Notebook

May 18, 2025

ESCUELA POLITÉCNICA NACIONAL

Facultad de Ingeniería en Sistemas

Ingeniería en Ciencias Computacionales

Métodos Numéricos

Estudiante:

Estéfano Condoy

Docente:

Carlos Ayala

Semestre:

25-A

U2 Preliminares A

0.1 Métodos Analíticos

El método analítico nos da soluciones exactas. Nota:Todas mis variables y funcione stendran Ec(Para hacer referencia a mi nombre Estéfano Condoy)

[23]:
```
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import sympy as sp

# Definimos la variable, en este caso la 'e'
e = sp.symbols('e')

# Definimos la ecuación a resolver
EcEcuacion = 3 + e**2 - 4*e

# Resolvemos la ecuación de manera analitica
EcSoluciones = sp.solve(EcEcuacion, e)

# Imprimimos la solución de nuestra ecuación
print("Las soluciones son:", EcSoluciones)
```

```
Las soluciones son: [1, 3]
```

0.2 Métodos Númericos

Permite obtener soluciones cercanas para problemas cuya resolución exacta no es posible mediante métodos analíticos, es decir es un aproximado, se basa en calculos iterativos (Por ejemplo, método de Newton,Secante,Bisección, etc). Para este codigo del ejemplo, opte por investigar otro metodo aparte de los aprendido y escogi el **Método numérico de la regla del trapecio.**

[24]:
```
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

# Ponemos los parámetros del intervalo y el numero de trapecios.
EcA = 0          # Límite inferior de integración
EcB = 1          # Límite superior de integración
EcN = 100        # Número de trapecios para dividir el intervalo
```

```python
# Definimos la función a integrar: f(x) = e^(-x^2)
def EcFuncion(EcX):
    return math.exp(-EcX**2)

# Implementamos la regla del trapecio
def EcReglaTrapecio(EcA, EcB, EcN):
    # Calculamos la base de cada trapecio
    EcH = (EcB - EcA) / EcN

    # Inicializamos la suma con la mitad de la función evaluada en los extremos
    EcSuma = 0.5 * (EcFuncion(EcA) + EcFuncion(EcB))

    # Recorremos cada punto interno del intervalo [a,b]
    for EcI in range(1, EcN):
        # Calculamos la posición del punto i-ésimo
        EcXI = EcA + EcI * EcH
        # Sumamos el valor de la función en ese punto a la suma total
        # Estos puntos tienen peso completo (1) en la fórmula
        EcSuma += EcFuncion(EcXI)

    # Multiplicamos la suma por la base de os trapecios, para obtener el área␣
 ↪total
    return EcSuma * EcH

# Ejecutamos la función de integración
EcIntegralAproximada = EcReglaTrapecio(EcA, EcB, EcN)

# Imprimimos el resltado final
print(f"La integral aproximada calculada por la regla del trapecio es:␣
 ↪{EcIntegralAproximada}")
```

```
La integral aproximada calculada por la regla del trapecio es:
0.7468180014679697
```

0.3 Exactitud vs Precisión

Exactitud: Cerca del valor real.

Presicisión: Valores consistentes entre sí.

Este ejemplo muestra cómo se puede evaluar la calidad de un conjunto de mediciones,en el código que escribi hice que se calcule ambas usando tres mediciones simuladas y se imprime el promedio, la exactitud y la precisión para entender la diferencia entre Exactitud vs Precisión.

[25]:
```python
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math
```

```
# Le ponemos un valor verdadero para el calculo.
EcValorVerdadero = 100.0

# Le damos tres mediciones simuladas
EcMedicion1 = 98.0
EcMedicion2 = 98.2
EcMedicion3 = 98.1

# Promediamos las mediciones
EcPromedio = (EcMedicion1 + EcMedicion2 + EcMedicion3) / 3

# Exactitud: Cerca del valor real
EcExactitud = abs(EcPromedio - EcValorVerdadero)

# Precisión: Valores consistentes entre sí
EcPrecisión = max(EcMedicion1, EcMedicion2, EcMedicion3) - min(EcMedicion1,␣
 ↪EcMedicion2, EcMedicion3)

# Imprimimos los resultados
print("Mediciones:", EcMedicion1, EcMedicion2, EcMedicion3)
print("Promedio:", EcPromedio)
print("Exactitud (diferencia con el valor real):", EcExactitud)
print("Precisión (diferencia entre mediciones):", EcPrecisión)
```

```
Mediciones: 98.0 98.2 98.1
Promedio: 98.09999999999998
Exactitud (diferencia con el valor real): 1.90000000000002
Precisión (diferencia entre mediciones): 0.20000000000000284
```

0.4 Tipos de errores

0.4.1 Error por truncamiento

Si un algoritmo con truncamiento fijo no se implementa correctamente, puede proporcionar resultados exactos pero con una precisión limitada.

Haremos el ejemplo con pi, con 5 cifras significativas.

