**MATH 4500 Project #3** **Terry Phang**
Date: December 13, 2020

---

**Introduction to Numerical Integration Methods.** Here, we briefly introduce the numerical integration methods that will be used in **Problems 1** and **2**. The first method is the method of Gaussian Quadrature with $n = 5$ nodes, which is given by the equation

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\sum_{i=1}^{n} w_i f(x_i)$$

for some arbitary interval $[a, b]$. Here, $x_i = \frac{b-a}{2}t_i + \frac{a+b}{2}$, as we are performing a mapping of $[a, b]$ onto the reference interval $[-1, 1]$. The weights $w_i$ and the values $t_i$ for Gaussian Quadrature with $n = 5$ are given by tabulated values as follows:

$$w_i = [0.236927, 0.478629, 0.568889, 0.478629, 0.236927]$$

and

$$t_i = [-0.90618, -0.538469, 0, 0.538469, 0.90618]$$

where the $i$th element of each list corresponds to the $i$th $w$ or $t$ value starting from $i = 1$.

The second numerical integration method used is the Open and Closed Newton-Cotes Formulas, which are tabulated below for numerical integration of a function $f$ over $[a, b]$.

For $0 \le i \le n$, $x_i = a + ih$, $h = \frac{b-a}{n}$ and $f_i = f(x_i)$:

| $n$ | $h$ | Closed Newton-Cotes Formula | Error |
|---|---|---|---|
| 1 (Trapezoidal Rule) | $b-a$ | $\frac{h}{2}(f_0 + f_1)$ | $-\frac{h^3}{12}f''(\xi)$ |
| 2 (Simpson's Rule) | $\frac{b-a}{2}$ | $\frac{h}{3}(f_0 + 4f_1 + f_2)$ | $-\frac{h^5}{90}f^4(\xi)$ |
| 3 (Simpson's 3/8ths Rule) | $\frac{b-a}{3}$ | $\frac{3h}{8}(f + 0 + 3f_1 + 3f_2 + f_3)$ | $-\frac{3h^5}{80}f^4(\xi)$ |
| 4 (Boole's Rule) | $\frac{b-a}{4}$ | $\frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4)$ | $-\frac{8h^7}{945}f^6(\xi)$ |

For $0 \le i \le n$, $x_0 = a + h$, $x_i = x_0 + ih$, $h = \frac{b-a}{n+2}$, $f_i = f(x_i)$:

| $n$ | $h$ | Open Newton-Cotes Formula | Error |
|---|---|---|---|
| 0 (Midpoint Rule) | $\frac{b-a}{2}$ | $2hf_0$ | $-\frac{h^3}{3}f''(\xi)$ |
| 1 (No Name) | $\frac{b-a}{3}$ | $\frac{3h}{2}(f_0 + f_1)$ | $-\frac{3h^3}{4}f''(\xi)$ |
| 2 (No Name) | $\frac{b-a}{4}$ | $\frac{4h}{3}(2f_0 - f_1 + 2f_2)$ | $-\frac{28h^5}{90}f^4(\xi)$ |
| 3 (No Name) | $\frac{b-a}{5}$ | $\frac{5h}{24}(11f_0 + f_1 + f_2 + 11f_3)$ | $-\frac{95h^5}{144}f^4(\xi)$ |

While code was written to carry out each of these Open and Closed Newton-Cotes methods in the following sections, the higher order methods, i.e. Boole's Rule and Open Newton-Cotes formula with $n = 3$, were used to present the results in this report for the best accuracy.

Each of the above numerical integration methods can be adapted into a composite form by dividing the interval of integration $[a, b]$ into a specified number of subintervals, performing numerical integration over each subinterval, and summing the results. The composite methods have smaller associated errors, and thus will be implemented **Problems 1** and **2** for higher accuracy. The composite numerical integration methods and the code for such methods are discussed in detail in the following sections in the context of the problems.

**Problem 1.** In Problem 1, we are asked to determine the root mean square current $I_{RMS}$ as a function of the period for $0.0001 \leq T \leq 1$ s (analytically and numerically) given the following waveform:

$$I_{RMS} = \sqrt{\frac{1}{T}\int_0^T i^2(t)dt} \tag{1}$$

$$i(t) = \begin{cases} 10e^{-t/T}\sin(\frac{2\pi}{T}t), & \text{for} \quad 0 \leq t \leq \frac{T}{2}. \\ 0, & \text{for} \quad \frac{T}{2} \leq t \leq T. \end{cases} \tag{2}$$

**Analytical Solution:** Given Eqs. (1) and (2) we can focus on evaluating the following integral:

$$\int_0^T i^2(t)dt = \int_0^{\frac{T}{2}} 100e^{-\frac{2t}{T}}\sin^2\left(\frac{2\pi}{T}t\right)dt \tag{3}$$

since the integral picks up no contribution from the intergrand $i^2(t)$ over the interval $[\frac{T}{2}, T]$. Recalling the definitions

$$\sin(z) = \frac{e^{iz} - e^{-iz}}{2i}$$

and

$$\cos(z) = \frac{e^{iz} + e^{-iz}}{2}$$

we obtain the following after substitution and algebra on the integrand and the subsequent integration over $[0, \frac{T}{2}]$

$$\int_0^T i^2(t)dt = 100\int_0^{T/2} e^{-\frac{2t}{T}}\left(\frac{e^{\frac{2\pi i}{T}t} - e^{-\frac{2\pi i}{T}t}}{2i}\right)^2 dt \tag{4}$$

$$= -25\int_0^{\frac{T}{2}} e^{\frac{2}{T}(2\pi i-1)t} + e^{-\frac{2}{T}(2\pi i+1)t} - 2e^{-\frac{2t}{T}}dt \tag{5}$$

$$= -\frac{25}{2}\left[\frac{T}{2}\left(\frac{e^{\frac{2}{T}(2\pi i-1)t}}{2\pi i - 1}\right) - \frac{T}{2}\left(\frac{e^{-\frac{2}{T}(2\pi i+1)t}}{2\pi i + 1}\right) + Te^{-\frac{2t}{T}}\right]_0^{\frac{T}{2}} \tag{6}$$

$$= \frac{25T}{2}\left[\left(\frac{e^{\frac{2}{T}(1-2\pi i)t}}{1 - 2\pi i}\right) + \left(\frac{e^{-\frac{2}{T}(2\pi i+1)t}}{1 + 2\pi i}\right) - 2e^{-\frac{2t}{T}}\right]_0^{\frac{T}{2}} \tag{7}$$

$$= \frac{25T}{2}\left[\left(\frac{e^{\frac{2}{T}(1-2\pi i)t} - 1}{1 - 2\pi i}\right) + \left(\frac{e^{-\frac{2}{T}(2\pi i+1)t} - 1}{1 + 2\pi i}\right) - \frac{2}{e} + 2\right] \tag{8}$$

Multiplying the numerator and denominator of the first two fraction terms in the brackets by the complex conjugates of their respective denominators and simplifying using Euler's

formula, we obtain

$$= \frac{25T}{2e(1+4\pi^2)}\left[(e^{2\pi i}+e^{-2\pi i})+2\pi i(e^{2\pi i}-e^{-2\pi i})-2e-2(1+4\pi^2)+2e(1+4\pi^2)\right] \tag{9}$$

$$= \frac{25T}{2e(1+4\pi^2)}\left[2\cos(2\pi)-4\pi\sin(2\pi)-2e-2(1+4\pi^2)+2e(1+4\pi^2)\right] \tag{10}$$

$$= \frac{25T}{2e(1+4\pi^2)}\left[2-2+2e-2e+8\pi^2e-8\pi^2\right] \tag{11}$$

Simplifying, we obtain the value of the integral as a function of the period $T$:

$$\int_0^T i^2(t)dt = \left(\frac{100\pi^2(e-1)}{(1+4\pi^2)e}\right)T \tag{12}$$

Inserting into the equation for $I_{RMS}$ (Eq.(1)), we obtain the $I_{RMS}(T)$:

$$I_{RMS}(T) = \sqrt{\frac{100\pi^2(e-1)}{(1+4\pi^2)e}} = 3.92588945949... \tag{13}$$

which is constant in $T$.

**Numerical Solution:** Next, we seek to verify this result numerically, by implementing the integration method that yields the most accurate answer. We will use two distinct numerical integration methods: Gaussian Quadrature (GQ) and Newton-Cotes formulas. We expect the Composite Gaussian Quadrature (CGQ) method to yield the best results, since it is a higher order method and has a low error estimate. Code will be written using the Python programming language.

General Strategy: We begin by mapping out an algorithm for executing numerical composite integration. Note that our end goal is to plot $I_{RMS}$ as a function of $T$. For the following explanation, assume that some $T$ is given. Recall based on on Eq. (2) that the integrand is 0 on $[\frac{T}{2}, T]$, so the integral will not pick up a contribution from the integrand in that domain. Thus, we begin by splitting the interval $[0, \frac{T}{2}]$ into a desired number of subintervals of width $h$ as shown in **Fig. 1.1**. The larger the number of subintervals, the more accurate we expect the result to be.

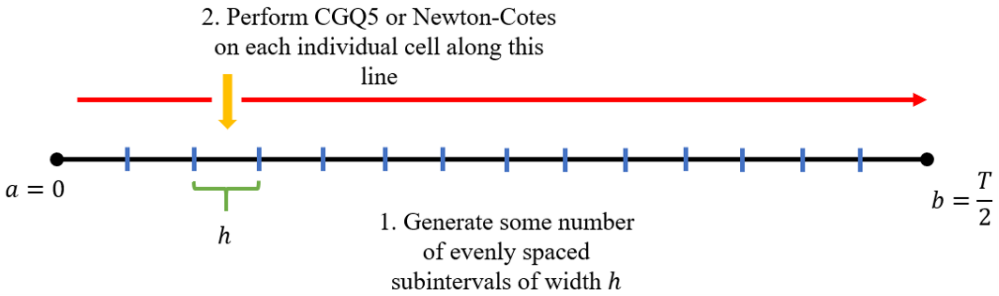


**Figure 1.1**:Visual scheme of composite numerical integration over the subinterval $[0, \frac{T}{2}]$.

We perform the composite numerical integration scheme with either Composite Gaussian Quadrature or or Newton-Cotes Formulas over each of these subintervals sequentially and

store the results in an array. Each of the elements in this array is summed to obtain the result of the numerical integral. Graphically, we are finding the areas under the curves shown in **Fig. 1.2** and plugging those values into Eq. (1) to obtain $I_{RMS}$.

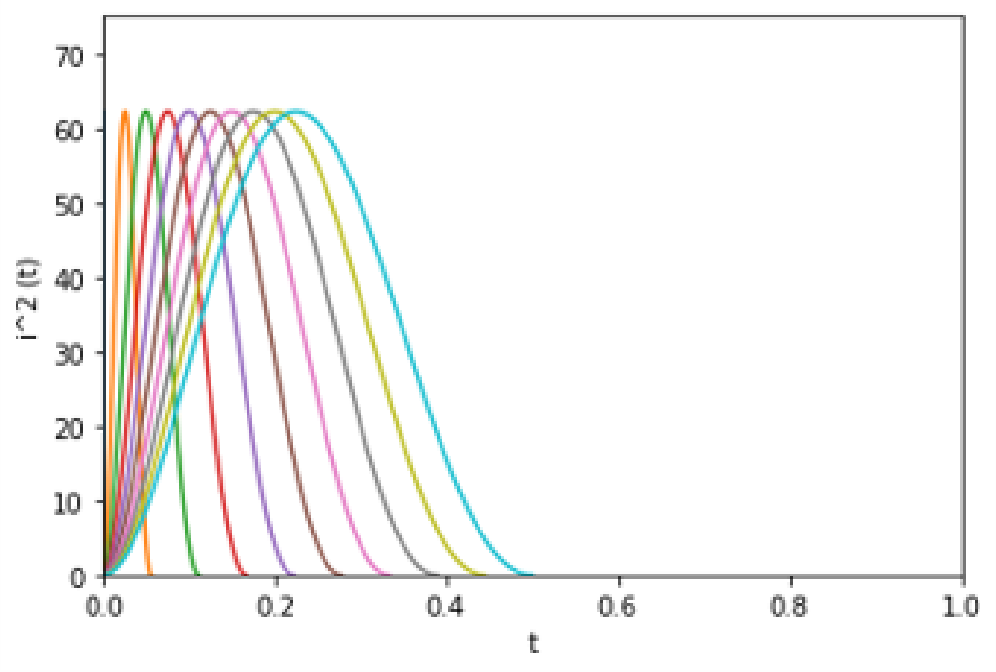


**Figure 1.2**: Example of $i^2(t)$ integrands for numerical integration

The absolute error is calculated using the analytical solution in the previous section. The described process is repeated for all $T \in [0.001, 1]$. The Composite Gaussian Quadrature of $n = 5$ (CGQ5) was chosen for the highest accuracy, and code was developed for all composite open (CONC) and closed (CCNC) Newton-Cotes formulas. The Python code for these methods is shown under **Code 1.1**, **Code 1.2**, and **Code 1.3**, respectively. In each section of code, the number of subintervals can be specified by setting "sInt" equal to a positive integer. A specific Newton-Cotes Formula can be selected by setting n = 1,2,3 or 4 for the Composite Closed Newton-Cotes Formulas and n = 0,1,2, or 3 for the Composite Open Newton-Cotes Formulas. To obtain the most accurate calculations, we choose higher order Newton-Cotes Formulas (i.e. Boole's Rule and ONC of n = 3) for the calculations.

```
# Method #1: Composite Gaussian Quadrature (n = 5) over each evenly spaced interval
I_RMS1 = []; # Empty RMS array. That is, each element is an integral over i2(t) from 0 to T/2 for all 0.001<T<1
w = [(322-13*np.sqrt(70))/900,
     (322+13*np.sqrt(70))/900,
     128/225,
     (322+13*np.sqrt(70))/900,
     (322-13*np.sqrt(70))/900]; #GQ weights
t = [-1/3*np.sqrt(5+2*np.sqrt(10/7)),
     -1/3*np.sqrt(5-2*np.sqrt(10/7)),
     0,
     1/3*np.sqrt(5-2*np.sqrt(10/7)),
     1/3*np.sqrt(5+2*np.sqrt(10/7))]; # GQ t values
x = [0]*5; # GQ x values
sInt = 100; # Number of subintervals for evaluating integral over 0 to T/2
for i in range(0,len(arrT)): # Loop over all 0.001<T<1
    storage = []; # Create empty storage
    a = 0; # Define lower limit of integral: a = 0
    b = arrT[i]/2; # Define upper limit of integral: b = T/2
    h = (b-a)/sInt; # Split this interval into "sInt" number of subintervals of equal width h (call them Sn's)
    a1 = 0; b1 = 0; # Initialize
    for j in range(0,sInt): # Loop over each Sn of the interval (0,T/2)
        if a1 < a+(sInt)*h and b1 < a+(sInt+1)*h: # Check to make sure we don't loop over Sn's outside the interval
            a1 = a+j*h; # The lower limit of the jth Sn
            b1 = a+(j+1)*h; # The upper limit of the jth Sn
        for k in range (0,5): # Find x values over the jth Sn
            x[k] = ((b1-a1)/2)*t[k]+(b1+a1)/2; # Generate x values over the jth Sn
        storage.append(((b1-a1)/2)*(w[0]*i2(x[0])
                                    +w[1]*i2(x[1])
                                    +w[2]*i2(x[2])
                                    +w[3]*i2(x[3])
                                    +w[4]*i2(x[4]))); # Append the GQ over jth Sn to "storage"
    IntCGQ = 0; # Initialize a value for integral by Composite Gaussian Quadrature (CGQ)
    for l in range(0,len(storage)): # Add all the values in "storage" to obtain i2(t) by Composite Gaussian Quadrature
        IntCGQ = IntCGQ+storage[l];
    I_RMS1.append(IRMS(arrT[i],IntCGQ)); # Calculate and append root mean square current value to I_RMS array for each T
print("CGQ5 = ", I_RMS1); # Print I_RMS array

errorCGQ = []; # Define an array for the errors;
for i in range(0,len(arrT)): # Calculate and append error of Composite Gaussian Quadrature on each subinterval
    errorCGQ.append(abs(I_RMS1[i]-asol[i]));
print("error = ", errorCGQ); # Print error
```

**Code 1.1**: Code for Composite Gaussian Quadrature of n = 5

```
# Method 2: Composite Closed Newton-Cotes Formulas
n = 4; # n = 1 --> Trapezoidal; n = 2 --> Simpson's; n = 3 --> Simpson's 3/8; n = 4 --> Boole's Rule
I_RMS2 = [];
sInt = 1;
for i in range(0,len(arrT)): # Loop over all 0.001<T<1
    storage = []; # Create empty storage
    a = 0; # Define lower limit of integral: a = 0
    b = arrT[i]/2; # Define upper limit of integral: b = T/2
    h = (b-a)/sInt; # Split this interval into "sInt" number of subintervals of equal width h (call them Sn's)
    a1 = 0; b1 = 0; # The first interval of width h
    for j in range(0,sInt): # Loop over each Sn of the interval (0,T/2)
        if a1 < a+(sInt)*h and b1 < a+(sInt+1)*h: # Check to make sure we don't loop over Sn's outside the interval
            a1 = a+j*h; # The lower limit of the jth Sn
            b1 = a+(j+1)*h; # The upper limit of the jth Sn
            h1 = (b1-a1)/n;
            if n == 1: # Trapezoidal Rule Implementation
                x0 = a1; x1 = x0+h1;
                f0 = i2(x0); f1 = i2(x1);
                Itr = (h1/2)*(f0+f1);
                storage.append(Itr);
            elif n == 2: # Simpson's Rule implementation
                x0 = a1; x1 = x0+h1; x2 = x0+2*h1;
                f0 = i2(x0); f1 = i2(x1); f2 = i2(x2);
                Isr = (h1/3)*(f0+4*f1+f2);
                storage.append(Isr);
            elif n == 3: # Simpson's 3/8 rule implementation
                x0 = a1; x1 = x0+h1; x2 = x0+2*h1; x3 = x0+3*h1;
                f0 = i2(x0); f1 = i2(x1); f2 = i2(x2); f3 = i2(x3);
                Ister = (3*h1/8)*(f0+3*f1+3*f2+f3);
                storage.append(Ister);
            else: # Boole's Rule implementation
                x0 = a1; x1 = x0+h1; x2 = x0+2*h1; x3 = x0+3*h1; x4 = x0+4*h1;
                f0 = i2(x0); f1=i2(x1); f2=i2(x2); f3=i2(x3); f4=i2(x4);
                Ibr = (2*h1/45)*(7*f0+32*f1+12*f2+32*f3+7*f4);
                storage.append(Ibr);
    IntCCNC = 0; # Initialize value for integral by composite closed Newton-Cotes formulas
    for l in range(0,len(storage)): # Add all the values in "storage" to obtain i2(t) by CCNC
        IntCCNC = IntCCNC+storage[l];
    I_RMS2.append(IRMS(arrT[i],IntCCNC)); # Calculate and append root mean square current value to I_RMS array for each T
print("CCNC"+str(n)+" = ", I_RMS2); # Print I_RMS array

errorCCNC = []; # Define an array for the errors;
for i in range(0,len(arrT)): # Calculate error of CCNC on each subinterval
    errorCCNC.append(abs(I_RMS2[i]-asol[i]));
print("error = ", errorCCNC); # Print Error
```

**Code 1.2**: Code for Composite Closed Newton-Cotes Formulas

```
# Method #3: Composite Open Newton-Cotes Formulas
n = 3; # n = 1 --> Trapezoidal; n = 2 --> Simpson's; n = 3 --> Simpson's 3/8; n = 4 --> Boole's Rule
I_RMS3 = [];
sInt = 100;
for i in range(0,len(arrT)): # Loop over all 0.001<T<1
    storage = []; # Create empty storage
    a = 0; # Define lower limit of integral: a = 0
    b = arrT[i]/2; # Define upper limit of integral: b = T/2
    h = (b-a)/sInt; # Split this interval into "sInt" number of subintervals of equal width h (call them Sn's)
    a1 = 0; b1 = 0; # The first interval of width h
    for j in range(0,sInt): # Loop over each Sn of the interval (0,T/2)
        if a1 < a+(sInt)*h and b1 < a+(sInt+1)*h: # Check to make sure we don't loop over Sn's outside the interval
            a1 = a+j*h; # The lower limit of the jth Sn
            b1 = a+(j+1)*h; # The upper limit of the jth Sn
            h1 = (b1-a1)/(n+2);
            if n == 0: # Midpoint Rule
                x0 = a1+h1;
                f0 = i2(x0);
                I0 = 2*h1*f0;
                storage.append(I0);
            elif n == 1: # Open Newton-Cotes n = 1
                x0 = a1+h1; x1 = x0+h1;
                f0 = i2(x0); f1 = i2(x1);
                I1 = (3*h1/2)*(f0+f1);
                storage.append(I1);
            elif n == 2:# Open Newton-Cotes n = 2
                x0 = a1+h1; x1 = x0+h1; x2 = x0+2*h1;
                f0 = i2(x0); f1 = i2(x1); f2 = i2(x2);
                I2 = (4*h1/3)*(2*f0-f1+2*f2);
                storage.append(I2);
            else: # Open Newton-Cotes n = 3
                x0 = a1+h1; x1 = x0+h1; x2 = x0+2*h1; x3 = x0+3*h1;
                f0 = i2(x0); f1=i2(x1); f2=i2(x2); f3=i2(x3);
                I3 = (5*h1/24)*(11*f0+f1+f2+11*f3);
                storage.append(I3);
    IntCONC = 0;
    for l in range(0,len(storage)):
        IntCONC = IntCONC+storage[l];
    I_RMS3.append(IRMS(arrT[i],IntCONC));
print("CONC"+str(n)+" = ", I_RMS3);

errorCONC = []; # Define an array for the errors;
for i in range(0,len(arrT)): # Calculate error of Composite Gaussian Quadrature on each subinterval
    errorCONC.append(abs(I_RMS3[i]-asol[i]));
print("error = ", errorCONC);
```

**Code 1.3**: Code for Composite Open Newton-Cotes Formulas

Results: Shown in **Fig. 1.3(a)** are the graphical comparisons of the results for regular Gaussian Quadrature and Newton-Cotes integration (obtained by setting sInt = 1) with their composite counterparts (obtained by using sInt = 100 subintervals) in **Fig. 1.3(b)**. In the figures, we have used 25 points in the code to span the domain $0.001 \leq T \leq 1$ for ease of visual appraisal.

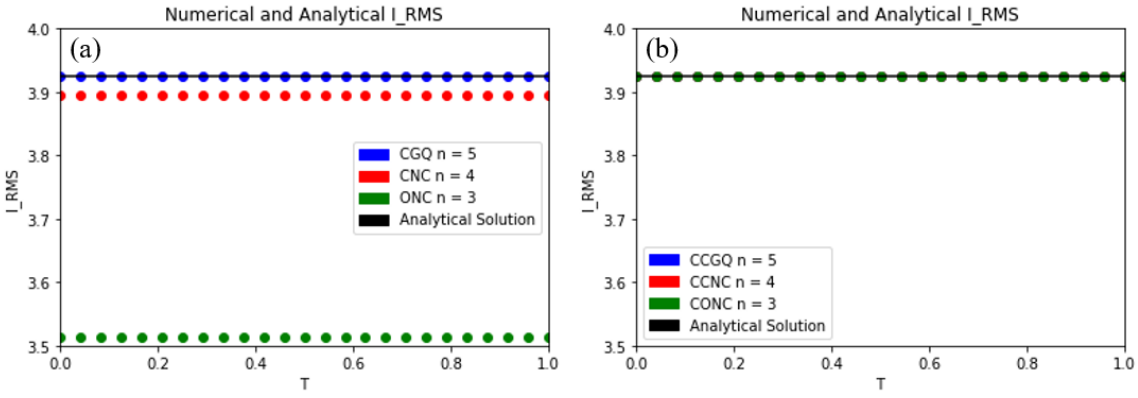


**Figure 1.3**: (a) Gaussian Quadrature of n = 5 (GQ5), Boole's Rule (CNC4), and Open Newton-Cotes Formula of n = 3 (ONC3). (b) Composite Gaussian Quadrature of n = 5 (CGQ5), Composite Boole's Rule (CCNC4), and Composite Open Newton-Cotes Formula of n = 3 (CONC3)

Visually, the larger amount of subintervals leads to a better result, as previously predicted. We further verify this by looking at the tabulated values of $I_{RMS}$ and their corresponding errors. The values for the regular (sInt = 1) Gaussian Quadrature and Newton-Cotes methods are shown in **Table 1.1**.

Numerical Solutions Gaussian Quadrature and Newton-Cotes Formulas

| T | GQ5 | error_GQ5 | CNC4 | error_CNC | ONC3 | error_ONC |
|---|---|---|---|---|---|---|
| 0.001000 | 3.925893415296340 | 3.955810544464811E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.042625 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256694 | 3.165973322910132E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.084250 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256694 | 3.165973322910176E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.125875 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.167500 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256694 | 3.165973322910132E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.209125 | 3.925893415296341 | 3.955810545352989E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.250750 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.292375 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.334000 | 3.925893415296340 | 3.955810544464811E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.375625 | 3.925893415296341 | 3.955810545352989E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.417250 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.458875 | 3.925893415296341 | 3.955810545352989E-06 | 3.894229726256694 | 3.165973322910132E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.500500 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.542125 | 3.925893415296341 | 3.955810545352989E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.583750 | 3.925893415296341 | 3.955810545352989E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.625375 | 3.925893415296340 | 3.955810544464811E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.667000 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256694 | 3.165973322910132E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.708625 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.750250 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446470E-01 |
| 0.791875 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.833500 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256694 | 3.165973322910132E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.875125 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.916750 | 3.925893415296340 | 3.955810544464811E-06 | 3.894229726256694 | 3.165973322910132E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 0.958375 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |
| 1.000000 | 3.925893415296340 | 3.955810544908900E-06 | 3.894229726256695 | 3.165973322910087E-02 | 3.512435466941148 | 4.134539925446474E-01 |

**Table 1.1**: Numerical results corresponding to **Fig. 1.2(a)** for Gaussian Quadrature of n = 5 (GQ5), Boole's Rule (CNC4), and Open Newton-Cotes Formula of n = 3 (ONC3)

The value of $I_{RMS}$ and the corresponding errors for the composite (sInt = 100) methods are shown in **Table 1.2**.

Numerical Solutions for Composite Gaussian Quadrature and Composite Newton-Cotes Formulas

| T | CGQ5 | error_CGQ5 | CCNC | error_CCNC | CONC | error_CONC |
|---|---|---|---|---|---|---|
| 0.001000 | 3.925889459485796 | 4.440892098500626E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033258254627526E-10 |
| 0.042625 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485788 | 7.105427357601002E-15 | 3.925889458982470 | 5.033258254627526E-10 |
| 0.084250 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485789 | 6.661338147750939E-15 | 3.925889458982469 | 5.033262695519625E-10 |
| 0.125875 | 3.925889459485796 | 8.881784197001252E-16 | 3.925889459485787 | 7.993605777301127E-15 | 3.925889458982471 | 5.033240491059132E-10 |
| 0.167500 | 3.925889459485796 | 4.440892098500626E-16 | 3.925889459485789 | 6.661338147750939E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.209125 | 3.925889459485795 | 4.440892098500626E-16 | 3.925889459485787 | 7.993605777301127E-15 | 3.925889458982469 | 5.033262695519625E-10 |
| 0.250750 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485789 | 6.661338147750939E-15 | 3.925889458982471 | 5.033244931951231E-10 |
| 0.292375 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982468 | 5.033271577303822E-10 |
| 0.334000 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485786 | 8.881784197001252E-15 | 3.925889458982470 | 5.033258254627526E-10 |
| 0.375625 | 3.925889459485796 | 4.440892098500626E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033249372843329E-10 |
| 0.417250 | 3.925889459485797 | 1.332267629550188E-15 | 3.925889459485787 | 7.993605777301127E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.458875 | 3.925889459485795 | 4.440892098500626E-16 | 3.925889459485788 | 7.105427357601002E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.500500 | 3.925889459485795 | 4.440892098500626E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.542125 | 3.925889459485795 | 4.440892098500626E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033249372843329E-10 |
| 0.583750 | 3.925889459485794 | 8.881784197001252E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033249372843329E-10 |
| 0.625375 | 3.925889459485797 | 1.332267629550188E-15 | 3.925889459485786 | 8.881784197001252E-15 | 3.925889458982470 | 5.033258254627526E-10 |
| 0.667000 | 3.925889459485794 | 8.881784197001252E-16 | 3.925889459485787 | 8.437694987151190E-15 | 3.925889458982470 | 5.033258254627526E-10 |
| 0.708625 | 3.925889459485796 | 4.440892098500626E-16 | 3.925889459485787 | 7.993605777301127E-15 | 3.925889458982470 | 5.033249372843329E-10 |
| 0.750250 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485787 | 7.993605777301127E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.791875 | 3.925889459485796 | 4.440892098500626E-16 | 3.925889459485787 | 8.437694987151190E-15 | 3.925889458982470 | 5.033249372843329E-10 |
| 0.833500 | 3.925889459485796 | 8.881784197001252E-16 | 3.925889459485788 | 7.105427357601002E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.875125 | 3.925889459485795 | 0.000000000000000E+00 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033258254627526E-10 |
| 0.916750 | 3.925889459485796 | 8.881784197001252E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 0.958375 | 3.925889459485796 | 8.881784197001252E-16 | 3.925889459485788 | 7.549516567451064E-15 | 3.925889458982470 | 5.033253813735428E-10 |
| 1.000000 | 3.925889459485796 | 4.440892098500626E-16 | 3.925889459485788 | 7.105427357601002E-15 | 3.925889458982470 | 5.033253813735428E-10 |

**Table 1.2**: Numerical results corresponding to **Fig. 1.2(b)** for Composite Gaussian Quadrature of n = 5 (CGQ5), Composite Boole's Rule (CCNC4), and Composite Open Newton-Cotes Formula of n = 3 (CONC3)

Based on the data taken at each $T$ in **Tables 1.1** and **1.2**, it is clear that the composite approximation methods yield better results by simply considering the absolute error. For example, regular Gaussian Quadrature (n = 5) and the Closed and Open Newton-Cotes Formulas have errors on the order of 10E-6, 10E-2, and 10E-1 respecitvely, whereas their composite counterparts have errors on the order of 10E-16, 10E-15, and 10E-10 respecitvely. Thus, using composite integration with 100 subintervals reduces the error by nearly ten orders of magnitude. Of the composite methods, CGQ5 is closest to the analytical solution. We arrive at the results for our numerical evaluation of $I_{RMS}$, which are summarized in **Table 1.3** below.

| | CGQ5 | CCNC4 | CONC3 |
|---|---|---|---|
| $I_{RMS}(T)$ | 3.925889459485796 | 3.925889459485788 | 3.925889458982470 |
| Error | 4.440892098500626E-16 | 7.549516567451064E-15 | 5.033258254627526E-10 |

**Table 1.3**: Results (on average) for each of the three composite numerical methods. The most accurate numerical result, given by CGQ5, is shaded in green.

**Problem 2.** In Problem 2, we are asked to integrate the area between two curves $f(x)$ and $g(x)$ that compose the cross section of an airfoil. We are given the data set shown in **Table 2.1** and the figure shown in **Fig. 2.1**.

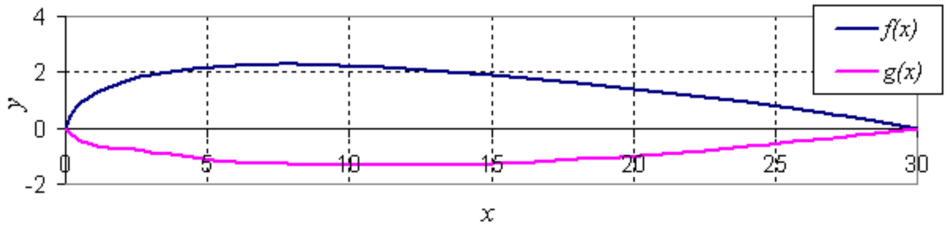


**Figure 2.1**: Profile of the airfoil cross section formed by $f(x)$ and $g(x)$.

| $x$ | $f(x)$ | $g(x)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0.375 | 0.801 | -0.369 |
| 0.75 | 1.083 | -0.513 |
| 1.5 | 1.473 | -0.678 |
| 2.25 | 1.74 | -0.783 |
| 3 | 1.929 | -0.876 |
| 4.5 | 2.157 | -1.05 |
| 6 | 2.25 | -1.191 |
| 7.5 | 2.28 | -1.284 |
| 9 | 2.265 | -1.338 |
| 12 | 2.142 | -1.344 |
| 15 | 1.923 | -1.251 |
| 18 | 1.641 | -1.101 |
| 21 | 1.308 | -0.9 |
| 24 | 0.924 | -0.648 |
| 27 | 0.504 | -0.369 |
| 28.5 | 0.276 | -0.21 |
| 30 | 0 | 0 |

**Table 2.1**: Given data sets for curves $f(x)$ and $g(x)$.

**Numerical Solution:** We can implement the composite methods used in **Problem 1** to calculate the integral between these curves. In particular, CGQ5 was proven to be a powerful method for numerical integration in **Problem 1**, so we once again resort to this method and check the answer with the Composite Trapezoidal Rule (CCNC1). However, both of these methods require a function for evaluation at the nodes. Thus, we must select an interpolation method to generate such a function between two adjacent $x$ data points. Here, after testing the Lagrange, Newton, Hermite, and Spline interpolation methods, we select the Natural Cubic Spline as the method of choice as it fits closely to **Fig. 2.1**. Furthermore, we are integrating between two curves, so it is convenient to work with a single function defined by $\text{diff}(x) := f(x) - g(x)$ rather than integrating $f(x)$ and $g(x)$ separately and taking the difference of the results.

General Strategy: As in **Problem 1**, we begin by developing an algorithm for executing the natural cubic spline interpolation (hereafter referred to as spline interpolation) and the subsequent CGQ5 or CCNC1. A visual scheme for such an algorithm is shown in **Fig. 2.2**. It is important to note that the subintervals on the full interval $[0, 30]$ given by the provided data set are not evenly spaced. Thus, spline interpolation should be performed in the space between each consecutive data value for $x$.

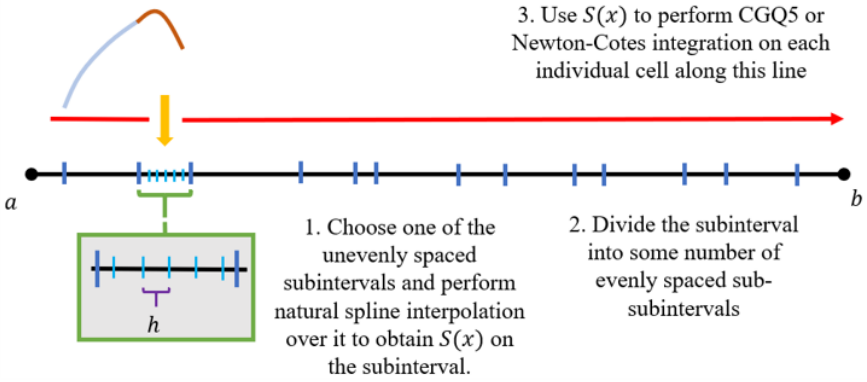


**Figure 2.2**: Given data sets for curves $f(x)$ and $g(x)$.

The algorithm begins by selecting the first two adjacent $x$ data values from **Table 2.1**, which, in this case, are 0 and 0.375. The interval bounded by these points $[0, 0.375]$ is a subinterval of the full interval $[0, 30]$. We divide $[0, 0.375]$ into some number of evenly spaced subintervals specified by the user with width $h$. Spline interpolation is performed over the interval $[0, 0.375]$ to obtain a function $S_0(x)$ on that interval using the spline interpolation code shown in **CODE 2.1**.

```
# MATH 4500 - Final Project: Problem 2
# Terry Phang

from scipy import *
from matplotlib import pyplot as plt
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.patches as mpatches
import pandas as pd
import site

# Define data sets
x = [0,0.375,0.75,1.5,2.25,3,4.5,6,7.5,9,12,15,18,21,24,27,28.5,30];
f = [0,0.881,1.083,1.473,1.74,1.929,2.157,2.25,2.28,
     2.265,2.142,1.923,1.641,1.308,0.924,0.504,0.276,0];
g = [0,-0.369,-0.513,-0.678,-0.783,-0.876,-1.05,-1.191,
     -1.284,-1.338,-1.344,-1.251,-1.101,-0.9,-0.648,-0.369,-0.21,0];
diff = [0]*len(x);
for i in range(0,len(x)):
    diff[i] = f[i]-g[i];

# Perform numerical integration using Spline interpolation

# Initialize and load data arrays
dataX = x;
dataY = diff;

# Initialize n and h array
n = len(dataX)-1;
h = [0]*n;

# Load h array
for j in range(0,n):
    h[j] = dataX[j+1]-dataX[j];

# Initialize and load A matrix
rows, cols = (n+1, n+1) ;
A = [[0 for k in range(cols)] for l in range(rows)];

A[0][0] = 1;
A[n][n] = 1;

for j in range (1,n):
    A[j][j-1] = h[j-1];
    A[j][j+1] = h[j];
    A[j][j] = 2*(h[j-1]+h[j]);

print("A = ", A);

# Initialize and load b matrix
vecb = [0]*(n+1);
for j in range(1,n):
    vecb[j] = 3/h[j]*(dataY[j+1]-dataY[j]) - 3/h[j-1]*(dataY[j]-dataY[j-1]);
vecb = np.transpose(vecb);

# Solve the linear relationship Ax = b
c = np.float_(np.linalg.solve(A, vecb));
```

```
# Obtain a,b,d
a = dataY;
b = [0]*(n+1);
d = [0]*(n+1);
for j in range (0,n):
    b[j] = (1/h[j])*(a[j+1]-a[j]) - (h[j]/3)*(2*c[j]+c[j+1]);
    d[j] = (c[j+1] - c[j])/(3*h[j]);

# Remove last entries of c, transpose a b,c, and d
a = np.delete(a,n);
b = np.delete(b,n);
c = np.delete(c,n);
d = np.delete(d,n);

#print a,b,c,d
print("a = ", a);
print("b = ", b);
print("c = ", c);
print("d = ", d);

MyOB = open('SplineFunctions.txt','w');
MyOB.write(\
    'Natural Cubic Spline Interpolation Functions'+'\n'
    +'Subinterval'+'                    '+'Spline'
    '\n')

# Print Spline Interpolation Function over all intervals
for i in range(0,n):
    xk = dataX[i]
    acoef = a[i];
    bcoef = b[i];
    ccoef = c[i];
    dcoef = d[i];
    print("S_"+str(i)+"(x) = "+str(acoef)+" + "+str(bcoef)+"(x-"+str(xk)+") + "
          +str(ccoef)+"(x-"+str(xk)+")^2 + "+str(dcoef)+"(x-"+str(xk)+")^3");
    MyOB.write('\n'
               + '['
               + '%0.3f' % dataX[i]
               + ','
               + '%0.3f' % dataX[i+1]
               +']'
               + '                    S_'+str(i)+'(x) = '
               +str(acoef)+' + '+str(bcoef)+'(x-'+str(xk)+') + '
               +str(ccoef)+'(x-'+str(xk)+')^2 + '+str(dcoef)+'(x-'+str(xk)+')^3''\n')
```

**Code 2.1**: Spline Interpolation code

The algorithm uses $S_0(x)$ as the function for performing numerical integration by CGQ5 or CCNC1 over the interval $[0, 0.375]$, stores the result in an array, and proceeds to the next adjacent $x$ values, i.e. $[0.375, 0.75]$. The process is repeated for the remaining spline interpolation functions $S_1(x)...S_{16}(x)$ and integrates until the entire interval $[0, 30]$ has been covered. Each of the 17 Spline Interpolation functions are shown in **Table 2.2**. The results for numerical integration over each of the "cells" (given by the first column of **Table 2.2**), which were previously stored in an array, are summed to obtain the final value of the integral over $[0, 30]$.

Natural Cubic Spline Interpolation Functions

| Subinterval | Spline |
|---|---|
| [0.000,0.375] | S_0(x) = 0.0 + 3.623570628851962(x-0) + 0.0(x-0)^2 + -3.580946940583983(x-0)^3 |
| [0.375,0.750] | S_1(x) = 1.17 + 2.112858742296075(x-0.375) + -4.028565030815698(x-0.375)^2 + 3.796289025847552(x-0.375)^3 |
| [0.750,1.500] | S_2(x) = 1.596 + 0.6929944019637376(x-0.75) + 0.24226012326279792(x-0.75)^2 + -0.23944799006370815(x-0.75)^3 |
| [1.500,2.250] | S_3(x) = 2.1510000000000002 + 0.652316103625427(x-1.5) + -0.2964978543805454(x-1.5)^2 + 0.11743517717330113(x-1.5)^3 |
| [2.250,3.000] | S_4(x) = 2.523 + 0.4057411835345545(x-2.25) + -0.03226870574061787 4(x-2.25)^2 + -0.009848274185050894(x-2.25)^3 |
| [3.000,4.500] | S_5(x) = 2.805 + 0.34071916223635434(x-3) + -0.05442732265698238 5(x-3)^2 + 0.003965254110719558(x-3)^3 |
| [4.500,6.000] | S_6(x) = 3.207 + 0.2042026595127642(x-4.5) + -0.03658367915874437(x-4.5)^2 + 0.0029657152112677007(x-4.5)^3 |
| [6.000,7.500] | S_7(x) = 3.441 + 0.11447019971258808(x-6) + -0.02323796070803972(x-6)^2 + 0.0010607739330985076(x-6)^3 |
| [7.500,9.000] | S_8(x) = 3.564 + 0.05191654163688385(x-7.5) + -0.018464478009096435(x-7.5)^2 + 0.0007911890563381783(x-7.5)^3 |
| [9.000,12.000] | S_9(x) = 3.603 + 0.0018636337398772462(x-9) + -0.014904127255574632(x-9)^2 + 0.0004276386696496113(x-9)^3 |
| [12.000,15.000] | S_10(x) = 3.4859999999999998 + -0.07601488571303103(x-12) + -0.01105537922872813(x-12)^2 + 0.0005756692665794987(x-12)^3 |
| [15.000,18.000] | S_11(x) = 3.174 + -0.12680409088775335(x-15) + -0.005874355829512642(x-15)^2 + 4.746204181014509e-05(x-15)^3 |
| [18.000,21.000] | S_12(x) = 2.742 + -0.16076875073595528(x-18) + -0.005447197453221336(x-18)^2 + -9.885076715340384e-05(x-18)^3 |
| [21.000,24.000] | S_13(x) = 2.208 + -0.1961209061684252(x-21) + -0.006336854357601971(x-21)^2 + 0.0003479410268034539 3(x-21)^3 |
| [24.000,27.000] | S_14(x) = 1.572 + -0.22474762459034378(x-24) + -0.0032053851163708854(x-24)^2 + 0.0001515311043840462 2(x-24)^3 |
| [27.000,28.500] | S_15(x) = 0.873 + -0.23988859547019986(x-27) + -0.0018416051769144695(x-27)^2 + -0.006821776339745 98(x-27)^3 |
| [28.500,30.000] | S_16(x) = 0.486 + -0.29146040129422857(x-28.5) + -0.03253959870577138(x-28.5)^2 + 0.007231021934615863(x-28.5)^3 |

**Table 2.2**: Natural Cubic Spline Interpolation functions generated over $[0, 30]$

The code for executing CGQ5 and CCNC1 using the Spline Interpolation functions as described above is shown in **CODE 2.2**.

```
# Method 1: Use Composite Gaussian Quadrature with n = 5 to perform integration over sub-subintervals
# Obtain integral between two adjacent data points
# Sum all integrals over full interval [0,30]
w = [(322-13*np.sqrt(70))/900,
     (322+13*np.sqrt(70))/900,
     128/225,
     (322+13*np.sqrt(70))/900,
     (322-13*np.sqrt(70))/900]; #GQ weights
t = [-1/3*np.sqrt(5+2*np.sqrt(10/7)),
     -1/3*np.sqrt(5-2*np.sqrt(10/7)),
     0,
     1/3*np.sqrt(5-2*np.sqrt(10/7)),
     1/3*np.sqrt(5+2*np.sqrt(10/7))]; # GQ t values
xCGQ = [0]*5; # GQ x values
sInt = 100; # Number of subintervals for evaluating integral over x_i to x_{i+1}
storage2 = []; # Create empty storage for integral over [0,30]
for i in range(0,len(dataX)-1): # Loop over all dataX
    storage = []; # Create empty storage
    a1 = dataX[i]; # Define lower limit of integral: a1 = x_i
    b1 = dataX[i+1]; # Define upper limit of integral: b1 = x_{i+1}
    h = (b1-a1)/sInt; # Split this subinterval into "sInt" number of sub-subintervals of equal width (call them Sn's)
    a2 = 0; b2 = 0; # Initialize
    xarr = np.linspace(dataX[i],dataX[i+1],10000);
    def y(x):
        return a[i] + b[i]*(x-dataX[i]) + c[i]*(x-dataX[i])**2 + d[i]*(x-dataX[i])**3;
    plt.plot(xarr,y(xarr));
    plt.xlabel('x');
    plt.ylabel('f(x)-g(x)');
    plt.title('Spline Interpolation of f(x)-g(x)');
    for j in range(0,sInt): # Loop over all sub-subintervals
        if a2 < a1+(sInt)*h and b2 < a1+(sInt+1)*h: # Check to make sure we don't loop over Sn's outside the interval
            a2 = a1+j*h; # The lower limit of the jth Sn
            b2 = a1+(j+1)*h; # The upper limit of the jth Sn
        for k in range (0,5): # Find x values over the jth Sn
            xCGQ[k] = ((b2-a2)/2)*t[k]+(b2+a2)/2; # Generate x values over the jth Sn;
        storage.append(((b2-a2)/2)*(w[0]*y(xCGQ[0])
                                    +w[1]*y(xCGQ[1])
                                    +w[2]*y(xCGQ[2])
                                    +w[3]*y(xCGQ[3])
                                    +w[4]*y(xCGQ[4]))); # Append the GQ over jth Sn to "storage"
    IntCGQ = 0; # Initialize a value for integral by CGQ on a sub-subinterval
    for l in range(0,len(storage)): # Add values in "storage" to obtain integral by CGQ5 on a subinterval
        IntCGQ = IntCGQ+storage[l];
    storage2.append(IntCGQ); # Calculate and append values to an array for all subintervals
TotInt1 = 0; # Initialize
for m in range(0,len(storage2)): # Add all values in this array to obtain total integral over [0,30]
        TotInt1 = TotInt1+storage2[m];
print("CGQ5 = ", TotInt1); # Print array
```

**Code 2.1**: Composite Gaussian Quadrature using Spline Interpolation Functions

```
# Method 2: Use Closed Newton-Cotes formulas to perform integration over sub-subintervals
# Obtain integral between two adjacent data points
# Sum all integrals over full interval [0,30]
storage2 = []; # Define empty storage for integral over [0,30]
n = 100; # n = 1 --> Trapezoidal; n = 2 --> Simpson's; n = 3 --> Simpson's 3/8; n = 4 --> Boole's Rule
sInt = 1; # Number of subintervals for evaluating integral over x_i to x_{i+1}
for i in range(0,len(dataX)-1): # Loop over all 0.001<T<1
    storage = []; # Create empty storage
    a1 = dataX[i]; # Define lower limit of integral: a = 0
    b1 = dataX[i+1]; # Define upper limit of integral: b = T/2
    h = (b1-a1)/sInt; # Split this interval into "sInt" number of subintervals of equal width h (call them Sn's)
    a2 = 0; b2 = 0; # The first interval of width h
    xarr = np.linspace(dataX[i],dataX[i+1],10000);
    def y(x):
        return a[i] + b[i]*(x-dataX[i]) + c[i]*(x-dataX[i])**2 + d[i]*(x-dataX[i])**3;
    for j in range(0,sInt): # Loop over each Sn of the interval (0,T/2)
        if a1 < a1+(sInt)*h and b1 < a1+(sInt+1)*h: # Check to make sure we don't loop over Sn's outside the interval
            a2 = a1+j*h; # The lower limit of the jth Sn
            b2 = a1+(j+1)*h; # The upper limit of the jth Sn
            h = (b2-a2)/n;
            if n == 1: # Trapezoidal Rule Implementation
                x0 = a2; x1 = x0+h;
                f0 = y(x0); f1 = y(x1);
                Itr = (h/2)*(f0+f1);
                storage.append(Itr);
            elif n == 2: # Simpson's Rule implementation
                x0 = a2; x1 = x0+h; x2 = x0+2*h;
                f0 = y(x0); f1 = y(x1); f2 = y(x2);
                Isr = (h/3)*(f0+4*f1+f2);
                storage.append(Isr);
            elif n == 3: # Simpson's 3/8 rule implementation
                x0 = a2; x1 = x0+h; x2 = x0+2*h; x3 = x0+3*h;
                f0 = y(x0); f1 = y(x1); f2 = y(x2); f3 = y(x3);
                Ister = (3*h/8)*(f0+3*f1+3*f2+f3);
                storage.append(Ister);
            else: # Boole's Rule implementation
                x0 = a2; x1 = x0+h; x2 = x0+2*h; x3 = x0+3*h; x4 = x0+4*h;
                f0 = y(x0); f1=y(x1); f2=y(x2); f3=y(x3); f4=y(x4);
                Ibr = (2*h/45)*(7*f0+32*f1+12*f2+32*f3+7*f4);
                storage.append(Ibr);
    IntCCNC = 0; # Initialize a value for integral by CCNC on a subinterval
    for l in range(0,len(storage)): # Add values in "storage" to obtain integral by CCNC on a subinterval
        IntCCNC = IntCCNC+storage[l];
    storage2.append(IntCCNC); # Calculate and append values to an array for all subintervals
TotInt2 = 0; # Initialize
for m in range(0,len(storage2)): # Add all values in this array to obtain total integral over [0,30]
    TotInt2 = TotInt2+storage2[m];
print("CCNC"+str(n)+" = ", TotInt2); # Print array
```

**Code 2.2**: Trapezoidal Rule using Spline Interpolation Functions

Results: To demonstrate the efficacy of the spline interpolation algorithm, we execute the code on the data set for $f(x)$ and $g(x)$, expecting to reproduce a likeness of their corresponding curves in **Fig. 2.1**. Indeed, as shown in **Fig. 2.3**, the piecewise curve segments

reproduce these curves.

