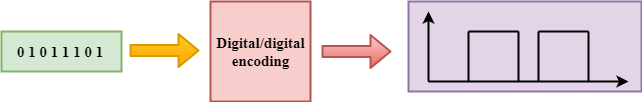
Lecture 1

Digital Transmission

Data can be represented either in analog or digital form. The computers used the digital form to store the information. Therefore, the data needs to be converted in digital form so that it can be used by a computer.

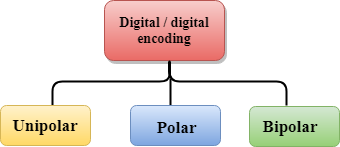
DIGITAL-TO-DIGITAL CONVERSION

Digital-to-digital encoding is the representation of digital information by a digital signal. When binary 1s and 0s generated by the computer are translated into a sequence of voltage pulses that can be propagated over a wire, this process is known as digital-to-digital encoding.



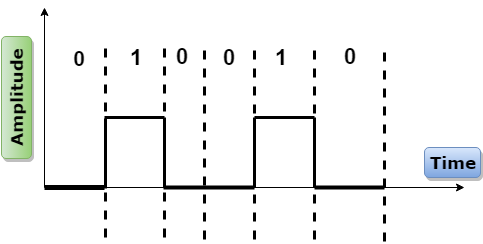
Digital-to-digital encoding is divided into three categories:

* Unipolar Encoding
* Polar Encoding
* Bipolar Encoding



Unipolar

* Digital transmission system sends the voltage pulses over the medium link such as wire or cable.
* In most types of encoding, one voltage level represents 0, and another voltage level represents 1.
* The polarity of each pulse determines whether it is positive or negative.
* This type of encoding is known as Unipolar encoding as it uses only one polarity.
* In Unipolar encoding, the polarity is assigned to the 1 binary state.
* In this, 1s are represented as a positive value and 0s are represented as a zero value.
* In Unipolar Encoding, '1' is considered as a high voltage and '0' is considered as a zero voltage.
* Unipolar encoding is simpler and inexpensive to implement.

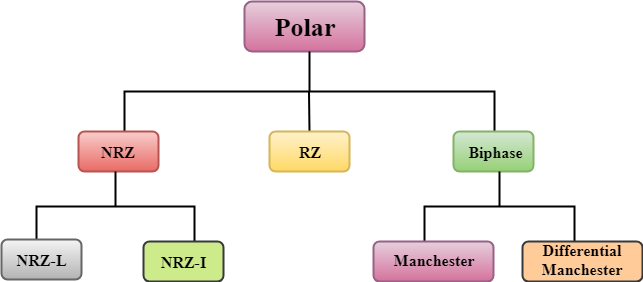


Unipolar encoding has two problems that make this scheme less desirable:

* DC Component
* Synchronization

Polar

* Polar encoding is an encoding scheme that uses two voltage levels: one is positive, and another is negative.
* By using two voltage levels, an average voltage level is reduced, and the DC component problem of unipolar encoding scheme is alleviated.



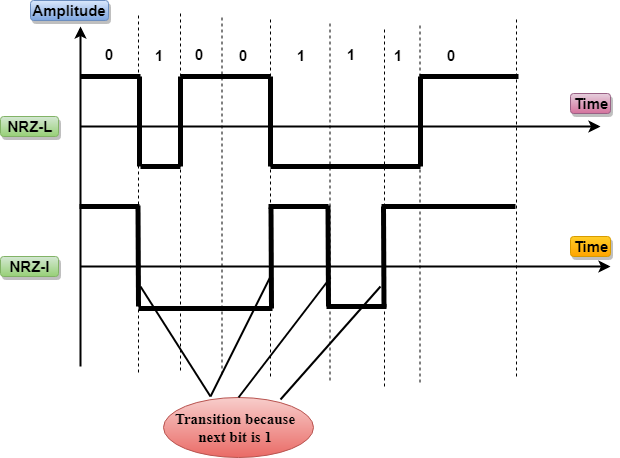
NRZ

* NRZ stands for Non-return zero.
* In NRZ encoding, the level of the signal can be represented either positive or negative.

**The two most common methods used in NRZ are:**

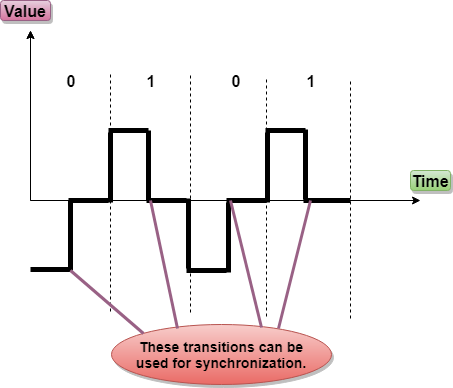
**NRZ-L:** In NRZ-L encoding, the level of the signal depends on the type of the bit that it represents. If a bit is 0 or 1, then their voltages will be positive and negative respectively. Therefore, we can say that the level of the signal is dependent on the state of the bit.

**NRZ-I:** NRZ-I is an inversion of the voltage level that represents 1 bit. In the NRZ-I encoding scheme, a transition occurs between the positive and negative voltage that represents 1 bit. In this scheme, 0 bit represents no change and 1 bit represents a change in voltage level.



RZ

* RZ stands for Return to zero.
* There must be a signal change for each bit to achieve synchronization. However, to change with every bit, we need to have three values: positive, negative and zero.
* RZ is an encoding scheme that provides three values, positive voltage represents 1, the negative voltage represents 0, and zero voltage represents none.
* In the RZ scheme, halfway through each interval, the signal returns to zero.
* In RZ scheme, 1 bit is represented by positive-to-zero and 0 bit is represented by negative-to-zero.



**Disadvantage of RZ:**

It performs two signal changes to encode one bit that acquires more bandwidth.

Biphase

* Biphase is an encoding scheme in which signal changes at the middle of the bit interval but does not return to zero.

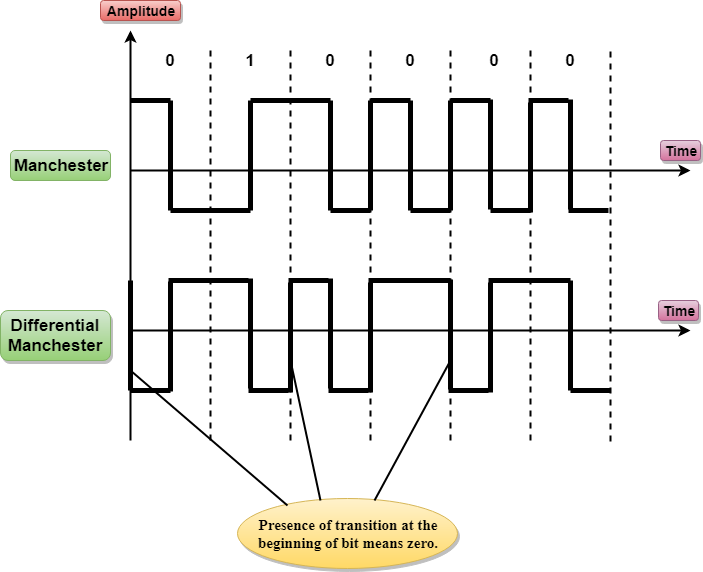
Biphase encoding is implemented in two different ways:

**Manchester**

* It changes the signal at the middle of the bit interval but does not return to zero for synchronization.
* In Manchester encoding, a negative-to-positive transition represents binary 1, and positive-to-negative transition represents 0.
* Manchester has the same level of synchronization as RZ scheme except that it has two levels of amplitude.

