Gabriel Rios
ME I4600
March 10, 2022

Homework 2

1. **Consider the following data, which are obtained from a smooth function also known as Runge's function,**

$$y = \frac{1}{1+25x^2} \tag{1}$$

x_i	-1.00	-0.80	-0.60	-0.40	-0.20	0.00	0.20	0.40	0.60	0.80	1.00
y_i	0.038	0.058	0.100	0.200	0.500	1.00	0.500	0.200	0.100	0.058	0.038

The Lagrange polynomial interpolation, for which the value at any point x is simply:

$$P(x) = \sum_{j=0}^{n} y_j \prod_{i=0}^{n} \frac{x - x_i}{x_j - x_i} \tag{2}$$

Write a computer program for Lagrange interpolation. Test your program by verifying that $P(0.7) = -0.226$.

(a) **Using the above data in the table, find the interpolated value at $x = 0.9$.** (10 pts)

The interpolated value at $x = 0.9$ is ∼1.579. This value is significantly higher than the actual value (∼0.047), which shows the inability of this interpolation to properly capture the behavior of the function near the endpoints of the range. See Figure 1 for the superposition of this point on the interpolation plots from part (b). The program used to determine this interpolation is included in the Appendix.

(b) **Use Runge's function to generate a table of 21 equally spaced data points. Interpolate these data using a Lagrange polynomial of order 20. Plot this polynomial and comment on the comparison between your result and the plot of part (a).** (10 pts)

See Figure 1 for plots of the plotted data from part (a) (N = 10), the interpolation of order 20 (N = 20), and the actual function plot. As mentioned in part (a), the interpolation is relatively accurate around $x = 0$ but becomes more variable towards the endpoints, improvement for the interpolation scheme is needed, which can be performed by methods such as adding data points near the domain edges and adding points beyond the domain.

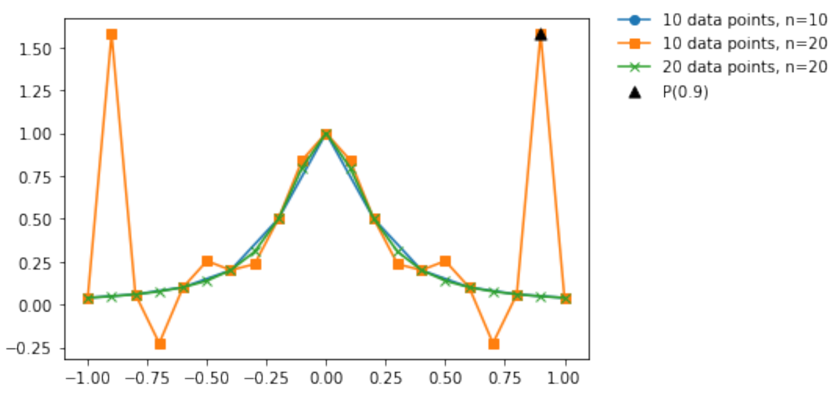


Figure 1: Data from the provided table (blue, circular markers), an interpolation of this data of order 20 (orange, square markers), and actual outputs from the function at 20 data points (green, crossed markers).

2. **The concentration of a certain toxin in a system of lakes downwind of an industrial area has been monitored very accurately at intervals from 1993 to 2007 as shown in the table below. It is believed that the concentration has varied smoothly between these data points:**

Year	Toxin Concentration
1993	12.0
1995	12.7
1997	13.0
1999	15.2
2001	18.2
2003	19.8
2005	24.1
2007	28.1
2009	???

(a) **Interpolate the data with the Lagrange polynomial** (5 pts). **Plot the polynomial and the data points** (5 pts). **Use the polynomial to predict the condition of the lakes in 2009** (5 pts). **Discuss this prediction** (5 pts).

