FCMII - Programming Report 4

Note on Implementation

We implement our routines in septate MATLAB function.m files for easy organization. The test driver and also manual testing examples are done in live script file (.mlx). The vectorization provided by MATLAB is used as much as possible to avoid loops, since these operations are highly optimized in MATLAB. I used generative AI to streamline some plotting code and data visualization, as well as some organization, specifically CHATGPT 4-o. This was to avoid repetitively coding the plots, and design the testing more efficiently.

1 Introduction

In this project we want to test and implement various quadrature methods and analyze theoretical numerical predictions about convergence properties. We only describe the single interval methods, and use them to implement the more general composite forms for each method. In addition, we implement two adaptive methods using global mesh refinement. The methods are as follows:

- The closed Newton–Cotes method that uses one point (the left endpoint), i.e., the Left Rectangle Rule.
- The closed Newton–Cotes method that uses two points (the two endpoints), i.e., the Trapezoidal Rule.
- The closed Newton–Cotes method that uses three points (the two endpoints and the midpoint), i.e., Simpson's First Rule.
- The open Newton–Cotes method that uses one point (the midpoint), i.e., the Midpoint Rule.
- The open Newton–Cotes method that uses two points (the points at 1/3 and 2/3 across the interval of integration).
- The two-point Gauss-Legendre method.
- The adaptive Trapezoidal Rule with global mesh refinement factor of $\alpha = \frac{1}{2}$. This is implemented with **complete re-use** to minimize function evaluations.
- The adaptive Mid-point Rule with global mesh refinement factor of $\alpha = \frac{1}{3}$. This is implemented with **complete re-use**.

2 Quadrature Routines and Implementation

We describe each of the composite quadrature routines and describe their implementations. We also discuss efficiency and code optimization for our implementations, in particular the complete re-use for the adaptive methods and the optimized global mesh refinement. We exclusively work on a global uniform mesh for this purpose. We also discuss the global error expressions for each method, and the error estimation for adaptive methods. The required derivations are done within each section, and the previously derived results are quoted from the lecture notes.

Note that for the Newton-Cotes methods, as well as the Gauss-Legendre, there are no storage costs (except of course temporary ones), and the major cost and complexity is determined by the number of function evaluations.

2.1 Left Rectangle Rule

The Left Rectangle Rule is the closed 1-point Newton-Cotes rule using a constant interpolant. Locally, the form and the error is:

$$I_0^{(i)} : h_1(f_0), \quad \text{Left Rectangle rule}$$

$$E_1^{(i)} = -\frac{(b-a)^2}{2} f'(\xi) + \mathcal{O}((b-a)^3)$$

The composite form is also simple since there is no overlap:

$$I(f) = H_n \sum_{i=1}^{n} f_{i-1}$$

The composite error can be derived by summing over all the intervals, substituting $h_1 = H$ in the local error, and using the discrete Mean Value Theorem:

$$\sum_{i=1}^{n} -\frac{H^2}{2} f^{(1)}(x_i) + \sum_{i=1}^{n} \mathcal{O}(H^3)$$

$$= -\frac{H^2}{2} \left(n f^{(1)}(\xi) \right) + \mathcal{O}(H^2)$$

$$= -\frac{(b-a)}{2} H f^{(1)}(\xi) + \mathcal{O}(H^2)$$

The method is easily implemented as follows.

```
function [int,error] = leftrectangle(func,mesh,H,n,true_int)
    fvals = func(mesh(1:n));

    int = sum(H*fvals);

    error = true_int-int;
end
```

Cost: m evaluations

2.2 Trapezoidal Rule

The Trapezoidal Rule is the closed 2-point Newton-Cotes rule using a linear interpolant. Locally, the form and the error is:

$$I_1^{(i)} : \frac{h_1}{2} \left[f_0 + f_1 \right], \quad \text{Trapezoidal rule}$$

$$E_1^{(i)} = -\frac{h_1^3}{12} f^{(2)}(\eta) = -\frac{h_1^3}{12} f^{(2)}(x_i) + \mathcal{O}(h_1^4),$$

The composite form (optimized to minimize function evaluations) is:

$$I(f) = \frac{H_n}{2} \left[f(x_0) + f(x_n) + 2 \sum_{i=1}^{n-1} f(x_i) \right]$$

The composite error over the general interval $[a, b]$ is given by:

$$E_1 = -\frac{1}{12} (b-a) H_n^2 f''(\mu) + \mathcal{O}(H_n^3)$$

where for n intervals, we have

$$H_n = \frac{b-a}{n}$$

Note for the trapezoidal, no function evaluations besides on the global mesh are required, and then we simple evaluate the composite formula as it minimizes repeated evaluations:

```
function [int,error] = trapezoidal(func,mesh,H,n,true_int)
    fvals = func(mesh);

    int = 2.0*sum(fvals(2:n));
    int = int+fvals(1)+fvals(n+1);
    int = int*(H/2.0);

    error = true_int-int;
    end
```

Cost: $m + 1$ evaluations

2.3 Simpson's Rule

Simpson's Rule is the closed 3-point Newton-Cotes rule using a quadratic interpolant. Locally, the form and the error is:

$$I_2 : \frac{h_2}{3}\left[f_0 + 4f_1 + f_2\right], \quad \text{Simpson's 1st rule}$$

$$E_2 = -\frac{h_2^5}{90} f^{(4)}(\eta) = -\frac{h_2^5}{90} f^{(4)}(x_i) + \mathcal{O}(h_2^6),$$

The composite form (optimized to minimize function evaluations) is

$$I(f) = \frac{H_n}{6}\left[f(a) + f(b) + 2\sum_{i=1}^{n-1} f(b_i) + 4\sum_{i=1}^{n} f(c_i)\right].$$

where c_i is the midpoint of each local interval. The total composite error is given by:

$$E_2 = -(b-a)\cdot\frac{1}{2880} H_n^4 f^{(4)}(\zeta) + \mathcal{O}(H_n^5)$$

Given a global mesh, this method is also simply implemented:

```
function [int,error] = simpson(func,mesh,H,n,true_int)
    fvals = func(mesh);
    h = H/2.0;
    c = mesh(1:n) + h;          % Broadcasting for c_i values
    fc_vals = func(c);

    int = 2.0*sum(fvals(2:n));
    int = int+(4.0*sum(fc_vals));
    int = int+fvals(1)+fvals(n+1);
    int = int*(H/6.0);

    error = true_int-int;
end
```

Note the local size is $\frac{H}{2}$ and this is used to find all the extra evaluation points in the middle using a shift on the mesh less the last point.

Cost: $2m + 1$ evaluations

2.4 Mid-point Rule

The Mid-point Rule is the open 1-point Newton-Cotes rule using a constant interpolant. Locally, the form and the error is:

$$I_0 : 2h_0\left[f_0\right], \quad \text{Midpoint rule}$$

$$E_0 = \frac{h_0^3}{3} f^{(2)}(\eta) = \frac{h_0^3}{3} f^{(2)}(x_i) + \mathcal{O}(h_0^4)$$

The composite form of the midpoint rule is:

$$I(f) = H_n \sum_{i=1}^{n} f_i$$

The composite error is:

$$= (b-a)\cdot\frac{1}{24} H_n^2 f^{(2)}(\eta) + \mathcal{O}(H_n^3)$$

Note that the evaluation points are in the middle of each local interval, so we use the same shifting and broadcasting as before:

```
function [int,error] = midpoint(func,mesh,H,n,true_int)
    h = H/2.0;
    c = mesh(1:n) + h;          % Broadcasting for c_i values
    fvals = func(c);

    int = H*sum(fvals);

    error = true_int-int;
    end
```

Cost: m evaluations

2.5 Two-point Open Newton Cotes

The Two-point Newton Cotes Rule uses a linear interpolant so locally it uses the points $\{a_i + \frac{H}{3}, b_i - \frac{H}{3}\}$ as the points of interpolation. Locally, the form and the error is:

$$I_1 : \frac{3h_1}{2}\left[f_0 + f_1\right]$$

$$E_1 = \frac{3h_1^3}{4}f^{(2)}(\eta) = \frac{3h_1^3}{4}f^{(2)}(x_i) + \mathcal{O}(h_1^4)$$

For the composite, we get the form

$$I(f) = \frac{3H}{2}\sum_{i=1}^{n}(f(x_-) + f(x_+))$$

since there is no overlap between evaluation points, where the points are given as described above. The composite error can be derived by substituting $h_1 = H/3$ and summing over all intervals:

$$\sum_{i=1}^{n}\frac{3(H/3)^3}{4}f^{(2)}(x_i) + \mathcal{O}((H/3)^4)$$

$$= \sum_{i=1}^{n}\frac{3H^3}{108}f^{(2)}(x_i) + \sum_{i=1}^{n}\mathcal{O}(H^4)$$

$$= \frac{3H^3}{108}\left(nf^{(2)}(\xi)\right) + \mathcal{O}(H^4)$$

$$= (b-a)\frac{3}{108}H^2f^{(2)}(\xi) + \mathcal{O}(H^3)$$

Cost: $2m$ evaluations

2.6 Two-point Gauss Legendre

For the Gauss Legendre Quadrature rule in general, we get

- x_i are the roots of the $n+1$–st Legendre polynomial P_{n+1}.
- The weights are
$$\gamma_i = \int_{-1}^{1}\ell_i(x)$$
- The approximation
$$\int_{-1}^{1}f(x)\,dx \approx \int_{-1}^{1}p_n(x)\,dx = \sum_{i=0}^{n}\gamma_i f(x_i)$$
has degree of exactness $2n+1$.
- The formulas are open since the roots of the orthogonal polynomials are interior to the interval of definition.

We can integrate over a general interval $[a, b]$ by a change of variables to $[-1, 1]$

$$a \leq x \leq b \quad \text{and} \quad -1 \leq z \leq 1$$

$$x = \frac{z(b-a)+b+a}{2} \quad \Rightarrow \quad dx = \frac{b-a}{2}\, dz$$

$$\int_a^b f(x)\, dx = \frac{b-a}{2} \int_{-1}^{1} f\left(\frac{z(b-a)+b+a}{2}\right) dz$$

$$I_n(f) = \frac{b-a}{2} \sum_{i=0}^{n} \gamma_i f(u_i)$$

$$u_i = \frac{z_i(b-a)+b+a}{2} \quad \text{and} \quad P_{n+1}(z_i) = 0$$

where we transport the roots into each sub-interval of the global mesh and use a local linear interpolant to integrate. Note that the Two-point Gauss-Legendre uses a linear interpolant with

$$x_\pm = \pm\frac{1}{\sqrt{3}}, \quad \gamma_\pm = 1$$

so each of these two roots is transported. Locally, note that the expression becomes

$$\int_{a_i}^{b_i} f(x)\, dx = \frac{b_i - a_i}{2} \left(f(x_-) + f(x_+)\right) = \frac{H}{2}(f(x_-) + f(x_+))$$

since we have a uniform mesh. For the composite rule, we simply do this for each interval and get the full sum over the total number of intervals (our value of n) as the estimate i.e.

