# Problem 1

## Table of answer

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Stirling’s approximation | Absolute error | Relative error |
| 1 | 1 | 0.922 | 0.078 | 0.078 |
| 2 | 2 | 1.919 | 0.081 | 0.040 |
| 3 | 6 | 5.836 | 0.164 | 0.027 |
| 4 | 24 | 23.506 | 0.494 | 0.021 |
| 5 | 120 | 118.019 | 1.981 | 0.017 |
| 6 | 720 | 710.078 | 9.922 | 0.014 |
| 7 | 5040 | 4980.396 | 59.604 | 0.012 |
| 8 | 40320 | 39902.395 | 417.605 | 0.010 |
| 9 | 362880 | 359536.873 | 3343.127 | 0.009 |
| 10 | 3628800 | 3598695.619 | 30104.381 | 0.008 |

From the table, the relative error of the Stirling approximation decreases as n increases. This means that the accuracy of the Stirling approximation increases as n increases.

## Code

% n n! Stirling Approx Absolute Error Relative Error

% -----------------------------------------------------------------------

% 1 1 0.922137 0.077863 0.077863

% 2 2 1.919004 0.080996 0.040498

% 3 6 5.836210 0.163790 0.027298

% 4 24 23.506175 0.493825 0.020576

% 5 120 118.019168 1.980832 0.016507

% 6 720 710.078185 9.921815 0.013780

% 7 5040 4980.395832 59.604168 0.011826

% 8 40320 39902.395453 417.604547 0.010357

% 9 362880 359536.872842 3343.127158 0.009213

% 10 3628800 3598695.618741 30104.381259 0.008296

n\_values = 1:10;

fprintf(' n n! Stirling Approx Absolute Error Relative Error\n');

fprintf('-----------------------------------------------------------------------\n');

for i = 1:length(n\_values)

n = n\_values(i);

n\_fact = factorial(n);

stirling\_val = stirling\_approx(n);

absolute\_error = abs(n\_fact - stirling\_val);

relative\_error = absolute\_error / n\_fact;

fprintf('%2d %7d %14f %12f %7f\n', n, n\_fact, stirling\_val, absolute\_error, relative\_error);

end

function sa = stirling\_approx(n)

sa = n^n \* exp(-n) \* sqrt(2 \* pi \* n);

end

# Problem 2

## Table of Answer

|  |  |  |
| --- | --- | --- |
|  |  | Relative error |
| 6 | 1.000000000 | 0.14159265359 |
| 12 | 0.517638090 | 0.03576411236 |
| 24 | 0.261052384 | 0.00896404031 |
| 48 | 0.130806258 | 0.00224245054 |
| 96 | 0.065438166 | 0.00056070270 |
| 192 | 0.032723463 | 0.00014018130 |
| 384 | 0.016362279 | 0.00003504568 |
| 768 | 0.008181208 | 0.00000876144 |
| 1536 | 0.004090613 | 0.00000219035 |
| 3072 | 0.002045307 | 0.00000054755 |
| 6144 | 0.001022654 | 0.00000013700 |
| 12288 | 0.000511327 | 0.00000003495 |
| 24576 | 0.000255663 | 0.00000000827 |
| 49152 | 0.000127832 | 0.00000000827 |
| 98304 | 0.000063916 | 0.00000000827 |
| 196608 | 0.000031958 | 0.00000000827 |
| 393216 | 0.000015979 | 0.00000101626 |
| 786432 | 0.000007989 | 0.00000034978 |
| 1572864 | 0.000003995 | 0.00001604264 |
| 3145728 | 0.000001997 | 0.00000581393 |
| 6291456 | 0.000000999 | 0.00008161143 |
| 12582912 | 0.000000499 | 0.00008161143 |

% Initialize variables

n = 6;

Ln = 1; % Starting value for L\_6

pi\_true = pi; % MATLAB's built-in value of pi

% Initialize a matrix to store the results

results = [];