The program used to determine this interpolation is included in the Appendix. See Figure 2 for the interpolation of the complete dataset using the Lagrange polynomial method. The polynomial interpolation predicts a toxin concentration of -38.4 in 2009. This prediction seems illogical due to the monotonous increase in toxin concentration with time, such that a concentration greater than that of 2007 is expected in 2009. This illogical prediction is likely a result of the polynomial variability near the limit of the domain, as was observed in Problem 1. Therefore, this interpolation cannot be used for prediction outside of the interpolation domain (i.e., extrapolation).

(b) **Interpolation may also be used to fill 'holes' in the date. Say the data from 1997 and 1999 disappeared. Predict these values using the Lagrange polynomial fitted through the other known data points** (10 pts).

The interpolated values for 1997 and 1999 are 16.23 and 17.88, respectively. The interpolation for this data can be seen in Figure 2. This interpolation using the dataset with missing years overpredicts the concentration values and predicts a more oscillatory increase in concentration with time than the actual data.

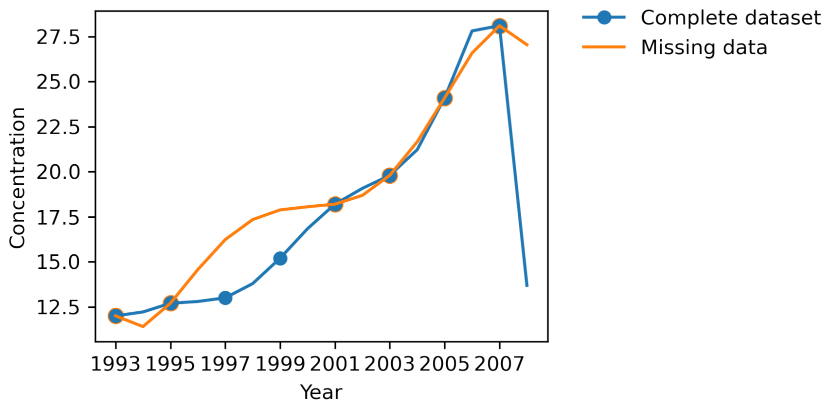


Figure 2: Interpolation from the given data (blue, circular markers) and the interpolation of this data when concentrations from 1997 and 1999 are excluded (orange).

3. **An example of a peaky function is the Lorentz profile, $1/\left(1+\frac{x^2}{a^2}\right)$. For large a, it is well behaved but for small a, it is strongly peaked near the origin. For $a = 0.1$ and 1:**

   (a) **Plot the function from 0 to 1.** (5 pts)

   See Figure 3 for the plot of the function at both values of a.

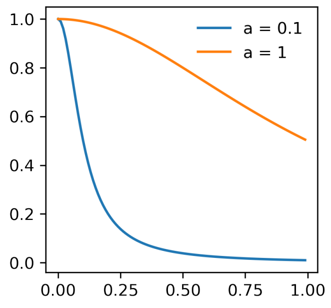
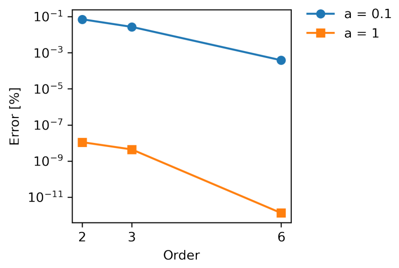


Figure 3: Lorentz profile for different values of a from 0 to 1.

   (b) **Integrate this function from 0 to 1 using Newton-Cotes methods of various orders ($n = 2, 3, 6$) with a fixed number of intervals ($N = 12$).** (10 pts)

   See the table below for results of this integration method and the figure for errors relative to the exact integration for the functions over the given domain. Results truncated after 6 decimal places for clarity. The program used to determine this interpolation is included in the Appendix.