[32]:
```
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

EcPiReal = math.pi
EcPiStr = str(EcPiReal)

# Truncar a 5 cifras significativas
```

```
EcPiTruncado = float(EcPiStr[:6]) #Toma desde la posicion 0 hasta la posicion 5

print("Truncamiento de pi a 5 cifras significativas:", EcPiTruncado)
```

```
Truncamiento de pi a 5 cifras significativas: 3.1415
```

0.4.2 Error por redondeo

Un algoritmo con redondeo puede tener mucha aproximación pero no exactitud.

Haremos el ejemplo con pi, con 5 cifras significativas.

[34]:
```
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

EcPiReal = math.pi

# Redondear a 5 cifras significativas -> 3.1416
EcPiRedondeado = round(EcPiReal, 4)  # 4 decimales para total 5 cifras␣
 ↪significativas

print("Redondeo de pi a 5 cifras significativas:", EcPiRedondeado)
```

```
Redondeo de pi a 5 cifras significativas: 3.1416
```

0.5 Cálculo de error

El ejemplo se realizara para los cuatro errores con $p = \sqrt{3}$ y $p* = 1.73205$

0.5.1 Error real

[48]:
```
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

# Definir los valores
EcValorReal = math.sqrt(3)
EcValorAprox = 1.73205# Aproximación truncada

# Calcular el error real
EcErrorReal = EcValorReal - EcValorAprox
print("1. Error real:", EcErrorReal)
print("2. Valor real:", EcValorReal)
print("3. Valor aproximado:", EcValorAprox)
```

```
1. Error real: 8.075688771036482e-07
2. Valor real: 1.7320508075688772
3. Valor aproximado: 1.73205
```

0.5.2 Error absoluto

[49]:
```python
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

# Definir los valores reales y aproximados
EcValorReal = math.sqrt(3)
EcValorAprox = 1.73205 # Aproximación truncada

# Calcular el error absoluto
EcErrorAbsoluto = abs(EcValorReal - EcValorAprox)

# Mostrar el resultado
print("1. Error absoluto:", EcErrorAbsoluto)
print("2. Valor real:", EcValorReal)
print("3. Valor aproximado:", EcValorAprox)
```

```
1. Error absoluto: 8.075688771036482e-07
2. Valor real: 1.7320508075688772
3. Valor aproximado: 1.73205
```

0.5.3 Error relativo

[50]:
```python
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

# Definir los valores
EcValorReal = math.sqrt(3)
EcValorAprox = 1.73205

# Calcular el error relativo (error absoluto dividido por el valor real)
EcErrorRelativo = math.fabs(EcValorReal - EcValorAprox) / EcValorReal
print("1. Error relativo:", EcErrorRelativo)
print("2. Valor real:", EcValorReal)
print("3. Valor aproximado:", EcValorAprox)
```

```
1. Error relativo: 4.6625010858495513e-07
2. Valor real: 1.7320508075688772
3. Valor aproximado: 1.73205
```

0.5.4 Error relativo porcentual

[53]:
```
#@autor: Estefano Condoy
#@fecha: 2025-30-04
#@version: 1.0
#@asignatura: Métodos Numéricos
import math

# Definir los valores
EcValorReal = math.sqrt(3)
EcValorAprox = 1.73205

# Calcular el error relativo porcentual
EcErrorPorcentual = (math.fabs(EcValorReal - EcValorAprox) / EcValorReal) * 100
print("1. Error relativo porcentual:", EcErrorPorcentual, "%")
print("2. Valor real:", EcValorReal)
print("3. Valor aproximado:", EcValorAprox)
```

```
1. Error relativo porcentual: 4.6625010858495514e-05 %
2. Valor real: 1.7320508075688772
3. Valor aproximado: 1.73205
```

U2 Preliminares B

0.6 Representación Numérica 32 bits

- 1 bit se usa para el signo (0 para positivo, 1 para negativo).
- 8 bits corresponden al exponente.
- 23 bits se destinan a la mantisa.

[ ]:
```
#@autor: Estefano Condoy
#@fecha: 2025-06-05
#@version: 1.0
#@asignatura: Métodos Numéricos

# Este numero se eligio fue porque el ingeniero nos lo dio en clases
EcNumero = 263.3

# En el paso 1 determinamos el signo del Bit
EcSigno = 0
if EcNumero < 0:
    EcSigno = 1
    EcNumero = -EcNumero
```

```python
# 2.En el paso 2 convertimos en binario la parte entera
EcParteEntera = int(EcNumero)  # Tomamos la parte entera de 263
EcBinEntera = bin(EcParteEntera)[2:]  # Convertimos 263 a binario

# En el paso 3 convertimos en binrio la parte decimal
EcParteDecimal = EcNumero - EcParteEntera  # Obtenemos la parte decimal → 0.3
EcBinDecimal = ""
EcTemp = EcParteDecimal
while len(EcBinDecimal) < 23:
    EcTemp *= 2
    bit = int(EcTemp)
    EcBinDecimal += str(bit)
    EcTemp -= bit
    if EcTemp == 0:
        break

# En el paso 4 lo convertimos en notación científica
EcExp = len(EcBinEntera) - 1  # Le desplazamos la coma a la izquierda hasta␣
 ↪antes del primer bit
EcMantisa = EcBinEntera[1:] + EcBinDecimal
EcMantisa = EcMantisa[:23]  # Solo tomamos los primeros 23 bits para la mantisa

# En el paso 5 Sumamos el exponente mas 127 para el formato IEEE 754
EcExponente = EcExp + 127
EcExponenteBin = format(EcExponente, '08b')  # Representamos el exponente en 8␣
 ↪bits binarios

# En el ultimo paso unimos al format IEEE 754
Ec32bits = str(EcSigno) + EcExponenteBin + EcMantisa
# Imprimimos los resultados
print("Bit del signo:", EcSigno)
print("Bits de la parte del exponente:", EcExponenteBin)
print("Bits de la parte de la mantisa:", EcMantisa)
print("IEEE 754 (32 bits):", Ec32bits)
```

```
Bit del signo: 0
Bits de la parte del exponente: 10000111
Bits de la parte de la mantisa: 00000111010011001100110
IEEE 754 (32 bits): 01000011100000111010011001100110
```

