Using the Numpy `fft` Library

With `%pylab` loaded all of Numpy is in the namespace, including the `fft` library. To get to these functions you have to use the prefix `fft`

In [1]:
```
# %pylab inline
from numpy import *
from matplotlib.pyplot import *
%matplotlib inline
import sk_dsp_comm.sigsys as ss
# import simple_sa_new as sa_new
import scipy.signal as signal
from IPython.display import Audio, display
from IPython.display import Image, SVG

%config InlineBackend.figure_formats=['svg'] # SVG inline viewing
```

In [ ]:
```
# Explore with
# press 'tab' following '.' in 'fft' to see the functions
fft.
```

In [3]:
```
x1 = ones(10)
x1
```

Out[3]:
```
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Timing Test

The `FFTW` is very fast even without choosing a power of 2.

In [11]:
```
# not a power of 2 length
Nfft = 1000
%timeit X1 = fft.fft(x1,Nfft) # zero pad to 1000 points
#X1
```

```
5.34 µs ± 14.9 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops e
ach)
```

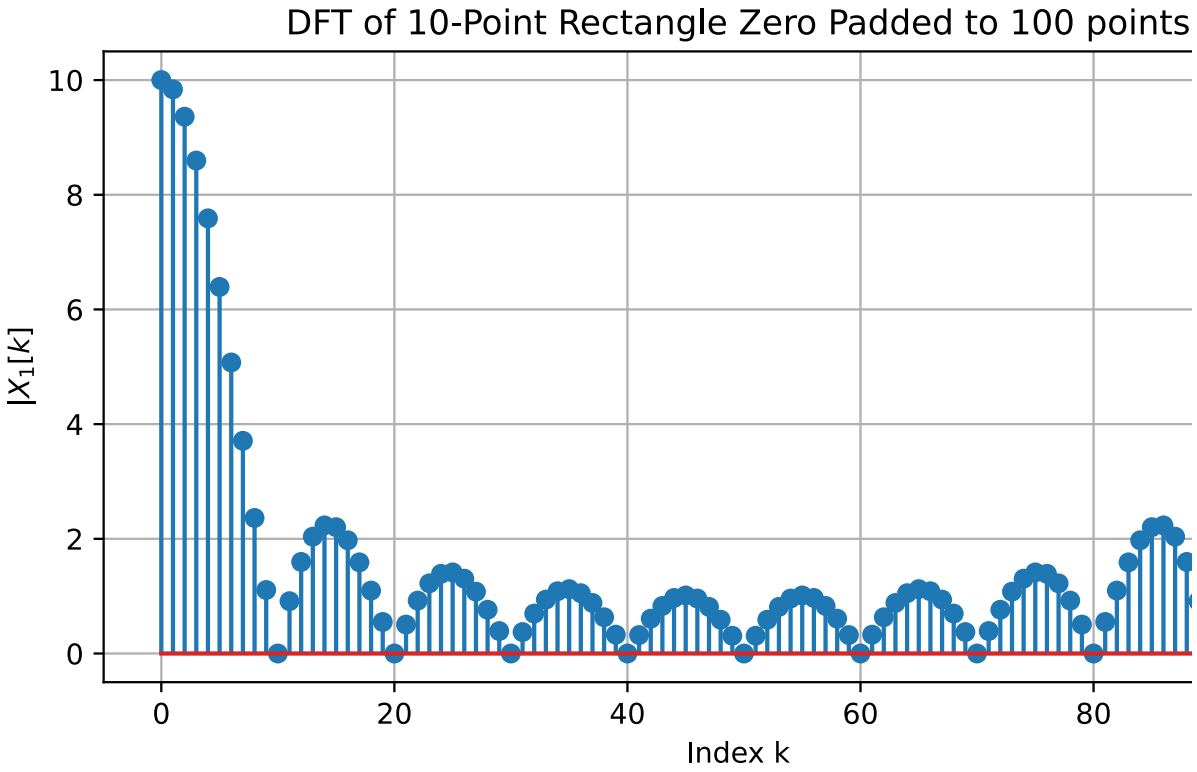
In [12]:
```
# choose a power of two length
Nfft = 1024
%timeit X1 = fft.fft(x1,Nfft) # zero pad to 1024
#X1
```

