Física Numérica

Julio César Avila Torreblanca

26 de agosto del 2021

Tarea 1

1. Escriba un programa que determine los límites de underflow y overflow para Python (dentro de un factor de 2) en su computadora.

Solución:

- El siguiente algoritmo aproxima el valor del underflow a través de un ciclo while. Se definen dos variables: n contará las veces en que se repite el ciclo while y $underflow$ que será la variable que se hará pequeña en cada repetición del ciclo, hasta alcanzar la aproximación deseada.

  La variable $underflow$, después de un cierto número de repeticiones llegará a ser tan pequeña que la computadora la interpretará como cero. Cuando esto suceda, el ciclo while se romperá y el contador n tendrá el mínimo entero tal que la computadora interpreta lo siguiente:

  $$underflow \times \left(\frac{1}{2^n}\right) = 0$$

  Por lo que, para el entero $n-1$ la computadora aún interpretará la operación distinta de cero. Es decir:

  $$underflow \times \left(\frac{1}{2^{n-1}}\right) \neq 0$$

  Ese número distinto de cero, será la aproximación obtenida para el underflow de nuestra computadora por un factor de 2. A pesar de no haber obtenido el valor exacto del underflow, hemos obtenido dos cotas que contienen este valor exacto. Estas cotas son los extremos del intervalor $[2^{-(n-1)}, 2^{-n}]$.

[28]:
```
n = 0
underflow = 1

while underflow != 0:
    n+=1
    underflow /= 2


print(f"n = {n}")
print(f"Underflow = {(1/2**(n-1))}")
```

```
n = 1075
Underflow = 5e-324
```

- Ahora obtendremos una aproximación para el overflow. El algoritmo es muy similar, solo que esta vez se multiplicará por un factor de dos para que la variable $overflow$ se haga cada vez más grande, hasta que la computadora lo interprete como infinito. De esa forma, nuestro contador m guardará el mínimo entero tal que la computadora interpreta lo siguiente:

$$overflow \times (2^m) \approx \infty$$

Al ser m el mínimo entero, entonces el entero $m-1$ cumple que el resultado de la operación $overflow \times \left(2^{m-1}\right)$ es finito para la computadora y no arroja infinito. De esa forma, el programa arrojará la aproximación obtenida para el overflow para nuestra computadora por un factor de 2. Aquí tampoco hemos obtenido el valor exacto del overflow, pero nuevamente hemos encontrado que se encuentra dentro del intervalo $[2^{(m-1)}, 2^m]$.

[29]:
```
m=0
overflow = 1.0

while overflow != float('inf'):
    overflow *= 2
    m+=1

print(f"m = {m}")
print(f"Overflow = {2.0**(m-1)}")
```

```
m = 1024
Overflow = 8.98846567431158e+307
```

2. Escriba un programa y determine la precisión de máquina ϵ_m (dentro de un factor de 2) de su computadora.

Solución: El siguiente algoritmo comienza definiendo dos variables: *epsilon* (ϵ) y *uno_computacional* (1_C). La variable ϵ es donde guardaremos la precisión de máquina y 1_C será de la forma:

$$1_C = 1 + \epsilon$$

Depués de la definición de las variables, usamos un cilo while que se repetirá continuamente mientras se cumpla que $1_C \neq 1.0$. Para que se cumpla la igualdad y se rompa el ciclo while, es necesario que la computadora interprete a ϵ como cero. Por lo que la acción del ciclo while es dividir la variable ϵ por la mitad en cada repetición y aplicar la definición de uno computacional: $1_C = 1 + \epsilon$, hasta que llegue el momento en que $\epsilon \approx 0$. Cuando eso suceda, habremos encontrado la precisión de máquina dentro de un factor de 2 de nuestra computadora.

[42]:
```
epsilon = 1.0
uno_computacional = 1.0 + epsilon

while uno_computacional != 1.0:
    epsilon = epsilon/2
    uno_computacional = 1 + epsilon
```

```
print(f"Epsilon = {epsilon}" )
```

```
Epsilon = 1.1102230246251565e-16
```

3. Considere la serie infinita para $\sin x$:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}$$

El problema consiste en desarrollar un programa que calcule $\$\sin x\$ para x < 2\pi$ y $x > 2\pi$, con un error absoluto menor a una parte en 10^8.

