Fourier Transforms

June 6, 2024

1 CSC225: COMPUTATIONAL METHODS

1.1 TOPIC 1: FOURIER TRANSFORM

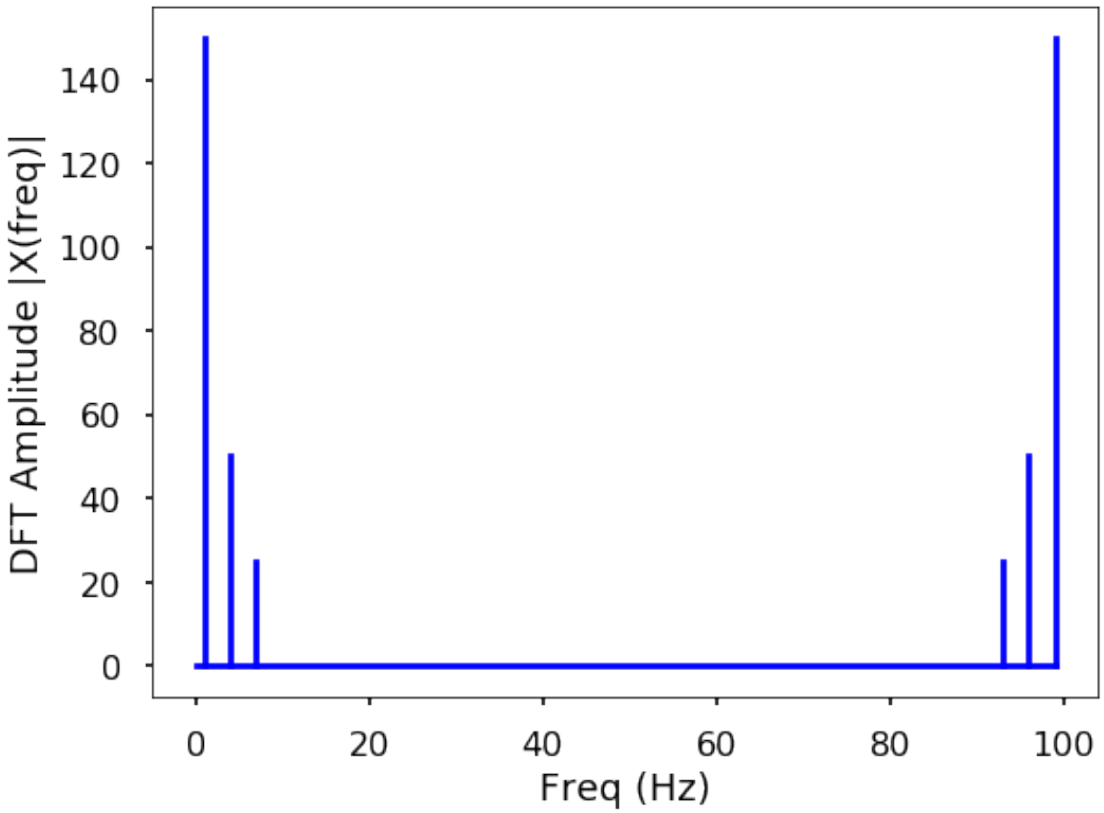
1.2 Let start with a question: Generate three sine waves with frequencies 1, 4, and 7 Hz, amplitudes 3, 1, and 0.5, and phases being all zeros. Add these three sine waves together with a sampling rate of 100 Hz.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
plt.style.use("seaborn-poster")
%matplotlib inline
```

In [2]:
```
 #sampling rate
sr = 100.0
# sampling interval
ts = 1.0/sr
t = np.arange(0,1,ts)
```

In [3]:
```
sr = 100.0
ts = 1.0/sr
t = np.arange(0,1,ts)
freq = 1.
x = 3*np.sin(2*np.pi*freq*t)
freq = 4
x += np.sin(2*np.pi*freq*t)
freq = 7
x += 0.5* np.sin(2*np.pi*freq*t)
def DFT(x):
    N = len(x)
    n = np.arange(N)
    k = n.reshape(N, 1)
    e = np.exp(-2j * np.pi * k * n / N)
    X = np.dot(e, x)
    return X
X = DFT(x)
N = len(X)
n = np.arange(N)
```

```
T = N/sr
freq = n/T
plt.figure(figsize = (8, 6))
plt.stem(freq, abs(X), "b", markerfmt=" ", basefmt="-b")
plt.xlabel("Freq (Hz)")
plt.ylabel("DFT Amplitude |X(freq)|")
plt.show()
```



In [4]:
```
freq = 1.
x = 3*np.sin(2*np.pi*freq*t)
freq = 4
x += np.sin(2*np.pi*freq*t)
freq = 7
x += 0.5* np.sin(2*np.pi*freq*t)
```

In [5]:
```
print(" the length of x in tinme domain is:", len(x))
print("")
print(x)
```

