Assignment 1

COSC2500
Ryan White
44990392

24th of August 2020

Required

R1.1

The *xy* data for the smoothed and averaged intensity of the laser beam creates a 'curve' as shown in Figure 1:

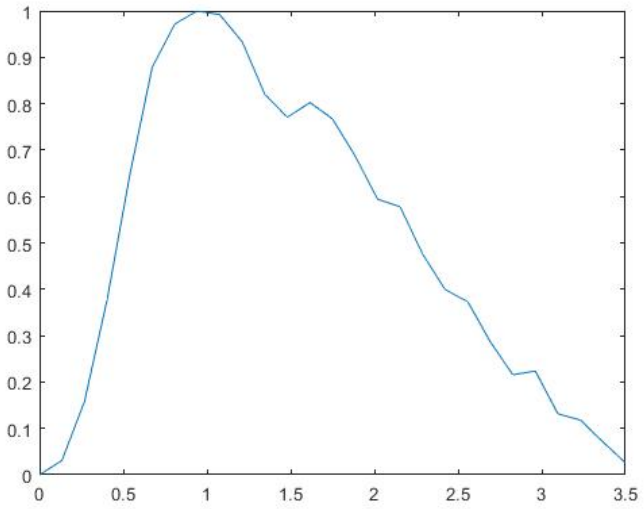


Figure 1: Data for Smoothed Laser Beam Intensity

Using the Matlab functions `polyfit` and `polyval`, and the code shown in Appendix 0.1, the three following trendline approximations were created:

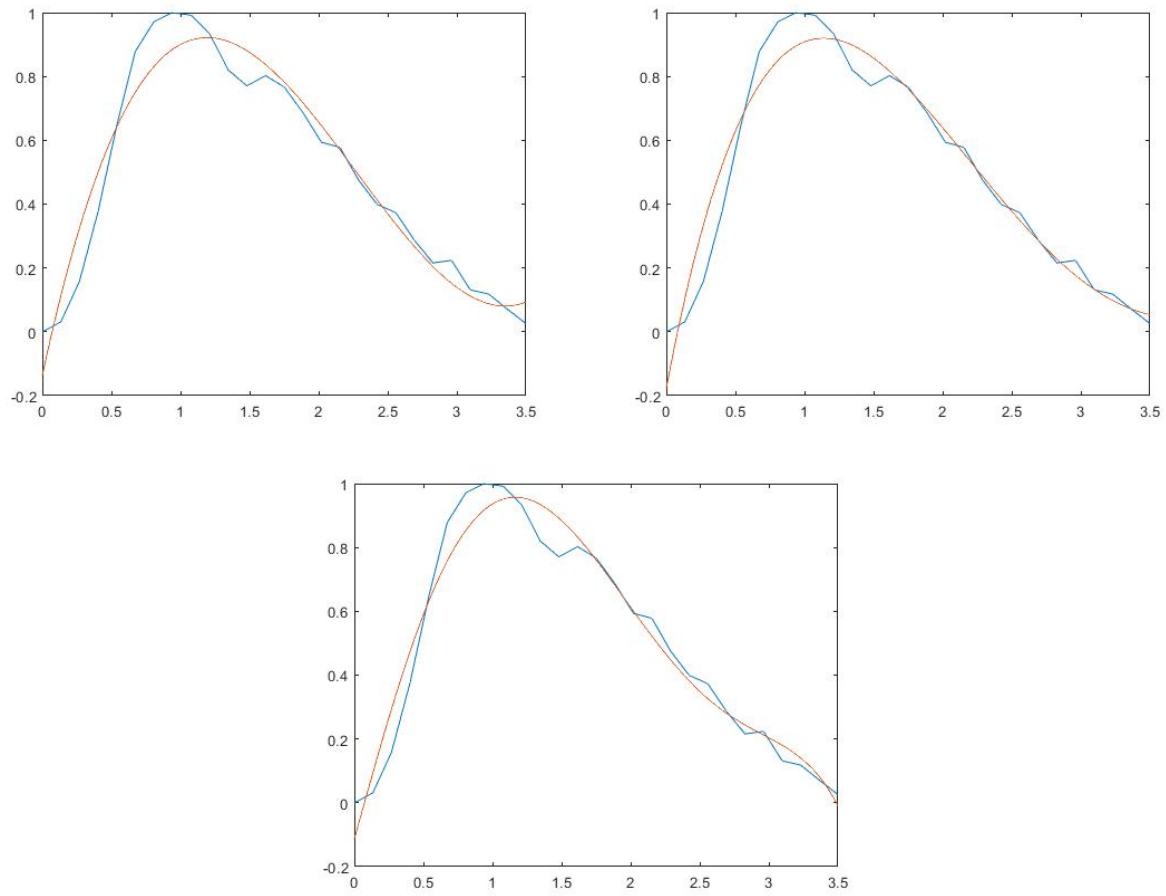


Figure 2: Polynomial Approximations of Orders 3 to 5
From Top-Left to Right-Bottom: $n = 3$, $n = 4$, $n = 5$

Trial-and-error was used to determine the optimal polynomial order to approximate the xy data. It was determined that a polynomial approximation of order $n = 4$ was ideal, mainly due to it's close fit to the 'tail' of the data, and the fact that the curve didn't approach a positive gradient again towards the end of the appropriate domain (as it had done with $n = 3$). It appears that the $n = 4$ polynomial more closely approximated the tail than even the $n = 5$ approximation, and that together with it's lower-order formula, was the determining factor in the chosen order.

The hint given in the "Programming Hints" document on Blackboard was utilised in plotting the final graphs. The use of 1000 points (as opposed to the 27 given data points) allows for more accurate predictions of data based on the model, specifically at points where the original experiment yielded no data.

R1.2

Given the polynomical $f(x) = x^4 - 2x^3$, the first and second derivatives were found analytically to be

$$\begin{aligned} f'(x) &= x^4 - 2x^3 \frac{\mathrm{d}}{\mathrm{d}x} \\ &= 4x^3 - 6x^2 \end{aligned} \tag{1}$$

and

$$\begin{aligned} f''(x) &= 4x^3 - 6x^2 \frac{\mathrm{d}}{\mathrm{d}x} \\ &= 12x^2 - 12x \end{aligned} \tag{2}$$

a. For $x = 0$, there seems to be no discernible difference between the three differentiation methods in terms of their relative error from the exact value. This is directly contrasted by the cases where $x \neq 0$ where the central difference method routinely comes with the least error (often for a significantly larger step size). Table 1 shows the respective minimum error and it's corresponding step size across the 0, 1, 1.5, and 2 x values:

x Value	Minimum Error ($\times 10^{-13}$)			Step Size ($\times 10^{-6}$)		
	F	B	C	F	B	C
0	-	-	-	-	-	-
1	3.406	1.846×10^{1}	3.064×10^{-1}	2.694	4.906	2.549
1.5	4.767×10^{5}	5.179×10^{5}	7.256×10^{2}	1.863×10^{-2}	1.715×10^{-2}	6.12
2	7.484×10^{3}	2.165×10^{4}	1.62×10^{1}	9.462×10^{-3}	6.789×10^{-3}	4.184×10^{-1}

Table 1: Minimum Errors and Corresponding Step Sizes for Different Methods/x Values

Where in the above table, F is for Forward, B for Backward, and C for Central Difference.

b. Manually adding error to the polynomial significantly increases the minimum error, and increases the corresponding step size of said minimum error. The larger the manual error added, the larger the resultant minimum error becomes. It also seems that manually increasing the error also shifts the point of minimum error to larger and larger step sizes. For example, take Figure 3

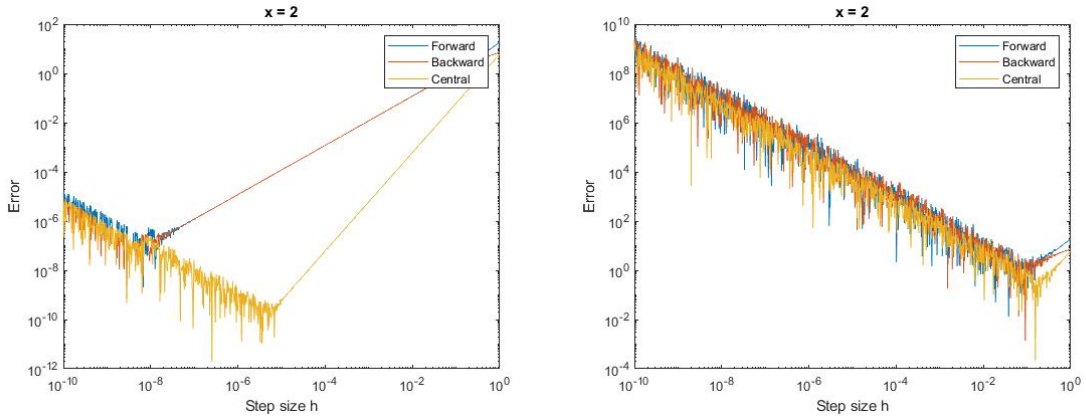


Figure 3: Relative Error vs Step Size for Standard and Manual Error Models
Left: Standard Model, Right: Added Error