$$I(f) \approx \frac{H}{2} \sum_{i=1}^{n} (f(x_-^{(i)}) + f(x_+^{(i)}))$$

so for our implementation, we only need to evaluate the points where the function needs to be evaluated in terms of the global mesh, then sum the function evaluations at these values, and then multiply by the appropriate scale (which is a single scaler due to the uniformity). We implement this as follows:

```
function [int,error] = gausslege(func,mesh,H,n,true_int)
    H_left = (H/2.0)*(1-(1/sqrt(3)));
    H_right = (H/2.0)*(1+(1/sqrt(3)));

    xprime = mesh(1:n) + H_left;            % Broadcasting for x_i' values
    xprimeprime = mesh(1:n) + H_right;      % Broadcasting for x_i'' values

    fvals_left = func(xprime);
    fvals_right = func(xprimeprime);

    int = (H/2) * sum(fvals_left + fvals_right);

    error = true_int-int;
end
```

where to optimize the code, we find a correct scale to find the values to evaluate f on. Note that, to linearly transform from the interval $[a, b]$ into $[c, d]$ we get the easily derived formula for a line: we map the standard interval $[-1, 1]$ to $[x_i, x_{i+1}]$ using an affine transformation. Let $z \in [-1, 1]$ be the reference coordinate and $x \in [x_i, x_{i+1}]$ be the physical coordinate. The affine mapping is given by:

$$x(z) = \frac{x_i + x_{i+1}}{2} + \frac{x_{i+1} - x_i}{2} z$$

This transformation satisfies:

$$z = -1 \Rightarrow x = x_i, \quad z = +1 \Rightarrow x = x_{i+1}$$

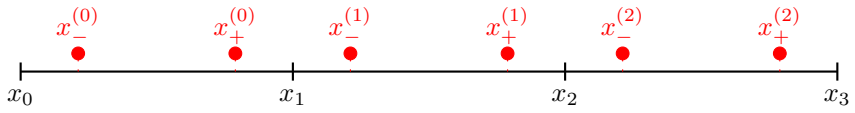
If the mesh is uniform with interval length $H = x_{i+1} - x_i$, then the mapping simplifies to:

$$x(z) = x_i + \frac{H}{2}(1 + z)$$

For the 2-point Gauss–Legendre rule with $z = \pm\frac{1}{\sqrt{3}}$, the mapped evaluation points become:

$$x_i^{\pm} = x_i + \frac{H}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right)$$

We can then simply scale our mesh by these values using MATLAB broadcasting as is done in the code to get the evaluation points. The remaining code is simple and obvious.



Global interval $[a, b]$ split into subintervals

The local error for the method is given by the following theorem:

Theorem 2.1. *If a Gauss-Legendre quadrature formula is used to approximate a definite integral on* $[-1, 1]$*, we have*

$$E_n(x) = \frac{2^{2n+3}\left((n+1)!\right)^4}{(2n+3)\left((2n+2)!\right)^3} f^{(2n+2)}(\xi)$$

with $-1 < \xi < 1$.

Since this is the form of the locaal error with $n = 1$, we get that the local error is given by

$$E_n(x) = \frac{2^5\left((2)!\right)^4}{(5)\left((4)!\right)^3} f^{(4)}(\xi) = \frac{2^5 2^4}{2^9 3^3 5} f^{(4)}(\xi) = \frac{1}{135} f^{(4)}(\xi)$$

since each local sum in our approximation estimates an integral on the interval of reference (therefore the theorem applies). Then for the total error for the composite method we get that, for $-1 < \xi, \zeta < 1$:

$$E_1 = I - I_1 = \frac{H}{2}\sum_{i=1}^{n} E_1^{(i)} = \frac{H}{2}\sum_{i=1}^{n} \frac{1}{135} f^{(4)}(\xi) = \frac{H}{2} f^{(4)}(\zeta)\left(n\frac{1}{135}\right) = \frac{b-a}{270} f^{(4)}(\zeta)$$

where we have used the discrete Mean Value Theorem.

Cost: $2m$ evaluations

2.7 Adaptive Trapezoidal

For the Trapezoidal, we derived the following recurrence in the lecture notes:

$$I_{2n} = \frac{1}{2}\left[I_n + h_n\,(\text{sum of new } f_i)\right]$$

where $\alpha = \frac{1}{2}$ and the locations of the new values are in the middle of each local interval. These can then be found by the same method of shifting as we have done before. Note that we have to store and keep track of the global mesh as the refinements continue. Thus after each iteration, we have to update the mesh and the number of points, as well as the values of the local interval size (the points are doubled and the local size is halved). Since our shifting implementation for new values directly gives us the array of new values, the implementation of the recurrence is easy, and we preserve complete re-use.

The update step for the global mesh then becomes slightly complicated. It is not hard to notice that we simply have to interleaf values from current mesh array and the new values one by one, and this clearly gives us the new global mesh, updated for the next iteration. This is easily implemented in MATLAB using the appropriate access of the array and is indicated in the code:

```
function [integrals, errors, estimates] = trapadaptive(func,mesh,H,n,true_int, ref_num)
    % Initializing results
    integrals = zeros(1,ref_num+1);
    errors = zeros(1,ref_num+1);

    % Starting recurrence
    [int_start, error_start] = trapezoidal(func,mesh,H,n,true_int);
```

```
    integrals(1) = int_start;
    errors(1) = error_start;

    int = int_start;
    h_shift = H;
    mesh_current = mesh;

    for i = 2:ref_num+1
        h_old = h_shift;

        % New mesh values (same as evaluation values since H = h)
        h_shift = h_shift/2.0;
        mesh_new = mesh_current(1:n) + h_shift;

        % Complete re-use - only evaluating at new values
        fvals_new = func(mesh_new);

        % Recurrence
        int  = (1/2.0)*(int + h_old*(sum(fvals_new)));

        % Update Global Mesh
        mesh_temp = zeros(1,(2*n+1));
        mesh_temp(1:2:end) = mesh_current;
        mesh_temp(2:2:end) = mesh_new;
        mesh_current = mesh_temp;
        n = 2*n;

        % Store results
        integrals(i) = int;
        errors(i) = true_int - int;
    end

    % Error estimate recurrence
    estimates = (4/3)*(integrals(2:end) - integrals(1:end-1));
end
```

In general, for a mesh refinement for a method of some order, we get that

$$\hat{E}_f = \frac{1}{\alpha^r}\hat{E}_c \to \frac{\alpha^r}{(\alpha^r - 1)}(I_f - I_c) = \hat{E}_c + \mathcal{O}(h^{r+1}) = E_c + \mathcal{O}(h^{r+1})$$

where r is the order of the method. This allows us to calculate the error estimates for both adaptive methods.

2.8 Adaptive Midpoint

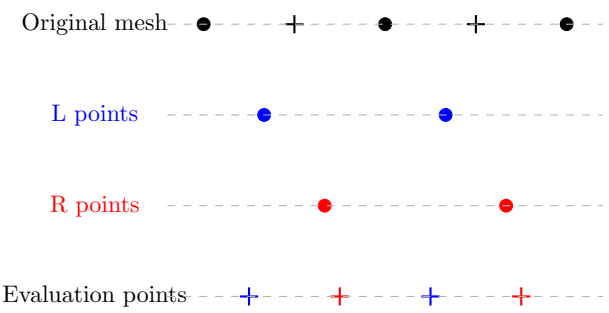
We can similarly derive the recursion and error expression for the adaptive Midpoint using $\alpha = \frac{1}{3}$. The recursion turns out to be almost identical:

$$I_{3n} = \frac{1}{3}\left[I_n + h_n \,(\text{sum of new } f_i)\right]$$

We can figure this out by the same method of populating values in a mesh, and then producing the finer mesh by subdividing each interval into 3 parts. Note the same, somewhat more involved construction of these new values. If we consider the current mesh, we can split the new values obtained into LEFT (L) and RIGHT (R) arrays of size n (the full mesh is size n+1). The L array is obtained by adding $\frac{H}{3}$ into the endpoints less the final point and R is similarly obtained by subtracting the same value from the endpoints less the first point. These values now make the Global Mesh (the new mesh) - but this too needs to be constructed as before. Note the interleaving order again (by construction), but this time we interleave in the order: current mesh, L, R, and this returns the global mesh for the next iteration. This allows us to have complete re-use since we only evaluate the function at the new values determined by the new points of the global mesh in L and R.

Note also that the points of the evaluation are not the points of the global mesh i.e. the points in L and R. The evaluation points are at the mid points of the global mesh endpoints. However, we don't want to iterate over the full mesh since we only require the new values. However, the new evaluation values are completely

determined by the L and R values; we divide the new local length ($\frac{H}{3}$) by 2 and add to the L array (shift using broadcasting) and subtract from the R array similarly, and these determine all the new evaluation points.




```
function [integrals, errors, estimates] = midadaptive(func,mesh,H,n,true_int, ref_num)

    % Initializing results
    integrals = zeros(1,ref_num+1);
    errors = zeros(1,ref_num+1);
    estimates = zeros(1,ref_num);

    % Starting recurrence
    [int_start, error_start] = midpoint(func,mesh,H,n,true_int);
    integrals(1) = int_start;
    errors(1) = error_start;

    int = int_start;
    h_shift = H;
    mesh_current = mesh;

    for i = 2:ref_num+1
        h_old = h_shift;

        % New mesh values
        h_shift = h_shift/3.0;
        mesh_left = mesh_current(1:n) + h_shift;
        mesh_right = mesh_current(2:n+1) - h_shift;

        % New values for evaluation (since h = H/2)
        h_local = h_shift/2.0;
        eval_left = mesh_left - h_local;
        eval_right = mesh_right + h_local;

        % Complete re-use - only evaluating at new values
        fvals_new_left = func(eval_left);
        fvals_new_right = func(eval_right);

        % Recurrence
        int_old = int;
        int = (1/3.0)*(int + h_old*(sum(fvals_new_left) +
        sum(fvals_new_right)));

        % Update Global Mesh
        mesh_temp = zeros(1, 3*n + 1);
        mesh_temp(1:3:end) = mesh_current;
        mesh_temp(2:3:end) = mesh_left;
        mesh_temp(3:3:end) = mesh_right;
        mesh_current = mesh_temp;
        % I_{3n)
        n = 3*n;
```

```
        % Store results
        integrals(i) = int;
        errors(i) = true_int - int;
        % Error estimate recurrence
        estimates(i-1) = (9/8)*(int - int_old);
    end
end
```

2.9 Summary

We can summarize the properties of the methods as follows.

Method	Order	Degree of Exactness	Error Term	Complexity
Left Rectangle Rule	1	0	$-\frac{(b-a)}{2} H_n f^{(1)}(\xi) + \mathcal{O}(H_n^2)$	n
Trapezoidal Rule	2	1	$-\frac{(b-a)}{12} H_n^2 f^{(2)}(\xi) + \mathcal{O}(H_n^3)$	$n+1$
Simpson's Rule	4	3	$-\frac{(b-a)}{2880} H_n^4 f^{(4)}(\xi) + \mathcal{O}(H_n^5)$	$2n+1$
Midpoint Rule	2	1	$\frac{(b-a)}{24} H_n^2 f^{(2)}(\xi) + \mathcal{O}(H_n^3)$	n
Two-Point Open NC	2	1	$\frac{(b-a)}{36} H^2 f^{(2)}(\xi) + \mathcal{O}(H^3)$	$2n$
Two-Point Gauss–Legendre	4	3	$\frac{(b-a)}{270} f^{(4)}(\xi) + \mathcal{O}(H_n^5)$	$2n$

3 Numerical Results

We first provide evidence for the correctness of our code by testing the degree of exactness of our methods for simple monomial functions of increasing degree, and also demonstrate convergence on some higher order terms. I used the interval

$$I = [0, 10]$$

and used values of $N \in [1, 30]$.