```
 the length of x in tinme domain is: 100
```

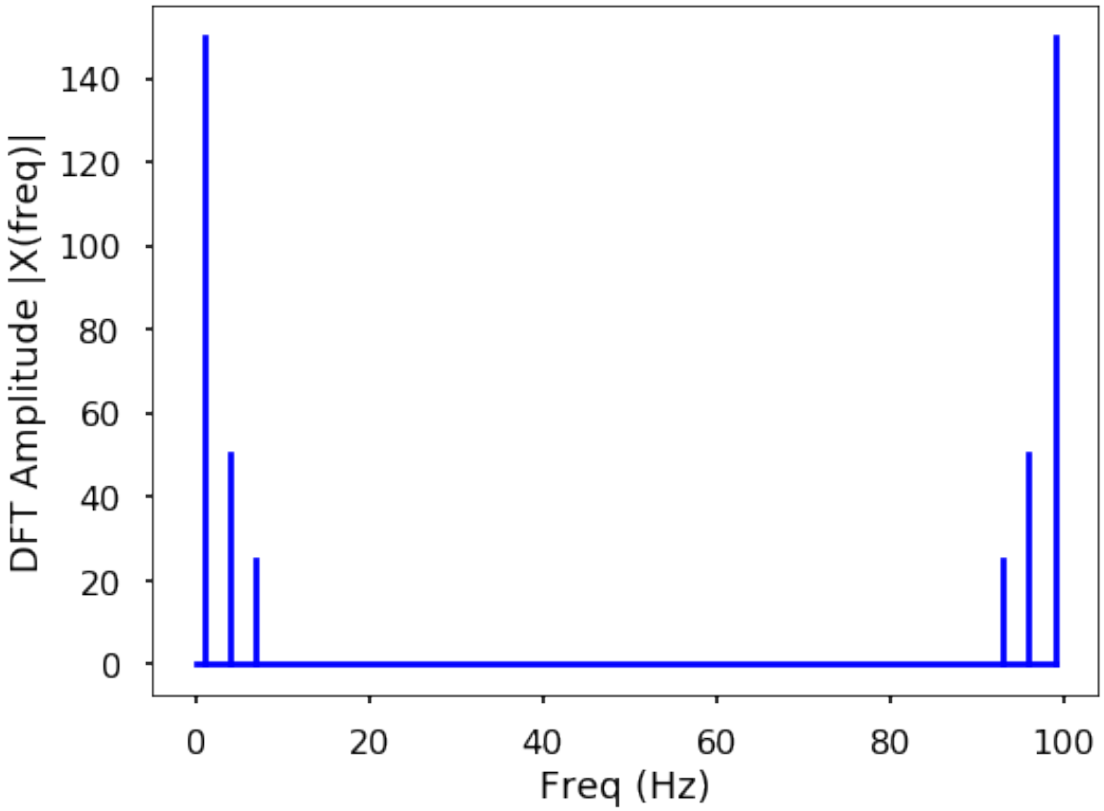
```
[ 0.00000000e+00  6.49951092e-01  1.24301000e+00  1.73098263e+00
  2.08154121e+00  2.28261600e+00  2.34327722e+00  2.29102039e+00
  2.16602580e+00  2.01350931e+00  1.87561275e+00  1.78433917e+00
  1.75681059e+00  1.79365925e+00  1.88074856e+00  1.99377423e+00
  2.10474409e+00  2.18898123e+00  2.23120727e+00  2.22945607e+00
  2.19600566e+00  2.15511222e+00  2.13796970e+00  2.17587843e+00
  2.29297677e+00  2.50000000e+00  2.79035655e+00  3.13938578e+00
  3.50706391e+00  3.84376807e+00  4.09811869e+00  4.22550953e+00
  4.19578177e+00  3.99863534e+00  3.64577057e+00  3.16934473e+00
  2.61699766e+00  2.04432572e+00  1.50614412e+00  1.04809007e+00
  7.00042246e-01  4.72482828e-01  3.56371693e-01  3.26445884e-01
  3.47223767e-01  3.80502964e-01  3.92885361e-01  3.61888418e-01
  2.79502648e-01  1.52571317e-01  3.06161700e-16 -1.52571317e-01
 -2.79502648e-01 -3.61888418e-01 -3.92885361e-01 -3.80502964e-01
 -3.47223767e-01 -3.26445884e-01 -3.56371693e-01 -4.72482828e-01
 -7.00042246e-01 -1.04809007e+00 -1.50614412e+00 -2.04432572e+00
 -2.61699766e+00 -3.16934473e+00 -3.64577057e+00 -3.99863534e+00
 -4.19578177e+00 -4.22550953e+00 -4.09811869e+00 -3.84376807e+00
 -3.50706391e+00 -3.13938578e+00 -2.79035655e+00 -2.50000000e+00
 -2.29297677e+00 -2.17587843e+00 -2.13796970e+00 -2.15511222e+00
 -2.19600566e+00 -2.22945607e+00 -2.23120727e+00 -2.18898123e+00
 -2.10474409e+00 -1.99377423e+00 -1.88074856e+00 -1.79365925e+00
 -1.75681059e+00 -1.78433917e+00 -1.87561275e+00 -2.01350931e+00
 -2.16602580e+00 -2.29102039e+00 -2.34327722e+00 -2.28261600e+00
 -2.08154121e+00 -1.73098263e+00 -1.24301000e+00 -6.49951092e-01]
```

```
In [6]: def DFT(x):
            """
            Function to calculate the
            discrete Fourier Transform
            of a 1D real-valued signal x
            """
            N = len(x)
            n = np.arange(N)
            k = n.reshape(N, 1)
            e = np.exp(-2j * np.pi * k * n / N)
            X = np.dot(e, x)
            return X
```