Notice the change in Error order of magnitude from left to right, and the shifting of the minima to larger step sizes.

R1.3

a. Using the code shown in Appendix 0.5, Simpson's Rule and the Trapezoid Rule were compared in terms of relative error over a range of step sizes h. For the functions and domains listed in the criteria, the comparisons are shown below in Figure 4.

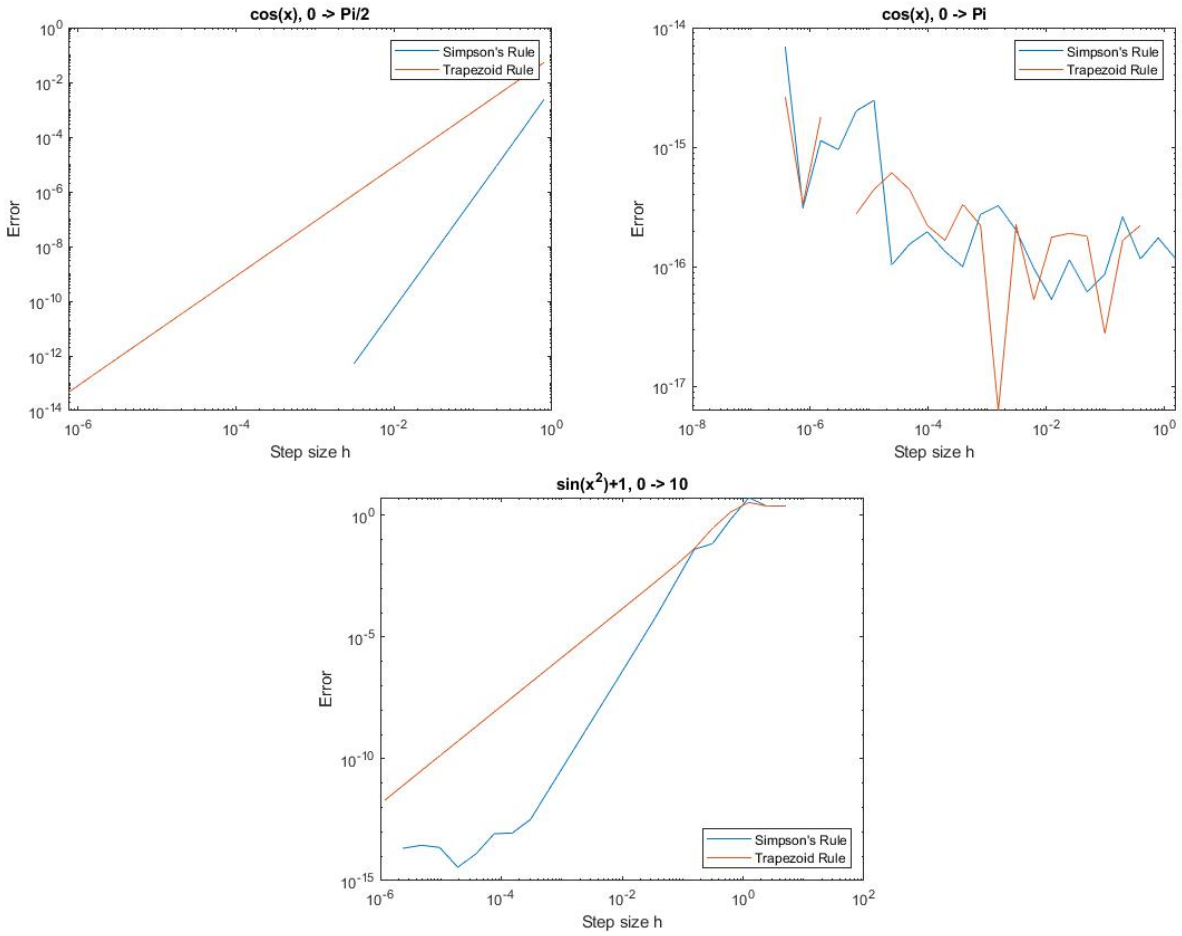


Figure 4: Comparison of Simpson's vs Trapezoid Rule Accuracy for Specified Functions and Domains

