**DAILY ASSESSMENT FORMAT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date:** | **26-05-2020** | **Name:** | **Anand kumar k** |
| **Course:** |  | **USN:** | **4al16ec002** |
| **Topic:** | **Fast fourier transform** | **Semester & Section:** | **8th sem ‘A’ sec** |
| **Github Repository:** | **Anand-courses** |  |  |

|  |
| --- |
| **FORENOON SESSION DETAILS** |
| **Image of session** |
|  |
| **Fast Fourier transform** (**FFT**) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). Fourier analysis converts a signal from its original domain to a representation in the frequency domain and vice versa. The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields, but computing it directly from the definition is often too slow to be practical. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result, it manages to reduce the complexity of computing the DFT from{\displaystyle O\left(N^{2}\right)}, which arises if one simply applies the definition of DFT, to {\displaystyle O(N\log N)}, where {\displaystyle N} is the data size. The difference in speed can be enormous, especially for long data sets where *N* may be in the thousands or millions. In the presence of round-off error, many FFT algorithms are much more accurate than evaluating the DFT definition directly or indirectly. There are many different FFT algorithms based on a wide range of published theories, from simple complex-number arithmetic to group theory and number theory.  **FIR filter**  In [signal processing](https://en.wikipedia.org/wiki/Signal_processing), a finite impulse response (FIR) filter is a [filter](https://en.wikipedia.org/wiki/Filter_(signal_processing)) whose [impulse response](https://en.wikipedia.org/wiki/Impulse_response) is of *finite* duration, because it settles to zero in finite time. This is in contrast to [infinite impulse response](https://en.wikipedia.org/wiki/Infinite_impulse_response) (IIR) filters, which may have internal feedback and may continue to respond indefinitely.  The [impulse response](https://en.wikipedia.org/wiki/Impulse_response) of an Nth-order discrete-time FIR filter lasts exactly *N* + 1 samples before it then settles to zero.  **IIR filter**  **Infinite impulse response** (**IIR**) is a property applying to many [linear time-invariant systems](https://en.wikipedia.org/wiki/Linear_time-invariant_system) that are distinguished by having an [impulse response](https://en.wikipedia.org/wiki/Impulse_response) *h*(*t*) which does not become exactly zero past a certain point, but continues indefinitely. This is in contrast to a [finite impulse response](https://en.wikipedia.org/wiki/Finite_impulse_response) (FIR) system in which the impulse response *does* become exactly zero at times *t* > *T* for some finite *T*, thus being of finite duration. Common examples of linear time-invariant systems are most [electronic](https://en.wikipedia.org/wiki/Electronic_filter) and [digital filters](https://en.wikipedia.org/wiki/Digital_filter). Systems with this property are known as IIR filter.  In practice, the impulse response, even of IIR systems, usually approaches zero and can be neglected past a certain point. However the physical systems which give rise to IIR or FIR responses are dissimilar, and therein lies the importance of the distinction. For instance, analog electronic filters composed of resistors, capacitors, and/or inductors are generally IIR filters.  On the other hand, [discrete-time filters](https://en.wikipedia.org/wiki/Discrete-time_filter) based on a tapped delay line employing no feedback are necessarily FIR filters. The capacitors in the analog filter have a "memory" and their internal state never completely relaxes following an impulse. But in the latter case, after an impulse has reached the end of the tapped delay line, the system has no further memory of that impulse and has returned to its initial state; its impulse response beyond that point is exactly zero.  The CWT and the discrete wavelet transforms differ in how they discretize the scale parameter. The CWT typically uses exponential scales with a base smaller than 2, for example 21/12 . The discrete wavelet transform always uses exponential scales with the base equal to 2. The scales in the discrete wavelet transform are powers of 2. Keep in mind that the physical interpretation of scales for both the CWT and discrete wavelet transforms requires the inclusion of the signal’s sampling interval if it is not equal to one.  The DWT provides a sparse representation for many natural signals. In other words, the important features of many natural signals are captured by a subset of DWT coefficients that is typically much smaller than the original signal. This “compresses” the signal. With the DWT, you always end up with the same number of coefficients as the original signal, but many of the coefficients may be close to zero in value. As a result, you can often throw away those coefficients and still maintain a high-quality signal approximation. With the CWT, you go from N samples for an N-length signal to a M-by-N matrix of coefficients with M equal to the number of scales.  The Short-time Fourier transform (STFT), is a Fourier-related transform used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time.[[1]](https://en.wikipedia.org/wiki/Short-time_Fourier_transform#cite_note-1) In practice, the procedure for computing STFTs is to divide a longer time signal into shorter segments of equal length and then compute the Fourier transform separately on each shorter segment. This reveals the Fourier spectrum on each shorter segment. One then usually plots the changing spectra as a function of time, known as a spectrogram or waterfall plot. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date:** | **26-05-2020** | **Name:** | **Anand kumar k** | |
| **Course:** |  | **USN:** | **4al16ec002** | |
| **Topic:** | **python** | **Semester & Section:** | **8th sem ‘A’ sec** | |
| **AFTERNOON SESSION DETAILS** | | | |
| **Image of session** | | | |
| Using loops we can call any function multiple times, even your own functions. Let's suppose we defined this function:   1. def celsius\_to\_kelvin(cels): 2. return cels + 273.15   That is a function that gets a number as input, adds 273.15 to it and returns the result. A *for* loop allows us to execute that function over a list of numbers:   1. monday\_temperatures = [9.1, 8.8, -270.15] 3. for temperature in monday\_temperatures: 4. print(celsius\_to\_kelvin(temperature))   The output of that would be:  282.25 281.95 3.0   * **For loops** are useful for executing a command over a large number of items. * You can create a **for loop** like so:  1. for letter in 'abc': 2. print(letter.upper())   Output:  A B C  **While loops** will run as long as a condition is true:   1. while datetime.datetime.now() < datetime.datetime(2090, 8, 20, 19, 30, 20): 2. print("It's not yet 19:30:20 of 2090.8.20")   The loop above will print out the string inside print() over and over again until the 20th of August, 2090. | | | |