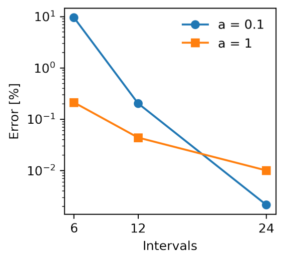
a	n	Result
0.1	2	0.147009
0.1	3	0.147074
0.1	6	0.147112
1	2	0.785398
1	3	0.785398
1	6	0.785398



(c) **Integrate this function from 0 to 1 using trapezoidal rule ($n = 1$) with a variable number of intervals ($N = 6, 12, 24$).** (10 pts)

See the table below for results of this integration method and the figure for errors relative to the exact integration for the functions over the given domain. Results truncated after 6 decimal places for clarity. The program used to determine this interpolation is included in the Appendix.

a	N	Result
0.1	6	0.161237
0.1	12	0.147413
0.1	24	0.147110
1	6	0.783732
1	12	0.785054
1	24	0.785319



(d) **Plot the error versus the order, n, for (b) and versus the number of intervals for (c).** (10 pts)

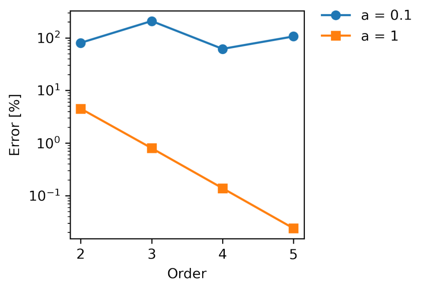
See parts (b) and (c) for the respective error plots. For both parts (b) and (c), a decrease in error is associated with an increase in order and number of intervals, as expected. The Newton-Cotes method produces a significantly more accurate estimate than the trapezoidal method, rendering it the preferable method for the integration of this function. With regards to part (b), the error for the Lorentz profile at $a = 1$ is orders of magnitude lower than that for $a = 0.1$, which can be attributed to the less variable function values when $a = 0.1$. This makes less of a difference in part (b), where $a = 1$ even features lower error at $N = 24$.

4. **Integrate the function of Problem 3 using Gauss quadrature of various numbers of Gauss points ($n \leq 5$) with single interval** (20 pts). **Plot the error versus the number of Gauss points.** (15 pts)

See the table below for results of this integration method and the figure for errors relative to the exact integration for the functions over the given domain. Results truncated after 6 decimal places for clarity. The program used to determine this interpolation is included in the Appendix.

Increasing the order of quadrature for the Lorentz profiles had differing results for each value of a. For $a = 1$, an increasing order of n correlated with a decrease in error, but for $a = 0.1$, there was no noticeable correlation between n and error. The minimum error for $a = 1$ was below 1%, indicating that this is likely an acceptable integration method at the corresponding order ($n = 5$). However, the minimum error for $a = 0.1$ hovered around 100%, indicating that Gauss quadrature with a single interval is not an appropriate method for this profile. This is likely due to the sharp gradient in values for the $a = 0.1$ Lorentz profile, whereas $a = 1$ is more linear. Reference calculations on quadrature calculators showed that convergence to acceptable error ($\sim$1%) occurred at $n \geq 40$), which is impractical for most applications. Therefore, a remedy to this may consist of adding subintervals to speed convergence.

a	n	Result
0.1	2	0.029126
0.1	3	0.453552
0.1	4	0.056556
0.1	5	0.303252
1	2	0.750000
1	3	0.791667
1	4	0.784314
1	4	0.785586



Appendix

Homework 2 - ME I4600, Computational Fluid Mechanics, Spring 2022, City College of New York

General imports

In [554…

```python
import matplotlib.pyplot as plt
import numpy as np
```

Problem 1

In [555…

```python
def runge(x):
    '''
    Runge's function.

    Input(s):
        x: float
    Output(s):
        y: float
    '''
    return 1/(1 + 25*x**2)

def lpi(x, arr_x, arr_y, n=None):
    '''
    Lagrange polynomial interpolation.

    Input(s):
        x: (float) the point on the x-axis at which interpolation is desired
        arr_x: (list or NumPy array) array of points on the x-axis
        arr_y: (list or NumPly array) array of functions based on points from the x-axis
        n: (int, optional) order of interpolation
    Outputs(s):
        p: (float) interpolated point that is a function of 'x'
    '''

    # If no order was specified, set order to be equal to umber of points
    if not n:
        n = len(arr_x)

    # Intialize interpolation
    p = 0
    # Iterate through products
    for j in range(0, n):
        # Initialize inner sum
        s = 1
        # Iterate through sums
        for i in range(0, n):
            # i =/= j condition
            if i != j:
                s = s * (x - arr_x[i])/(arr_x[j] - arr_x[i])
        p += arr_y[j]*s

    return p
```

