Práctica 1 - Computación numérica

Máster en Ciencia de Datos

José Antonio Álvarez Ocete
Francisco Javier Sáez Maldonado

October 21, 2021

1 Apartado 1: Errores

1.1 Cuestiones Previas

1) Dado que el redondeo en base 10 de la suma o producto de dos números es $\pm 0.5 \cdot \epsilon$, donde ϵ es el valor la última cifra significativa y que el dígito eliminado es aleatorio ¿podemos suponer que el redondeo es una variable aleatoria uniforme?

Dado que ϵ es una variable aleatoria uniforme, todos los números tendrán la misma probabilidad, luego sí podemos suponer que el redondeo es una v.a. uniforme.

2) Si dibujo la gráfica de función de densidad del error por redondeo ¿Como debería ser dicha gráfica?

La gráfica de la densidad debería parecerse a la función de densidad de una uniforme en el intervalo $[0 - \epsilon, 0 + \epsilon]$ pues el error de redondeo puede ser tanto por exceso como por defecto.

3) Si asumimos que el error de redondeo es una variable aleatoria uniforme entre $-0.5 \cdot \epsilon$ y $0.5 \cdot \epsilon$ ¿Cuál debería ser error absoluto promedio de la suma (o el producto) de un número elevado de números en coma flotante?, da una respuesta razonada.

Debemos aplicar que el error absoluto tanto en la suma como en el producto de dos valores es igual a la suma de los errores de ambos. Por tanto, si sumamos o multiplicamos n términos, deberemos sumar los errores de los n términos. En promedio, puesto que habrá errores tanto por exceso como por defecto, al sumar todos los errores el promedio tenderá a cero por el teorema central del límite.

1.2 Ejercicio 1

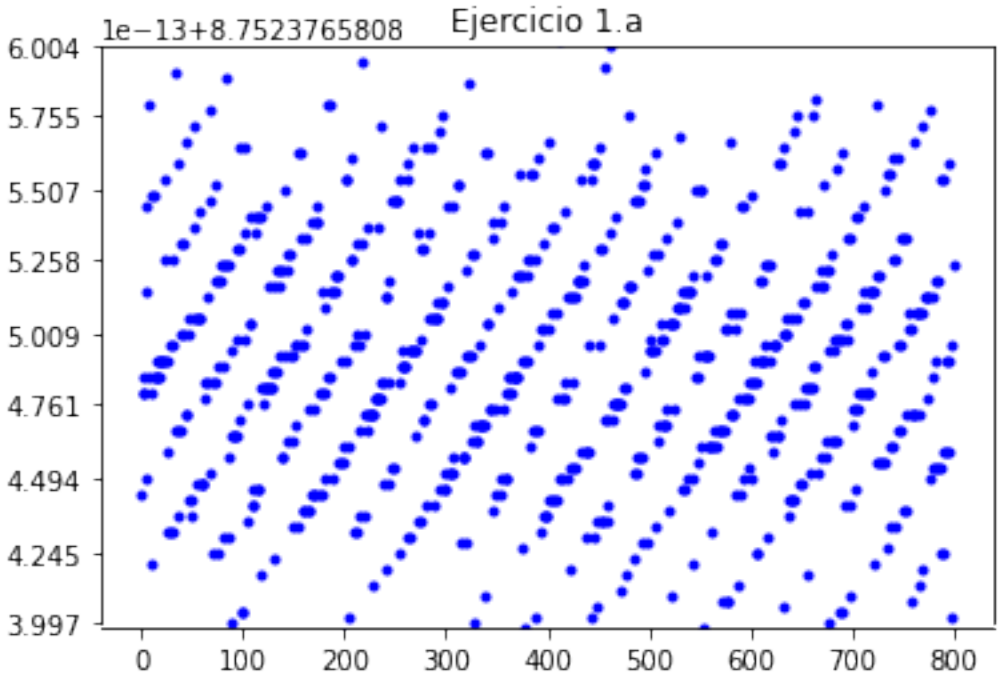
Consideramos la función

$$f(x) = \frac{4x^4 - 59x + 324x^2 - 751x + 622}{x^4 - 14x^3 + 72x^2 - 151x + 112}.$$

Vamos a considerar también los valores $x = 1,606 + 2^{-52}i$ con $i = 0, 1, \ldots, 800$ y a dibujar la función en esos puntos.

1) ¿Sale una figura continua? ¿Por qué? ¿Puedes explicar el patrón que sale? ¿Qué consecuencias puedes sacar sobre el redondeo?

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

def plot_func(f, i_range, y_lims, i_shift=1.606, title=""):
    values = f(i_shift + 2**(-52) * i_range)
    plt.plot(i_range, values, 'b.')
    plt.ylim(y_lims[0], y_lims[1])
    plt.title(title)
    plt.show()

