Cómputo Científico
Tarea 2
Descomposición QR y mínimos cuadrados

Iván Irving Rosas Domínguez

13 de septiembre de 2023

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

   Solución: Este algoritmo se implementó con el nombre de 'GMmodificado', el cual hace alusión al método para crear la base ortonormal que se calcula en el procedimiento. Este código se puede ver en el script 'GM modificado'. Como recordatorio, observemos que nosotros estamos hallando la descomposición QR reducida, es decir

   $$A = \hat{Q}\hat{R}.$$

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

   Solución: Este algoritmo se implementó con el nombre de 'QRsol', y se puede encontrar en el script 'Minimos cuadrados' de esta tarea.

   Como nota técnica, el algoritmo realiza la descomposición $A = QR$, de tal forma que para 'solucionar' el sistema $Ax = y$, se realiza lo siguiente:

   $$Ax = y \quad \Longrightarrow \quad QRx = y \quad \Longrightarrow \quad Q^*QRx = Q^*y \quad \Longrightarrow \quad Rx = Q^*y,$$

   por lo que solucionar $Ax = y$ equivale a solucionar $Rx = Q^*y$, pero ya conocemos R, por lo que solo resta hallar Q^*y, lo cual es sencillo pues simplemente transponemos Q de la factorización QR y terminamos.

3. Generar $\mathbf{Y}$ compuesto de $y_i = \text{sen}(x_i) + \varepsilon_i$, donde $\varepsilon_i \sim N(0, \sigma^2)$, con $\sigma = 0{,}11$ para $x_i = \frac{4\pi i}{n}$, para $i \in \{1, ..., n\}$. Hacer un ajuste de mínimos cuadrados a $\mathbf{Y}$, con descomposición QR, ajustando un polinomio de grado $p - 1$.

   - Considerar los 12 casos: $p = 3, 4, 6, 100$ y $n = 100, 1000, 10000$.
   - Graficar el ajuste en cada caso.
   - Medir el tiempo de ejecución de su algoritmo, comparar con descomposición QR de scipy y graficar los resultados.

   Solución: Consideramos los 12 casos que se nos pueden presentar. En nuestra presentación elegimos la variable m como la variable n mencionada en este ejercicio, para mayor familiaridad con una matriz de $m \times n$, m observaciones, n regresores.

   El script relacionado con este ejercicio se denomina 'Ejercicio 3', y en él se encuentran los algoritmos para crear una matriz de Vandermonde, realizar la regresión, y el ajuste polinomial con cualesquiera parámetros m, y p, siempre que sean adecuados $m \geq p - 1$. Después de realizar el siguiente ajuste, obtenemos las siguientes gráficas para cada uno de los casos:

   Como podemos observar en las figuras 1 a 4 correspondientes al caso $m = 100, p = 3, 4, 6, 100$, el ajuste polinomial es razonable en las primeras tres, mientras que en la última el ajuste es bastante malo. Este comportamiento no es para nada esperado, puesto que, por ejemplo, para el caso en el que $p = 100$, se estaría ajustando un polinomio de grado 99 a 100 puntos, lo que prácticamente debería otorgarnos una polinomio que conectara punto a punto los datos observados,

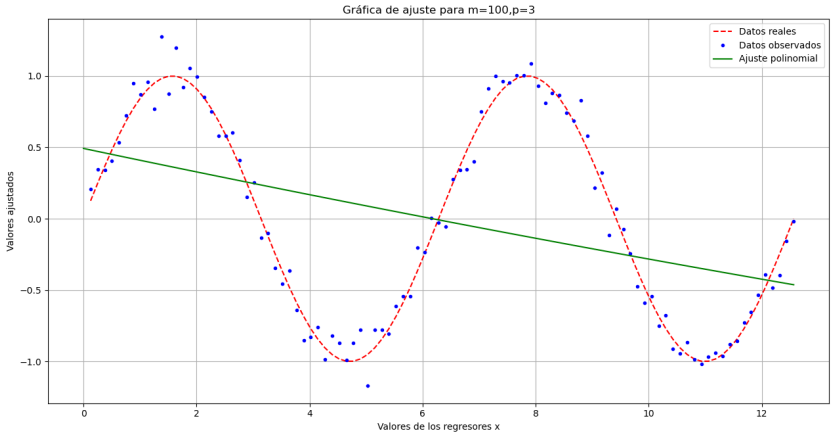

Figura 1:

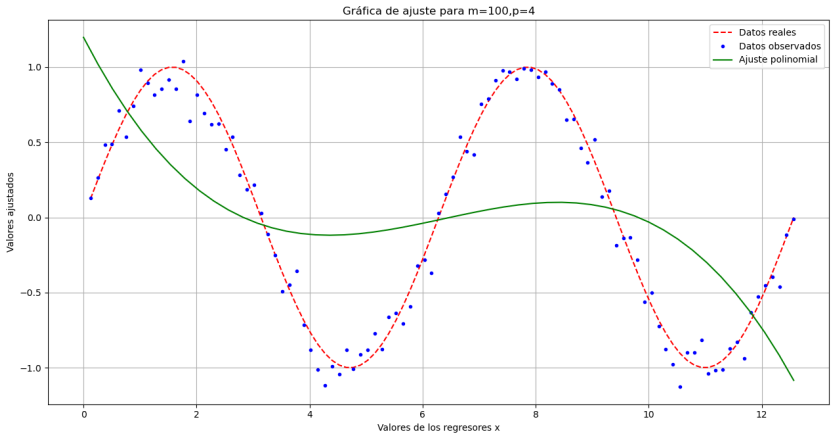

Figura 2:

fenómeno que no se observa.

El resto de ajustes presenta un problema similar. En particular, podemos observar que el ajuste polinomial se asemeja en cierto modo a una gráfica de $2\,\text{sen}(x)$, en vez del $\text{sen}(x)$ que se está tratando de ajustar. Este fenómeno se percibe fuertemente en las figuras 8 y 12 respectivamente. El ajuste en las demás figuras no es el que intuitivamente se esperaría, por ejemplo, en la figura 7, a intuición dice que puede mejorarse el polinomio que aproxime a los datos, sin embargo, el ajuste de ese polinomio es mejor que el que se presenta en la figura 8 por ejemplo.

Finalmente, realizamos la comparación con el algoritmo QR de scipy. Para ello, usamos la función

```
scipy.linalg.qr(-  , mode='economic')
```

el cual calcula QR reducido de acuerdo a la documentación de Scipy. El resultado es que la descomposición de Scipy es notoriamente mejor que la del QR implementado en este ejercicio, como se puede ver en la figura:

4. Hacer $p = 0{,}1n$, o sea, diez veces más observaciones que coeficientes en la regresión. ¿Cuál es la n máxima que puede manejar su computadora?

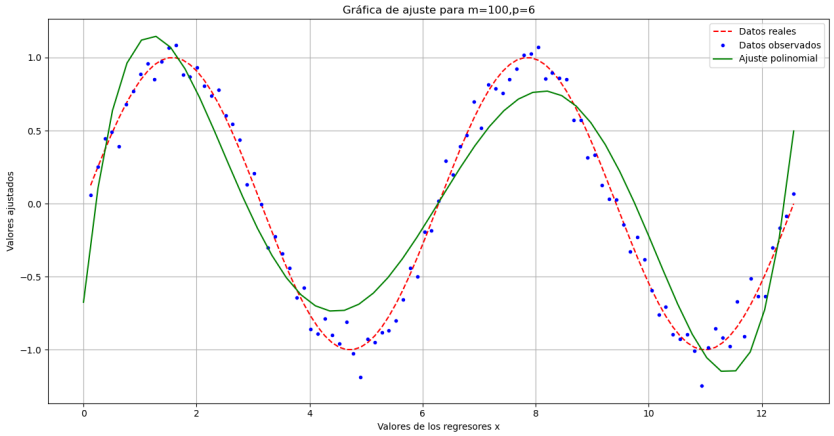

Figura 3:

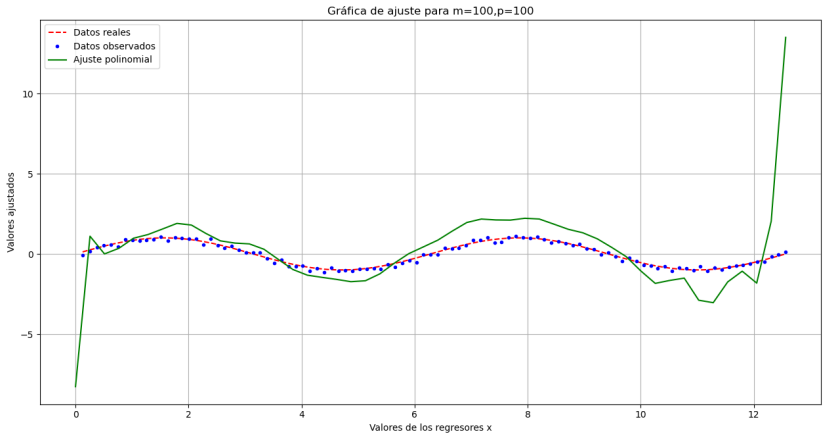

Figura 4:

Solución: Después de realizar varias pruebas, observamos dos comportamientos interesantes en ciertas cantidades:

a) **m=2800 vs m=2810:** aquí observamos que, cuando hay $m = 2800$ datos con $p = 280$ regresores (incluyendo el intercepto), se tiene que el algoritmo trabaja 'correctamente' en el sentido de que el vector y la matriz resultante tienen valores numéricos (aunque, como es posible observar en la figura, la mayoría de las respuestas del vector solución s son cero.)

No obstante, cuando el número de datos llega 2810, y el polinomio es de grado 281, se empiezan a obtener valores en la matriz X que interpretan como infinito, por lo que el vector empieza a regresar respuestas no numéricas 'nan', como se puede observar en la figura. De esta forma, el algoritmo 'deja de ser útil' en $m = 2810$.

b) **m=1540 vs m=1550:** a pesar de que técnicamente Python nos regresa una matriz X y un vector de soluciones s para el ítem a), desde que $m = 1550$, empiezan a surgir errores del tipo 'overflow', como se puede observar en las figuras, esto es, estamos ya fuera del rango que puede representarse con cierto número de dígitos en Python. Esto coincide precisamente con el primer instante en el que el vector solución s empieza a presentar ceros como parte de los coeficientes de la solución.

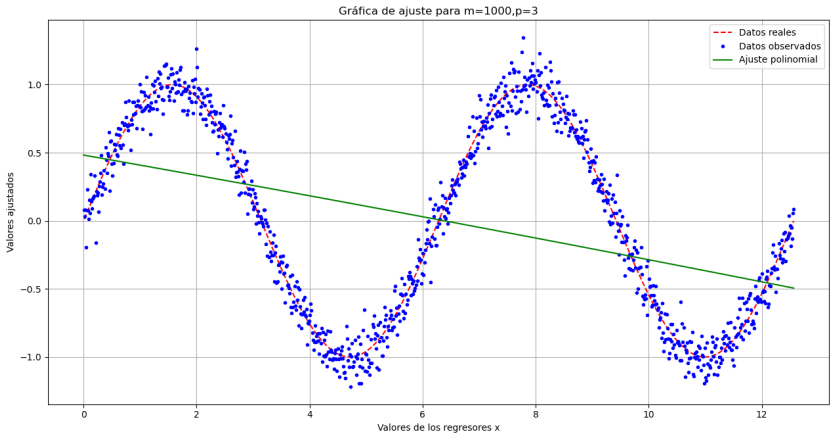

Figura 5:

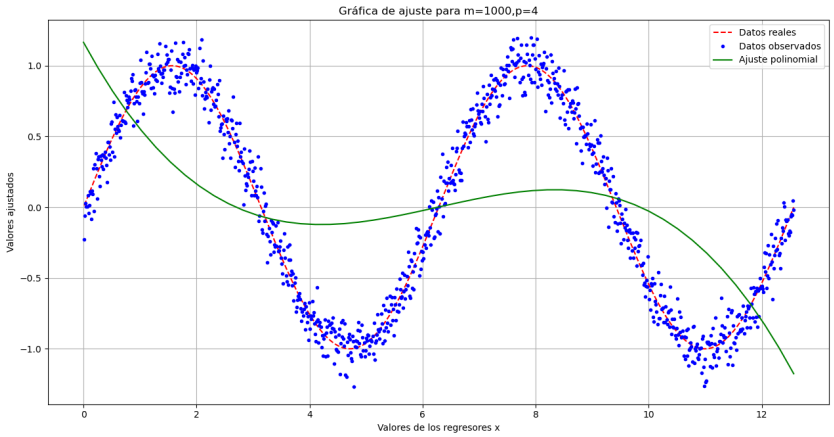

Figura 6:

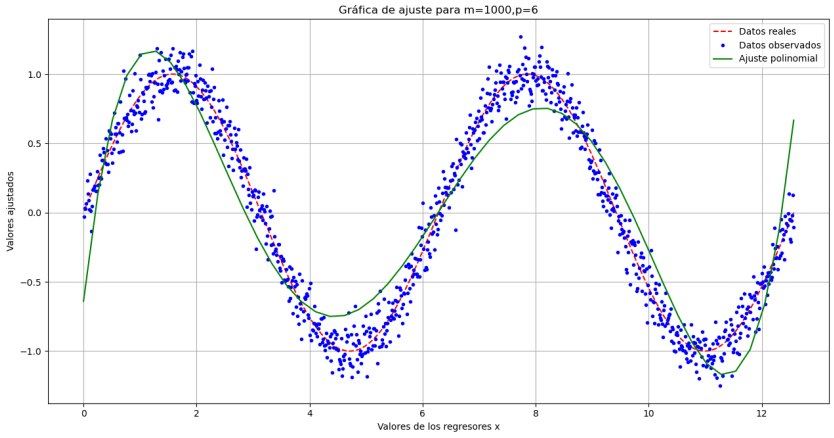

Figura 7:

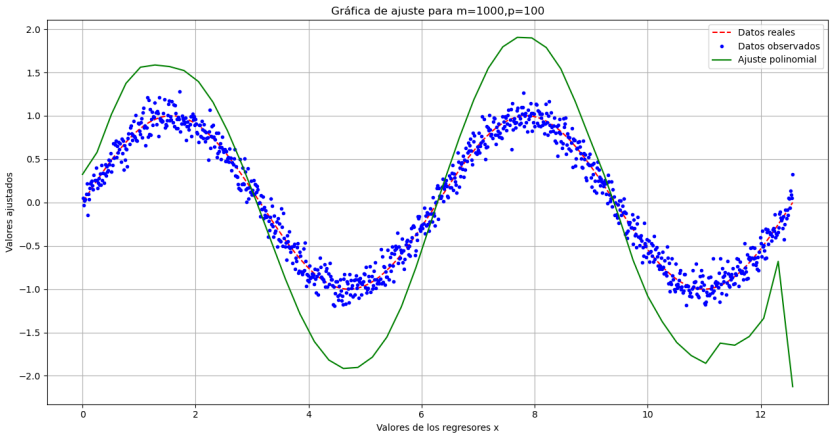

Figura 8:

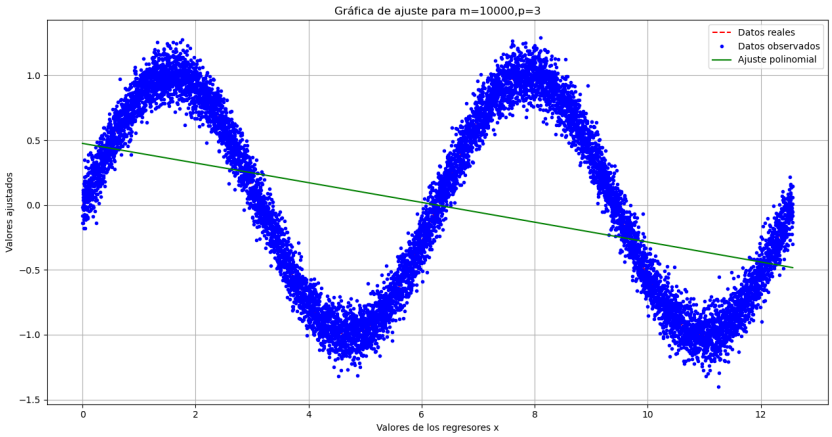

Figura 9:

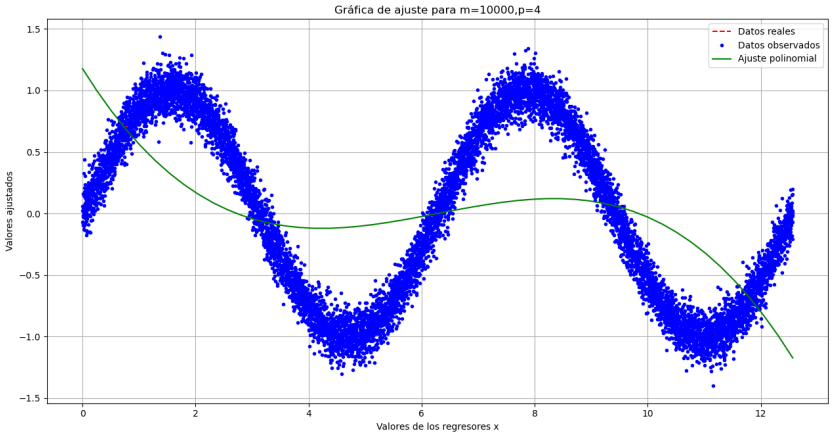

Figura 10:

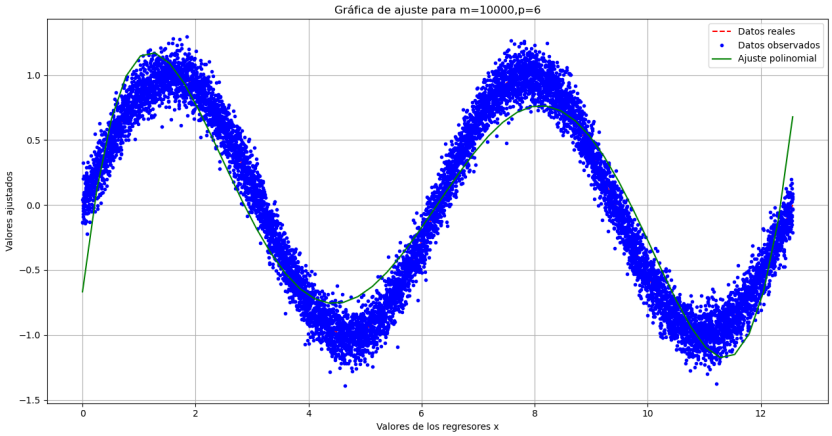

Figura 11:

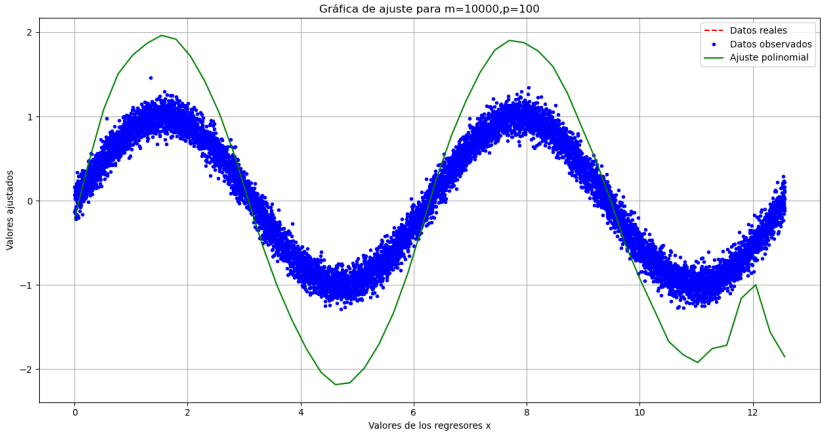

Figura 12:

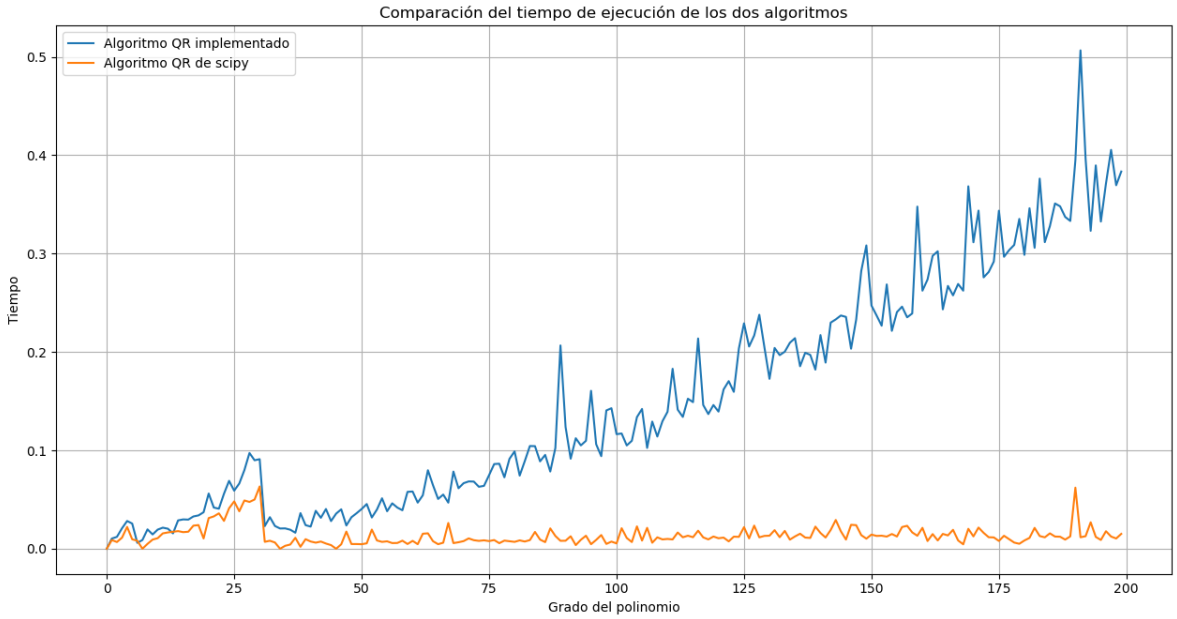

Figura 13: Tiempos de ejecución del algoritmo QR implementado vs QR de scipy.

```
>>> X
array([[1.00000000e+000, 4.48798951e-003, 2.01420498e-005, ...,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [1.00000000e+000, 8.97597901e-003, 8.05681992e-005, ...,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [1.00000000e+000, 1.34639685e-002, 1.81278448e-004, ...,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       ...,
       [1.00000000e+000, 1.25573946e+001, 1.57688160e+002, ...,
        3.11941322e+305, 3.91717028e+306, 4.91894531e+307],
       [1.00000000e+000, 1.25618826e+001, 1.57800895e+002, ...,
        3.44520651e+305, 4.32782797e+306, 5.43656670e+307],
       [1.00000000e+000, 1.25663706e+001, 1.57913670e+002, ...,
        3.80489081e+305, 4.78136681e+306, 6.00844273e+307]])
>>> s
array([ 2.27200840e-001,  1.22193506e+000,  2.66856388e+001,
       -1.92043383e+002,  6.36483772e+002, -1.27757418e+003,
        1.73924112e+003, -1.69700292e+003,  1.22152579e+003,
       -6.58833430e+002,  2.68061657e+002, -8.21542868e+001,
        1.87418227e+001, -3.09443271e+000,  3.48414703e-001,
       -2.32533721e-002,  5.50511781e-004,  1.40646104e-005,
       -2.20118669e-007,  8.12243290e-007, -2.03577235e-007,
        1.79927047e-008, -6.56975516e-010,  7.80972545e-012,
        2.10723430e-012, -1.62935974e-013, -4.37297289e-014,
        4.25811041e-015,  6.76576964e-017,  7.12446541e-018,
       -1.78033141e-018, -9.99360659e-020,  8.39643188e-021,
        1.31967773e-022, -3.85336292e-025,  3.37554142e-024,
        1.09093866e-025, -2.76329815e-026, -1.74712369e-027,
       -3.12988525e-029,  1.64097852e-029, -6.96422565e-031,
       -2.55121398e-033,  3.35842971e-033,  6.06609265e-034,
       -1.73030203e-035, -3.55620374e-036,  1.66355389e-037
```

Figura 14: Resultado del algoritmo cuando m=2800, p=280

```
        6.96334319e-147,  5.42904193e-148, -5.65072441e-150,
0,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000,  0.00000000e+000,
        0.00000000e+000,  0.00000000e+000])
>>> 
```

Figura 15: Resultado del algoritmo cuando m=2800, p=280

```
>>> X
array([[1.00000000e+000, 4.47201801e-003, 1.99989451e-005, ...,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [1.00000000e+000, 8.94403602e-003, 7.99957804e-005, ...,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       [1.00000000e+000, 1.34160540e-002, 1.79990506e-004, ...,
        0.00000000e+000, 0.00000000e+000, 0.00000000e+000],
       ...,
       [1.00000000e+000, 1.25574266e+001, 1.57688962e+002, ...,
        3.91995132e+306, 4.92245008e+307,             inf],
       [1.00000000e+000, 1.25618986e+001, 1.57801296e+002, ...,
        4.32936345e+306, 5.43850246e+307,             inf],
       [1.00000000e+000, 1.25663706e+001, 1.57913670e+002, ...,
        4.78136681e+306, 6.00844273e+307,             inf]])
>>> s
array([nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan
, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan
, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan
, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan
, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan
, nan,
       nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan
, nan,
```

Figura 16: Resultado del algoritmo cuando m=2810, p=281

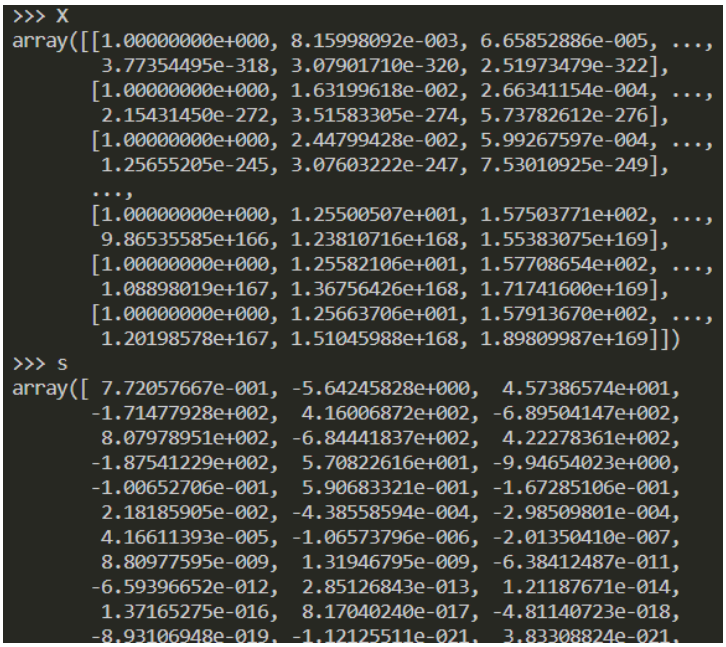

Figura 17: Resultado del algoritmo cuando m=1540, p=154

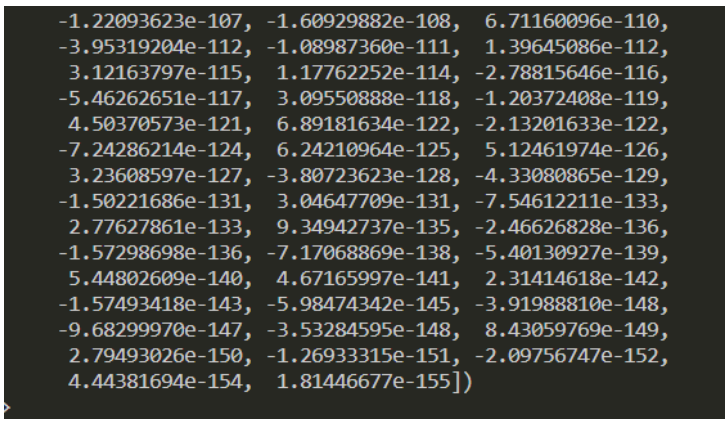

Figura 18: Resultado del algoritmo cuando m=1540, p=154

```
<stdin>:7: RuntimeWarning: overflow encountered in matmul
>>> X
array([[1.00000000e+000, 8.10733588e-003, 6.57288951e-005, ...,
        1.14376197e-320, 9.38724727e-323, 0.00000000e+000],
       [1.00000000e+000, 1.62146718e-002, 2.62915580e-004, ...,
        1.30601664e-274, 2.11766312e-276, 3.43372124e-278],
       [1.00000000e+000, 2.43220076e-002, 5.91560056e-004, ...,
        1.14264506e-247, 2.77914219e-249, 6.75943177e-251],
       ...,
       [1.00000000e+000, 1.25501559e+001, 1.57506414e+002, ...,
        1.23969742e+168, 1.55583959e+169, 1.95260295e+170],
       [1.00000000e+000, 1.25582633e+001, 1.57709977e+002, ...,
        1.36844168e+168, 1.71852509e+169, 2.15816905e+170],
       [1.00000000e+000, 1.25663706e+001, 1.57913670e+002, ...,
        1.51045988e+168, 1.89809987e+169, 2.38522264e+170]])
>>> s
array([ 7.19859790e-001, -3.05964119e+000,  7.84509469e+000,
        4.14730333e+001, -2.23669467e+002,  4.99254673e+002,
       -6.62022321e+002,  5.74649379e+002, -3.38256973e+002,
        1.34070163e+002, -3.26848576e+001,  2.55112475e+000,
        1.43770089e+000, -6.86608090e-001,  1.61222716e-001,
       -2.37809728e-002,  2.13458081e-003, -7.04916371e-005,
       -9.15532921e-006,  1.54471394e-006, -1.13125880e-007,
        5.54090497e-009, -2.59223773e-010, -5.37848833e-012,
        2.95342787e-012,  1.42515176e-014, -3.84981366e-014,
        6.63299535e-016,  3.37368605e-016, -1.71355827e-017,
       -5.02337065e-019,  4.01169545e-020, -8.14028933e-021,
        1.29923356e-021, -6.33653845e-023, -7.74161185e-025,
        3.62058733e-026,  1.29198105e-026,  2.35983965e-027,
       -4.58790725e-028,  2.05233932e-029, -8.46879001e-031,
```

Figura 19: Resultado del algoritmo cuando m=1550, p=155

```
    1.34227172e-094, -4.58269490e-096,  1.52881788e-097,
    1.22567642e-097,  5.64691981e-099,  1.59773917e-100,
   -2.71426105e-101, -1.05993558e-102,  3.80277148e-103,
   -2.16385374e-104,  2.03948182e-105, -3.16350195e-106,
    1.93321102e-107, -8.90933593e-110, -1.60172577e-109,
    2.33963582e-111,  6.94263506e-112,  4.38994010e-113,
    5.35776889e-115, -8.20426776e-115,  3.45377870e-116,
   -7.34326329e-117,  4.97827376e-118, -1.13992156e-119,
    9.04241490e-122,  1.65826438e-121, -1.00707674e-122,
    1.90449282e-123,  3.81088992e-126, -6.04893943e-126,
   -2.40148403e-127, -1.84322015e-129, -1.24064392e-129,
    1.49907160e-130, -3.26184681e-131,  2.04589968e-133,
    3.99841181e-134,  9.15475981e-135, -2.46730577e-136,
   -4.52833411e-137,  8.96610176e-138,  3.36369548e-139,
   -2.00119185e-140, -9.32269468e-142, -9.90328990e-143,
   -1.12728136e-143, -3.36898289e-145, -5.21763667e-147,
    2.35183531e-147,  6.43023598e-148,  1.05008092e-149,
   -3.17703549e-150,  1.68844644e-152,  1.83199358e-152,
   -2.12986904e-153,  6.84786307e-155,  0.00000000e+000])
```

Figura 20: Resultado del algoritmo cuando m=1550, p=155