EQDLabII

Link a este repositorio
Segundo Laboratorio para la materia de Ecuaciones Diferenciales

Descripción

Para este laboratorio se tuvieron que implementar varios
Métodos númericos enseñados en la clase de *Ecuaciones Diferenciales y Métodos Números*.

Los implementados en este código fueron:

- Fórmulas de 2, 3 y 5 puntos.
- Método de Newton-Raphson.
- Diferencias Divididas.
- Regla del Trapecio Compuesto.
- Fórmulas de Newton-Cotes.

Formato de de las funciones

El formato es muy simple, solo usar parentesis, no llaves ni corchetes.
todas las funciones deben ser escritas en minusculas como `sin`, `cos` o `ln`.
El número de euler es una `E` mayuscula y π es `pi`.
todas las multiplicaciones deben de ser explicitas usando '*',
y los exponenciales deben ponerse como doble asterisco '**'.

Ejemplos:

- cos(x) - x
- 2**ln(x)
- 1 - 2*x + 3*x**2 - 5*x**3

Instalación de las librerias necesarias.

Antes de intentar instalar las librerias y ejecutar por consola,
hay un ejecutable .exe en la carpeta *build*
puesto que quise hacer todo lo más fácil posible,
pero mi computador me dice que la aplicación es un virus a pesar

de que yo mismo programé la aplicación,
tal vez por el método que utilicé para hacer el .exe.
Si te aparece lo mismo entoces toca abrir una consola en la carpeta
de este repositorio y seguir los siguientes pasos:

Para ejecutar este proyecto se necesita activar un entorno virtual,
instalar los paquetes necesarios y finalmete ejecutar el *src/main.py*

Para crear el entorno virtual unicamente es necesario ejecutar el
siguiente comando:

```
python -m venv venv
```

Activarlo en linux con:

```
source ./venv/bin/activate
```

o en Windows con:

```
.\venv\Scripts\activate
```

Luego de creado y activado el entorno virtual por primera vez,
se tienen que instalar las dependencias:

```
python -m pip -r requirements.txt
```

Hecho todo lo anterior ya se puede abrir la aplicación:

```
python ./src/main.py
```

Métodos Usados

Estimación de una derivada usando las fórmulas de 2, 3 y 5 puntos

```
def calculate_two_points(func, x_val, h_val) -> tuple[float, float]:
    """ Calculates a derivative estimation using the method of the two points

    Args:
        func (Sympy expr): Function to obtain the value of.
```

```
        x_val (float): value in x of the func.
        h_val (val): delta's value.
    Return:
        tuple[float, float]: the estimated value and the error
    """
    point_one = func.subs({x: x_val})
    point_two = func.subs({x: x_val + h_val})
    result = (point_two - point_one)/h_val
    error = Abs((h_val/2)*diff(diff(func, x), x).subs({x: x_val}))
    # error = (h_val/2)*(diff(Derivative(func)), x_val)
    return round(result, 5), round(error, 5)
```

```
def calculate_three_points(func, x_val, h_val):
    """Calculates a derivative estimation using the method of the three points

    Args:
        func (Sympy expr): Function to obtain the value of.
        x_val (float): value in x of the func.
        h_val (val): delta's value.
    Return:
        tuple[float, float]: the estimated value and the error
    """
    point_one = func.subs({x: x_val - h_val})
    point_two = func.subs({x: x_val + h_val})
    result = float((point_two - point_one)/(2*h_val))
    error = float(Abs(((h_val**2)/6)*diff(diff(diff(func, x), x),x).subs({x:
x_val})))
    return round(result, 7), round(error, 7)
```

```
def calculate_five_points(func, x_val, h_val):
    """Calculates a derivative estimation using the method of the five points

    Args:
        func (Sympy expr): Function to obtain the value of.
        x_val (float): value in x of the func.
        h_val (val): delta's value.
    Return:
        tuple[float, float]: the estimated value and the error
    """
    val_1= func.subs({x: x_val - 2*h_val})
    val_2 = 8*func.subs({x: x_val - h_val})
    val_3 = 8*func.subs({x: x_val + h_val})
    val_4 = func.subs({x: x_val + 2*h_val})
    result = float((val_1-val_2+val_3-val_4)/(12*h_val))
```

```
    fifth_derivative = diff(diff(diff(diff(diff(func, x), x), x), x), x),x)
    error = float(Abs(((h_val**2)/6)*fifth_derivative.subs({x: x_val})))
    return round(result, 10), round(error, 10)
```

Estimación de raices usando el método de Newton-Raphson

```
def newton_raphson(func, x_0, tol=1e-4, max_iteration=100):
    """Estimates the next root of the expression between a tolerance value.

    Args:
        func (Sympy expr): Function to operate with.
        x_0 (float): Next value to the root.
        tol (float): Tolerance.
        max_iteration: Max number of iterations.
    Returns:
        float: the next root of the function.
    """
    derivative = diff(func, x)
    prev = None
    for i in range(max_iteration):
        f_val = func.subs({x: x_0})
        diff_val = derivative.subs({x: x_0})

        prev = x_0
        x_0 = x_0 - (f_val/diff_val)

        if abs(f_val) < tol or round(prev, 6) == round(x_0, 6):
            return round(float(x_0), 6), i + 1

    return round(float(x_0), 6), max_iteration
```