(a). Escriba un programa que calcule $\sin x$. Presente los resultados en una tabla con títulos N, $suma$ y $\left|\frac{suma - \sin x}{\sin x}\right|$, donde $\sin x$ es la función correspondiente de Python. Note que la última columna es el error relativo de su cálculo. Realice el cálculo de la suma inteligentemente (sin factoriales) e inicie con una tolerancia (error absoluto) de 10^{-8}, compare con el error relativo.

(b). Utilice la identidad $\sin(x + 2n\pi) = \sin x$ para calcular $\sin x$ para valores grandes de x ($x > 2\pi$).

(c). Ponga ahora su nivel de tolerancia menor a la precisión de máquina y vea cómo esto afecta su cálculo.

Solución:

- Necesitaremos de la libreria math para tener acceso a la función $\sin x$ de python y al número π.

[31]:
```
import math
```

Lo siguiente será crear una función llamada *coef_n* que nos permita carcular los coeficientes $a_n(x)$ de la serie:

$$\sin x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} = \sum_{n=1}^{\infty} a_n(x)$$

Para ello utilizaremos la siguiente fórmula de recursión vista en clase:

$$a_n(x) = (-1)\frac{x^2}{(2n-1)(2n-1)} a_{n-1}(x)$$

Note que esta función no está definida para $n = 1$ y $n = 2$. Por lo que deberemos usar la definición de $a_n(x)$ en el desarrollo en series de potencias de $\sin x$ para calcular esos términos.

Realizando ese proceso, encontramos que $a_1(x) = x$ y $a_2(x) = -\frac{x^3}{3!}$. A partir de estos dos términos determinaremos el resto.

Para crear la función que calcule el coeficiente $a_n(x)$ con n y x dados, usaremos recursividad dentro de la función. Es decir, la función se llamará a si misma.

El código de la función es el siguiente:

[32]:
```
def coef_n(n: int,x: float ) -> float:
    """Esta función calcula recursivamente el n-ésimo coeficiente
    del desarrollo en series de potencias de la función sen(x)"""
```

```
    a_n=0
    if x == 0.0: #Si x=0, entonces todo vale cero
        return a_n

    if n ==1:
        a_n = x
        return a_n

    if n == 2:
        a_n = -x**3/6
        return a_n


    if n > 2:
       a_n = -(x**2)/( (2*n - 2) * (2*n - 1) ) * coef_n(n-1,x)

    return a_n
```

Los parámetros de la función son n (*int*) que denota el índice del término a_n a ser calculado y x (*float*), el cual es el argumento de cual se quiere obtener $\sin x$. Además, esta función regresará un *float*.

En la función, primeramente comenzamos definiendo la variable $a_n = 0$. En esta variable almacenaremos el valor numérico del n -ésimo término del desarrollo en series de potencias de $\sin x$. Después, agregamos cuatro sentencias *if*, donde se consideran los casos para $x = 0$, $n = 1$, $n = 2$ y $n > 2$. La razón de $x = 0$ es que en ese valor $\sin x = 0$, y debemos considerarlo aparte porque más adelante podríamos tener problemas de que se divida por cero. Los casos $n = 1$ y $n = 2$, como ya se dijo, no están definidos en nuestra fórmula de recursión así que deben ser considerados aparte. Para el caso $n > 2$ deberemos usar nuestra fórmula de recursión aquí para obtener el resto de los términos. Finalmete, esta función nos arroja el valor numérico de $a_n(x)$ (*float*).

Ahora crearemos una función *suma* que calcule la suma de los primeros n términos $a_n(x)$ en el desarrollo de potencias de $\sin x$. La función es la siguiente:

[33]:
```
def suma(n: int, x = 1.0)-> float:
    """Esta función suma los primeros n términos del desarrollo en
    serie de potencias de la función sen(x)"""
    suma = 0.0

    if x == 0.0:
      return suma

    for i in range(n):
        suma += coef_n(i+1,x)
    return suma
```

Los parámetros de esta función al igual son n (*int*) y x (*float*). Denotan lo mismo que la primera función. Primeramente comenzamos definiendo a la variable *suma*, donde almacenaremos la sumatoria de los n términos $a_n(x)$. Note que hemos considerando aparte el caso cuando $x = 0$, ya que

de otra forma estaríamos sumando ceros infinitamente y eso nos traería problemas. De esta forma, cuando $x = 0$ la función arroja $suma = 0$.