```
5 µs ± 19.4 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops eac
h)
```

Zero Padd the DFT (FFT)

Making the DFT act more like the DTFT using zero padding.

In [16]:
```
Nfft = 100
X1 = fft.fft(x1,Nfft)
figure(figsize=(8,4))
stem(abs(X1))
title(f'DFT of 10-Point Rectangle Zero Padded to {Nfft} points')
xlabel('Index k')
ylabel(r'$|X_1[k]$')
grid();
#savefig()
```



DFT and IDFT

Consider some basic DFT/IDFT operations using `fft.fft` as a fast and efficient version of the theoretical DFT and `fft.ifft` as a fast and efficient version of the theoretical IDFT.

In [ ]:
```
x3 = [1,2,3,4,5]
X3 = fft.fft(x3)
X3
```

Out[ ]:
```
array([15. +0.j        , -2.5+3.4409548j , -2.5+0.81229924j,
       -2.5-0.81229924j, -2.5-3.4409548j ])
```

In [ ]:
```
x3p = fft.ifft(X3)
x3p.real
```

Out[ ]:
```
array([1., 2., 3., 4., 5.])
```

- From the above we see that the `ifft` does return the original sequence. We take the real part to rove any small numerical precision errors which introduce small imaginary parts. Since we start a real sequence we should end up with a real sequence.

