20-3-2025

Métodos Numéricos

Unidad 3 Investigación

- Ingeniería en Desarrollo y Gestión de Software -

Integrantes:

Alvarado Salazar Anthony Willians

Diaz Rosales Gerardo Antonio

González Romero Joshua

Grupo: IDYGS82

Matemáticas para Ingeniería II

Métodos Numéricos para Ecuaciones Diferenciales

Introducción

Los métodos numéricos constituyen el puente entre las matemáticas teóricas y las aplicaciones prácticas en el mundo real. Cuando nos enfrentamos a problemas físicos, ingenieriles o científicos que involucran ecuaciones diferenciales, con frecuencia descubrimos que las soluciones exactas son imposibles de obtener analíticamente. Aquí es donde los métodos numéricos emergen como herramientas indispensables.

Este documento se centrar en explorar:

1. **Una exploración profunda de los fundamentos teóricos** donde cada concepto es detallado, incluyendo:
   a. Explicaciones detalladas de qué son los métodos numéricos.
   b. Un análisis de la estimación de error y su importancia crítica.
   c. Descripciones del método de Euler y sus limitaciones
   d. Una exploración en los métodos de Runge-Kutta y sus variantes
2. **Implementación práctica** donde demostraremos cómo estos conceptos abstractos se materializan en una aplicación que fue desarrollada a lo largo de la unidad.

El objetivo es explorar de forma detallada los conceptos anteriormente mencionados, conocer sus variantes, aprender cómo y dónde aplican. Al final se detallará el cómo fue desarrollada la aplicación, junto con evidencia de su mismo uso para resolver métodos numéricos.

1. Métodos Numéricos:

Definición

Los métodos numéricos son algoritmos sistemáticos diseñados para aproximar soluciones a problemas matemáticos que no admiten soluciones analíticas exactas. Su desarrollo histórico está ligado al avance del cálculo en los siglos XVII y XVIII, cuando matemáticos como Newton y Euler sentaron las bases de lo que hoy conocemos como análisis numérico.

Naturaleza fundamental: Estos métodos convierten problemas continuos en discretos mediante:

- Discretización del dominio
- Aproximación de operadores diferenciales
- Iteración controlada hacia soluciones aproximadas

Clasificación

1. **Métodos para ecuaciones algebraicas**:
   a. Bisección
   b. Newton-Raphson
   c. Secante
2. **Métodos para ecuaciones diferenciales ordinarias**:

    a. Euler (explícito e implícito)
    b. Runge-Kutta (variantes de 2º a 8º orden)
    c. Métodos multipaso (Adams-Bashforth, Adams-Moulton)
3. **Métodos para ecuaciones diferenciales parciales**:
    a. Diferencias finitas
    b. Elementos finitos
    c. Volúmenes finitos

Proceso de Resolución Paso a Paso

Ejemplo: Resolución de $f(x) = x^3 - 2x - 5 = 0$ usando Newton-Raphson

1. **Paso 1**: Selección del punto inicial $x_0 = 2$
2. **Paso 2**: Cálculo de la derivada $f'(x) = 3x^2 - 2$
3. **Paso 3**: Aplicación de la fórmula iterativa: $x_{n+1} = x_n - f(x_n)/f'(x_n)$
4. **Iteraciones**:
    a. $x_1 = 2 - (-1)/10 = 2.1$
    b. $x_2 = 2.1 - 0.061/11.23 \approx 2.0945$
    c. $x_3 \approx 2.0945$ (convergencia alcanzada)

Análisis: Se observa convergencia cuadrática típica del método.

2. Estimación de Error:

Definición

El error en métodos numéricos representa la discrepancia entre la realidad matemática y nuestra capacidad de representarla computacionalmente. Comprender el error no es simplemente calcular una diferencia, sino desarrollar una intuición sobre las acumulaciones y la propagación de incertidumbres en los procesos numéricos.

Taxonomía de Errores

1. **Error inherente**:
    a. Errores en datos iniciales
    b. Limitaciones en modelos matemáticos
2. **Error de truncamiento**:
    a. Resultado de aproximar procesos infinitos con sumas finitas
    b. Ejemplo: truncar series de Taylor
3. **Error de redondeo**:
    a. Limitaciones en representación de números reales
    b. Efectos de aritmética de precisión finita
4. **Error algorítmico**:
    a. Errores introducidos por la estructura del método
    b. Propagación de errores en operaciones sucesivas

Metodología de Cálculo de Error

Caso de estudio: Aproximación de la derivada primera

1. **Diferencias finitas hacia adelante**: $f'(x) \approx (f(x+h) - f(x))/h$ Error: $O(h)$
2. **Diferencias centradas**: $f'(x) \approx (f(x+h) - f(x-h))/2h$ Error: $O(h^2)$

Ejemplo numérico: Para $f(x) = \sin(x)$ en $x = \pi/4$

h	Adelante Error	Centrada Error
0.1	0.0678	0.0022
0.01	0.0067	0.000022

Análisis: La reducción de h mejora la precisión, pero hasta cierto límite antes de que dominen los errores de redondeo.

3. Método de Euler:

Fundamentos

El método de Euler representa el punto de entrada a las soluciones numérica de ecuaciones diferenciales. Es un pilar pedagógico debido a su aparente simplicidad conceptual, sin embargo, sus limitaciones prácticas son significativas.

Deducción:

Partiendo de la EDO $dy/dt = f(t,y)$, integramos formalmente: $y(t_{n+1}) = y(t_n) + \int[t_n \rightarrow t_{n+1}] f(t,y)\, dt$

Aproximando la integral por el método del rectángulo: $\int[t_n \rightarrow t_{n+1}] f(t,y)\, dt \approx h \cdot f(t_n,y_n)$

Obtenemos así la fórmula de Euler: $y_{n+1} = y_n + h \cdot f(t_n,y_n)$

Análisis de Error

El error local de truncamiento (LTE) viene dado por el término descartado en la expansión de Taylor: $LTE = (h^2/2)y''(\xi)$

El error global acumulado después de N pasos es $O(h)$, lo que clasifica a Euler como método de primer orden.

Estabilidad numérica:

Consideremos la ecuación test $y' = \lambda y$. La solución numérica será estable si $|1 + h\lambda| \leq 1$. Esto impone restricciones severas sobre h para λ grandes (sistemas rígidos).

Implementación Práctica

Problema modelo: Circuito RC con dV/dt = -V/RC, V(0) = 5V, R=1kΩ, C=1mF

1. **Paso 1**: Definir parámetros
    a. h = 0.1s
    b. Pasos totales = 50
    c. Constante de tiempo RC = 1s
2. **Paso 2**: Algoritmo

```
def euler_rc(V0, R, C, h, steps):
    V = [V0]
    for _ in range(steps):
        V.append(V[-1] + h*(-V[-1]/(R*C)))
    return V
```

3. **Paso 3**: Análisis de resultados
    a. Comparación con solución exacta $V(t) = 5e^{-t}$
    b. Error máximo en t=2s: ≈ 12% con h=0.1s

4. Métodos de Runge-Kutta:

Marco Teórico

Los métodos de Runge-Kutta representan una evolución sofisticada sobre el método de Euler, logrando mayor precisión sin necesidad de calcular derivadas de orden superior. La familia RK generaliza el concepto de usar combinaciones de evaluaciones de la función en puntos estratégicos dentro del intervalo.

Derivación general:

Un método RK de s etapas tiene la forma: $y_{n+1} = y_n + h\sum b_i k_i$ donde $k_i = f(t_n + c_i h, y_n + h\sum a_{ij} k_j)$

Los coeficientes a, b, c determinan las propiedades del método y se representan tradicionalmente en una tabla de Butcher.

RK (RK4):

El método RK4 es uno de los pilares de la solución numérica de EDOs por su equilibrio entre precisión y complejidad computacional.

Analisis:

1. $k_1 = f(t_n, y_n)$ (pendiente al inicio)
2. $k_2 = f(t_n + h/2, y_n + hk_1/2)$ (pendiente en punto medio usando k_1)
3. $k_3 = f(t_n + h/2, y_n + hk_2/2)$ (pendiente en punto medio mejorada)
4. $k_4 = f(t_n + h, y_n + hk_3)$ (pendiente al final del intervalo)

5. $y_{n+1} = y_n + (h/6)(k_1 + 2k_2 + 2k_3 + k_4)$ (combinación ponderada)

Análisis de error:

- Error local: $O(h^5)$
- Error global: $O(h^4)$

Implementación

Problema: Péndulo no lineal $d^2\theta/dt^2 + (g/L)\sin\theta = 0$

1. **Conversión a sistema de primer orden**:
   a. $y_1 = \theta$
   b. $y_2 = d\theta/dt$
   c. $dy_1/dt = y_2$
   d. $dy_2/dt = -(g/L)\sin(y_1)$
2. **Implementación RK4**:

```
def rk4_pendulum(theta0, omega0, g, L, h, steps):
    y = np.zeros((steps+1, 2))
    y[0] = [theta0, omega0]
    for i in range(steps):
        k1 = h * np.array([
            y[i,1],
            -(g/L)*np.sin(y[i,0])
        ])
        k2 = h * np.array([
            y[i,1] + 0.5*k1[1],
            -(g/L)*np.sin(y[i,0] + 0.5*k1[0])
        ])
        k3 = h * np.array([
            y[i,1] + 0.5*k2[1],
            -(g/L)*np.sin(y[i,0] + 0.5*k2[0])
        ])
        k4 = h * np.array([
            y[i,1] + k3[1],
            -(g/L)*np.sin(y[i,0] + k3[0])
        ])
        y[i+1] = y[i] + (k1 + 2*k2 + 2*k3 + k4)/6
    return y
```

3. **Análisis de resultados**:
   a. Conservación de energía (comparación con método de Euler)
   b. Precisión en periodos largos
   c. Adaptación a diferentes amplitudes

Variantes de Runge-Kutta

1. **RK2 (Método del Punto Medio)**:
    a. $y_{n+1} = y_n + hf(t_n + h/2, y_n + (h/2)f(t_n,y_n))$
    b. Orden 2, error global $O(h^2)$
2. **RK3 (Variante de Kutta)**:
    a. Tres evaluaciones por paso
    b. Orden 3, mejor equilibrio precisión-costo
3. **RK Adaptativos**:
    a. Control automático del tamaño de paso
    b. Algoritmos como Fehlberg (RKF45)
    c. Dormand-Prince (ode45 en MATLAB)

Referencias

Burden, R. L. (2010). *Numerical Analysis.* Brooks/Cole.

Chapra, S. C. (2015). *Numerical Methods for Engineers.*

Murray, J. D. (2002). *Mathematical Biology: I. An Introduction.* Springer.

(Murray, 2002)

Atkinson, K. E. (1989). An Introduction to Numerical Analysis (2nd ed.). Wiley

Stoer, J., & Bulirsch, R. (2002). Introduction to Numerical Analysis (3rd ed.). Springer

Butcher, J. C. (1996). A History of Runge-Kutta Methods. Applied Numerical Mathematics, 20(3), 247-260

Hairer, E., Nørsett, S. P., & Wanner, G. (1993). Solving Ordinary Differential Equations I: Nonstiff Problems. Springer

MathWorld by Wolfram. Runge-Kutta Methods.: https://mathworld.wolfram.com/Runge-KuttaMethod.html

Métodos Numéricos para resolver Ecuaciones Diferenciales

En esta actividad, hemos aplicado distintos métodos numéricos para resolver una ecuación diferencial y encontrar la raíz de una ecuación no lineal.

Los siguientes métodos se aplicaron a la ecuación diferencial:

$$\frac{dy}{dx} = y\cos(x) - x^2$$

con la condición inicial $y(0) = 1$.

Objetivo: Aproximar la solución de la ecuación diferencial en el intervalo $x \in [0,1]$ con un paso $h = 0$.

Euler

Implementación del método de Euler para resolver una EDO. En cada paso (h = 0.1), calcula la solución aproximada ($y = y + h * f(x,y)$) y compara con la solución exacta, registrando el error porcentual.