f1 = lambda x: (4*x**4 - 59*x**3 + 324*x**2-751*x + 622) / \
                (x**4 - 14*x**3 + 72*x**2 - 151*x + 112)
i_range = np.arange(800)
y_lims= [8.7523765807784, 8.7523765807786]
plot_func(f1, i_range, y_lims, title = "Ejercicio 1.a")
```



Como vemos, la figura obtenida no es una figura continua, sino que los puntos se distribuyen en líneas oblicuas. El gráfico debería ser una función continua pues el denominador no tiene raíces reales, luego nuestra función inicial es una función continua. Es por ello que esto nos muestra que el error de redondeo podría no seguir realmente una distribución uniforme al contrario de lo que hemos comentado teóricamente en las preguntas anteriores.

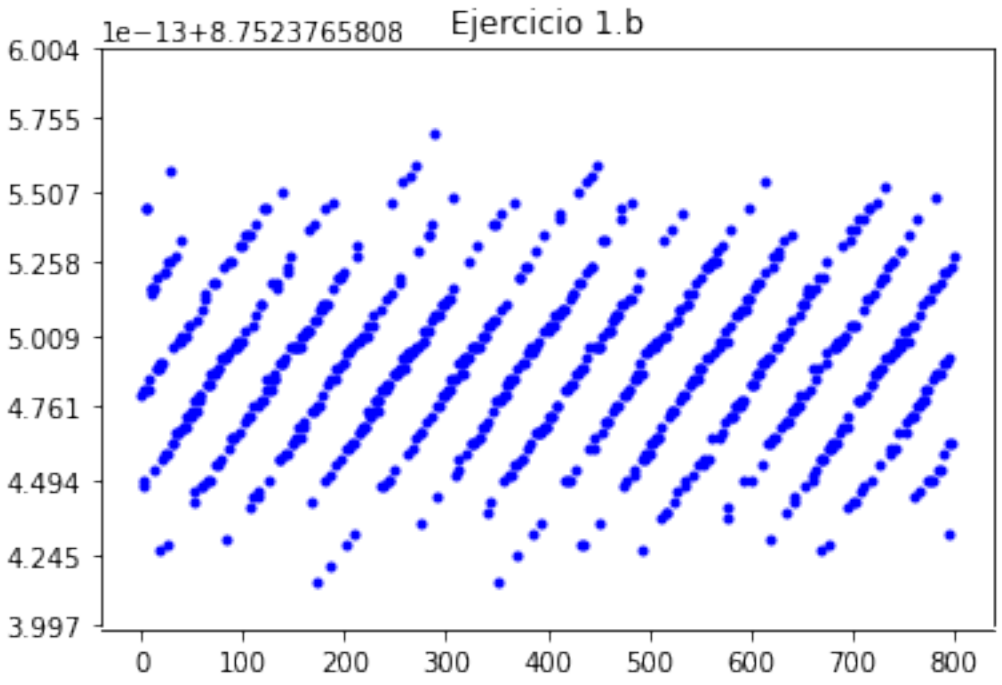
2) Aplicación de la regla de Horner

Consideramos la función

$$f(x) = \frac{622 + x\,(-751 + x(324 + x(-59 + 4x)))}{112 + x \cdot (-151 + x(72 + x(-14 + x)))}$$

Esta función es la misma que la anterior, solo que hemos extraído x en común cuando era posible. La definimos y la representamos

[2]:
```
f2 = lambda x: (622 + x * (-751 + x * (324 + x * (-59 + 4 * x))) ) / \
               (112 + x * (-151 + x * (72 + x * (-14 + x))) )
plot_func(f2, i_range, y_lims, title = "Ejercicio 1.b")
```



Se puede observar que ahora los valores que obtiene la función, aunque siguen separados en líneas oblícuas, se encuentran más concentrados. Es decir, se tiene una varianza menor en estas líneas. Esto acercaría un poco los resultados a que la distribución del error de redondeo se parezca algo más a una uniforme aunque sigue lejos de serlo.

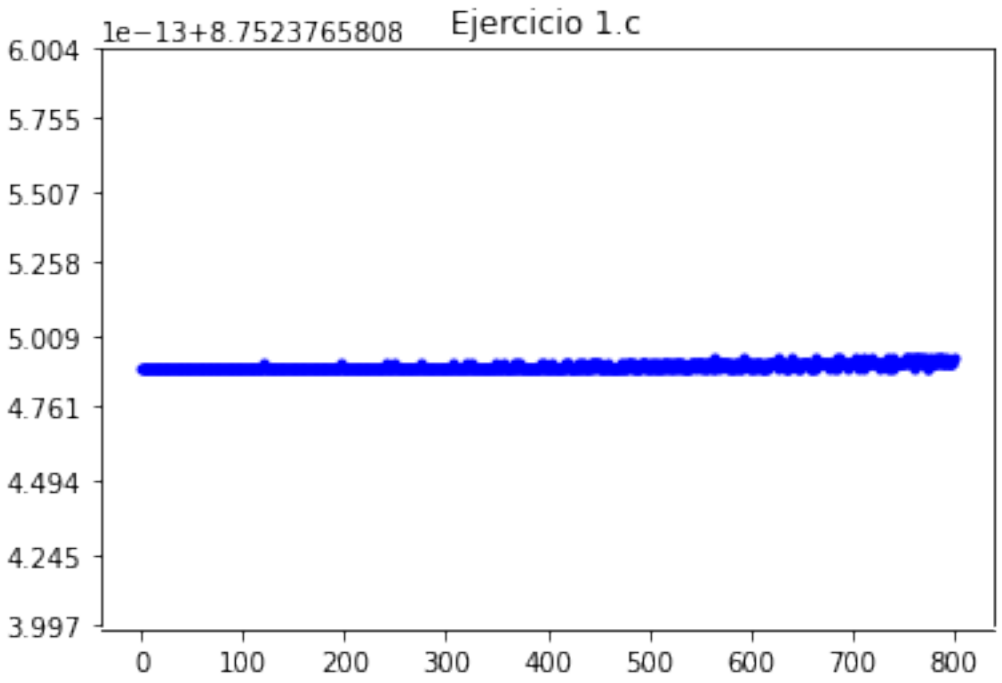
3) Con análisis previo

Vamos ahora a dibujar la gráfica de la función:

$$f(x) = 4 - \frac{3(x-2)[(x-5)^2 + 4}{x + (x-2)^2[(x-5)^2 + 3]}$$

Vamos a representarla en los mismos valores de x y ver qué ocurre en este caso.

[3]:
```
f3 = lambda x: 4 - ( 3 * (x-2) * ((x-5)**2 + 4) ) / \
                ( x + (x-2)**2 * ((x-5)**2 + 3) )
plot_func(f3, i_range, y_lims, title = "Ejercicio 1.c")
```



Vemos como en esta función, los valores quedan mucho más cercanos a ser una función continua que en los casos anteriores. Consideramos como función $f(x)$ la última escrita. Hemos de ver lo siguiente para apoyar nuestro argumento:

$$f(x) = 4 - \frac{3x^3 - 36x^2 + 147x - 174}{x^4 - 14x^3 + 72x^2 - 151x + 112} = \frac{4x^4 - 59x^3 + 324x^2 - 751x + 622}{x^4 - 14x^3 + 72x^2 - 151x + 112}$$

que es la misma función que teníamos en el primer apartado. Esto nos está indicando que la forma de escribir las funciones en `python` puede afectar a cómo se distribuye finalmente el error de redondeo cuando evaluamos estas funciones. Esto es una consecuencia de que ciertas operaciones pueden inducir a mayores errores en el redondeo, como realizar potencias a un número decimal.

1.2.1 Cuestiones finales

1) ¿Las tres funciones que hemos pintado son la misma función, solo que escrita de diferente manera? Da una respuesta razonada.

Hemos comentado y probado en los dos apartados anteriores que que la función es la misma escrita de tres formas diferentes.

2) ¿Podemos afirmar ahora que la distribución del error por redondeo es una variable aleatoria uniforme?

Podemos ver que, aunque de forma teórica la distribución del error por redondeo es una variable aleatoria uniforme, de forma empírica demostramos que no siempre sigue esta distribución, sino que depende también de cómo realicemos las operaciones con los números en coma flotante.

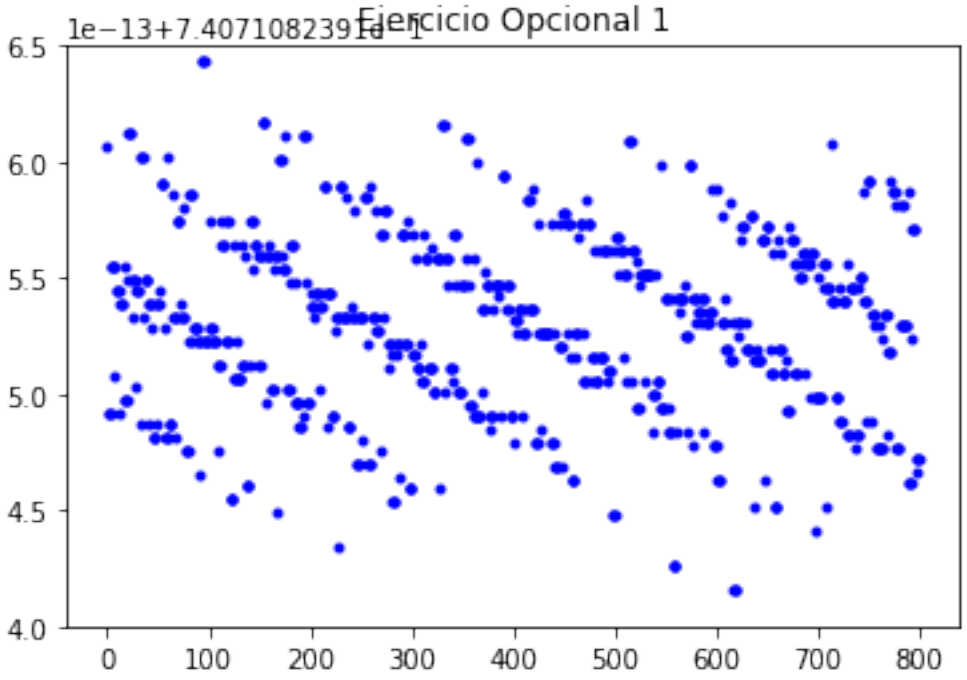
3) Comenta los resultados obtenidos.

A la vista de lo anterior, vemos que los números aleatorios se agrupan formando hiperplanos en el espacio en el que se representan (en este caso $\mathbb{R}^2$), en lugar de uniformemente como se creía inicialmente

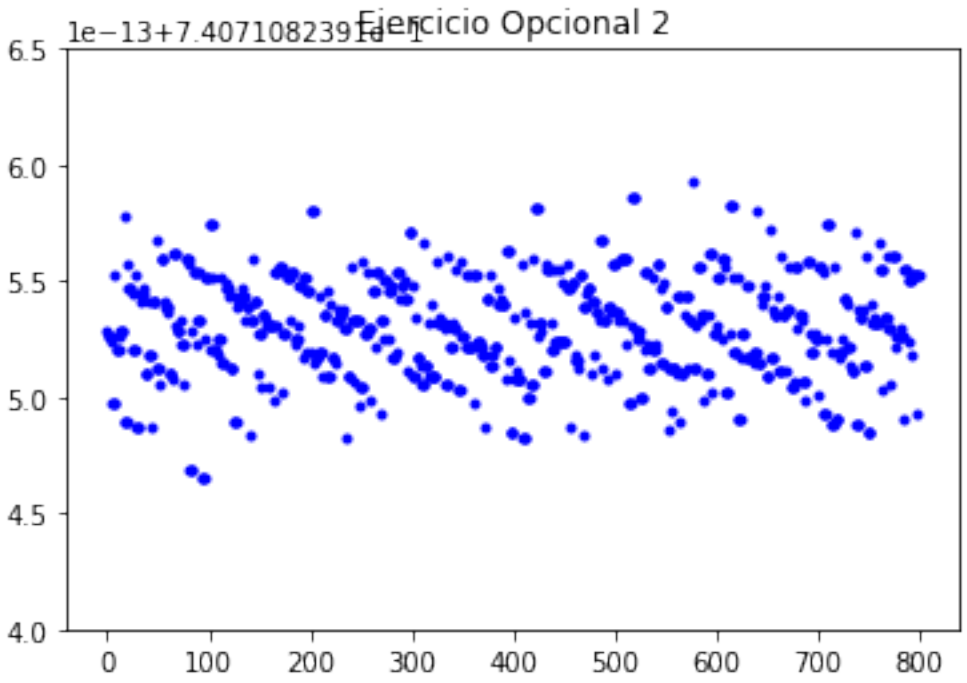
1.3 Ejercicio opcional

Vamos a dibujar ahora las mismas gráficas para valores de $x = 2.4 + 2^{-52}i$ para $i = 0, \ldots, 800$, el mismo rango que el anterior. Cambiamos también los valores d elos límites de y.

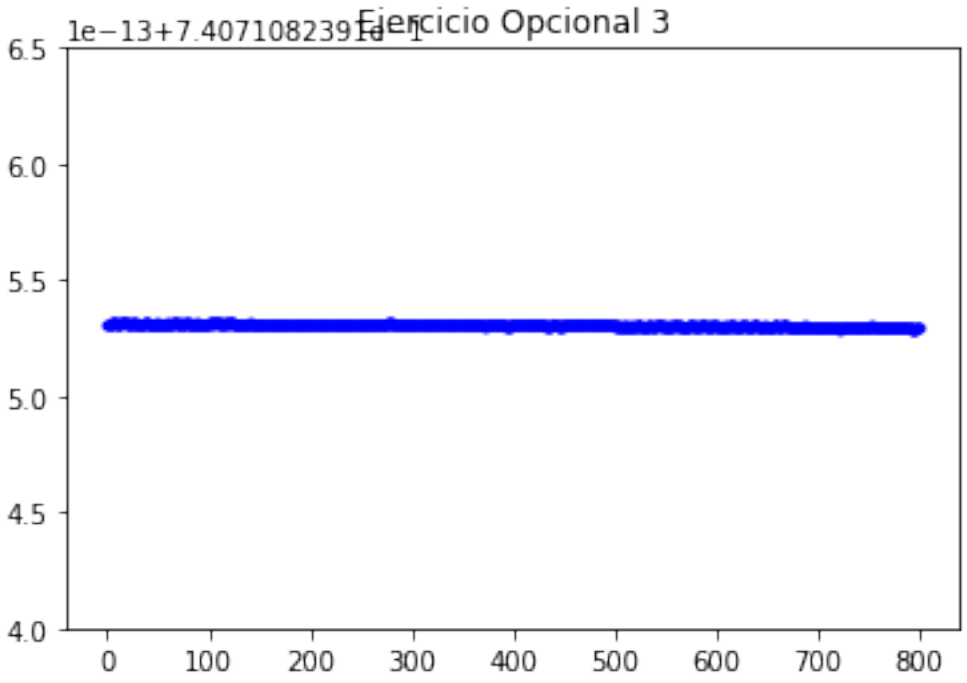
[4]:
```
i_shift2 = 2.4
y_lims2 = [.7407108239094, .74071082390965]
plot_func(f1, i_range, y_lims2, i_shift2, "Ejercicio Opcional 1")
```



[5]:
```
plot_func(f2, i_range, y_lims2, i_shift2, "Ejercicio Opcional 2")
```

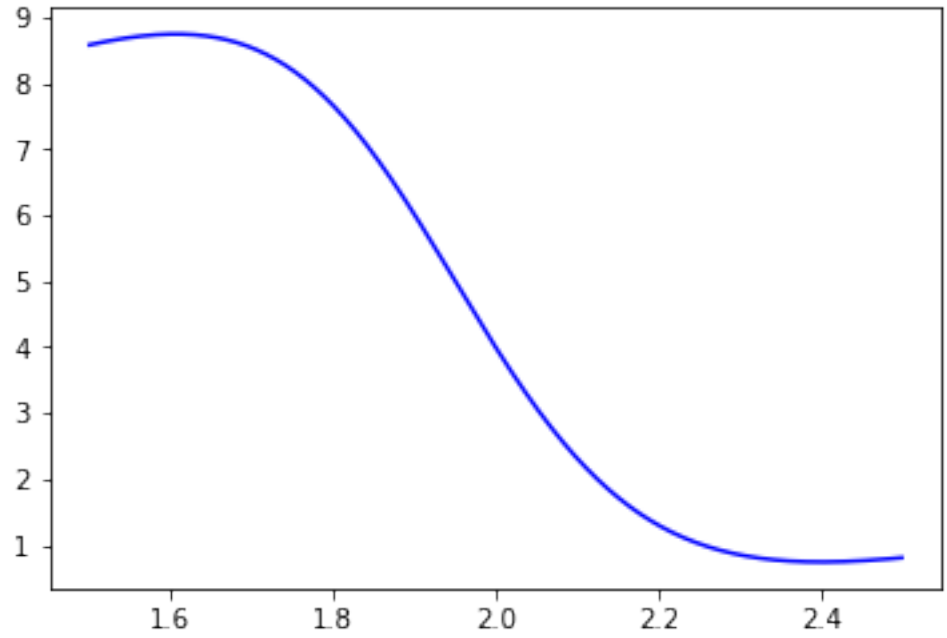


[6]:
```
plot_func(f3,i_range,y_lims2,i_shift2, "Ejercicio Opcional 3")
```



Podemos ver como han cambiado las direcciones de los hiperplanos en los que los puntos se acumulan. Tratamos de dibujar la función para ver si podemos obtener más información sobre la misma.

[7]:
```
x_vals = np.linspace(1.5,2.5,1000)
y_vals = f3(x_vals)