In [9]:
```
Y1 = fft.fft([1,1,2,2])
Y1
```

Out[9]:
```
array([ 6.+0.j, -1.+1.j,  0.+0.j, -1.-1.j])
```

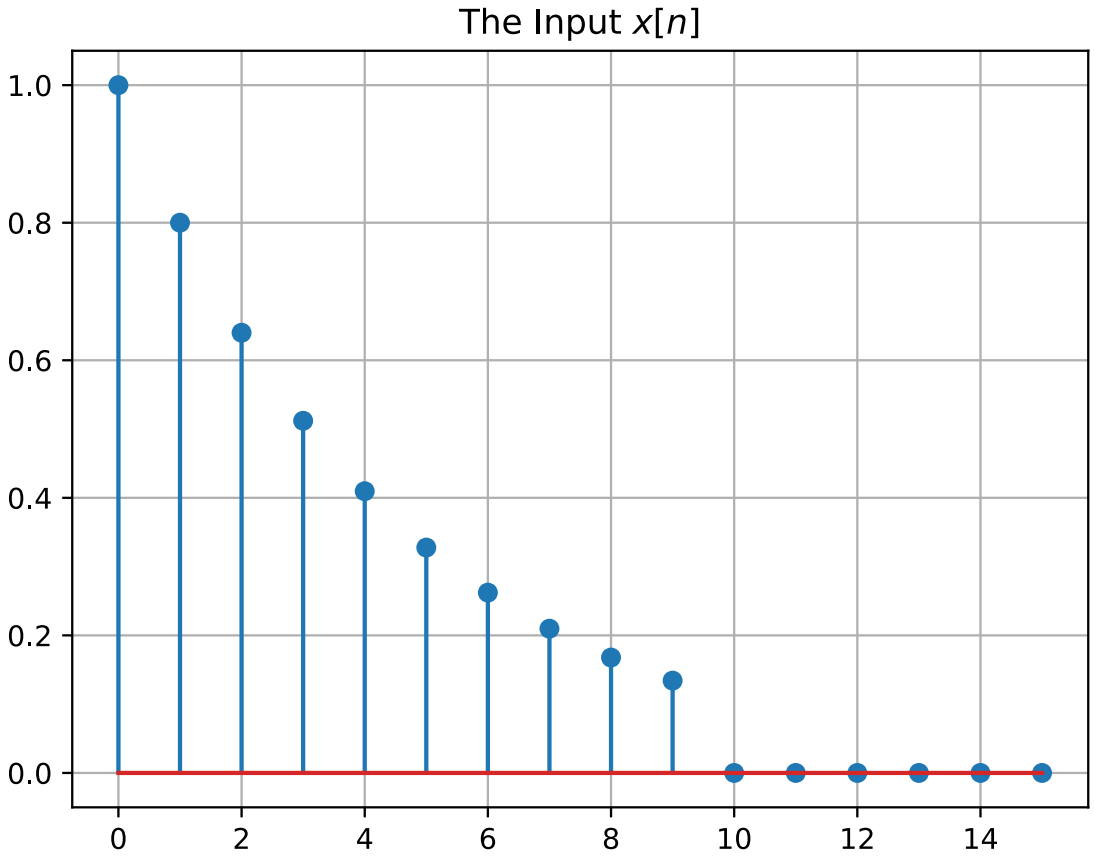
In [10]:
```
y1 = fft.ifft(Y1)
y1.real
```

Out[10]:
```
array([1., 1., 2., 2.])
```

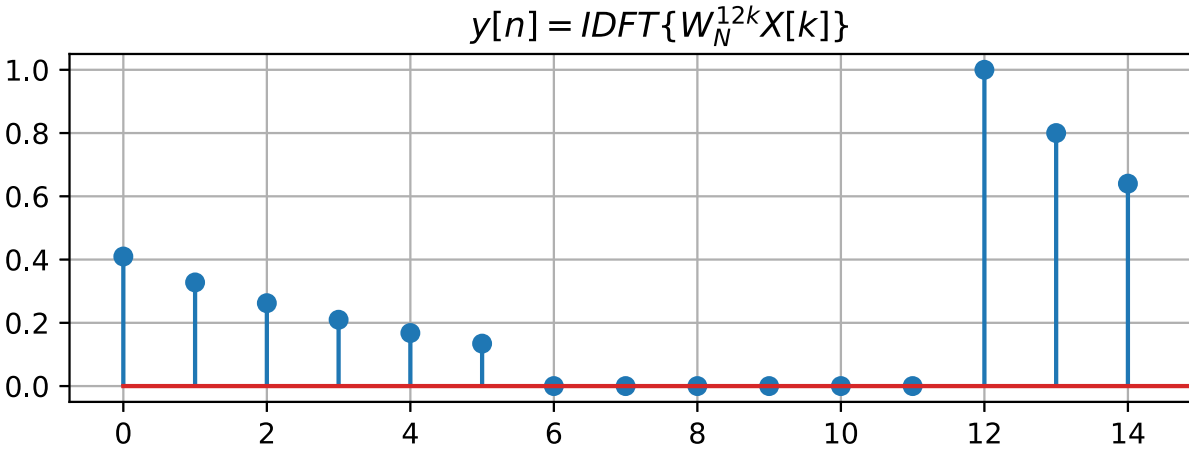
Circular Shift

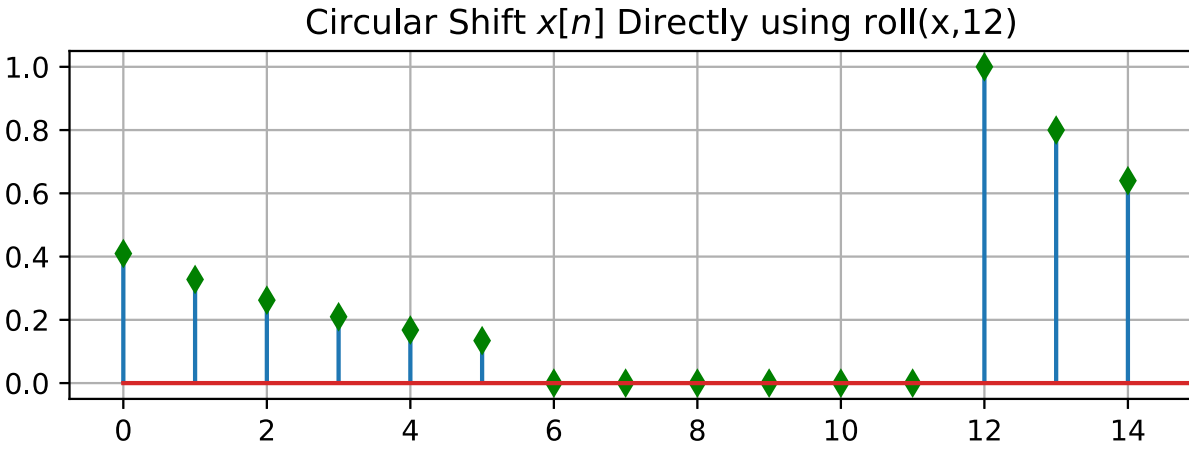
Consider a basic circular shift in the DFT/IDFT domain and compare it with the `numpy` function `roll`. Define a truncated exponential as the input sequence. Then shift it to the right by 12 samples out of the total length of 16 samples.

In [17]:
```
n = arange(16)
a = 0.8
x = a**n * ss.drect(n,10)
stem(n,x)
title(r'The Input $x[n]$')
grid();
```



In [18]:
```
k = arange(16)
X = fft.fft(x)
Y = X*exp(-1j*2*pi/16*k*12) #
y = fft.ifft(Y).real
subplot(211)
stem(n,y)
title(r'$y[n] = IDFT\{W_N^{12k}X[k]\}$')
grid();
subplot(212)
stem(n,roll(x,12),markerfmt='gd')
title(r'Circular Shift $x[n]$ Directly using roll(x,12)')
grid();
tight_layout()
```





Quick Circular Shift Example