As can be seen, Simpson's Rule is typically more accurate for any given step size, while both the Trapezoid and Simpson's Rule are extremely accurate for $\cos(x)$ from $0 \to \pi$.

b. Figure 5 compares the accuracy of the Trapezoid and Simpson's rules vs the Matlab functions `quad()` and `integral()`

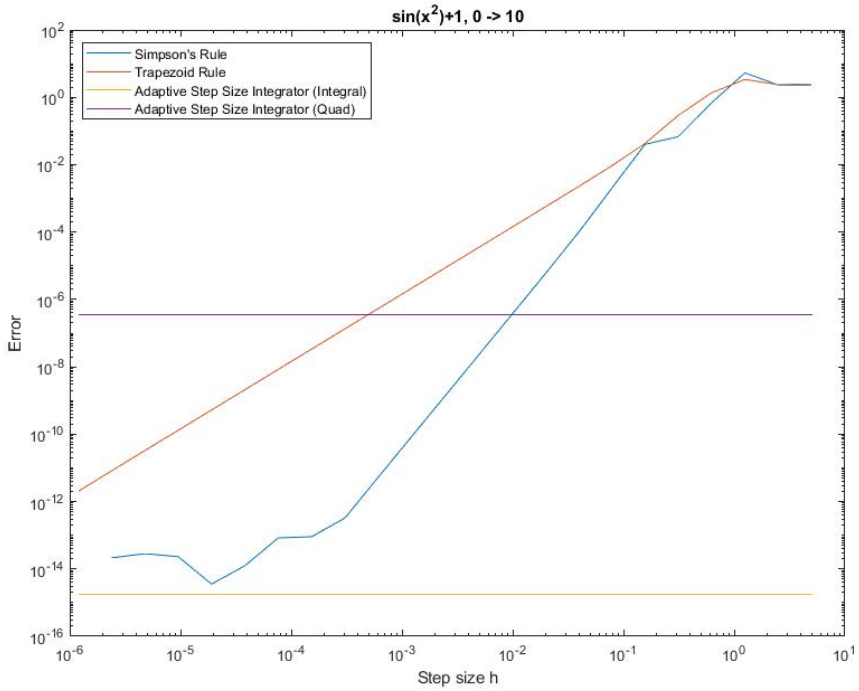


Figure 5: Accuracies of Different Integration Methods vs Step Size

As can be seen, the two adaptive step size integrators have a constant error, which makes them more desirable than the Trapezoid or Simpson's rules for larger step sizes. In the domain tested, the `integral()` function has a lower error than any of the other functions tested.

c. Figure 6 shows the relative error of the two integration rules when there is some manual (but random) error added.

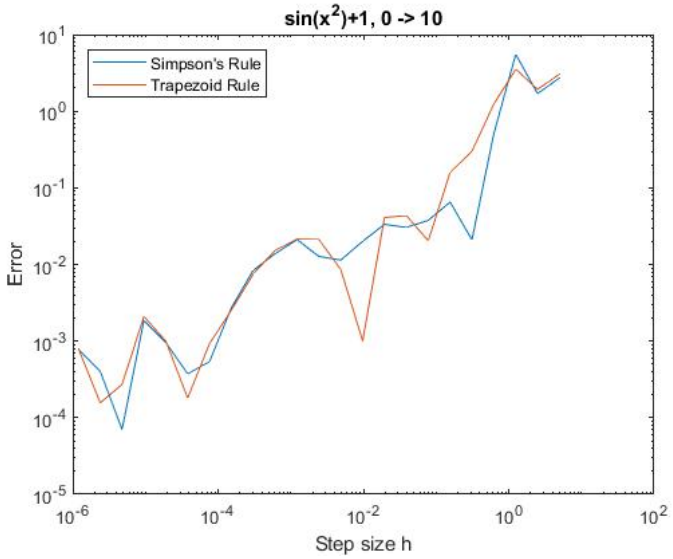


Figure 6: Relative Error vs Step Size for Integration Rules with Random Error

Comparison of Figures 4 and 6 show just how much the relative error of the functions increased with random noise error added to them. For small step sizes, the error is as much as 10 orders of magnitude larger. The random error also effectively dismissed the advantage of using Simpson's Rule over the Trapezoid Rule, as Figure 6 shows that the error between them is more or less comparable over the domain of step sizes.