```
In [7]: X = DFT(x)
        # calculate the frequency
        N = len(X)
        n = np.arange(N)
        T = N/sr
        print("the value of T is: ",T)
        freq = n/T
        plt.figure(figsize = (8, 6))
```

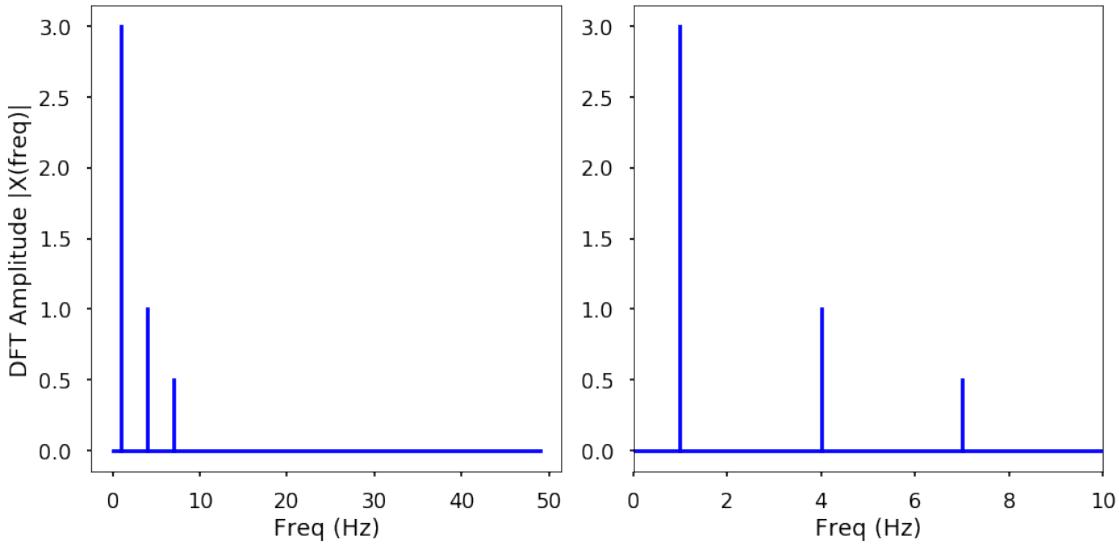
```
        plt.stem(freq, abs(X), "b", markerfmt=" ", basefmt="-b")
        plt.xlabel("Freq (Hz)")
        plt.ylabel("DFT Amplitude |X(freq)|")
        plt.show()
```

```
the value of T is:  1.0
```




```
In [ ]: n_oneside = N//2
        # get the one side frequency
        f_oneside = freq[:n_oneside]
        # normalize the amplitude
        X_oneside =X[:n_oneside]/n_oneside
        plt.figure(figsize = (12, 6))
        plt.subplot(121)
        plt.stem(f_oneside, abs(X_oneside), "b", markerfmt=" ", basefmt="-b")
        plt.xlabel("Freq (Hz)")
        plt.ylabel("DFT Amplitude |X(freq)|")
        plt.subplot(122)
        plt.stem(f_oneside, abs(X_oneside), "b", markerfmt=" ", basefmt="-b")
        plt.xlabel("Freq (Hz)")
```

```
plt.xlim(0, 10)
plt.tight_layout()
plt.show()
```



In [ ]:
```
## The values of X in frequency domain
print("the length of X is: ", len(X))
print("")
print(X)
```

```
the length of X is:  100

[-1.37667655e-14+0.00000000e+00j -1.36313052e-14-1.50000000e+02j
  2.25440429e-14+1.04743232e-14j -1.61609726e-14+7.20914309e-15j
 -4.53405371e-14-5.00000000e+01j -1.26558963e-14-3.78790318e-15j
  1.05357830e-14-1.71572004e-15j  1.93281229e-14-2.50000000e+01j
  1.15899682e-14-3.59790266e-14j -2.12861338e-14+7.19502279e-15j
  1.77121157e-14+3.52493083e-15j -1.32959968e-14+2.98753109e-14j
 -2.77039445e-14+1.27860910e-14j -5.91343476e-14-5.19418237e-14j
 -2.40408701e-14-1.94515350e-14j  3.94428298e-14-8.91983966e-15j
  2.96515218e-14-1.13687270e-14j  1.43660736e-13+1.30067597e-13j
  4.92303027e-14-8.14017890e-14j  2.57484215e-14-1.22350955e-13j
 -1.28073177e-13-2.92548660e-14j -2.67955631e-14-8.92824634e-14j
  3.80052003e-14-1.76086427e-14j  4.14320011e-14+6.85421309e-14j
 -1.37518001e-14+3.67234063e-14j  7.72582802e-14+6.44010731e-15j
  1.55207218e-13-1.27198308e-16j  1.15562868e-13-8.37436003e-14j
  2.12897571e-14-3.24967810e-14j  3.17414107e-14+1.59037202e-13j
  1.14623282e-13+5.12476281e-14j -5.07904388e-15+1.53242121e-13j
 -2.09946831e-13+1.56739986e-14j  4.14427806e-14+4.61058526e-14j
 -6.72160801e-15-3.96464670e-14j -1.23435136e-13-4.09817497e-13j
```

