Numerical Methods: Assignment #5

Hong Chenhui
drredthered.github.io

Zhejiang University — April 15, 2024

1 Problem

- **Question** Following equation described a current flow passing through a resistor.

$$i(t) = (60 - t)^2 + (60 - t)\sin(t) \tag{1}$$

With the resistor value of

$$r(t) = 10i(t) + 2i(t)^{\frac{2}{3}} \tag{2}$$

calculate the average voltage dissipated in the resistor with integration from $t = 0$ to $t = 60$.

1.1 Theoretical viewpoint

Question

Several method could be chosen to solve the problem. The most straightforward one is to calculate the average value of the function $i(t) \times r(t)$, which is the average voltage dissipated in the resistor. The formula is as following.

First are Newton-Cotes ones, which are based on polynomial interpolation. One would easily choose trapezoidal rule

Algorithm 1: Trapezoidal Rule

Data: Function $f(x)$, interval $[a, b]$, number of subintervals n
Result: Approximation of the definite integral $\int_a^b f(x)\,dx$

$h = \frac{b-a}{n}$;
$S = \frac{1}{2}f(a) + \frac{1}{2}f(b)$;
for $i = 1$ **to** $n - 1$ **do**
  $x_i = a + i \cdot h$;
  $S = S + f(x_i)$;
end
$S = h \cdot S$;

As well as Simpson's one-third rule.

This part is intentially left blank

Question

Algorithm 2: Simpson's One-Third Rule

Data: Function $f(x)$, interval $[a, b]$, number of subintervals n
Result: Approximation of the definite integral $\int_a^b f(x)\,dx$

$h = \frac{b-a}{n}$;
$S = f(a) + f(b)$;
$x_0 = a$;
$x_n = b$;
for $i = 1$ **to** $n - 1$ **do**
  $x_i = a + i \cdot h$;
  if i *is even* **then**
    $S = S + 2 \cdot f(x_i)$;
  end
  else
    $S = S + 4 \cdot f(x_i)$;
  end
end
$S = \frac{h}{3} \cdot S$;

Romberg integration and Gaussian quadrature are other options because Newton-Cotes methods frequently call for a large number of iterations.

Algorithm 3: Romberg Iteration

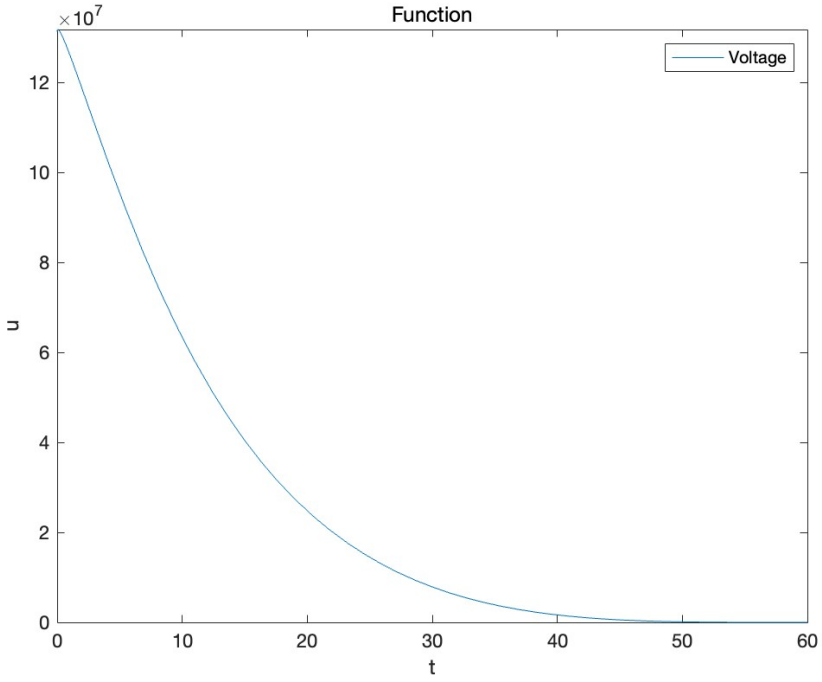
Data: Function $f(x)$, interval $[a, b]$, number of iteration n
Result: Approximation of the definite integral $\int_a^b f(x)\,dx$
$h \leftarrow b - a$;
$R_{1,1} \leftarrow \frac{h}{2}(f(a) + f(b))$;
for $i \leftarrow 2$ **to** n **do**
  $h \leftarrow \frac{h}{2}$;
  $R_{i,1} \leftarrow \frac{1}{2}R_{i-1,1} + h\sum_{k=1}^{2^{i-2}} f(a + (2k-1)h)$;
  for $j \leftarrow 2$ **to** i **do**
    $R_{i,j} \leftarrow R_{i,j-1} + \frac{R_{i,j-1} - R_{i-1,j-1}}{4^{j-1}-1}$;
  end
end
return $R_{n,n}$;