The results are as follows:

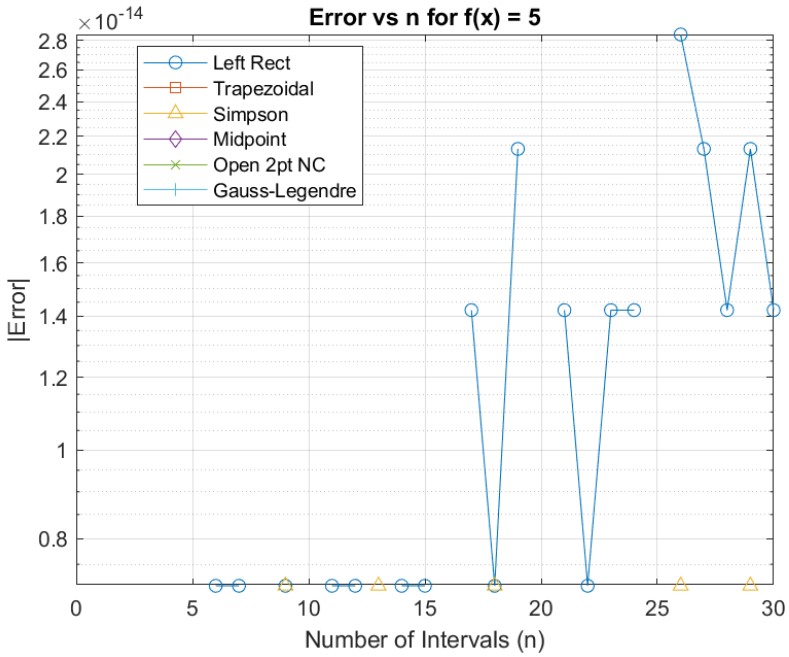


Figure 1: Error vs n for $f(x) = 5$

All methods have error on the order of double precision i.e. all results are exact for the constant function on this interval, as we expect (the only variation is pure precision error).

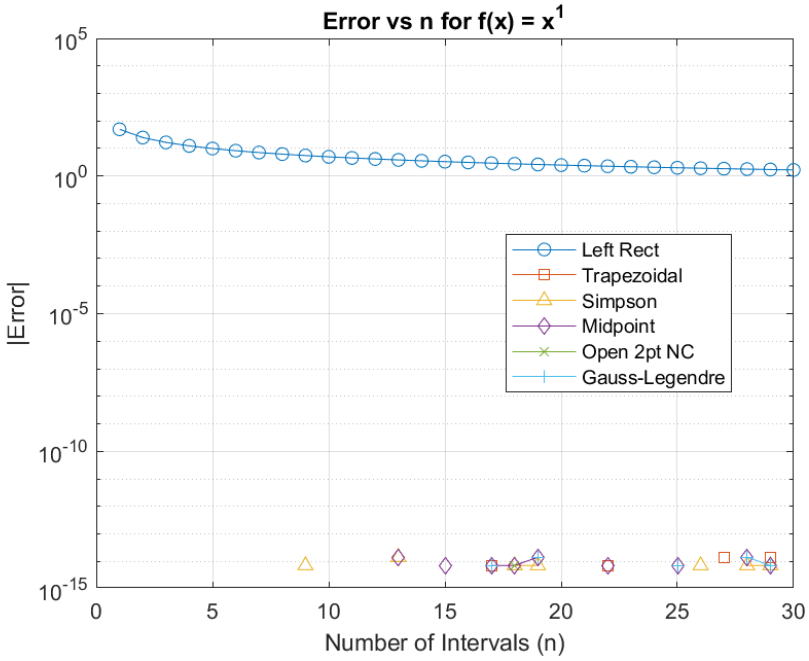


Figure 2: Error vs n for $f(x) = x^1$

All methods are still exact except the Left Rectangle Rule that has 0 degree of exactness. It is however, very slowly convergent. Mid-point is still exact, just as we expect.

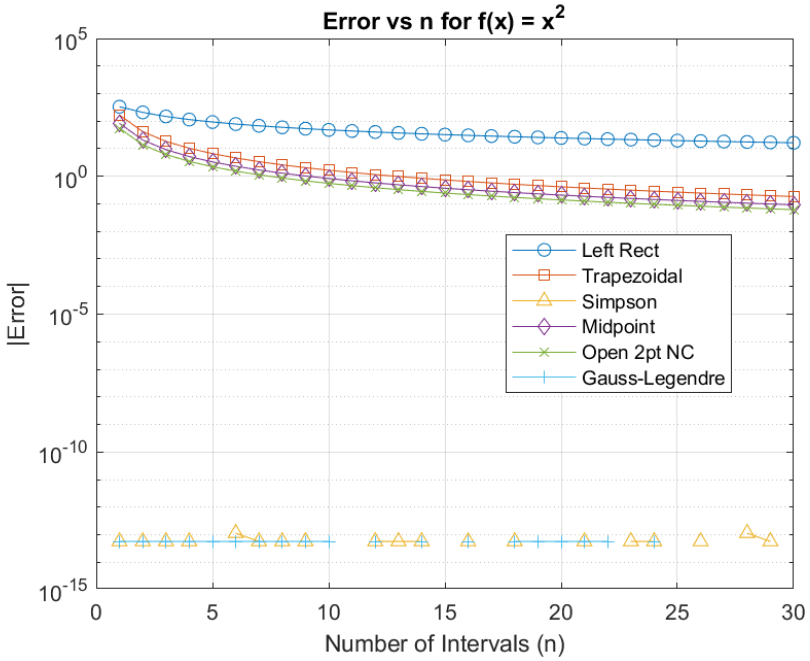


Figure 3: Error vs n for $f(x) = x^2$

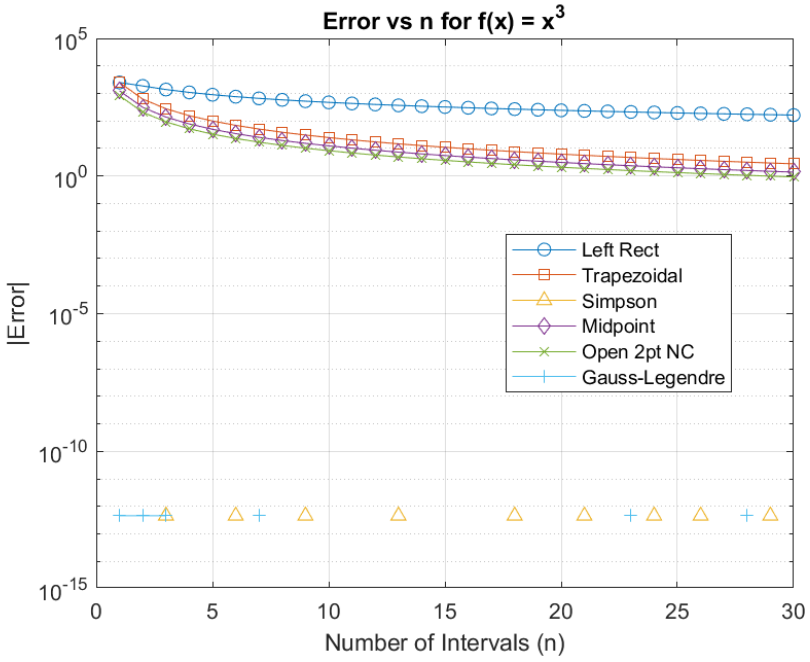


Figure 4: Error vs n for $f(x) = x^3$

Only Simpson's and Gauss-Legendre are exact for a degree 2 and 3 polynomials. The others are convergent still.

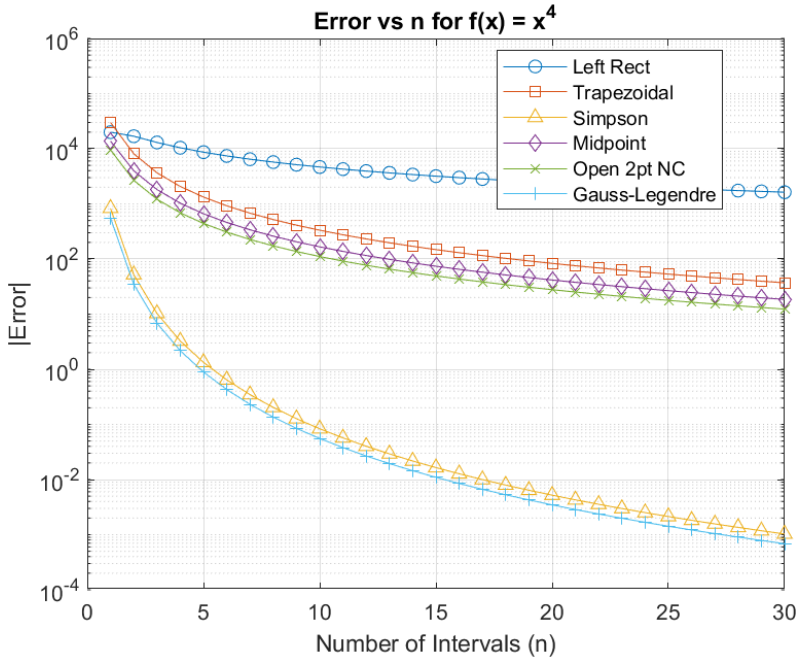


Figure 5: Error vs n for $f(x) = x^4$

None of the methods are exact now, and now we can see the convergence more clearly. Gauss-Legendre and Simpson's converge the fastest. and Gauss-Legendre does better than Simpson's everywhere.

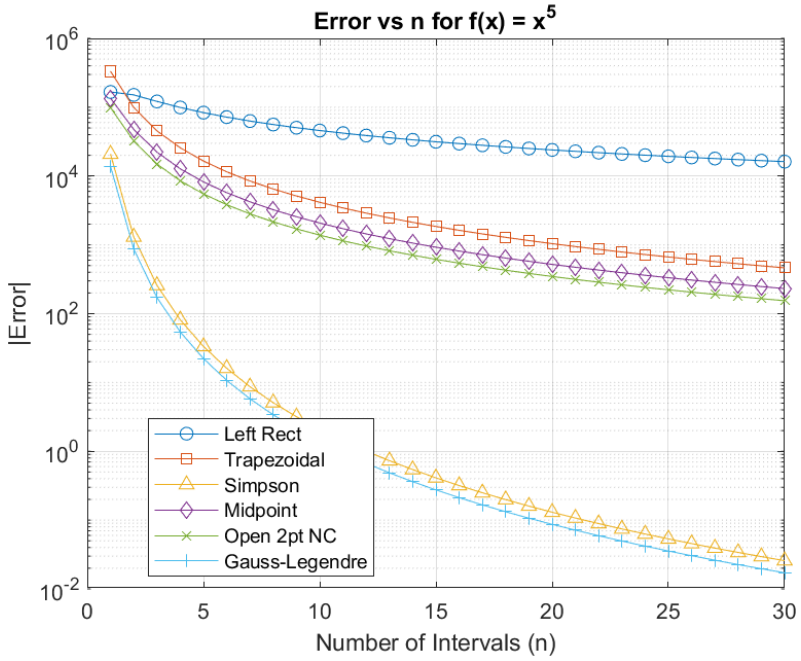


Figure 6: Error vs n for $f(x) = x^5$

The same observations can be made. We can also see that the convergence rate is getting slower for the higher degree monomial. This is also what we expect.

