Tarea 1

Universidad de Costa Rica I Ciclo, 2024

Sede de Occidente Sección de Matemática MA-0323 Métodos Númericos

Profesores:

Jessica Jimenez Moscoso
Adrian Moya Fernandez

Estudiantes:

Steven Sánchez López, B87331
Javier Acuña Mora, A60057
Juan Ignacio Orozco Zamora, C05688

1. Fórmula de Stirling para cálculo de factoriales

1.1 Contexto:

1. La fórmula de Stirling, aproxima $n!$ de la siguiente manera:

$$n! \approx n^n \cdot e^{-n} \cdot \sqrt{2\pi \cdot n}$$

   $a)$ Escriba un algoritmo para aproximar los valores de $n!$ con la fórmula de Stirling.

   $b)$ Utilice el algoritmo de la parte **a.** para aproximar $n!$ con $n = 0, 1, ..., 25$. Organice los resultados de la forma:

n	$n!$	Aproximación de Stirling	Error Absoluto	Error Relativo

   $c)$ Analice los resultados

1.2 Librerías necesarias para trabajar.

In [1]:
```
import math
import pandas as pd
```

1.3 Implementación de las funciones necesarias para calcular la fórmula de Stirling así como las funciones para calcular los errores absolutos y relativos.

La función aproximación_stirling calcula la aproximación definida en el ejercicio 1, según la premisa indicada.

La función `error_absoluto()` calcula el error absoluto de la aproximación.

La función `error_relativo()` calcula el error relativo de la aproximación.

nf es la variable para almacenar el número factorial.

Se define el rango que incluya de 0 a 25 según lo indicado en el inciso 2.

La condición de **x == 0** se incluye para evitar una excepción que se genera por división entre 0.

Para la condición **x == 1** se incluye para evitar que la definición del valor de nf, no se mantenga de manera permanente en 0 y calcule de manera correcta los subsecuentes valores factoriales.

In [2]:
```
def aproximacion_stirling(n):
  return math.pow(n, n) * math.pow(math.e, -n) * math.sqrt(2 * math.pi * n)

def error_absoluto(real, aprox):
  return abs(real - aprox)

def error_relativo(error_absoluto, real):
  try:
    return abs(error_absoluto) / abs(real)
  except ZeroDivisionError:
    return None

def calculate_stirling(n):
  nf = 0
  data = []
  for x in range(n+1):
    if x == 0 or x == 1:
      nf = x
    nf *= x
    aprox = aproximacion_stirling(x)
    error_abs = error_absoluto(nf, aprox)
    error_rel = error_relativo(error_abs, nf)
    row = {'n': x, 'n!': nf, 'Aproximación de Stirling': aprox,
           'Error Absoluto': error_abs,
           'Error Relativo': error_rel}
    data.append(row)
  df = pd.DataFrame(data)
  return df
```

1.4 Input de valores para el funcionamiento de la función principal `calculate_stirling()` definiendo que el rango 0 < x < 25.

In [3]:
```
df = calculate_stirling(25)

df
```

Out[3]:

	n	n!	Aproximación de Stirling	Error Absoluto	Error Relativo
0	0	0	0.000000e+00	0.000000e+00	NaN
1	1	1	9.221370e-01	7.786299e-02	0.077863
2	2	2	1.919004e+00	8.099565e-02	0.040498
3	3	6	5.836210e+00	1.637904e-01	0.027298
4	4	24	2.350618e+01	4.938249e-01	0.020576
5	5	120	1.180192e+02	1.980832e+00	0.016507
6	6	720	7.100782e+02	9.921815e+00	0.013780
7	7	5040	4.980396e+03	5.960417e+01	0.011826
8	8	40320	3.990240e+04	4.176045e+02	0.010357
9	9	362880	3.595369e+05	3.343127e+03	0.009213
10	10	3628800	3.598696e+06	3.010438e+04	0.008296
11	11	39916800	3.961563e+07	3.011749e+05	0.007545
12	12	479001600	4.756875e+08	3.314114e+06	0.006919
13	13	6227020800	6.187239e+09	3.978132e+07	0.006389
14	14	87178291200	8.666100e+10	5.172895e+08	0.005934
15	15	1307674368000	1.300431e+12	7.243646e+09	0.005539
16	16	20922789888000	2.081411e+13	1.086755e+11	0.005194
17	17	355687428096000	3.539483e+14	1.739099e+12	0.004889
18	18	6402373705728000	6.372805e+15	2.956908e+13	0.004618
19	19	121645100408832000	1.211128e+17	5.323138e+14	0.004376
20	20	2432902008176640000	2.422787e+18	1.011516e+16	0.004158
21	21	51090942171709440000	5.088862e+19	2.023248e+17	0.003960
22	22	1124000727777607680000	1.119751e+21	4.249233e+18	0.003780
23	23	25852016738884976640000	2.575853e+22	9.349137e+19	0.003616
24	24	620448401733239439360000	6.182979e+23	2.150475e+21	0.003466
25	25	15511210043330985984000000	1.545959e+25	5.161521e+22	0.003328