In [19]:
```
n = arange(8)
x = [1,1,1,1,1,0,0,0]
# Shift to left by 1
y = fft.ifft(exp(1j*2*pi*1*n/8)*fft.fft(x)).real
y
```

Out[19]:
```
array([1.00000000e+00, 1.00000000e+00, 1.00000000e+00, 1.00000000e+00,
       0.00000000e+00, 0.00000000e+00, 5.55111512e-17, 1.00000000e+00])
```

Circular Convolution

Following the notes example 7.2, we circularly convolve $x_1[n]$ and $x_2[n]$ using the DFT/IDFT.

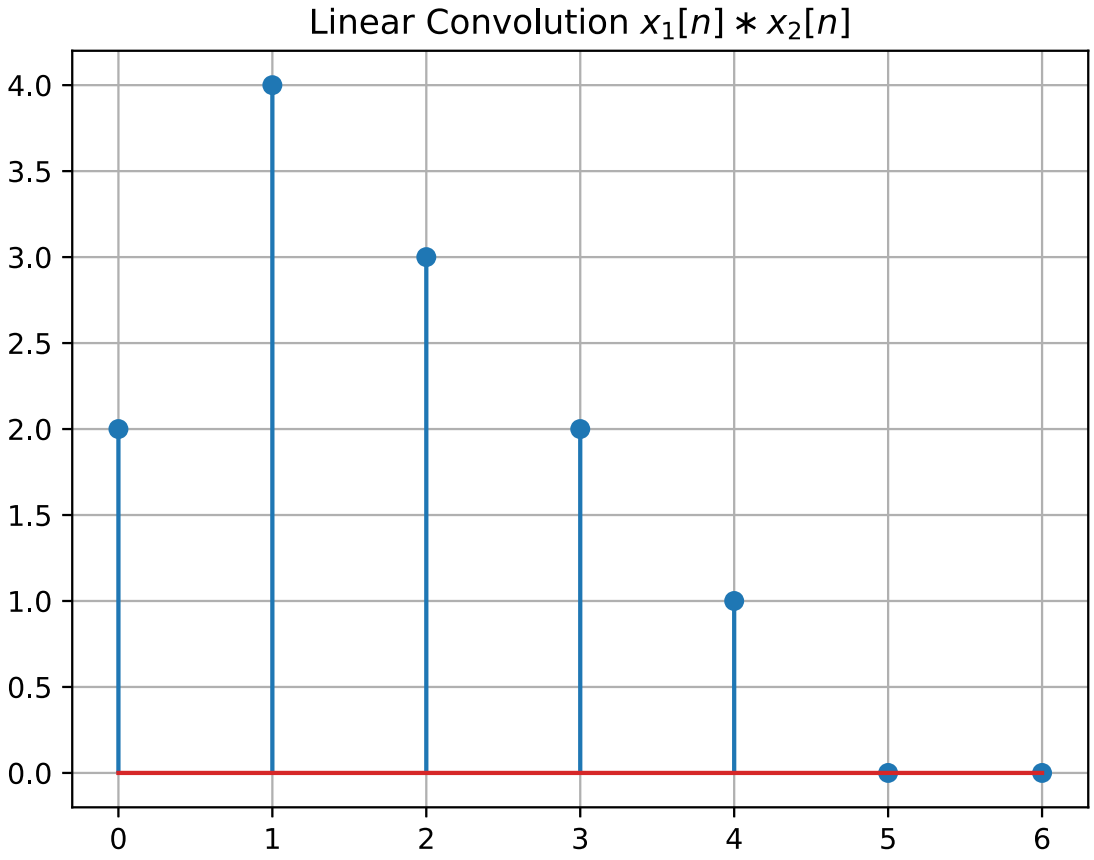
In [20]:
```
x1 = [2,2,1,1]
x2 = [1,1,0,0]
```

In [21]:
```
X3 = fft.fft(x1)*fft.fft(x2)
x3 = fft.ifft(X3)
x3.real
```

Out[21]:
```
array([3., 4., 3., 2.])
```

- Consider the linear convolution of $x_1[n]$ and $x_2[n]$ using the DFT/IDFT. We know that to make circular convolution behave as linear convolution we must zero pad the sequence up to $N = L + P - 1 = 4 + 4 - 1 = 7$. First we compute the linear convolution for reference using the `signal.convolution` function:

In [22]:
```
stem(signal.convolve(x1,x2))
title(r'Linear Convolution $x_1[n] \ast x_2[n]$')
grid();
```