Proceso en el Código:

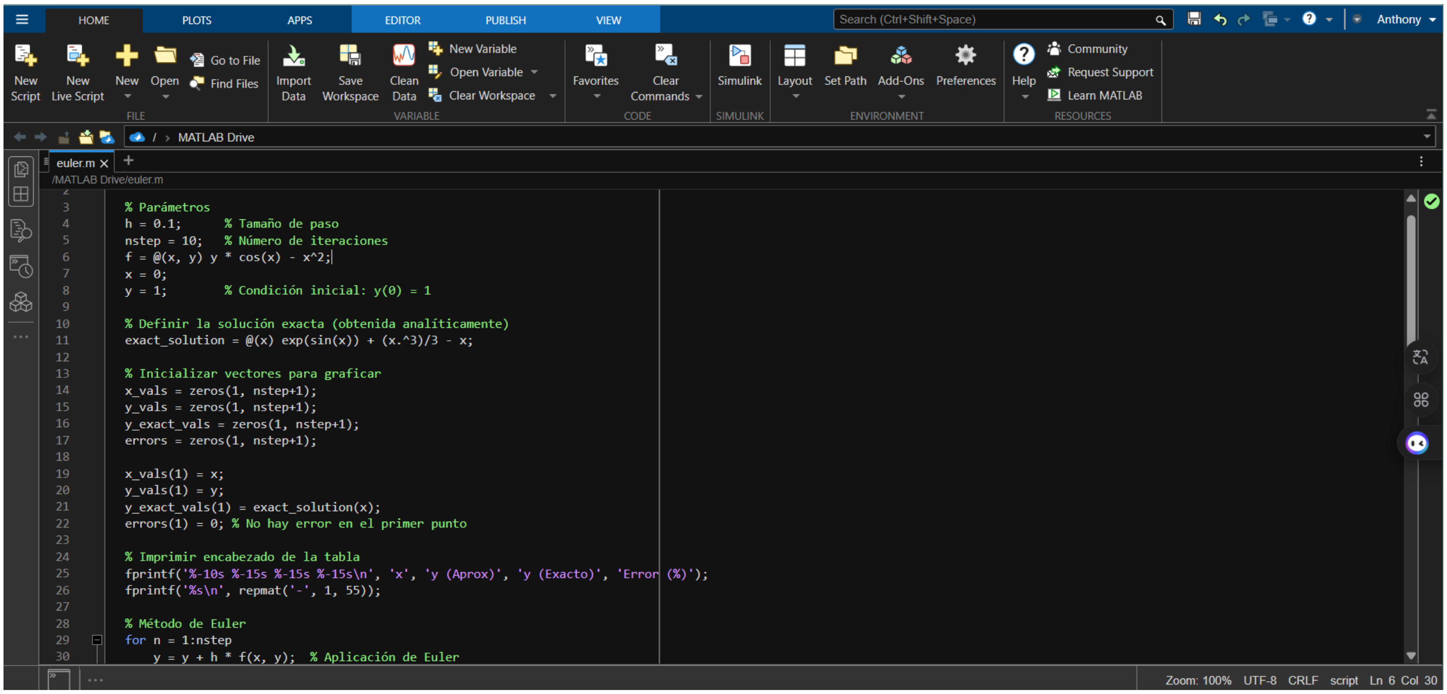
Inicialización: Se define h , condición inicial $(x0, y0)$, y la EDO $\frac{dy}{dx} = f(x,y)$.

Iteración:

- Calcula $yn + 1$ usando la fórmula de Euler.
- Compara con la solución exacta para calcular el error.

Salida: Tabla con valores de x , solución aproximada, exacta y error porcentual.

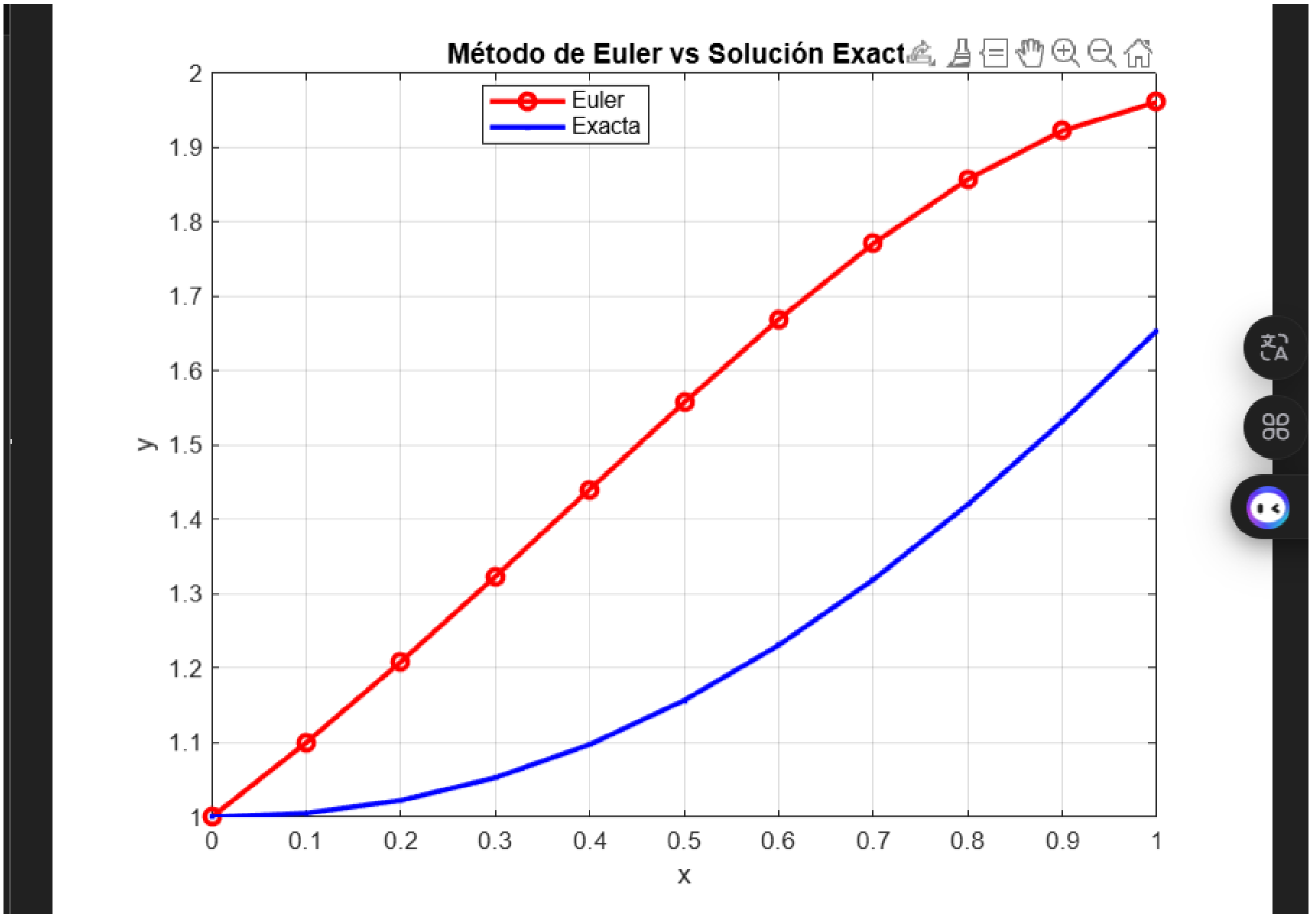
Limitaciones: Error alto debido a su naturaleza de primer orden $\left((O(h))\right)$.



Código:

```
% Parámetros
h = 0.1;        % Tamaño de paso
nstep = 10;     % Número de iteraciones
f = @(x, y) y * cos(x) - x^2;
x = 0;
y = 1;          % Condición inicial: y(0) = 1

% Definir la solución exacta (obtenida analíticamente)
exact_solution = @(x) exp(sin(x)) + (x.^3)/3 - x;

% Inicializar vectores para graficar
x_vals = zeros(1, nstep+1);
y_vals = zeros(1, nstep+1);
y_exact_vals = zeros(1, nstep+1);
errors = zeros(1, nstep+1);

x_vals(1) = x;
y_vals(1) = y;
y_exact_vals(1) = exact_solution(x);
errors(1) = 0; % No hay error en el primer punto

% Imprimir encabezado de la tabla
fprintf('%-10s %-15s %-15s %-15s\n', 'x', 'y (Aprox)', 'y (Exacto)', 'Error (%)');
fprintf('%s\n', repmat('-', 1, 55));

% Método de Euler
for n = 1:nstep
    y = y + h * f(x, y);   % Aplicación de Euler
    x = n * h;             % Actualizar x
    exact = exact_solution(x);  % Valor exacto
    error = 100 * abs(y - exact) / exact; % Error porcentual

    % Guardar valores para graficar
```

```
    x_vals(n+1) = x;
    y_vals(n+1) = y;
    y_exact_vals(n+1) = exact;
    errors(n+1) = error;

    % Imprimir en tabla
    fprintf('%-10.2f %-15.6f %-15.6f %-15.6f\n', x, y, exact, error);
end

% Graficar resultados
figure;
plot(x_vals, y_vals, 'ro-', 'LineWidth', 2, 'MarkerSize', 6);
hold on;
plot(x_vals, y_exact_vals, 'b.-', 'LineWidth', 2);
xlabel('x'); ylabel('y');
title('Método de Euler vs Solución Exacta');
legend('Euler', 'Exacta', 'Location', 'Best');
grid on;
```




```
% Método de Euler
for n = 1:nstep
    y = y + h * f(x, y);  % Aplicación de Euler
    x = n * h;            % Actualizar x
    exact = exact_solution(x);  % Valor exacto
    error = 100 * abs(y - exact) / exact; % Error porcentual

    % Guardar valores para graficar
    x_vals(n+1) = x;
    y_vals(n+1) = y;
    y_exact_vals(n+1) = exact;
    errors(n+1) = error;

    % Imprimir en tabla
```

Command Window

x	y (Aprox)	y (Exacto)	Error (%)
0.10	1.100000	1.005320	9.417879
0.20	1.208450	1.022445	18.192196
0.30	1.322887	1.052825	25.651114
0.40	1.440267	1.097455	31.236950
0.50	1.556924	1.156813	34.587375
0.60	1.668557	1.230819	35.564804
0.70	1.770269	1.318830	34.230287
0.80	1.856667	1.419675	30.781084
0.90	1.922022	1.531742	25.479493
1.00	1.960497	1.653110	18.594443

Euler mejorado

Aplicación del método de Euler mejorado (Heun), usando un promedio de pendientes para mayor precisión que Euler estándar. Calcula errores y almacena resultados en vectores para graficar.

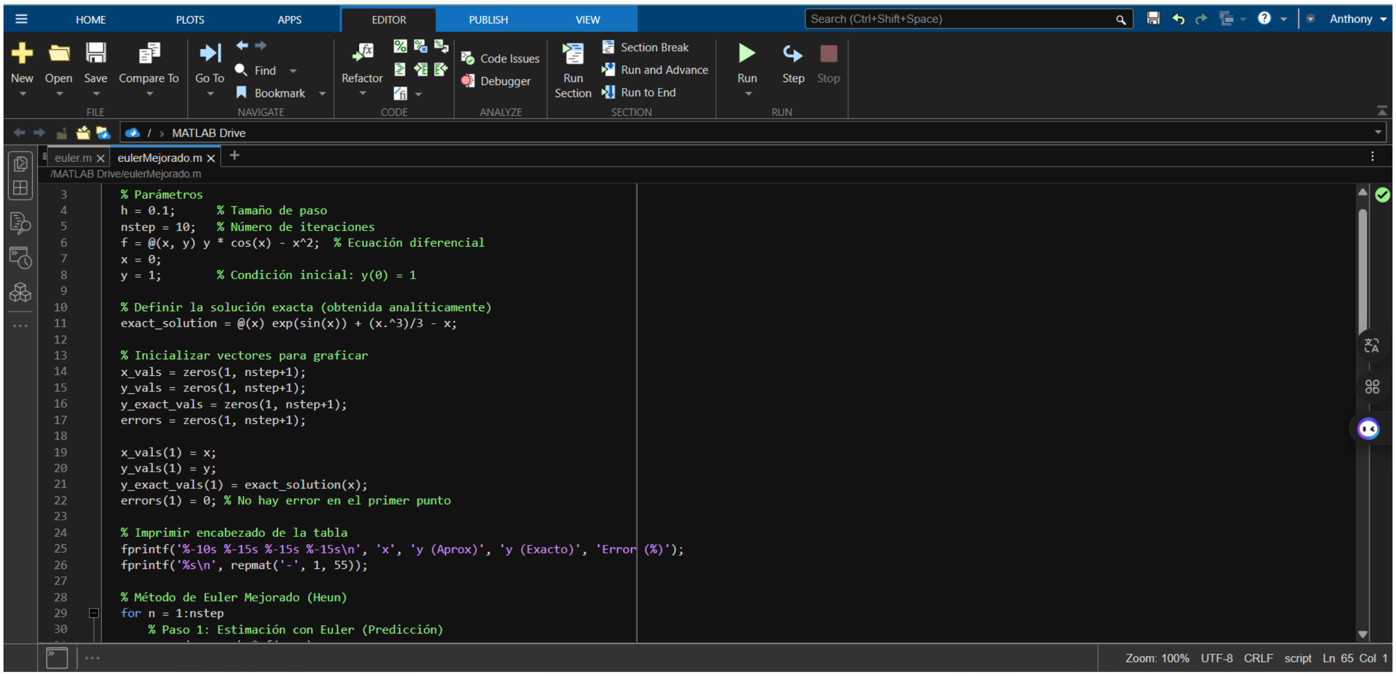
Proceso en el Código:

Pendiente inicial: Calcula $k1 = f(xn, yn)$.

