, 100$ y $n = 100, 1000, 10000$.
   - Graficar el ajuste en cada caso.
   - Medir tiempo de ejecución de su algoritmo, comparar con descomposición QR de scipy y graficar los resultados.

Se tiene la expresión

$$y_i = \sin(x_i) + \epsilon_i \tag{4}$$

con $\epsilon_i \sim N(0, \sigma)$, $\sigma = 0.11$ y $x_i = \frac{4\pi i}{n}$ para $i = 1, \ldots, n$. Se requiere ajustar un polinomio de grado $p - 1$ a estos datos, es decir, se busca una relación del tipo:

$$y_i \approx \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_{p-1} x_i^{p-1}. \tag{5}$$

para $i = 1, \ldots, n$. Se puede escribir la ecuación anterior de forma matricial:

$$\mathbf{Y} = \mathbf{X}\beta \tag{6}$$

con

$$\mathbf{Y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}, \quad \mathbf{X} = \begin{pmatrix} x_1^{p-1} & x_1^{p-2} & \cdots & x_1 & 1 \\ x_2^{p-1} & x_2^{p-2} & \cdots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^{p-1} & x_n^{p-2} & \cdots & x_n & 1 \end{pmatrix}, \quad \beta = \begin{pmatrix} \beta_{p-1} \\ \beta_{p-2} \\ \vdots \\ \beta_0 \end{pmatrix} \tag{7}$$

Dado esto, se puede usar a la matriz $\mathbf{X}$ como matriz de diseño para resolver el problema:

$$\min_{\beta} \|\mathbf{Y} - \mathbf{X}\beta\|^2. \tag{8}$$

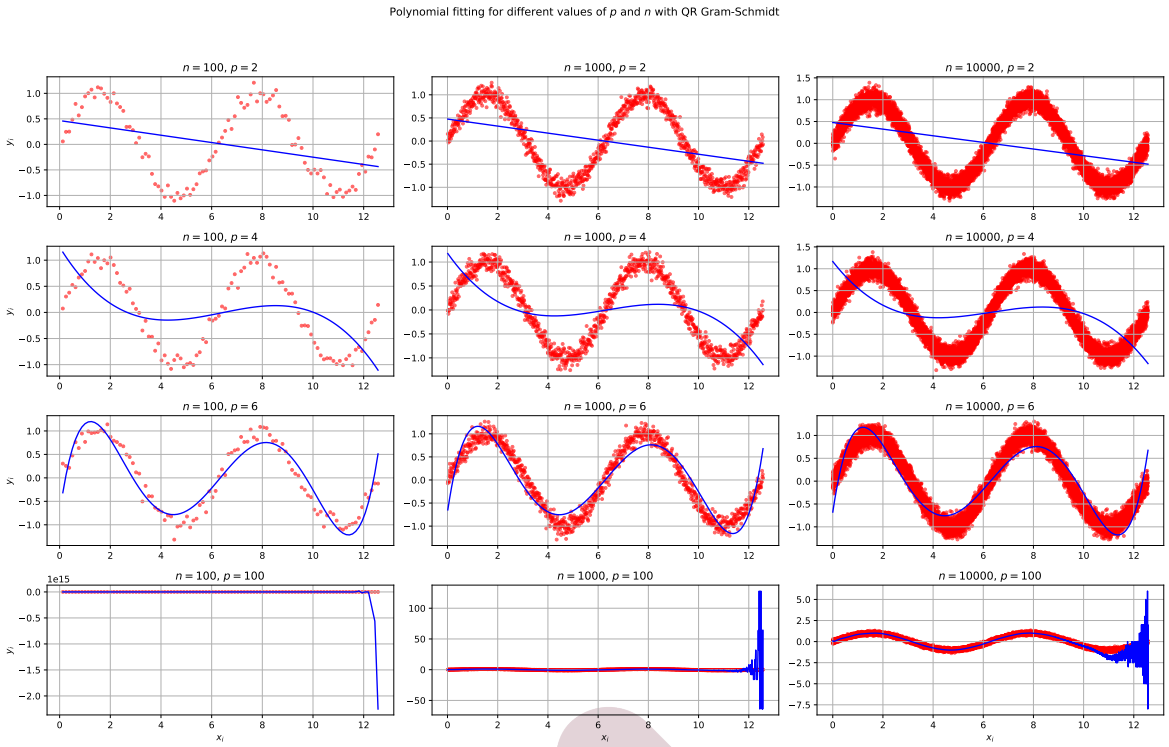
Tomando la factorización QR de $\mathbf{X}$ ($\mathbf{X} = \mathbf{QR}$), se puede resolver el sistema:

$$\mathbf{R}\beta = \mathbf{Q}^T\mathbf{Y} \tag{9}$$

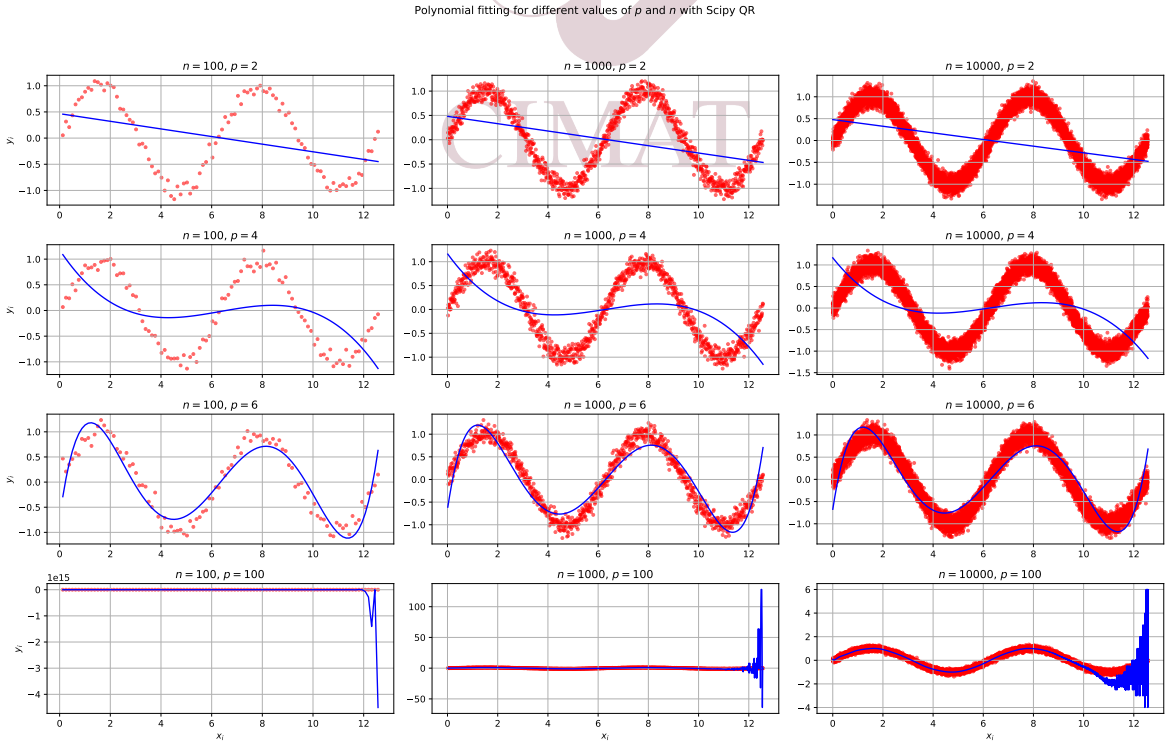
con ayuda de *backward substitution* (ya que $\mathbf{R}$ es triangular superior).

En el archivo ejercicio3.py se implementa la función *AJUSTE_QR()* que da la generación de $\mathbf{Y}$ como se muestra en (4). Con ayuda de la función *vander()* de la librería *numpy*, se crea la matriz $\mathbf{X}$ dada en (7) y se resuelve el sistema (9) para β con ayuda de la función *BACKWARD_SUBST()* de la Tarea 1 y con la factorización obtenida por *MODIFIED_GRAM_SCHMIDT()*. Posteriormente, se calcula la estimación $\hat{\mathbf{Y}} = \mathbf{X}\beta$ y tiene como salidas: x_i, y_i, β y $\hat{y}_i$.

Se consideran los casos para $p = 2, 4, 6, 100$ y para $n = 100, 1000, 10000$. No se imprime ninguna de las características $x_i, y_i, \beta, \hat{y}_i$ para no abrumar el espacio de código. Se generó cada una de las gráficas de los ajustes y se imprimen a continuación en una sola figura. Es importante destacar que en este caso, se usó la función *MODIFIED_GRAM_SCHMIDT()* para la factorización QR involucrada.