d. My PC had run into memory issues before rounding error became noticeable. Step sizes on the order of magnitude of 10^{-8} (and a total of about 270 million steps) were calculated with no significant rounding error visible. Based on this, I would argue that time/memory issues are much more prevalent in running these functions as opposed to the associated rounding error.

R1.4

Figure 7 shows the relative error of the two integration rules for a fixed number of steps (specifically 999 steps) for a varying bound of integration.

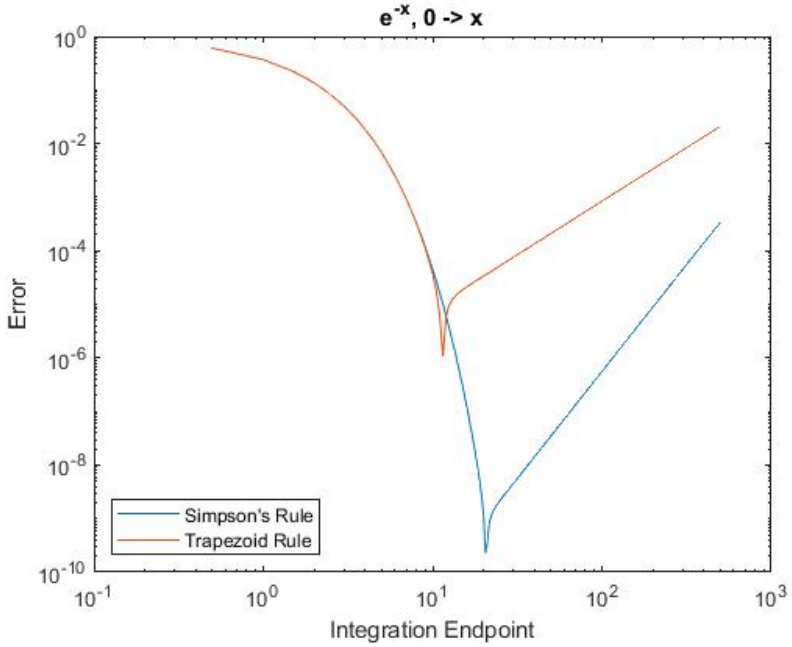


Figure 7: Error vs Integration Endpoint for $f(x) = e^{-x}$, $x = 0 \to x$

As can be seen, Simpson's Rule is comparable (in terms of error) to the Trapezoid Rule for $x < 10$, and much more accurate for values thereafter (excluding $x \approx 11.5$). Figure 8 compares the error vs integration endpoint for a fixed step size vs fixed number of steps.