```
 1.21038840e-13+1.01552278e-13j  7.08691550e-14+9.68297358e-14j
-3.35261778e-13-6.88793394e-14j -8.94166019e-14+5.41631485e-14j
-3.13325854e-14-1.07774816e-14j  1.63603410e-13-1.53435539e-14j
-1.33994646e-13+1.24048870e-14j -1.78264251e-13+1.29674049e-13j
 1.07655451e-14-6.32297131e-14j -7.52730732e-15-1.26257290e-13j
 2.40113567e-13-9.23705556e-14j -4.88312807e-13+5.78485471e-13j
-6.05033392e-14+5.75512708e-14j -6.80016339e-14+1.42108547e-14j
-2.97539771e-14+6.18945917e-14j  4.62374277e-14-1.42108547e-13j
-4.25174684e-13+3.23391700e-13j  2.74661917e-14-9.81325795e-14j
-1.45116905e-14+1.10134124e-13j -1.56380676e-14+1.37538743e-13j
 2.89985931e-14+9.07698523e-14j -2.59585923e-13+1.95399252e-13j
 1.93434729e-14-2.54042364e-13j -6.49195424e-14+8.09796049e-14j
 2.25428830e-13+1.17133480e-13j  9.17262677e-14-1.05033437e-13j
 5.40081303e-13-8.73193476e-14j  3.18097883e-14+3.78513168e-13j
 5.56987753e-13+4.31108732e-14j  2.77059866e-13-5.00015920e-13j
 7.75527592e-15+5.63077179e-14j  3.50987358e-13-1.00376618e-13j
 6.97273752e-13-2.27704871e-13j -4.56266005e-14-4.01127949e-13j
 3.22464033e-13+4.12209274e-13j -9.66747615e-14+1.81265833e-13j
-1.12621647e-13-1.88051363e-13j -3.63991340e-13-1.15624938e-13j
-2.58242277e-13+2.22828590e-13j  3.77321763e-13-1.77787850e-13j
-9.31666879e-14+6.82647164e-13j  1.14693397e-13+5.89745733e-14j
-1.21263885e-13-6.02976264e-13j -3.05629817e-13-8.00061104e-14j
 4.89916705e-13-1.73729959e-13j  1.12462297e-14-5.77425046e-14j
-3.81603668e-14-3.83571682e-13j -2.32246240e-13-1.62598926e-13j
 2.85742988e-13+2.15787223e-13j  4.49425598e-13-1.23746251e-13j
-8.47226315e-15-1.76475405e-13j -3.05142730e-13-2.56508361e-13j
-8.18365100e-14+5.64040240e-15j  3.58387291e-14-2.40169388e-13j
 4.31194269e-13-1.59457736e-13j -1.62423944e-13-6.95458677e-14j
-2.52443700e-13+4.33295480e-13j -8.08430863e-13+2.50000000e+01j
 2.65056489e-13-4.39714213e-13j -5.22415163e-13-1.48211154e-14j
 3.44034403e-13+5.00000000e+01j  3.41750414e-14-1.21668376e-13j
 6.03813597e-16+3.49809439e-13j  3.18332913e-12+1.50000000e+02j]
```

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i \cdot 2\pi kn/N}.$$

image1.png!

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N} \\ &= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i2\pi k(2m)/N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i2\pi k(2m+1)/N} \\ &= \sum_{} x_{2m} \cdot e^{-i2\pi kn/(N/2)} + e^{-i2\pi k/N} \sum_{} x_{2m+1} \cdot e^{-i2\pi km/(N/2)}. \end{aligned}$$

image2.png

$$0 \leq m \leq \frac{N}{2}, \text{ but } 0 \leq k \leq N;$$

image1.png!

2 THE INVERSE DFT

2.0.1 The inverse transform of the DFT can be computed using:

2.1 Write a program to compute in the inverse transform of X

3 Fast Fourier Transform (FFT)

3.1 The Fast Fourier Transform (FFT) is an efficient algorithm used to calculate the DFT of a sequence. It is a divide-and-conquer algorithm that recursively breaks the DFT into smaller DFTs to reduce the number of computations.

3.2 As a result, it successfully reduces the complexityof the DFT from $O(n^2)$ to $O(nlogn)$, where n is the size of the data.

3.3 This reduction in computation time is significant, especially for data with large N, and FFT is widely used in engineering, science, and mathematics for this reason.

3.3.1 By dividing the data points into two part ie even and odd data points

3.3.2 The equation, are two smaller DFTs and For each term,

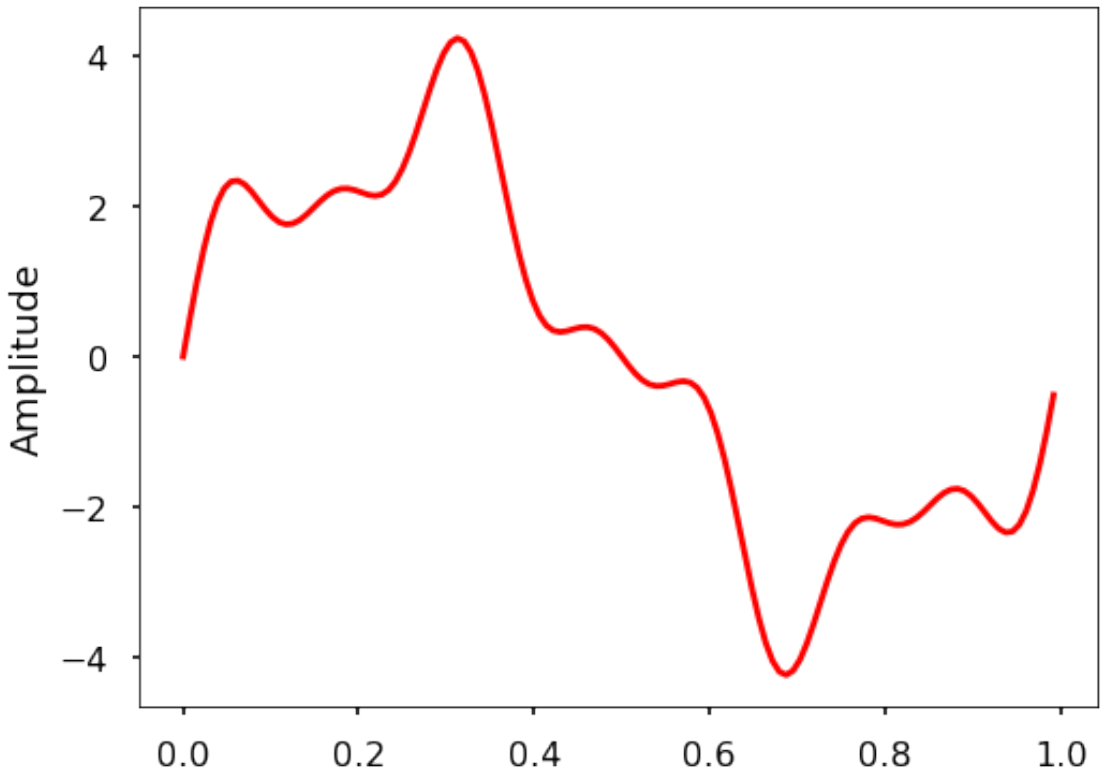
1. FFT works using this recursive approach.

2. Note that the input signal to FFT should have a length of power of 2. If it is not, then you need to fill up zeros to the next power of 2 size