Lo mismo se repitió haciendo uso de la descomposición QR de scipy, generando los siguientes resultados gráficos:



Además, se obtuvo la siguiente tabla al momento de comparar los tiempos de ejecución en segundos de ambos métodos:

	QR de Gram-Schmidt	QR de Scipy	Scipy - Gram-Schmidt
$p = 2$ y $n = 100$	7.5667e-05	0.0001	4.4916e-05
$p = 2$ y $n = 1000$	7.6832e-05	7.6791e-05	-4.1999e-08
$p = 2$ y $n = 10000$	0.0216	0.0003	-2.1351e-02
$p = 4$ y $n = 100$	6.5249e-05	4.2833e-05	-2.2417e-05
$p = 4$ y $n = 1000$	0.0001	7.1166e-05	-3.7043e-05
$p = 4$ y $n = 10000$	0.0228	0.0004	-2.2407e-02
$p = 6$ y $n = 100$	7.7917e-05	4.2042e-05	-3.5875e-05
$p = 6$ y $n = 1000$	0.0001	8.6792e-05	-6.3749e-05
$p = 6$ y $n = 10000$	0.0199	0.0018	-1.8163e-02
$p = 100$ y $n = 100$	0.0129	0.0958	8.2915e-02
$p = 100$ y $n = 1000$	0.0244	0.2927	2.6833e-01
$p = 100$ y $n = 10000$	0.2044	0.3147	1.1026e-01

Cuadro 1: Comparación de tiempos de ejecución en segundos.

NOTA: Los resultados pueden variar en cada ejecución del archivo ejercicio3.py ya que depende de la situación de la memoria del dispositivo actual. El comportamiento en cada uno de ellos es bastante similar.

Se exhiben los tiempos de ejecución de ambos algoritmos para cada caso y en la tercera columna se da la diferencia de estos tiempos. Se puede notar que casi todas estas diferencias son negativas, por lo que concluye que la implementación de la factorización QR de Scipy es más rápida que la implementada en el ejercicio 1.

---

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

En el archivo ejercicio4.py, se tiene la implementación y graficación para diversos valores de n y $p = 0.1n$. Se intentó para valores muy grandes de n (en particular, el entero más grande reconocible por la computada es 10^{4300}), sin embargo, lo que "desaparecía" era el ajuste polinomial bastante antes. Es decir, para valores grandes de n, la generación de puntos se realizaba después de un largo tiempo, pero el polinomio de grado p nunca se generaba debido a indeterminaciones en la ejecución de la función *MODIFIED_GRAM_SCHMIDT()*.

Debido a esto, fue necesario encontrar el n tal que $p = 0.1n$ no generara problemas a la hora de hacer el polinomio de grado p. Se llegó a la conclusión que para valores de p mayores a 281, $i.e., n > 2819$ resultaba imposible hacer el ajuste:

Además, se hicieron simulaciones para n todavía más grande, sin embargo, el resultado era el mismo: los datos con ruido se generan, pero el polinomio no:

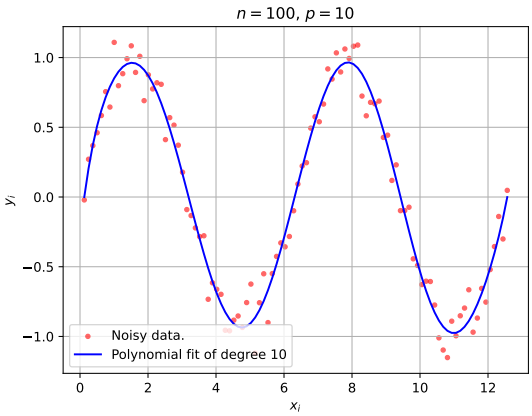


Figura 1: $n = 100, p = 10$.

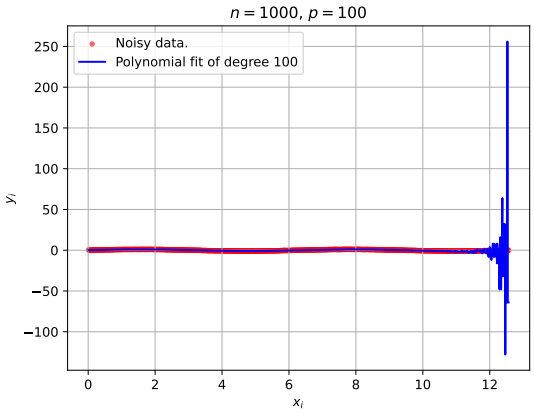


Figura 2: $n = 1000, p = 100$.