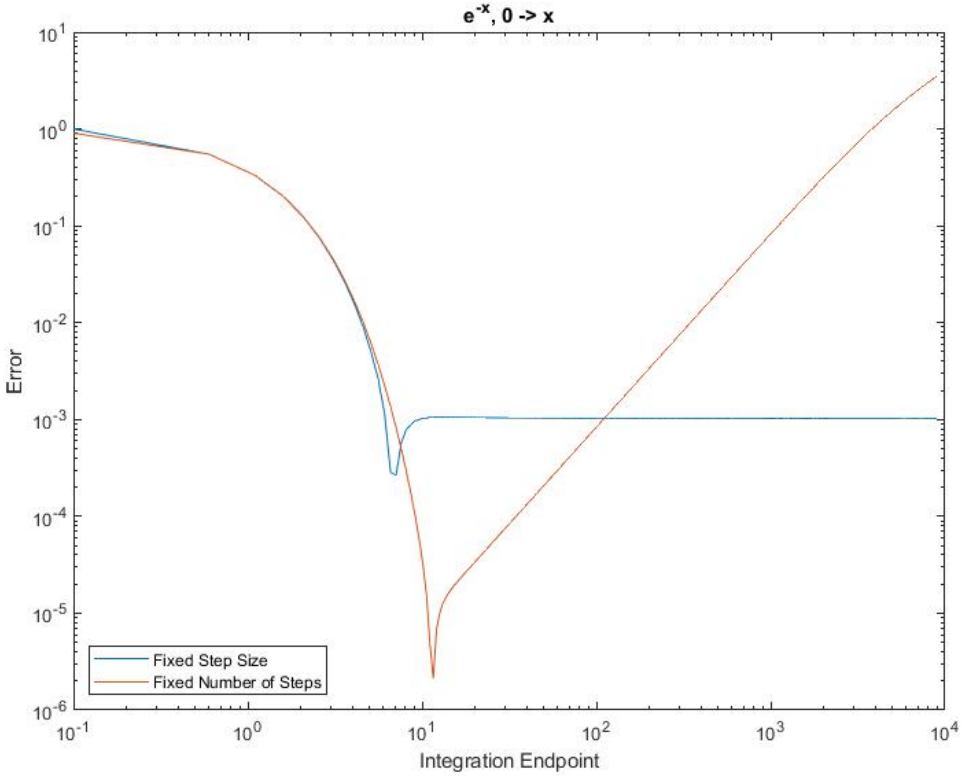


Figure 8: Error vs Integration Endpoint for Different Step Methods

As can be seen, a fixed number of steps for the trapezoid rule yields higher and higher error as the endpoint is increased. In contrast, the error for a fixed step size converges to about 10^{-3} as the endpoint is increased. Thus, for values of about $7 \leq x \leq 90$, it is more accurate to use a fixed number of steps, while outside of this domain it would be better to use a fixed step size.