We now test accuracy and convergence for our methods on the following set of problems:

$$\int_0^3 e^x \, dx = e^3 - 1 \tag{1}$$

$$\int_0^{\pi/3} e^{\sin(2x)} \cos(2x) \, dx = \frac{1}{2}\left(-1 + e^{\sqrt{3}/2}\right) \tag{2}$$

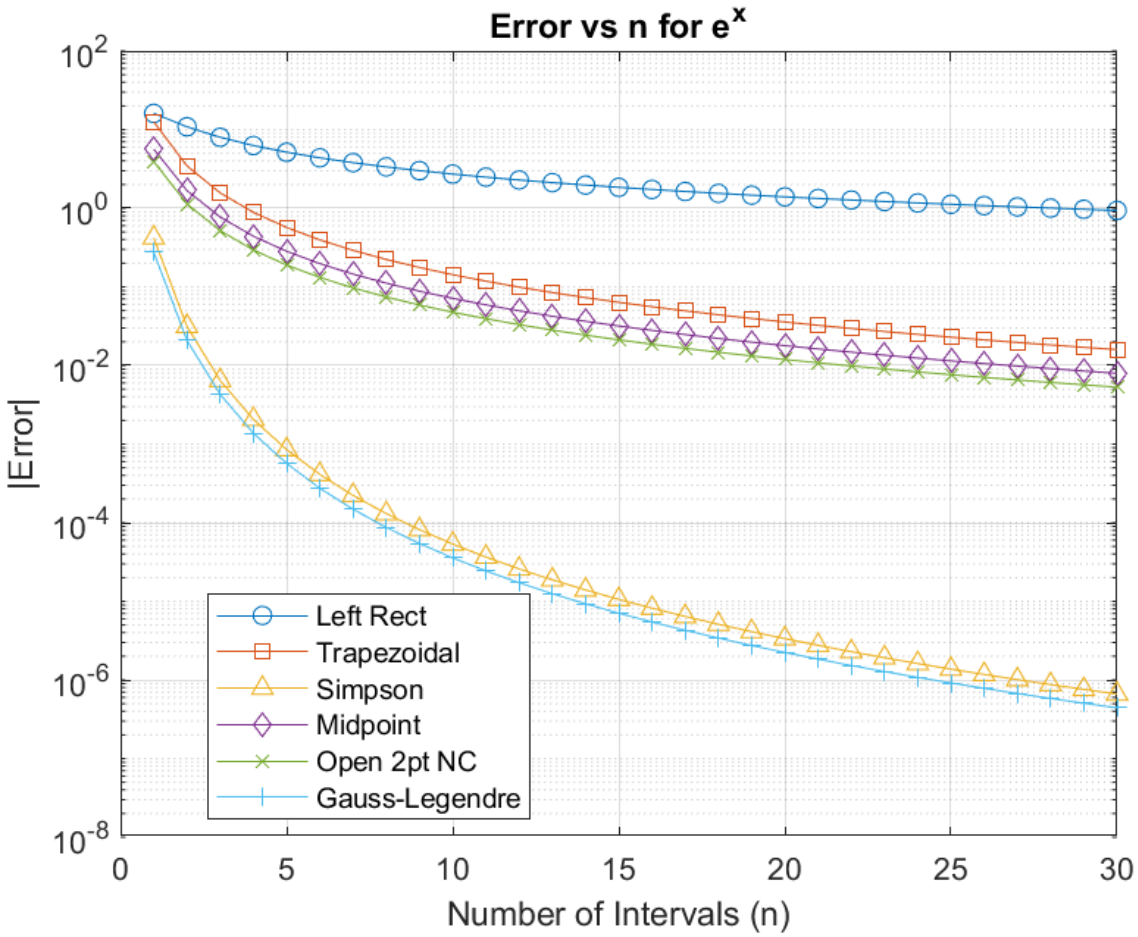
$$\int_{-2}^{1} \tanh(x) \, dx = \ln\left(\frac{\cosh(1)}{\cosh(2)}\right) \tag{3}$$

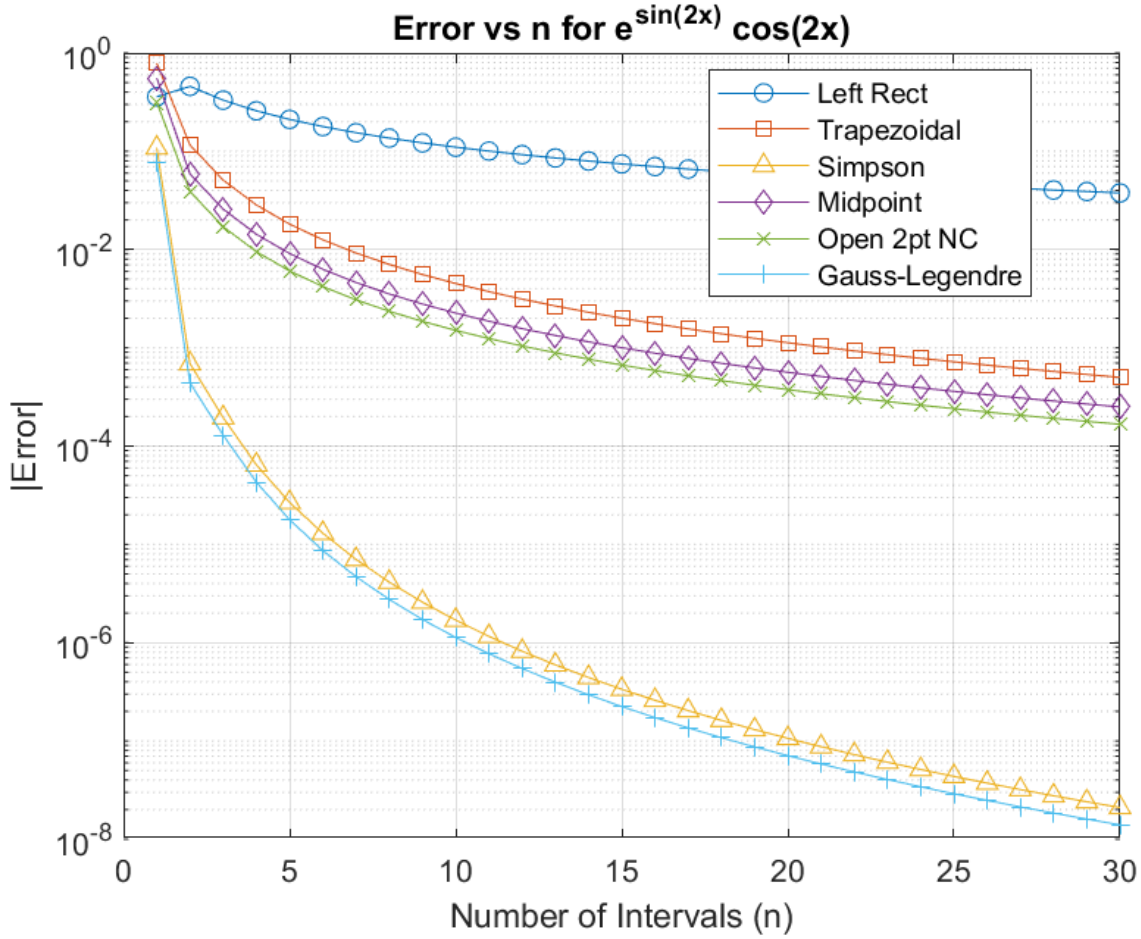
$$\int_0^{3.5} x\cos(2\pi x) \, dx = -\frac{1}{2\pi^2} \tag{4}$$

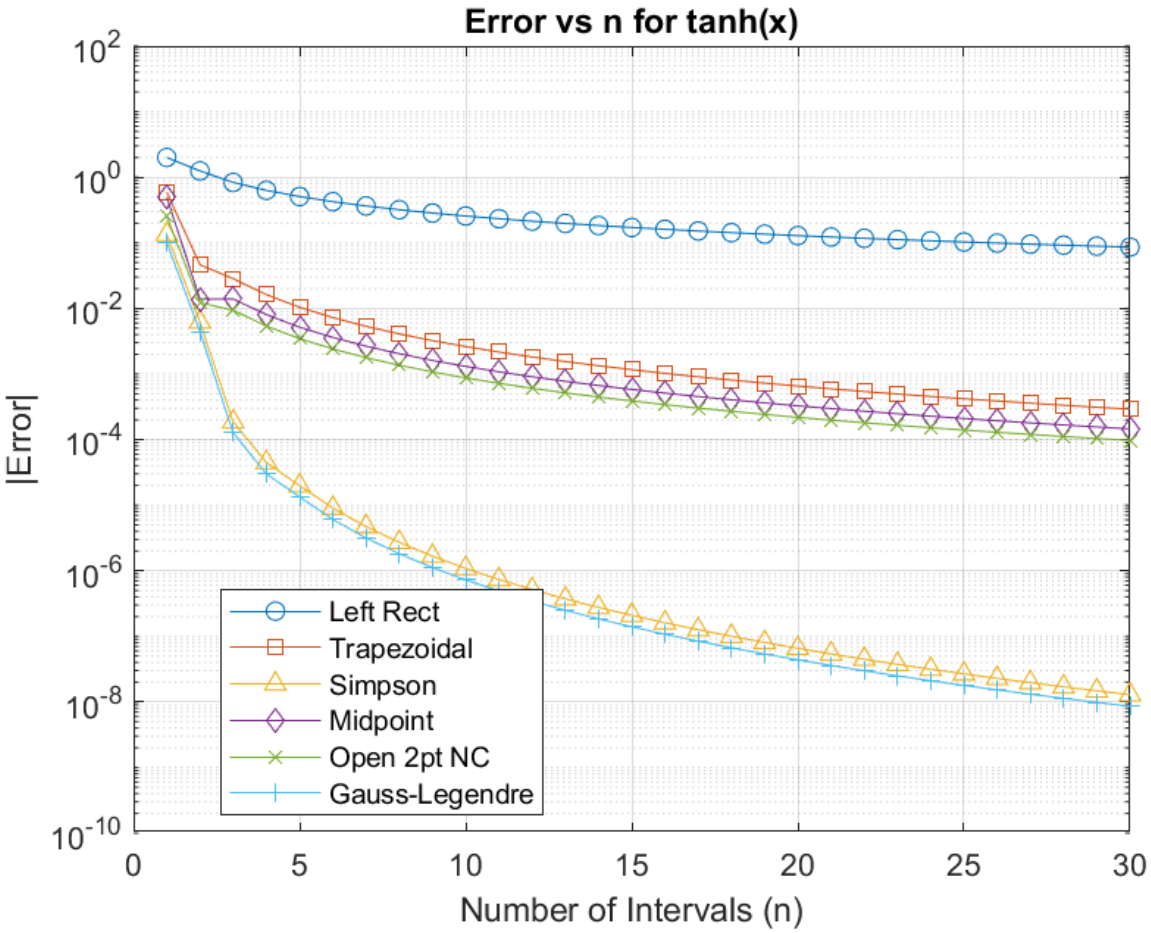
$$\int_{0.1}^{2.5} \left(x + \frac{1}{x}\right) dx = \frac{2.5^2 - 0.1^2}{2} + \ln(2.5/0.1) \tag{5}$$