plt.plot(x_vals,y_vals,'b')
plt.show()
```



Como podemos ver, alrededor del valor $x = 1.6$ (que es del cual estamos cerca en el primer ejercicio), la función tiene un máximo, y alrededor del valor $x = 2.4$ que usamos en la segunda parte, la funcion parece tener un mínimo local o cambio de curvatura. Es por ello que la curvatura o la derivada de la función podrían tener que ver en la dirección de los hiperplanos en los que se distribuyen los puntos.

2 Apartado 2: Aproximación de funciones

2.1 Cuestiones previas

Antes de implementar el apartado, responded a las siguientes preguntas.

1) ¿Dado un conjunto de n puntos, existe siempre un polinomio de grado m < n-1 que pase por dichos puntos?

Esto no es cierto. Podemos poner un contraejemplo.

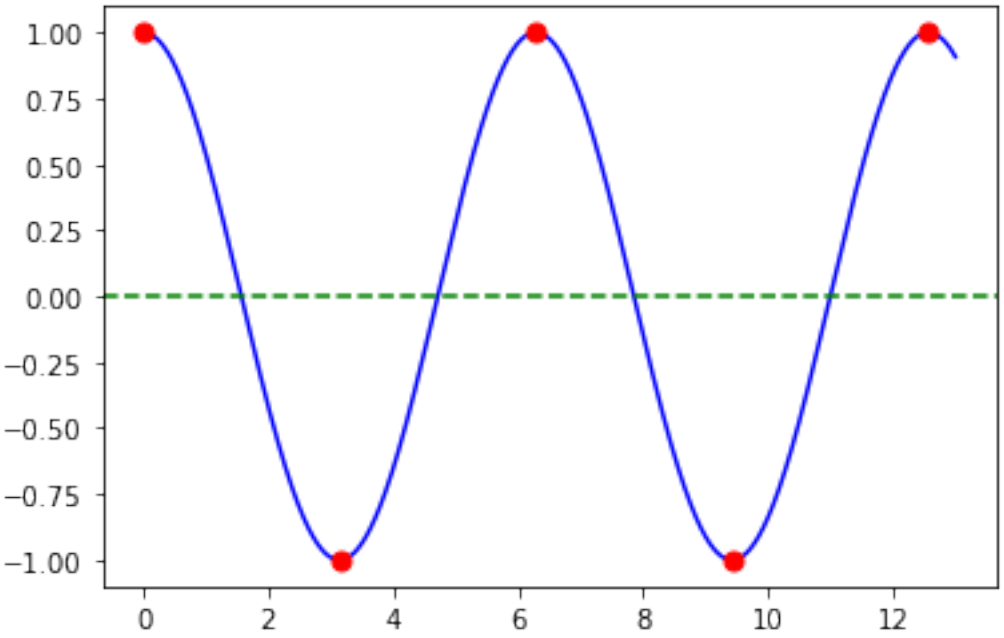
Supongamos que tenemos el siguiente conjunto de puntos de $\mathbb{R}^2$: $\{(x_1, 1), (x_2, -1), (x_3, 1), \ldots, (x_n, \pm 1)\}$, con $x_1 < x_2 < \cdots < x_n$. Un polinomio que pase

por todos esos puntos ha de pasar por 0 al menos $n-1$ veces (aplicando el Teorema de Bolzano), luego tendrá al menos $n-1$ raíces. Por lo tanto, para este conjunto de puntos no existe un polinomio de grado menor que $n-1$ que pase por todos ellos.

Cabe destacar que para este razonamiento no basta cualquier conjunto de puntos. Si tomasemos todos los puntos con coordenadas $y = 0$, una línea recta pasaría por todos ellos.

Veamos un ejemplo gráfico de este razonamiento para $n = 5$ puntos. Para que nuestro polinomio pase por todos los puntos necesitamos que nuestra pase 4 veces por el cero, es decir, tener un polinomio de grado 4 al menos. Nigún polinomio de grado 3 podría cumplirlo.

[8]:
```
n = 5
func = lambda x : np.cos(x)
x_vals = np.linspace(0,13,1000)
values = func(x_vals)
plt.plot(x_vals, values, 'b')
for i in range(n):
    plt.plot(i*np.pi,func(i*np.pi),'r.',markersize=14)
plt.axhline(y=0, color='g', linestyle='--')
plt.show()
#plot_func(func,)
```



2) ¿Se te ocurre una manera en la cual el cálculo de los valores singulares de una matriz permita calcular un polinomio de regresión adecuado?

Existe un ejemplo conocido en el ámbito del aprendizaje automático. Queremos encontrar polinomio que, escrito en forma matricial, nos resuelva un problema de la forma:

$$Xw = y,$$