1.5 Análisis de los resultados:

Según se identifica en los datos reflejados el algoritmo de stirling aproxima con cierta amplitud que va creciendo conforme el valor de referencia sea más alto.

Esa diferencia del crecimiento de amplitud se demuestra de manera clara en el valor absoluto que se vuelve más grande conforme se utiliza un número más alto que el anterior donde la aproximación no es tan cercana.

Caso contrario sucede con el error relativo, que conforme crece el valor de referencia la aproximación va mejorando, pero a un ritmo muy lento al contrario del escenario del error absoluto que crece de manera bastante marcada.

2. Factorización PALU

2.1 Contexto:

2. Escriba un algoritmo que determine las matrices de tamaño $n \times n$:

   - P: matriz de permutación
   - L: triangular inferior
   - U: triangular superior

   Al ingresar una matriz A, de manera que

   $$P \cdot A = L \cdot U$$

   Aplique dicho algoritmo a la matriz

$$A = \begin{pmatrix} 2 & -1 & 4 & 1 & -1 \\ -1 & 3 & -2 & -1 & 2 \\ 5 & 1 & 3 & -4 & 1 \\ 3 & -2 & -2 & -2 & 3 \\ -4 & -1 & -5 & 3 & -4 \end{pmatrix}.$$

2.2 Librerías necesarias para trabajar.

In [1]:
```
#Se importa numpy la cual
#es una libreria de python para
# realizar calculos numericos entre otras cosas
import numpy as np

#Librería tabulate para mostrar los datos en forma de tabla.
from tabulate import tabulate
```

2.3 Implementación de la función para calcular la factorización PALU de una matriz cuadrada.

El código realiza la factorización PA=LU, utilizando la función establecida `perform_PLU_decomposition()`. Primeramente en este método se busca el pivote, se hace intercambio de filas, luego se busca el factor que será el valor que se multiplicará por cada entrada de una respectiva fila y se suma a la siguiente, y esto se vuelve a empezar y se

realizará n-1 cantidad de veces. (n-1 es porque las matrices empiezan desde el 0 no en 1) Una vez finalizado el ciclo hasta n-1, se imprimen la matriz original (A), U, L y P.

Para verificar que el P.A = L.U, con el método np.dot() de la librería numpy, se multiplican las matrices por lo tanto np.dot(P,A) y np.dot(L,U) deben dar el mismo resultado.

In [2]:
```
def perform_PLU_decomposition(A):
    tamano = np.shape(A)
    n = tamano[0]
    L = np.identity(n, dtype=float)
    P = np.identity(n)
    U = np.copy(A)

    for i in range(0, n-1):
        columna = abs(U[i:, i])
        nMayor = np.argmax(columna)
        if nMayor != 0:
            tempU = np.copy(U[i, :])
            tempP = np.copy(P[i, :])
            U[i, :] = U[nMayor+i, :]
            P[i, :] = P[nMayor+i, :]
            U[nMayor+i, :] = tempU
            P[nMayor+i, :] = tempP
            for z in range(i):
                tempL = L[i, z]
                L[i, z] = L[nMayor+i, z]
                L[nMayor+i, z] = tempL
        pivote = U[i, i]
        siguiente = i+1
        for k in range(siguiente, n, 1):
            factor = U[k, i] / pivote
            U[k, :] = U[k, :] - U[i, :] * factor
            L[k, i] = factor

    print("Matriz Original A: ")
    print(tabulate(A, tablefmt="fancy_grid"))
    print('Matriz U: ')
    print(tabulate(U, tablefmt="fancy_grid"))
    print('matriz L: ')
    print(tabulate(L, tablefmt="fancy_grid"))
    print('matriz P: ')
    print(tabulate(P, tablefmt="fancy_grid"))
    print("PA")
    print(tabulate(np.dot(P, A), tablefmt="fancy_grid"))
    print("LU")
    print(tabulate(np.dot(L, U), tablefmt="fancy_grid"))
```