% Iterate to calculate

for i = 0:21

n\_current = n \* 2^i; % Current number of sides

pi\_approx = n\_current \* Ln / 2; % Approximation of pi

abs\_error = abs(pi\_true - pi\_approx); % Absolute error

% Store the results in the matrix

results = [results; n\_current, Ln, abs\_error];

% Update Ln for the next iteration

Ln = next\_side\_length(Ln);

end

% Write results to a CSV file

writematrix(results, 'pi\_approximations.csv');

% Define a function to calculate L\_{2n} from L\_n

function Ln = next\_side\_length(Ln)

Ln = sqrt(2 - sqrt(4 - Ln^2));

End

## A Different Form of Equation (\*).

We have

We can rewrite this expression in the form of , where and need to be determined. In this case, we can set and .

Now, using the given formula, we can rewrite it as:

## New Table

|  |  |  |
| --- | --- | --- |
|  |  | Relative error |
| 6 | 1.000000000 | 0.14159265359 |
| 12 | 0.517638090 | 0.03576411236 |
| 24 | 0.261052384 | 0.00896404031 |
| 48 | 0.130806258 | 0.00224245054 |
| 96 | 0.065438166 | 0.00056070270 |
| 192 | 0.032723463 | 0.00014018130 |
| 384 | 0.016362279 | 0.00003504568 |
| 768 | 0.008181208 | 0.00000876144 |
| 1536 | 0.004090613 | 0.00000219036 |
| 3072 | 0.002045307 | 0.00000054759 |
| 6144 | 0.001022654 | 0.00000013690 |
| 12288 | 0.000511327 | 0.00000003422 |
| 24576 | 0.000255663 | 0.00000000856 |
| 49152 | 0.000127832 | 0.00000000214 |
| 98304 | 0.000063916 | 0.00000000053 |
| 196608 | 0.000031958 | 0.00000000013 |
| 393216 | 0.000015979 | 0.00000000003 |
| 786432 | 0.000007989 | 0.00000000001 |
| 1572864 | 0.000003995 | 0.00000000000 |
| 3145728 | 0.000001997 | 0.00000000000 |
| 6291456 | 0.000000999 | 0.00000000000 |
| 12582912 | 0.000000499 | 0.00000000000 |

% Initialize variables

n = 6;

Ln = 1; % Starting value for L\_6

pi\_true = pi; % MATLAB's built-in value of pi

% Initialize a matrix to store the results

results = [];

% Iterate to calculate

for i = 0:21

n\_current = n \* 2^i; % Current number of sides

pi\_approx = n\_current \* Ln / 2; % Approximation of pi

abs\_error = abs(pi\_true - pi\_approx); % Absolute error

% Store the results in the matrix

results = [results; n\_current, Ln, abs\_error];

% Update Ln for the next iteration

Ln = next\_side\_length\_new(Ln);

end

% Write results to a CSV file

writematrix(results, 'pi\_approximations\_2.csv');

function Ln = next\_side\_length\_new(Ln)

Ln = sqrt(Ln^2 / (2 + sqrt(4 - Ln^2)));

end

## Compare

It can be seen that the method using the original formula may result in two very similar numbers being subtracted to give a number very close to zero, which is , making the calculation less accurate. The method of calculation using the new formula avoids this problem and therefore improves the accuracy of the calculation.