donde X serán los datos a los que se le añade una columna de 1 para obtener el término independiente del polinomio, w serán los coeficientes del polinomio e y serán los valores que queremos que se obtengan. Con esto, podemos utilizar la **pseudoinversa de Moore-Penrose** y obtener que

$$w = (X'X)^{-1}X'y,$$

donde al término $(X'X)^{-1}X' = X^+$ se le denomina la **pseudoinversa** anteriormente comentada. Además, vemos que:

$$\begin{aligned} A^+ &= (A^*A)^{-1}A^* \\ &= (V\Sigma U^* U\Sigma V^*)^{-1}V\Sigma U^* \\ &= (V\Sigma^2 V^*)^{-1}V\Sigma U^* \\ &= (V^*)^{-1}\Sigma^{-2}V^{-1}V\Sigma U^* \\ &= V\Sigma^{-2}\Sigma U^* \\ &= V\Sigma^{-1}U^* \end{aligned}$$

Por lo que podemos expresar la pseudoinversa utilizando la descomposición SVD de una matriz y, por tanto, los valores singulares. Así, estaríamos resolviendo el problema de regresión polinomial usando los valores singulares. este ha sido también el proceso explicado en clase para calcular el polinomio de regresión para un conjunto de datos.

3) ¿Como podrías transformar el problema de encontrar un polinomio de regresión en un problema de producto de matrices?

El metodo sería exactamente el mismo que el descrito anteriormente.

4) ¿Crees que, además, es posible calcular el error de regresión a partir de los valores singulares?

Con la notación de la segunda pregunta, sabemos que el error se calcula como:

$$E(w) = ||Xw - y||,$$

por lo que podríamos descomponer X en valores singulares y utilizarlos para calcular este error.

2.2 Apartado 1

Vamos a considerar el conjunto de datos

i	x_i	y_i
1	0	1
2	0.25	1.2840
3	0.5	1.6487

i	x_i	y_i
4	0.75	2.1170
5	1	2.7183

Y vamos a buscar un polinomio de grado 2, $P_2(x) = a_0 + a_1 x + a_2 x^2$, que aproxime a estos datos minimizando el error.

Para ello, escribiremos primero la matriz de Vandermonde 5×3. Esta matriz tendrá una fila por cada elemento del conjunto de datos. La fila i-ésima tendrá los valores $[x_i^0, x_i^1, x_i^2] = [1, x_i, x_i^2]$. De esta forma, al multiplicarla por el vector a obtenemos la evaluación del polinomio de regresión para dichos coeficientes.

Tras componer la matrix X calcularemos su descomposición en valores singulares $X = USV^T$.

[9]:
```
# Escribimos el conjunto de datos en una matriz X
# Valores dados
x = np.array([0.0,0.25,0.5,0.75,1.0])
y = np.array([1,1.2840,1.6487,2.1170,2.7183])
n = 3