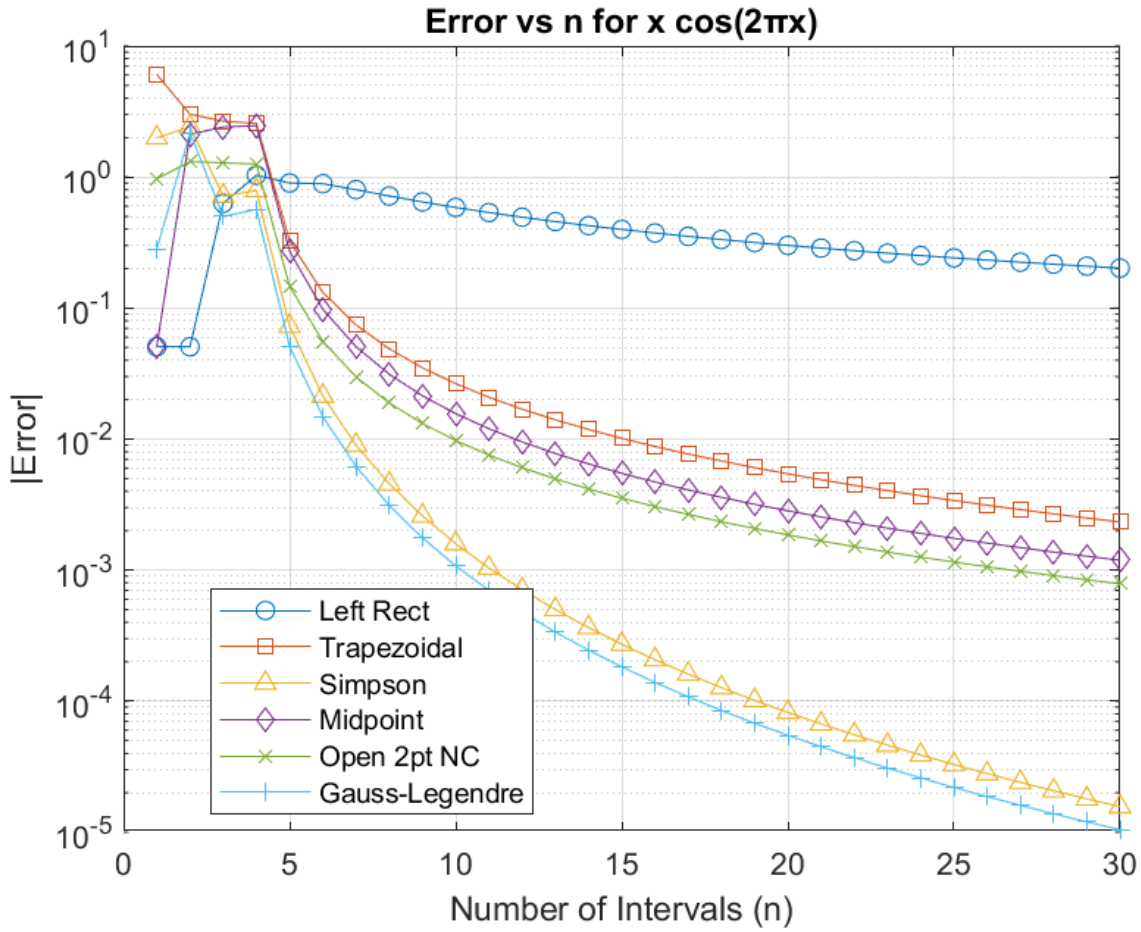
Function	**Interval**	$\lvert f^{(1)}(x)\rvert$	$\lvert f^{(2)}(x)\rvert$	$\lvert f^{(3)}(x)\rvert$	$\lvert f^{(4)}(x)\rvert$
1. e^x	$[0, 3]$	$\leq e^3$	$\leq e^3$	$\leq e^3$	$\leq e^3$
2. $e^{\sin(2x)}\cos(2x)$	$[0, \frac{\pi}{3}]$	$\leq 2e$	$\leq 4e$	$\leq 8e$	$\leq 16e$
3. $\tanh(x)$	$[-2, 1]$	≤ 1	≤ 1	≤ 1	≤ 1
4. $x\cos(2\pi x)$	$[0, 3.5]$	≤ 22.0	≤ 200.0	≤ 1800.0	≤ 16000.0
5. $x + \frac{1}{x}$	$[0.1, 2.5]$	≤ 100.0	≤ 2000.0	≤ 40000.0	≤ 800000.0

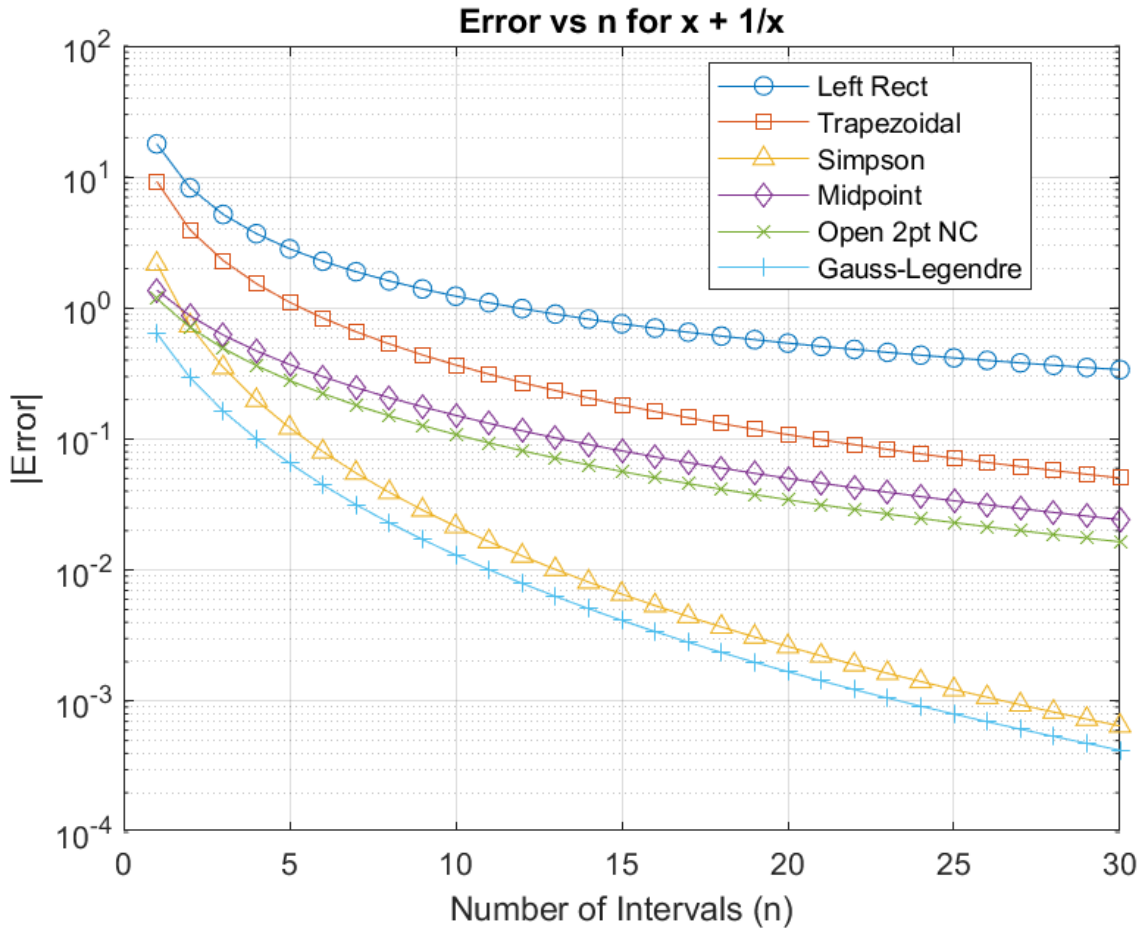
We use more and more points and plot the error for each function for each method:


Error vs n for e^x
|Error|
Number of Intervals (n)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre

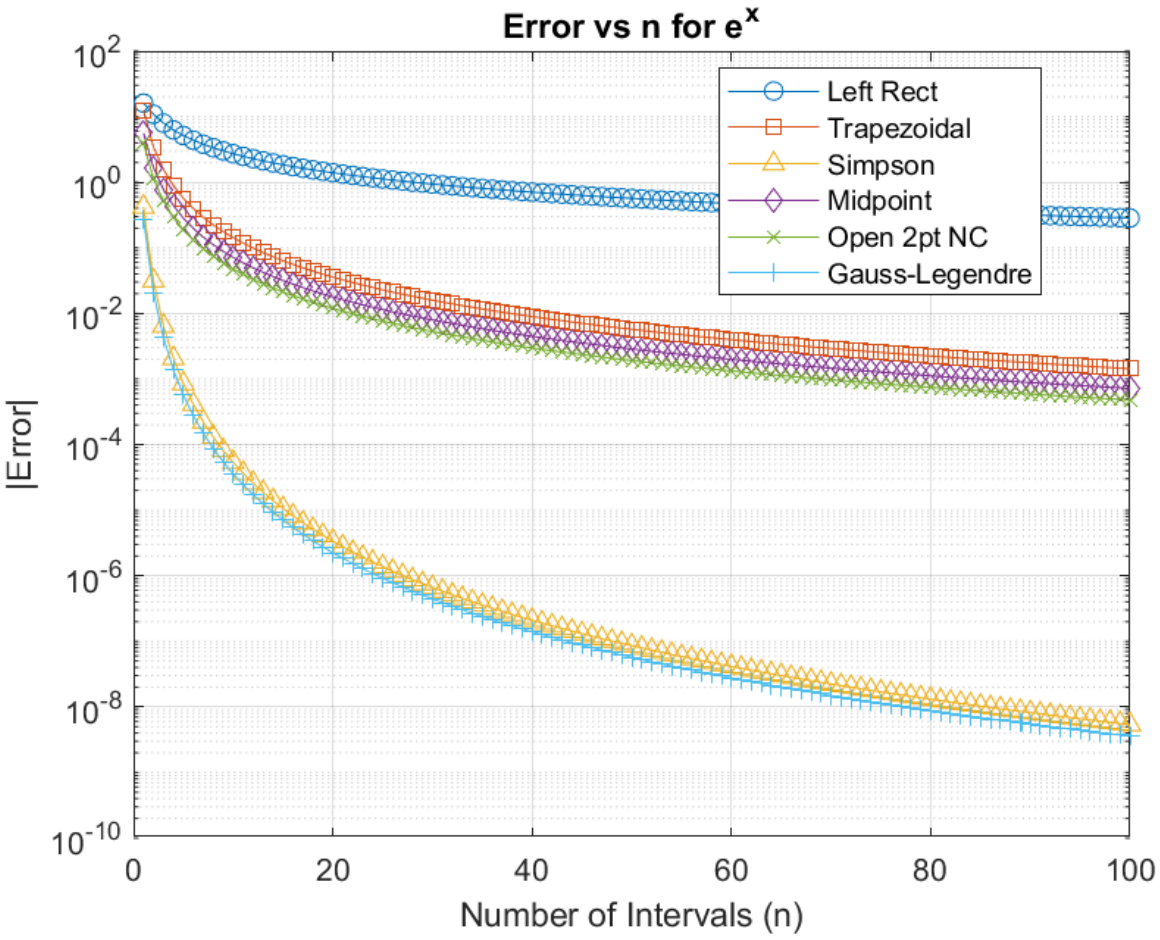

Error vs n for e^sin(2x) cos(2x)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre
|Error|
10^0
10^-2
10^-4
10^-6
10^-8
0
5
10
15
20
25
30
Number of Intervals (n)