Pendiente corregida:

- Estima $y^{\cdot} = yn + h \cdot k1$.
- Calcula $k2 = f(xn + 1, y^{\cdot})$.

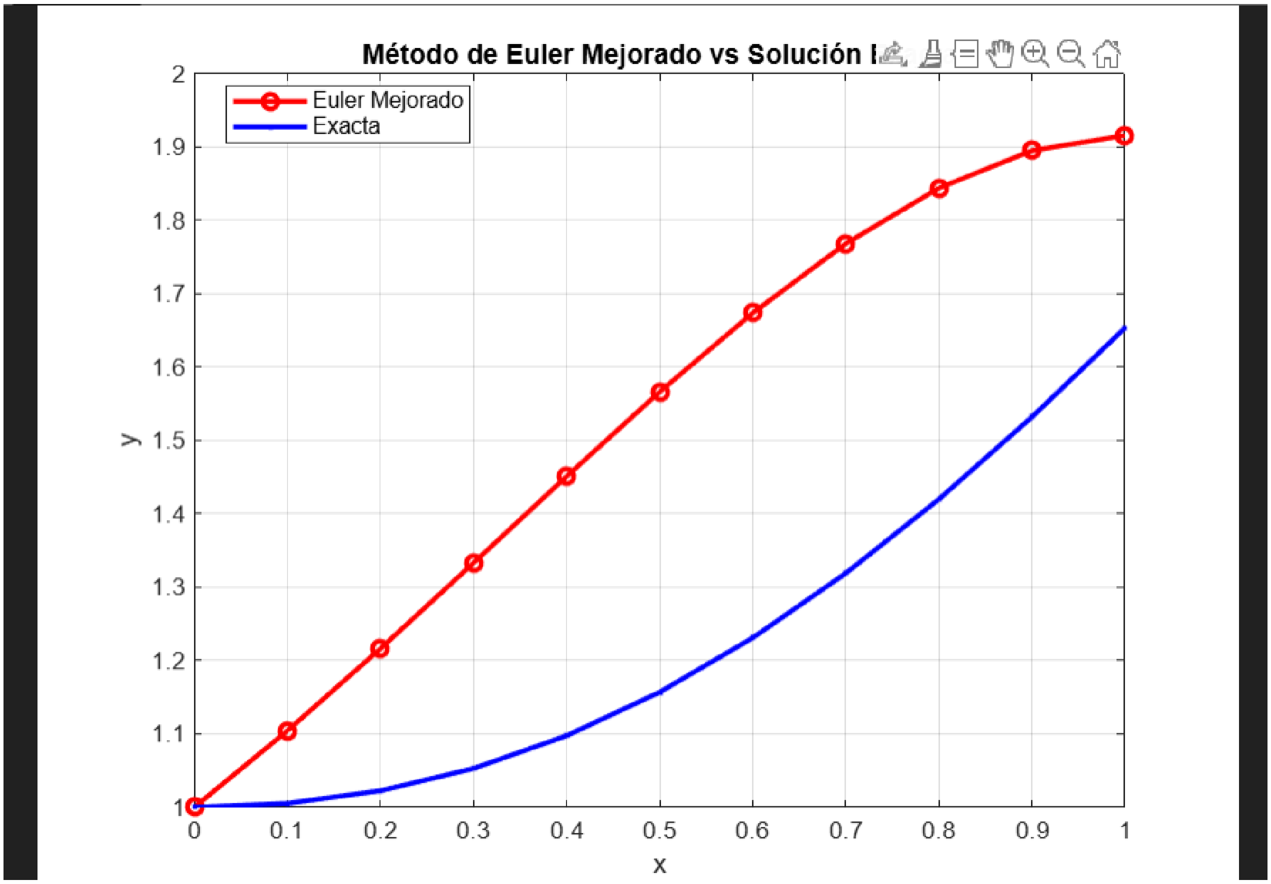
Actualización: Usa el promedio $\frac{(k1+k2)}{2}$ para mejorar la precisión.

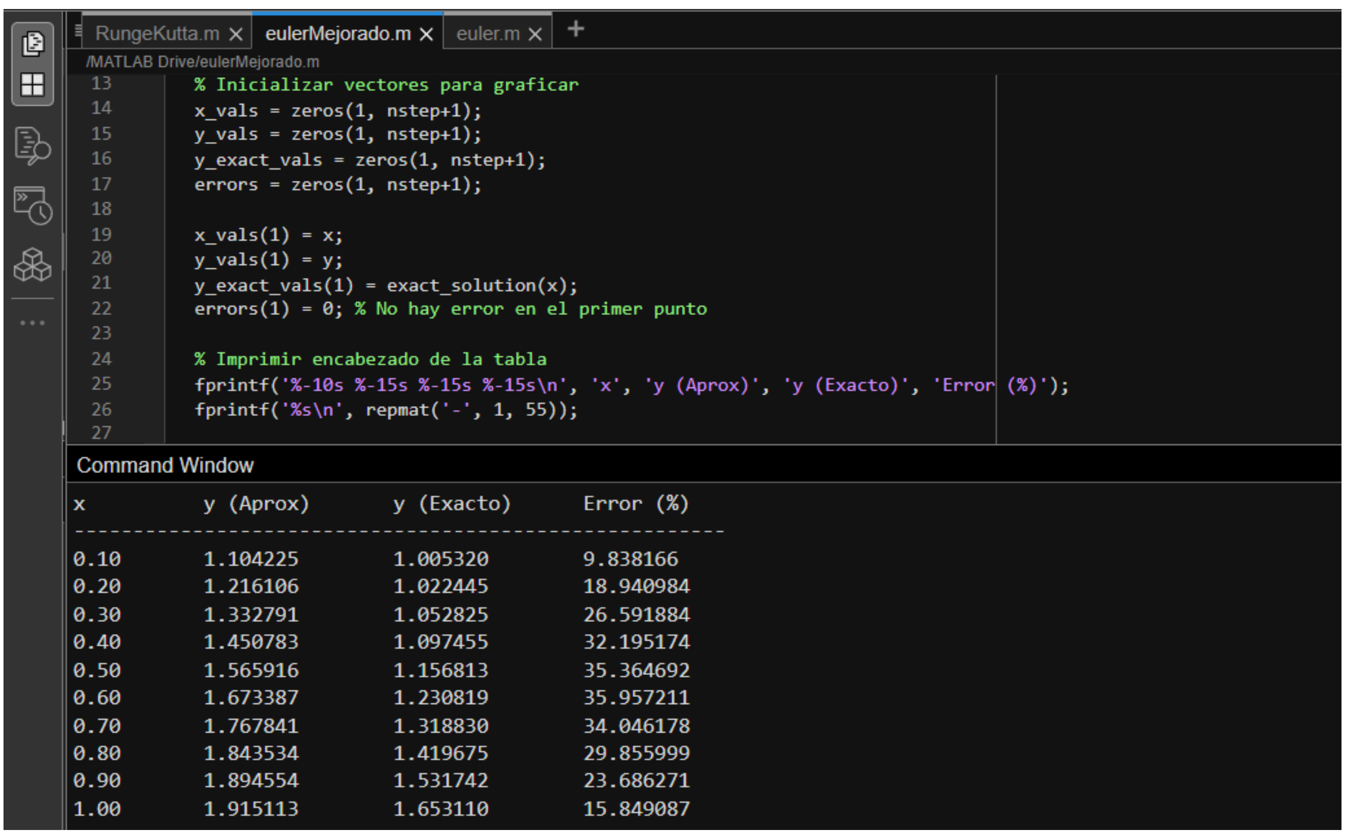


Código:

```
% Parámetros
h = 0.1;        % Tamaño de paso
nstep = 10;     % Número de iteraciones
f = @(x, y) y * cos(x) - x^2;   % Ecuación diferencial
x = 0;
y = 1;          % Condición inicial: y(0) = 1

% Definir la solución exacta (obtenida analíticamente)
exact_solution = @(x) exp(sin(x)) + (x.^3)/3 - x;

% Inicializar vectores para graficar
x_vals = zeros(1, nstep+1);
y_vals = zeros(1, nstep+1);
y_exact_vals = zeros(1, nstep+1);
errors = zeros(1, nstep+1);

x_vals(1) = x;
```

```
y_vals(1) = y;
y_exact_vals(1) = exact_solution(x);
errors(1) = 0; % No hay error en el primer punto

% Imprimir encabezado de la tabla
fprintf('%-10s %-15s %-15s %-15s\n', 'x', 'y (Aprox)', 'y (Exacto)', 'Error (%)');
fprintf('%s\n', repmat('-', 1, 55));

% Método de Euler Mejorado (Heun)
for n = 1:nstep
    % Paso 1: Estimación con Euler (Predicción)
    y_pred = y + h * f(x, y);

    % Paso 2: Cálculo de la pendiente corregida
    avg_slope = (f(x, y) + f(x + h, y_pred)) / 2;

    % Paso 3: Corrección usando el promedio de pendientes
    y = y + h * avg_slope;

    % Actualizar x
    x = n * h;

    % Calcular solución exacta y error
    exact = exact_solution(x);
    error = 100 * abs(y - exact) / exact;

    % Guardar valores para graficar
    x_vals(n+1) = x;
    y_vals(n+1) = y;
    y_exact_vals(n+1) = exact;
    errors(n+1) = error;

    % Imprimir en tabla
    fprintf('%-10.2f %-15.6f %-15.6f %-15.6f\n', x, y, exact, error);
end

% Graficar resultados
figure;
plot(x_vals, y_vals, 'ro-', 'LineWidth', 2, 'MarkerSize', 6);
hold on;
plot(x_vals, y_exact_vals, 'b.-', 'LineWidth', 2);
xlabel('x'); ylabel('y');
title('Método de Euler Mejorado vs Solución Exacta');
legend('Euler Mejorado', 'Exacta', 'Location', 'Best');
grid on;
```





RungeKutta.m × eulerMejorado.m × euler.m ×

/MATLAB Drive/eulerMejorado.m

```
% Inicializar vectores para graficar
x_vals = zeros(1, nstep+1);
y_vals = zeros(1, nstep+1);
y_exact_vals = zeros(1, nstep+1);
errors = zeros(1, nstep+1);

x_vals(1) = x;
y_vals(1) = y;
y_exact_vals(1) = exact_solution(x);
errors(1) = 0; % No hay error en el primer punto

% Imprimir encabezado de la tabla
fprintf('%-10s %-15s %-15s %-15s\n', 'x', 'y (Aprox)', 'y (Exacto)', 'Error (%)');
fprintf('%s\n', repmat('-', 1, 55));
```

Command Window

x	y (Aprox)	y (Exacto)	Error (%)
0.10	1.104225	1.005320	9.838166
0.20	1.216106	1.022445	18.940984
0.30	1.332791	1.052825	26.591884
0.40	1.450783	1.097455	32.195174
0.50	1.565916	1.156813	35.364692
0.60	1.673387	1.230819	35.957211
0.70	1.767841	1.318830	34.046178
0.80	1.843534	1.419675	29.855999
0.90	1.894554	1.531742	23.686271
1.00	1.915113	1.653110	15.849087

Runge-Kutta

Método de alto orden que usa cuatro cálculos intermedios para mejorar la precisión de la solución.