# Matriz X
X = np.vander(x,n,increasing=True)
print("Matriz de Vandermonde 5x3 para nuestras x_i: \n",X)
```

```
Matriz de Vandermonde 5x3 para nuestras x_i:
 [[1.     0.     0.    ]
 [1.     0.25   0.0625]
 [1.     0.5    0.25  ]
 [1.     0.75   0.5625]
 [1.     1.     1.    ]]
```

[10]:
```
# Descomposición en valores singulares usando numpy
u, s, vh = np.linalg.svd(X, full_matrices=True)
print("U = \n{}".format(u.round(4)))
print("\nS = {}".format(s))
print("\nV = \n{}".format(np.transpose(vh.round(4))))
print("-------------")
print("Dimensiones: ", u.shape, s.shape, vh.shape)
```

```
U =
[[-0.2945  0.6327  0.6314 -0.0258 -0.3371]
 [-0.3466  0.455  -0.2104  0.2809  0.7414]
 [-0.4159  0.1942 -0.5244 -0.6882 -0.2017]
 [-0.5025 -0.1497 -0.3107  0.6367 -0.4725]
 [-0.6063 -0.5767  0.4308 -0.2036  0.2698]]

S = [2.71168512 0.93707467 0.16268803]
```

```
V =
[[-0.7987  0.5929  0.1027]
 [-0.4712 -0.5102 -0.7195]
 [-0.3742 -0.6231  0.6869]]
-------------
Dimensiones:  (5, 5) (3,) (3, 3)
```

Hemos visto que no se devuelve la matriz S como la esperábamos, así que tenemos que construirla. Para ello, lo hacemos del siguiente modo:

[11]:
```
# Creamos la matriz que contendrá a s
S = np.zeros((5,3))

# Hacemos a S matriz diagonal y la sumamos con la matriz de ceros
S[0:len(s),0:len(s)] += np.diag(s)
print("Matriz S: \n",S)
```

```
Matriz S:
 [[2.71168512 0.         0.        ]
 [0.         0.93707467 0.        ]
 [0.         0.         0.16268803]
 [0.         0.         0.        ]
 [0.         0.         0.        ]]
```

[12]:
```
print("Matriz X: \n", X)
print("\nProducto de su descomposición, U * S * V^T: \n", u @ S @ vh)
```

```
Matriz X:
 [[1.     0.     0.    ]
 [1.     0.25   0.0625]
 [1.     0.5    0.25  ]
 [1.     0.75   0.5625]
 [1.     1.     1.    ]]

Producto de su descomposición, U * S * V^T:
 [[ 1.00000000e+00 -4.92966929e-16 -4.93853507e-16]
 [ 1.00000000e+00  2.50000000e-01  6.25000000e-02]
 [ 1.00000000e+00  5.00000000e-01  2.50000000e-01]
 [ 1.00000000e+00  7.50000000e-01  5.62500000e-01]
 [ 1.00000000e+00  1.00000000e+00  1.00000000e+00]]
```

Hemos hallado así las matrices que se nos pedían para obtener la descomposición SVD de X.

2.3 Apartado 2

Se nos indica ahora que los valores del polinomio que interpola nuestros datos pueden obtenerse como

$$a = Vz$$

donde $a = (a_0, a_1, a_2)$ son los coeficientes de $P_2(x)$ que buscamos, y $z = \left(\frac{c_1}{s_1}, \frac{c_2}{s_2}, \frac{c_3}{s_3}\right)$ sonde s_i son los valores singulares de X y c_i los tres primeros valores obtenidos de multiplicar $U^T y$.

Demostración.- Consideramos la descomposición SVD de la matriz $X = USV^T$ con U, V ortogonales. Si multiplicamos en ambos miembros de la ecuación $Xa = y$ por U^T obtenemos:

$$SV^T a = U^T y.$$

Sea ahora S^+ una matriz tal que $S^+ S = I$. Si consideramos $S^{-1} = \{s_1^{-1}, s_2^{-1}, s_3^{-1}\}$, esta matriz viene dada por la expresión:

$$S^+ = \left(S^{-1} | 0_{3\times 2}\right).$$

Basta entonces multiplicar por la izquierda por S^+ en ambos lados de la ecuación para obtener:

$$a = Vz,$$

como queríamos demostrar.

Ya tenemos V, así que vamos a obtener el vector z y a continuación los coeficientes del polinomio:

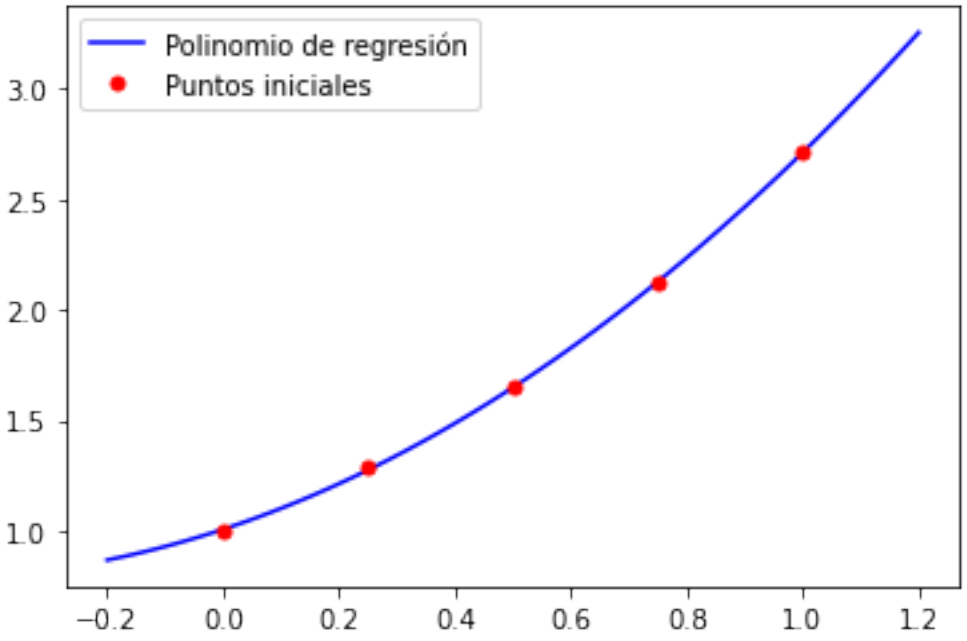
[13]:
```
# Obtenemos los c_i
c = (np.transpose(u) @ y)