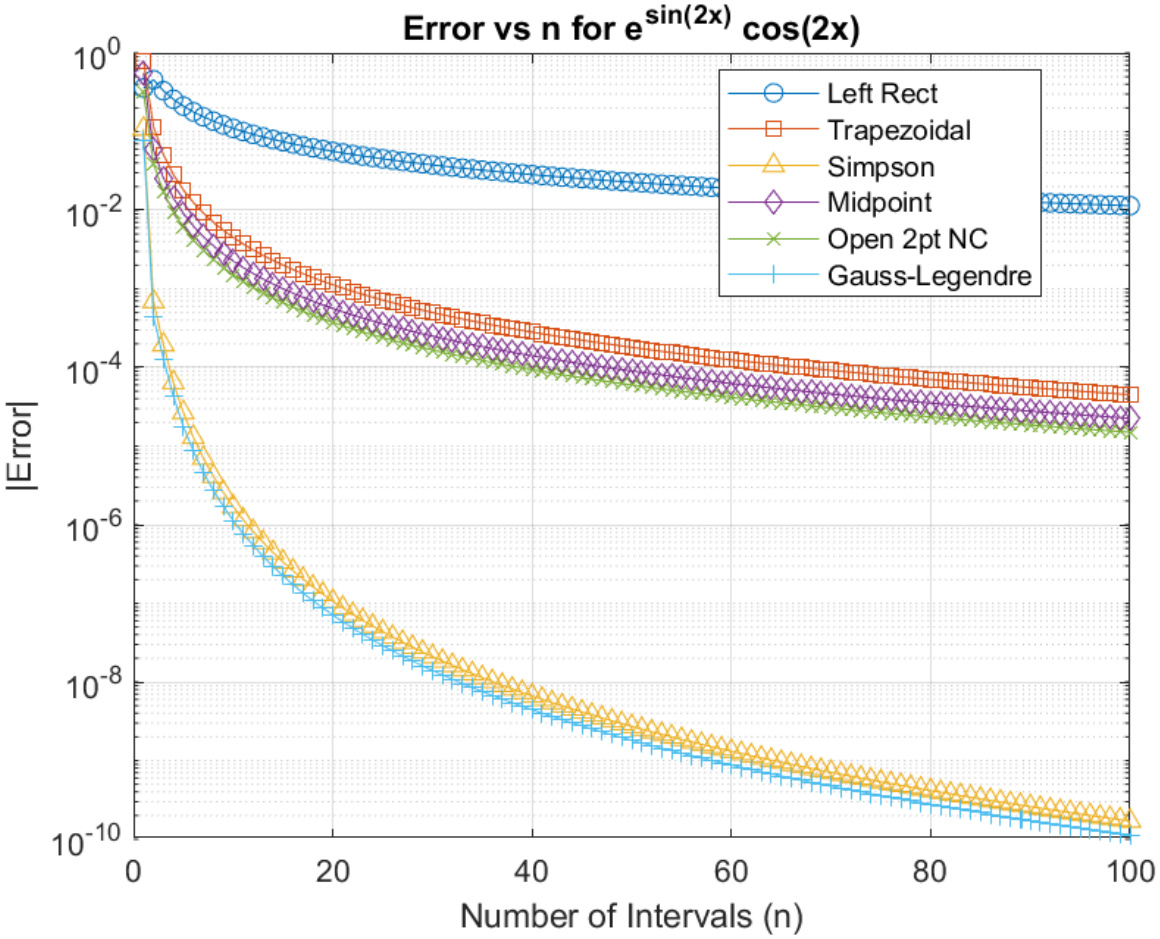

Error vs n for tanh(x)
|Error|
Number of Intervals (n)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre

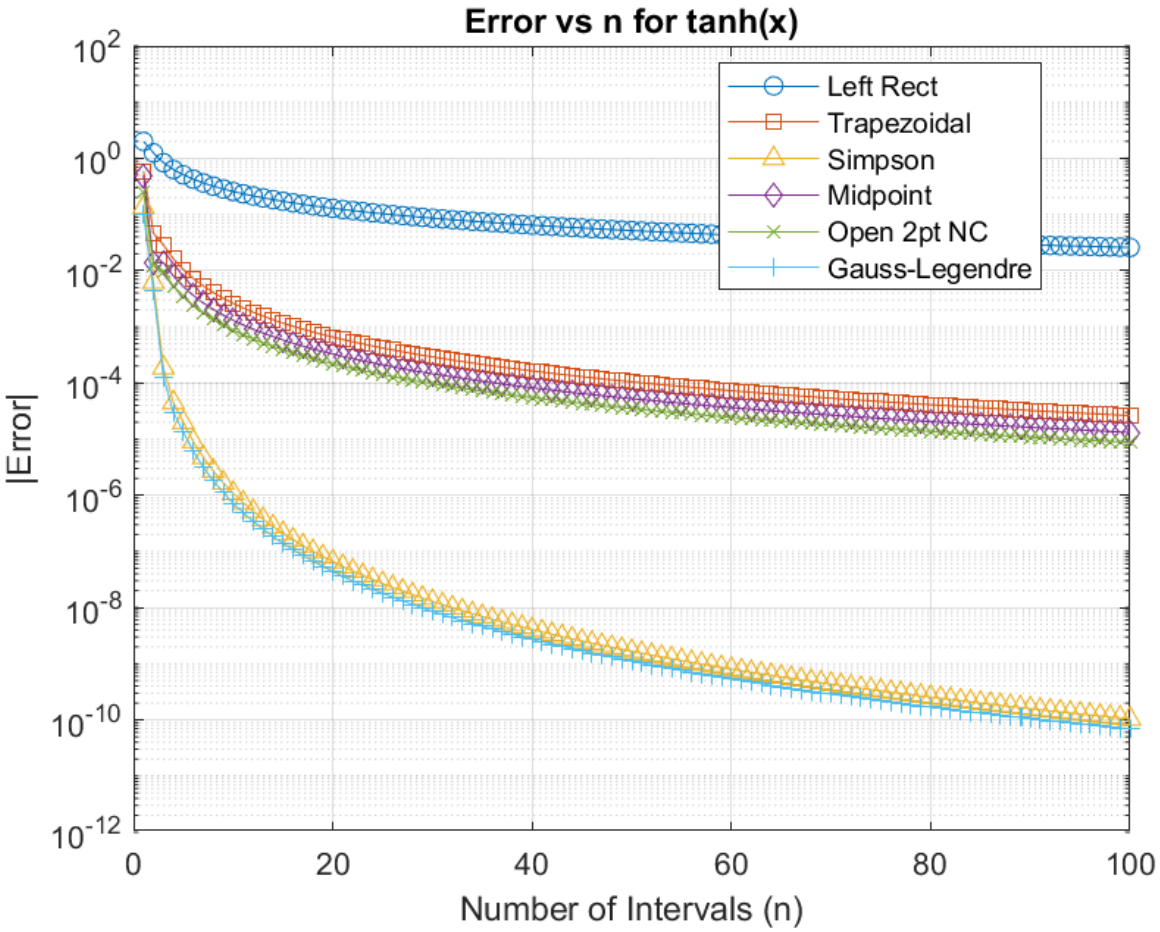

Error vs n for x cos(2πx)
|Error|
Number of Intervals (n)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre

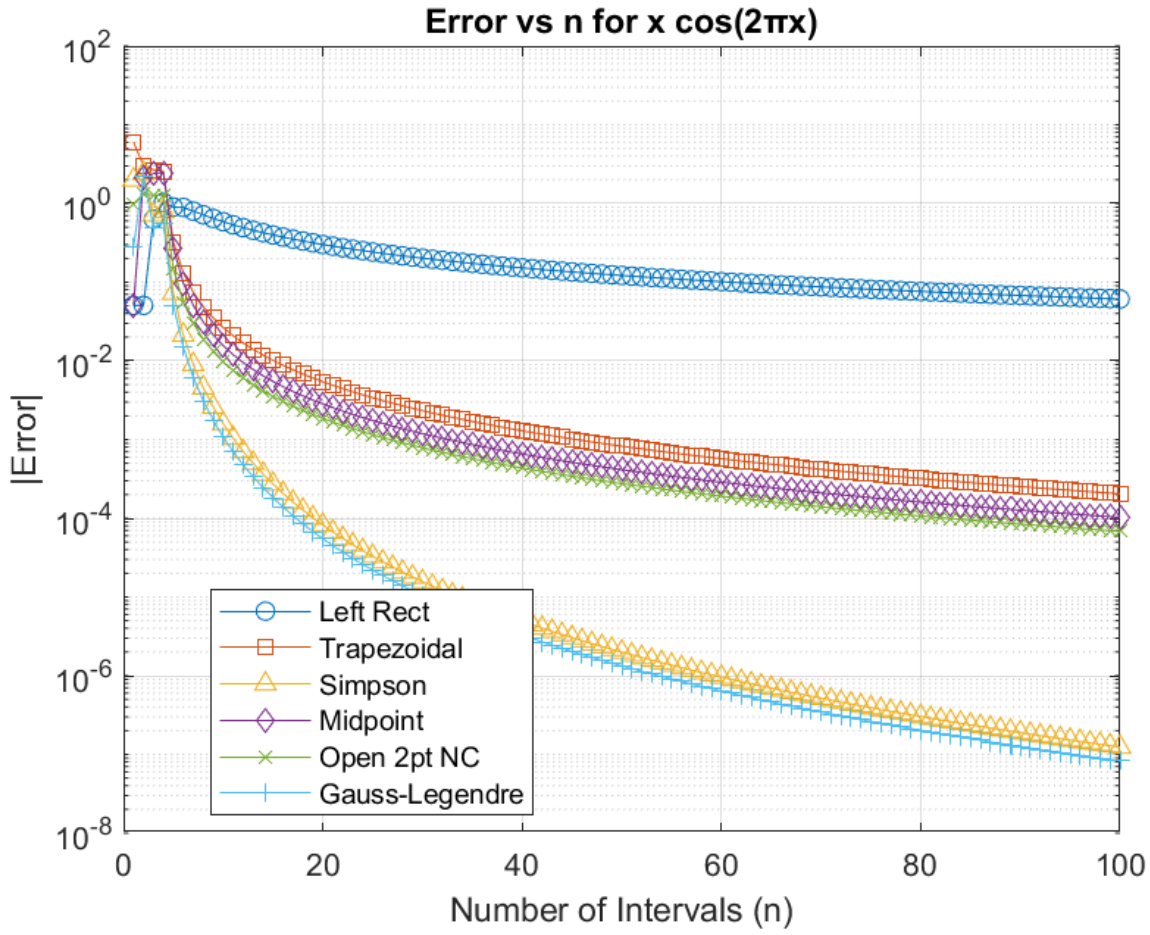

Error vs n for x + 1/x
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre
|Error|
Number of Intervals (n)
10^2
10^1
10^0
10^-1
10^-2
10^-3
10^-4
0
5
10
15
20
25
30