Proceso en el Código:

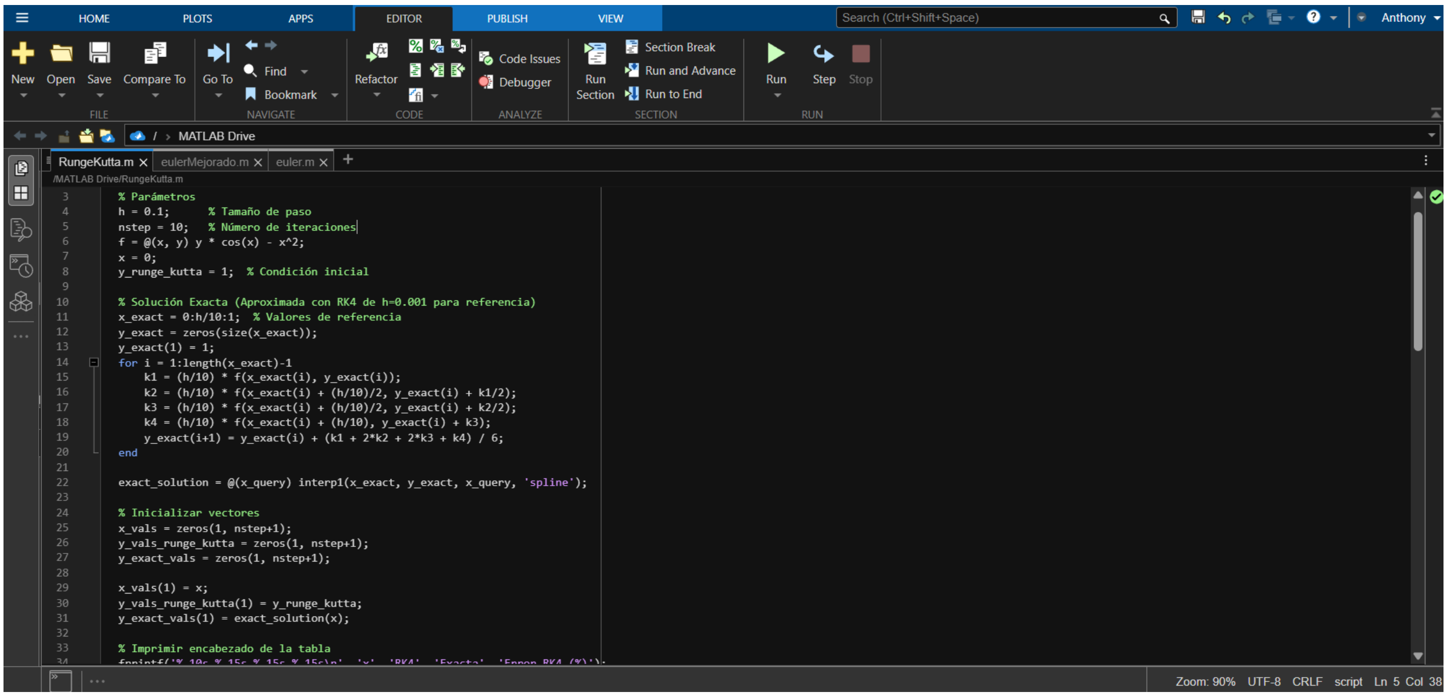
Inicialización: Se definen h, $(x0, y0)$ y la ecuación diferencial.

Iteración:

- Calcula cuatro pendientes $k1, k2, k3, k4$.
- Usa un promedio ponderado de estas pendientes para estimar $yn + 1$.

Salida: Tabla con valores de x, solución aproximada, exacta y error porcentual.

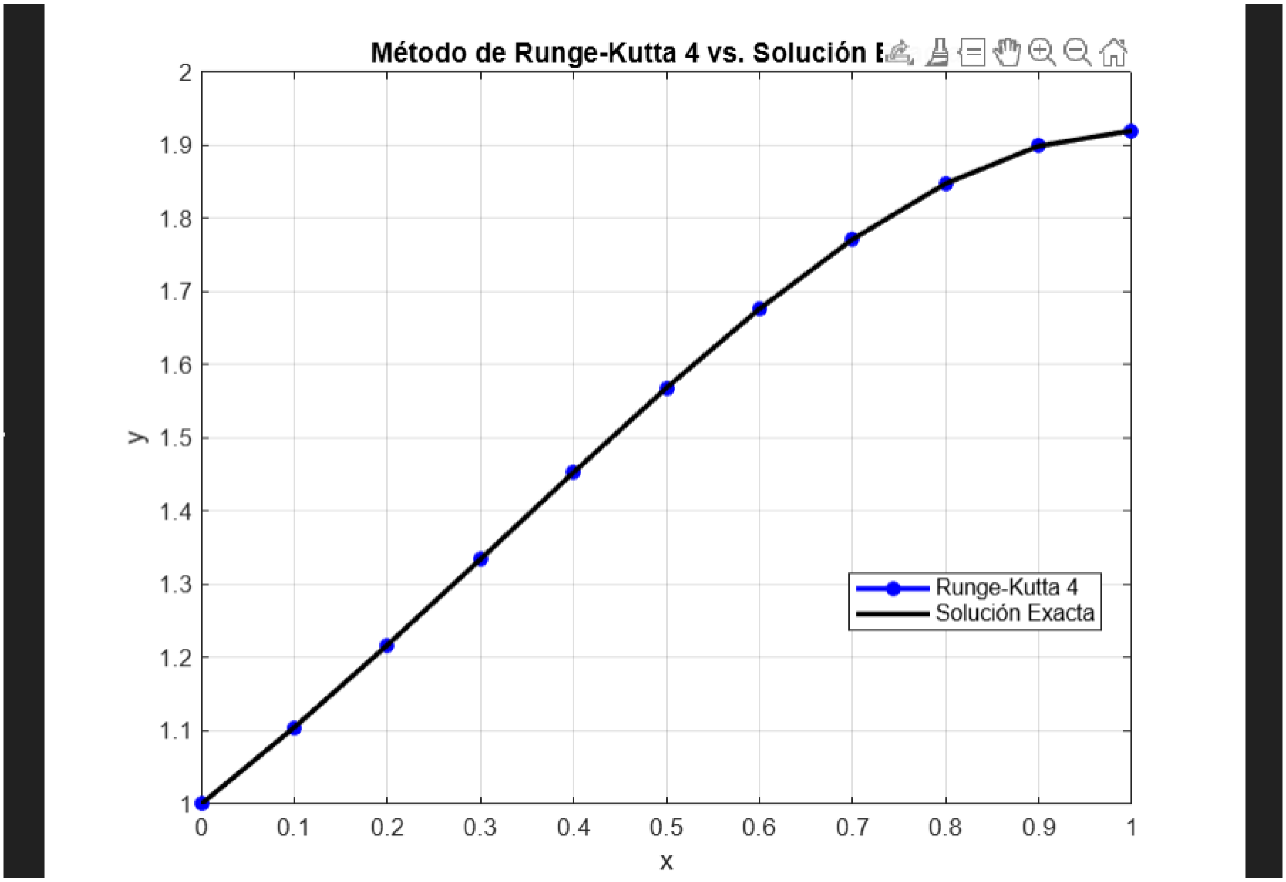
Ventajas: Mayor precisión que Euler y Euler mejorado, con error de cuarto orden $O(h\text{^}4)$

