1 Midpoint

This method of integration was fairly straightforward. I simply found the height of the function halfway between the two integration bounds. The resulting height is then multiplied by the length of the integration bounds. This rectangular area is a very approximate value for the area under the curve. Somewhat surprisingly only one rectangle was used for the approximation. Many times multiple midpoint rectangles are used to approximate an integral as this provides a more accurate value for complex integrals. Irregardless the use of a single rectangle made the code simpler. In fact the midpoint function has a constant run time, giving it a big O complexity of O(1).

2 Simpson's 1/3

This integration method is more complex than the midpoint version, but it is still quite familiar. Here a roughly quadratic function is used to approximate the area under a curve. The quadratic portion of Simpson's 1/3 rule is scaled by one sixth of the range of the integration bounds. Like the midpoint method it is common to use multiple quadratic approximations to get the area under a curve. Again like the midpoint method that only one quadratic approximation was used, providing a simpler code at the cost of accuracy. Simpson's 1/3 rule also runs at a constant time, giving it a complexity of O(1).

3 Simpson's 3/8

Simpson's 3/8 rule is similar to Simpson's 1/3 rule. The key difference is that a roughly cubic function is used to approximate the area under the curve and that the cubic approximation is scaled by one eighth the distance of the integration bounds this time. As above the implementation of Simpson's 3/8 for this lab sacrificed an accurate integral for the sake of simplicity. This does maintain a complexity of O(1), indicating constant run time.

4 Gauss Quadrature

I had not heard of the Gauss quadrature before this lab. It is an interesting process, figured out by Gauss, that the area under a function can be calculated by taking the values of the function at very specific points, t. The values at a point is then multiplied by some weight, w, also determined by Gauss. The resulting weighted values are then and added up as

$$Area = m * \sum_{i=1}^{n} w_i * f(c + m * t_i) \tag{1}$$

This will give an exact answer of the area under a curve. This works properly only on integration bounds from -1 to 1. Fortunately these can be scaled using the equations

$$c = \frac{a+b}{2} \tag{2}$$

and

$$m = \frac{b-a}{2} \tag{3}$$

which reset any arbitrary integration bounds to be compatible with the Gauss quadrature.

The other caveat is that the specific points the function is calculated at as well as the weights are dependent upon the order of the function being integrated. So to get a proper integration value the order of the Gauss quadrature must match the order of the function. This means that more (slightly different) points and weights are needed to evaluate the Gauss Quad for higher ordered functions. Since more terms needed to be added the higher the order of the function, the implementation of this code actually resulted in a loop. The reset integration bounds, c and m, were constant throughout the loop. Additionally, while the values of t and w did change for each iteration, their values were only read from an array, rather than written. Furthermore the use of a loop in this context results in a linear run time, giving a big O complexity of O(n).