print("El producto U^T*y da como resultado: \n\t", c)

# Obtenemos z
z = np.array(c[0:3]/s)
print("\nEl vector z es:\n\t",z)

# Obtenemos los coeficientes del polinomio
a = np.transpose(vh) @ z
print("\nLos coeficientes (a0,a1,a2) obtenidos del polinomio son:\n\t", a)
```

```
El producto U^T*y da como resultado:
         [-4.1372342  -0.3473217   0.00991447 -0.00539697  0.01565264]

El vector z es:
         [-1.52570598 -0.37064463  0.06094163]

Los coeficientes (a0,a1,a2) obtenidos del polinomio son:
         [1.00513714 0.86418286 0.84365714]
```

Veamos si los coeficientes obtenidos interpolan bien los datos. Vamos a dibujar el polinomio y los puntos que tenemos.

[14]:
```
P2 = lambda x : a[0] + a[1]*x + a[2]*x*x
x_vals = np.linspace(-0.2,1.2,100)
values = P2(x_vals)
plt.plot(x_vals, values, 'b',label="Polinomio de regresión")
plt.plot(x,y,'r.',markersize=10,label="Puntos iniciales")
```

```
plt.legend()
plt.show()
```



Como podemos observar, el polinomio que conseguimos es prácticamente perfecto ajustándose a nuestros datos. Sabemos que esta regresión es un ejemplo muy sencillo de una regresión lineal que se puede hacer usando *sklearn*, por lo que nos gustaría comparar los coeficientes del polinomio que hemos obtenido con los que esta librería nos proporciona. Los calculamos:

[15]:
```
from sklearn.linear_model import LinearRegression
sklearn_pol = LinearRegression(fit_intercept=False).fit(X,y)
print("Coeficientes sklearn: \n\t",sklearn_pol.coef_)
print("\nCoeficientes calculados por nosotros: \n\t", a)
```

```
Coeficientes sklearn:
        [1.00513714 0.86418286 0.84365714]

Coeficientes calculados por nosotros:
        [1.00513714 0.86418286 0.84365714]
```

Como vemos, son exactamente los mismos, lo que reafirma nuestra idea de que el polinomio ajusta bien nuestros datos. En el siguiente apartado estudiaremos la bondad del ajuste utilizando el error.

2.4 Apartado 3

Se puede demostrar que el error cometido con la regresión es $E = \sqrt{c_4^2 + c_5^2}$.

Demostración.-

Tenemos que U es una matriz ortogonal, por lo que $||U^T x||_2 = ||x||_2$ para cualquier x así que, si multiplicamos ambos miembros del error por U^T, obtenemos:

$$E = ||Xa - y|| = ||U^T Xa - U^T y||$$

descomponiendo ahora X usando su descomposición SVD obtenemos:

$$E = ||U^T USV^T a - U^T y|| = ||SV^T a - c|| = ||Sz - c|| = ||(SS^+ - I)c|| = || - (c_4, c_5)||,$$

como queríamos ver.

Vamos a calcularlo ahora empíricamente y a compararlo con el error que obtenemos al realizar el error de forma clásica, es decir haciendo

$$E = \sqrt{\sum_{i=1}^{5} (P_2(x_i) - y_i)^2}.$$

[16]:
```
E = np.sqrt(c[3]**2 + c[4]**2)
E_old = np.sqrt(np.sum((P2(x) - y)**2))
print("Error usando c4 y c5: \n\t",E)
print("\nError manual: \n\t",E_old)
print("\nDiferencia Error-Error_manual: \n\t", np.abs(E-E_old))
```

```
Error usando c4 y c5:
        0.016556949339433427

Error manual:
        0.01655694933943349

Diferencia Error-Error_manual:
        6.245004513516506e-17
```

Podemos ver dos cosas: - El error está lejos de ser cero, lo cual podíamos intuir viendo la representación. Aún así, es bajo, por lo que podemos decir que el polinomio de regresión ajusta bien nuestros datos. - Vemos que la diferencia entre ambos es ínfima, por lo que podemos decir que el ajuste está bien realizado y el error bien calculado.

2.5 Ejercicio opcional

1) ¿Qué obtenemos cuando aplicamos el método de los apartados 1 y 2 a la construcción de un polinomio de grado 4?

Debemos rehacer los pasos cambiando la matriz inicial, usando en este caso la matriz de Vandermonde de 5×5. La creamos utilizando el método de `NumPy`, realizamos su descomposición SVD y calculamos de nuevo c, z y finalmente a usando estos dos y V.

[17]:
```
n = 5