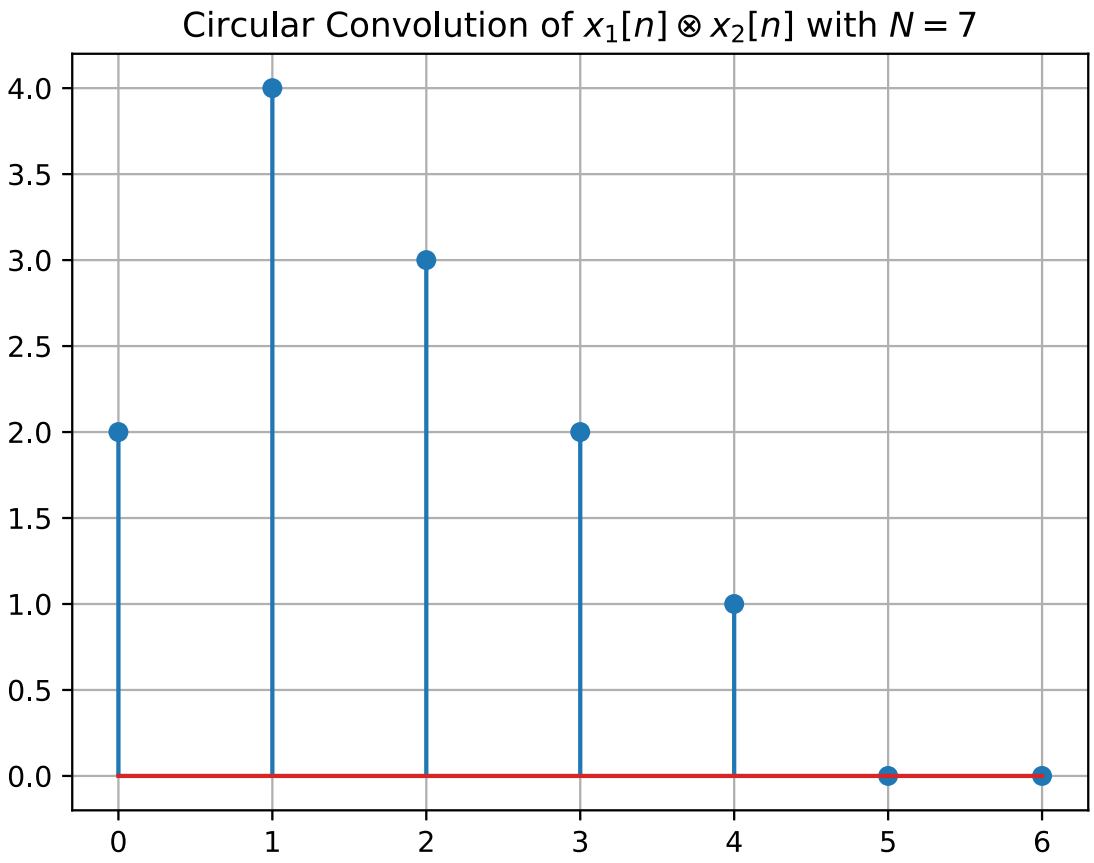


- Now zero pad the circular convolution result and place all the calculations in one line:

In [23]:
```
# Verify as a circular convolution
N = 7
x3_circ = fft.ifft(fft.fft(x1,N)*fft.fft(x2,N),N).real
x3_circ
```

Out[23]:
```
array([ 2.00000000e+00,  4.00000000e+00,  3.00000000e+00,  2.00000000e+00,
        1.00000000e+00,  2.53765263e-16, -2.53765263e-16])
```

In [24]:
```
stem(x3_circ.real)
title(r'Circular Convolution of $x_1[n] \otimes x_2[n]$ with $N = 7$')
grid();
```

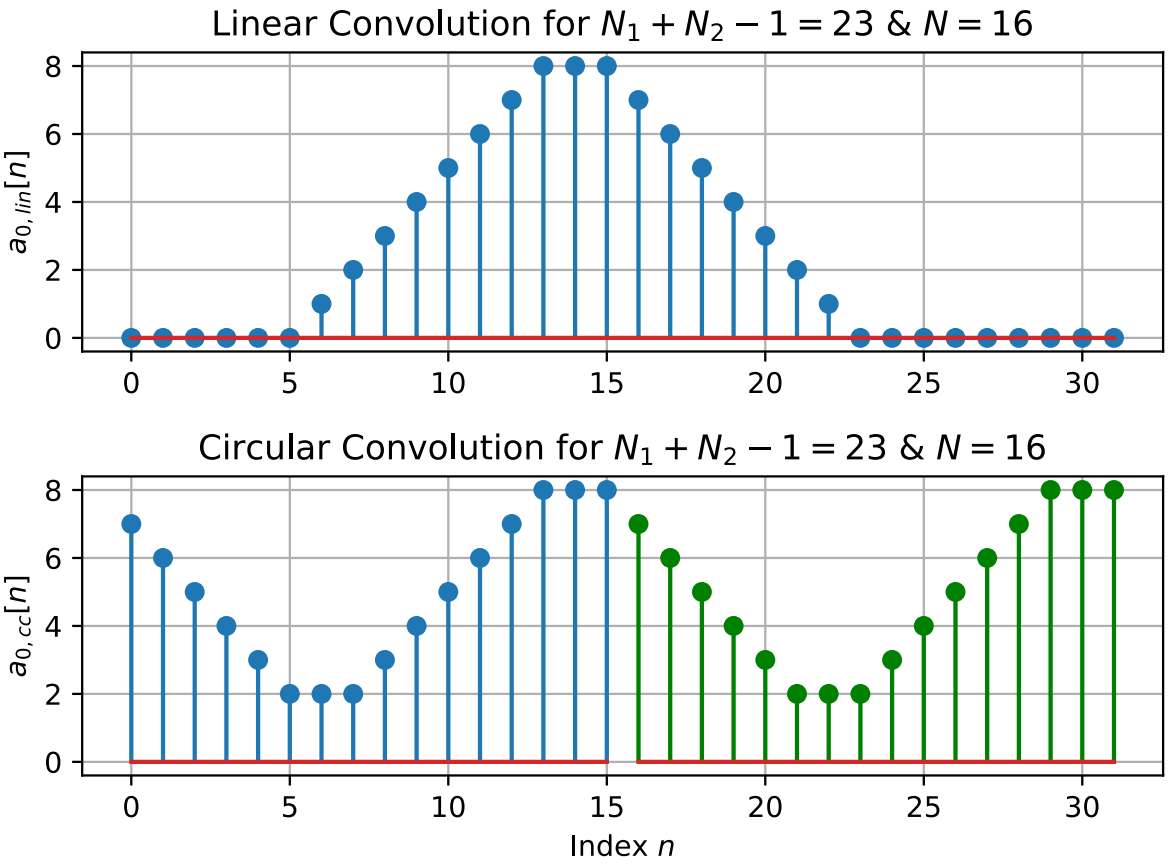


- The results agree; circular convolution can emulate linear convolution!

Notes Example 7.4

In [51]:
```
# Define the sequences
figure(figsize=(6,4.5))
N = 16 # try 20 and 24 to see need for N >= 23
nN = arange(2*N)
a1 = hstack((ones(10), zeros(len(nN)-10)))
a2 = array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1])
# Linear convolution
subplot(211)
a0 = signal.lfilter(a2,1,a1)
stem(nN,a0)
title(f'Linear Convolution for $N_1+N_2-1=23$ & $N=${N}')
ylabel(r'$a_{0,lin}[n]$')
grid();
```