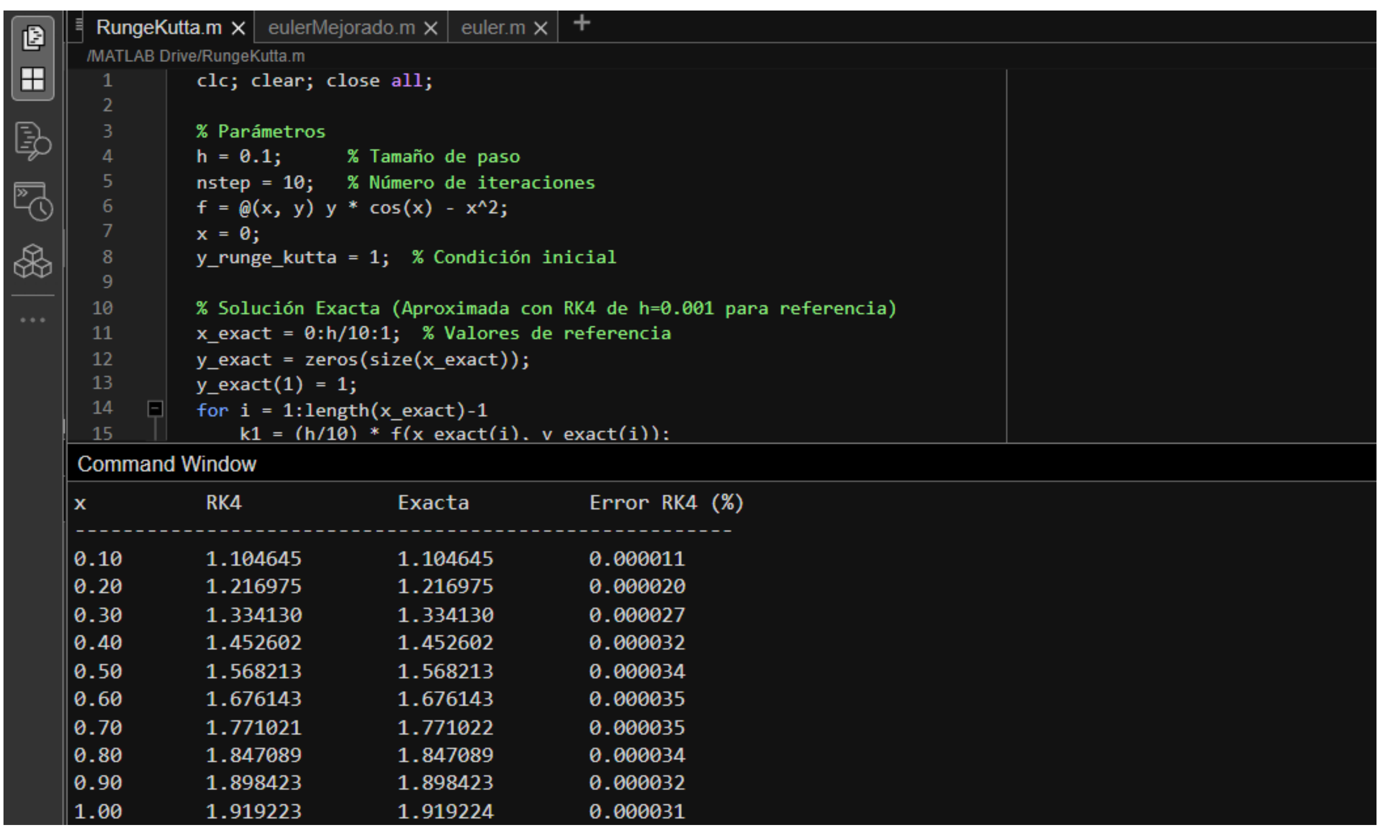


Código:

```
% Parámetros
h = 0.1;        % Tamaño de paso
nstep = 10;     % Número de iteraciones
f = @(x, y) y * cos(x) - x^2;
x = 0;
y_runge_kutta = 1;   % Condición inicial

% Solución Exacta (Aproximada con RK4 de h=0.001 para referencia)
x_exact = 0:h/10:1;   % Valores de referencia
y_exact = zeros(size(x_exact));
y_exact(1) = 1;
for i = 1:length(x_exact)-1
    k1 = (h/10) * f(x_exact(i), y_exact(i));
    k2 = (h/10) * f(x_exact(i) + (h/10)/2, y_exact(i) + k1/2);
    k3 = (h/10) * f(x_exact(i) + (h/10)/2, y_exact(i) + k2/2);
    k4 = (h/10) * f(x_exact(i) + (h/10), y_exact(i) + k3);
    y_exact(i+1) = y_exact(i) + (k1 + 2*k2 + 2*k3 + k4) / 6;
```

```
end

exact_solution = @(x_query) interp1(x_exact, y_exact, x_query, 'spline');

% Inicializar vectores
x_vals = zeros(1, nstep+1);
y_vals_runge_kutta = zeros(1, nstep+1);
y_exact_vals = zeros(1, nstep+1);

x_vals(1) = x;
y_vals_runge_kutta(1) = y_runge_kutta;
y_exact_vals(1) = exact_solution(x);

% Imprimir encabezado de la tabla
fprintf('%-10s %-15s %-15s %-15s\n', 'x', 'RK4', 'Exacta', 'Error RK4 (%)');
fprintf('%s\n', repmat('-', 1, 55));

for n = 1:nstep
    % Runge-Kutta 4 (RK4)
    k1 = h * f(x, y_runge_kutta);
    k2 = h * f(x + h/2, y_runge_kutta + k1/2);
    k3 = h * f(x + h/2, y_runge_kutta + k2/2);
    k4 = h * f(x + h, y_runge_kutta + k3);
    y_runge_kutta = y_runge_kutta + (k1 + 2*k2 + 2*k3 + k4) / 6;

    % Actualizar x
    x = n * h;

    % Calcular solución exacta y error de RK4
    exact = exact_solution(x);
    error_rk4 = 100 * abs(y_runge_kutta - exact) / exact;

    % Guardar valores para graficar
    x_vals(n+1) = x;
    y_vals_runge_kutta(n+1) = y_runge_kutta;
    y_exact_vals(n+1) = exact;

    % Imprimir valores en la tabla
    fprintf('%-10.2f %-15.6f %-15.6f %-15.6f\n', x, y_runge_kutta, exact, error_rk4);
end

% Graficar resultados
figure;
plot(x_vals, y_vals_runge_kutta, 'b-*', 'LineWidth', 2, 'MarkerSize', 6);
hold on;
plot(x_vals, y_exact_vals, 'k-', 'LineWidth', 2);
xlabel('x'); ylabel('y');
title('Método de Runge-Kutta 4 vs. Solución Exacta');
legend('Runge-Kutta 4', 'Solución Exacta', 'Location', 'Best');
grid on;
```






```
clc; clear; close all;

% Parámetros
h = 0.1;       % Tamaño de paso
nstep = 10;    % Número de iteraciones
f = @(x, y) y * cos(x) - x^2;
x = 0;
y_runge_kutta = 1;  % Condición inicial

% Solución Exacta (Aproximada con RK4 de h=0.001 para referencia)
x_exact = 0:h/10:1;  % Valores de referencia
y_exact = zeros(size(x_exact));
y_exact(1) = 1;
for i = 1:length(x_exact)-1
    k1 = (h/10) * f(x_exact(i), y_exact(i));
```

x	RK4	Exacta	Error RK4 (%)
0.10	1.104645	1.104645	0.000011
0.20	1.216975	1.216975	0.000020
0.30	1.334130	1.334130	0.000027
0.40	1.452602	1.452602	0.000032
0.50	1.568213	1.568213	0.000034
0.60	1.676143	1.676143	0.000035
0.70	1.771021	1.771022	0.000035
0.80	1.847089	1.847089	0.000034
0.90	1.898423	1.898423	0.000032
1.00	1.919223	1.919224	0.000031

Newton-Raphson

Método numérico para encontrar raíces de ecuaciones $f(x) = 0$ mediante iteraciones sucesivas.

Proceso en el Código:

Inicialización: Se establece una estimación inicial $x0$ y una tolerancia para la convergencia.

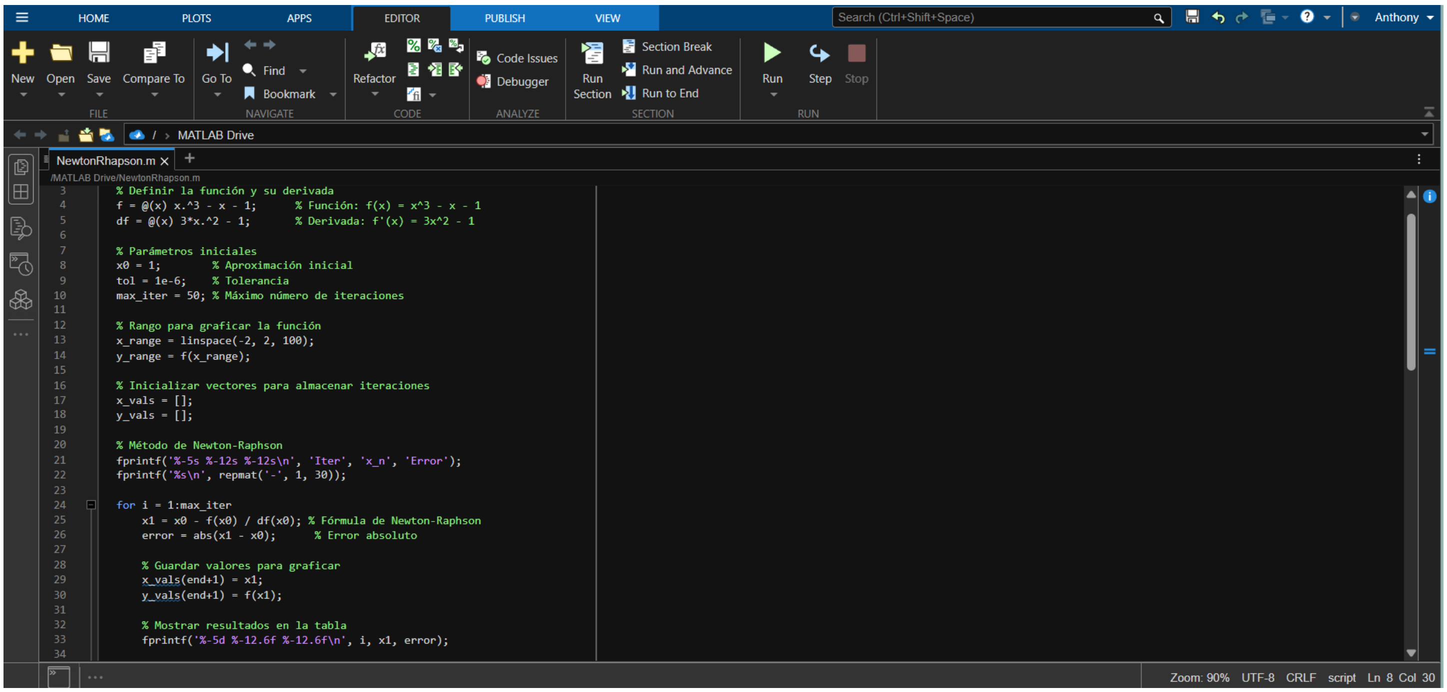
Iteración:

- Calcula $xn + 1 = xn - \frac{f(xn)}{f'(xn)}$
- Repite hasta alcanzar la tolerancia deseada.

Salida: Valor aproximado de la raíz con número de iteraciones.

Ventajas: Converge rápidamente si la estimación inicial es buena, con error cuadrático $O(h^2)$.

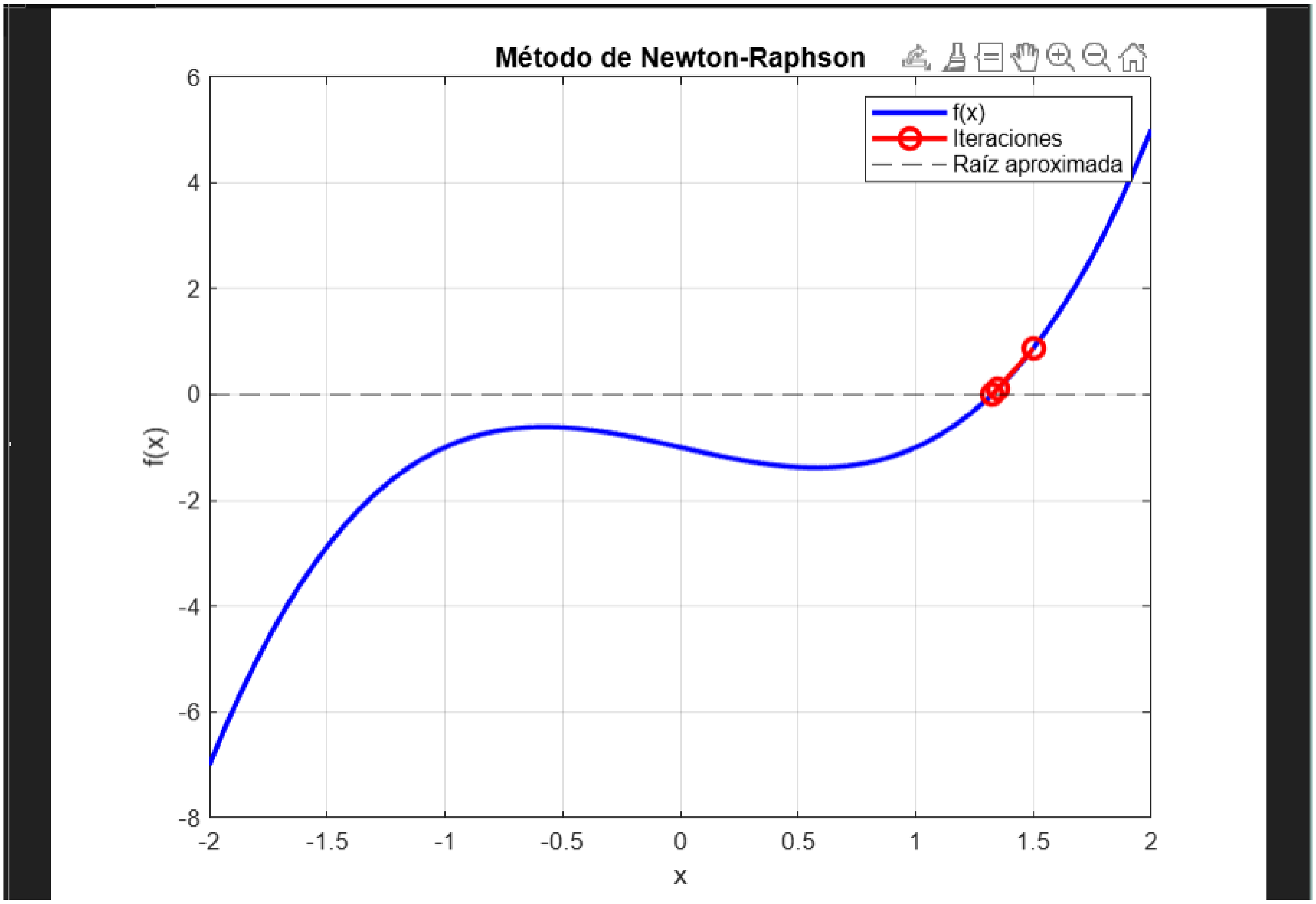
Limitaciones: Puede fallar si $f'(x)$ es muy pequeño o si la estimación inicial está lejos de la raíz.



Código:

```
% Definir la función y su derivada
f = @(x) x.^3 - x - 1;        % Función: f(x) = x^3 - x - 1
df = @(x) 3*x.^2 - 1;         % Derivada: f'(x) = 3x^2 - 1

% Parámetros iniciales
x0 = 1;         % Aproximación inicial
tol = 1e-6;     % Tolerancia
max_iter = 50; % Máximo número de iteraciones

% Rango para graficar la función
x_range = linspace(-2, 2, 100);
y_range = f(x_range);

% Inicializar vectores para almacenar iteraciones
x_vals = [];
```

```
y_vals = [];

% Método de Newton-Raphson
fprintf('%-5s %-12s %-12s\n', 'Iter', 'x_n', 'Error');
fprintf('%s\n', repmat('-', 1, 30));

for i = 1:max_iter
    x1 = x0 - f(x0) / df(x0); % Fórmula de Newton-Raphson
    error = abs(x1 - x0);      % Error absoluto

    % Guardar valores para graficar
    x_vals(end+1) = x1;
    y_vals(end+1) = f(x1);

    % Mostrar resultados en la tabla
    fprintf('%-5d %-12.6f %-12.6f\n', i, x1, error);

    % Criterio de convergencia
    if error < tol
        fprintf('\nRaíz encontrada: x = %.6f\n', x1);
        break;
    end

    x0 = x1; % Actualizar x0 para la siguiente iteración
end

% Verificar si no convergió
if i == max_iter
    fprintf('El método no convergió después de %d iteraciones.\n', max_iter);
end

% Graficar la función y el proceso de Newton-Raphson
figure;
plot(x_range, y_range, 'b', 'LineWidth', 2); hold on;
plot(x_vals, y_vals, 'ro-', 'MarkerSize', 8, 'LineWidth', 2);
yline(0, '--k'); % Línea horizontal en y = 0
xlabel('x'); ylabel('f(x)');
title('Método de Newton-Raphson');
legend('f(x)', 'Iteraciones', 'Raíz aproximada');
grid on;
```




```
% Definir la función y su derivada
f = @(x) x.^3 - x - 1;      % Función: f(x) = x^3 - x - 1
df = @(x) 3*x.^2 - 1;       % Derivada: f'(x) = 3x^2 - 1

% Parámetros iniciales
x0 = 1;         % Aproximación inicial
tol = 1e-6;     % Tolerancia
max_iter = 50; % Máximo número de iteraciones

% Rango para graficar la función
x_range = linspace(-2, 2, 100);
y_range = f(x_range);

% Inicializar vectores para almacenar iteraciones
x_vals = [];
```

Command Window

```
Iter  x_n           Error
-----------------------------
1     1.500000      0.500000
2     1.347826      0.152174
3     1.325200      0.022626
4     1.324718      0.000482
5     1.324718      0.000000

Raíz encontrada: x = 1.324718
```

Análisis Comparativo de los Métodos Numéricos

Al analizar los métodos implementados (Euler, Euler Mejorado, Runge-Kutta y Newton-Raphson), se pueden destacar sus diferencias en precisión, estabilidad y aplicabilidad.