Si $x \neq 0$, debemos realizar las n evaluaciones para cada $a_n(x)$. Por lo que, a través de un ciclo *for* llamamos a la función *coef_n* para hacer la evaluación de los n términos y cada valor obtenido se lo sumamos a la variable *suma*. Cuando termine de hacer todas las evaluaciones y sumas, la función nos arrojará la variable *suma* que es un *float* y contendrá el valor numérico de la siguiente expresión:

$$suma = \sum_{m=1}^{n} a_m(x)$$

Lo siguiente que haremos será definir una función llamada *n_lim*. El objetivo de esta función es estimar el valor del mínimo entero n, el cual denotará el número de términos que deben considerarse en el desarrollo en serie de potencias de la función sin x para que la aproximación buscada tenga un error absoluto menor a 10^{-8}. En otras palabras, encontraremos el mínimo entero N tal que:

$$\left| \frac{a_N(x)}{suma(N-1)} \right| \leq 10^{-8}, \quad \text{donde:} \quad suma(N) = \sum_{n=1}^{N} a_n(x)$$

El código de la función es el siguiente:

[34]:
```
def n_lim(x = 1.0, n = 1) ->int:
    """Esta función obtiene el número n que hace referencia a la cantidad
    de términos que debe considerarse en el desarrollo en series de potencias
    de la función sen(x) para tener una aproximación con error absoluto menor a
    una parte en 10^8"""

    if x == 0.0:
        return 0

    while True:
        try:
            if abs(coef_n(n,x)/suma(n-1, x)) > 10**(-8):
                n += 1
            else:
                break
        except ZeroDivisionError:
            n += 1
    return n
```

Los parámetros de esta función son el argumento x (*float*) y el entero n (*int*). Note que nuevamente se ha considerado el caso cuando $x = 0$, pues si esto pasa no es necesario hacer ninguna evaluación y la función nos regresa el entero 0.

Cuando $x \neq 0$, a través de un ciclo while evaluamos la condición:

$$\left| \frac{a_n(x)}{suma(n-1)} \right| > 10^{-8}$$

Donde se ha llamado a la función *suma* que creamos previamente. Si es verdadera esta condición, entonces el valor de n aumenta en 1 hasta encontrar el valor en el que n ya no cumpla la condición

anterior. Cuando se encuentre este número, el ciclo while se romperá y nos arrojará el entero n que cumple la condición:

$$\left|\frac{a_n(x)}{suma(n-1)}\right| \leq 10^{-8}$$

Note que se ha usado una sentencia *try-except*. Esto se debe a que en la condición evaluada, la función $suma(n-1,x)$ puede valer cero para algunos valores de n. Al estar dividiendo, podría generar errores en el progragra. Afortunadamente, la sentencia *try-except* permite que el programa no se dentenga con la aparición de errores. En este caso, el error que queremos evitar se llama *ZeroDivisionErro*. Si pasa este error, le pedimos al programa que aumente en 1 el valor de n y nuevamente se ejecuta el while.

Ya con las funciones *n_lim*, *suma* y *coef_n* podemos estimar el valor de $\sin x$ para algún x dado. Para ello creemos la siguiente función llamada *sen_serie_potencias*. EL código es el siguiente:

[35]:
```
def sen_serie_potencias(x: float)->float:
    """Esta función obtiene una aproximación para sex(x) mediante el
    desarrollo por serie de potencias"""



    if abs(x) >= 2 * math.pi:
        x2 = math.floor(abs(x/(2*pi))) #este me da el entero 2n pi
        if x>0:
            x = x - 2*x2*pi
        else:
            x = x + 2*x2*pi

    n = n_lim(x)
    senx = math.sin(x) #Esta variable nos almacenará el valor de sinx de python␣
↪para obtener el error relativo
    p1="N"
    p2="suma(N)"
    p3="error relativo"
    print("Error rabsolto menor a 10^(-8):")
    print(f"{p1:>3} | {p2:^25} | {p3:^25}") #Esto solo nos imprime los títulos␣
↪de la tabla
    for i in range(1,n+1):
        s = suma(i,x)

        if x == 0:
            error = 0
        else:
            error = abs((s - senx)/senx)
        print(f"{i:3d} | {s:25.15f} | {error:25.15f}")#Esto imprime los valores␣
↪de la tabla
```

El parámetro de esta función es el valor de x (*float*), del cual se quiere calcular el $\sin x$. Esta función no regresará nada, solamente imprimirá en pantalla las aproximaciones hechas en forma de tabla

como se pide en el programa.

Dentro de la función, primeramente consideramos el caso para cuando $x > 2\pi$. Si esto sucede, usaremos la identidad:

$$\sin x = \sin(x + 2n\pi), \quad |x| < 2\pi$$

Por lo que debemos estimar el valor de n para así obtener la x. Para obtener n, usamos un *if* donde se evalua la condición $|x| \geq 2\pi$. Si esto se cumple, entonces debemos descomponer el argumento. Para ello, notemos lo siguiente:

$$(x + 2n\pi) \times \frac{1}{2\pi} = \frac{x}{2\pi} + n$$

Y como $|x| < 2\pi$, entonces:

$$\frac{|x|}{2\pi} < 1$$

Así, la expresión $|\frac{x}{2\pi} + n| \geq 0$ será la suma de un entero más un número fraccionario. Por lo que, si aplicamos la función piso a esa expresión, obtendremos el valor de la n buscada.

Ya con el valor de n, simplemente le restamos $2n\pi$ al parámetro inicial de esta función y obtendremos el valor de x en la identidad $\sin x = \sin(x + 2n\pi)$. Solo nos resta usar todas las funciones ya creadas para obtener el valor de $\sin x$.

Lo siguiente en esta función, es obtener el entero que nos dirá la cantidad de términos a considerar en la serie de potencias. Para ello llamaremos a la función *n_lim* y lo almacenaremos en la variable entera n. Enseguida, con ayuda de un ciclo for haremos las n evaluaciones a través de la función *suma*. Para obtener el error relativo, debemos considerar el caso cuando $x = 0$, ya que en la expresión $\left|\frac{suma - \sin x}{\sin x}\right|$ se dividiría por cero y habría problemas. Afortunadamente, en la creación de nuestro algoritmo siempre tuvimos en mente ese caso. Por lo que el error relativo será cero cuando eso suceda. Si $x \neq 0$, entonces se calcula el error relativo en cada evaluación de la función *suma* y se imprime en pantalla la tabla pedida para ver como convergen los valores.

A continuación podemos ver un ejemplo donde hemos estimado el valor de $\sin\left(-\frac{33\pi}{2}\right) = \sin\left(-\frac{\pi}{2} - 16\pi\right)$.

[43]:
```
pi = math.pi
x = - (33*pi/2)
sen_serie_potencias(x)
```

```
Error rabsolto menor a 10^(-8):
  N |          suma(N)          |      error relativo
  1 |       -1.570796326794898 |         0.570796326794898
  2 |       -0.924832229288650 |         0.075167770711350
  3 |       -1.004524855534817 |         0.004524855534817
  4 |       -0.999843101399499 |         0.000156898600501
  5 |       -1.000003542584286 |         0.000003542584286
  6 |       -0.999999943741051 |         0.000000056258949
  7 |       -1.000000000662780 |         0.000000000662780
  8 |       -0.999999999993977 |         0.000000000006023
```

Como podemos ver, el valor del entero n donde se ha cortado la serie de potencia es 8, ya que solo se hicieron 8 evaluaciones. En la segunda columna podemos ver que entre más evalauciones

hacemos, más nos acercamos al valor de $\sin\left(-\frac{33\pi}{2}\right) = -1$. Esto lo podemos confirmar en la tercer columna, pues el error relativo se hace cada vez más y más pequeño.