```
# Circular convolution using the FFT/IFFT (DFT/IDFT)
# Take real-part since we know convolution is real &
# a very small imaginary part may (will) exist
a0_circ = real(fft.ifft(fft.fft(a1,N)*fft.fft(a2,N)))
subplot(212)
stem(nN[:N],a0_circ)
stem(nN[N:],a0_circ,'g', markerfmt='go')
title(f'Circular Convolution for $N_1+N_2-1=23$ & $N=${N}')
ylabel(r'$a_{0,cc}[n]$')
xlabel(r'Index $n$')
grid();
tight_layout()
```



Overlap and Add Transform Domain Filtering

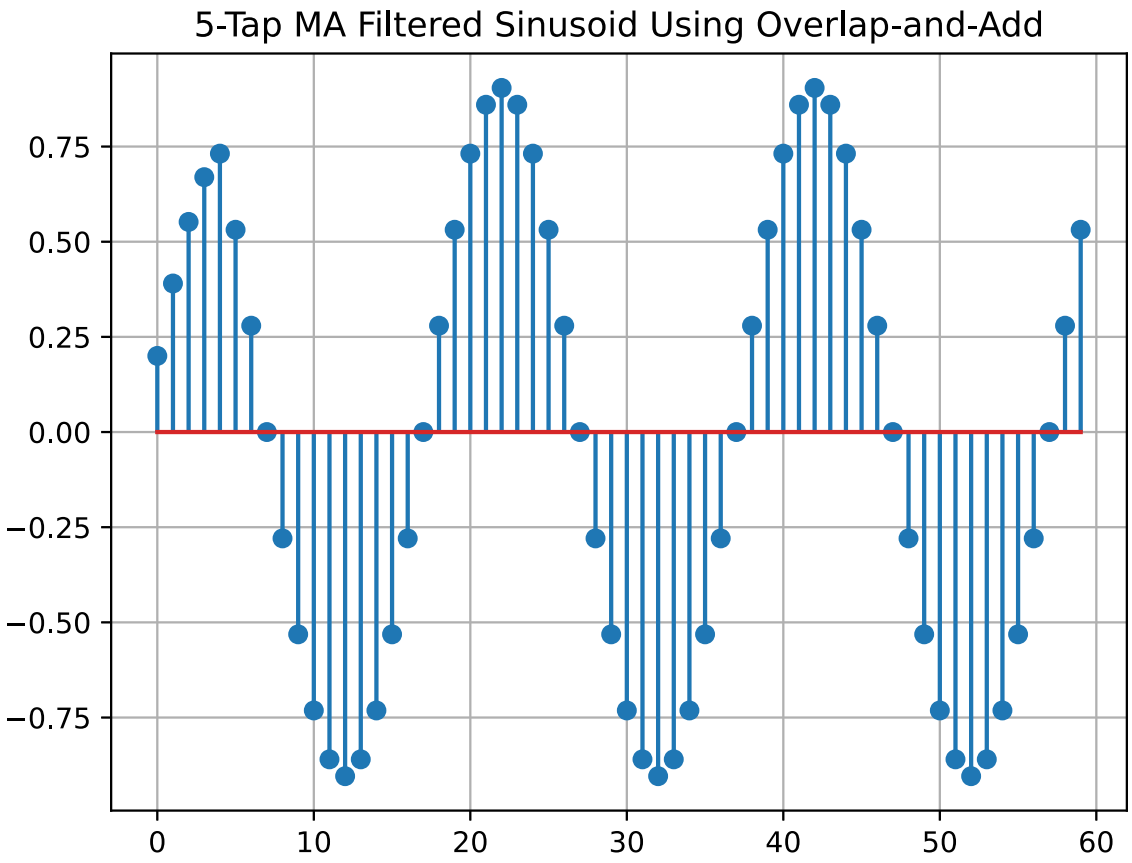
Consider a long sequence being filter by an FIR filter. The overlap-and-add method simplfies the process by breaking the input $x[n]$ into sjort subsequences, which are induvidually zero padded.

In [30]:
```
n = arange(60)
x = cos(2*pi*n/20)
h = ones(5)/5
```

In [33]:
```
# Overlap and Add
```

