Lab 2 – Numerical integration and its applications

Thibaud Germain (GTZJ79) & Baglan Aitu (G956V1)

I. Exercise 1

a. Implementation of rules in Matlab

Here are the implementations of the three rules: Midpoint, Trapezoidal and Simpsons:

```
function [sum] = quad_midpoint(f, a, b, N)
    h = (b - a) / N;
    % Calculation of extremity points
    sum = f(a)*h/2 + f((N+1/2)*h)*h/2;

    % Calculation of intermediate points
    for i = 2:N
        x_i = (i-(1/2))*h;
        sum = sum + (f(x_i)*h);
    end
end

function [sum] = quad_trapezoidal(f, a, b, N)
    h = (b - a) / N;
    % Calculation of extremity points
    sum = f(a)*h/2 + f(N*h)*h/2;

    % Calculation of intermediate points
    for i = 2:N
        x_i = (i-1)*h;
        sum = sum + (f(x_i)*h);
    end
end


function [sum] = quad_simpsons(f, a, b, N)
    h = (b - a) / N;
    % Calculation of extremity points
    sum = f(a)*h/6 + f(N*h)*h/6;

    % Calculation of intermediate points
    for i = 2:2*N+1
        x_i = (i-1)*h/2;
        if mod(i,2) == 0
            sum = sum + (f(x_i)*(4*h/6));
        else
            sum = sum + (f(x_i)*(2*h/6));
        end
    end
end
```

When executing with the function sin(x) between 0 and $\frac{\pi}{2}$, we have the following results:

Calculation with Midpoint rule and 4 iterations:

1.1224

Calculation with Trapezoidal rule and 4 iterations:

0.9871

Calculation with Simpsons rule and 4 iterations:

1.0000

We can observe that the results for each method are close to 1 while Simpson's method is most accurate.

With 100 iterations, the trapezoidal method shows the accurate result as well:

Calcul with midpoint rule and 100 iterations

1.0077

Calcul with trapezoidal rule and 100 iterations

1.0000

Calcul with simpsons rule and 100 iterations

1.0000

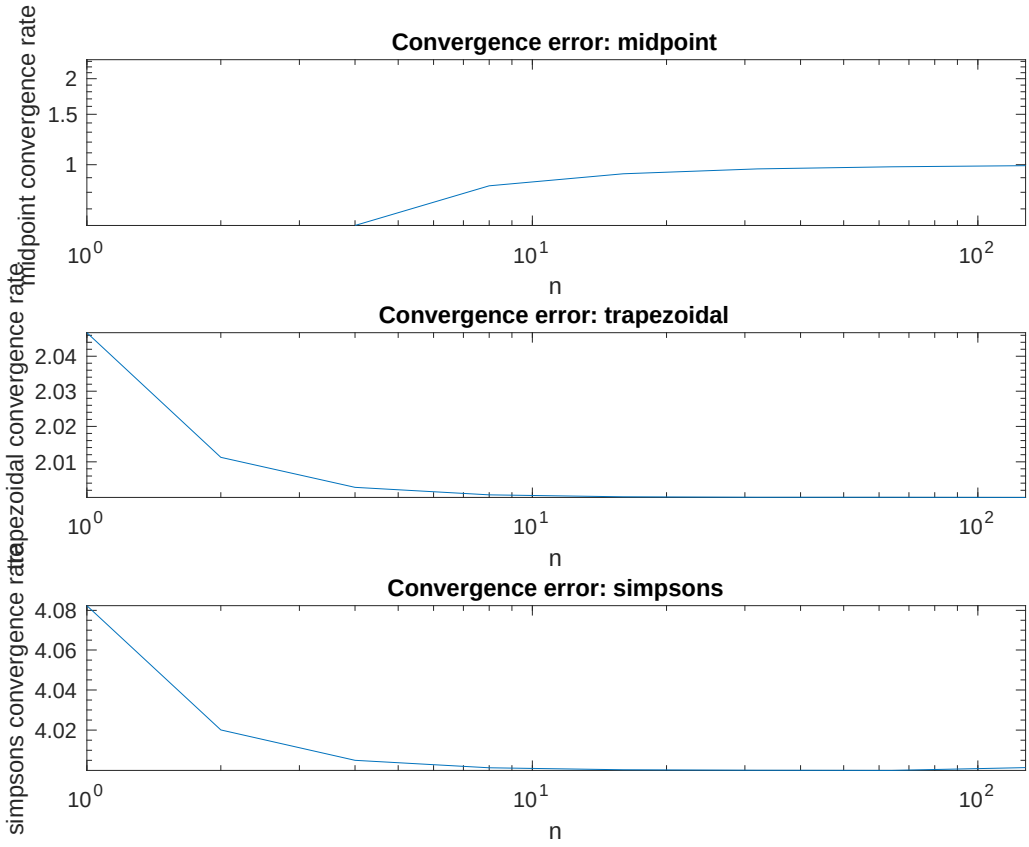
By this we proved that Simpson's method is best approximation amid the rest rules.