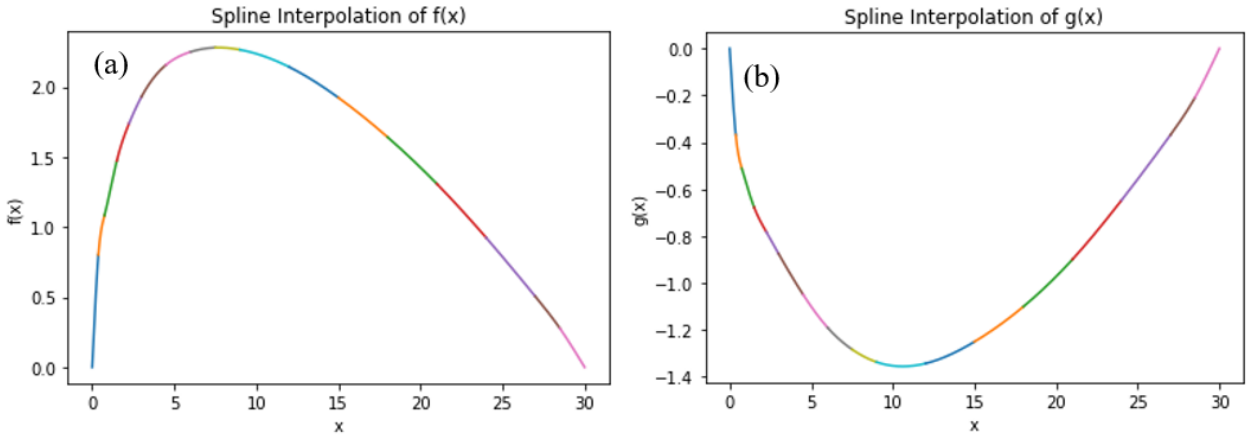


**Fig2.3**: Spline interpolation of (a) $f(x)$ and (b) $g(x)$ over $[0, 30]$. Each color represents a segment of the total spline interpolation function $S(x)$

Next, we perform integration using CGQ5 and CCNC1 over the function defined by $\text{diff}(x) := f(x) - g(x)$. The spline interpolation function for $\text{diff}(x)$ is shown in **Fig. 2.4**, where we have chosen the number of nodes used for the interpolation to be 100. This graph closely resembles the shape of $f(x)$, but has a larger height. This is to be expected, since $g(x)$ is negative on $[0, 30]$ and it is being subtracted from $f(x)$.

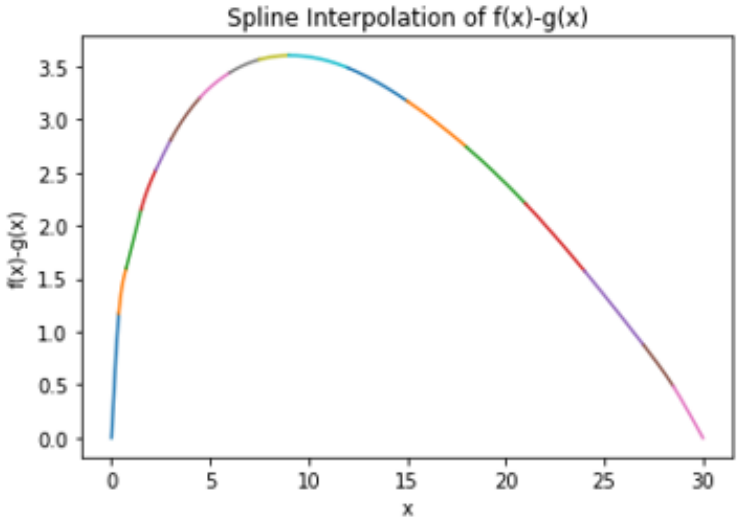


**Fig 2.4**: Spline interpolation of $\text{diff}(x) := f(x) - g(x)$ over $[0, 30]$. Each color represents a segment of the total spline interpolation function $S(x)$

The results of the integration are shown in **Table 2.3**. Evidently, the results agree well, differing by only 10E-5.

| | CGQ5 | CCNC1 | Difference |
|---|---|---|---|
| $\int_0^{30} f(x) - g(x)dx$ | 73.97615165124971 | 73.97612019858363 | 3.1452666E−5 |

**Table 2.3**: Results of numerical integration over $[0, 30]$

While the CGQ5 yielded good results, the CGQ of n = 3 or 4 would have worked just as well in comparing the result to the Trapezoidal rule based on the order of their error formulas.

**Conclusion:** It is important to consider the ways in which the code in both problems can be improved to yield even more accurate results. Consider, for example, the CGQ5 results in **Table 1.2** from **Problem 1**. While the values of the numerical integral appear to be constant for all $T$, as they should be, closer inspection reveals that they are only constant through 16 significant digits, after which small discrepancies are introduced. One plausible explanation for this error is the fact that the number of subintervals used to evaluate each integral is the same, regardless of the value of $T$, i.e. the same number of subintervals is being used to perform numerical integration over $[0, 0.001]$ as the interval $[0, 1]$. In principle, this can be fixed by finding a way to dynamically change the number of subintervals used as $T$ increases. However, this is difficult to accomplish without overloading Python and causing the program to run for extended periods of time. In the case of the Newton-Cotes formulas, the numerical integration can also be improved by simply increasing the number of subintervals used to, say, 1,000 or 10,000 based on their respective error formulas, which depend on $h$, the width of a subinterval. This is also true for **Problem 2**, which uses code adapted from **Problem 1**.

I faced one significant challenge in developing the CGQ5 code for this project. While I followed the correct thought process (outlined in the "General Strategy" section of Problem 1), my method of incrementing between subintervals was initially incorrect. Namely, the "if" statement I was using to terminate the loop after performing CGQ5 over each cell on the interval $[a, b]$ failed . This was because rather than checking that I had arrived at the last cell with reference to $a$, I was checking this with reference to $b$, resulting in the failure of the conditional statement at the final iteration and thus an overestimated numerical integral. I fixed this by setting the "if" statement to terminate the loop when the position of the lower limit of a cell $a1$ was given by $a1 = a + (sInt) * h$ and the position of the upper limit of the same cell $b1$ was given by $b1 = a + (sInt + 1) * h$, where $sInt$ is the number of subintervals of width $h$ that were used in the composite integration.

**Outlook:** In reflecting on my semester in Numerical Analysis I, I am happy to say that I have achieved my goal for the class: to practice my skills with coding and learn numerical approximation methods which can be applied to real life examples. As a Physics major, the skills I have learned in Numerical Analysis I will be useful for understanding and making accurate predictions based on experimental data sets. My skills with Python coding have also significantly improved since the beginning of the semester, thanks to the many opportunities I had to practice. Going forward, I expect that Numerical Analysis II will introduce the

concept of solving Partial Differential Equations (PDE's). I find this topic particularly interesting, as they can be used to model many physical systems.