```
In [ ]: import matplotlib.pyplot as plt
        import numpy as np
        plt.style.use("seaborn-poster")
        %matplotlib inline
        def FFT(x):
            """
            A recursive implementation of
            the 1D Cooley-Tukey FFT, the
            input should have a length of
            power of 2.
            """
            N = len(x)
            if N == 1:
                return x
            else:
                X_even = FFT(x[::2])
```

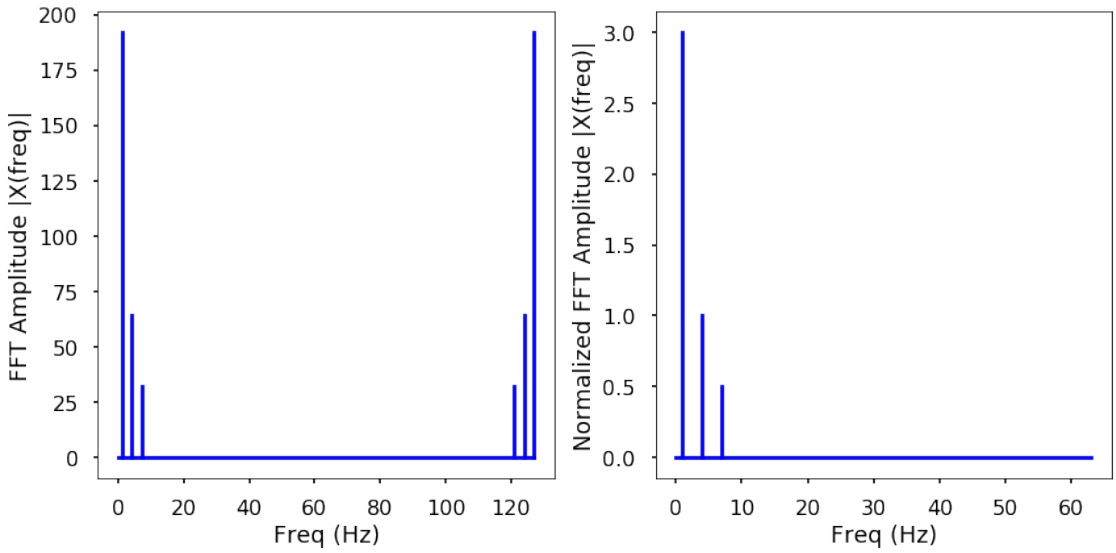
```
    X_odd = FFT(x[1::2])
    factor = np.exp(-2j*np.pi*np.arange(N)/ N)
    X = np.concatenate(\
    [X_even+factor[:int(N/2)]*X_odd,
    X_even+factor[int(N/2):]*X_odd])
    return X
```

In [ ]:
```
# sampling rate
sr = 128
# sampling interval
ts = 1.0/sr
t = np.arange(0,1,ts)
freq = 1.
x = 3*np.sin(2*np.pi*freq*t)
freq = 4
x += np.sin(2*np.pi*freq*t)
freq = 7
x += 0.5* np.sin(2*np.pi*freq*t)
plt.figure(figsize = (8, 6))
plt.plot(t, x, "r")
plt.ylabel("Amplitude")
plt.show()
```



3.4 Que: Use the FFT function to calculate the Fourier transform of the above signal. Plot the amplitude spectrum for both the two- and one-sided frequencies.