0.7 Representación Numérica 64 bits

- 1 bit se usa para el signo (0 para positivo, 1 para negativo).
- 11 bits corresponden al exponente.
- 52 bits se destinan a la mantisa.

[ ]:
```python
#@autor: Estefano Condoy
#@fecha: 2025-06-05
#@version: 1.0
#@asignatura: Métodos Numéricos

# Este numero se eligio fue porque el ingeniero nos lo dio en clases
EcNumero = 263.3

# En el paso 1 determinamos el signo del Bit
EcSigno = 0
if EcNumero < 0:
    EcSigno = 1
    EcNumero = -EcNumero

# 2.En el paso 2 convertimos en binario la parte entera
EcParteEntera = int(EcNumero)  # Tomamos la parte entera de 263
EcBinEntera = bin(EcParteEntera)[2:]  # Convertimos 263 a binario

# En el paso 3 convertimos en binrio la parte decimal
EcParteDecimal = EcNumero - EcParteEntera  # Obtenemos la parte decimal → 0.3
EcBinDecimal = ""
EcTemp = EcParteDecimal
while len(EcBinDecimal) < 52:
    EcTemp *= 2
    bit = int(EcTemp)
    EcBinDecimal += str(bit)
    EcTemp -= bit
    if EcTemp == 0:
        break

# En el paso 4 lo convertimos en notación científica
EcExp = len(EcBinEntera) - 1  # Le desplazamos la coma a la izquierda hasta␣
 ↪antes del primer bit
EcMantisa = EcBinEntera[1:] + EcBinDecimal
EcMantisa = EcMantisa[:52]  # Solo tomamos los primeros 23 bits para la mantisa

# En el paso 5 Sumamos el exponente mas 127 para el formato IEEE 754
EcExponente = EcExp + 1023
EcExponenteBin = format(EcExponente, '011b')  # Representamos el exponente en 8␣
 ↪bits binarios

# En el ultimo paso unimos al format IEEE 754
Ec64bits = str(EcSigno) + EcExponenteBin + EcMantisa
# Imprimimos los resultados
print("Bit del signo:", EcSigno)
print("Bits de la parte del exponente:", EcExponenteBin)
print("Bits de la parte de la mantisa:", EcMantisa)
```

```
print("IEEE 754 (32 bits):", Ec64bits)
```

```
Bit del signo: 0
Bits de la parte del exponente: 10000000111
Bits de la parte de la mantisa:
00000111010011001100110011001100110011001100110011001101
IEEE 754 (32 bits):
01000000011100000111010011001100110011001100110011001100110011001101
```

0.8 Aritmética de dígitos finitos / de computador

Es la representación en punto flotante y sus operaciones son:

$$x \oplus y = \text{fl}(\text{fl}(x) + \text{fl}(y))$$

$$x \ominus y = \text{fl}(\text{fl}(x) - \text{fl}(y))$$

$$x \otimes y = \text{fl}(\text{fl}(x) \times \text{fl}(y))$$

$$x \oslash y = \text{fl}\left(\frac{\text{fl}(x)}{\text{fl}(y)}\right)$$

[ ]:
```
#@autor: Estefano Condoy
#@fecha: 2025-06-05
#@version: 1.0
#@asignatura: Métodos Numéricos

Ec_x = 3.141592
Ec_y = 2.718281

def Ec_fl_redondeo(numero):
    # Simulamos la aritmetica de computadora
    if numero == 0:
        return 0.0

    negativo = False
    if numero < 0:
        negativo = True
        numero = -numero

    exponente = 0
    # Normalizamos para que el numero siempre empieze en 0....( ya que la␣
 ↪computadora solo maneja 0 y 1)
    while numero >= 1:
        numero = numero / 10
```

```
        exponente += 1
    while numero < 0.1:
        numero = numero * 10
        exponente -= 1

    # Redondeamos a 3 cifras significativas
    numero = int(numero * 1000 + 0.5) / 1000.0

    # Reconstruye el número completo
    numero = numero * (10 ** exponente)
    if negativo:
        numero = -numero
    return numero

# Definimos la función para convertir a formato científico
def Ec_formato_cientifico(numero):
    if numero == 0:
        return "0.000e+00"

    negativo = numero < 0
    if negativo:
        numero = -numero

    exponente = 0
    while numero >= 1:
        numero = numero / 10
        exponente += 1
    while numero < 0.1:
        numero = numero * 10
        exponente -= 1

    numero = int(numero * 1000 + 0.5) / 1000.0
    signo = "-" if negativo else ""
    return f"{signo}0.{str(numero)[2:]:<3}e{exponente:+03d}"

# realizamos las operaciones
def Ec_suma(x, y):
    return Ec_fl_redondeo(Ec_fl_redondeo(x) + Ec_fl_redondeo(y))

def Ec_resta(x, y):
    return Ec_fl_redondeo(Ec_fl_redondeo(x) - Ec_fl_redondeo(y))

def Ec_multiplicacion(x, y):
    return Ec_fl_redondeo(Ec_fl_redondeo(x) * Ec_fl_redondeo(y))

def Ec_division(x, y):
    return Ec_fl_redondeo(Ec_fl_redondeo(x) / Ec_fl_redondeo(y))
```

```
# Imprimimos el cambio a aritmetica de computadoras los datos dados.
print("x =", Ec_fl_redondeo(Ec_x), "→",␣
 ↪Ec_formato_cientifico(Ec_fl_redondeo(Ec_x)))
print("y =", Ec_fl_redondeo(Ec_y), "→",␣
 ↪Ec_formato_cientifico(Ec_fl_redondeo(Ec_y)))
# Imprimimos los resultados de las operaaciones en resultado normal y tambien␣
 ↪en formato de computadoras]
print("\nResultados de las operaciones:")
print("\nx   y =",Ec_formato_cientifico(Ec_suma(Ec_x, Ec_y)))
print("x   y =",Ec_formato_cientifico(Ec_resta(Ec_x, Ec_y)))
print("x   y =",Ec_formato_cientifico(Ec_multiplicacion(Ec_x, Ec_y)))
print("x   y =",Ec_formato_cientifico(Ec_division(Ec_x, Ec_y)))
```

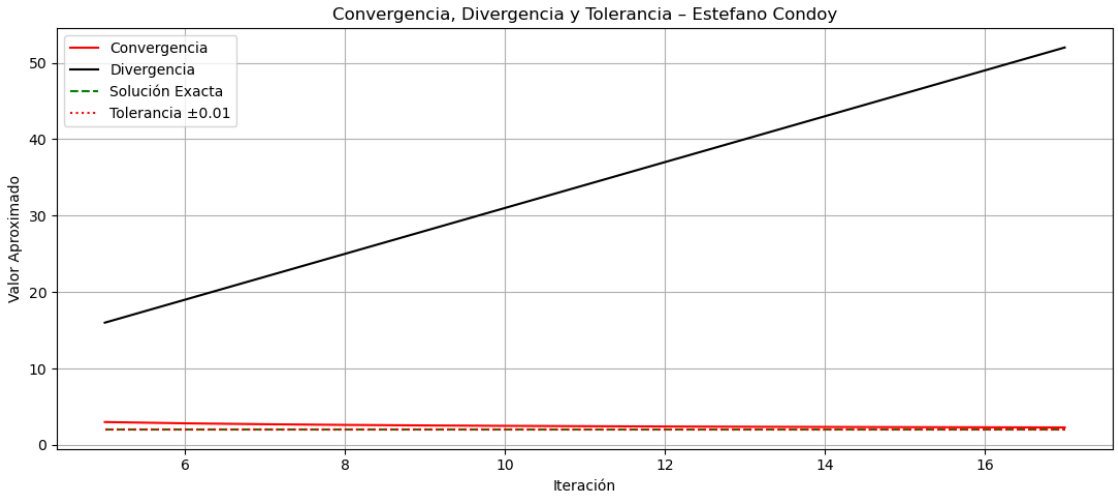
```
x = 3.14 → 0.314e+01
y = 2.72 → 0.272e+01

Resultados de las operaciones:

x   y = 0.586e+01
x   y = 0.42 e+00
x   y = 0.854e+01
x   y = 0.115e+01
```

0.9 Convergencia, Divergencia y Tolerancia

- **Convergencia:** Al aumentar el número de iteraciones, los resultados se acerca a la solución exacta.
- **Divergencia:** Si los Resultados se alejan o nunca se estabilizan.
- **Tolerancia:** Es la cantidad de error que se permite en la solución, o tambien se utilizan los criterios de parada.