Polinomio de Lagrange usando Diferencias divididas

```
def divided_differences(x_data, fx_data):
    """Creates a table with the dividide differences of the data.

    Args:
        x_data list[float]: Points in X.
        fx_data list[float]: Points of Fx.
    Returns:
        dd_table (list): Dividide differences.
    """
    n = len(x_data)
```

```
    dd_table = []
    for _ in range(n): dd_table.append([None for _ in range(n)])

    for i in range(n): dd_table[i][0] = fx_data[i]

    for j in range(1, n):
        for i in range(n - j):
            dd_table[i][j] = (dd_table[i + 1][j - 1] - dd_table[i][j - 1]) /
(x_data[i + j] - x_data[i])

    return dd_table
```

```
def calculate_polynomial_coefficients(x_data, dd_table):
    """Calculates the polynome using dividide differences:

    Args:
        x_data list[float]: Points in X.
        dd_table (list): Dividide differences.
    Returns:
        coeffiecients (Sympy Expr): the obtained polynome.
    """
    n = len(x_data)
    coefficients = [dd_table[0][0]]

    for i in range(1, n):
        term = round(float(dd_table[0][i]),4)
        for j in range(i):
            prev_term = UnevaluatedExpr(term) if not isinstance(term,
UnevaluatedExpr) else term
            term = prev_term*UnevaluatedExpr(x - x_data[j])
        coefficients.append(term)

    return coefficients
```

Reglas del trapecio y trapecio compuesto

```
def trapezoid_rule(func, a, b, epsilon, is_cotes: bool = False):
    """Estimates the Value of an integral using the trapezoid rule.

    Args:
        func (Sympy expr): Function to obtain the value of.
        a (float): lower limit.
        b (float): upper limit.
        interval (int): num of intervals used to calculate the integral.
```

```
        epsilon (float): a number to calculate the error.
        is_cotes (bool): Boolean to know if it comes from newton-cotes view
    Returns:
        integral_estimation (float): the obtained value of the integral.
    """
    integral_approximation = ((b-a)*(func.subs(x, a) + func.subs(x, b)))/2
    error = (((b-a)**3)/12)*diff(diff(func, x), x).subs(x, epsilon)
    return integral_approximation - error if is_cotes else
(integral_approximation, error)
```

```
def composed_trapezoid_rule(func, a, b, interval, epsilon):
    """Estimates the Value of an integral using the composed trapezoid.

    Args:
        func (Sympy expr): Function to obtain the value of.
        a (float): lower limit.
        b (float): upper limit.
        interval (int): num of intervals used to calculate the integral.
        epsilon (float): a number to calculate the error.
    Returns:
        integral_estimation (float): the obtained value of the integral.
    """
    h = (b - a) / interval
    x_values = [a + i * h for i in range(interval + 1)]
    integral_approximation = 0
    second_diff = diff(diff(func, x), x)

    for val in x_values:
        if(val == a or val == b): integral_approximation += func.subs(x, val)
        else: integral_approximation += 2*func.subs(x, val)
    integral_approximation *= (h/2)
    integral_approximation -= ((b-a)/(12*interval**2))*second_diff.subs(x,
epsilon)
    return integral_approximation
```

Fórmulas cerradas de Newton-Cotes

```
def simpson_rule(func, a, b, epsilon):
    """Estimates the Value of an integral using the Simpson rule.

    Args:
        func (Sympy expr): Function to obtain the value of.
        a (float): lower limit.
        b (float): upper limit.
```

```
        interval (int): num of intervals used to calculate the integral.
        epsilon (float): a number to calculate the error.
    Returns:
        integral_estimation (float): the obtained value of the integral.
    """
    h = (b - a) / 2
    fx0 = func.subs(x, a)
    fx1 = func.subs(x, a + h)
    fx2 = func.subs(x, b)
    diff_4 = diff(diff(diff(diff(func, x), x), x), x)
    error = ((h**5)/90)*diff_4.subs(x, epsilon)
    integral_estimation = (h/3)*(fx0 + 4*fx1 + fx2) - error
    return integral_estimation
```

```
def three_eight_simpson_rule(func, a, b, epsilon):
    """Estimates the Value of an integral using the 3/8 Simpson Rule.

    Args:
        func (Sympy expr): Function to obtain the value of.
        a (float): lower limit.
        b (float): upper limit.
        interval (int): num of intervals used to calculate the integral.
        epsilon (float): a number to calculate the error.
    Returns:
        integral_estimation (float): the obtained value of the integral.
    """
    h = (b - a) / 3
    fx0 = func.subs(x, a)
    fx1 = func.subs(x, a + h)
    fx2 = func.subs(x, a + 2*h)
    fx3 = func.subs(x, b)
    diff_4 = diff(diff(diff(diff(func, x), x), x), x)
    error = (3*(h**5)/80)*diff_4.subs(x, epsilon)
    integral_estimation = (3*h/8)*(fx0 + 3*fx1 + 3*fx2 + fx3) - error
    return integral_estimation
```