Análisis numérico

Facultad de Ciencias, UNAM
Ejemplos de cancelación

Jorge Zavaleta Sánchez

Semestre 2022-1

[1]:
```
# Modulos necesarios
import numpy as np
```

Cálculo de la desviación estándar muestral

Para calcular la desviación estandar muestral se ocupa la fórmula

$$s = \sqrt{\frac{1}{N-1}\sum_{k=1}^{N}(x_k - \overline{x})^2}$$

donde

$$\overline{x} = \frac{1}{N}\sum_{k=1}^{N} x_k$$

es la media muestral y N es el tamaño de la muestra.

[2]:
```
def media(x):
    """Calcula la media muestral de x.

    Parametros:
    x - iterable. Debe ser un iterable con los valores de la muestra.

    Regresa:
    float. Valor de la media muestral calculada.
    """
    s = 0
    for k in x:
        s += k
    return s/len(x)

def devstd(x,opt = 1):
    """Calcula la desviacion estandar muestral de x, calculando primero la media␣
 ↪de x.
```

```
    Parametros:
    x - iterable. Debe ser un iterable con los valores de la muestra.
    opt - int. Si opt == 1 calcula s, si no calcula s**2.

    Regresa:
    float. Valor de la desviacion estandar muestral calculada.

    Dependencias:
    numpy - modulo. Para usar la funcion sqrt.
    media - funcion. Calcula la media muestral de x.
    """
    s,m = 0,media(x)
    for k in x:
        s += (k-m)**2
    if opt == 1:
        return np.sqrt(s/(len(x)-1))
    return s/(len(x)-1)
```

De la fórmula para s, si se desarrollan los términos y se utiliza la fórmula para $\overline{x}$, es posible reescribir la fórmula de la desviación estándar como:

$$s = \sqrt{\frac{1}{N-1}\left(\sum_{k=1}^{N} x_k^2 - N\overline{x}^2\right)} = \sqrt{\frac{1}{N-1}\left(\sum_{k=1}^{N} x_k^2 - \frac{1}{N}\left(\sum_{k=1}^{N} x_k\right)^2\right)}$$

[3]:
```
def devstd_1s(x,opt = 1):
    """Calcula la desviacion estandar muestral de x, calculando al mismo tiempo␣
 ↪la media de x.

    Parametros:
    x - iterable. Debe ser un iterable con los valores de la muestra.
    opt - int. Si opt == 1 calcula s, si no calcula s**2.

    Regresa:
    float. Valor de la desviacion estandar muestral calculada.

    Dependencias:
    numpy - modulo. Para usar la funcion sqrt.
    """
    N = len(x)
    m = sc = 0
    for k in x:
        m += k
        sc += k**2
    if opt == 1:
        return np.sqrt((sc-m**2/N)/(N-1))
    return (sc-m**2/N)/(N-1)
```

Para probar ambas funciones, se usaran diferentes muestras las cuales tienen la misma desviación estándar, pero los valores cambian en magnitud.

[4]:
```
# Muestras
X1 = np.array([4.,7.,13.,16.])
X2 = X1 + 1e8
X3 = X1 + 1e9
```

[5]:
```
# Pruebas
opt = 2
X = eval(input("Dame una muestra: "))
print(f"El resultado de la dev. std. calculado con el algoritmo de un paso es:␣
 ↪{devstd_1s(X,opt)}")
print(f"El resultado de la dev. std. calculado con el algoritmo de dos pasos es:␣
 ↪{devstd(X,opt)}")
```

```
Dame una muestra: X3
El resultado de la dev. std. calculado con el algoritmo de un paso es:
-170.66666666666666
El resultado de la dev. std. calculado con el algoritmo de dos pasos es: 30.0
```

Cálculo de las raíces reales de una ecuación de segundo grado

Considérese la ecuación de segundo grado

$$ax^2 + bx + c = 0$$

para la cual se quieren calcular sus raíces reales mediante el uso de la fórmula general

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Por lo tanto se debe clasificar las raíces a partir del discriminante $D = b^2 - 4ac$ en los siguientes casos:

- $D > 0$, se tienen dos raíces reales distintas.
- $D = 0$, se tienen una raíz real (de multiplicidad 2).
- $D < 0$, no hay raíces reales

[6]:
```
def raicescuadstd(a,b,c):
    """Calcula las raíces de una ecuación de segundo grado mediante la formula␣
 ↪general.

    Parametros:
    a - float. Coeficiente del termino cuadratico.
    b - float. Coeficiente del termino lineal.
    c - float. Coeficiente del termino constante.
```

```
    Regresa:
    x1,x2 - float. Raices de la ecuacion. Regresa NaN si el valor no existe o no␣
↪es real.

    Dependencias:
    numpy - modulo. Para usar la funcion sqrt y el valor NaN.
    """
    x1 = x2 = np.NaN
    if a == 0 and b!= 0:
        x1 = -c/b
    elif a != 0:
        D = b**2 - 4*a*c
        if D > 0:
            x1,x2 = (-b + np.sqrt(D))/(2*a),(-b - np.sqrt(D))/(2*a)
        elif D == 0:
            x1 = x2 = -b/(2*a)
    return x1,x2
```

También es posible usar la fórmula alternativa

$$x = \frac{2c}{-b \mp \sqrt{b^2 - 4ac}}$$

para calcular las raíces. Una implementación cuidadosa se logra al usar una combinación de ambas fórmulas, para que cada una de las raíces se calculen evitando la suma de números con signos opuestos. De esta forma, cuando el discriminante es positivo, *i.e.*, $b^2 - 4ac > 0$, y $b \neq 0$ se puede calcular las raíces mediante

$$x_1 = \frac{-b - \operatorname{sgn}(b)\sqrt{b^2 - 4ac}}{2a}$$

y

$$x_2 = \frac{2c}{-b - \operatorname{sgn}(b)\sqrt{b^2 - 4ac}}$$

donde

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

[7]:
```
def raicescuadmod(a,b,c):
    """Calcula las raíces de una ecuación de segundo grado mediante la␣
↪combinación de formulas.

    Parametros:
    a - float. Coeficiente del termino cuadratico.
    b - float. Coeficiente del termino lineal.
    c - float. Coeficiente del termino constante.

    Regresa:
```

```
    x1,x2 - float. Raices de la ecuacion. Regresa NaN si el valor no existe o no␣
↪es real.

    Dependencias:
    numpy - modulo. Para usar las funciones sqrt, sgn y el valor NaN.
    """
    x1 = x2 = np.NaN
    if a == 0 and b!= 0:
        x1 = -c/b
    elif a != 0:
        D = b**2 - 4*a*c
        if D > 0:
            if b != 0:
                x1,x2 = (-b - np.sign(b)*np.sqrt(D))/(2*a),(2*c)/(-b - np.
↪sign(b)*np.sqrt(D))
            else:
                x1,x2 = np.sqrt(D)/(2*a),-np.sqrt(D)/(2*a)
        elif D == 0:
            x1 = x2 = -b/(2*a)
    return x1,x2
```

[8]:
```
# Coeficientes de prueba
dat1 = (1.,-1.786737601482363,2.054360090947453e-8)
dat2 = 94906265.625,-189812534,94906268.375
```

[9]:
```
# Pruebas
a,b,c = eval(input('Dame los valores de los coeficientes: '))
x1,x2 = raicescuadstd(a,b,c)
print(f"\nLa primer raíz es: {x1}")
print(f"La segunda raíz es: {x2}")

x1,x2 = raicescuadmod(a,b,c)
print(f"\nLa primer raíz es: {x1}")
print(f"La segunda raíz es: {x2}")
```

```
Dame los valores de los coeficientes: dat1

La primer raíz es: 1.7867375899845355
La segunda raíz es: 1.1497827689943563e-08

La primer raíz es: 1.7867375899845355
La segunda raíz es: 1.1497827674657215e-08
```

Función exponencial

Para calcular la función exponencial podemos recurrir a su expasión en series de Taylor alrededor de 0,esto es

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

Haciendo una implementación simple en *python* esta quedaría de la siguiente manera:

[10]:
```
def miexp(x,n):
    """Calcula e^x mediante n términos de su serie de Taylor

    Parametros:
    x - float. Valor donde se quiere evaluar la función exponencial.
    n - int. Numero de terminos a utilizar de la serie para aproximar e^x.

    Regresa:
    float. Valor de la aproximacion de e^x.

    Dependencias:
    numpy - modulo. Para usar las funcion factorial.
    """
    return sum(list(map(lambda n:x**n/np.math.factorial(n),range(n+1))))
```

[11]:
```
n = 75
x = float(input("Dame un valor de x para evualuar e^x: "))
print(f"Utilizando numpy e^{x} = {np.exp(x)}")
print(f"Utilizando la funcion propia e^{x} = {miexp(x,n)}")
```

```
Dame un valor de x para evualuar e^x: -20
Utilizando numpy e^-20.0 = 2.061153622438558e-09
Utilizando la funcion propia e^-20.0 = 1.3078303877739301e-09
```

Al evaluar la función en valores negativos del argumento se produciran errores. Esto se puede remediar al considerar que $e^{-x} = 1/e^x$. Usando esto se puede generar una función que evite este problema.

[12]:
```
def miexp2(x,n):
    """Calcula e^x mediante n términos de su serie de Taylor

    Parametros:
    x - float. Valor donde se quiere evaluar la función exponencial.
    n - int. Numero de terminos a utilizar de la serie para aproximar e^x.

    Regresa:
    float. Valor de la aproximacion de e^x.
```

```
    Dependencias:
    numpy - modulo. Para usar las funcion factorial.
    """
    v = sum(list(map(lambda n:abs(x)**n/np.math.factorial(n),range(n+1))))
    if x < 0:
        return 1/v
    return v
```

[13]:
```
n = 75
x = float(input("Dame un valor de x para evualuar e^x: "))
print(f"Utilizando numpy e^{x} = {np.exp(x)}")
print(f"Utilizando la funcion propia e^{x} = {miexp2(x,n)}")
```

```
Dame un valor de x para evualuar e^x: -20
Utilizando numpy e^-20.0 = 2.061153622438558e-09
Utilizando la funcion propia e^-20.0 = 2.0611536224385583e-09
```

También se tiene el efecto de cancelación cuando calculamos la función exponencial cerca de cero. Por ejemplo al calcular $e^x - 1$

[14]:
```
x = float(input("Dame un valor de x para evualuar e^x-1: "))
print(f"Utilizando explicitamente e^{x}-1 = {np.exp(x)-1}")
print(f"Utilizando la funcion expm1 e^{x}- 1= {np.expm1(x)}")
```

```
Dame un valor de x para evualuar e^x-1: 1e-10
Utilizando explicitamente e^1e-10-1 = 1.000000082740371e-10
Utilizando la funcion expm1 e^1e-10- 1= 1.00000000005e-10
```

Utilizando la serie de Taylor para $e^x - 1$, esto es,

$$e^x - 1 = x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{k=1}^{\infty} \frac{x^k}{k!},$$

se puede generar una rutina que de resultados correctos.

[15]:
```
def miexpm1(x,n):
    """Calcula e^x-1 mediante n términos de su serie de Taylor

    Parametros:
    x - float. Valor donde se quiere evaluar la función exponencial.
    n - int. Numero de terminos a utilizar de la serie para aproximar e^x-1.

    Regresa:
    float. Valor de la aproximacion de e^x-1.

    Dependencias:
    numpy - modulo. Para usar las funcion factorial.
    """
    return sum(list(map(lambda n:x**n/np.math.factorial(n),range(1,n+1))))
```

[16]:
```
n = 40
x = float(input("Dame un valor de x para evualuar e^x-1: "))
print(f"Utilizando la funcion propia e^{x}-1 = {miexpm1(x,n)}")
print(f"Utilizando la funcion expm1 e^{x}- 1= {np.expm1(x)}")
```

```
Dame un valor de x para evualuar e^x-1: 1e-10
Utilizando la funcion propia e^1e-10-1 = 1.000000000005e-10
Utilizando la funcion expm1 e^1e-10- 1= 1.000000000005e-10
```