Problem 1a: Using the above data in the table, find the interpolated value at x=0.9. (10 pts)

In [558…

```python
# Define points along the x-axis
step_x = 0.2
arr_x_a = np.arange(-1, 1+step_x, step_x)
# Define functions of the arr_x points
```

```
arr_y_a = np.array([runge(x) for x in arr_x_a])

# Define point at which interpolation is desired
x = 0.9
# Calculate the interpolation
p_a = lpi(x, arr_x_a, arr_y_a)
```

Problem 1b: Use Runge's function to generate a table of 21 equally spaced data points. Interpolate these data using a Lagrange polynomial of order 20. Plot this polynomial and comment on the comparison between your result and the plot of part (a). (10 pts)

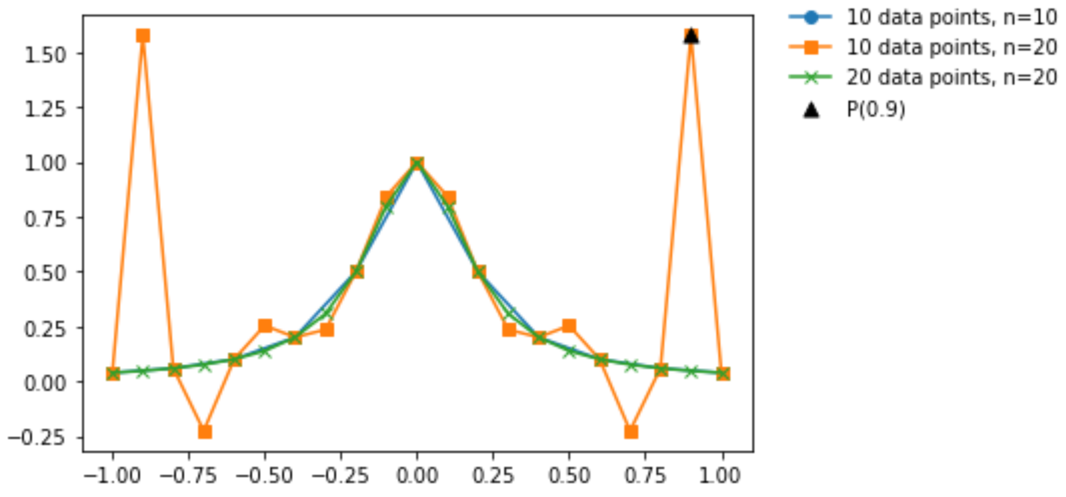
In [569…

```
# Define points along the x-axis
arr_x_b = np.linspace(-1, 1, 21)
# Define functions of the arr_x points
arr_y_b = np.array([runge(x) for x in arr_x_b])

# Define point at which interpolation is desired
x = 0.9
# Calculate the interpolation
p_b_10 = [lpi(i, arr_x_a, arr_y_a) for i in arr_x_b]
p_b_20 = [lpi(i, arr_x_b, arr_y_b) for i in arr_x_b]
print(arr_y_b[-2])
```

```
0.047058823529411 75
```

In [570…

```
fig, ax = plt.subplots()
ax.plot(arr_x_a, arr_y_a, marker='o', label='10 data points, n=10')
ax.plot(arr_x_b, p_b_10, marker='s', label='10 data points, n=20')
ax.plot(arr_x_b, p_b_20, marker='x', label='20 data points, n=20')
ax.scatter(x, p_a, marker='^', c='k', s=50, zorder=10, label='P(0.9)')
fig.legend(loc='upper right', bbox_to_anchor=(1.275, 0.925), frameon=False);
```



Problem 2

Problem 2a: Interpolate the data with the Lagrange polynomial (5 pts). Plot the polynomial and the data points (5 pts). Use the polynomial to predict the condition of the lakes in 2009 (5 pts). Discuss this prediction (5 pts).