b. Convergence rate

```
function [convergence_rate] = calculate_convergence_rate_q1(f, a, b, N, method)
    real_integral_h = integral(f,a,b);

    calculated_integral_h = method(f,a,b,N);
    error_h = abs(real_integral_h - calculated_integral_h);

    calculated_integral_h_2 = method(f,a,b,2*N);
    error_h_2 = abs(real_integral_h - calculated_integral_h_2);

    convergence_rate = log2(error_h/error_h_2);
end

function plot_convergence_error(f, a, b, N, method)
    n = 1;
    convergence_rates = zeros(1,N);
    n_values = zeros(1,N);
    for k = 1:N
        convergence_rates(k) = calculate_convergence_rate_q1(f, a, b, n, method);
        n_values(k) = n;
        n = n*2;
    end
    loglog(n_values,convergence_rates)
    grid on
end
```

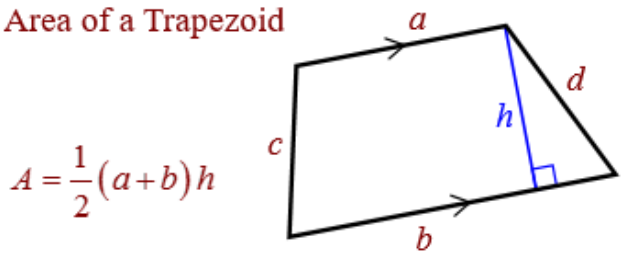
The convergence error functions above allow us to provide the following graphs, we can observe that the midpoint method is converging to one. This means that at a certain point, doubling the number of steps does not give more precision about the integral calculation.

However, the trapezoidal is converging to two, that means that each time we double the number of step we double the precision. In Simpson's method we increase the precision by 4 times.



c. Convergence rate

If we change the weights at the endpoints for the trapezoidal rule from h/2 to h, then the convergence rate becomes 1. The reason for this changing the endpoints to h is similar with elimaniting "1/2" from equation of trapezoid area (picture down below) which doubles the result.



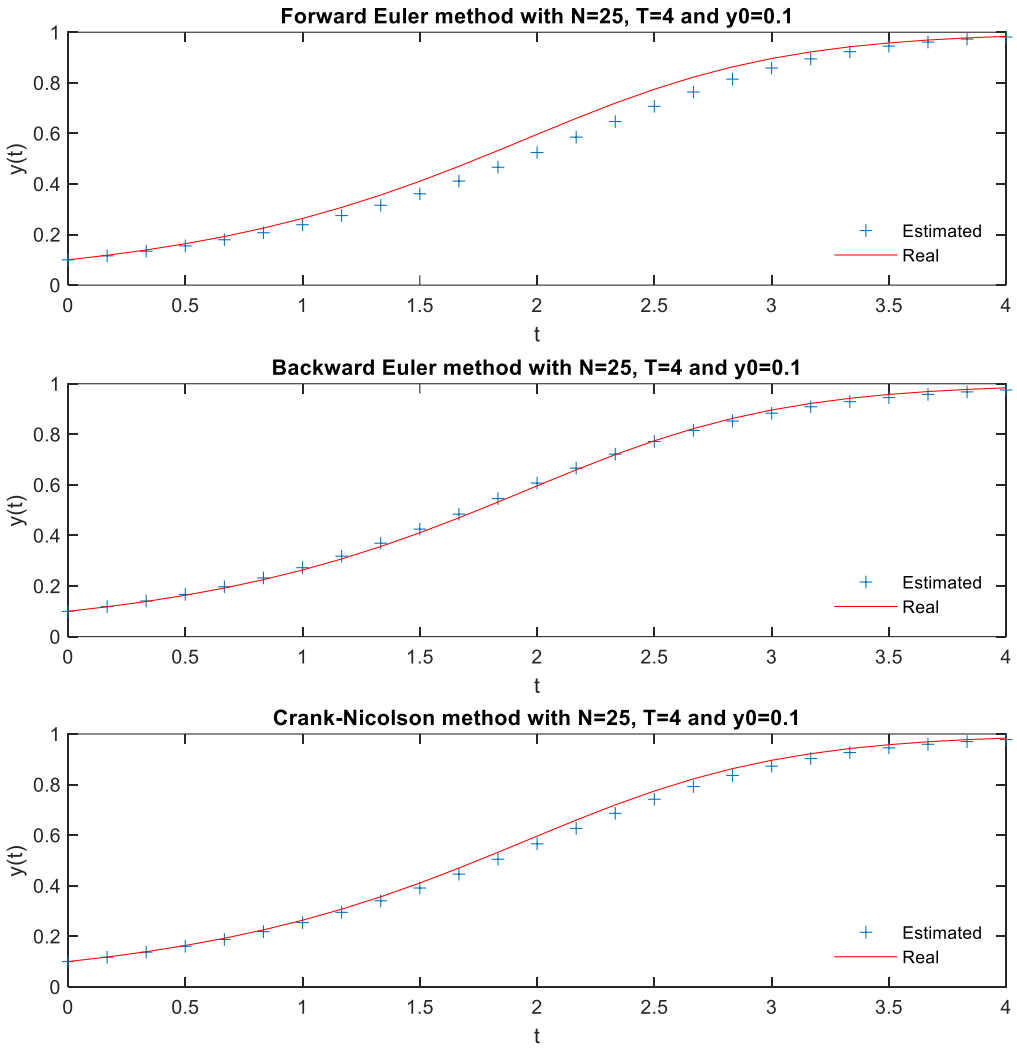
https://www.onlinemathlearning.com/area-trapezoid.html