Algorithm 4: Gaussian Quadrature

Input: $f(x)$: the function to be integrated, n: number of points
Output: Approximation of the integral using Gaussian Quadrature
Compute the roots x_i and weights w_i for n-point Gaussian quadrature;
$Q \leftarrow 0$;
for $i \leftarrow 1$ **to** n **do**
  $Q \leftarrow Q + w_i \cdot f(x_i)$;
end
return Q;

In this assignment one would choose the typical Gaussian-Legendre method to determine the value of w_i as well as the node x_i. Due to its own complicity, the calculation are not shown here.

2 implementation



Method	Value $\times 10^7$	Rel.Error%	Segment/Iteration
Trapezoidal Rule	2.636121517695977	2.65×10^{-8}	1000000
Simpson's one-third	2.636121518120261	1.04×10^{-8}	1000000
Romberg Integration	2.636121513507557	1.85×10^{-7}	18
Gauss Quadrature	2.636121518742559	1.31×10^{-8}	1000

3 Analysis

3.1 Comparison

There is no denying that the Newton-Cotes rule evolves more accurate with the increase of the number of segments. As the chart says,

Segments (n)	Points	Name	Formula	Truncation Error
1	2	Trapezoidal rule	$(b-a)\frac{f(x_0)+f(x_1)}{2}$	$-(1/12)h^3f''(\xi)$
2	3	Simpson's 1/3 rule	$(b-a)\frac{f(x_0)+4f(x_1)+f(x_2)}{6}$	$-(1/90)h^5f^{(4)}(\xi)$
3	4	Simpson's 3/8 rule	$(b-a)\frac{f(x_0)+3f(x_1)+3f(x_2)+f(x_3)}{8}$	$-(3/80)h^5f^{(4)}(\xi)$
4	5	Boole's rule	$(b-a)\frac{7f(x_0)+32f(x_1)+12f(x_2)+32f(x_3)+7f(x_4)}{90}$	$-(8/945)h^7f^{(6)}(\xi)$
5	6		$(b-a)\frac{19f(x_0)+75f(x_1)+50f(x_2)+50f(x_3)+75f(x_4)+19f(x_5)}{288}$	$-(275/12,096)h^7f^{(6)}(\xi)$

Yet the Romberg iteration and Gaussian quadrature are more efficient in terms of the number of iterations. The Romberg iteration is the most efficient one, with only 18 iterations. The Gaussian quadrature is the second one, with 1000 iterations. Although the Romberg one seems to have a larger relative error, yet considering the number of iteration, it is an achievement.

3.2 Gauss Point Weight Calculation

Typical Gaussian-Legendre method to determine the value of w_i and the node x_i is not trival as it seems. With the n-th polynomial normalized to give $P_n(1) = 1$, the i-th Gauss node, x_i, is the i-th root of P_n and the weights are given by the formula

$$w_i = \frac{2}{(1 - x_i^2)[P_n'(x_i)]^2} \tag{3}$$

The w_i and x_i can be obtained from the eigenvalue decomposition of the symmetric, tridiagonal Jacobi matrix

$$\mathcal{J} = \begin{bmatrix} a_0 & \sqrt{b_1} & 0 & \cdots & 0 \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & \ddots & \vdots \\ 0 & \sqrt{b_2} & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & a_{n-2} & \sqrt{b_{n-1}} \\ 0 & \cdots & 0 & \sqrt{b_{n-1}} & a_{n-1} \end{bmatrix} \tag{4}$$

J and $\mathcal{J}$ are [[similar matrices]] and therefore have the same eigenvalues (the nodes). The weights can be computed from the corresponding eigenvectors: If $\phi^{(j)}$ is a normalized eigenvector (i.e., an eigenvector with euclidean norm equal to one) associated with the eigenvalue x_j, the corresponding weight can be computed from the first component of this eigenvector, namely:

$$w_j = \mu_0 \left(\phi_1^{(j)}\right)^2 \tag{5}$$

where μ_0 is the integral of the weight function

$$\mu_0 = \int_a^b \omega(x)dx \tag{6}$$

for Legendre polynomials, detailed information could be found in Gaussian Quadrature and the Eigenvalue Problem. In general, the practice looks like this.