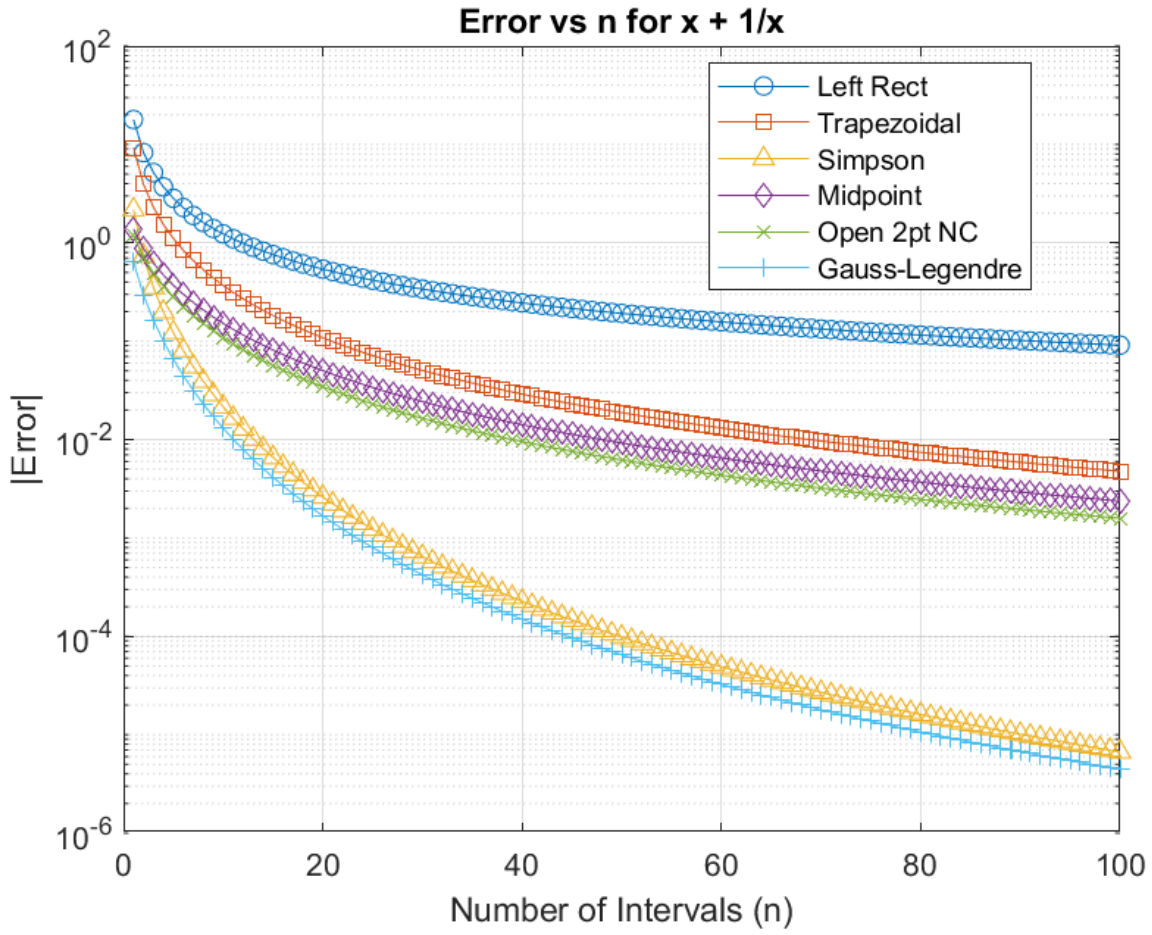
We can then increase the **resolution** to a 100 intervals to fully see the asymptotic behavior of the methods:




Error vs n for e^sin(2x) cos(2x)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre
|Error|
10^0
10^-2
10^-4
10^-6
10^-8
10^-10
0
20
40
60
80
100
Number of Intervals (n)


Error vs n for tanh(x)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre
|Error|
Number of Intervals (n)
10^2
10^0
10^-2
10^-4
10^-6
10^-8
10^-10
10^-12
0
20
40
60
80
100


Error vs n for x cos(2πx)
|Error|
Number of Intervals (n)
Left Rect
Trapezoidal
Simpson
Midpoint
Open 2pt NC
Gauss-Legendre



Conclusions: The hierarchy of method accuracy is maintained as before (Left Rectangle < Trapezoidal < Midpoint < Open 2-NC < Gauss Legendre), and the convergence rates can also be visually substantiated. We can also see that for the last two functions with very large values for the derivatives, the convergence slows down for each method (note the scale of the y-axes). This is also what we expect. For oscillatory functions, the smaller number of interval have chaotic behavior.

3.1 Adaptive Methods

We can test this code for the same functions as before. Note that simple examples and the table for the integral and the estimates from the slides were reproduced in the live script and are available in the code.

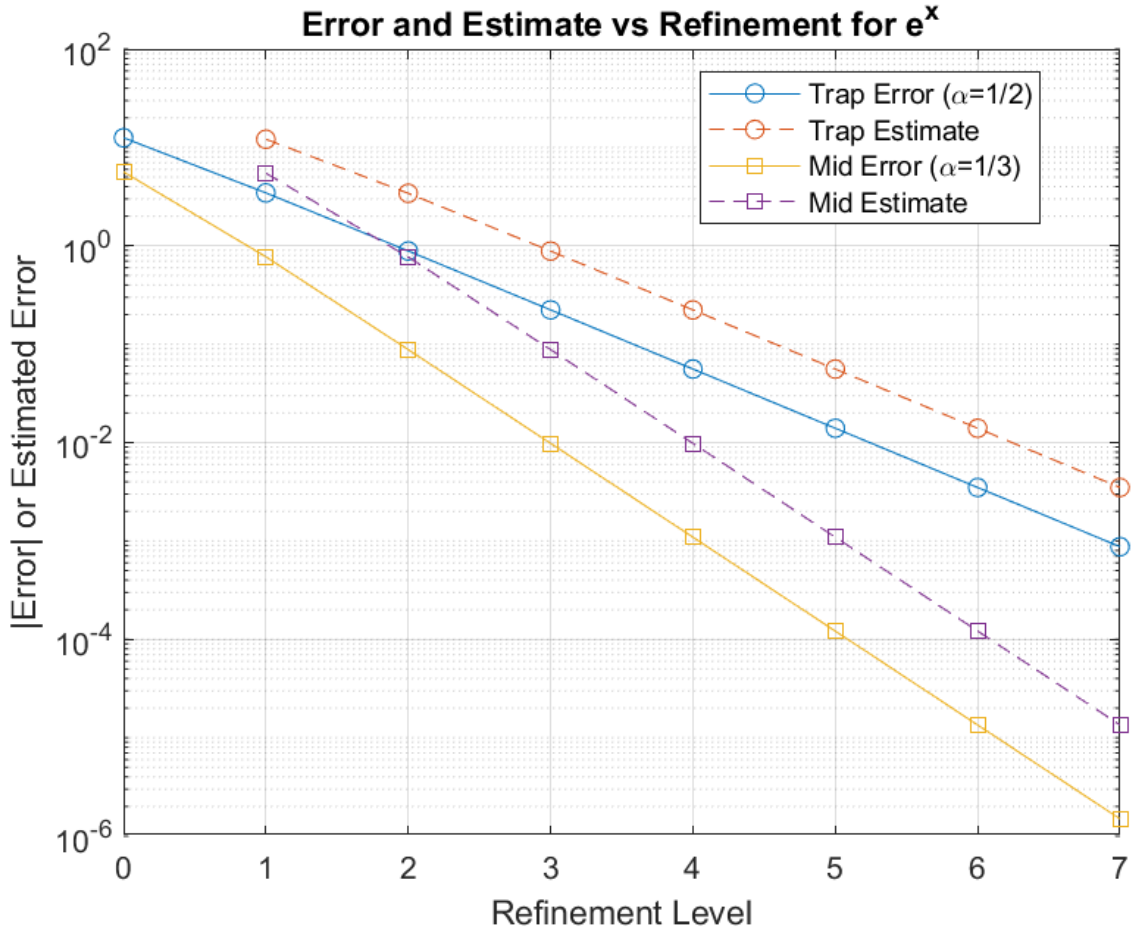


Figure 7: Error and Estimated Error vs Refinement for $f(x) = e^x$

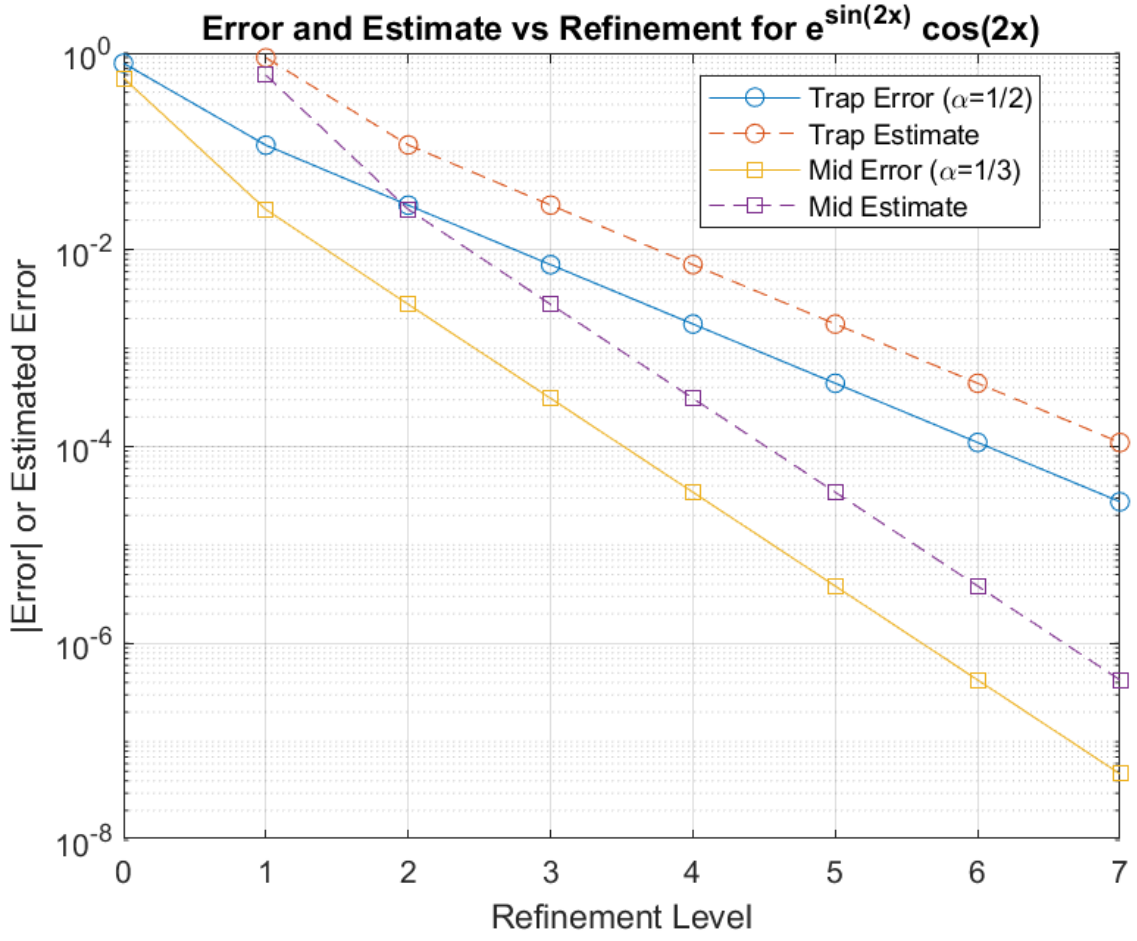


Figure 8: Error and Estimated Error vs Refinement for $f(x) = e^{\sin(2x)}\cos(2x)$