II. Exercise 2

a. Implementation of Euler, Backward Euler and Crank-Nicolson methods

The first method is using only previous steps. The second and third needs to solve an algebraic equation and we used secant method of first lab to do so.

```
% y0 => result of function at 0
% N => number of steps
% T => final time
function [y] = euler_method(f,y0,T,N)
    % Method implemented : y_n_1 = yn + h*f(tn,yn)
    y = zeros(1,N);
    y(1) = y0;
    % delta time
    h = T/N;
    for k = 2:N
        y(k) = y(k-1) + h * f(y(k-1));
    end
end

function [y] = backward_euler_method(f,y0,T,N)
    % Method implemented : y_n_1 = yn + h*f(tn_1,yn_1)
    y = zeros(1,N);
    h = T/N;
    y(1) = y0;
    for i = 1:N-1
        y(i+1) = secant(@(Y) y(i) + h*f(Y) - Y, i*h, i*(h+1), 1e-06);
    end
end

function [y] = crank_nicolson_method(f,y0,T,N)
    % Method implemented : y_n_1 = yn + 0.5*h*f(tn,yn) + 0.5*h*f(tn_1,yn_1)
    y = zeros(1,N);
    h = T/N;
    y(1) = y0;
    for i = 1:N-1
        y(i+1) = secant(@(Y) y(i) + 0.5*h*f(y(i)) + 0.5*h*f(Y) - Y, i*h, i*(h+1), 1e-06);
    end
end
```



With this set of values (N=25, T=4 and y0=0.1), the backward Euler method seems to be the most appropriate to find the solution.

b. Asymptotic order of the methods

```
function [changement,order] = check_changement(previous_rate, new_rate, precision)
    changement = true;
    if (abs(previous_rate - new_rate) <= precision) || (abs(new_rate - round(new_rate)) <= precision)
        changement = false;
    end
    order = round(new_rate);
end

function [convergence_rate] = calculate_convergence_rate_q2(method,y0,T,N,real_y,f)
    calculated_Y = method(f,y0,T,N);
    calculated_y_2N = method(f,y0,T,2*N);
    error_1 = abs(real_y - calculated_Y(end));
    error_2 = abs(real_y - calculated_y_2N(end));
    convergence_rate = log2(error_1/error_2);
end


function [order] = calculate_order(f,y0,T,N,p, real_y, method)
    changement = true;
    convergence_rates = zeros(1,N);
    for i=1:N
        convergence_rates(i) = calculate_convergence_rate_q2(method,y0,T,i,real_y(T,y0),f);
        if(i > 1)
            [changement,order] = check_changement(convergence_rates(i-1),convergence_rates(i),p);
        end
        if(~changement)
            break;
        end
    end
end
```

Order of forward euler method

1

Order of backward euler method

1

Order of Crank-Nicolson method

1