2.4 Inputa de valores para probar el funcionamiento de la función `perform_PLU_decomposition()`.

In [3]:
```
#Valores por defecto de la matriz A,
# estos valores se pueden
# cambiar por cualquier otra matriz cuadrada.
```

```
A = np.array([[2,-1,4,1,-1],
              [-1,3,-2,-1,2],
              [5,1,3,-4,1],
              [3,-2,-2,-2,3],
              [-4,-1,-5,3,-4]], dtype=float)
perform_PLU_decomposition(A)
```

Matriz Original A:

2	-1	4	1	-1
-1	3	-2	-1	2
5	1	3	-4	1
3	-2	-2	-2	3
-4	-1	-5	3	-4

Matriz U:

5	1	3	-4	1
0	3.2	-1.4	-1.8	2.2
0	0	-4.9375	-1.0625	4.1875
0	0	0	1.34177	1.41772
0	0	0	0	-5.62264

matriz L:

1	0	0	0	0
-0.2	1	0	0	0
0.6	-0.8125	1	0	0
0.4	-0.4375	-0.443038	1	0
-0.8	-0.0625	0.544304	0.198113	1

matriz P:

0	0	1	0	0
0	1	0	0	0
0	0	0	1	0
1	0	0	0	0
0	0	0	0	1

PA

5	1	3	-4	1
-1	3	-2	-1	2
3	-2	-2	-2	3

2	-1	4	1	-1
-4	-1	-5	3	-4

LU

5	1	3	-4	1
-1	3	-2	-1	2
3	-2	-2	-2	3
2	-1	4	1	-1
-4	-1	-5	3	-4

3. Método iterativo de bisección

3. 1 Contexto:

En el año 1225 Leonardo de Pisa estudió la ecuación

$$x^3 + 2x^2 + 10x - 20 = 0$$

y obtuvo $x = 1{,}368808107$. Nadie sabe con que método logró obtener ese valor, pero es una buena aproximación de la solución teniendo en cuenta que tiene 9 digitos decimales exactos.

3.2 Librerías necesarias para trabajar

In [13]:
```
import pandas as pd
```

3.3 Implementación del método de bisección para encontrar la raíz de una función.

Argumentos:

- funcion (str): La función como una cadena de texto que se evaluará utilizando la función interna `eval()` de Python.
- a (float): El extremo izquierdo del intervalo inicial.
- b (float): El extremo derecho del intervalo inicial.
- tol (float): La tolerancia para la convergencia.

Retornos:

- (dictionary): Un diccionario que contiene una clave (por ejemplo 'a') y un valor que tendrá la lista de valores para el criterio durante toda la ejecución.

Este diccionario se utilizará para crear un cuadro de datos con la librería pandas y mostrar los resultados.

In [14]:
```
def bisection(funcion, a, b, tol):

    #Función que obtiene la imagen a partir de una función definida.
    def f(x):
        return eval(funcion)

    #Cálculo inicial del error.
    error = abs(b - a)

    #Vector para almacenar los valores de a, b, c
```

```
    # y el error luego de cada iteración.
    data = []

    #El criterio del ciclo consiste en que la ejecución seguirá hasta
    # que se sobrepase la tolerancia definida.
    while error > tol:

        #Cálculo del punto medio
        c = (b + a) / 2

        if f(a) * f(b) >= 0:
            return None  # No se encontró una raíz -
        #caso de error, la función se detiene.

        if f(c) == 0:
            return (0, c, c)  # Raíz exacta encontrada

        if f(c) * f(a) < 0:
            #El punto medio se convierte en el extremo
            # derecho del intervalo en desarollo.
            b = c
        else:
            #El punto medio se convierte en el extremo
            # izquierdo del intervalo en desarollo.
            a = c

        #Para todas las operaciones se cálcula el error
        # de manera que el ciclo pueda cerrarse
        # en algún momento de la ejecución.
        error = abs(b - a)

        #Se agregan los elementos a (extremo izquierdo),
        # b (extremo derecho), c (punto medio)
        # y el error al vector que los almacena.
        row = {'A': a, 'B': b, 'f(Pn)': c, 'Error': error}
        data.append(row)

    return pd.DataFrame(data)
```