In [581…

```
# List of years
interval = 2
years = np.arange(1993, 2009, interval)
# Toxin concentrations
conc = np.array([12, 12.7, 13, 15.2, 18.2, 19.8, 24.1, 28.1])
```

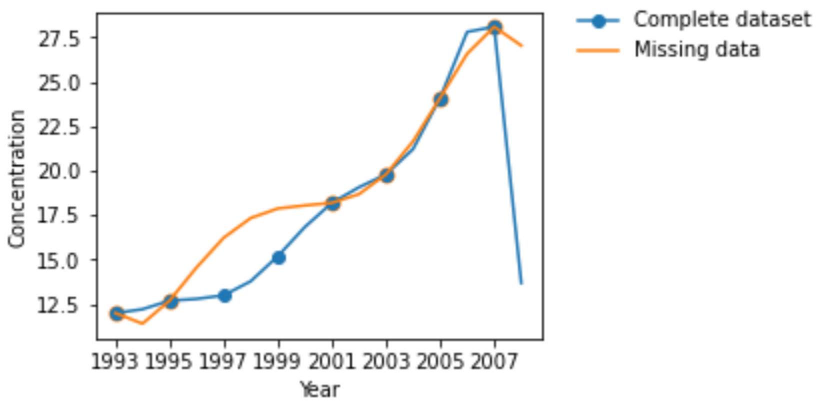
```
# Interpolate concentrations annually
years_annual = np.arange(1993, 2009, interval/2)
conc_p = [lpi(year, years, conc) for year in years_annual]

# Predict concentration in 2009
conc_p_2009 = lpi(2009, years, conc)
print('Toxin concentration prediction for 2009: {0:.2f}'.format(conc_p_2009))
```

```
Toxin concentration prediction for 2009: -38.40
```

Problem 2b: Interpolation may also be used to fill “holes” in the date. Say the data from 1997 and 1999 disappeared. Predict these values using the Lagrange polynomial fitted through the other known data points (10 pts).

In [584…

```
# Boolean mask
mask = np.where((years != 1997) & (years != 1999))
# List of years
years_missing = years[mask]
# Toxin concentrations
conc_missing = conc[mask]

# Interpolate concentrations annually
years_annual = np.arange(1993, 2009, interval/2)
conc_p_hole = [lpi(year, years_missing, conc_missing) for year in years_annual]

# Plot the concentrations
fig, ax = plt.subplots(figsize=(4, 3))
ax.plot(years_annual, conc_p, marker='o', markevery=interval, label='Complete dataset')
ax.scatter(years_missing, conc_missing, s=50, color='tab:orange')
ax.plot(years_annual, conc_p_hole, color='tab:orange', label='Missing data')
ax.set_xlabel('Year')
ax.set_xticks(years)
ax.set_ylabel('Concentration')
fig.legend(loc='upper right', bbox_to_anchor=(1.4, 0.925), frameon=False);
```



In [576…

```
for i in range(0, len(conc_p_hole)):
    print(years_annual[i], conc_p_hole[i])
```

```
1993.0 12.0
1994.0 11.406171874999998
1995.0 12.7
1996.0 14.569140624999996
1997.0 16.23333333333333
1998.0 17.343359375
1999.0 17.880000000000003
2000.0 18.052994791666666
2001.0 18.2
2002.0 18.685546875000004
2003.0 19.8
```

```
2004.0 21.658515625000003
2005.0 24.1
2006.0 26.586067708333335
2007.0 28.1
2008.0 27.045703125000003
```