# Problem 3

## Euler’s Method

% Time Points, Approximate Values, and Errors:

% t = 0.000000, y = 1.000000, Error = 0.000000

% t = 0.100000, y = 1.100000, Error = 0.005171

% t = 0.200000, y = 1.209485, Error = 0.011917

% t = 0.300000, y = 1.329254, Error = 0.020605

% t = 0.400000, y = 1.460150, Error = 0.031675

% t = 0.500000, y = 1.603065, Error = 0.045656

% t = 0.600000, y = 1.758932, Error = 0.063187

% t = 0.700000, y = 1.928726, Error = 0.085027

% t = 0.800000, y = 2.113455, Error = 0.112086

% t = 0.900000, y = 2.314156, Error = 0.145447

% t = 1.000000, y = 2.531887, Error = 0.186395

% Parameters

a = 0; % Start of the interval

b = 1; % End of the interval

alpha = 1; % Initial condition y(0) = 1

N = 10; % Number of steps

% Solve the differential equation using Euler's method

[T, Y, errors] = euler\_method(@diff\_eq, a, b, alpha, N);

% Display the results and errors

disp('Time Points, Approximate Values, and Errors:');

for i = 1:length(T)

fprintf('t = %f, y = %f, Error = %f\n', T(i), Y(i), errors(i));

end

% Define the function for the differential equation y' = y^2 \* exp(-t)

function dy = diff\_eq(t, y)

dy = y^2 \* exp(-t);

end

% Euler's method implementation

function [T, Y, errors] = euler\_method(f, a, b, alpha, N)

h = (b - a) / N; % Calculate the step size

T = a:h:b; % Create a vector of time points

Y = zeros(1, length(T)); % Initialize the solution vector

Y(1) = alpha; % Set the initial condition

errors = zeros(1, length(T)); % Initialize the error vector

% Iterate over each time step

for i = 1:length(T) - 1

t = T(i);

Y(i + 1) = Y(i) + h \* f(t, Y(i)); % Euler's formula

exact\_val = exp(T(i + 1)); % Exact value

errors(i + 1) = abs(Y(i + 1) - exact\_val); % Calculate error

end

end

## Second-order Taylor Method

% Time Points, Approximate Values, and Errors:

% t = 0.000000, y = 1.000000, Error = 0.000000

% t = 0.100000, y = 1.105000, Error = 0.000171

% t = 0.200000, y = 1.221005, Error = 0.000397

% t = 0.300000, y = 1.349165, Error = 0.000694

% t = 0.400000, y = 1.490748, Error = 0.001077

% t = 0.500000, y = 1.647153, Error = 0.001569

% t = 0.600000, y = 1.819923, Error = 0.002195

% t = 0.700000, y = 2.010763, Error = 0.002989

% t = 0.800000, y = 2.221550, Error = 0.003991

% t = 0.900000, y = 2.454355, Error = 0.005248

% t = 1.000000, y = 2.711460, Error = 0.006822

% Parameters

a = 0; % Start of the interval

b = 1; % End of the interval

alpha = 1; % Initial condition y(0) = 1

N = 10; % Number of steps

% Solve the differential equation using the second-order Taylor method

[T, Y, errors] = taylor2\_method(@diff\_eq, @dfdt, @dfdy, a, b, alpha, N);

% Display the results and errors

disp('Time Points, Approximate Values, and Errors:');

for i = 1:length(T)

fprintf('t = %f, y = %f, Error = %f\n', T(i), Y(i), errors(i));

end

% Define the function for the differential equation y' = y^2 \* exp(-t)

function dy = diff\_eq(t, y)

dy = y^2 \* exp(-t);

end

% Define the partial derivatives of the function

function dfdt\_val = dfdt(t, y)

dfdt\_val = -y^2 \* exp(-t);

end

function dfdy\_val = dfdy(t, y)

dfdy\_val = 2 \* y \* exp(-t);

end

% Second-order Taylor method implementation

function [T, Y, errors] = taylor2\_method(f, dfdt, dfdy, a, b, alpha, N)

h = (b - a) / N; % Calculate the step size

T = a:h:b; % Create a vector of time points

Y = zeros(1, length(T)); % Initialize the solution vector

Y(1) = alpha; % Set the initial condition

errors = zeros(1, length(T)); % Initialize the error vector

% Iterate over each time step

for i = 1:length(T) - 1

t = T(i);

w = Y(i);