**Differential Manchester**

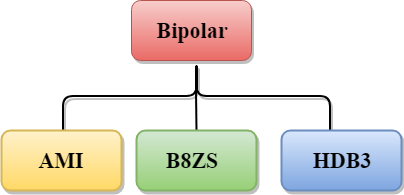
* It changes the signal at the middle of the bit interval for synchronization, but the presence or absence of the transition at the beginning of the interval determines the bit. A transition means binary 0 and no transition means binary 1.
* In Manchester Encoding scheme, two signal changes represent 0 and one signal change represent 1.



Bipolar

* Bipolar encoding scheme represents three voltage levels: positive, negative, and zero.
* In Bipolar encoding scheme, zero level represents binary 0, and binary 1 is represented by alternating positive and negative voltages.
* If the first 1 bit is represented by positive amplitude, then the second 1 bit is represented by negative voltage, third 1 bit is represented by the positive amplitude and so on. This alternation can also occur even when the 1bits are not consecutive.

**Bipolar can be classified as:**



AMI

* AMI stands for ***alternate mark inversion*** where mark work comes from telegraphy which means 1. So, it can be redefined as **alternate 1 inversion**.
* In Bipolar AMI encoding scheme, 0 bit is represented by zero level and 1 bit is represented by alternating positive and negative voltages.

**Advantage:**

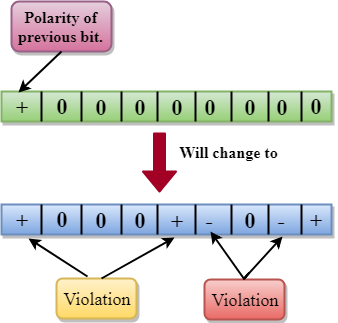
* DC component is zero.
* Sequence of 1s bits are synchronized.

**Disadvantage:**

* This encoding scheme does not ensure the synchronization of a long string of 0s bits.

B8ZS

* B8ZS stands for **Bipolar 8-Zero Substitution**.
* This technique is adopted in North America to provide synchronization of a long sequence of 0s bits.
* In most of the cases, the functionality of B8ZS is similar to the bipolar AMI, but the only difference is that it provides the synchronization when a long sequence of 0s bits occur.
* B8ZS ensures synchronization of a long string of 0s by providing force artificial signal changes called violations, within 0 string pattern.
* When eight 0 occurs, then B8ZS implements some changes in 0s string pattern based on the polarity of the previous 1 bit.
* If the polarity of the previous 1 bit is positive, the eight 0s will be encoded as zero, zero, zero, positive, negative, zero, negative, positive.

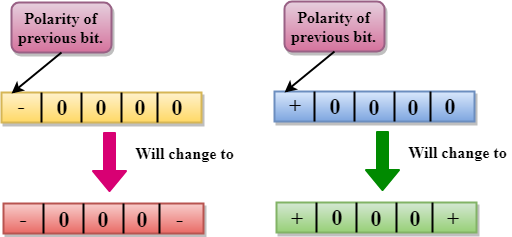


* If the polarity of previous 1 bit is negative, then the eight 0s will be encoded as zero, zero, zero, negative, positive, zero, positive, negative.

HDB3

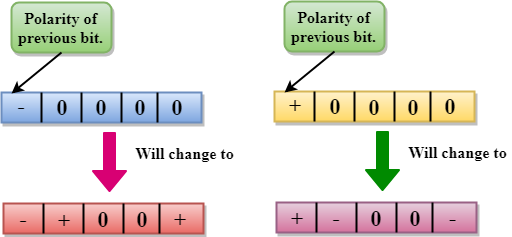
* HDB3 stands for **High-Density Bipolar 3**.
* HDB3 technique was first adopted in Europe and Japan.
* HDB3 technique is designed to provide the synchronization of a long sequence of 0s bits.
* In the HDB3 technique, the pattern of violation is based on the polarity of the previous bit.
* When four 0s occur, HDB3 looks at the number of 1s bits occurred since the last substitution.
* If the number of 1s bits is odd, then the violation is made on the fourth consecutive of 0. If the polarity of the previous bit is positive, then the violation is positive. If the polarity of the previous bit is negative, then the violation is negative.

**If the number of 1s bits since the last substitution is odd.**



If the number of 1s bits is even, then the violation is made on the place of the first and fourth consecutive 0s. If the polarity of the previous bit is positive, then violations are negative, and if the polarity of the previous bit is negative, then violations are positive.

**If the number of 1s bits since the last substitution is even.**



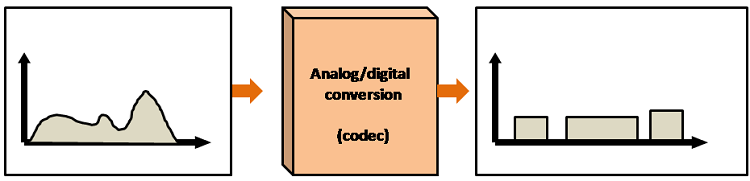
ANALOG-TO-DIGITAL CONVERSION

* When an analog signal is digitalized, this is called an analog-to-digital conversion.
* Suppose human sends a voice in the form of an analog signal, we need to digitalize the analog signal which is less prone to noise. It requires a reduction in the number of values in an analog message so that they can be represented in the digital stream.
* In analog-to-digital conversion, the information contained in a continuous wave form is converted in digital pulses.

Techniques for Analog-To-Digital Conversion

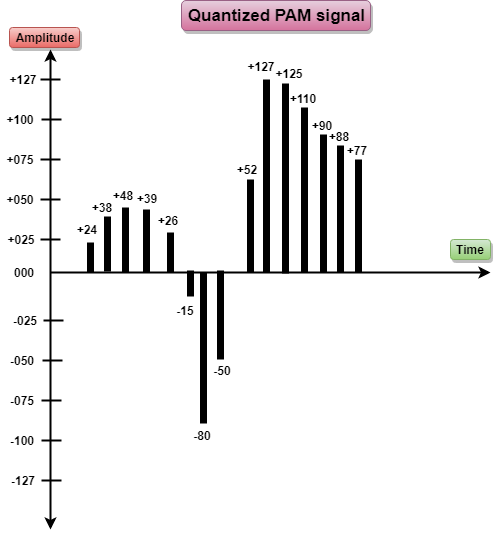
PAM

* PAM stands for **pulse amplitude modulation**.
* PAM is a technique used in analog-to-digital conversion.
* PAM technique takes an analog signal, samples it, and generates a series of digital pulses based on the result of sampling where sampling means measuring the amplitude of a signal at equal intervals.
* PAM technique is not useful in data communication as it translates the original wave form into pulses, but these pulses are not digital. To make them digital, PAM technique is modified to PCM technique.

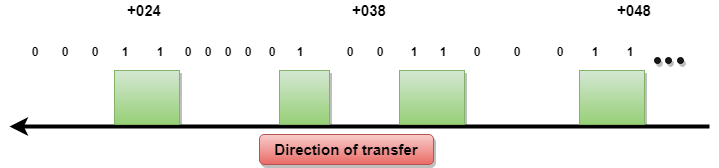
  


PCM

* PCM stands for **Pulse Code Modulation**.
* PCM technique is used to modify the pulses created by PAM to form a digital signal. To achieve this, PCM quantizes PAM pulses. Quantization is a process of assigning integral values in a specific range to sampled instances.
* PCM is made of four separate processes: PAM, quantization, binary encoding, and digital-to-digital encoding.