Con lo anterior, ya hemos resuelto los incisos (a) y (b). Para el inciso (c) solo debemos modificar la función *n_lim* que es donde se ha evaluado la condición del error abosluto. Esta vez es necesario encontrar un n tal que cumpla la condición:

$$\left|\frac{a_n(x)}{suma(n-1)}\right| \leq \epsilon_C$$

Donde $\epsilon_C = 1.1102230246251565 \times 10^{-16}$ es el error computacional estimado en el ejercicio 2.

La función *n_lim* modificada es:

[37]:
```
epsilon = 1.0
uno_computacional = 1.0 + epsilon

while uno_computacional != 1.0:
    epsilon = epsilon/2
    uno_computacional = 1 + epsilon

def n_lim_modificada(x = 1.0, n = 1) ->int:
    """Esta función obtiene el número n que hace referencia a la cantidad
    de términos que debe considerarse en el desarrollo en series de potencias
    de la función sen(x) para tener una aproximación con error absoluto menor a
    una parte en 10^8"""

    if x == 0.0:
        return 0

    while True:
        try:
            if abs(coef_n(n,x)/suma(n-1, x)) > epsilon: #esta es la única parte␣
 ↪modificada
                n += 1
            else:
                break
        except ZeroDivisionError:
            n += 1
    return n
```

Note que nuevamente hemos tenido que obtener el valor de ϵ_C para guardarlo en una variable que luego será utilizada en la función *n_lim_modificada*.

Otra función que debemos modificar es la función *sen_serie_potencias* para que use la función modificada anteriormente. El nuevo código es:

[40]:
```
def sen_serie_potencias_modificada(x: float)->float:
    """Esta función obtiene una aproximación para sex(x) mediante el
    desarrollo por serie de potencias"""
```

```
    if abs(x) >= 2 * math.pi:
        x2 = math.floor(abs(x/(2*pi))) #este me da el entero 2n pi
        if x>0:
            x = x - 2*x2*pi
        else:
            x = x + 2*x2*pi

    n = n_lim_modificada(x)#parte modificada
    senx = math.sin(x) #Esta variable nos almacenará el valor de sinx de python␣
↪para obtener el error relativo
    p1="N"
    p2="suma(N)"
    p3="error relativo"
    print(f"Error absoluto menor a {epsilon} (error de máquina):")
    print(f"{p1:>3} | {p2:^25} | {p3:^25}") #Esto solo nos imprime los títulos␣
↪de la tabla
    for i in range(1,n+1):
        s = suma(i,x)

        if x == 0:
            error = 0
        else:
            error = abs((s - senx)/senx)
        print(f"{i:3d} | {s:25.15f} | {error:25.15f}")#Esto imprime los valores␣
↪de la tabla
```

Haremos el mismo ejemplo de $\sin\left(-\frac{33\pi}{2}\right) = -1$ para ver como cambia.

[41]:
```
pi = math.pi
x = - (33*pi/2)
sen_serie_potencias_modificada(x)
```

```
Error absoluto menor a 1.1102230246251565e-16 (error de máquina):
  N |          suma(N)           |      error relativo
  1 |        -1.570796326794898 |         0.570796326794898
  2 |        -0.924832229288650 |         0.075167770711350
  3 |        -1.004524855534817 |         0.004524855534817
  4 |        -0.999843101399499 |         0.000156898600501
  5 |        -1.000003542584286 |         0.000003542584286
  6 |        -0.999999943741051 |         0.000000056258949
  7 |        -1.000000000662780 |         0.000000000662780
  8 |        -0.999999999993977 |         0.000000000006023
  9 |        -1.000000000000044 |         0.000000000000044
 10 |        -1.000000000000000 |         0.000000000000000
```

```
11 |        -1.000000000000000 |         0.000000000000000
12 |        -1.000000000000000 |         0.000000000000000
```

Como podemos ver, esta vez se hicieron 12 evaluaciones. Esto permitió que fueramos más precisos en nuestra aproximación, hasta el grado en el que la computadora no puede agregar más números decimales e imprime que nuestro error relativo es cero. Por lo tanto, esta modificación ha permitido ser más precisos en nuestra aproximación.