Problem 3

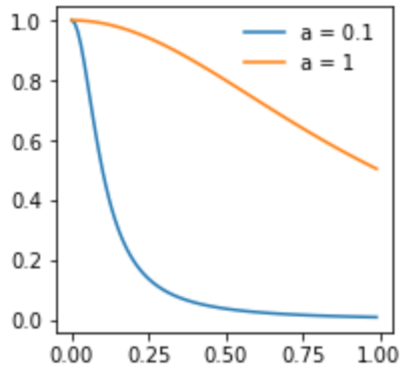
In [578…

```
def lorentz(x, a):
    '''
    Lorentz profile.

    Input(s):
        x: float
        a: float
    Output(s):
        y: float
    '''
    return 1/(1 + x**2/(a**2))
```

Problem 3a: For a = 0.1 and 1, (a) plot the function from 0 to 1 (5 pts).

In [580…

```
arr_x = np.arange(0, 1, 0.01)
y1 = [lorentz(x, 0.1) for x in arr_x]
y2 = [lorentz(x, 1) for x in arr_x]

fig, ax = plt.subplots(figsize=(3, 3))
ax.plot(arr_x, y1, label='a = 0.1')
ax.plot(arr_x, y2, label='a = 1')
ax.legend(frameon=False);
```



Problem 3b: Integrate this function from 0 to 1 using Newton-Cotes methods of various orders (n = 2, 3, and 6) with fixed number of intervals (number of intervals = 12) (10 pts).

In [588…

```
# Define Newton-Cotes coefficients.
def ncc(n):
    '''
    Newton-Cotes coefficients generator.

    Input(s):
        n: int, order of Newton-Cotes coefficients
    Output(s):
        coeffs: array, array of floats with Newton-Cotes coefficients

    '''

    # Define array of values to set up coefficient matrix
    arr = np.arange(0, n+1, 1)
    # Define column vector to process matrix with
    B = [(n**(i+1))/(i+1) for i in range(0, n+1)]
    # Generate Vandermonde matrix
    M = np.vander(arr, max(arr)+1, increasing=True)
    # Generate coefficients
    coeffs = np.dot(np.linalg.inv(M.T), B)/n

    return coeffs
```

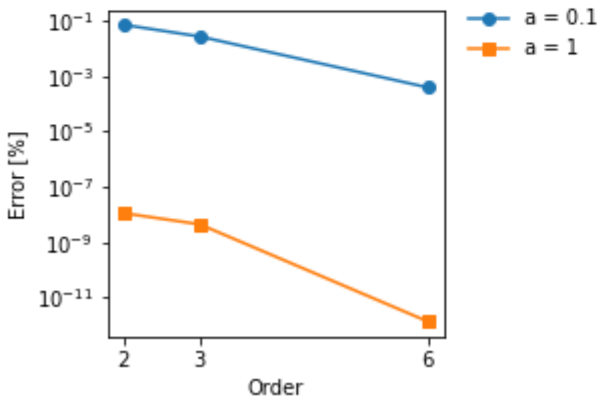
In [586…

```
def ncc_integration(n, lower, upper, a):

    # Define lower and upper bounds
    arr = np.linspace(lower, upper, n+1)
    # Generate Newton-Cotes coefficients
    coeffs = ncc(n)
    # Initialize integration
    s = 0
    # Perform integration
    s = (upper - lower)*np.nansum([coeffs[k]*lorentz(arr[k], a) for k in range(0, n+1)])

    return s
```