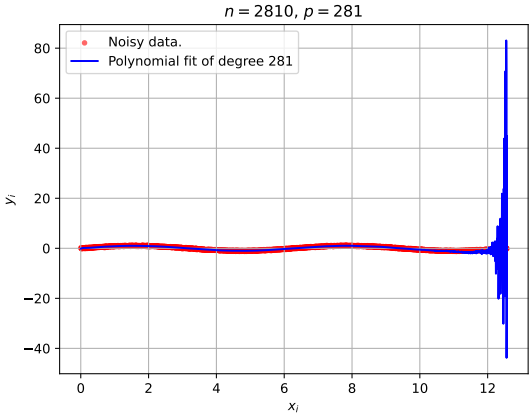


Figura 3: $n = 2810, p = 281$.

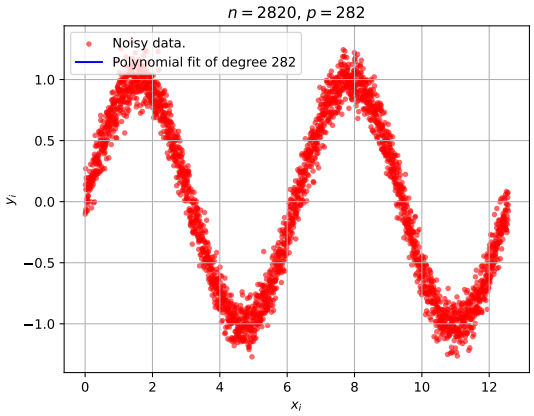
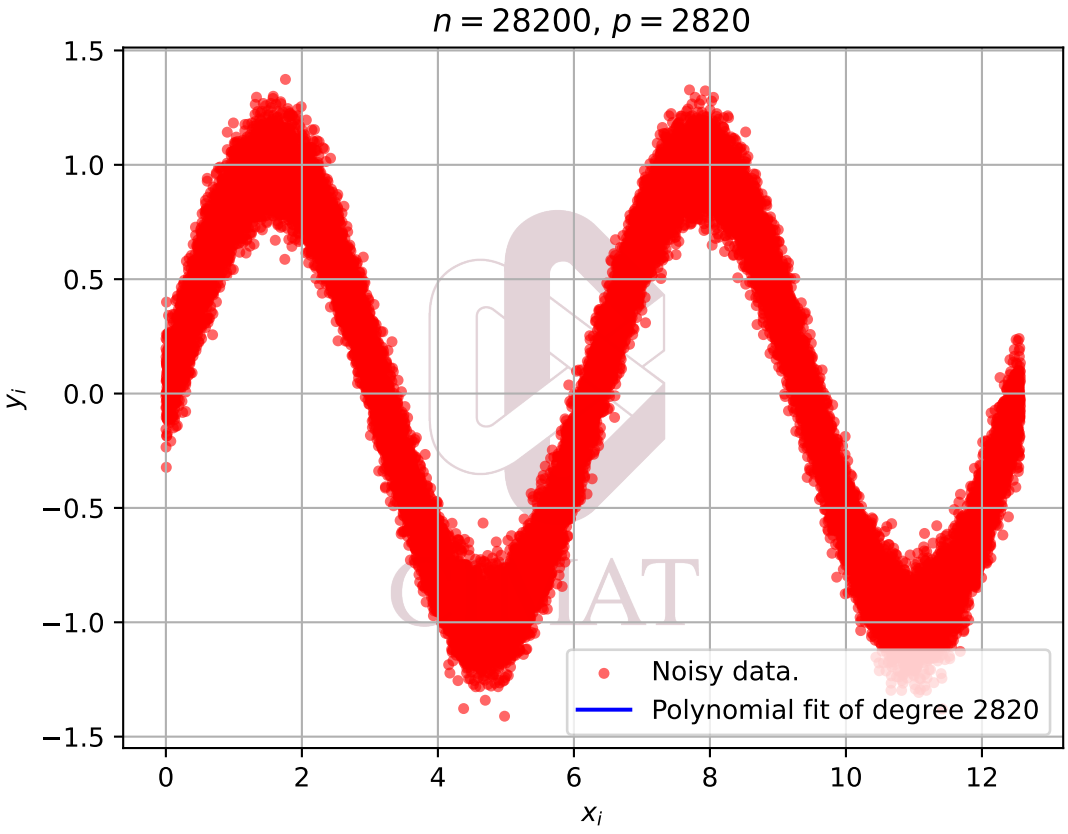


Figura 4: $n = 2820, p = 282$.


$n = 28200$, $p = 2820$
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y_i
0
2
4
6
8
10
12
x_i
Noisy data.
Polynomial fit of degree 2820