[ ]:
```
#@autor: Estefano Condoy
#@fecha: 2025-06-05
#@version: 1.0
#@asignatura: Métodos Numéricos

import matplotlib.pyplot as plt # Utilizo esta libreria solo para poder␣
 ↪graficar.

Ec_solucion = 2.0
Ec_tolerancia = 0.01
Ec_iteraciones = list(range(5, 18))

# Convergencia: se acerca a la solución
Ec_convergencia = [2 + 5 / e for e in Ec_iteraciones]

# Divergencia: se aleja de la solución
```

```
Ec_divergencia = [1 + e * 3 for e in Ec_iteraciones]

# Aqui estaran las lineas de referencia
Ec_sol_line = [Ec_solucion] * len(Ec_iteraciones)
Ec_tolerancia_sup = [Ec_solucion + Ec_tolerancia] * len(Ec_iteraciones)
Ec_tolerancia_inf = [Ec_solucion - Ec_tolerancia] * len(Ec_iteraciones)

# Gráfica
plt.figure(figsize=(11, 5))
plt.plot(Ec_iteraciones, Ec_convergencia, color='red', label='Convergencia')
plt.plot(Ec_iteraciones, Ec_divergencia, color='black', label='Divergencia')
plt.plot(Ec_iteraciones, Ec_sol_line, 'g--', label='Solución Exacta')
plt.plot(Ec_iteraciones, Ec_tolerancia_sup, 'r:', label='Tolerancia ±0.01')
plt.plot(Ec_iteraciones, Ec_tolerancia_inf, 'r:')

plt.title("Convergencia, Divergencia y Tolerancia - Estefano Condoy")
plt.xlabel("Iteración")
plt.ylabel("Valor Aproximado")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```



0.10 Criterios de parada

1. **Error absoluto**

$$|x_n - x_{n-1}| < \varepsilon$$

2. **Error relativo**

$$\frac{|x_{n+1} - x_n|}{|x_{n+1}|} < \varepsilon$$

3. **Número máximo de iteraciones**

$$K \geq i$$

4. **Tiempo de ejecución máximo**

   - Se detiene si el tiempo total excede un límite predefinido **T**.

[ ]:
```
#@autor: Estefano Condoy
#@fecha: 2025-06-05
#@version: 1.0
#@asignatura: Métodos Numéricos
import time

def Ec_f(x):
    # Función x^2 - 2
    return x**2 - 2

def Ec_biseccion(Ec_a, Ec_b, Ec_K=5, Ec_epsilon_abs=1e-5, Ec_epsilon_rel=1e-5,␣
 ↪Ec_epsilon_func=1e-5, Ec_T=1):
    Ec_start_time = time.time()

    if Ec_f(Ec_a)*Ec_f(Ec_b) >= 0:
        print("Error: f(a) y f(b) deben tener signos opuestos")
        return None

    Ec_x_anterior = Ec_a
    for Ec_i in range(1, Ec_K+1):
        Ec_x_nuevo = (Ec_a + Ec_b) / 2
        Ec_f_xnuevo = Ec_f(Ec_x_nuevo)

        # Criterios de parada
        Ec_error_abs = abs(Ec_x_nuevo - Ec_x_anterior)
        Ec_error_rel = Ec_error_abs / abs(Ec_x_nuevo) if Ec_x_nuevo != 0 else␣
 ↪float('inf')
        Ec_tiempo_transcurrido = time.time() - Ec_start_time

        print(f"Iteración {Ec_i}: x={Ec_x_nuevo:.6f}, f(x)={Ec_f_xnuevo:.6e},␣
 ↪error_abs={Ec_error_abs:.6e}, error_rel={Ec_error_rel:.6e},␣
 ↪tiempo={Ec_tiempo_transcurrido:.2f}s")

        if Ec_error_abs < Ec_epsilon_abs:
            print("Parada por error absoluto")
            return Ec_x_nuevo
```

```
        if Ec_error_rel < Ec_epsilon_rel:
            print("Parada por error relativo")
            return Ec_x_nuevo
        if abs(Ec_f_xnuevo) < Ec_epsilon_func:
            print("Parada por valor de función pequeño")
            return Ec_x_nuevo
        if Ec_tiempo_transcurrido > Ec_T:
            print("Parada por tiempo máximo")
            return Ec_x_nuevo

        # Actualizar intervalo
        if Ec_f(Ec_a)*Ec_f_xnuevo < 0:
            Ec_b = Ec_x_nuevo
        else:
            Ec_a = Ec_x_nuevo

        Ec_x_anterior = Ec_x_nuevo

    print("Parada por número máximo de iteraciones")
    return Ec_x_nuevo

Ec_raiz = Ec_biseccion(1, 2) # Definimos el intervalo [1, 2] para la bisecciónS
print(f"Raíz aproximada: {Ec_raiz}")
```