Algorithm 5: Computing Gauss quadrature points and weights

Data: number of subintervals n
Result: Gauss quadrature points x_i and weights w_i

Function `gauss_points_weights(`n`)`**:**
- $k \leftarrow 1$ **to** $n - 1$;
- $a \leftarrow \frac{1}{\sqrt{4 - \frac{1}{(k^2)}}}$;
- $A \leftarrow \text{diag}(a, 1) + \text{diag}(a, -1)$;
- $[V, D] \leftarrow \text{eig}(A)$;
- $x_i \leftarrow \text{diag}(D)$;
- $w_i \leftarrow 2 \times V(1, :).^2$;
- **return** x_i, w_i;

Which would obtain extremely satisfying results.

4 Code

//function
multiple_application_trapezoidal_rule.m

```
function result = multiple_application_trapezoidal_rule(f, a, b, n)

    h = (b - a) / n;

    result = 0;
```

```
    for i = 1:n
        % Calculate the endpoints of the current segment
        x0 = a + (i - 1) * h;
        x1 = a + i * h;

        % Apply trapezoidal rule to the current segment
        result = result + (f(x0) + f(x1)) / 2 * (x1 - x0);
    end

end
```

multiple_application_simpsons_one_third_rule.m

```
function result = multiple_application_simpsons_one_third_rule(f, a, b, n)

    h = (b - a) / n;


    result = 0;

    for i = 1:2:n-1
        % Calculate the endpoints of the current pair of segments
        x0 = a + (i - 1) * h;
        x1 = a + i * h;
        x2 = a + (i + 1) * h;

        % Apply Simpson's 1/3 Rule to the current pair of segments
        result = result + (h / 3) * (f(x0) + 4*f(x1) + f(x2));
    end

    % Adjust the result for odd number of segments
    if mod(n, 2) == 1
        % If the number of segments is even, apply Simpson's 1/3 Rule to
        % the last segment separately
        x0 = a + (n - 1) * h;
        x1 = a + (n - 0.5) * h;
        x2 = a + n * h;
        result = result + ((h/2) / 3) * (f(x0) + 4*f(x1) + f(x2));
    end
end
```

romberg_integration.m

```
function I = romberg_integration(f, a, b, n)

    % Initialize the Romberg table
    R = zeros(n, n);

    % Compute the first column of the table using trapezoidal rule
    h = (b - a);
    R(1, 1) = h/2 * (f(a) + f(b));

    for i = 2:n
        h = h / 2;
        sum_f = 0;
        for k = 1:(2^(i-2))
            sum_f = sum_f + f(a + (2*k - 1)*h);
        end
        R(i, 1) = 0.5 * R(i-1, 1) + h * sum_f;
```

```
        % Extrapolate to higher orders of accuracy
        for j = 2:i
            R(i, j) = R(i, j-1) + (R(i, j-1) -
                        R(i-1, j-1)) / ((4^(j-1)) - 1);
        end
    end

    % The final result is in the last row and last column of the table
    I = R(n, n);
end
```

gauss_quadrature.m

```
function I = gauss_quadrature(f, a, b, n)
    % Define the Gauss quadrature points and weights
    [nodes, weights] = gauss_points_weights(n);

    % Map nodes from [-1, 1] to [a, b]
    nodes = ((b - a) * nodes + (b + a)) / 2;

    % Perform the quadrature sum
    I = weights * f(nodes) * (b - a) / 2;
end

function [nodes, weights] = gauss_points_weights(n)
    % Compute the Gauss quadrature points and weights for n points

    % Precompute constants for recurrence relation
    k = 1:n-1;
    a = 1 ./ sqrt(4-1./(k.^2) );

    % Compute eigenvalues of the tridiagonal matrix
    A = diag(a, 1) + diag(a, -1);
    [V, D] = eig(A);

    % Gauss nodes are the eigenvalues
    nodes = diag(D);

    % Gauss weights are proportional to the squares of the elements of the
    % first row of the eigenvector matrix
    weights = 2 * V(1, :).^2;
end
```

//script

Current.m

```
%Current Integral
clear;
clc;

%initalization on function
i = @(t) (60-t).^2+(60-t).*sin(sqrt(t));

r = @(t) 10.*i(t)+2.*i(t).^(2/3);

u = @(t) i(t).*r(t);

fplot(u,[0 60]);
```

```
legend('Voltage');
xlabel('t');
ylabel('u');
title('Function');

u_ave(1) = integral(u,0,60) / 60;

u_ave(2) = multiple_application_trapezoidal_rule(u,0,60,1000000) / 60;

u_ave(3) = multiple_application_simpsons_one_third_rule(u,0,60,1000000)
/ 60;

u_ave(4) = romberg_integration(u,0,60,18) / 60;

u_ave(5) = gauss_quadrature(u,0,60,1000) / 60;

disp(u_ave)
```