Comparación de Métodos para Resolver EDOs
Los tres primeros métodos (Euler, Euler Mejorado y Runge-Kutta) se utilizan para resolver ecuaciones diferenciales ordinarias (EDOs).

- **Euler** es el más simple y rápido, pero también el menos preciso. Debido a su error de primer orden ($O(h)$), la aproximación se aleja rápidamente de la solución exacta conforme avanzamos en los pasos. Aunque es útil para comprender la idea básica de aproximación numérica, su uso en problemas más exigentes es limitado.

- **Euler Mejorado (Heun)** introduce una corrección basada en el promedio de pendientes, lo que reduce el error y mejora la precisión. Su error es de segundo orden $\left(O(h^2)\right)$, lo que significa que, con un paso de integración adecuado, proporciona mejores resultados sin un aumento significativo en el costo computacional.

- **Runge-Kutta de 4to Orden (RK4)** es el más preciso de los tres, con un error de cuarto orden $\left(O(h^4)\right)$. Aunque requiere más cálculos en cada iteración, la precisión obtenida justifica este costo computacional adicional. Este método es ampliamente utilizado en aplicaciones reales debido a su equilibrio entre precisión y eficiencia.

En términos generales, si la precisión es la prioridad y el costo computacional no es una limitación, RK4 es la mejor opción. Sin embargo, en casos donde se requiere rapidez y la precisión no es crítica, Euler Mejorado puede ser una alternativa viable. Euler simple, aunque útil para ilustrar conceptos, es poco recomendable en aplicaciones prácticas debido a su alto error.

Comparación con el Método de Newton-Raphson
A diferencia de los métodos anteriores, Newton-Raphson no resuelve EDOs, sino que encuentra raíces de ecuaciones no lineales.

- Su principal ventaja es su rápida convergencia cuando la estimación inicial está cerca de la raíz. Con un error de segundo orden $\left(O(h^2)\right)$, puede encontrar soluciones con pocas iteraciones en comparación con otros métodos de búsqueda de raíces.

- Sin embargo, **su éxito depende de una buena estimación inicial**. Si el punto de partida está demasiado lejos de la raíz, el método puede divergir o quedar atrapado en ciclos sin solución. También presenta problemas cuando la derivada de la función es muy pequeña, ya que esto puede provocar divisiones por valores cercanos a cero y generar inestabilidad numérica.

Puntos de Vista

Cada método tiene su propósito y aplicación ideal. Si bien Euler es el más básico, Runge-Kutta de 4to orden es el más recomendable para resolver EDOs debido a su alta precisión y estabilidad. Por otro lado, Newton-Raphson es excelente para encontrar raíces cuando se tiene una estimación inicial adecuada.

Desde un punto de vista práctico, la elección del método depende del balance entre precisión, estabilidad y costo computacional. En aplicaciones donde la eficiencia es clave y se pueden aceptar errores pequeños, Euler Mejorado puede ser suficiente. Sin embargo, para modelos más complejos donde la precisión es fundamental, Runge-Kutta y Newton-Raphson son las mejores opciones.

Proyecto

Esta aplicación fue desarrollada en MATLAB y tiene como objetivo resolver numéricamente una ecuación diferencial ordinaria (EDO) utilizando tres métodos distintos: Euler, Euler Mejorado y Runge-Kutta de cuarto orden. Se ha aplicado específicamente al modelo de crecimiento logístico de bacterias, el cual describe cómo una población de bacterias crece en función del tiempo considerando una tasa de crecimiento limitada por la capacidad de carga del entorno.

La ecuación diferencial que modela este fenómeno es:

$$\frac{dy}{dx} = r * y * (1 - \frac{y}{K})$$

Donde:

- y es la población de bacterias.
- r es la tasa de crecimiento.
- K es la capacidad de carga del entorno.
- x representa el tiempo.

Metodología para Resolver la Ecuación Diferencial

Método de Euler

El método de Euler es una aproximación simple basada en la ecuación:

$$y_n + 1 = y_n + h * f(x_n, y_n)$$

Es fácil de implementar, pero tiene un error relativamente grande, ya que usa solo la pendiente en el punto inicial para estimar el siguiente valor.

Método de Euler Mejorado

También conocido como método de Heun, mejora la precisión de Euler al usar una estimación en dos pasos:

$$y^* = y_n + h * f(x_n, y_n)$$

$$y_n + 1 = y_n + \frac{h * f(x_n, y_n) + f(x_n + 1, y^*)}{2}$$

Este método toma en cuenta una mejor estimación de la pendiente promediando dos valores.

Método de Runge-Kutta de Cuarto Orden (RK4)

RK4 es más preciso y calcula la solución usando una combinación ponderada de varias pendientes:

$$k_1 = f(x_n, y_n)$$

$$k_2 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2} k_1)$$

$$k_3 = f(x_n + \frac{h}{2}, y_n + \frac{h}{2} k_2)$$

$$k_4 = f(x_n + h, y_n + hk_3)$$

$$y_n + 1 = y_n + \frac{h(k_1 + 2k_2 + 2k_3 + k_4)}{6}$$

Este método ofrece una aproximación mucho más precisa que los anteriores.

Desarrollo del Código

El código implementa los tres métodos y los compara con la solución exacta del modelo logístico. Se han seguido los siguientes pasos:

Definición de Parámetros y Variables Iniciales

Se definen las condiciones iniciales:

- $x_0 = 0$ (tiempo inicial)
- $y_0 = 1$ (población inicial)
- $h = 0.1$ (paso de iteración)
- $nstep = 20$ (número de pasos)
- $r = 0.5$ (tasa de crecimiento)
- $k = 100$ (capacidad de carga)

Se define también la ecuación diferencial como una función anónima en MATLAB:

```
f = @(x,y) r * y * (1 - y/K);
```

Se establece la solución exacta para comparaciones:

```
exact_solution = @(x) K / (1 + ((K - y0) / y0) * exp(-r*x));
```

$$y(x) = \frac{k}{1 + \left(\frac{K - y_0}{y_0}\right) e^{-rx}}$$

Implementación de los Métodos Numéricos

Se inicializan variables para almacenar los valores de cada método y se ejecuta un bucle que calcula las soluciones paso a paso. Dentro del bucle, se aplican las fórmulas de los tres métodos y se calcula la solución exacta para comparar los resultados.

Los valores se almacenan en vectores para su posterior visualización.

Cálculo de Errores

Se calcula el error porcentual de cada método en comparación con la solución exacta:

```
error_euler = 100 * abs(y_euler - exact) / exact;
error_mejorado = 100 * abs(y_mejorado - exact) / exact;
error_rk = 100 * abs(y_rk - exact) / exact;
```