```
y, y_mat = ss.oa_filter(x,h,20,True)
# Overlap and Save
# y, y_mat = ss.os_filter(x,h,20,True)
```

In [32]:
```
stem(n,y)
title(r'5-Tap MA Filtered Sinusoid Using Overlap-and-Add')
grid();
```



In [42]:
```
y_mat.shape
# stem(y_mat[3,:])
```

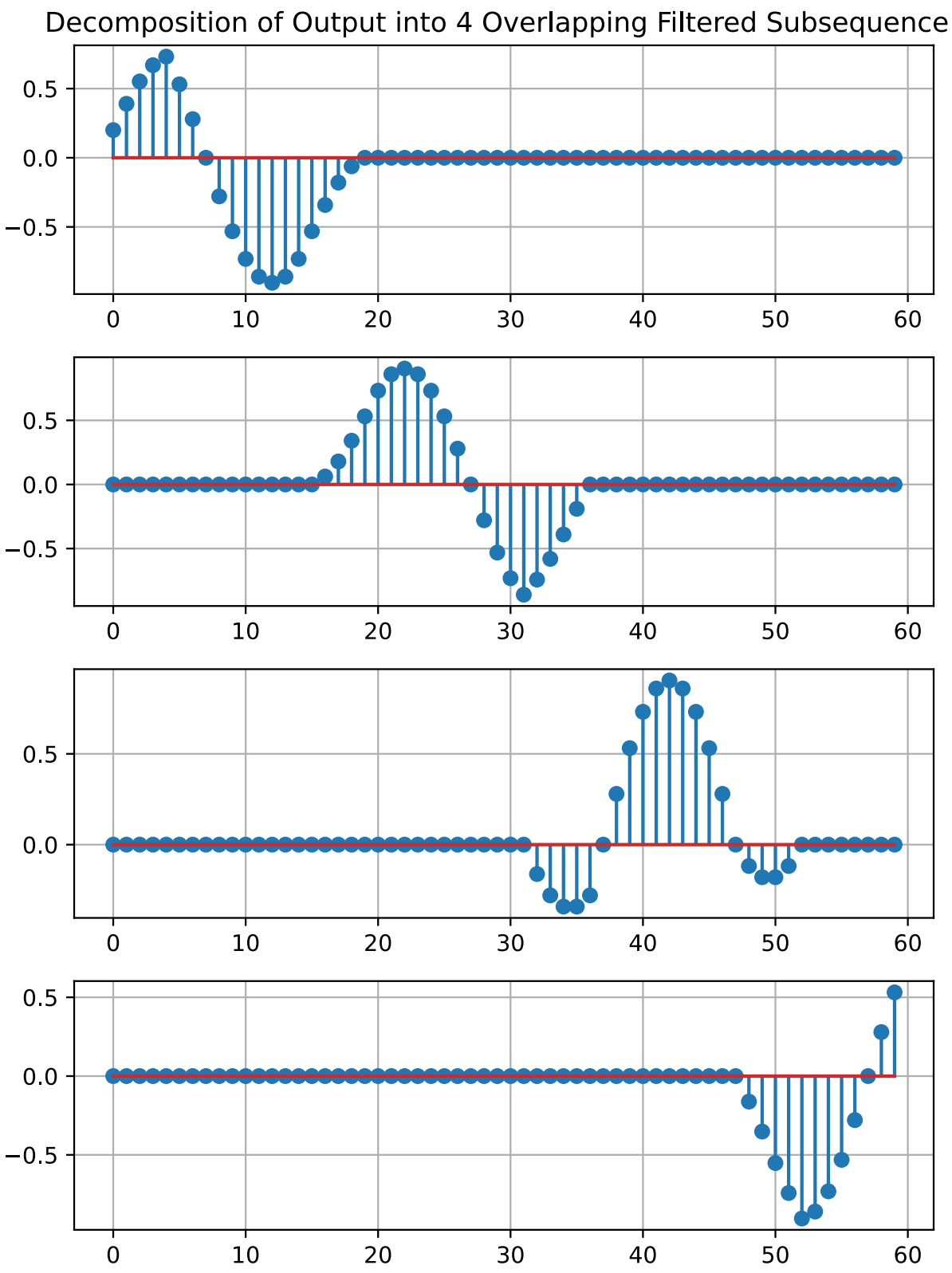
Out[42]: (4, 60)

Vizualize How the Overlap and Save Forms the Output

Filter a possibly infinite sequence using the FFT we can use overlap and save or overlap and add techniques. Recall the `fft_filter_bank()` function in Project 2 (Fall 2025) uses the overlap and save method.

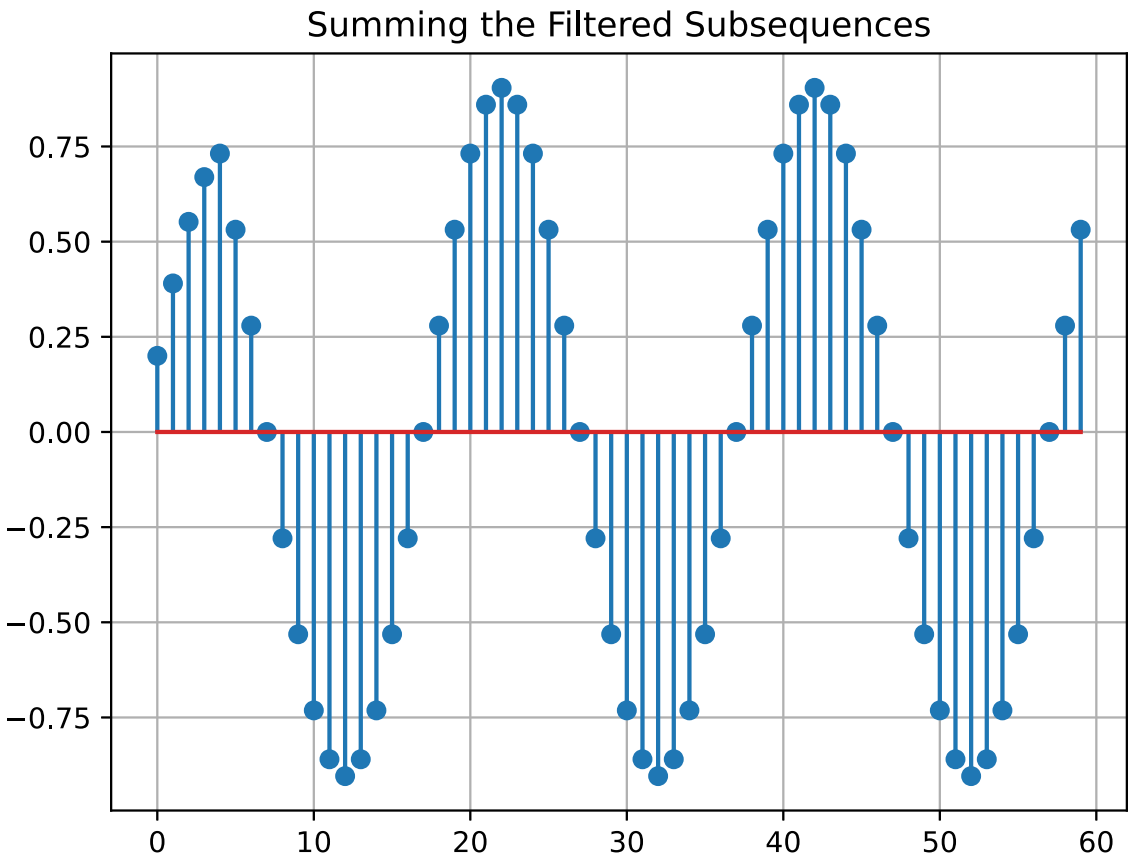
In [54]:
```
figure(figsize=(6,8))
subplot(4,1,1)
stem(n,y_mat[0,:])
title(r'Decomposition of Output into 4 Overlapping Filtered Subsequenc
```