```
In [ ]: X=FFT(x)
        # calculate the frequency
        N = len(X)
        n = np.arange(N)
        T = N/sr
        freq = n/T
        plt.figure(figsize = (12, 6))
        plt.subplot(121)
        plt.stem(freq, abs(X), "b", markerfmt=" ", basefmt="-b")
        plt.xlabel("Freq (Hz)")
        plt.ylabel("FFT Amplitude |X(freq)|")
        # Get the one-sided spectrum
        n_oneside = N//2
        # get the one side frequency
        f_oneside = freq[:n_oneside]
        # normalize the amplitude
        X_oneside =X[:n_oneside]/n_oneside
        plt.subplot(122)
        plt.stem(f_oneside, abs(X_oneside), "b", markerfmt=" ", basefmt="-b")
        plt.xlabel("Freq (Hz)")
        plt.ylabel("Normalized FFT Amplitude |X(freq)|")
        plt.tight_layout()
        plt.show()
```



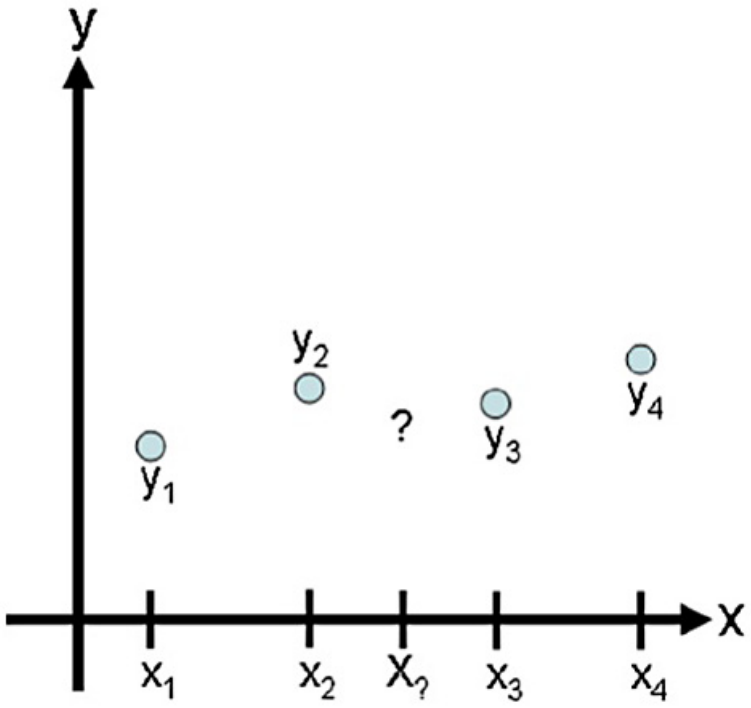


image1.png!

3.5 QUE: Generate a simple signal of length 2048, and record the time it will take to run the FFT; compare the speed with the DFT.

3.6 FFT IN PYTHON

3.6.1 Python has very mature FFT functions both in NumPy and SciPy

3.6.2 Que Use the fft and ifft functions from NumPy to calculate the FFT amplitude spectrum and inverse FFT to obtain the original signal. Plot both results. Time the fft function using this 2000-length signal.

3.7 TOPIC 2: INTERPOLATION

3.7.1 2.1 Assume we have a dataset consisting of independent data values, x_i , and dependent data values, y_i where $ i = 1, . . . , n $. We would like to find an estimation function $y'(x_i)$ such that $y(x_i) = y_i$ for every point in our dataset. This means the estimation function goes through our data points. Given a new x , we can interpolate its function value using $y'(x)$. In this context, $y'(x)$ is called an interpolation function. The Figure below shows the interpolation problem statement.

3.8 Some of the Interpolation methods are:

1. LINEAR INTERPOLATION
2. CUBIC SPLINE INTERPOLATION
3. LAGRANGE POLYNOMIAL INTERPOLATION
4. NEWTON'S POLYNOMIAL INTERPOLATION

$$\hat{y}(x) = y_i + \frac{(y_{i+1} - y_i)(x - x_i)}{(x_{i+1} - x_i)}.$$

image1.png!

3.9 2.2 LINEAR INTERPOLATION

3.9.1 In linear interpolation, the estimated point is assumed to lie on the line joining the nearest points to the left and right. Then the linear interpolation at x is:

3.9.2 Que: Find the linear interpolation at x = 1.5 based on the data x = [0, 1, 2], y = [1, 3, 2]. Verify the result using SciPy's function interp1d.

3.10 2.3 CUBIC INTERPOLATION

3.10.1 In cubic spline interpolation (shown below), the interpolation function is a set of piecewise cubic functions. Specifically, we assume that the points(x_i, y_i) and $(x_{i+1}$, $y_{i+1}) $ are joined by a cubic polynomial $S_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$ that is valid for x_i $x $ x_{i+1} for $i = 1, ..., n1$. To find the interpolation function, we must first determine the coefficients a_i, b_i, c_i, d_i for each of the cubic functions. For n points, there are $n1$ cubic functions to find, and each cubic function requires four coefficients. Therefore we have a total of $4(n1)$ unknowns, and so we need $4(n1)$ independent equations to find all the coefficients

3.10.2 Point to note: cubic functions must intersect the data the points on the left and on the right:

3.10.3 Constrain the splines to have continuous first and second derivatives at the data points i = 2, . . . , n 1:

3.10.4 The last two constraints are assume that the second derivatives are zero at the endpoints. This means that the curve is a "straight line" at the end points.

3.10.5 Example: Use Cubic Spline to plot the cubic spline interpolation of the dataset x = [0, 1, 2] and y = [1, 3, 2] for 0 x 2.