**PCM**



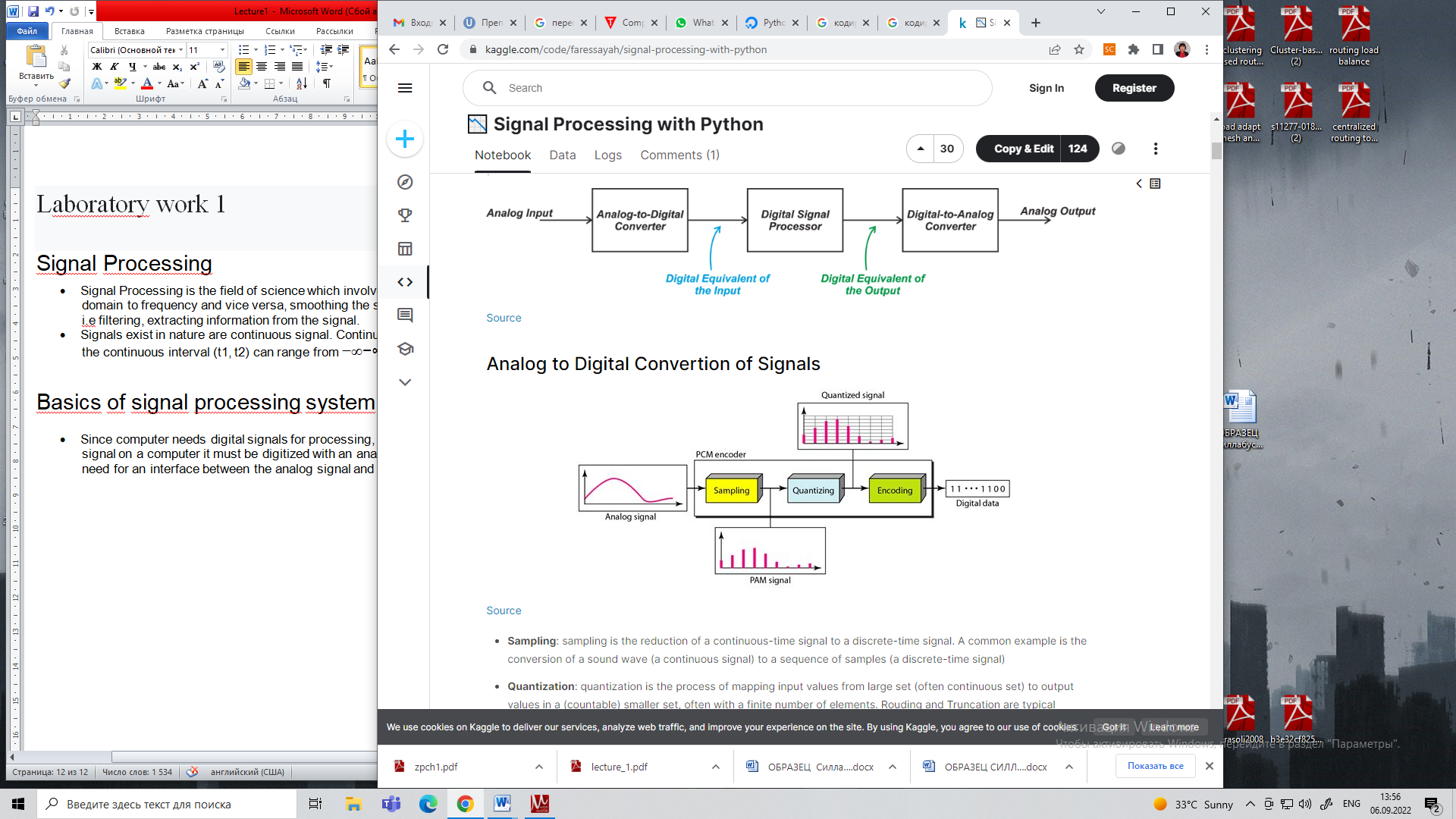
Laboratory work 1

Signal Processing

* Signal Processing is the field of science which involves the manipulation of signal from time domain to frequency and vice versa, smoothing the signal, separating the noise from signal i.e filtering, extracting information from the signal.
* Signals exist in nature are continuous signal. Continuous-time (or analog) signals exist for the continuous interval (t1, t2) can range from −∞−∞ to +∞+∞

Basics of signal processing system

* Since computer needs digital signals for processing, therefore, in order to use an analog signal on a computer it must be digitized with an analog-to-digital converter. Thus, there is a need for an interface between the analog signal and the digital signal processor.



* **Sampling**: sampling is the reduction of a continuous-time signal to a discrete-time signal. A common example is the conversion of a sound wave (a continuous signal) to a sequence of samples (a discrete-time signal)
* **Quantization**: quantization is the process of mapping input values from large set (often continuous set) to output values in a (countable) smaller set, often with a finite number of elements. Rouding and Truncation are typical examples of quantization processes.
* **Encoding**: After each sample is quantized and the number of bits per sample is decided, each sample can be changed to an nb-bit code word. The number of bits for each sample is determined from the number of quantization levels. If the number of quantization levels is L, the number of bits is nb=log2.L

import numpy as np

import matplotlib.pyplot as plt

import scipy

from scipy import signal

In [2]:

t = np.arange(0, 11)

x = (0.85) \*\* t

Continuous Signal

In [3]:

plt.figure(figsize = (10,8)) *# set the size of figure*

*# 1. Plotting Analog Signal*

plt.subplot(2, 2, 1)

plt.title('Analog Signal', fontsize=20)

plt.plot(t, x, linewidth=3, label='x(t) = (0.85)^t')

plt.xlabel('t' , fontsize=15)

plt.ylabel('amplitude', fontsize=15)

plt.legend(loc='upper right')

*# 2. Sampling and Plotting of Sampled signal*

plt.subplot(2, 2, 2)

plt.title('Sampling', fontsize=20)

plt.plot(t, x, linewidth=3, label='x(t) = (0.85)^t')

n = t

markerline, stemlines, baseline = plt.stem(n, x, label='x(n) = (0.85)^n')

plt.setp(stemlines, 'linewidth', 3)

plt.xlabel('n' , fontsize = 15)

plt.ylabel('amplitude', fontsize = 15)

plt.legend(loc='upper right')

*# 3. Quantization*

plt.subplot(2, 2, 3)

plt.title('Quantization', fontsize = 20)

plt.plot(t, x, linewidth =3)

markerline, stemlines, baseline=plt.stem(n,x)

plt.setp(stemlines, 'linewidth', 3)

plt.xlabel('n', fontsize = 15)

plt.ylabel('Range of Quantizer', fontsize=15)

plt.axhline(y = 0.1, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.2, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.3, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.4, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.5, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.6, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.7, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.8, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.9, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 1.0, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.subplot(2, 2, 4)

plt.title('Quantized Signal', fontsize = 20)

xq = np.around(x,1)

markerline, stemlines, baseline = plt.stem(n,xq)

plt.setp(stemlines, 'linewidth', 3)

plt.xlabel('n', fontsize = 15)

plt.ylabel('Range of Quantizer', fontsize=15)

plt.axhline(y = 0.1, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.2, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.3, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.4, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.5, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.6, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

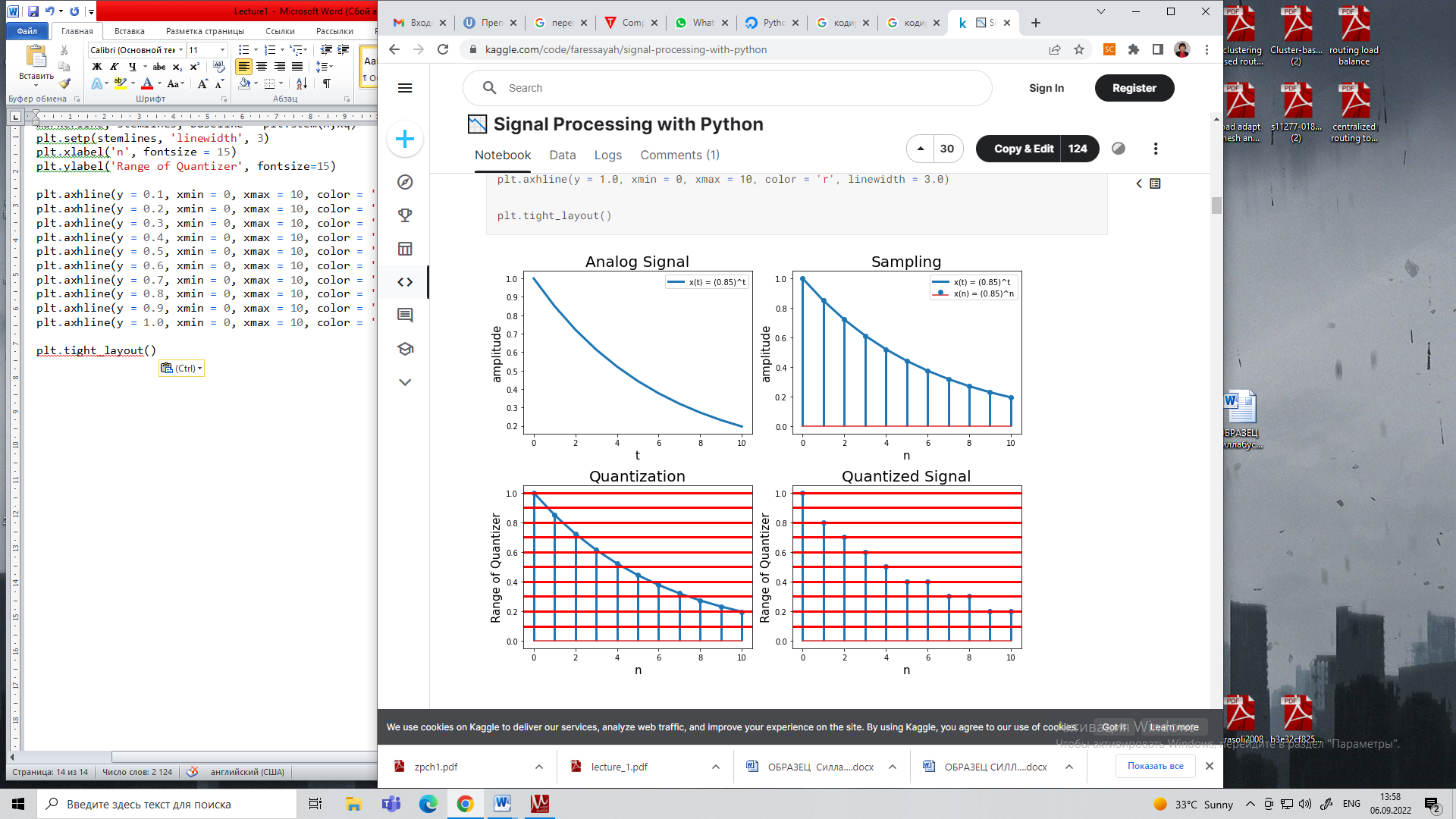
plt.axhline(y = 0.7, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.8, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 0.9, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.axhline(y = 1.0, xmin = 0, xmax = 10, color = 'r', linewidth = 3.0)

plt.tight\_layout()



# Unit Impulse Signal

In [4]:

impulse = signal.unit\_impulse(10, 'mid')

shifted\_impulse = signal.unit\_impulse(7, 2)

*# Sine wave*

t = np.linspace(0, 10, 100)

amp = 5 *# Amplitude*

f = 50

x = amp \* np.sin(2 \* np.pi \* f \* t)

*# Exponential Signal*

x\_ = amp \* np.exp(-t)

In [5]:

linkcode

plt.figure(figsize=(10, 8))

plt.subplot(2, 2, 1)

plt.plot(np.arange(-5, 5), impulse, linewidth=3, label='Unit impulse function')

plt.ylim(-0.01,1)

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 2)

plt.plot(shifted\_impulse, linewidth=3, label='Shifted Unit impulse function')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 3)

plt.plot(t, x, linewidth=3, label='Sine wave')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 4)

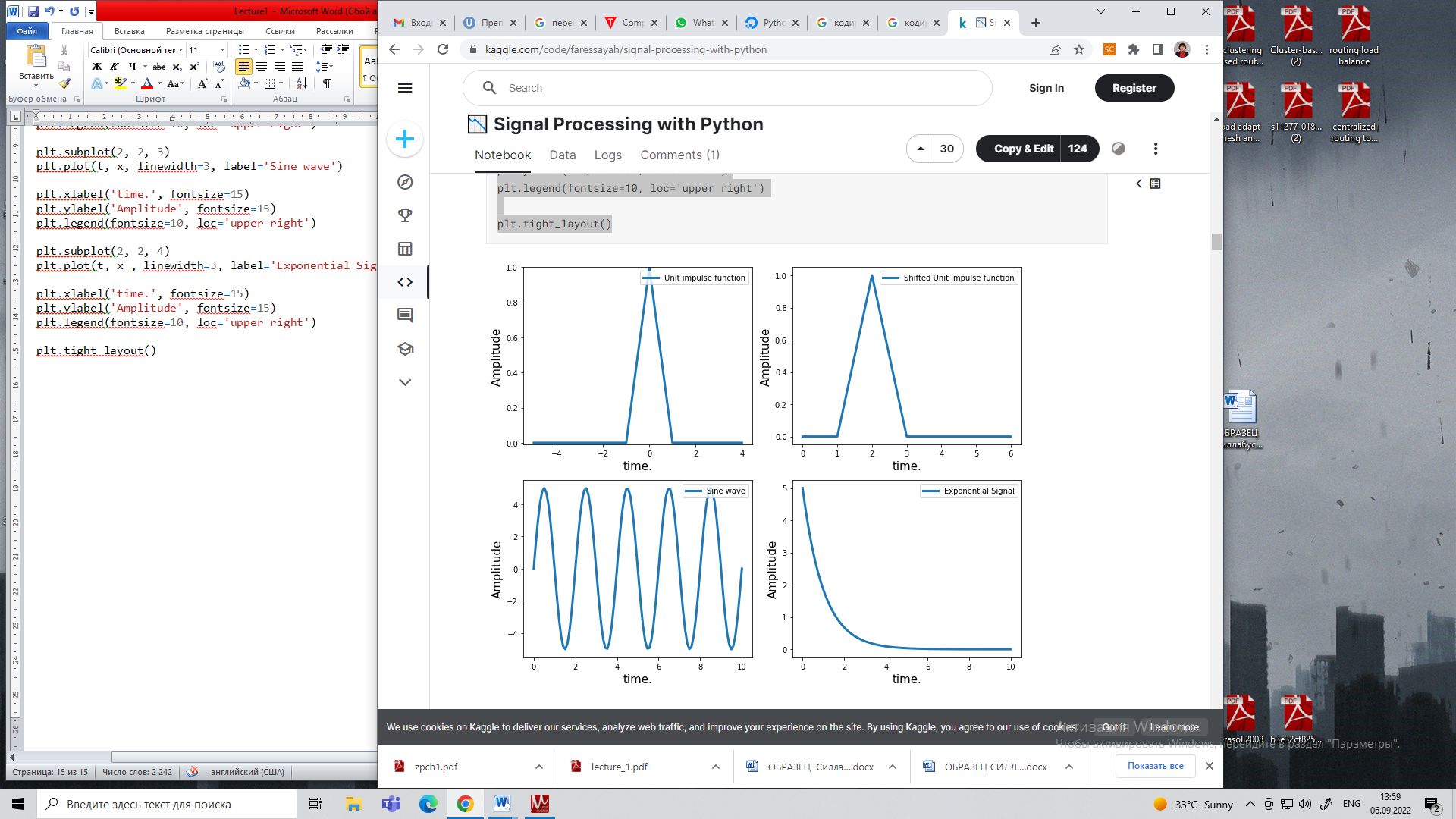
plt.plot(t, x\_, linewidth=3, label='Exponential Signal')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.tight\_layout()



Sine wave

In [6]:

*# Sine wave*

n = np.linspace(0, 10, 100)

amp = 5 *# Amplitude*

f = 50

x = amp \* np.sin(2 \* np.pi \* f \* n)

*# Exponential Signal*

x\_ = amp \* np.exp(-n)

Discrete Signals

In [7]:

plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)

plt.stem(n, x, 'yo', label='Sine wave')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 2)

plt.stem(n, x\_, 'yo', label='Exponential Signal')

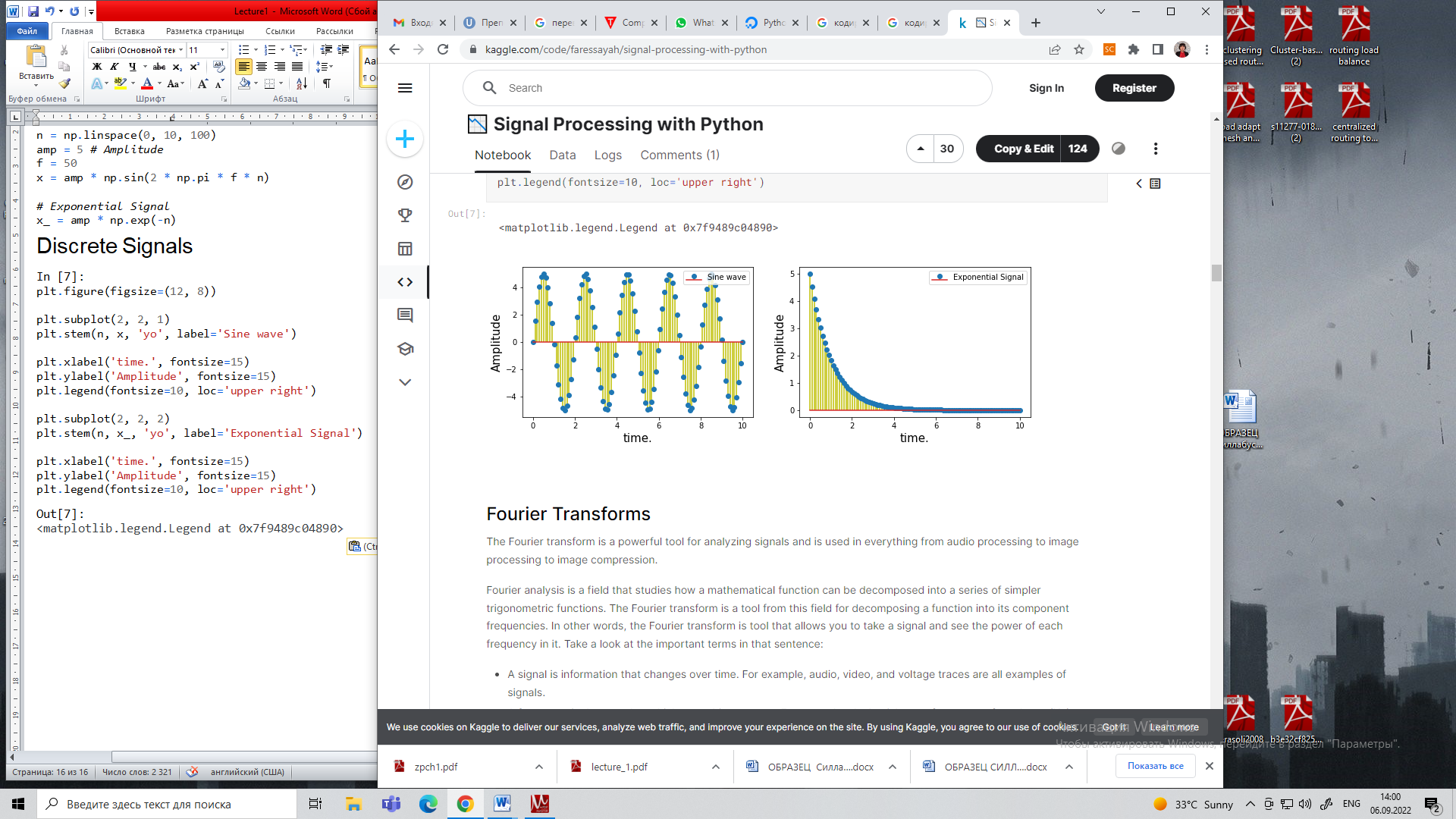
plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

Out[7]:

<matplotlib.legend.Legend at 0x7f9489c04890>



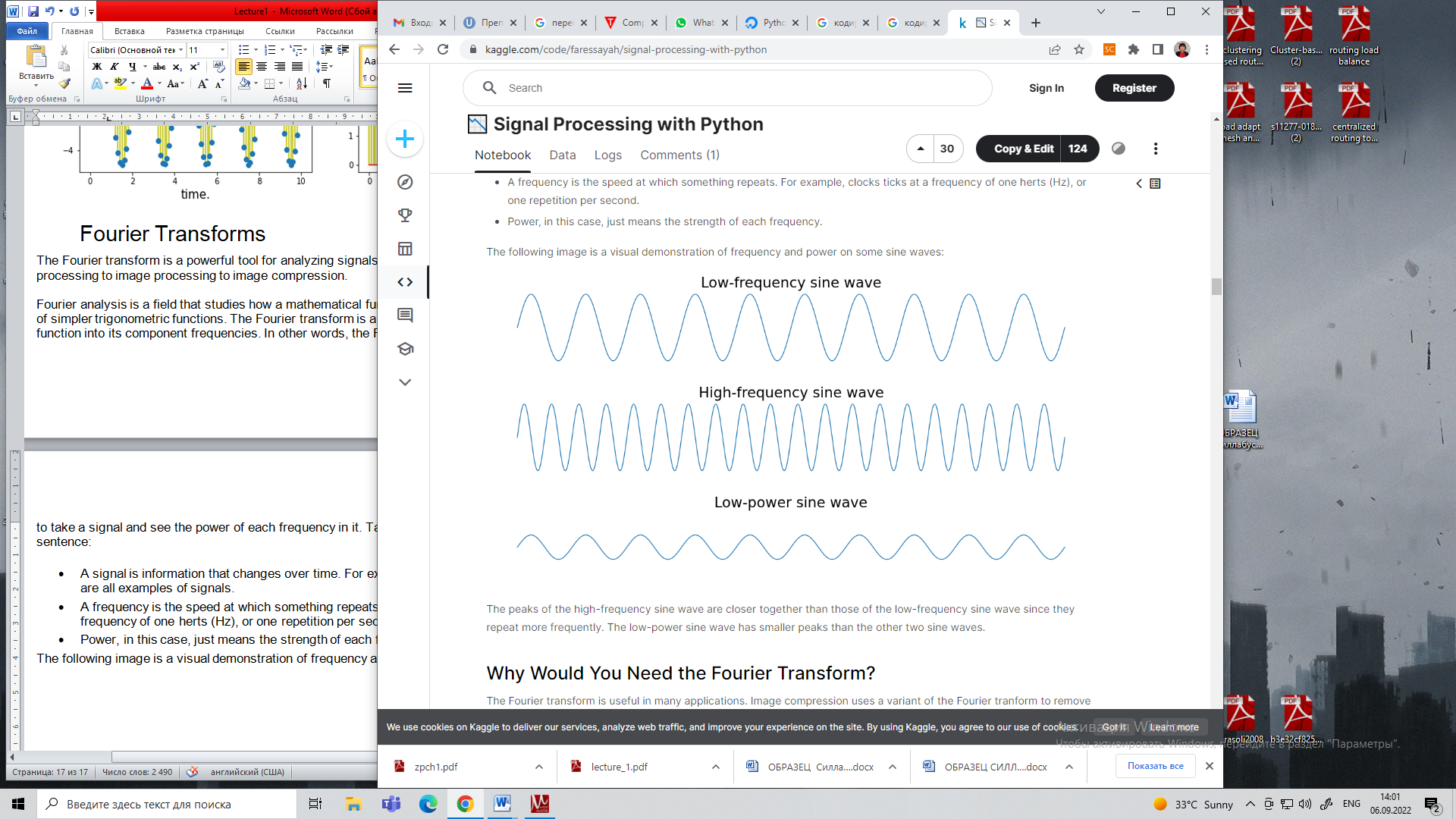
# Fourier Transforms

The Fourier transform is a powerful tool for analyzing signals and is used in everything from audio processing to image processing to image compression.

Fourier analysis is a field that studies how a mathematical function can be decomposed into a series of simpler trigonometric functions. The Fourier transform is a tool from this field for decomposing a function into its component frequencies. In other words, the Fourier transform is tool that allows you to take a signal and see the power of each frequency in it. Take a look at the important terms in that sentence:

* A signal is information that changes over time. For example, audio, video, and voltage traces are all examples of signals.
* A frequency is the speed at which something repeats. For example, clocks ticks at a frequency of one herts (Hz), or one repetition per second.
* Power, in this case, just means the strength of each frequency.

The following image is a visual demonstration of frequency and power on some sine waves:



# Why Would You Need the Fourier Transform?

The Fourier transform is useful in many applications. Image compression uses a variant of the Fourier tranform to remove the high-frequency components of images. Speech recognition uses the Fourier transform and related transforms to recover the spoken words from raw audio.

In general, you need the Fourier transform if you need to look at the frequencies in a signal. If working with a signal in the time domain is difficult, then using the Fourier transform to move it into the frequency domain is worth trying.

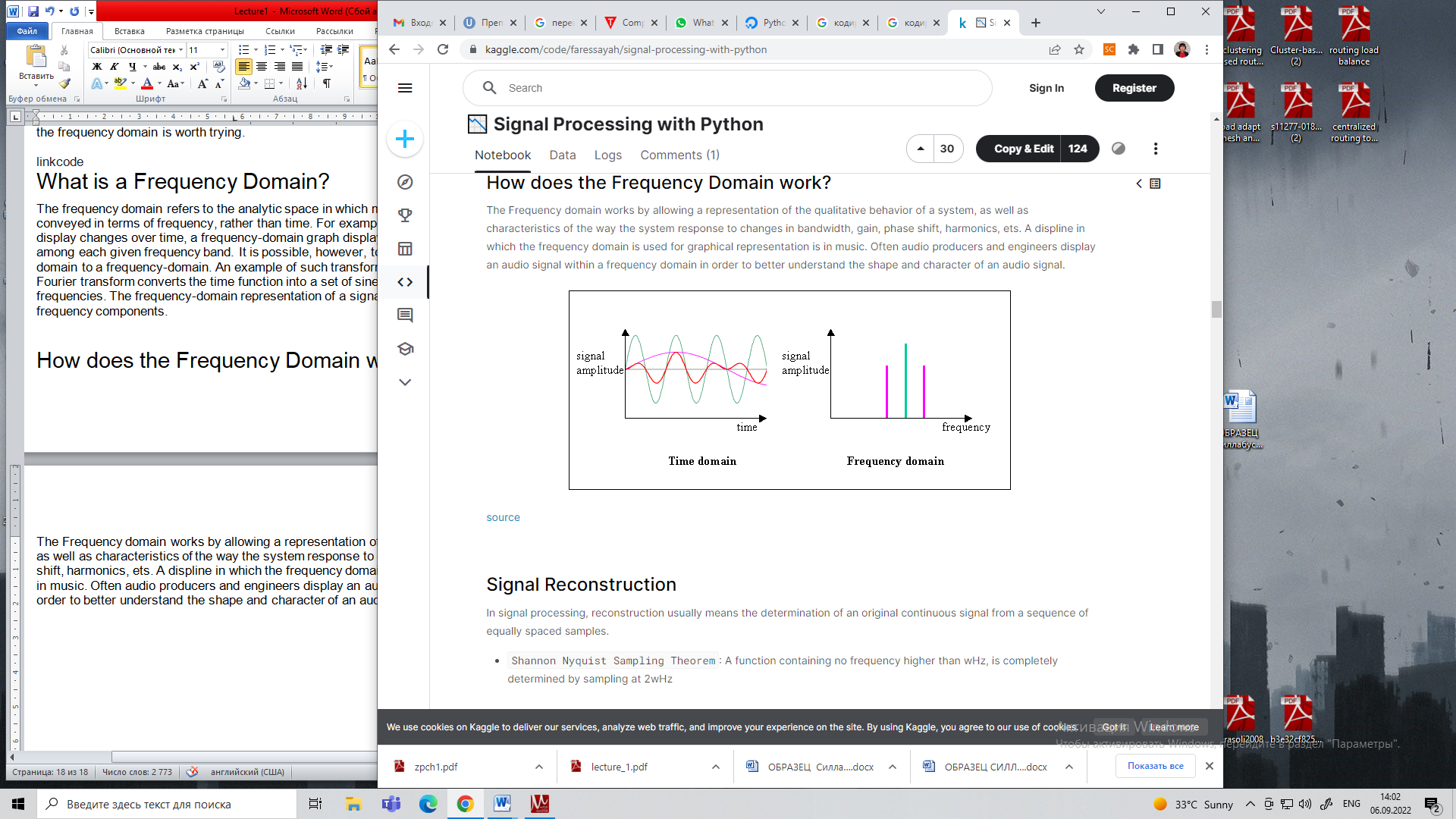
linkcode

What is a Frequency Domain?

The frequency domain refers to the analytic space in which mathematical functions or signals are conveyed in terms of frequency, rather than time. For example, where a time-domain graph may display changes over time, a frequency-domain graph displays how much of the signal is present among each given frequency band. It is possible, however, to convert the information from a time-domain to a frequency-domain. An example of such transformation is a Fourier transfrom. The Fourier transform converts the time function into a set of sine waves that represent different frequencies. The frequency-domain representation of a signal is known as the spectrum of frequency components.

How does the Frequency Domain work?

The Frequency domain works by allowing a representation of the qualitative behavior of a system, as well as characteristics of the way the system response to changes in bandwidth, gain, phase shift, harmonics, ets. A displine in which the frequency domain is used for graphical representation is in music. Often audio producers and engineers display an audio signal within a frequency domain in order to better understand the shape and character of an audio signal.



# Signal Reconstruction[¶](https://www.kaggle.com/code/faressayah/signal-processing-with-python#Signal-Reconstruction)

In signal processing, reconstruction usually means the determination of an original continuous signal from a sequence of equally spaced samples.

* Shannon Nyquist Sampling Theorem: A function containing no frequency higher than wHz, is completely determined by sampling at 2wHz

Nyquist sampling theorem

The Nyquist sampling theorem states that "The sampling frequency fs should be greater or equal than twice the maximum frequency of the signal (continuous time signal) to be sampled".

If Fmax is the maximum frequency of the signal then according to sampling theorem: fs >= 2Fmax

Sampling theorem is very important if we want to reconstruct the signal after sampling.

Sampling & Reconstruction

f = 20 *# Hz*

t = np.linspace(0, 0.5, 200)

x1 = np.sin(2 \* np.pi \* f \* t)

s\_rate = 35 *# Hz. Here the sampling frequency is less than the requirement of sampling theorem*

T = 1 / s\_rate

n = np.arange(0, 0.5 / T)

nT = n \* T

x2 = np.sin(2 \* np.pi \* f \* nT) *# Since for sampling t = nT.*

In [9]:

print(len(t))

print(len(nT))

200

18

In [10]:

linkcode

plt.figure(figsize=(10, 8))

plt.suptitle("Sampling a Sine Wave of Fmax=20Hz with fs=35Hz", fontsize=20)

plt.subplot(2, 2, 1)

plt.plot(t, x1, linewidth=3, label='SineWave of frequency 20 Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 2)

plt.plot(nT, x2, 'ro', label='Sample marks after resampling at fs=35Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 3)

plt.stem(nT, x2, 'm', label='Sample after resampling at fs=35Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 4)

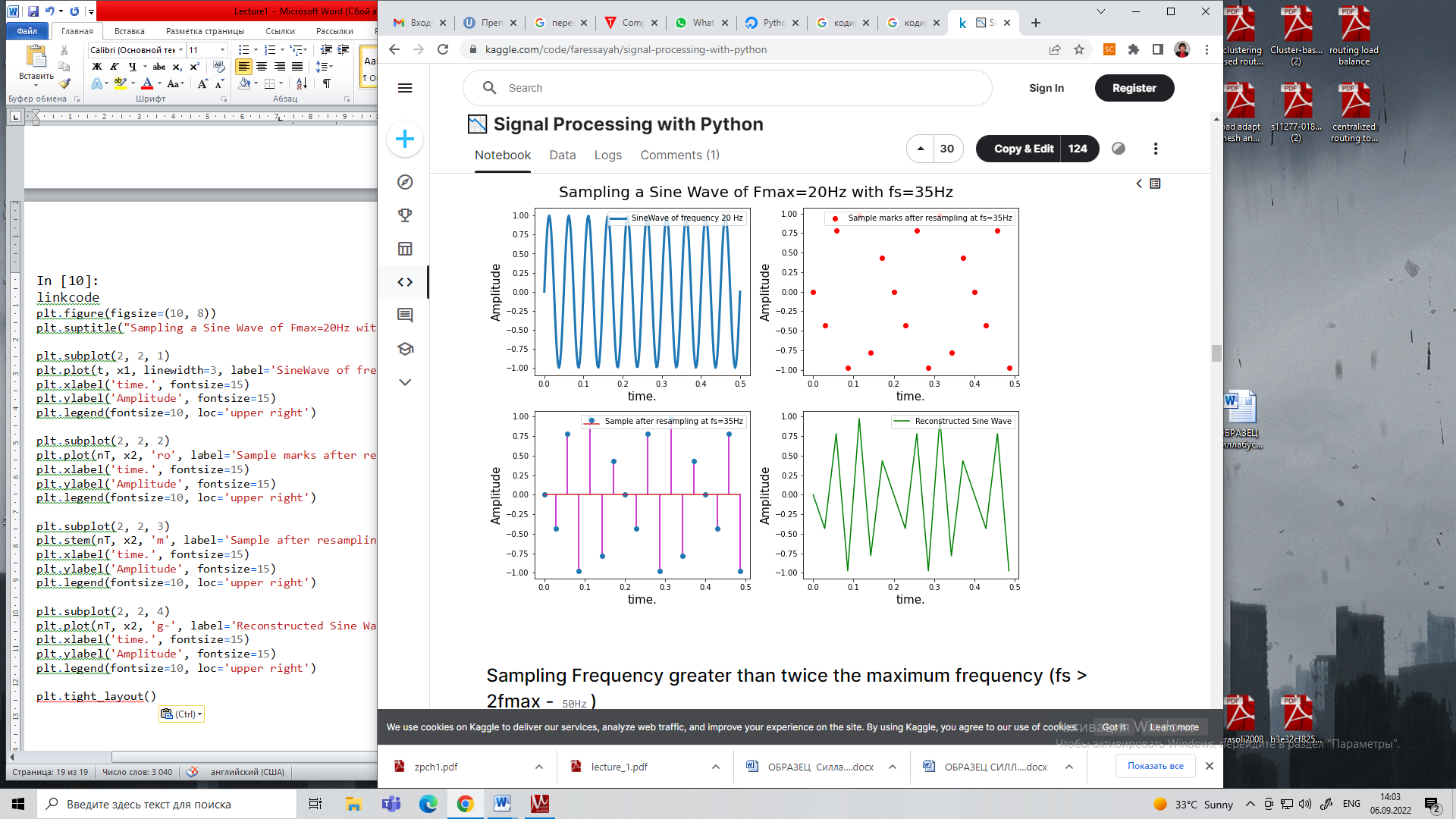
plt.plot(nT, x2, 'g-', label='Reconstructed Sine Wave')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.tight\_layout()



# Sampling Frequency greater than twice the maximum frequency (fs > 2fmax - 50Hz)

In [11]:

f = 20 *# Hz*

t = np.linspace(0, 0.5, 200)

x1 = np.sin(2 \* np.pi \* f \* t)

s\_rate = 50 *# Hz. Here the sampling frequency is less than the requirement of sampling theorem*

T = 1 / s\_rate

n = np.arange(0, 0.5 / T)

nT = n \* T

x2 = np.sin(2 \* np.pi \* f \* nT) *# Since for sampling t = nT.*

In [12]:

linkcode

plt.figure(figsize=(10, 8))

plt.suptitle("Sampling a Sine Wave of Fmax=20Hz with fs=50Hz", fontsize=20)

plt.subplot(2, 2, 1)

plt.plot(t, x1, linewidth=3, label='SineWave of frequency 20 Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 2)

plt.plot(nT, x2, 'ro', label='Sample marks after resampling at fs=50Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 3)

plt.stem(nT, x2, 'm', label='Sample after resampling at fs=50Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 4)

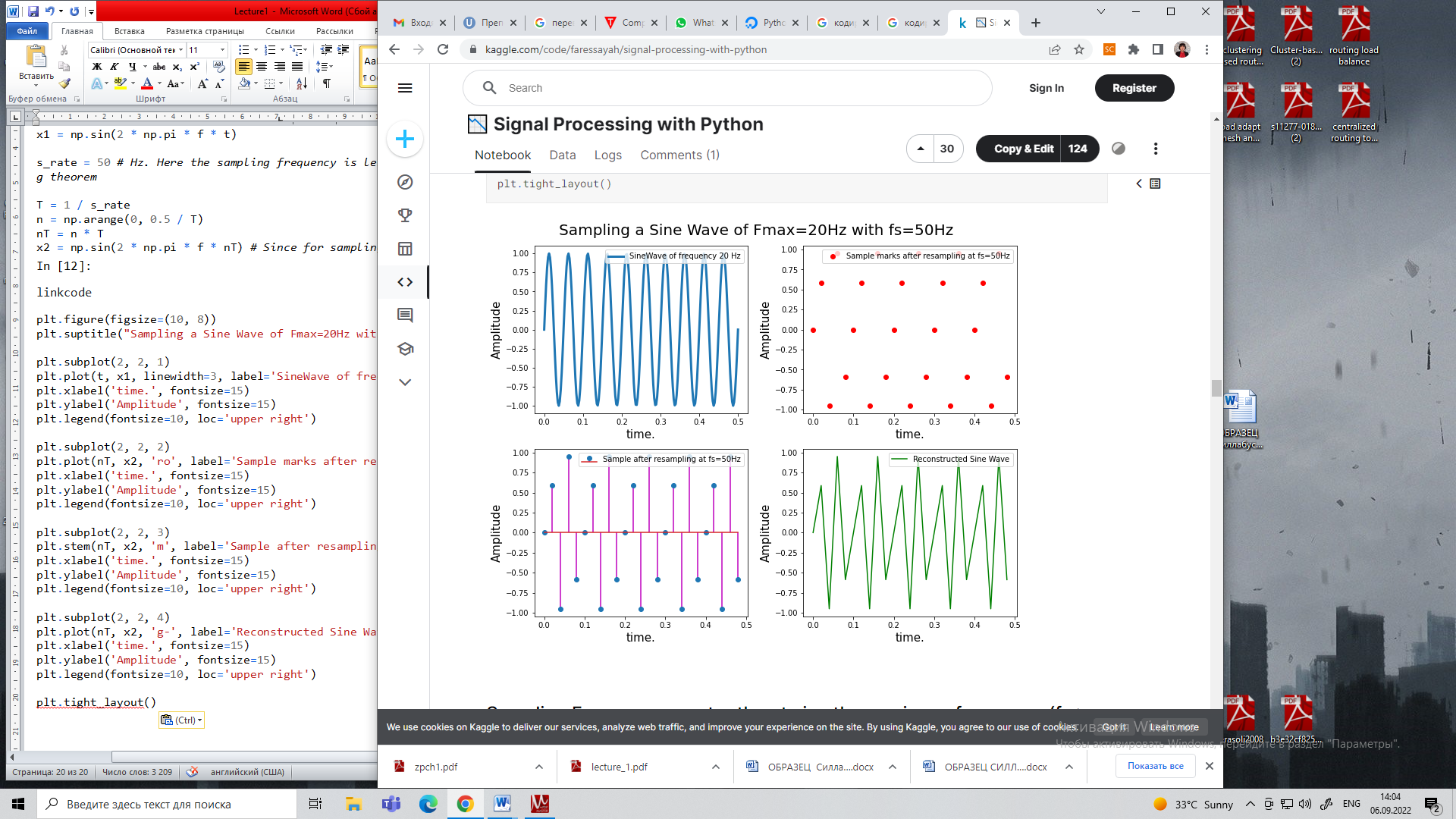
plt.plot(nT, x2, 'g-', label='Reconstructed Sine Wave')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.tight\_layout()



# Sampling Frequency greater than twice the maximum frequency (fs > 5fmax - 100Hz)

In [13]:

f = 20 *# Hz*

t = np.linspace(0, 0.5, 200)

x1 = np.sin(2 \* np.pi \* f \* t)

s\_rate = 100 *# Hz. Here the sampling frequency is less than the requirement of sampling theorem*

T = 1 / s\_rate

n = np.arange(0, 0.5 / T)

nT = n \* T

x2 = np.sin(2 \* np.pi \* f \* nT) *# Since for sampling t = nT.*

In [14]:

linkcode

plt.figure(figsize=(10, 8))

plt.suptitle("Sampling a Sine Wave of Fmax=20Hz with fs=100Hz", fontsize=20)

plt.subplot(2, 2, 1)

plt.plot(t, x1, linewidth=3, label='SineWave of frequency 20 Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 2)

plt.plot(nT, x2, 'ro', label='Sample marks after resampling at fs=100Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 3)

plt.stem(nT, x2, 'm', label='Sample after resampling at fs=100Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 4)

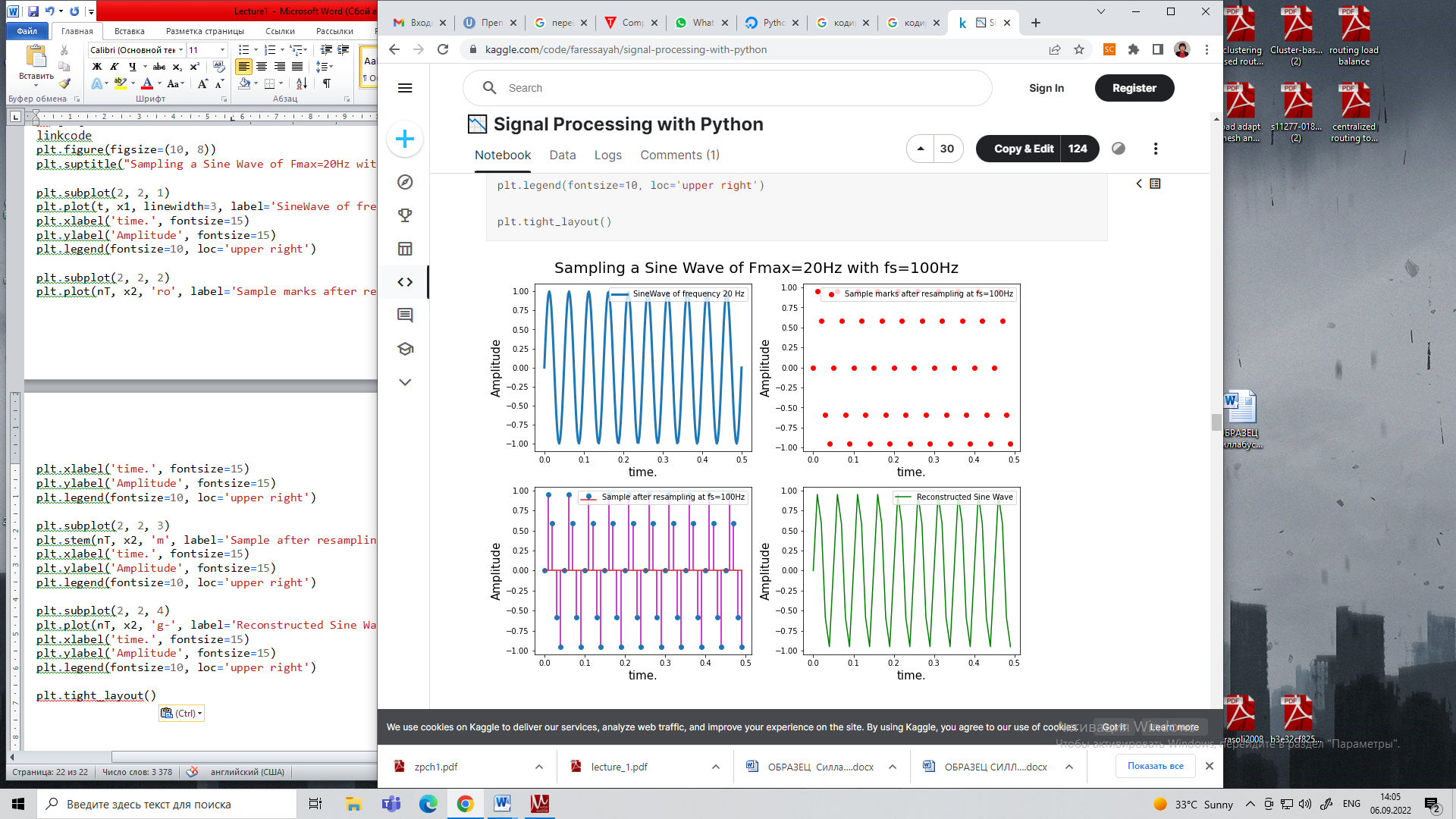
plt.plot(nT, x2, 'g-', label='Reconstructed Sine Wave')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.tight\_layout()



# Sampling and Reconstruction of sum of two sine waves

In [15]:

t = np.linspace(0, 0.5, 200)

x1 = 2 \* np.sin(2 \* np.pi \* 10 \* t) + np.sin(2 \* np.pi \* 20 \* t)

s\_rate = 100 *# Hz. Here the sampling frequency is less than the requirement of sampling theorem*

T = 1 / s\_rate

n = np.arange(0, 0.5 / T)

nT = n \* T

x2 = 2 \* np.sin(2 \* np.pi \* 10 \* nT) + np.sin(2 \* np.pi \* 20 \* nT) *# Since for sampling t = nT.*

In [16]:

linkcode

plt.figure(figsize=(10, 8))

plt.suptitle("Sampling a Two Sine Wave of Fmax=20Hz with fs=100Hz", fontsize=20)

plt.subplot(2, 2, 1)

plt.plot(t, x1, linewidth=3, label='SineWave of frequency 20 Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 2)

plt.plot(nT, x2, 'ro', label='Sample marks after resampling at fs=100Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 3)

plt.stem(nT, x2, 'm', label='Sample after resampling at fs=100Hz')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.subplot(2, 2, 4)

plt.plot(nT, x2, 'g-', label='Reconstructed Sine Wave')

plt.xlabel('time.', fontsize=15)

plt.ylabel('Amplitude', fontsize=15)

plt.legend(fontsize=10, loc='upper right')

plt.tight\_layout()