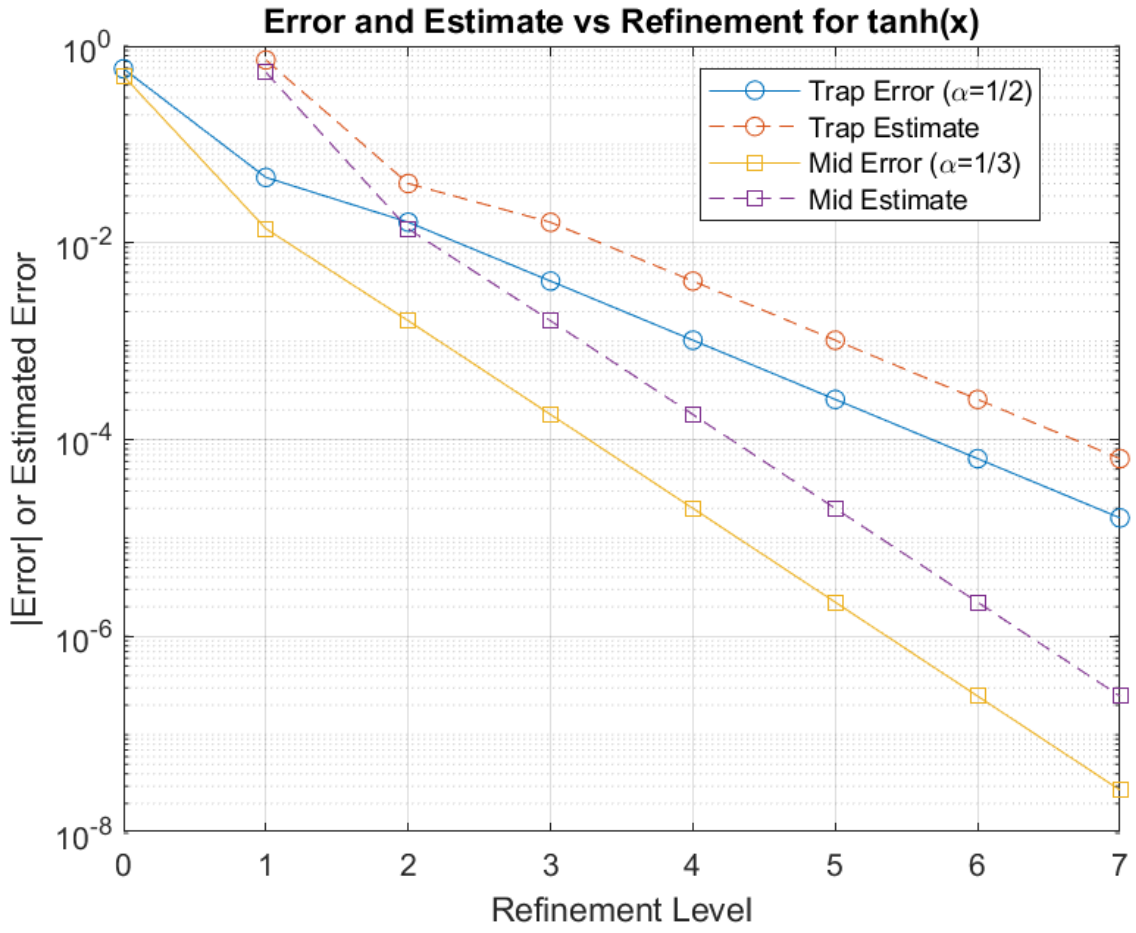


Figure 9: Error and Estimated Error vs Refinement for $f(x) = \tanh(x)$

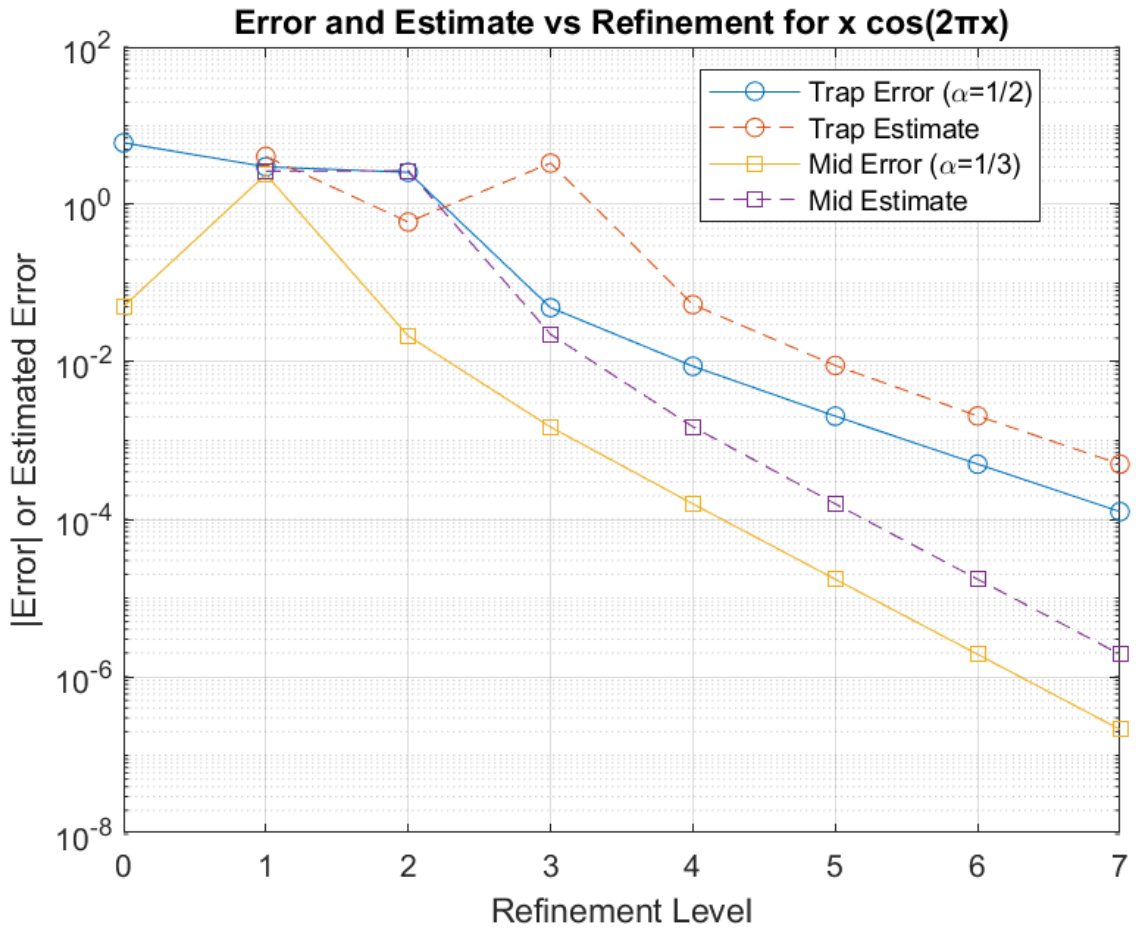


Figure 10: Error and Estimated Error vs Refinement for $f(x) = x\cos(2\pi x)$

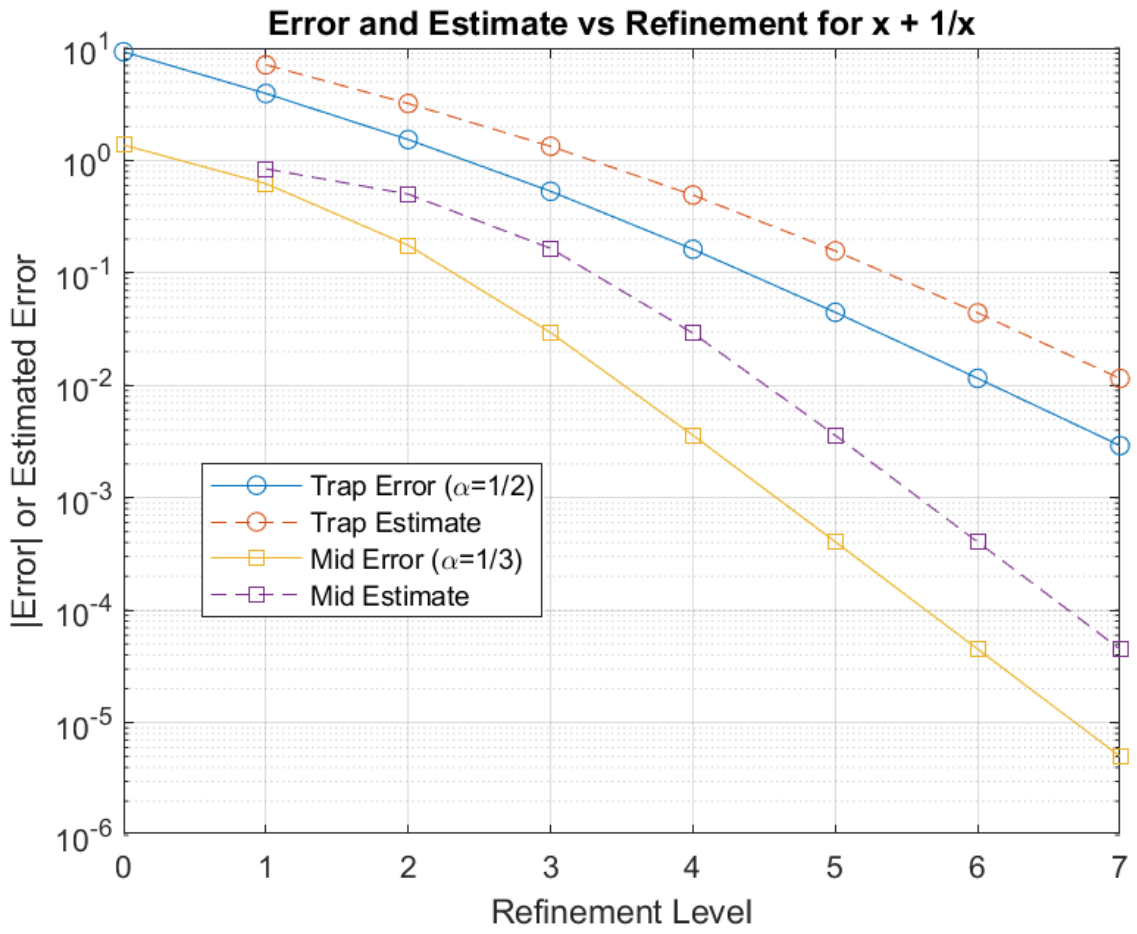


Figure 11: Error and Estimated Error vs Refinement for $f(x) = x + \frac{1}{x}$

We can see that the adaptive midpoint performs better than the trapezoidal, as we expect, and the error estimates track the true error very well (in fact up to an order, since the difference in the graphs is staying constant). The linear decline in error is also observed, due to the constant step refinement. Note that the graphs are comparable to the composite methods by matching the number of points on the axis using the appropriate value of alpha.

4 Conclusions

We designed, described and implemented composite quadrature routines as efficiently as possible, and then evaluated the correctness and efficiency of our codes. We also compared the numerical results to theoretical predictions, and then in particular evaluated the crude composite routines against adaptive methods for two particular types of methods and mesh refinements.