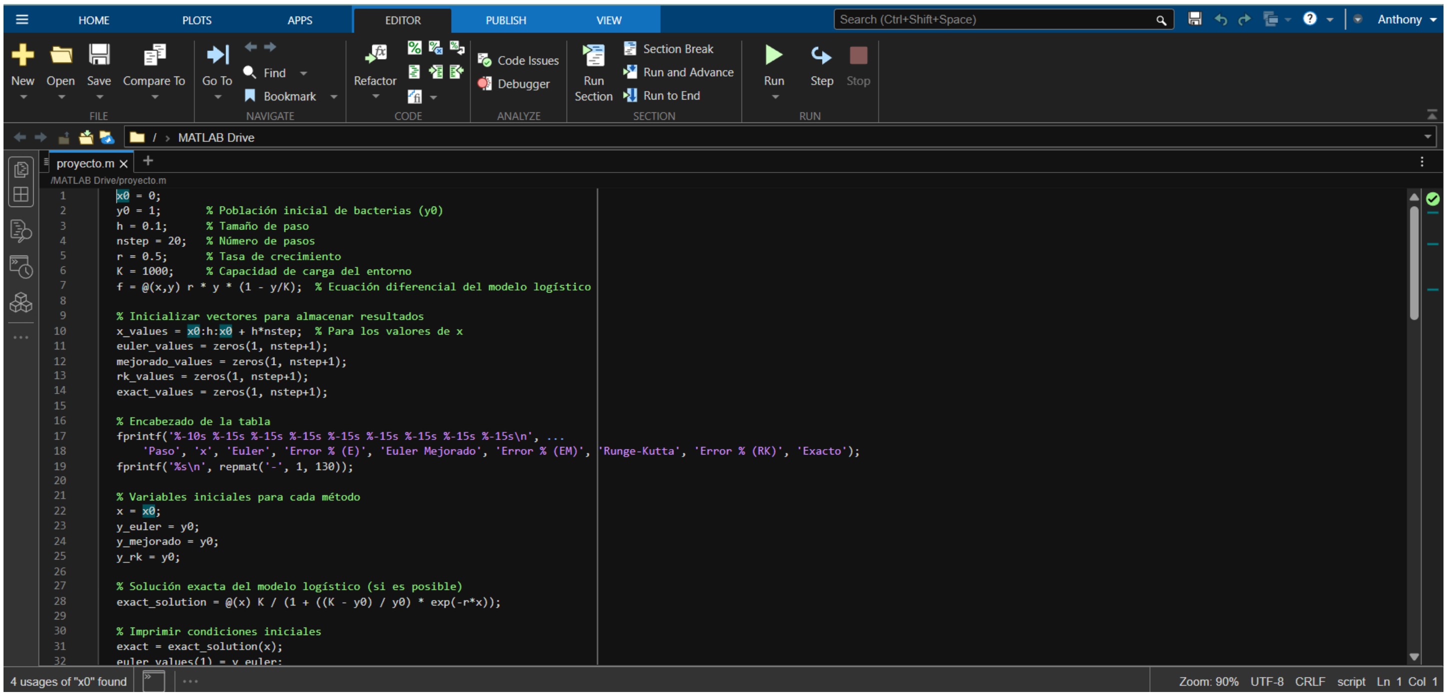
Visualización de Resultados

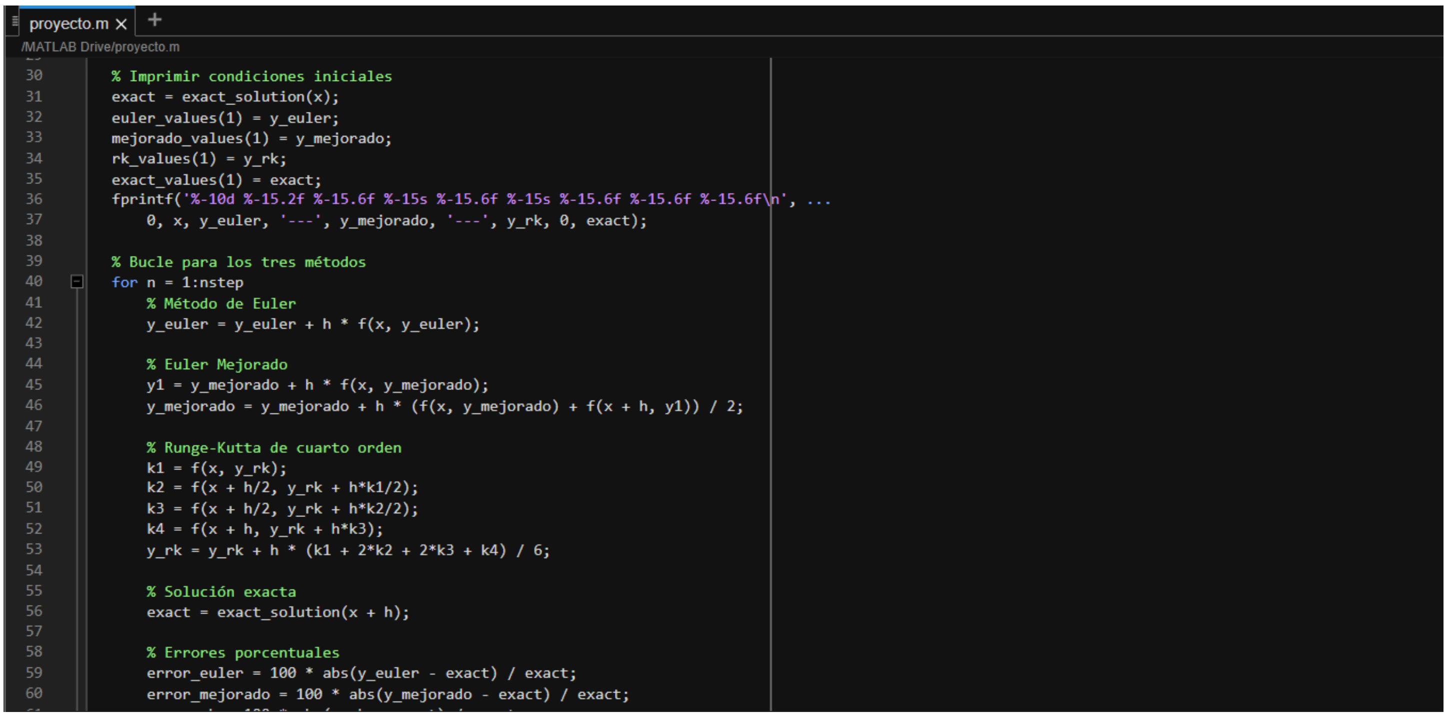
Los resultados se presentan en una tabla en la consola, y además, se genera una gráfica que muestra la evolución de la población de bacterias en función del tiempo para cada método junto con la solución exacta.

Se usa plot() en MATLAB para graficar los resultados:

```
figure;
hold on;
plot(x_values, euler_values, '-o', 'DisplayName', 'Euler');
plot(x_values, mejorado_values, '-s', 'DisplayName', 'Euler Mejorado');
plot(x_values, rk_values, '-^', 'DisplayName', 'Runge-Kutta');
plot(x_values, exact_values, '--', 'DisplayName', 'Exacto', 'LineWidth', 2);
legend('show');
grid on;
hold off;
```

Código


proyecto.m
/MATLAB Drive/proyecto.m
x0 = 0;
y0 = 1;   % Población inicial de bacterias (y0)
h = 0.1;   % Tamaño de paso
nstep = 20;   % Número de pasos
r = 0.5;   % Tasa de crecimiento
K = 1000;   % Capacidad de carga del entorno
f = @(x,y) r * y * (1 - y/K); % Ecuación diferencial del modelo logístico
% Inicializar vectores para almacenar resultados
x_values = x0:h:x0 + h*nstep; % Para los valores de x
euler_values = zeros(1, nstep+1);
mejorado_values = zeros(1, nstep+1);
rk_values = zeros(1, nstep+1);
exact_values = zeros(1, nstep+1);
% Encabezado de la tabla
fprintf('%-10s %-15s %-15s %-15s %-15s %-15s %-15s %-15s %-15s\n', ...
    'Paso', 'x', 'Euler', 'Error % (E)', 'Euler Mejorado', 'Error % (EM)', 'Runge-Kutta', 'Error % (RK)', 'Exacto');
fprintf('%s\n', repmat('-', 1, 130));
% Variables iniciales para cada método
x = x0;
y_euler = y0;
y_mejorado = y0;
y_rk = y0;
% Solución exacta del modelo logístico (si es posible)
exact_solution = @(x) K / (1 + ((K - y0) / y0) * exp(-r*x));
% Imprimir condiciones iniciales
exact = exact_solution(x);
4 usages of "x0" found
Zoom: 90% UTF-8 CRLF script Ln 1 Col 1


proyecto.m
/MATLAB Drive/proyecto.m
% Imprimir condiciones iniciales
exact = exact_solution(x);
euler_values(1) = y_euler;
mejorado_values(1) = y_mejorado;
rk_values(1) = y_rk;
exact_values(1) = exact;
fprintf('%-10d %-15.2f %-15.6f %-15s %-15.6f %-15s %-15.6f %-15.6f %-15.6f\n', ...
    0, x, y_euler, '---', y_mejorado, '---', y_rk, 0, exact);
% Bucle para los tres métodos
for n = 1:nstep
    % Método de Euler
    y_euler = y_euler + h * f(x, y_euler);
    % Euler Mejorado
    y1 = y_mejorado + h * f(x, y_mejorado);
    y_mejorado = y_mejorado + h * (f(x, y_mejorado) + f(x + h, y1)) / 2;
    % Runge-Kutta de cuarto orden
    k1 = f(x, y_rk);
    k2 = f(x + h/2, y_rk + h*k1/2);
    k3 = f(x + h/2, y_rk + h*k2/2);
    k4 = f(x + h, y_rk + h*k3);
    y_rk = y_rk + h * (k1 + 2*k2 + 2*k3 + k4) / 6;
    % Solución exacta
    exact = exact_solution(x + h);
    % Errores porcentuales
    error_euler = 100 * abs(y_euler - exact) / exact;
    error_mejorado = 100 * abs(y_mejorado - exact) / exact;

proyecto.m × +

/MATLAB Drive/proyecto.m

```
        % Errores porcentuales
        error_euler = 100 * abs(y_euler - exact) / exact;
        error_mejorado = 100 * abs(y_mejorado - exact) / exact;
        error_rk = 100 * abs(y_rk - exact) / exact;

        % Almacenar los resultados en los vectores
        euler_values(n+1) = y_euler;
        mejorado_values(n+1) = y_mejorado;
        rk_values(n+1) = y_rk;
        exact_values(n+1) = exact;

        % Imprimir los resultados en la tabla
        fprintf('%-10d %-15.2f %-15.6f %-15.4f %-15.6f %-15.4f %-15.6f %-15.4f %-15.6f\n', ...
            n, x + h, y_euler, error_euler, y_mejorado, error_mejorado, y_rk, error_rk, exact);

        % Avanzar en x
        x = x + h;
    end

    % Graficar los resultados
    figure;
    hold on;  % Mantener todas las gráficas en la misma figura
    plot(x_values, euler_values, '-o', 'DisplayName', 'Euler');
    plot(x_values, mejorado_values, '-s', 'DisplayName', 'Euler Mejorado');
    plot(x_values, rk_values, '-^', 'DisplayName', 'Runge-Kutta');
    plot(x_values, exact_values, '--', 'DisplayName', 'Exacto', 'LineWidth', 2);

    % Personalización de la gráfica
    xlabel('x');
    ylabel('Población de Bacterias (y)');
    title('Comparación de Métodos Numéricos - Modelo Logístico');
```

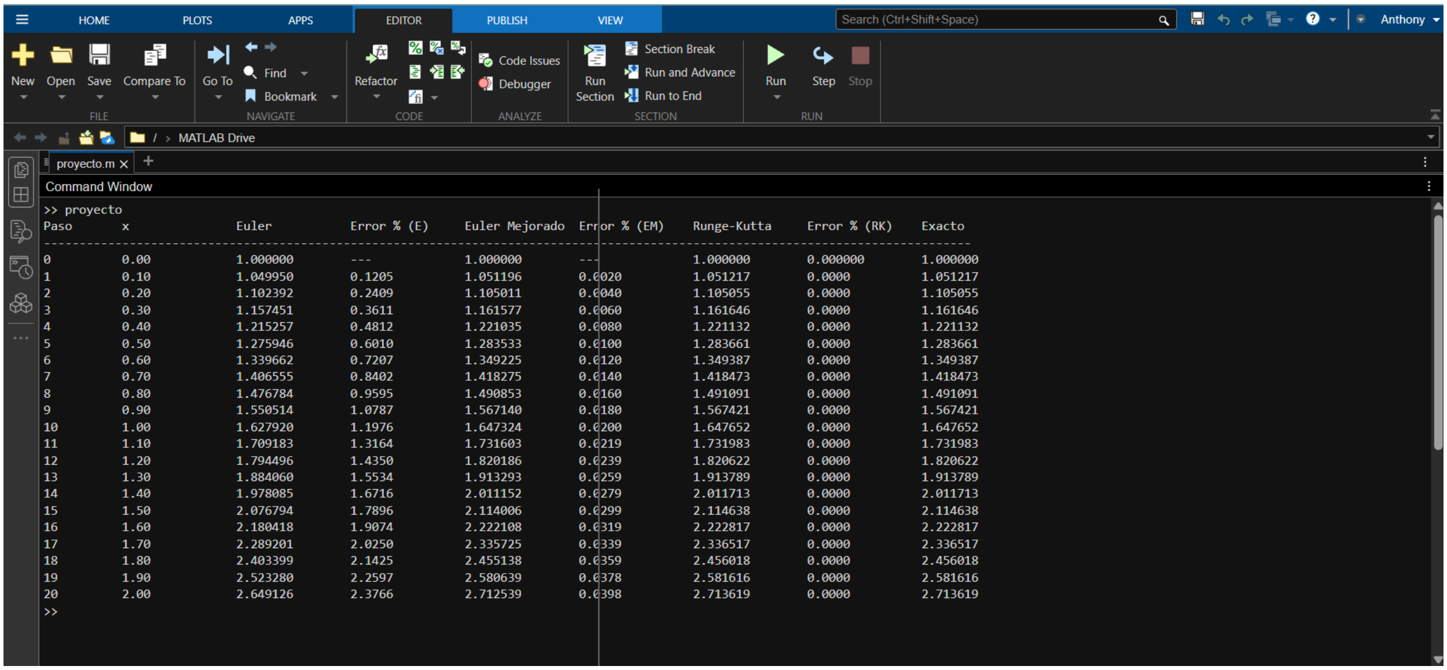
proyecto.m × +

/MATLAB Drive/proyecto.m

```
    % Graficar los resultados
    figure;
    hold on;  % Mantener todas las gráficas en la misma figura
    plot(x_values, euler_values, '-o', 'DisplayName', 'Euler');
    plot(x_values, mejorado_values, '-s', 'DisplayName', 'Euler Mejorado');
    plot(x_values, rk_values, '-^', 'DisplayName', 'Runge-Kutta');
    plot(x_values, exact_values, '--', 'DisplayName', 'Exacto', 'LineWidth', 2);

    % Personalización de la gráfica
    xlabel('x');
    ylabel('Población de Bacterias (y)');
    title('Comparación de Métodos Numéricos - Modelo Logístico');
```