```
Iteración 1: x=1.500000, f(x)=2.500000e-01, error_abs=5.000000e-01,
error_rel=3.333333e-01, tiempo=0.00s
Iteración 2: x=1.250000, f(x)=-4.375000e-01, error_abs=2.500000e-01,
error_rel=2.000000e-01, tiempo=0.00s
Iteración 3: x=1.375000, f(x)=-1.093750e-01, error_abs=1.250000e-01,
error_rel=9.090909e-02, tiempo=0.00s
Iteración 4: x=1.437500, f(x)=6.640625e-02, error_abs=6.250000e-02,
error_rel=4.347826e-02, tiempo=0.00s
Iteración 5: x=1.406250, f(x)=-2.246094e-02, error_abs=3.125000e-02,
error_rel=2.222222e-02, tiempo=0.00s
Parada por número máximo de iteraciones
Raíz aproximada: 1.40625
```

U3 Soluciones de sistemas de ecuaciones no lineales A

0.11 Método de la Bisección

El **Método de Bisección** se fundamenta en el **Teorema del Valor Intermedio**, el cual asegura la existencia de una raíz dentro de un intervalo cerrado ([a, b]), siempre que la función (f(x)) sea continua en dicho intervalo y se cumpla la condición:

$$f(a) \cdot f(b) < 0$$

0.11.1 Fórmula del punto medio

Para cada iteración, se calcula el punto medio (m):

$$m = \frac{a+b}{2}$$

Con esto evaluamos (f(m)):

0.11.2 Condiciones de evaluación

- **Si (f(m) = 0):** (m) es una **raíz exacta**.
- **Si (f(a) * f(m) < 0):** la raíz está en el intervalo ([a, m]).
- **Si (f(b) * f(m) > 0):** la raíz está en el intervalo ([m, b]).

Se repite el procedimiento sobre el nuevo intervalo hasta que el **error absoluto** sea inferior a una **tolerancia** previamente establecida.

Ejemplo: Use el método de bisección para encontrar soluciones precisas dentro de (10^{-2}) para:

$$x^3 - 7x^2 + 14x - 6 = 0$$

Intervalo : ([0, 1])

[119]:
```
# @autor: Estefano Condoy
# @fecha: 2025-14-05
# @version: 1.0
# @asignatura: Métodos Numéricos

import math

def EcFuncion(x):
    return x**3 - 7*x**2 + 14*x - 6  # Esta función es de un deber

cA = 0            # Límite izquierdo
EcB = 1            # Límite derecho
EcTOL = 1e-2       # Tolerancia
EcN0 = 50          # Número máximo de iteraciones


def EcBiseccion(EcA, EcB, EcTOL, EcN0):
    EcFA = EcFuncion(EcA)

    print(f"{'Iter':<6}{'EcA':<12}{'EcB':<12}{'EcC':<12}{'EcF(A)':<12}{'EcF(B)':
↪<12}{'EcF(C)':<12}{'Tolerancia':<12}")
    print("-" * 90)
```

```
    for EcI in range(1, EcN0 + 1):
        EcC = (EcA + EcB) / 2
        EcFC = EcFuncion(EcC)
        EcFB = EcFuncion(EcB)
        EcError = abs(EcB - EcA) / 2

        print(f"{EcI:<6}{EcA:<12.6f}{EcB:<12.6f}{EcC:<12.6f}{EcFA:<12.6f}{EcFB:
↪<12.6f}{EcFC:<12.6f}{EcError:<12.6f}")

        if EcFC == 0 or EcError < EcTOL:
            print("\nEl Método de Bisección encontró la raíz.")
            print(f"Raíz aproximada: {EcC:.10f}")
            return

        if EcFA * EcFC > 0:
            EcA = EcC
            EcFA = EcFC
        else:
            EcB = EcC

    print("\nEl método fracasó después de", EcN0, "iteraciones.")

EcBiseccion(EcA, EcB, EcTOL, EcN0)
```

```
Iter  EcA         EcB         EcC         EcF(A)      EcF(B)      EcF(C)
Tolerancia
-------------------------------------------------------------------------------
----------
1     0.000000    1.000000    0.500000    -6.000000   2.000000    -0.625000
0.500000
2     0.500000    1.000000    0.750000    -0.625000   2.000000    0.984375
0.250000
3     0.500000    0.750000    0.625000    -0.625000   0.984375    0.259766
0.125000
4     0.500000    0.625000    0.562500    -0.625000   0.259766    -0.161865
0.062500
5     0.562500    0.625000    0.593750    -0.161865   0.259766    0.054047
0.031250
6     0.562500    0.593750    0.578125    -0.161865   0.054047    -0.052624
0.015625
7     0.578125    0.593750    0.585938    -0.052624   0.054047    0.001031
0.007812

El Método de Bisección encontró la raíz.
Raíz aproximada: 0.5859375000
```

U3 Soluciones de sistemas de ecuaciones no lineales B

0.12 Método de Newton (Newton-Raphson)

El método de **Newton-Raphson** es un procedimiento iterativo que obtiene aproximaciones a la raíz de una función no lineal (f(x) = 0).

Este método exige que la función sea **diferenciable**, ya que se fundamenta en la aproximación mediante su **tangente lineal**.

0.12.1 Fórmula iterativa

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad n \geq 1$$

Siempre inicia con un **valor inicial** y emplea la **derivada** de la función para aproximarse gradualmente a la raíz.

Su **convergencia cuadrática** es muy acelerada, siempre que la estimación inicial sea suficientemente cercana a la raíz y la función sea distinta de 0.

Ejemplo: Dada la función:

$$f(x) = x^4 - x - 1$$

Con valor inicial:

$$p_0 = 1$$

Use el **Método de Newton** para obtener soluciones precisas con una tolerancia de:

$$\varepsilon = 10^{-6}$$

[121]:
```python
# @autor: Estefano Condoy
# @fecha: 2025-16-05
# @versión: 1.0
# @asignatura: Métodos Numéricos