# Creamos la matriz de vandermonde
X_4 = np.vander(x,n,increasing=True)
print("Matriz de Vandermonde para el polinomio de grado 4:\n", X_4)

# Hacemos su descomposición svd
u_4, s_4, vh_4 = np.linalg.svd(X_4)

# Creamos la matriz que contendrá a s
S_4 = np.zeros((n,n))

# Hacemos a S matriz diagonal y la sumamos con la matriz de ceros
S_4 += np.diag(s_4)
print("\nMatriz S para el polinomio de grado 4:\n", S_4)
```

```
Matriz de Vandermonde para el polinomio de grado 4:
 [[1.         0.         0.         0.         0.        ]
 [1.         0.25       0.0625     0.015625   0.00390625]
 [1.         0.5        0.25       0.125      0.0625    ]
 [1.         0.75       0.5625     0.421875   0.31640625]
 [1.         1.         1.         1.         1.        ]]

Matriz S para el polinomio de grado 4:
 [[2.98659464 0.         0.         0.         0.        ]
 [0.         1.23790316 0.         0.         0.        ]
 [0.         0.         0.31721696 0.         0.        ]
 [0.         0.         0.         0.05382652 0.        ]
 [0.         0.         0.         0.         0.00435088]]
```

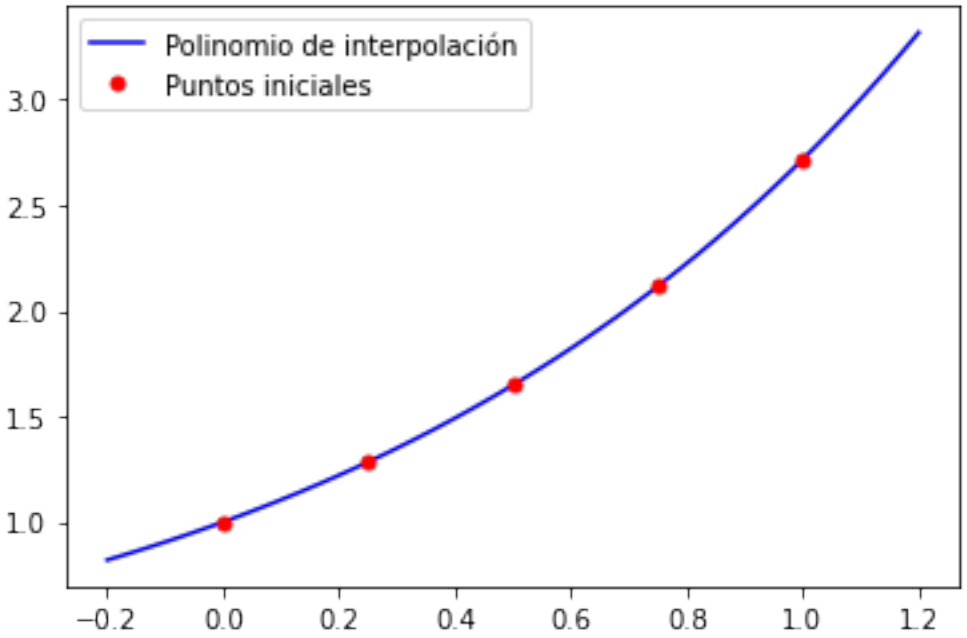
Ya.tenemos las matrices necesarias para calcular c, z y a:

[18]:
```
# Obtenemos los c_i
c_4 = (np.transpose(u_4) @ y)
print("El producto U^T y da como resultado: \n\t",c_4)
# Obtenemos z
z_4 = np.array(c_4/s_4)
print("El vector z es:\n\t",z_4)
# Obtenemos los coeficientes del polinomio
a_4 = np.transpose(vh_4) @ z_4


print("Los coeficientes (a0,a1,a2) obtenidos del polinomio son:\n\t",a_4)
```

```
El producto U^T y da como resultado:
         [-4.13888805e+00 -2.76855241e-01 -1.75114786e-01  1.78280339e-03
 -3.24371941e-04]
El vector z es:
         [-1.38582183 -0.22364855 -0.55203475  0.03312128 -0.07455322]
```

```
Los coeficientes (a0,a1,a2) obtenidos del polinomio son:
         [1.         0.99863333 0.51006667 0.14026667 0.06933333]
```

[19]:
```
P4 = lambda x : a_4[0] + a_4[1]*x + a_4[2]*x**2 + a_4[3]*x**3 + a_4[4]*x**4
x_vals = np.linspace(-0.2,1.2,100)
values = P4(x_vals)
plt.plot(x_vals, values, 'b',label="Polinomio de interpolación")
plt.plot(x,y,'r.',markersize=10,label="Puntos iniciales")
plt.legend()
plt.show()
```



Vemos que en este caso el polinomio parece ajustar mejor a los datos iniciales.

2) Comprueba que, efectivamente el polinomio obtenido se corresponde en este caso a un polinomio de interpolación en lugar de a un polinomio de regresión.

[20]:
```
E_old_4 = np.sqrt(np.sum((P4(x) - y)**2))
print("Error manual en los nodos: \n\t",E_old_4)
```

```
Error manual en los nodos:
        2.8608446115907795e-15
```

Efectivamente, el error en los nodos es equivalentemente 0, así que pasa por todos ellos y podemos decir que este polinomio es de interpolación. Este resultado era esperable, pues al tomar n puntos y un polinomio de regresión de grado $n-1$ que minimiza el error cuadrático medio, obtenemos el polinomio que pasa por dichos puntos. Esto es, el polinomio de interpolación.