Appendices

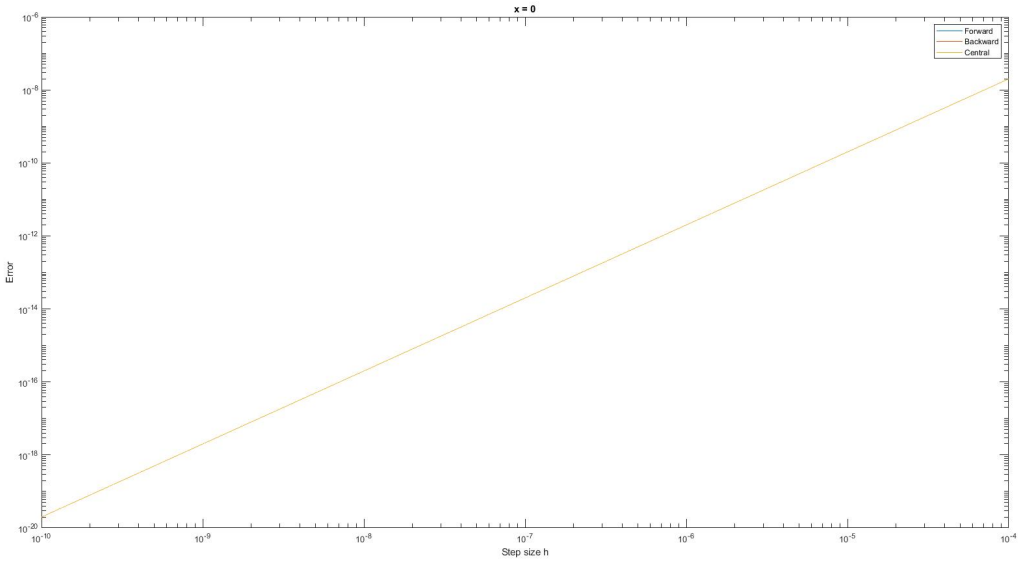
0.1 R1.1 Polyfit Code

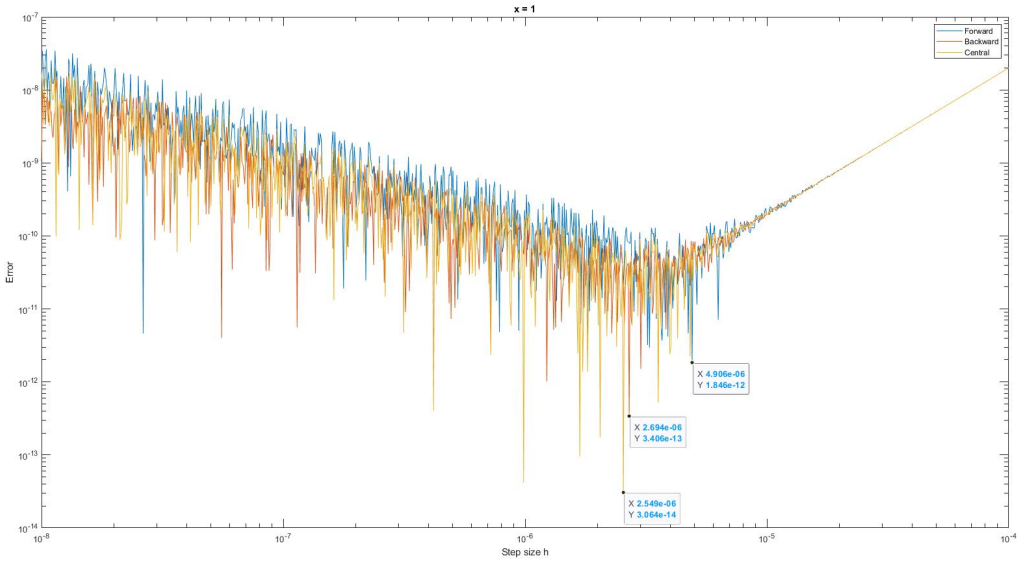
```
x = [0 0.1341 0.2693 0.4034 0.5386 0.6727 0.8079 0.9421 1.0767 1.2114 1.3460 1.4801 ...
1.6153 1.7495 1.8847 2.0199 2.1540 2.2886 2.4233 2.5579 2.6921 2.8273 2.9614 3.0966 ...
3.2307 3.3659 3.5000 ];
y = [0 0.0310 0.1588 0.3767 0.6452 0.8780 0.9719 1.0000 0.9918 0.9329 0.8198 0.7707 ...
0.8024 0.7674 0.6876 0.5937 0.5778 0.4755 0.3990 0.3733 0.2870 0.2156 0.2239 0.1314 ...
0.1180 0.0707 0.0259 ];

P = polyfit(x, y, 5);
x2 = linspace(0, 3.5, 1000);
y2 = polyval(P, x2);
plot(x, y, '-', x2, y2);
```