import math

def EcFuncion(x):
    return x**4-x-1

def EcDerivada(x):
    return 4*x**3 - 1

# Codificamos el método de Newton-Raphson
```

```python
def EcNewton(EcP0, EcTOL, EcN0):
    print(f"{'n':<6}{'EcXn-1':<12}{'EcF(EcXn-1)':<15}{'EcF\'(EcXn-1)':
 ↪<15}{'EcXn':<12}{'Tolerancia':<12}")
    print("-" * 75)

    for EcI in range(1, EcN0 + 1):
        EcF0 = EcFuncion(EcP0)
        EcDF0 = EcDerivada(EcP0)

        if EcDF0 == 0:
            print("Error: Derivada cero. No se puede continuar.")
            return

        EcP = EcP0 - EcF0 / EcDF0
        EcError = abs(EcP - EcP0)

        print(f"{EcI:<6}{EcP0:<12.6f}{EcF0:<15.6f}{EcDF0:<15.6f}{EcP:<12.
 ↪6f}{EcError:<12.6f}")

        if EcError < EcTOL:
            print("\nEl método de Newton-Raphson encontro la raíz.")
            print(f"Raíz aproximada: {EcP:.10f}")
            return

        EcP0 = EcP

    print("\nEl método falló después de", EcN0, "iteraciones.")

EcP0 = 1         # Aproximación inicial
EcTOL = 1e-6       # Tolerancia
EcN0 = 50          # Máximo número de iteraciones

EcNewton(EcP0, EcTOL, EcN0)
```

```
n     EcXn-1      EcF(EcXn-1)     EcF'(EcXn-1)    EcXn        Tolerancia
---------------------------------------------------------------------------
1     1.000000    -1.000000       3.000000        1.333333    0.333333
2     1.333333    0.827160        8.481481        1.235808    0.097525
3     1.235808    0.096596        6.549407        1.221059    0.014749
4     1.221059    0.001977        6.282323        1.220744    0.000315
5     1.220744    0.000001        6.276693        1.220744    0.000000

El método de Newton-Raphson encontro la raíz.
Raíz aproximada: 1.2207440846
```

0.13 Método de la Secante

El **Método de la Secante** es una variante del **Método de Newton**, pero en lugar de utilizar la derivada, emplea una aproximación basada en dos puntos cercanos de la función.

0.13.1 Fórmula iterativa

$$x_n = x_{n-1} - f(x_{n-1}) \times \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

0.13.2 Pasos del método

- Se eligen **dos valores iniciales** dentro del intervalo de interés.
- Se calcula el siguiente valor utilizando la ecuación iterativa.
- Se genera una **secuencia de valores** que se aproxima a la raíz de la ecuación.
- **El algoritmo se detiene** cuando se alcanza la raíz o una aproximación que cumple con un criterio de precisión, como **tolerancia** o otro criterio de parada.

Ejemplo: Dada la función:

$$f(x) = x^4 - x - 1$$

Con valor inicial:

$$p_0 = 1 p_1 = 1.4$$

Use el **Método de la secante** para obtener soluciones precisas con una tolerancia de:

$$\varepsilon = 10^{-6}$$

[ ]:
```
# @autor: Estefano Condoy
# @fecha: 2025-14-05
# @versión: 1.0
# @asignatura: Métodos Numéricos

import math

def EcFuncion(x):
    return x**4-x-1

# Codificar método de la secante
def EcSecante(EcP0, EcP1, EcTOL, EcN0):
    EcQ0 = EcFuncion(EcP0)
    EcQ1 = EcFuncion(EcP1)

    print(f"{'n-2':<12}{'n-1':<12}{'EcF(n-2)':<15}{'EcF(n-1)':<15}{'n':
<12}{'Tolerancia':<12}")
    print("-" * 80)

    for EcI in range(2, EcN0 + 1):
```

```
        if (EcQ1 - EcQ0) == 0:
            print("Error: División por cero (EcQ1 - EcQ0 = 0)")
            return

        EcP = EcP1 - EcQ1 * (EcP1 - EcP0) / (EcQ1 - EcQ0)
        EcError = abs(EcP - EcP1)

        print(f"{EcP0:<12.6f}{EcP1:<12.6f}{EcQ0:<15.6f}{EcQ1:<15.6f}{EcP:<12.
↪6f}{EcError:<12.6f}")

        if EcError < EcTOL:
            print("\nEl método de la Secante fue exitoso.")
            print(f"Raíz aproximada: {EcP:.10f}")
            return

        # Actualizamos los valores para la siguiente iteración
        EcP0, EcQ0 = EcP1, EcQ1
        EcP1, EcQ1 = EcP, EcFuncion(EcP)

    print("\nEl método falló después de", EcN0, "iteraciones.")