In [590…

```
# Define number of intervals
N = 12
# Define x-values
arr_x = np.linspace(0, 1, N)
# Initialize list of integrations and errors
ints, errors = [], []
# Define actual value (from Wolfram)
actual = [0.147113, np.pi/4]
# Define orders of interest
ns = [2, 3, 6]
# Perform integrations
for j, a in enumerate([0.1, 1]):
    # Initialize list to capture coefficient-specific errors
    error = []
    for n in ns:
        s = np.nansum([ncc_integration(n, arr_x[i], arr_x[i+1], a) for i in range(0, len(a
        print('When a is {0}, the integral of function with order {1} is: {2:.6f}'.format
        ints.append(s)
        # Catalogue error
        error.append(100*np.abs((s - actual[j])/actual[j]))
    # Append to master error list
    errors.append(error)

# Plotting
fig, ax = plt.subplots(figsize=(3,3))
ax.plot(ns, errors[0], marker='o', label='a = 0.1')
ax.plot(ns, errors[1], marker='s', label='a = 1')
ax.set_yscale('log')
ax.set_xticks(ns)
ax.set_xlabel('Order')
ax.set_ylabel('Error [%]')
fig.legend(loc='upper right', bbox_to_anchor=(1.3, 0.925), frameon=False);
```

```
When a is 0.1, the integral of function with order 2 is: 0.147009
When a is 0.1, the integral of function with order 3 is: 0.147074
When a is 0.1, the integral of function with order 6 is: 0.147112
When a is 1, the integral of function with order 2 is: 0.785398
When a is 1, the integral of function with order 3 is: 0.785398
When a is 1, the integral of function with order 6 is: 0.785398
```

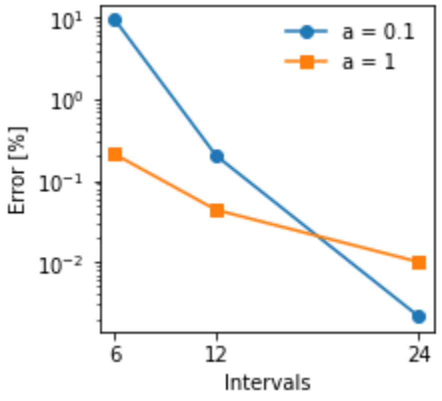


Problem 3c: Integrate this function from 0 to 1 using trapezoidal rule (n = 1) with variable number of intervals (number of intervals = 6, 12, and 24) (10 pts).

In [592…

```
# Define order
n = 1
# Define Lorentz coefficients
Ns = [6, 12, 24]
# Intialize list of actuals and errors
actual, errors = [0.147113, np.pi/4], []
# Perform integrations
for j, a in enumerate([0.1, 1]):
    error = []
    for N in Ns:
        arr_x = np.linspace(0, 1, N)
        s = np.nansum([ncc_integration(n, arr_x[i], arr_x[i+1], a) for i in range(0, len(a
        print('Integral of function with {0} intervals is: {1:.6f}'.format(N, s))
        error.append(100*np.abs((s - actual[j])/actual[j]))
    errors.append(error)

# Plotting
fig, ax = plt.subplots(figsize=(3, 3))
ax.plot(Ns, errors[0], marker='o', label='a = 0.1')
ax.plot(Ns, errors[1], marker='s', label='a = 1')
ax.set_yscale('log')
ax.set_xticks(Ns)
ax.set_xlabel('Intervals')
ax.set_ylabel('Error [%]')
ax.legend(frameon=False);
```

```
Integral of function with 6 intervals is: 0.161237
Integral of function with 12 intervals is: 0.147413
Integral of function with 24 intervals is: 0.147110
Integral of function with 6 intervals is: 0.783732
Integral of function with 12 intervals is: 0.785054
Integral of function with 24 intervals is: 0.785319
```



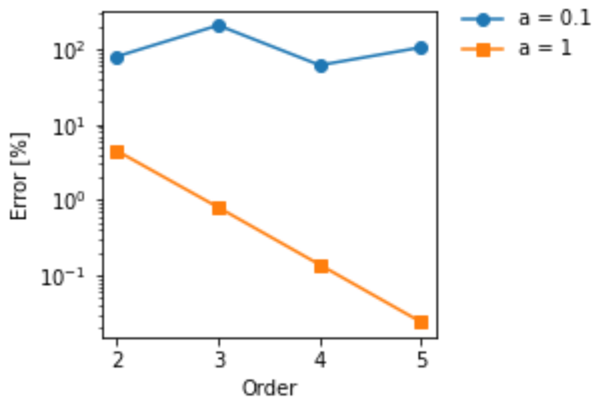
Problem 4: Integrate the function of Problem 3 using Gauss quadrature of various numbers of Gauss points