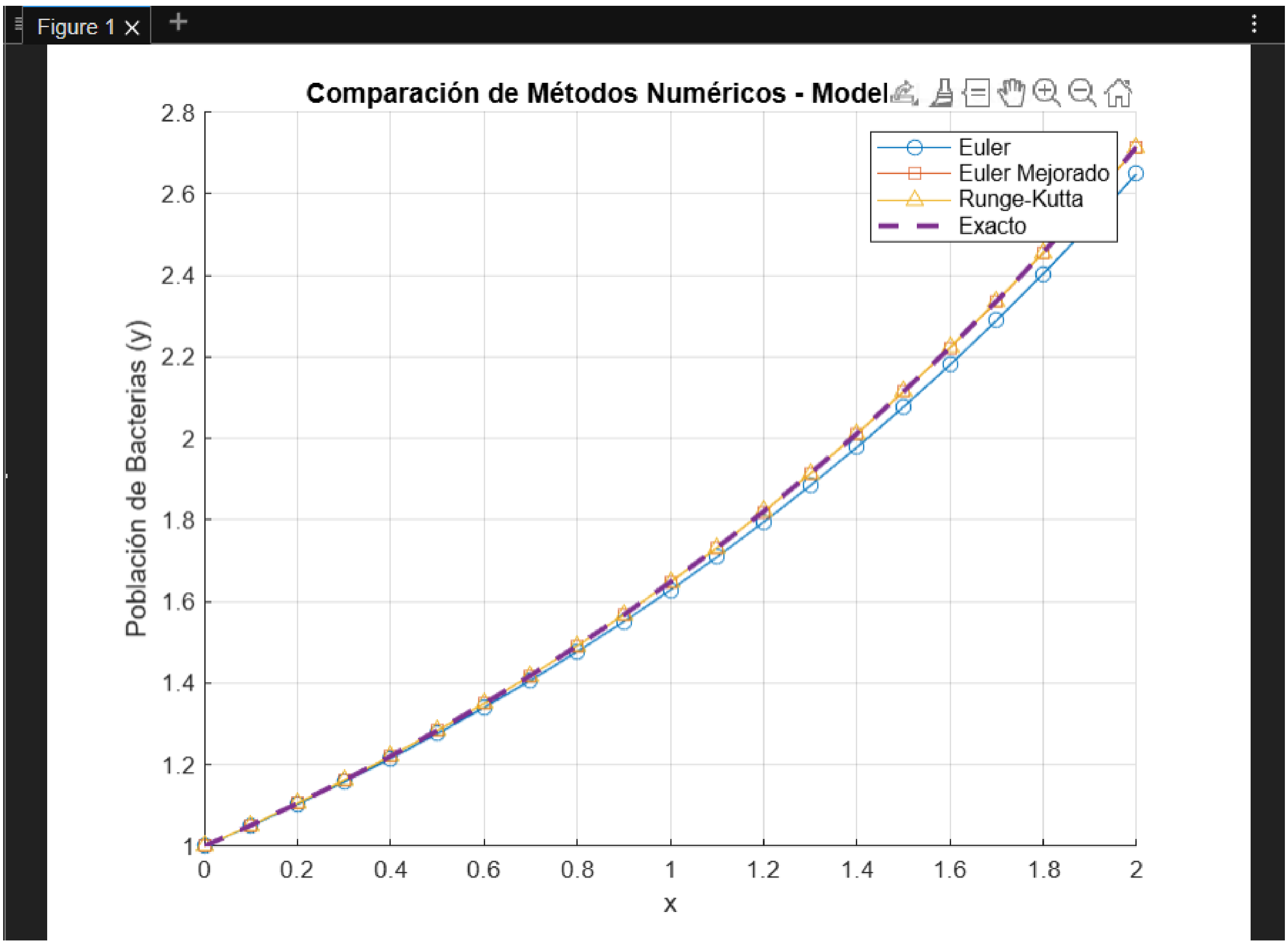

```
    legend('show');  % Mostrar leyenda
    grid on;  % Activar la cuadrícula

    hold off;  % Liberar la figura
```

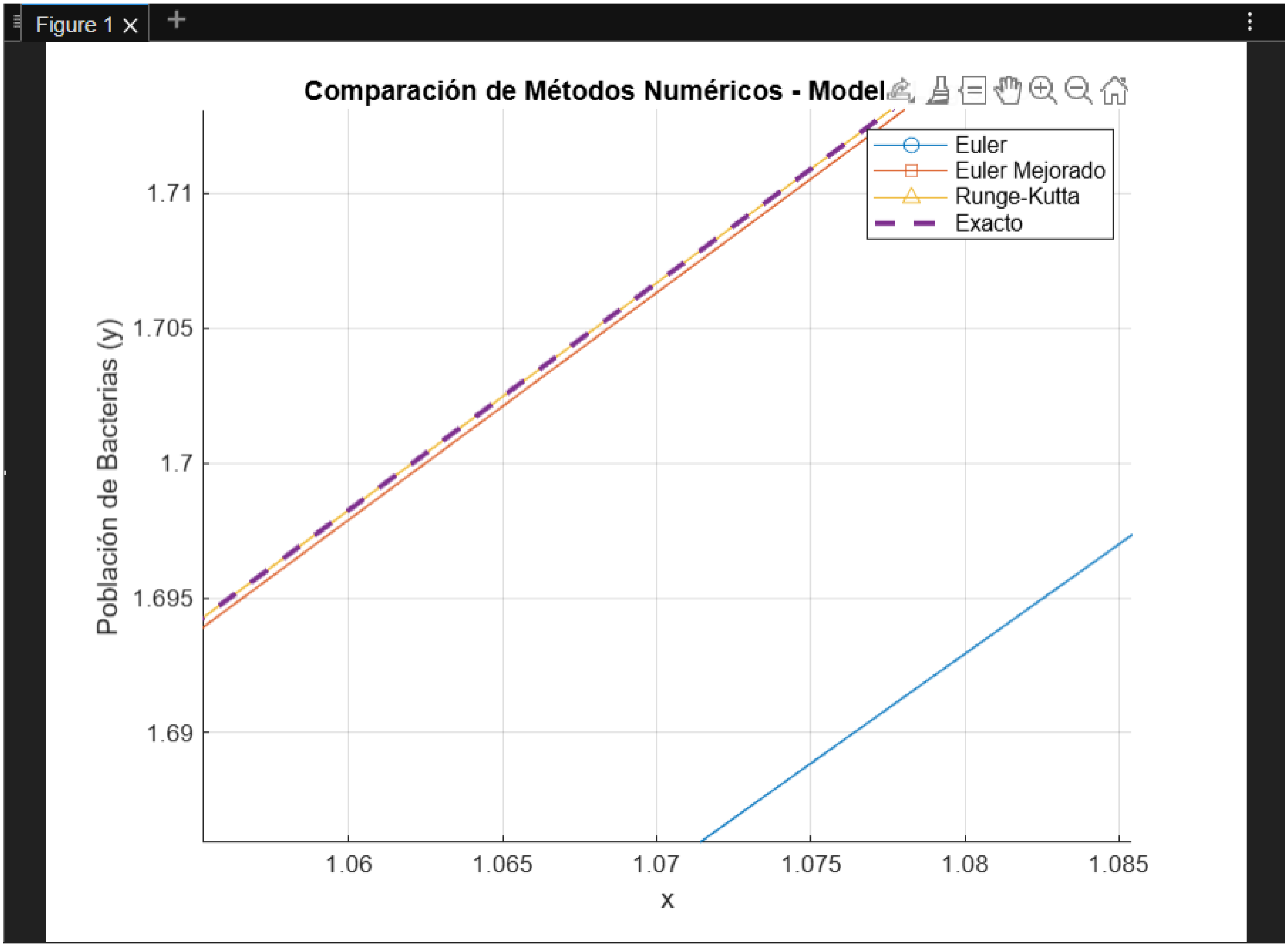
Resultados



Paso	x	Euler	Error % (E)	Euler Mejorado	Error % (EM)	Runge-Kutta	Error % (RK)	Exacto
0	0.00	1.000000	---	1.000000	---	1.000000	0.000000	1.000000
1	0.10	1.049950	0.1205	1.051196	0.0020	1.051217	0.0000	1.051217
2	0.20	1.102392	0.2409	1.105011	0.0040	1.105055	0.0000	1.105055
3	0.30	1.157451	0.3611	1.161577	0.0060	1.161646	0.0000	1.161646
4	0.40	1.215257	0.4812	1.221035	0.0080	1.221132	0.0000	1.221132
5	0.50	1.275946	0.6010	1.283533	0.0100	1.283661	0.0000	1.283661
6	0.60	1.339662	0.7207	1.349225	0.0120	1.349387	0.0000	1.349387
7	0.70	1.406555	0.8402	1.418275	0.0140	1.418473	0.0000	1.418473
8	0.80	1.476784	0.9595	1.490853	0.0160	1.491091	0.0000	1.491091
9	0.90	1.550514	1.0787	1.567140	0.0180	1.567421	0.0000	1.567421
10	1.00	1.627920	1.1976	1.647324	0.0200	1.647652	0.0000	1.647652
11	1.10	1.709183	1.3164	1.731603	0.0219	1.731983	0.0000	1.731983
12	1.20	1.794496	1.4350	1.820186	0.0239	1.820622	0.0000	1.820622
13	1.30	1.884060	1.5534	1.913293	0.0259	1.913789	0.0000	1.913789
14	1.40	1.978085	1.6716	2.011152	0.0279	2.011713	0.0000	2.011713
15	1.50	2.076794	1.7896	2.114006	0.0299	2.114638	0.0000	2.114638
16	1.60	2.180418	1.9074	2.222108	0.0319	2.222817	0.0000	2.222817
17	1.70	2.289201	2.0250	2.335725	0.0339	2.336517	0.0000	2.336517
18	1.80	2.403399	2.1425	2.455138	0.0359	2.456018	0.0000	2.456018
19	1.90	2.523280	2.2597	2.580639	0.0378	2.581616	0.0000	2.581616
20	2.00	2.649126	2.3766	2.712539	0.0398	2.713619	0.0000	2.713619

Gráfica:

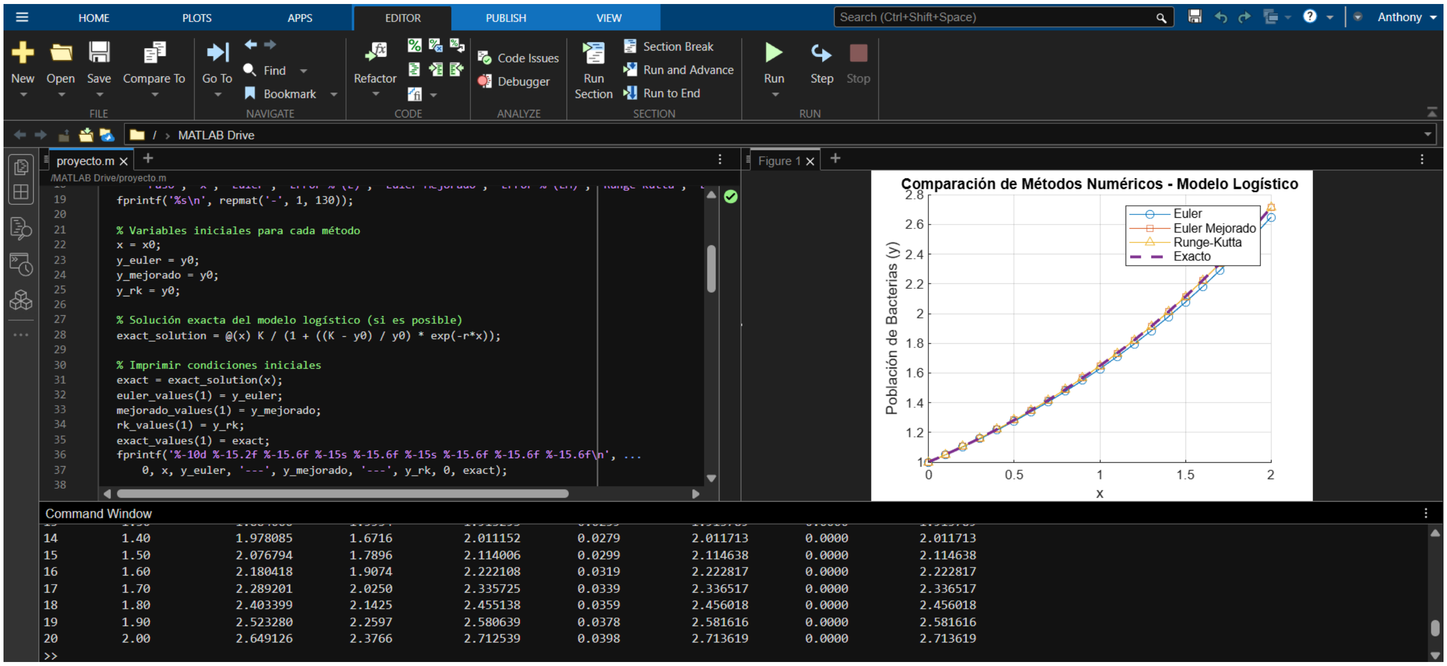




Análisis de Resultados

La gráfica permite observar que:

- **El método de Euler** tiende a presentar desviaciones significativas a medida que aumenta x. Esto se debe a su naturaleza de primer orden, que introduce errores acumulativos.
- **El método de Euler Mejorado** reduce el error de Euler al utilizar un paso intermedio, proporcionando una mejor aproximación.
- **El método de Runge-Kutta** se ajusta mucho más a la solución exacta, mostrando que es el más preciso de los tres.



El desarrollo de esta aplicación ha permitido comparar distintos métodos numéricos para la solución de ecuaciones diferenciales. Se observa que métodos más avanzados como **Runge-Kutta** ofrecen mayor precisión en comparación con **Euler** y **Euler Mejorado**.

El estudio de la ecuación diferencial aplicada al crecimiento bacteriano demuestra cómo estas técnicas pueden utilizarse en problemas biológicos y científicos, facilitando la predicción del comportamiento de sistemas dinámicos en el tiempo.