```
grid();
subplot(4,1,2)
stem(n,y_mat[1,:])
grid();
subplot(4,1,3)
stem(n,y_mat[2,:])
grid();
subplot(4,1,4)
stem(n,y_mat[3,:])
grid();
tight_layout()
```



- The subplots above are the filter outputs from the overlapping of 4 filtered subsequences. If we add each row in the `y_mat` matrix we obtain the overlap-and-add output

In [44]:
```
stem(sum(y_mat,0))
title(r'Summing the Filtered Subsequences')
grid();
```



FFT-Based FIR Filtering from Scratch

In [43]:
```
def fft_os_fir(x,N_fft,b):
    """
    Overlap and Save FIR filtering

    y = fft_os_fir(x,N_fft,b)

    x = real input signal ndarray
    N_fft = FFT half length, i.e. overlap length N
    b = real FIR  filter coefficients of length <= N_fft

    y = filtered output (real as input is real)

    Mark Wickert October 2021
    """
    b_zp = zeros(2*N_fft) # zero pad coeff. array
    b_zp[:len(b)] = b # place coeff. at the start
    B_zp = fft.fft(b_zp) # transform filter coeff.
    N_out_buf = len(x)//(N_fft) # Number of output buffers
    y = zeros(N_fft*N_out_buf) # define the output array
    x_state = zeros(N_fft) # state array to hold past inputs
```

```
    for k in range(N_out_buf):
        # Fill 2*N_fft array with old and new subarrays
        x_in = hstack((x_state,x[k*N_fft:(k+1)*N_fft]))
        Y_all = B_zp*fft.fft(x_in) # freq. domain filter
        y_all = fft.ifft(Y_all) # back to time domain
        # save the upper index points real-part
        y[k*N_fft:(k+1)*N_fft] = y_all[N_fft:].real
        x_state = x[k*N_fft:(k+1)*N_fft]
    return y
```