3.4 Input de valores para probar el correcto funcionamiento de la función principal de biyección

In [15]:

```
function = "x**3 + 2*x**2 + 10*x - 20" #función a evaluar
a = -10 #extremo izquierdo del intervalo
b = 10 #extremo derecho del intervalo
tol = 1e-9 #tolerancia
result = bisection(function, a, b, tol) #Llamado a la función

#Se agrega una columna con el número de iteración
result.insert(0, 'Número de iteración', result.index)


#Configuración necesaria para mostrar los resultados con 9 decimales.
pd.set_option('display.float_format', '{:.9f}'.format)
```

```
result
```

Out[15]:

	Número de iteración	A	B	f(Pn)	Error
0	0	0.000000000	10.000000000	0.000000000	10.000000000
1	1	0.000000000	5.000000000	5.000000000	5.000000000
2	2	0.000000000	2.500000000	2.500000000	2.500000000
3	3	1.250000000	2.500000000	1.250000000	1.250000000
4	4	1.250000000	1.875000000	1.875000000	0.625000000
5	5	1.250000000	1.562500000	1.562500000	0.312500000
6	6	1.250000000	1.406250000	1.406250000	0.156250000
7	7	1.328125000	1.406250000	1.328125000	0.078125000
8	8	1.367187500	1.406250000	1.367187500	0.039062500
9	9	1.367187500	1.386718750	1.386718750	0.019531250
10	10	1.367187500	1.376953125	1.376953125	0.009765625
11	11	1.367187500	1.372070312	1.372070312	0.004882812
12	12	1.367187500	1.369628906	1.369628906	0.002441406
13	13	1.368408203	1.369628906	1.368408203	0.001220703
14	14	1.368408203	1.369018555	1.369018555	0.000610352
15	15	1.368713379	1.369018555	1.368713379	0.000305176
16	16	1.368713379	1.368865967	1.368865967	0.000152588
17	17	1.368789673	1.368865967	1.368789673	0.000076294
18	18	1.368789673	1.368827820	1.368827820	0.000038147
19	19	1.368789673	1.368808746	1.368808746	0.000019073
20	20	1.368799210	1.368808746	1.368799210	0.000009537
21	21	1.368803978	1.368808746	1.368803978	0.000004768
22	22	1.368806362	1.368808746	1.368806362	0.000002384
23	23	1.368807554	1.368808746	1.368807554	0.000001192
24	24	1.368807554	1.368808150	1.368808150	0.000000596
25	25	1.368807852	1.368808150	1.368807852	0.000000298
26	26	1.368808001	1.368808150	1.368808001	0.000000149
27	27	1.368808076	1.368808150	1.368808076	0.000000075
28	28	1.368808076	1.368808113	1.368808113	0.000000037
29	29	1.368808094	1.368808113	1.368808094	0.000000019

	Número de iteración	A	B	f(Pn)	Error
30	30	1.368808104	1.368808113	1.368808104	0.000000009
31	31	1.368808104	1.368808108	1.368808108	0.000000005
32	32	1.368808106	1.368808108	1.368808106	0.000000002
33	33	1.368808107	1.368808108	1.368808107	0.000000001
34	34	1.368808108	1.368808108	1.368808108	0.000000001

3.5 Análisis de los resultados luego de la ejecución.

Observando la salida y los resultados podemos concluir que en la iteración número 33 (zero indexed) se llega al valor encontrado por Leonardo de Pisa con exactamente 9 decimales.

Por otra parte, se puede analizar que el código y la ejecución termina cuando el nivel de tolerancia ha sido superado en la condición establecida.