Y(i + 1) = w + h \* f(t, w) + (h^2 / 2) \* (dfdt(t, w) + dfdy(t, w) \* f(t, w)); % Taylor's formula

exact\_val = exp(T(i + 1)); % Exact value

errors(i + 1) = abs(Y(i + 1) - exact\_val); % Calculate error

end

end

## Midpoint Method

% Time Points, Approximate Values, and Errors:

% t = 0.000000, y = 1.000000, Error = 0.000000

% t = 0.100000, y = 1.104873, Error = 0.000298

% t = 0.200000, y = 1.220710, Error = 0.000693

% t = 0.300000, y = 1.348650, Error = 0.001208

% t = 0.400000, y = 1.489949, Error = 0.001876

% t = 0.500000, y = 1.645989, Error = 0.002732

% t = 0.600000, y = 1.818296, Error = 0.003823

% t = 0.700000, y = 2.008548, Error = 0.005205

% t = 0.800000, y = 2.218594, Error = 0.006947

% t = 0.900000, y = 2.450470, Error = 0.009134

% t = 1.000000, y = 2.706413, Error = 0.011869

% Parameters

a = 0; % Start of the interval

b = 1; % End of the interval

alpha = 1; % Initial condition y(0) = 1

N = 10; % Number of steps

% Solve the differential equation using the Midpoint Method

[T, Y, errors] = midpoint\_method(@diff\_eq, a, b, alpha, N);

% Display the results and errors

disp('Time Points, Approximate Values, and Errors:');

for i = 1:length(T)

fprintf('t = %f, y = %f, Error = %f\n', T(i), Y(i), errors(i));

end

% Define the function for the differential equation y' = y^2 \* exp(-t)

function dy = diff\_eq(t, y)

dy = y^2 \* exp(-t);

end

% Midpoint Method implementation

function [T, Y, errors] = midpoint\_method(f, a, b, alpha, N)

h = (b - a) / N; % Calculate the step size

T = a:h:b; % Create a vector of time points

Y = zeros(1, length(T)); % Initialize the solution vector

Y(1) = alpha; % Set the initial condition

errors = zeros(1, length(T)); % Initialize the error vector

% Iterate over each time step

for i = 1:length(T) - 1

t = T(i);

w = Y(i);

% Midpoint formula

Y(i + 1) = w + h \* f(t + h/2, w + (h/2) \* f(t, w));

exact\_val = exp(T(i + 1)); % Exact value

errors(i + 1) = abs(Y(i + 1) - exact\_val); % Calculate error

end

end

# Problem 4

A graph of a graph of a graph

Description automatically generated with medium confidence

x\_i = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

y\_i = [34.6588, 40.3719, 14.6448, -14.2721, -13.3570, 24.8234, 75.2795, 103.5743, 97.4847, 78.2392];

figure;

plot(x\_i, y\_i, 'ko', 'MarkerFaceColor', 'k');

hold on;

colors = ['b', 'r', 'g', 'm'];

for degree = 3:6

p = polyfit(x\_i, y\_i, degree);

x\_fit = linspace(1, 10, 400);

y\_fit = polyval(p, x\_fit);

plot(x\_fit, y\_fit, colors(degree-2), 'DisplayName', ['Degree ' num2str(degree)]);

end

legend('show');

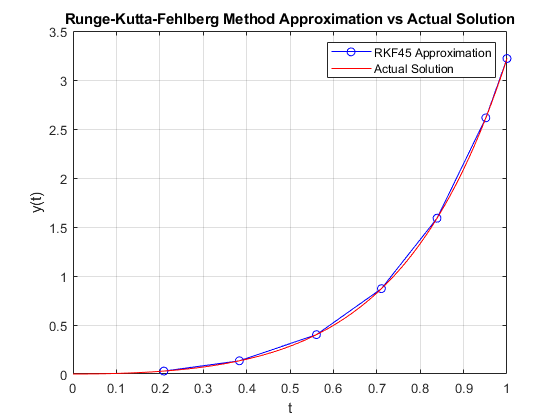
title('Comparison of Polynomial Fits');

xlabel('x');

ylabel('y');

hold off;