EcP0 = 1.0          # Aproximación inicial n-2
EcP1 = 1.4          # Aproximación inicial n-1
EcTOL = 1e-6        # Tolerancia
EcN0 = 50           # Número máximo de iteraciones

EcSecante(EcP0, EcP1, EcTOL, EcN0)
```

```
n-2          n-1          EcF(n-2)        EcF(n-1)        n            Tolerancia
-------------------------------------------------------------------------------------
1.000000     1.400000     -1.000000       1.441600        1.163827     0.236173
1.400000     1.163827     1.441600        -0.329175       1.207730     0.043903
1.163827     1.207730     -0.329175       -0.080182       1.221868     0.014138
1.207730     1.221868     -0.080182       0.007065        1.220723     0.001145
1.221868     1.220723     0.007065        -0.000132       1.220744     0.000021
1.220723     1.220744     -0.000132       -0.000000       1.220744     0.000000

El método de la Secante fue exitoso.
Raíz aproximada: 1.2207440846
```

U3 Soluciones de sistemas de ecuaciones no lineales C

0.14 Método de Posición Falsa (Regula Falsi)

Este método mantiene las propiedades del **Método de Bisección** y la **fórmula iterativa de la Secante**, pero se diferencia en la forma de encontrar nuevos intervalos.

A partir de **dos puntos iniciales**, se traza una **recta secante** y se determina su **intersección con el eje (x)**, lo que permite una aproximación más rápida a la raíz.

0.14.1 Fórmula iterativa

$$x_n = x_{n-1} - f(x_{n-1}) \times \frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-2})}$$

Se verifica la condición:

$$\text{sgn} f(p_2) \times \text{sgn} f(p_1) < 0$$

0.14.2 Pasos del método

- Se eligen **dos valores iniciales** (p_0) y (p_1) dentro del intervalo de interés, y se traza una **recta secante** entre ellos.
- Se verifica que (f(p_0)) y (f(p_1)) tengan **signos opuestos**, lo que asegura la existencia de una **raíz aproximada en el intervalo**.

Ejemplo: Dada la función:

$$f(x) = x^4 - x - 1$$

Con valor inicial:

$$p_0 = 1 p_1 = 1.4$$

Use el **Método de la Posición Falsa** para obtener soluciones precisas con una tolerancia de:

$$\varepsilon = 10^{-6}$$

[131]:
```
# @autor: Estefano Condoy
# @fecha: 2025-14-05
# @versión: 1.0
# @asignatura: Métodos Numéricos

import math

def EcFuncion(x):
    return x**4 - x - 1

# Método de la posición falsa (Regula Falsi)
def EcPosicionFalsa(EcP0, EcP1, EcTOL, EcN0):
    EcQ0 = EcFuncion(EcP0)
    EcQ1 = EcFuncion(EcP1)
```

```
    print(f"{'n-2':<12}{'n-1':<12}{'f(n-2)':<15}{'f(n-1)':<15}{'n':<12}{'Error':
↪<12}")
    print("-" * 80)

    for EcI in range(2, EcN0 + 1):
        if (EcQ1 - EcQ0) == 0:
            print("Error: División por cero (EcQ1 - EcQ0 = 0)")
            return

        EcP = EcP1 - EcQ1 * (EcP1 - EcP0) / (EcQ1 - EcQ0)
        EcError = abs(EcP - EcP1)

        print(f"{EcP0:<12.6f}{EcP1:<12.6f}{EcQ0:<15.6f}{EcQ1:<15.6f}{EcP:<12.
↪6f}{EcError:<12.6f}")

        if EcError < EcTOL:
            print("\nEl método de la Posición Falsa fue exitoso.")
            print(f"Raíz aproximada: {EcP:.10f}")
            return

        EcQ = EcFuncion(EcP)

        # Verificamos el signoS
        if EcQ * EcQ1 < 0:
            EcP0 = EcP1
            EcQ0 = EcQ1

        EcP1 = EcP
        EcQ1 = EcQ

    print("\nEl método falló después de", EcN0, "iteraciones.")

# Parámetros de entrada
EcP0 = 1.0        # Aproximación inicial p0
EcP1 = 1.4        # Aproximación inicial p1
EcTOL = 1e-6      # Tolerancia
EcN0 = 50         # Número máximo de iteraciones

# Ejecutar método
EcPosicionFalsa(EcP0, EcP1, EcTOL, EcN0)
```

```
n-2          n-1          f(n-2)          f(n-1)          n            Error
--------------------------------------------------------------------------------
1.000000     1.400000     -1.000000       1.441600        1.163827     0.236173
1.400000     1.163827     1.441600        -0.329175       1.207730     0.043903
1.400000     1.207730     1.441600        -0.080182       1.217861     0.010131
1.400000     1.217861     1.441600        -0.018025       1.220110     0.002249
```

```
1.400000    1.220110    1.441600       -0.003978      1.220605    0.000495
1.400000    1.220605    1.441600       -0.000874      1.220714    0.000109
1.400000    1.220714    1.441600       -0.000192      1.220737    0.000024
1.400000    1.220737    1.441600       -0.000042      1.220743    0.000005
1.400000    1.220743    1.441600       -0.000009      1.220744    0.000001
1.400000    1.220744    1.441600       -0.000002      1.220744    0.000000

El método de la Posición Falsa fue exitoso.
Raíz aproximada: 1.2207440136
```