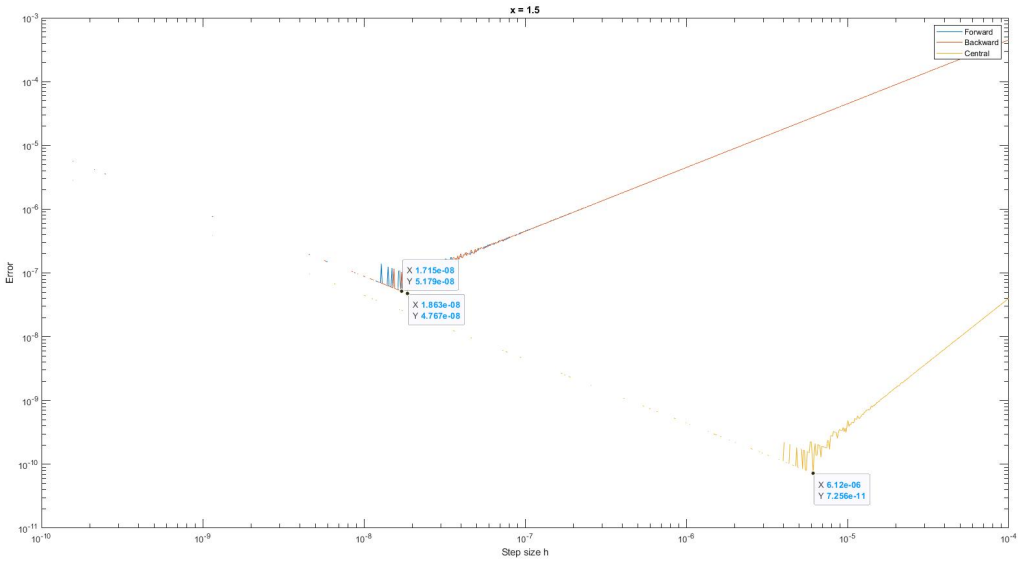
0.2 R1.2 Comparison Code

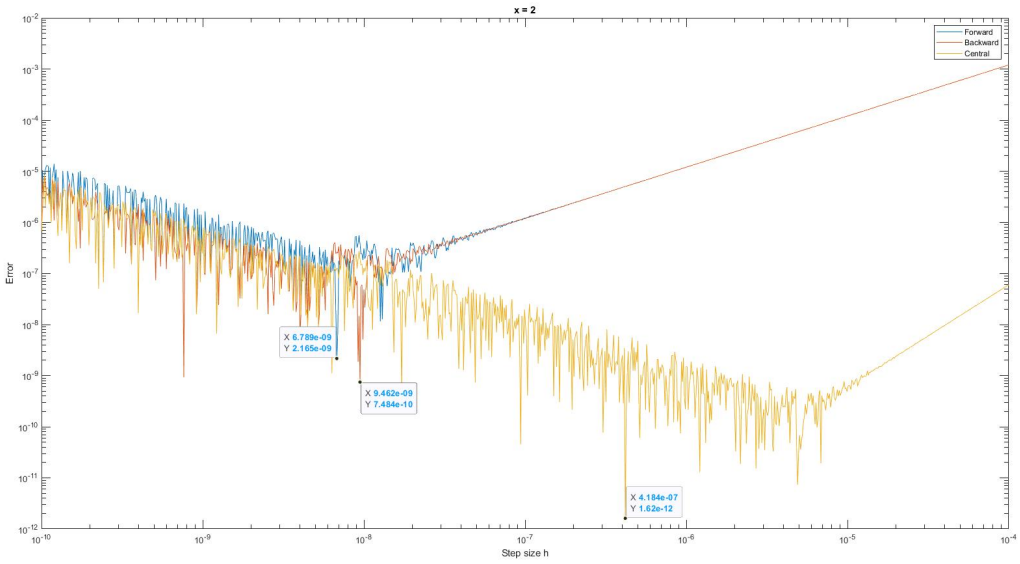
```
x = 2;
num_points = 1000;

h = logspace(-4, -10, num_points);

for n = 1:num_points
    forw = (f12(x+h(n))-f12(x)) / h(n);
    back = (f12(x)-f12(x-h(n))) / h(n);
    cent = (f12(x+h(n))-f12(x-h(n))) / (2*h(n));
    exac = 4*x^3 - 6*x^2;

    forw_err(n) = abs(forw - exac);
    back_err(n) = abs(back - exac);
    cent_err(n) = abs(cent - exac);
end

figure;
loglog(h, forw_err);
hold on
loglog(h, back_err);
loglog(h, cent_err);
title(['x = ' num2str(x)]);
ylabel('Error');
xlabel('Step size h');
legend("Forward", "Backward", "Central");

function y = f12(x)
y = x.^4 - 2*x.^3;
return
end
```

0.3 Comparisons of Error vs Step Size for $x = 0 \to 2$









0.4 Addition of Manual Error to Appendix 0.2 Code

```
function y = f12(x)
err = 0.1 * randn(size(x));
y = x.^4 - 2*x.^3 + err;
return
end
```

0.5 Comparison of Integration Methods Code

```
clear all
syms z

endpt = 10;

expMax = 23;                    %Exponent max
exponents = 1:expMax;           %vector of exponents
num_points = 2.^exponents+1;    %numPts stands for number of points

for n = 1:length(num_points)
    x = linspace(0, endpt, num_points(n));
    h(n) = x(2) - x(1);
    Y = sin(x.^2)+1;

    s_area = SIMP42(Y, h(n));
    t_area = trapz(x, Y);
    exact = 10.58367089992962334;

    s_err(n) = abs(s_area - exact);
    t_err(n) = abs(t_area - exact);
end

figure;
loglog(h, s_err);
hold on
loglog(h, t_err);
title('sin(x^2)+1, 0 -> 10');
ylabel('Error');
xlabel('Step size h');
legend("Simpson's Rule", "Trapezoid Rule", 'Location', 'southeast');

function Area = SIMP42(Y,h)
%Input:
%  Y = f(X), where f is the function to be integration;
%   X needs to be uniformly spaced vector with an odd number of points;
%   Please give Y as a row vector, or code will probably break...
%  h is the step-size;
[~,n] = size(Y);
if n == 1
    error('Code cannot do integration at a point. ')
elseif mod(n,2) == 0
    error('Y has an even number of points.\nPlease use an odd number of points.',1)
end

Y_mid = sum(4*Y(2:2:n-1));
Y_end = sum(2*Y(3:2:n-2));
Area = h/3*(Y_mid + Y_end + Y(1)+ Y(end));

end
```