# Problem 5



% t = 0.2094, Approximation = 0.029818, Actual = 0.029834, Error = 0.000015

% t = 0.3833, Approximation = 0.134326, Actual = 0.134349, Error = 0.000023

% t = 0.5610, Approximation = 0.401644, Actual = 0.401686, Error = 0.000042

% t = 0.7107, Approximation = 0.870837, Actual = 0.870888, Error = 0.000051

% t = 0.8388, Approximation = 1.589406, Actual = 1.589460, Error = 0.000054

% t = 0.9513, Approximation = 2.614023, Actual = 2.614077, Error = 0.000054

% t = 1.0000, Approximation = 3.219050, Actual = 3.219099, Error = 0.000050

% Given parameters

tol = 1e-4;

hmax = 0.25;

hmin = 0.05;

a = 0;

b = 1;

alpha = 0;

% Define the function

f = @(t, y) t .\* exp(3 \* t) - 2 \* y;

% Runge-Kutta-Fehlberg method

[t, y] = rkf45(f, a, b, alpha, tol, hmax, hmin);

% Calculate and print errors at each step

for i = 1:length(t)

y\_actual = actual\_solution(t(i));

error = abs(y\_actual - y(i));

fprintf('t = %.4f, Approximation = %.6f, Actual = %.6f, Error = %.6f\n', t(i), y(i), y\_actual, error);

end

% Plotting the results

t\_actual = linspace(0, 1, 300);

y\_actual\_vals = arrayfun(@(t) actual\_solution(t), t\_actual);

plot(t, y, 'bo-', t\_actual, y\_actual\_vals, 'r');

xlabel('t');

ylabel('y(t)');

title('Runge-Kutta-Fehlberg Method Approximation vs Actual Solution');

legend('RKF45 Approximation', 'Actual Solution');

grid on;

% Actual solution function

function y = actual\_solution(t)

y = (1/5) \* t .\* exp(3 \* t) - (1/25) .\* exp(3 \* t) + (1/25) .\* exp(-2 \* t);

end

function [t, y] = rkf45(f, a, b, alpha, tol, hmax, hmin)

t = a;

w = alpha;

h = hmax;

FLAG = 1;

output = [];

while FLAG == 1

k1 = h \* f(t, w);

k2 = h \* f(t + h/4, w + k1/4);

k3 = h \* f(t + 3\*h/8, w + 3\*k1/32 + 9\*k2/32);

k4 = h \* f(t + 12\*h/13, w + 1932\*k1/2197 - 7200\*k2/2197 + 7296\*k3/2197);

k5 = h \* f(t + h, w + 439\*k1/216 - 8\*k2 + 3680\*k3/513 - 845\*k4/4104);

k6 = h \* f(t + h/2, w - 8\*k1/27 + 2\*k2 - 3544\*k3/2565 + 1859\*k4/4104 - 11\*k5/40);

R = 1/h \* abs(k1/360 - 128\*k3/4275 - 2197\*k4/75240 + k5/50 + 2\*k6/55);

if R <= tol

t = t + h;

w = w + 25\*k1/216 + 1408\*k3/2565 + 2197\*k4/4104 - k5/5;

output = [output; t, w];

end

delta = 0.84 \* (tol/R)^(1/4);

if delta <= 0.1

h = 0.1 \* h;

elseif delta >= 4

h = 4 \* h;

else

h = delta \* h;

end

if h > hmax

h = hmax;

end

if t >= b

FLAG = 0;

elseif t + h > b

h = b - t;

elseif h < hmin

FLAG = 0;

disp('Minimum step size exceeded');

end

end

t = output(:, 1);

y = output(:, 2);

end