(n ≤ 5) with single interval (20 pts). Plot the error versus the number of Gauss points. (15 pts)

In [594…
```
def gq(n, a):
    '''
    Gauss quadrature for use with the Lorentz profile.

    Input(s):
        n: int
        a: int, for Lorentz profile
    Output(s):
        s: float
    '''

    # Catalogue abscissas and weights for orders (n = 2 to n = 5)
    if n == 2:
        arr_x = [-1, -np.sqrt(3)/3, np.sqrt(3)/3, 1]
        w = [1, 1, 1, 1]
    elif n == 3:
        arr_x = [-1, -np.sqrt(15)/5, 0, np.sqrt(15)/5, 1]
        w = [1, 5/9, 8/9, 5/9, 1]
    elif n == 4:
        arr_x = [-1,
                 -np.sqrt(525 + 70*np.sqrt(30))/35,
                 -np.sqrt(525 - 70*np.sqrt(30))/35,
                 np.sqrt(525 - 70*np.sqrt(30))/35,
                 np.sqrt(525 + 70*np.sqrt(30))/35,
                 1]
        w = [1,
             (18-np.sqrt(30))/36,
             (18+np.sqrt(30))/36,
             (18+np.sqrt(30))/36,
             (18-np.sqrt(30))/36, 1]
    elif n == 5:
        arr_x = [-1,
                 -np.sqrt(245 + 14*np.sqrt(70))/21,
                 -np.sqrt(245 - 14*np.sqrt(70))/21,
                 0,
                 np.sqrt(245 - 14*np.sqrt(70))/21,
                 np.sqrt(245 + 14*np.sqrt(70))/21,
                 1]
        w = [1,
             (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,
             1]
    else:
        return None

    # Get quadrature
    s = np.nansum([w[k]*lorentz(arr_x[k], a) for k in range(1, n+1)])

    return s
```

In [596…
```
# Initialize quadrature orders
ns = [2, 3, 4, 5]
# Intialize list of actuals and errors
actual, errors = [0.147113, np.pi/4], []

# Iterate over each value of a
for j, a in enumerate([0.1, 1]):
    error = []
    for n in ns:
        value = gq(n, a)/2
        print('Gauss quadrature for Lorentz profile with coefficient {0} and order {1} is:
        error.append(100*np.abs(actual[j] - value)/actual[j])
    errors.append(error)

fig, ax = plt.subplots(figsize=(3, 3))
ax.plot(ns, errors[0], marker='o', label='a = 0.1')
ax.plot(ns, errors[1], marker='s', label='a = 1')
ax.set_xticks(ns)
ax.set_xlabel('Order')
ax.set_ylabel('Error [%]')
ax.set_yscale('log')
fig.legend(loc='upper right', bbox_to_anchor=(1.3, 0.925), frameon=False);
```

```
Gauss quadrature for Lorentz profile with coefficient 0.1 and order 2 is: 0.029126
Gauss quadrature for Lorentz profile with coefficient 0.1 and order 3 is: 0.453552
Gauss quadrature for Lorentz profile with coefficient 0.1 and order 4 is: 0.056556
Gauss quadrature for Lorentz profile with coefficient 0.1 and order 5 is: 0.303252
Gauss quadrature for Lorentz profile with coefficient 1 and order 2 is: 0.750000
Gauss quadrature for Lorentz profile with coefficient 1 and order 3 is: 0.791667
Gauss quadrature for Lorentz profile with coefficient 1 and order 4 is: 0.784314
Gauss quadrature for Lorentz profile with coefficient 1 and order 5 is: 0.785586
```



In [ ]: