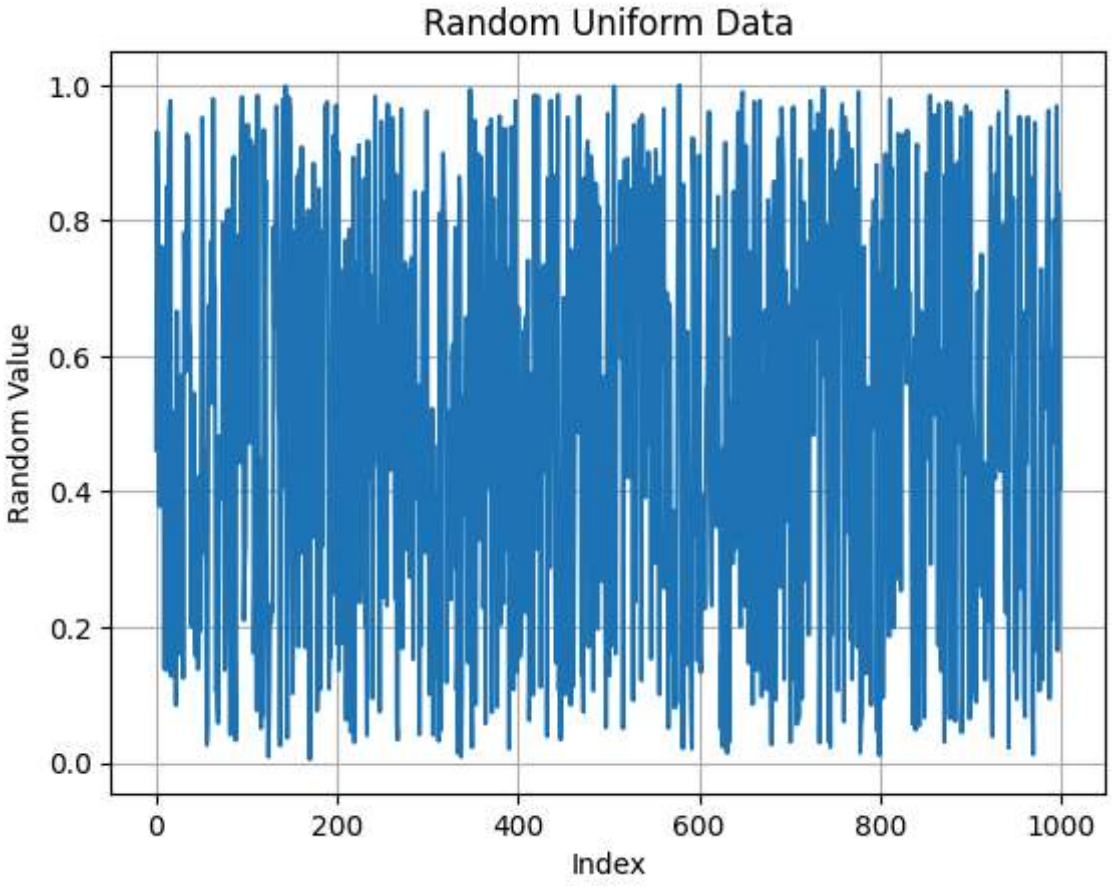
Homework 2 - DSP

submitted by

Aylon Feraru i.d: 325214492

Shay Bach i.d: 316147016

In [75]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.signal import lfilter
import scipy.signal
import dsp_hw1_py as hw1  # Importing the first homework's file converted to Python

hw1.Rect(1, 2, 60)


def Rand(N: int) -> np.ndarray:
    return np.random.uniform(0, 1, N)


random_data = Rand(1000)
hw1.basic_plot(range(1000), random_data, "Index", "Random Value", "Random Uniform D
```

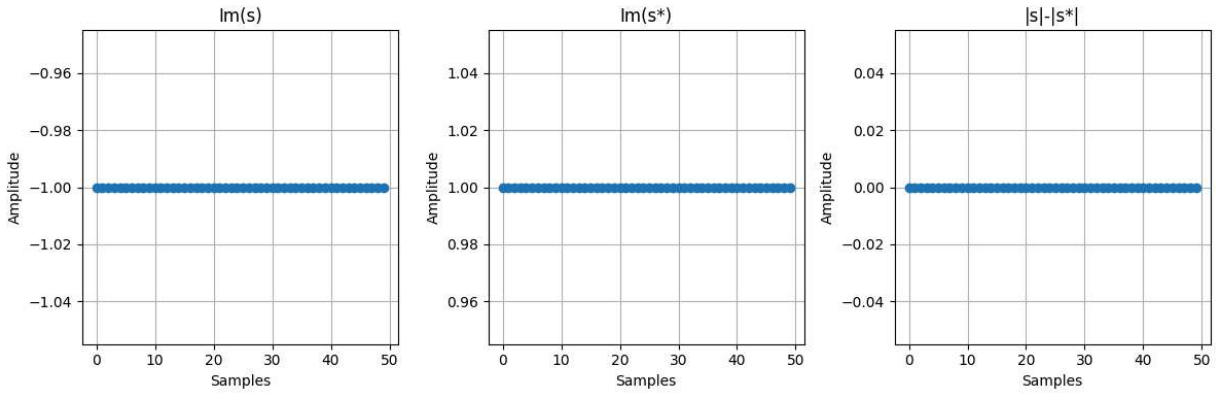


Uniform Explanation:

a uniform R.V is between $[0, 1]$, we have a vector of 1000 such samples

In [76]:
```
def Conj(x: np.ndarray) -> np.ndarray: #Conv in hw document
    y = np.conj(x)
    return y
```

In [77]:
```
signal = np.ones(50)*(-1j)
signal_conj = Conj(signal)
hw1.plot_three_side_by_side(np.arange(len(signal)),np.imag(signal), np.arange(len(s
```

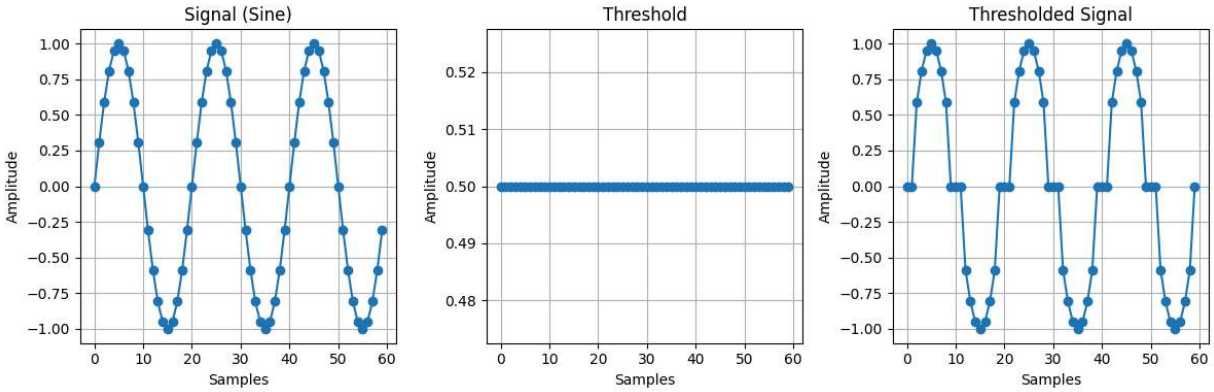


Conj Explanation:

We have a vector of $-j$, we look at its imaginary value, at its conjugate, and we verify that the magnitude did not change.

In [78]:
```
def Threshold(x: np.ndarray, th: float) -> np.ndarray:
    y = np.zeros_like(x)

    if x.ndim == 1:  # For 1D arrays
        for i in range(len(x)):
            if np.abs(x[i]) >= th:
                y[i] = x[i]
    elif x.ndim == 2:  # For 2D matrices
        for i in range(x.shape[0]):
            for j in range(x.shape[1]):
                if np.abs(x[i, j]) >= th:
                    y[i, j] = x[i, j]
    return y
```

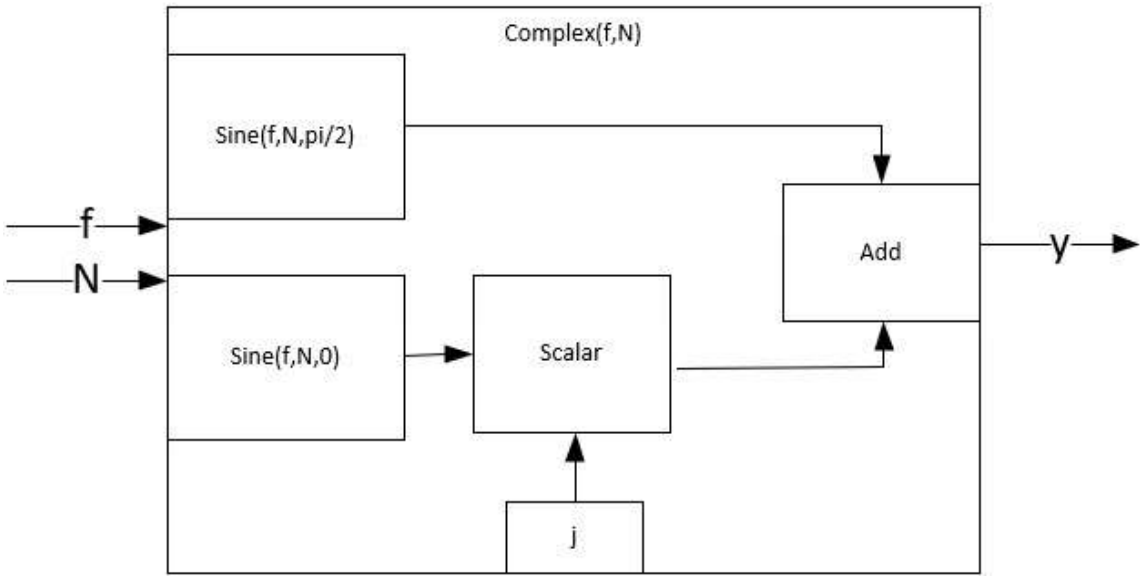
In [79]:
```
signal = hw1.Sine(3,60,0)
th = 0.5
hw1.plot_three_side_by_side(np.arange(60), signal, np.arange(60), th*np.ones(60), n
```



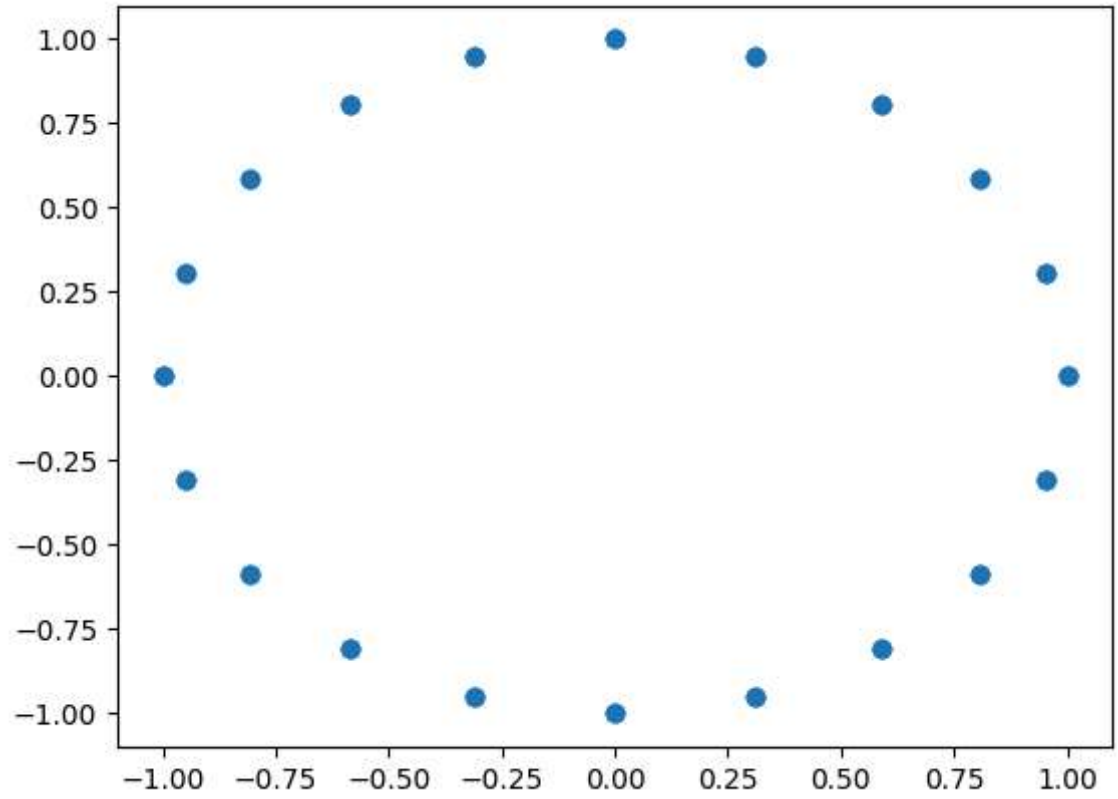
Threshold Explanation

A sine, when it's bigger than 0.5 (selected threshold) it retains its value, otherwise the value drops to 0

In [80]:
```
def Complex(f: float, N: int) -> np.ndarray:
    # Generate the complex signal
    real = hw1.Sine(f=f, N=N, ph=np.pi/2)  # Cosine component
    imaginary = hw1.Scalar(1j, hw1.Sine(f=f, N=N, ph=0))  # Sine component scaled b
    return hw1.Add(real, imaginary)
```



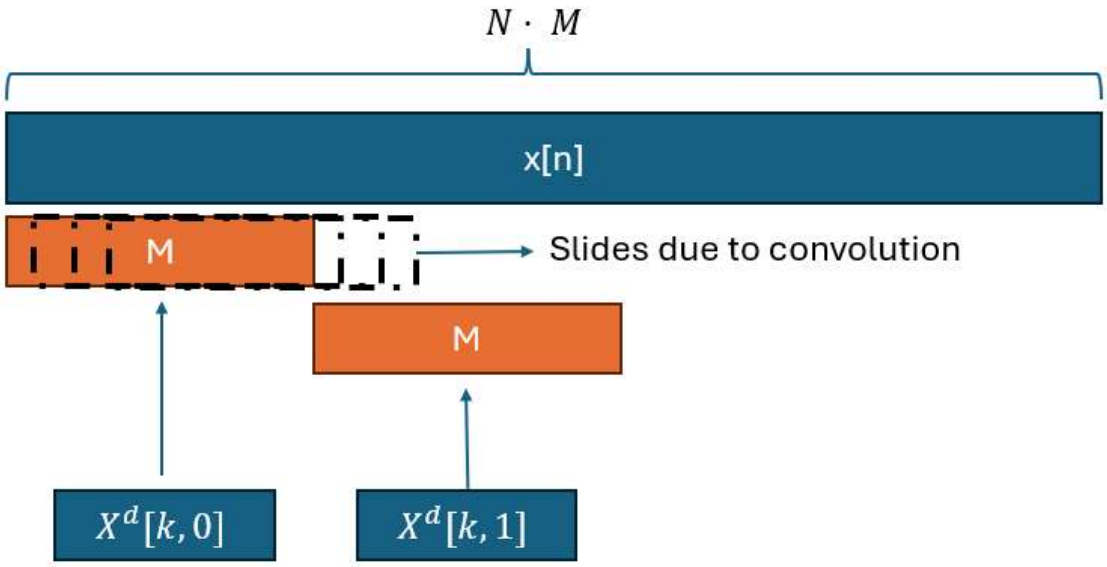
In [81]:
```
signal_c = Complex(f=3, N=60)
index = np.arange(60)
plt.scatter(signal_c.real,signal_c.imag)
hw1.plot_three_side_by_side(index, np.real(signal_c), index,
                            np.imag(signal_c), index,  Conj(signal_c).imag , titles
```



PART 2 - STFT

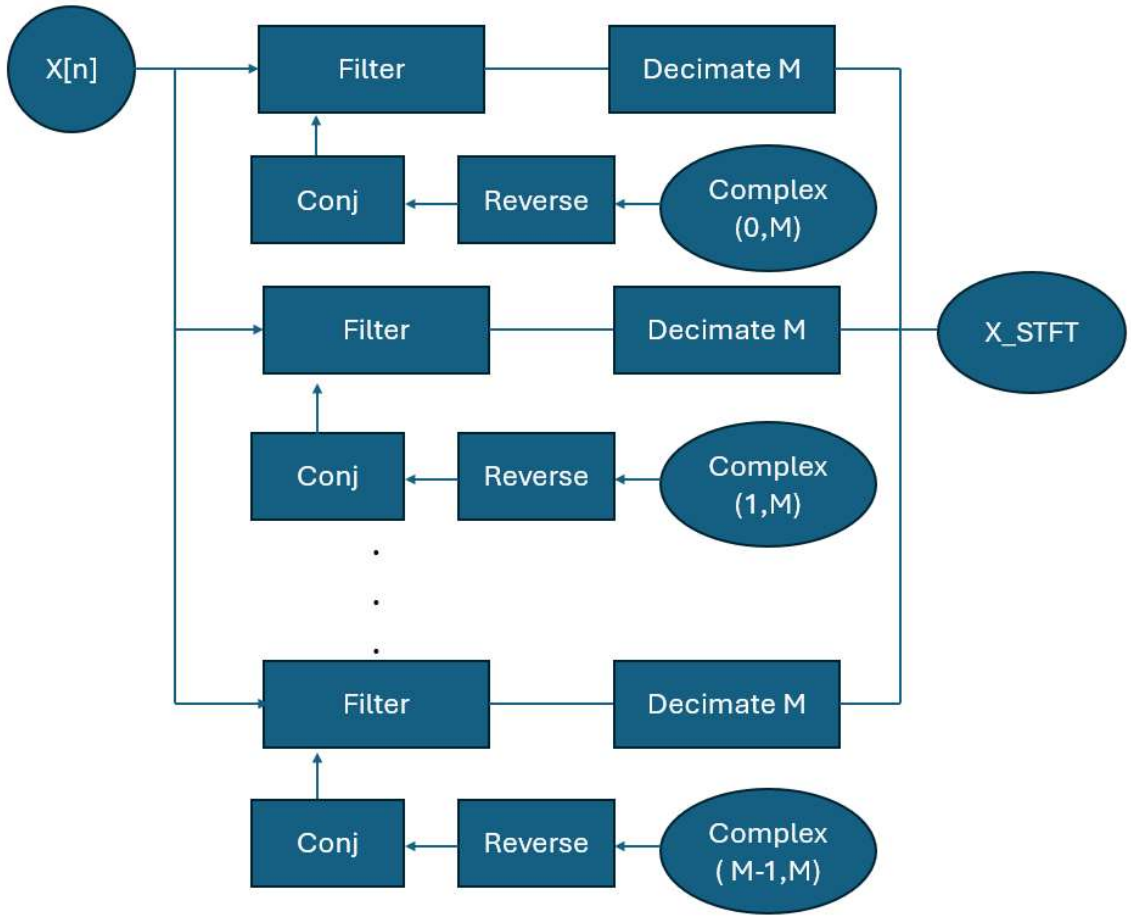
STFT explanation

In order to compute an STFT more efficiently we can look at convolving a Rectangle of size M and support N with the full signal,
multiply it by a complex twiddle with M in the denominator and decimate with a factor of $\downarrow M$.
This will give us for an exponent of frequency k_0, all the DFT points for all the windows that are related to said k_0. Why?



if we look at the first outcome of the valid convolution, we get $X^d[n=0,k_0] = \sum_{i=0}^{M-1} x[n]w[n]e^{-j\frac{2\pi k_0 m}{M}}$, this is the first sample of the DFT of the first window - and the first sample of the STFT's row.
since we're decimating with a factor of M, the next point we retain (after we discard $M-1$ points and the window has shifted $M-1$ positions) is the DFT of $x[n]w[n-M]$, or $X^d[n=1,k_0]$. We can see that as we continue that pattern, with one convolution we essentially get all the coefficients of the different DFTs for this particular frequency - or a single row of the STFT!
(sidenote: if we were to decimate by something smaller than the window size, we'd get built in overlap!).
so to get the entire STFT, all we have to do is the following block diagram:

STFT Block Diagram



Scipy's STFT (for reference)

In [82]:
```
signal = [1,2,3,4,5,6,7,8,9,10,-6,-42,2,100,2]
# STFT parameters
fs = 1  # Sampling frequency
nperseg = 5  # Window length
noverlap = 0  # No overlap
window = np.ones(nperseg)  # Rectangular window of ones
boundary = None  # No padding

# Compute the STFT using scipy.signal.stft
frequencies, times, Zxx = scipy.signal.stft(signal, fs=fs, window=window, nperseg=n
```

Reverse and Sampler Helper Functions:

Reverse

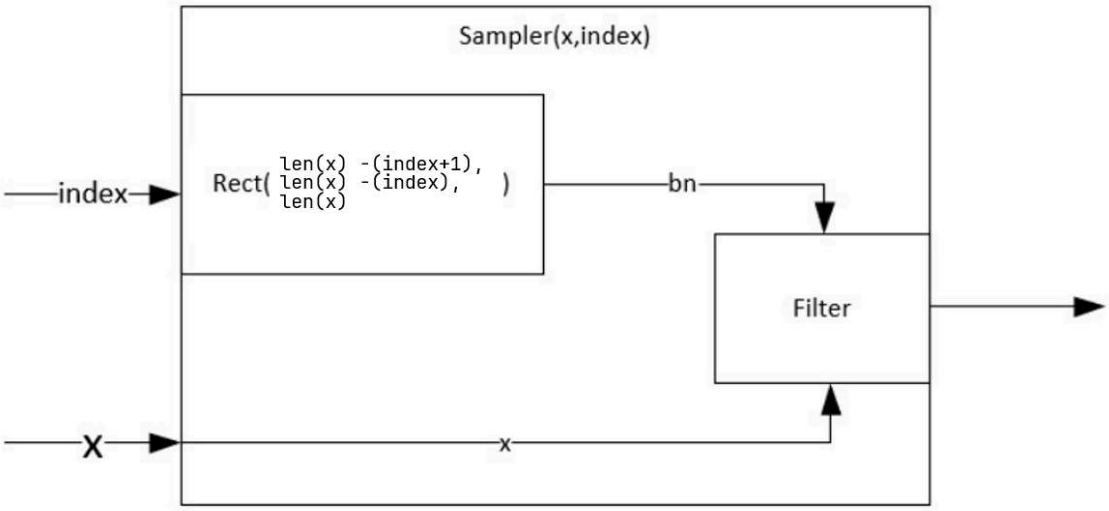
- given an array of size N that is an exponential, the function will return a reversed signal.
  How does it work?
  Conjugating a signal is the same as reversing it through time, multiplied by a constant phase.
  We conjugate the signal, and sample the complex exponential at a specific point to get that phase.

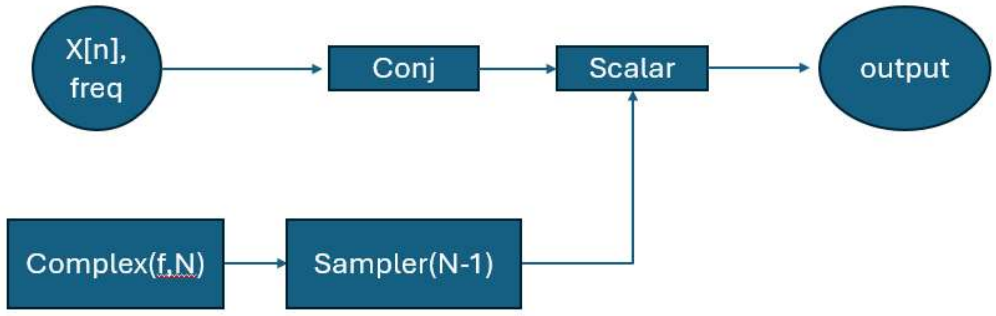
$$x[n] = e^{\frac{-j2\pi f}{N}n} \quad n = [0, 1, \ldots, N-1]$$

$$x_{\text{reversed}}[n] = \underbrace{\text{Conj}(x[n])}_{e^{j\frac{2\pi fn}{N}}} \cdot \underbrace{\text{Sampler}(x, N-1)}_{e^{-j\frac{2\pi f(N-1)}{N}}} = e^{-j\frac{2\pi f}{N}(N-1-n)} = e^{\frac{-j2\pi f}{N}k}$$

where $k = [N-1, N-2, \ldots, 0]$ essentially we got a time reversal.

Reverse_c Works the same as reverse using the same mathematics, but assumes an input of a conjugate instead of a regular Complex function, and applies conjugate inside the matrix. A really minor change so I did not add a block diagram. **Sampler**: to sample a signal at a specific point, we correlate it with a signal of the same length that contains a unit impulse at the specific point we desire.
Since convolution is correlation with a flipped signal we choose Rect's coefficients such that it'll be $\delta[N-1-n]$, and after the flip $\delta[n]$





In [83]:
```
def Sampler(x:np.ndarray, index: int)-> float:
    if index > len(x):
        raise ValueError("Index must be less than the length of the input signal")

    sampling_rect = hw1.Rect(len(x) -(index+1),len(x) -(index),len(x))
```

```
    return hw1.Filter([1],bn=sampling_rect,x=x)

def Reverse(exponential_signal: np.ndarray, frequency: int) -> np.ndarray:
    N = len(exponential_signal)
    return hw1.Scalar(Conj(exponential_signal), Sampler(exponential_signal,N-1))
def Reverse_c(exponential_signal: np.ndarray, frequency: int) -> np.ndarray:
    N = len(exponential_signal)
    return hw1.Scalar(Conj(exponential_signal), Sampler(Conj(Complex(frequency,N)),

print("===Reverse Test===")
print(Complex(3,7))
print(Reverse(Complex(3,7),3))
print("checking equivalence to [::-1]")
print(Complex(3,5) -Reverse(Complex(3,5),3)[::-1] )
```

```
===Reverse Test===
[ 1.      +0.j       -0.900969+0.433884j  0.62349 -0.781831j
 -0.222521+0.974928j -0.222521-0.974928j  0.62349 +0.781831j
 -0.900969-0.433884j]
[-0.900969-0.433884j  0.62349 +0.781831j -0.222521-0.974928j
 -0.222521+0.974928j  0.62349 -0.781831j -0.900969+0.433884j
  1.      -0.j      ]
checking equivalence to [::-1]
[-0.+0.j  0.+0.j  0.-0.j -0.+0.j  0.+0.j]
```

STFT Implementation:

In [84]:
```
def STFT(x: np.ndarray, M: int) -> np.ndarray:
    N = len(x) // M
    stft_matrix = np.zeros((M, N), dtype=complex)
    for f in range(M):
        exponential = Conj(Complex(f, M))
        correlated_signal = hw1.Filter([1],x, Reverse(exponential,f))
        stft_matrix[f, :] = hw1.Decimate(M,correlated_signal)
    return stft_matrix
```

ISTFT Explanation:

ISTFT is very similar to STFT, and is- in a naive implementation - the IDFT of every column/time index. if we look at the definition of the IDFT, for a particular section in time t:

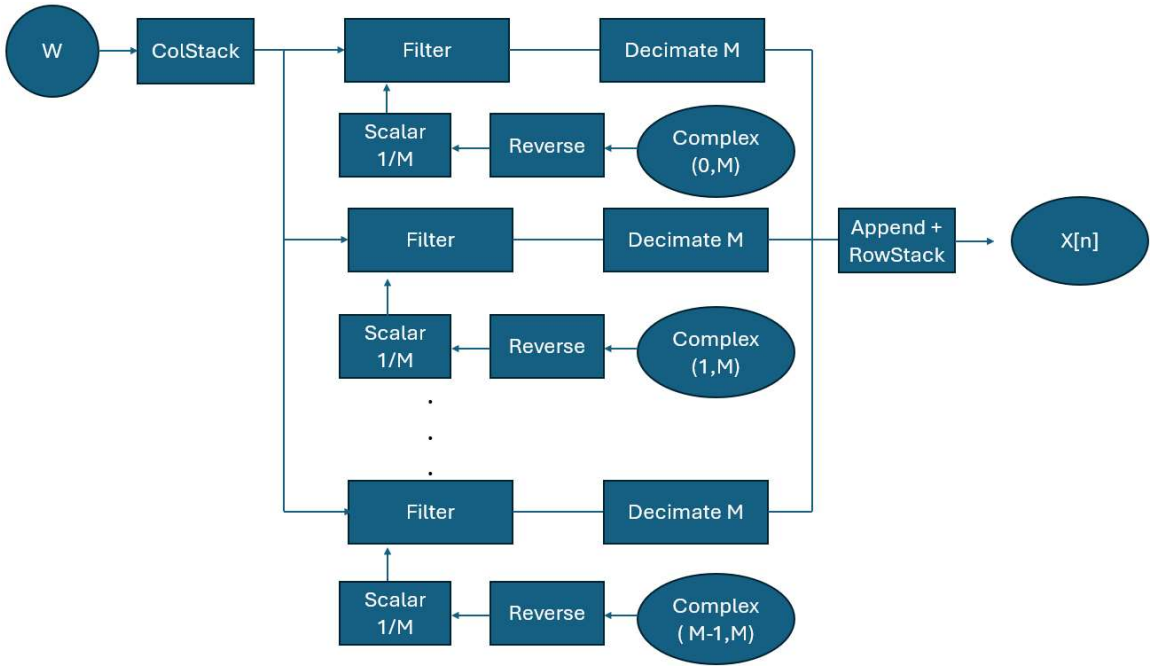
$$x_t[n] = \frac{1}{N}\sum_{k=0}^{N-1} X_t^d[k]e^{\frac{2j\pi kn}{N}}$$

for a particular time index n, we want to sum over k frequency points with the exponent, for that time.
for an efficient implementation, instead of treating each block seperately we will column stack our input matrix W such that we'll get:

$$[\underbrace{X_0^d[M-1], X_0^d[M-2], \ldots, X_0^d[0]}_{M}, \underbrace{X_1^d[M-1], \ldots, X_1^d[M-2]}_{M}, \ldots]$$

this is a vector of length $N \cdot M$. Now we notice that in this ordering, very similarly to STFT, if we iterate over every n, and decimate and normalize by a factor of of M (sliding window with discarding, similar to the drawing above), without conjugating, we'd definitionally get our original signal back.

ISTFT Block Diagram:



ISTFT Implementation:

In [85]:
```
def ISTFT(W: np.ndarray) -> np.ndarray:
    M, N = W.shape
    X_colstack = np.matrix.flatten(W, 'F')  # Flatten in column-major order
    x_matrix = np.zeros((N,M), dtype=complex)

    for n in range(M):
        exponential = hw1.Scalar(Complex(n, M),1/M) # Added conjugate here
        correlated_signal = hw1.Filter([1], X_colstack, Reverse(exponential,n))  #
        x_matrix[:,n] = hw1.Decimate(M, correlated_signal)

    return np.matrix.flatten(x_matrix)  # Added real part extraction
```

STFT and ISTFT check:

In [86]:
```
print(len(signal))

print("MY STFT:")
print(STFT(signal,M=5))  # Transpose for matching dimensions

print("\nSCIPY's STFT:") # Print the magnitude from scipy's STFT
print(Zxx*5)  # Print the magnitude from scipy's STFT
```

```
print("the original signal:\n" ,signal)
print("ISTFT(STFT(signal)): \n",(ISTFT(STFT(signal,M=5)))) #abs and int are there f
```

```
15
MY STFT:
[[  15.       +0.j         40.       +0.j         56.       +0.j      ]
 [  -2.5      +3.440955j   -2.5      +3.440955j -100.880413+99.449441j]
 [  -2.5      +0.812299j   -2.5      +0.812299j   57.880413-67.340987j]
 [  -2.5      -0.812299j   -2.5      -0.812299j   57.880413+67.340987j]
 [  -2.5      -3.440955j   -2.5      -3.440955j -100.880413-99.449441j]]

SCIPY's STFT:
[[  15.       +0.j         40.       +0.j         56.       +0.j      ]
 [  -2.5      +3.440955j   -2.5      +3.440955j -100.880413+99.449441j]
 [  -2.5      +0.812299j   -2.5      +0.812299j   57.880413-67.340987j]
 [  -2.5      -0.812299j   -2.5      -0.812299j   57.880413+67.340987j]
 [  -2.5      -3.440955j   -2.5      -3.440955j -100.880413-99.449441j]]
the original signal:
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -6, -42, 2, 100, 2]
ISTFT(STFT(signal)):
 [ 0.2+0.j  0.4+0.j  0.6-0.j  0.8-0.j  1. +0.j  1.2+0.j  1.4+0.j  1.6-0.j
  1.8-0.j  2. +0.j -1.2+0.j -8.4+0.j  0.4+0.j 20. +0.j  0.4+0.j]
```

ALTERNATE PART 2 - AN INEFFICIENT STFT PROGRAM

I kept it in because I think it's an intuitve and natural way of realizing STFT, despite being horribly inefficient I didn't want to discard the code completely.

Complex Explanation:

the complex function works by Add-ing a cosine generated with Sine, and a Sine generated with Sine multiplied by a Scalar j. We can see that the real and imaginary components correspond to Euler's formula, and we can see that Conj does work on the imaginary part as it did with any complex signal.

- additionally, we can see that the signal has magnitude one and a changing phase since it orbits in the unit circle as can be seen in the plot above.

In [87]:
```
def Take_Range(x:np.ndarray, a:int,b:int) -> np.ndarray:
    output = []
    for i in range(a,b):
        output.append(Sampler(x,i))
    return np.array(output).T


def DFT(x:np.ndarray) -> np.ndarray:
    N = len(x)
    summation_rect = hw1.Scalar(hw1.Rect(0,N,N),N)

    X_dft =np.zeros(N,dtype=complex)
    for k in range(N):
```

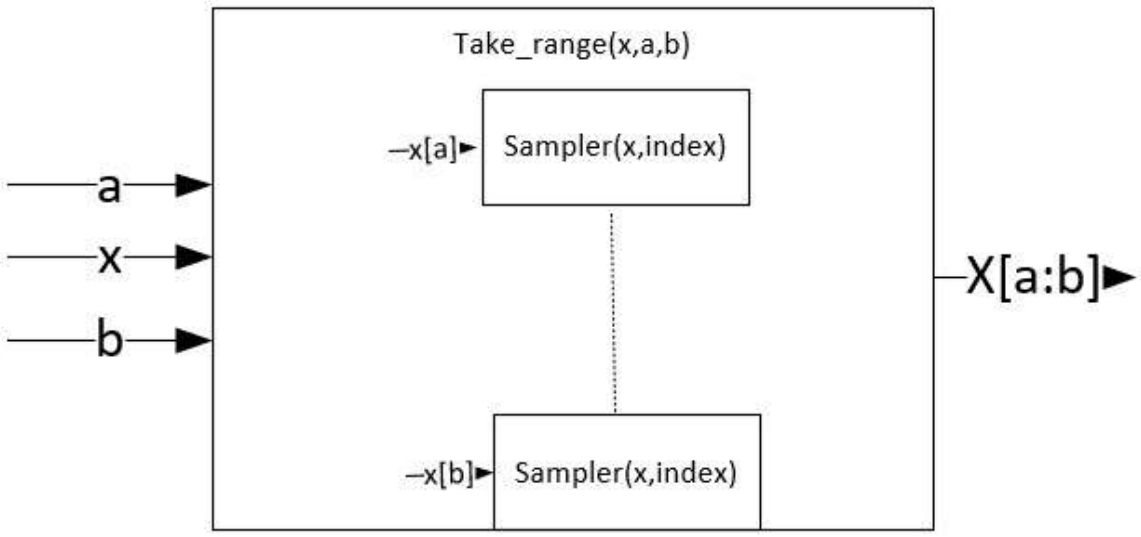
```
        exponent = Conj(Complex(k,N))
        x_comp =hw1.Prod(exponent,x)
        X_dft[k] = hw1.Filter([1],summation_rect,x_comp)
    return X_dft


signal = np.array([3,4,5,6,7,6,5,4,3],dtype=complex)
np.set_printoptions(precision=6, suppress=True)
print("DFT(signal):\n",DFT(signal))
print("Numpy's FFT: \n",np.fft.fft(signal))
```

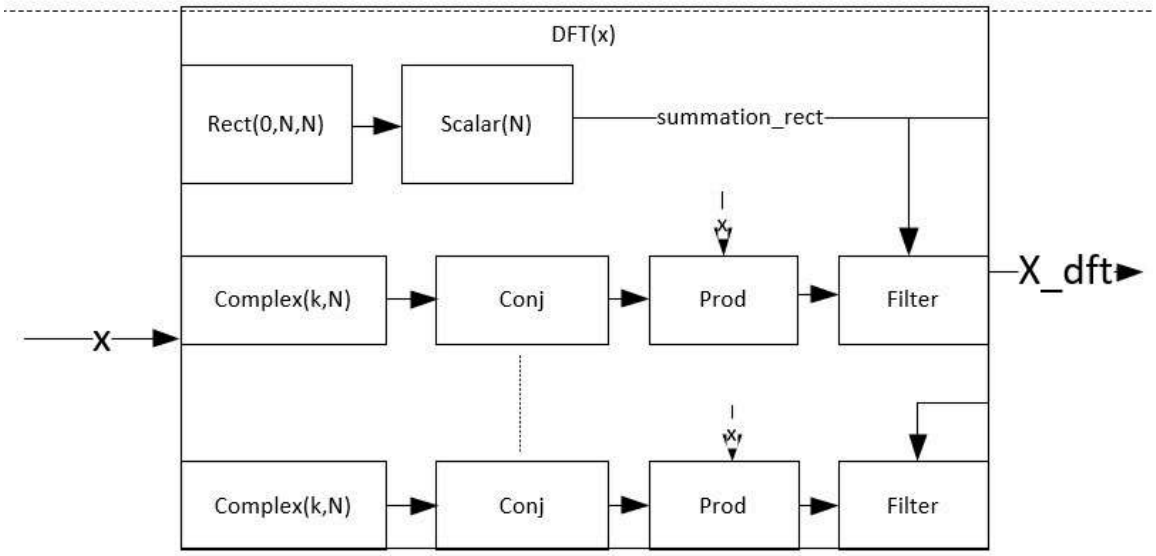
```
DFT(signal):
 [43.      +0.j       -7.790859-2.835641j  0.216881+0.181985j
 -0.5     -0.866025j  0.073978+0.41955j   0.073978-0.41955j
 -0.5     +0.866025j  0.216881-0.181985j -7.790859+2.835641j]
Numpy's FFT:
 [43.      +0.j       -7.790859-2.835641j  0.216881+0.181985j
 -0.5     -0.866025j  0.073978+0.41955j   0.073978-0.41955j
 -0.5     +0.866025j  0.216881-0.181985j -7.790859+2.835641j]
```

Original Helper Functions: Sampler, Take_Range, DFT - Explanation

- we sample a vector x at index i by convolving a window of legnth 1 at that point with our signal
- we repeat this action and append to get a Range
- DFT is just a helper function, since every FFT we take a slice of the signal and perform multiplication with an exponent on it, and then summation

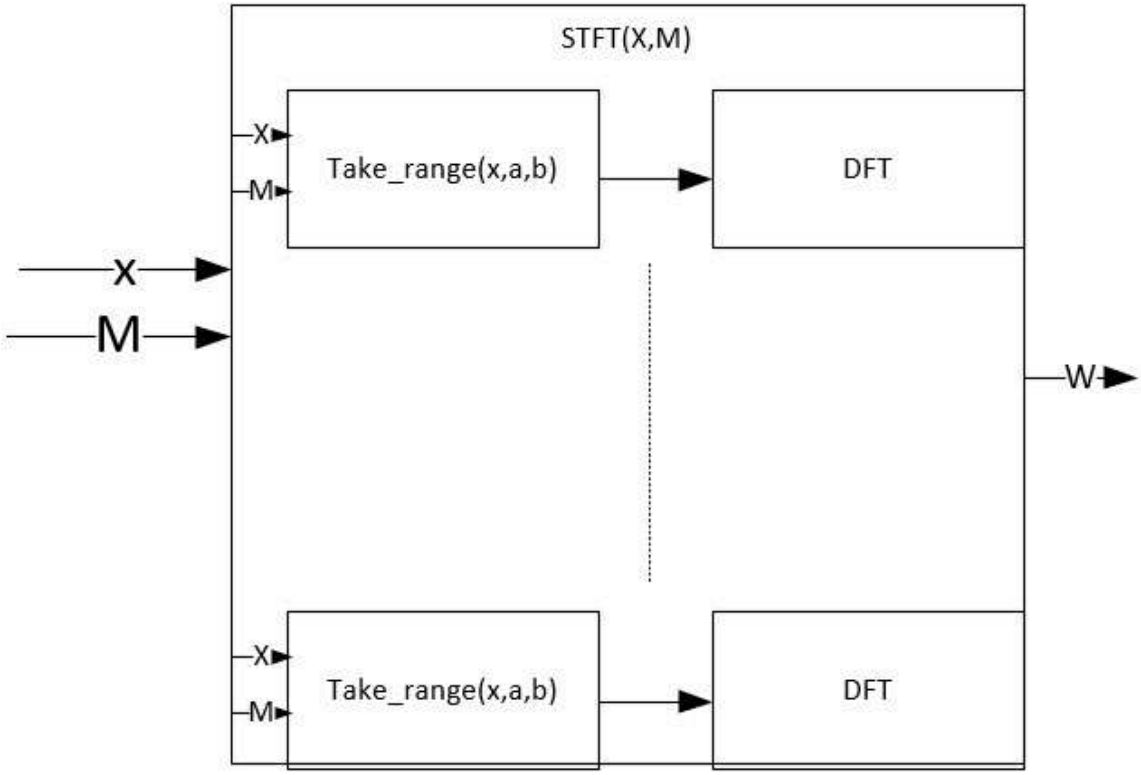
$$X^d[k] = \sum_{i=0}^{N-1} x[n] e^{\frac{-2j\pi kn}{N}}$$





In [88]:
```
import scipy.signal

def STFT_inefficient(x: np.ndarray, M: np.ndarray) -> np.ndarray:
    N = len(x)//M
    x_i_F = np.array([],dtype=complex)
    matrix =[]
    for i in range(N):
        x_i = Take_Range(x,i*M,(i+1)*M)
        x_i_F =DFT(x_i)
        matrix.append(x_i_F)
    return np.array(matrix, dtype=complex).T


# Define your signal
signal = np.array([0, 2, 2, 2, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4, 4])

# STFT parameters
fs = 1  # Sampling frequency
nperseg = 5  # Window length
noverlap = 0  # No overlap
window = np.ones(nperseg)  # Rectangular window of ones
boundary = None  # No padding

# Compute the STFT using scipy.signal.stft
frequencies, times, Zxx = scipy.signal.stft(signal, fs=fs, window=window, nperseg=n

# Set print options globally for numpy
np.set_printoptions(precision=6, suppress=True)  # Limit precision, suppress scient

# Print the results with consistent formatting

print("MY STFT:")
print(STFT_inefficient(signal,M=5))  # Transpose for matching dimensions

print("\nSCIPY's STFT:") # Print the magnitude from scipy's STFT
print(Zxx*5)  # Print the magnitude from scipy's STFT
```

```
MY STFT:
[[ 9.      +0.j       24.      +0.j       23.      +0.j      ]
 [-1.690983+0.951057j -1.      +0.j        0.5     -1.538842j]
 [-2.809017+0.587785j -1.      -0.j        0.5     +0.363271j]
 [-2.809017-0.587785j -1.      -0.j        0.5     -0.363271j]
 [-1.690983-0.951057j -1.      -0.j        0.5     +1.538842j]]

SCIPY's STFT:
[[ 9.      +0.j       24.      +0.j       23.      +0.j      ]
 [-1.690983+0.951057j -1.      +0.j        0.5     -1.538842j]
 [-2.809017+0.587785j -1.      +0.j        0.5     +0.363271j]
 [-2.809017-0.587785j -1.      -0.j        0.5     -0.363271j]
 [-1.690983-0.951057j -1.      -0.j        0.5     +1.538842j]]
```



ALTERNATE STFT explanation

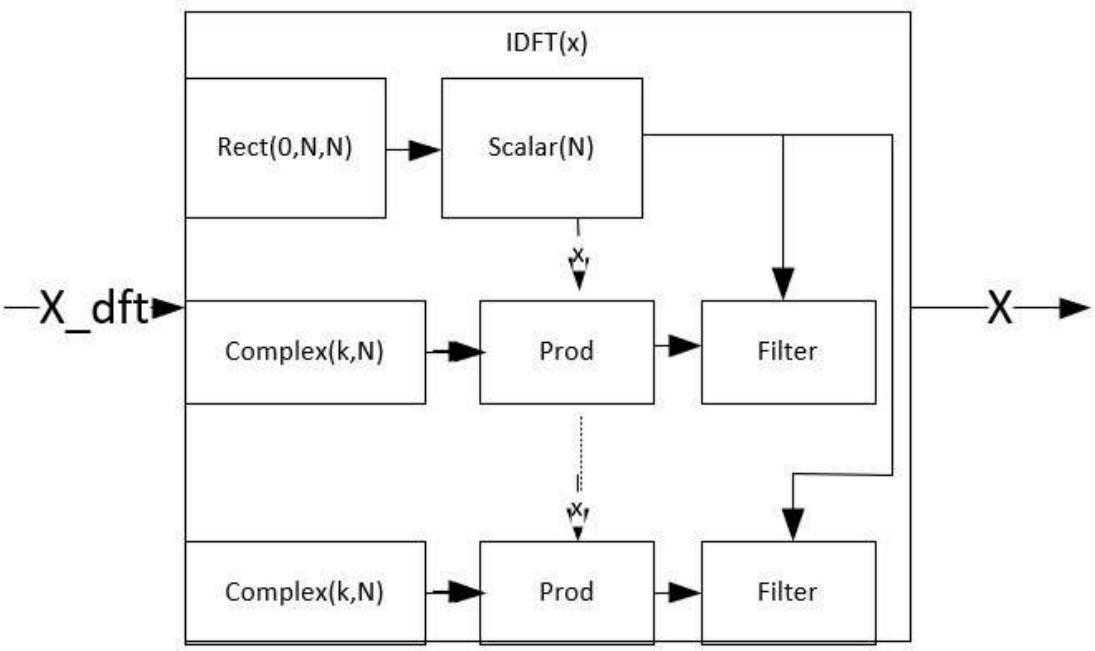
since we want STFT with no overlap and we made a range helper and a DFT helper, we select a range of values each time, perform a DTFT on them, and append the result in a matrix. We can compare with SCIPY, and we can see that the results match exactly up to transposing, and de-normalizing scipy's FFT by the window length.

IDFT Explanation - Helper function

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X^d[k] e^{\frac{2j\pi kn}{N}}$$

Converts a signal back from its frequency representation to its time representation, code is identical to DFT up to changing the sign of the exponent, and retaining the normalizing

factor of 1/N (that we discarded in the DFT).



In [89]:
```
def IDFT_inefficient(x:np.ndarray) -> np.ndarray:
    N = len(x)
    summation_rect = hw1.Rect(0,N,N)

    X_Idft =np.zeros(N,dtype=complex)
    for k in range(N):
        exponent = Complex(k,N)
        x_comp =hw1.Prod(exponent,x)
        X_Idft[k] = hw1.Filter([1],summation_rect,x_comp)
    return X_Idft
```

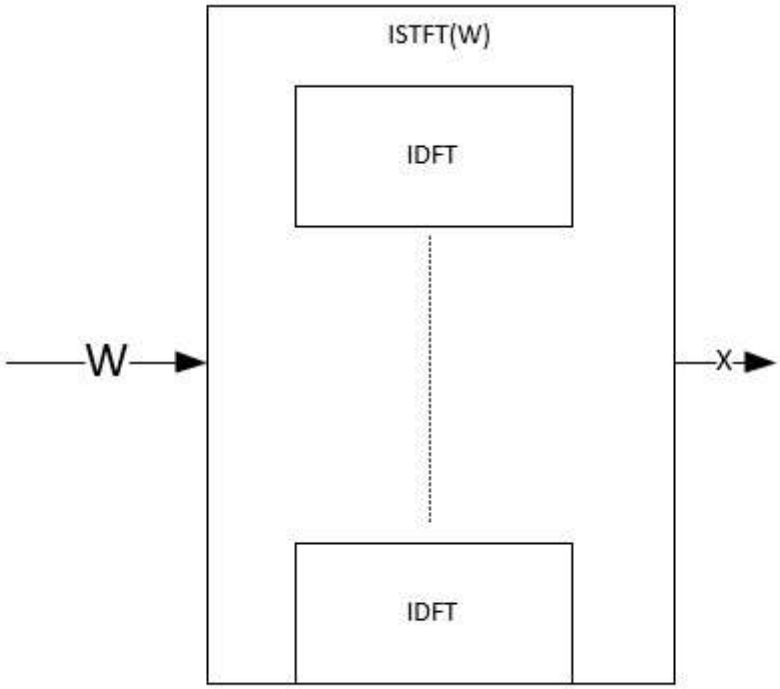
In [90]:
```
signal = np.array([3,4,5,6,7,6,5,4,3])
np.set_printoptions(precision=6, suppress=True)
print("signal:\n",signal)
print("DFT(signal):\n",DFT(signal))
print("IDFT(DFT(signal)): \n",np.abs(IDFT_inefficient(DFT(signal)))) #abs is there
```

```
signal:
 [3 4 5 6 7 6 5 4 3]
DFT(signal):
 [43.      +0.j       -7.790859-2.835641j  0.216881+0.181985j
 -0.5     -0.866025j  0.073978+0.41955j   0.073978-0.41955j
 -0.5     +0.866025j  0.216881-0.181985j -7.790859+2.835641j]
IDFT(DFT(signal)):
 [3. 4. 5. 6. 7. 6. 5. 4. 3.]
```

In [91]:
```
def ISTFT_inefficient(W: np.ndarray) -> np.ndarray:# every row is an
    M=W.shape[0]
    N=W.shape[1]
    x = np.zeros(M*N,dtype=complex)
    for i in range(N):
```

```
        x[i*M:(i+1)*M] = IDFT_inefficient(W[:,i])
    return x
signal = np.array([0, 2, 2, 2, 3, 4, 5, 5, 5, 5, 5, 5, 5, 4, 4])
print(len(signal))
print("the original signal:\n" ,signal)
print("ISTFT(STFT(signal)): \n",ISTFT_inefficient(STFT_inefficient(signal,M=5))) #a
```

```
15
the original signal:
 [0 2 2 2 3 4 5 5 5 5 5 5 5 4 4]
ISTFT(STFT(signal)):
 [-0.-0.j  2.+0.j  2.+0.j  2.-0.j  3.-0.j  4.-0.j  5.+0.j  5.-0.j  5.+0.j
  5.-0.j  5.-0.j  5.+0.j  5.+0.j  4.-0.j  4.-0.j]
 ---------------------------------------------------------------------------
```



ALTERNATE ISTFT explanation

we seperate each section which we previously transformed to the frequency domain with a DFT, and return it back to the time domain via the IDFT helper function.
Afterwards, we append each section of length M to a signal that matches the dimensions of the original time signal $(M \cdot N)$, and get the original signal back.

PART 3

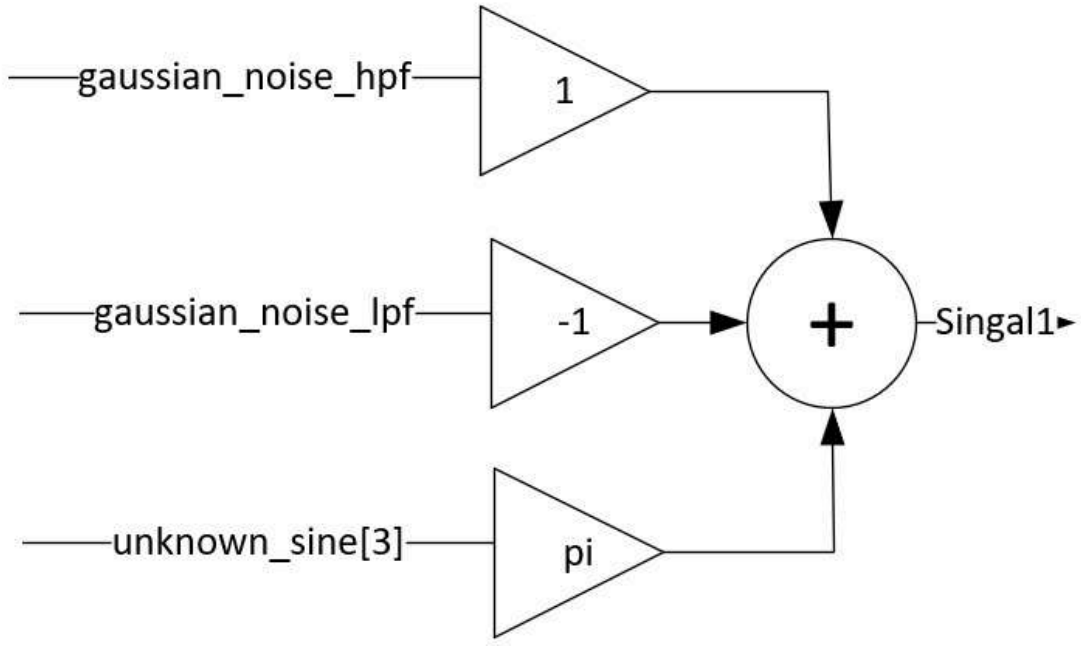
Setting Up Basic Signals:

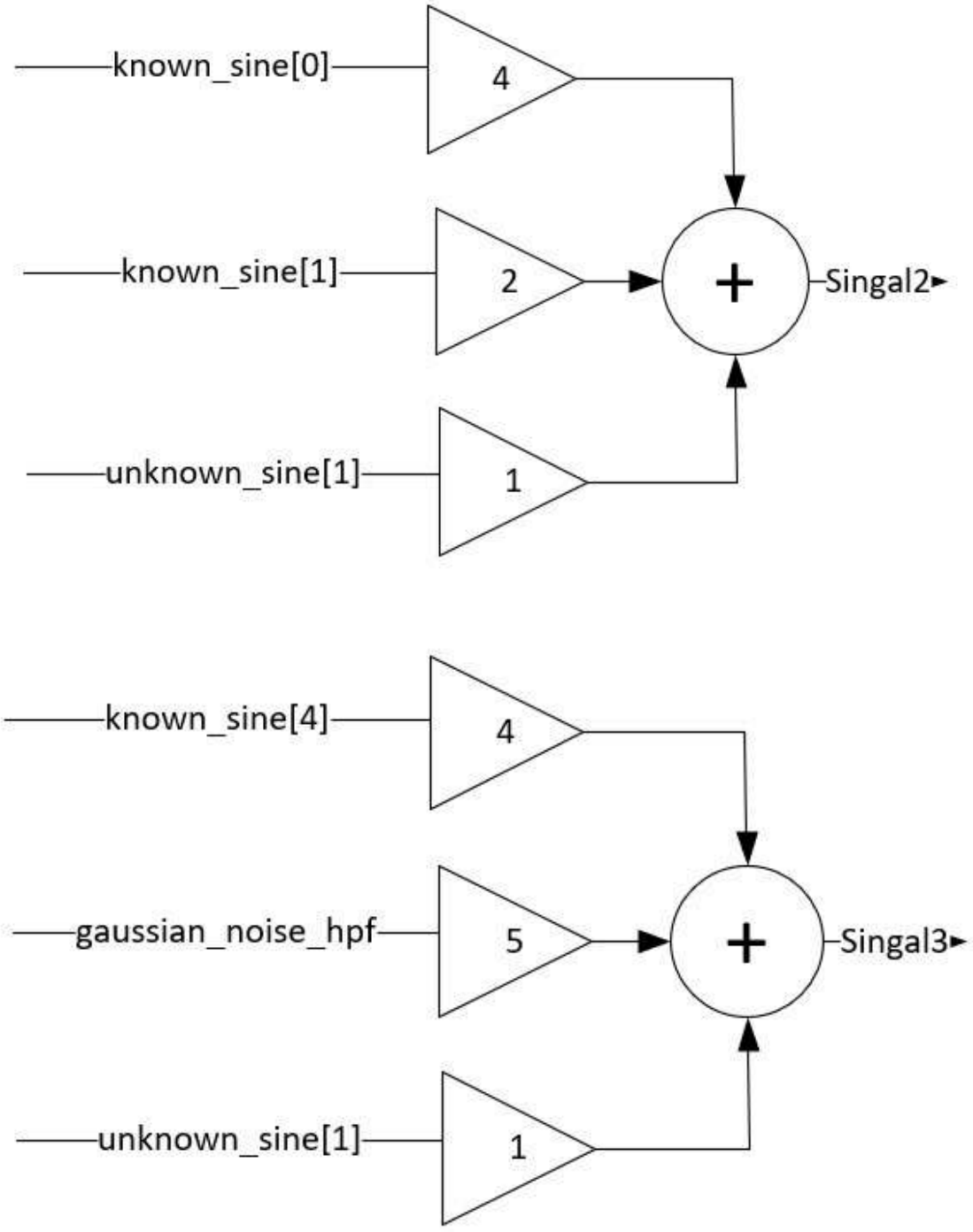
In [92]:
```
gaussian_noise_signal = hw1.WGN(4096,0,1)
low_pass_filter = hw1.Rect(0,6,6)
```

```
gaussian_noise_lpf = hw1.Filter([1],low_pass_filter, hw1.WGN(4101,0,1))
high_pass_filter = hw1.Add(hw1.Rect(0,1,2), hw1.Scalar(hw1.Rect(1,2,2),-1))
gaussian_noise_hpf =  hw1.Filter([1],high_pass_filter,hw1.WGN(4097,0,1))

known_sines = []
unknown_sines = []
random_frequencies = hw1.Scalar(Rand(5),5)
for i in range(5):
    known_sines.append(hw1.Sine(f=i*500,N=4096,ph=0))
    unknown_sines.append(hw1.Sine(f=random_frequencies[i]*500,N=4096,ph=0))
```

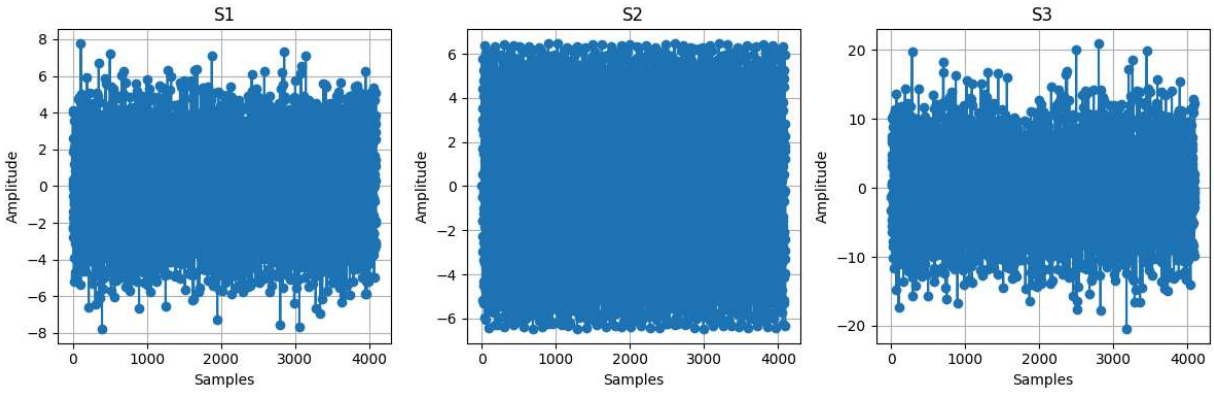
Creating Three Signals as a Linear Combination:

In [93]:
```
signal_1 = hw1.Add(hw1.Add(hw1.Scalar(gaussian_noise_hpf ,1),hw1.Scalar(gaussian_no
signal_2 = hw1.Add(hw1.Add(hw1.Scalar(known_sines[2] ,4),hw1.Scalar(known_sines[1],
signal_3 = hw1.Add(hw1.Add(hw1.Scalar(known_sines[4] ,4),hw1.Scalar(gaussian_noise_
```





In [94]:
```
index = np.arange(4096)
hw1.plot_three_side_by_side(index, signal_1, index, signal_2, index, signal_3, titl
```



The STFT of the Signals

We want to visualize Sine waves with the spectrogram, we know that inputting f into Sine outputs `np.sin((2 * np.pi * f * n )/N + ph)`,
meaning, that

$$\sin(2\pi f\frac{n}{N}) = \sin(2\pi f^* t) \underbrace{=}_{\text{sampled at}} x(t=\frac{n}{F_s})$$

$$\implies f^* = \frac{f \cdot F_s}{N}[Hz]$$

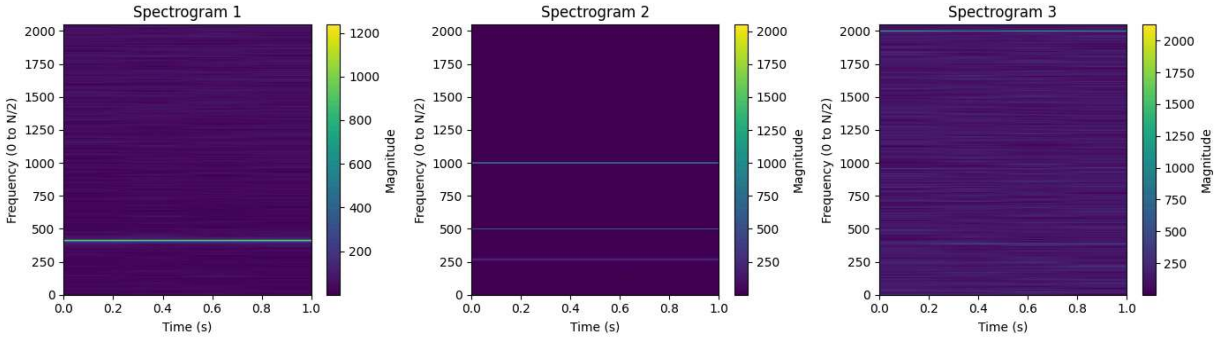
We want to display the signal with normalized frequency units (for example f=1, should appear at the first frequency bin)
if we set $F_s = N$ we get that, and we inadvertently also set the spectrogram to be over 1 second of the signal (though we normalize the axis)

A note about displaying the STFTs

we are using `np.abs` to display the STFTs, we can create a suitable substitute by taking `Prod(signal_1, Conj(Signal_1))`.
However we didn't do so due to numerical stability issues we encountered, and chose not to focus on since they didn't seem particularly relevant to this excersize.

In [95]:
```
def plot_three_spectrograms(signal_1,signal_2,signal_3):
    # Parameters
    N = 4096  # Total number of samples
    M = 1024   # Window size for STFT (ensures good frequency resolution)
    overlap = 0  # No overlap
    Fs = 4096  # Sampling frequency in normalized units



    matrix_1 = STFT(signal_1, M=M)
    magnitude_1 = np.abs(matrix_1)
    matrix_2 = STFT(signal_2, M=M)
    magnitude_2 = np.abs(matrix_2)
    matrix_3 = STFT(signal_3, M=M)
```

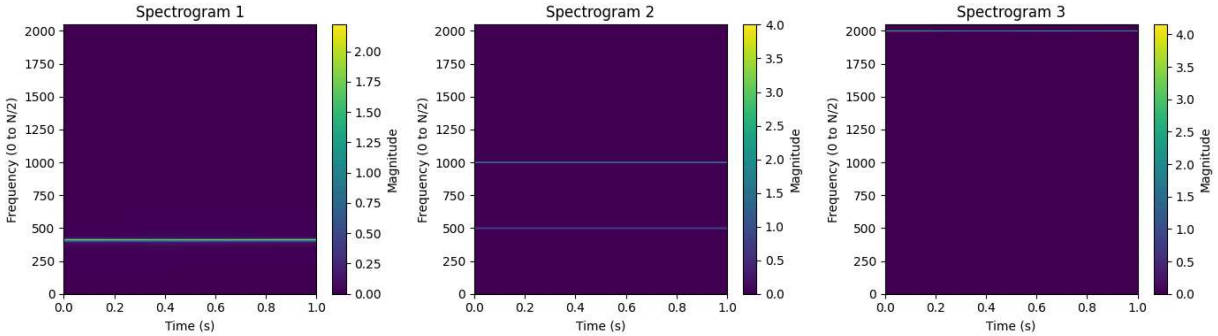
```
    magnitude_3 = np.abs(matrix_3)

    # Define frequency and time axes
    frequencies = np.linspace(0, 0.5 * N, M // 2 + 1)  # Frequency range
    time = np.linspace(0, N / Fs, (N - overlap) // (M - overlap))

    # Plot three identical spectrograms side by side
    plt.figure(figsize=(14, 4))

    # Spectrogram 1
    plt.subplot(1, 3, 1)
    plt.imshow(
        magnitude_1[:M // 2 + 1, :],  # Positive frequencies only
        extent=[time[0], time[-1], frequencies[0], frequencies[-1]],
        aspect='auto',
        origin='lower',
        cmap='viridis'
    )
    plt.colorbar(label="Magnitude")
    plt.xlabel("Time (s)")
    plt.ylabel("Frequency (0 to N/2)")
    plt.title("Spectrogram 1")

    # Spectrogram 2
    plt.subplot(1, 3, 2)
    plt.imshow(
        magnitude_2[:M // 2 + 1, :],  # Positive frequencies only
        extent=[time[0], time[-1], frequencies[0], frequencies[-1]],
        aspect='auto',
        origin='lower',
        cmap='viridis'
    )
    plt.colorbar(label="Magnitude")
    plt.xlabel("Time (s)")
    plt.ylabel("Frequency (0 to N/2)")
    plt.title("Spectrogram 2")

    # Spectrogram 3
    plt.subplot(1, 3, 3)
    plt.imshow(
        magnitude_3[:M // 2 + 1, :],  # Positive frequencies only
        extent=[time[0], time[-1], frequencies[0], frequencies[-1]],
        aspect='auto',
        origin='lower',
        cmap='viridis'
    )
    plt.colorbar(label="Magnitude")
    plt.xlabel("Time (s)")
    plt.ylabel("Frequency (0 to N/2)")
    plt.title("Spectrogram 3")

    plt.tight_layout()
    plt.show()
plot_three_spectrograms(signal_1,signal_2,signal_3)
```



Filtering In the Frequency Domain:

We will note that in all our spectrograms the noise seems to be at a weaker amplitude with respect to our chosen signal, we can use the Thresh function to our advantage in this case. We will notice that our filtering is essentially done in the first spectrogram and the third - we can get rid of the unwanted lower-amplitude disturbances with the thresh function, however we can't really do the same in the second spectrogram as our signals amplitude (at $f = 500$) is sandwiched inbetween the other signals, so we can remove
one annoying sine but not the other. (at least not without making more complicated manipulations)

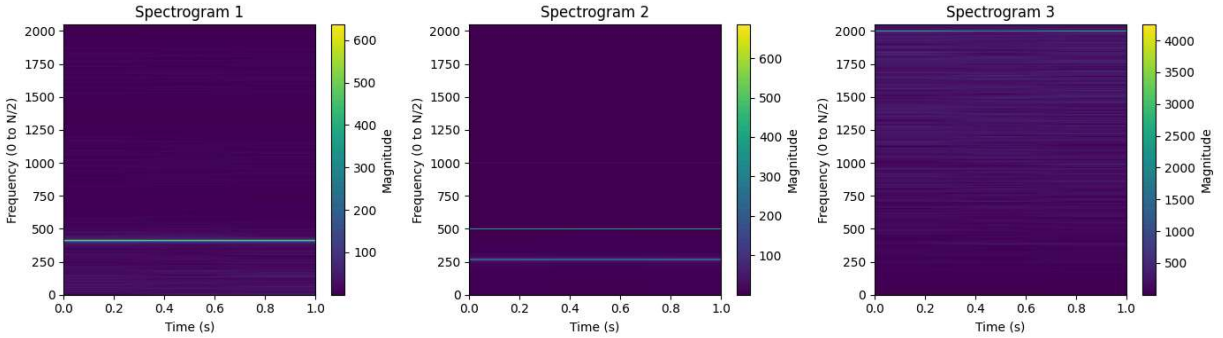
In [96]:
```
signal_1_stft = STFT(signal_1, M=512)
signal_1_stft_thresh = 120
filtered_stft_1 = Threshold(signal_1_stft, signal_1_stft_thresh)
signal_1_freq_filtered = ISTFT(filtered_stft_1)

signal_2_stft = STFT(signal_2, M=512)
signal_2_stft_thresh = 320
filtered_stft_2 = Threshold(signal_2_stft, signal_2_stft_thresh)
signal_2_freq_filtered = ISTFT(filtered_stft_2)

signal_3_stft = STFT(signal_3, M=512)
signal_3_stft_thresh = 470
filtered_stft_3 = Threshold(signal_3_stft, signal_3_stft_thresh)
signal_3_freq_filtered = ISTFT(filtered_stft_3)

plot_three_spectrograms(signal_1_freq_filtered,signal_2_freq_filtered,signal_3_freq
```



Filtering in the time domain:

we're having a harder time here when dealing with the noise - seeing as it encompasses all frequencies constructing a low pass filter or a high pass one does not really help us get rid of the white noise in all the spectrums. However, we can see how the usage of a low pass filter in spectrogram 2 lets us remove the high frequency sine noise that had a high intensity and was harder to remove with the thresh filtering

In [97]:
```
signal_1_time_filtered = hw1.Filter([1],low_pass_filter,signal_1)
signal_2_time_filtered = hw1.Filter(an=[1],bn= hw1.Rect(1,5,10), x=signal_2)
signal_3_time_filtered = hw1.Filter([1],high_pass_filter,signal_3)
plot_three_spectrograms(signal_1_time_filtered,signal_2_time_filtered,signal_3_time
```



Combined Filtering

We can see how a combined filtering action lets us utilize the advantages of both methods and as such give us some really good control.

In [98]:
```
signal_1_COMBIBED_filtered = hw1.Filter([1],low_pass_filter,signal_1_freq_filtered)
signal_2_COMBIBED_filtered = hw1.Filter(an=[1],bn= hw1.Rect(1,5,10), x=signal_2_fre
signal_3_COMBIBED_filtered = hw1.Filter([1],high_pass_filter,signal_3_freq_filtered
plot_three_spectrograms(signal_1_COMBIBED_filtered,signal_2_COMBIBED_filtered,signa
```