```
In [ ]: from scipy.interpolate import CubicSpline
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use("seaborn-poster")
        x = [0, 1, 2]
        y = [1, 3, 2]
        # use bc_type = "natural" adds the constraints
        f = CubicSpline(x, y, bc_type="natural")
        x_new = np.linspace(0, 2, 100)
        y_new = f(x_new)
        In [3]: plt.figure(figsize = (10,8))
        plt.plot(x_new, y_new, "b")
```

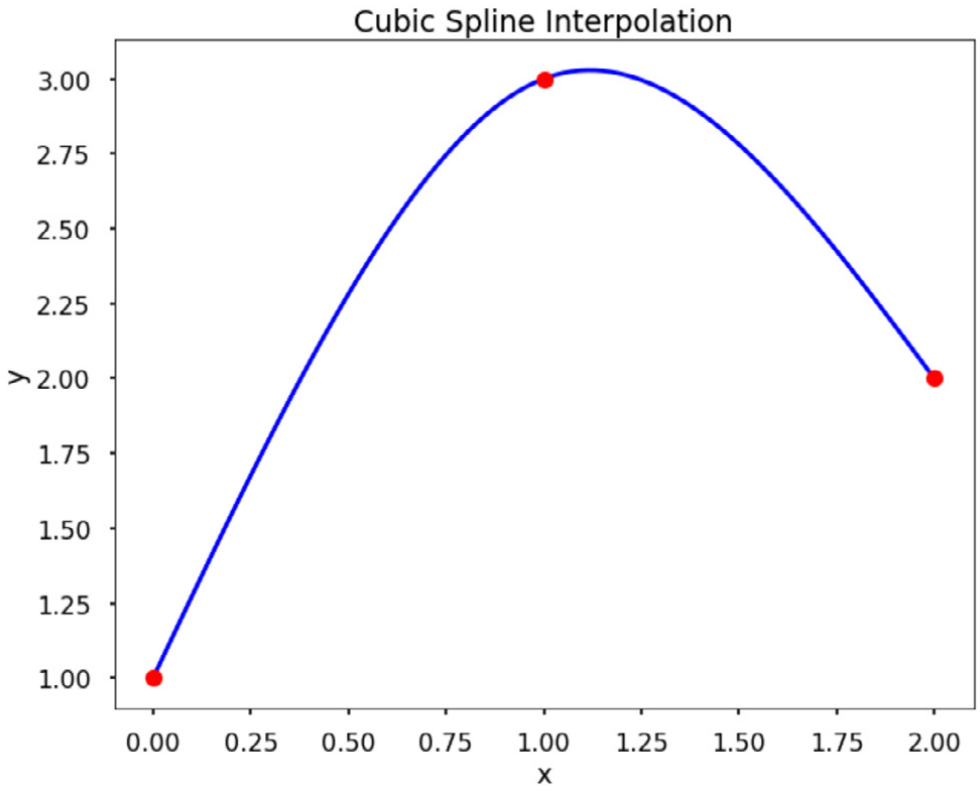


image1.png!

$$S_i(x_i) = y_i, \qquad i = 1, \ldots, n-1,$$
$$S_i(x_{i+1}) = y_{i+1}, \qquad i = 1, \ldots, n-1,$$

image1.png!

$$S_i'(x_{i+1}) = S_{i+1}'(x_{i+1}), \quad i = 1, \ldots, n-2,$$
$$S_i''(x_{i+1}) = S_{i+1}''(x_{i+1}), \quad i = 1, \ldots, n-2,$$

image1.png!

$$S_1''(x_1) = 0,$$
$$S_{n-1}''(x_n) = 0.$$

image1.png!

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)(x-a)^n}{n!},$$

image1.png!

```
        plt.plot(x, y, "ro")
        plt.title("Cubic Spline Interpolation")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.show()
```

```
In [ ]: print('Old values of y are: ', y)
        print('New values of y are: ', y_new)
```

```
In [ ]:
```

3.11 Assgmnet to be ssubmitted

3.11.1 Write a function my_cubic_spline(x, y, X) where x and y are arrays that contain experimental data points, and X is an array. Assume that x and X are in ascending order and have unique elements. The output argument, Y, should be an array the same size as X, where $Y[i]$ is cubic spline interpolation of $X[i]$. Do not use interp1d or CubicSpline.

3.12 TOPIC 3: Taylor Series

3.12.1 A sequence is an ordered set of numbers denoted by the list of numbers inside parentheses.

3.12.2 For example, $s = (s_1, s_2, s_3, ...)$ means s is the sequence $s_1, s_2, s_3, . . ., $ and so on. In this context, "ordered" means that s_1 comes before s_2, not that $s_1 < s_2$. Many sequences have a more complicated structure.

3.12.3 An infinite sequence is a sequence with an infinite number of terms, and an infinite series is the sum of an infinite sequence.

3.12.4 A Taylor series expansion is a representation of a function by an infinite series of polynomials around a point. Mathematically, the Taylor series of a function, $f(x)$, is defined as}

$$f(x) = \sum_{n=0}^{\infty} \frac{f^n(a)(x-a)^n}{n!} \tag{1}$$

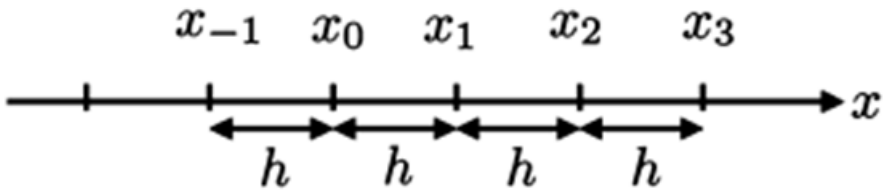


image1.png!

3.12.5 It is often useful to approximate functions by using an Nth order Taylor series approximation of a function, which is a truncation of its Taylor expansion at some$ n = N$. This technique is especially powerful especially when there is a point around which we have knowledge about a function and all its derivatives.

3.12.6 Example: Use Python to plot the sin function along with the first, third, fifth, and seventh order Taylor series approximations. Note that this involves the zeroth to third terms in the formula given earlier.

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use("seaborn-poster")
        x = np.linspace(-np.pi, np.pi, 200)
        y = np.zeros(len(x))
        labels = ["First Order", "Third Order", "Fifth Order", "Seventh Order"]
        plt.figure(figsize = (10,8))
        for n, label in zip(range(4), labels):
            y=y+((-1)**n*(x)**(2*n+1))/np.math.factorial(2*n+1)
            plt.plot(x,y, label = label)
            plt.plot(x, np.sin(x), "k", label = "Analytic")
        plt.grid()
        plt.title("Taylor Series Approximations of Various Orders")
        plt.xlabel("x")
        plt.ylabel("y")
        plt.legend()
        plt.show()
```

4 Topic 4: NUMERICAL DIFFERENTIATION

4.0.1 A numerical grid can be defined as an evenly spaced set of points over the domain of a function (i.e., the independent variable), over some interval. The spacing or step size of a numerical grid is the distance between adjacent points on the grid. For the purpose of this text, if x is a numerical grid, then x_j is the j th point in the numerical grid, and h is the spacing between x_{j1} and x_j. The Figure below shows an example of a numerical grid.

4.0.2 The purpose of the numerical differentiation is to derive the methods of approximating the derivative of $f(x)$ over a numerical grid and determine its accuracy

4.1 USING FINITE DIFFERENCE TO APPROXIMATE DERIVATIVES

4.1.1 The derivative $f'(x)$ of a function $f(x)$ at the point $x = a$ is defined as

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a}.$$

we shall be considering following methods of finding the derivative of $f(x)$: 1. Forward difference 2. Backward difference 3. Central difference

Forward difference estimates the slope of the function at xj using the line that connects $(x_j, f(x_j))$ and $(x_{j+1}, f(x_{j+1}))$:

$$f'(x_j) = \frac{f(x_{j+1}) - f(x_j)}{x_{j+1} - x_j}.$$

Backward difference estimates the slope of the function at xj using the line that connects $(x_j, f(x_j))$ and $(x_{j-1}, f(x_{j-1}))$:

$$f'(x_j) = \frac{f(x_j) - f(x_{j-1})}{x_j - x_{j-1}}.$$

Central difference estimates the slope of the function at xj using the line that connects $(x_{j+1}, f(x_{j+}))$ and $(x_{j-1}, f(x_{j-1}))$:

$$f(x) = \frac{f(x_j)(x-x_j)^0}{0!} + \frac{f'(x_j)(x-x_j)^1}{1!} + \frac{f''(x_j)(x-x_j)^2}{2!} + \frac{f'''(x_j)(x-x_j)^3}{3!} + \cdots.$$

Backward difference.png!

$$f(x_{j+1}) = \frac{f(x_j)(x_{j+1}-x_j)^0}{0!} + \frac{f'(x_j)(x_{j+1}-x_j)^1}{1!} + \frac{f''(x_j)(x_{j+1}-x_j)^2}{2!} + \frac{f'''(x_j)(x_{j+1}-x_j)^3}{3!} + \cdots.$$

Backward difference.png!

$$f'(x_j) = \frac{f(x_{j+1}) - f(x_{j-1})}{x_{j+1} - x_{j-1}}.$$

4.2 USING FINITE DIFFERENCE TO APPROXIMATE DERIVATIVES WITH TAYLOR SERIES

Taylor series to derive an approximation for the derivative of $f(x)$. For an arbitrary function $f(x)$, the Taylor series of f around $a = x_j$ is

If x is on a grid of points with spacing h, we can compute the Taylor series at $x = x_{j+1}$ to obtain Substituting $h = x_{j+1}x_j$ and solving for $f'(x_j)$ gives the equation

$$- \frac{f''(x_j)h}{2!} - \frac{f'''(x_j)h^2}{3!} - \cdots,$$

The terms that are in parentheses are called higher order terms of h. The higher order terms can be rewritten as

$$-\frac{f''(x_j)h}{2!} - \frac{f'''(x_j)h^2}{3!} - \cdots = h(\alpha + \epsilon(h)),$$

$$f'(x_j) = \frac{f(x_{j+1}) - f(x_j)}{h} + \left(-\frac{f''(x_j)h}{2!} - \frac{f'''(x_j)h^2}{3!} - \cdots\right).$$

Backward difference.png!

$$f'(x_j) = \frac{f(x_{j+1}) - f(x_j)}{h} +$$

Substituting O(h) into the previous equation gives
This gives the forward difference formula for approximating derivatives as

$$f'(x_j) \approx \frac{f(x_{j+1}) - f(x_j)}{h}$$

4.3 EXAMPLE:

4.3.1 Consider the function $f(x) = cos(x)$. We know that the derivative of $cos(x)$ is $sin(x)$. Although in practice we may not know the underlying function we are finding the derivative for, we use the simple example to illustrate the aforementioned numerical differentiation methods and their accuracy. The following code computes the derivatives numerically

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use("seaborn-poster")
        %matplotlib inline
         # step size
        h = 0.1
        # define grid
        x = np.arange(0, 2*np.pi, h)
        # compute function
        y = np.cos(x)
        # compute vector of forward differences
        forward_diff = np.diff(y)/h
        print('The forward solutions are: ', forward_diff)
        # compute corresponding grid
        x_diff = x[:-1:]
        # compute exact solution
        exact_solution = -np.sin(x_diff)
        print('the exact solution are: ', exact_solution)
        # Plot solution
        plt.figure(figsize = (12, 8))
        plt.plot(x_diff, forward_diff, "-", \
        label = "Finite difference approximation")
        plt.plot(x_diff, exact_solution, label = "Exact solution")
        plt.legend()
        plt.show()
```

$$f(x_{j-1}) = f(x_j) - hf'(x_j) + \frac{h^2 f''(x_j)}{2} - \frac{h^3 f'''(x_j)}{6} + \cdots$$

Backward difference.png!

```
# Compute max error between
# numerical derivative and exact solution
max_error = max(abs(exact_solution - forward_diff))
print('maximum error', max_error)
```

4.4 APPROXIMATING OF HIGHER ORDER DERIVATIVES

4.4.1 It also possible to use the Taylor series to approximate higher-order derivatives (e.g., $f''(x_j)$, $f'''(x_j)$, etc.). For example, taking the Taylor series around $a = x_j$ and then computing it at $x = x_{j1}$ and x_{j+1} gives

$$f(x_{j+1}) = f(x_j) + hf'(x_j) + \frac{h^2 f''(x_j)}{2} + \frac{h^3 f'''(x_j)}{6} + \cdots .$$

$$f(x_{j-1}) + f(x_{j+1}) = 2f(x_j) + h^2 f''(x_j) + \frac{h^4 f''''(x_j)}{24} + \cdots ,$$

$$f''(x_j) \approx \frac{f(x_{j+1}) - 2f(x_j) + f(x_{j-1})}{h^2}$$