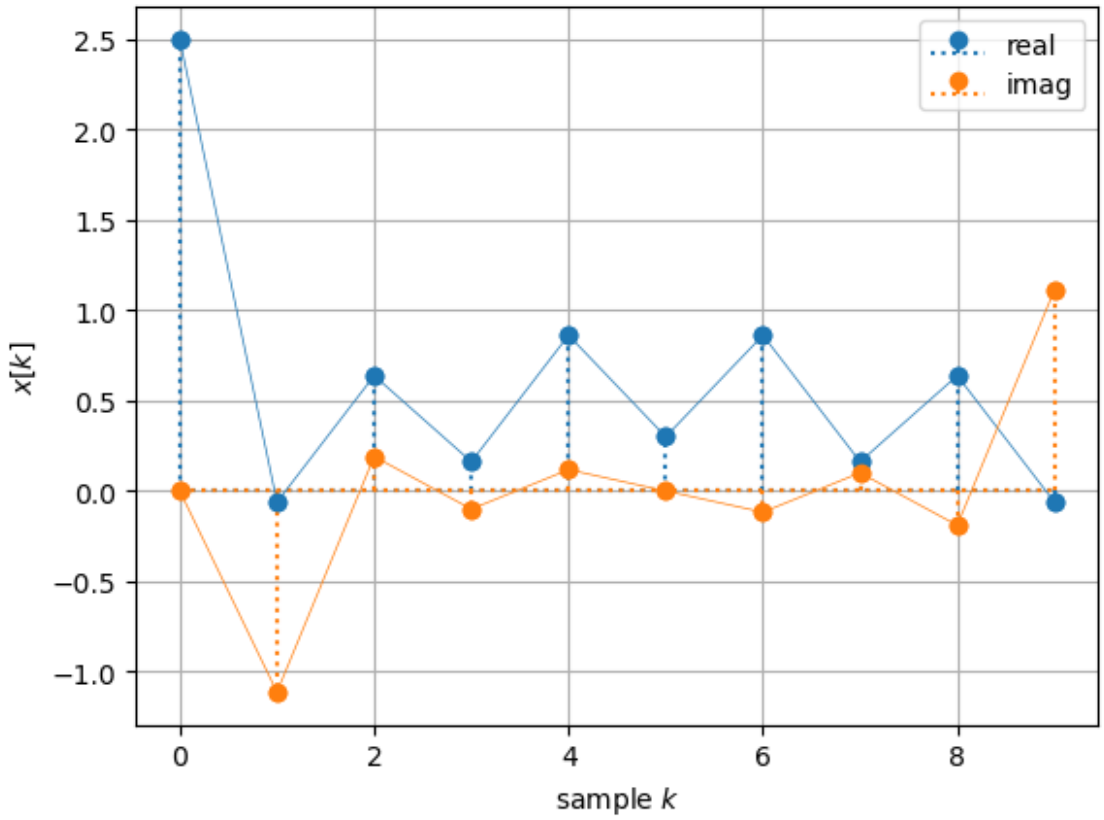
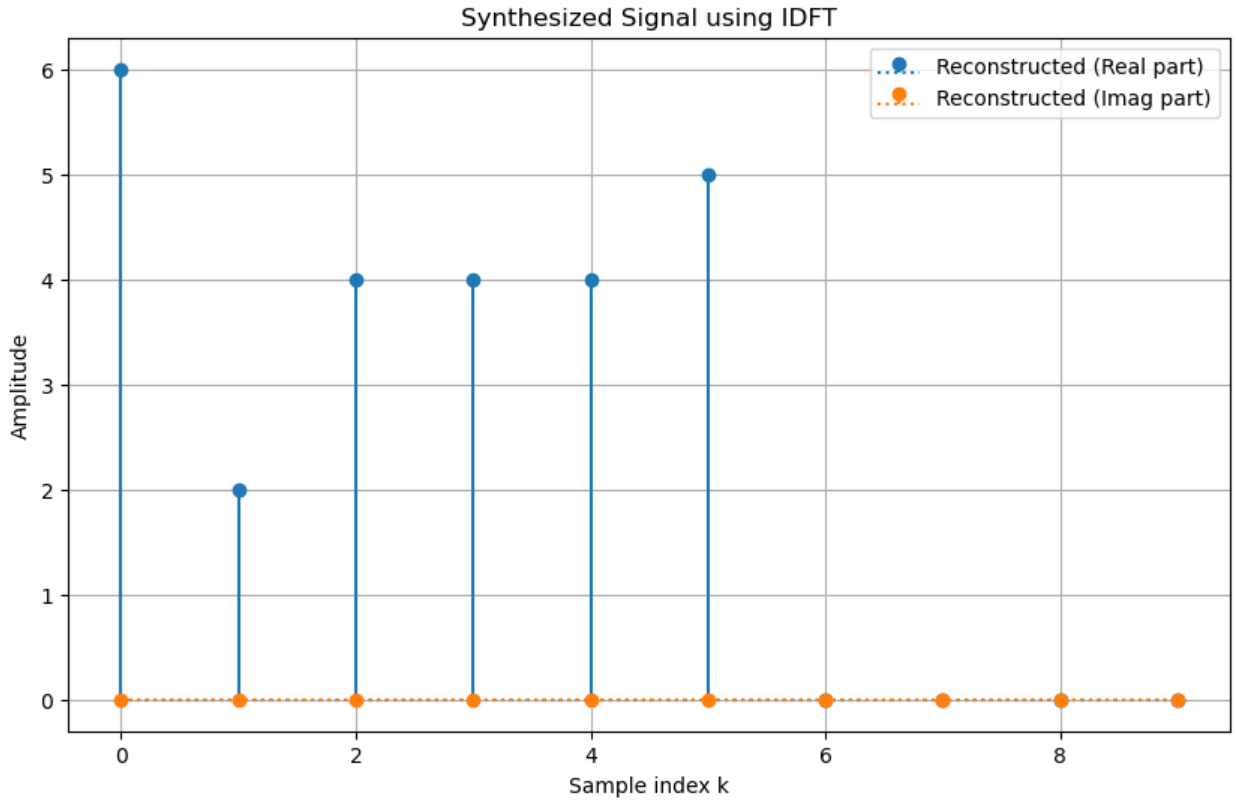
```python
import numpy as np
import matplotlib.pyplot as plt

# Define the signal
x_mu = np.array([6, 2, 4, 4, 4, 5, 0, 0, 0, 0], dtype=float)  #
Updated signal
N = len(x_mu)

# Create the index matrix K
k = np.arange(N)
mu = np.arange(N)
K = np.outer(k, mu)  # Compute K matrix

# Construct the Fourier matrix W for DFT
W = np.exp(-2j * np.pi * K / N)

# Compute DFT using the W matrix
X = np.dot(W, x_mu)

if N == 10:
    X_test = np.array([6, 2, 4, 4, 4, 5, 0, 0, 0, 0])  # Updated test
signal
    x_test = 1/N * np.matmul(W, X_test)

    plt.stem(k, np.real(x_test), label='real',
             markerfmt='C0o', basefmt='C0:', linefmt='C0:')
    plt.stem(k, np.imag(x_test), label='imag',
             markerfmt='C1o', basefmt='C1:', linefmt='C1:')
    plt.plot(k, np.real(x_test), 'C0o-', lw=0.5)
    plt.plot(k, np.imag(x_test), 'C1o-', lw=0.5)
    plt.xlabel(r'sample $k$')
    plt.ylabel(r'$x[k]$')
    plt.legend()
    plt.grid(True)

    print(np.allclose(np.fft.ifft(X_test), x_test))
    print('DC is 1 as expected: ', np.mean(x_test))

# Construct the inverse Fourier matrix W_inv for IDFT
W_inv = np.exp(2j * np.pi * K / N)

# Reconstruct the signal using IDFT
x_reconstructed = (1 / N) * np.dot(W_inv, X)

# Display the matrices K and W
print("Matrix K:\n", K)
print("\nMatrix W:\n", W.round(1))

# Display the reconstructed signal
print("\nReconstructed signal x (IDFT):\n", x_reconstructed)
```

```python
# Plot the synthesized signal (real and imaginary parts)
plt.figure(figsize=(10, 6))
plt.stem(k, np.real(x_reconstructed), markerfmt='C0o', basefmt='C0:',
linefmt='C0-', label='Reconstructed (Real part)')
plt.stem(k, np.imag(x_reconstructed), markerfmt='C1o', basefmt='C1:',
linefmt='C1--', label='Reconstructed (Imag part)')
plt.title("Synthesized Signal using IDFT")
plt.xlabel("Sample index k")
plt.ylabel("Amplitude")
plt.legend()
plt.grid(True)
plt.show()
```

```
False
DC is 1 as expected:  (0.6+1.332267629550188e-16j)
Matrix K:
 [[ 0  0  0  0  0  0  0  0  0  0]
 [ 0  1  2  3  4  5  6  7  8  9]
 [ 0  2  4  6  8 10 12 14 16 18]
 [ 0  3  6  9 12 15 18 21 24 27]
 [ 0  4  8 12 16 20 24 28 32 36]
 [ 0  5 10 15 20 25 30 35 40 45]
 [ 0  6 12 18 24 30 36 42 48 54]
 [ 0  7 14 21 28 35 42 49 56 63]
 [ 0  8 16 24 32 40 48 56 64 72]
 [ 0  9 18 27 36 45 54 63 72 81]]

Matrix W:
 [[ 1. +0.j   1. +0.j   1. +0.j   1. +0.j   1. +0.j   1. +0.j   1.
+0.j
   1. +0.j   1. +0.j   1. +0.j ]
 [ 1. +0.j   0.8-0.6j  0.3-1.j  -0.3-1.j  -0.8-0.6j -1. -0.j  -
0.8+0.6j
  -0.3+1.j   0.3+1.j   0.8+0.6j]
 [ 1. +0.j   0.3-1.j  -0.8-0.6j -0.8+0.6j  0.3+1.j   1. +0.j   0.3-1.j
  -0.8-0.6j -0.8+0.6j  0.3+1.j ]
 [ 1. +0.j  -0.3-1.j  -0.8+0.6j  0.8+0.6j  0.3-1.j  -1. -0.j   0.3+1.j
   0.8-0.6j -0.8-0.6j -0.3+1.j ]
 [ 1. +0.j  -0.8-0.6j  0.3+1.j   0.3-1.j  -0.8+0.6j  1. +0.j  -0.8-
0.6j
   0.3+1.j   0.3-1.j  -0.8+0.6j]
 [ 1. +0.j  -1. -0.j   1. +0.j  -1. -0.j   1. +0.j  -1. +0.j   1. +0.j
  -1. -0.j   1. +0.j  -1. +0.j ]
 [ 1. +0.j  -0.8+0.6j  0.3-1.j   0.3+1.j  -0.8-0.6j  1. +0.j  -
0.8+0.6j
   0.3-1.j   0.3+1.j  -0.8-0.6j]
 [ 1. +0.j  -0.3+1.j  -0.8-0.6j  0.8-0.6j  0.3+1.j  -1. -0.j   0.3-1.j
   0.8+0.6j -0.8+0.6j -0.3-1.j ]
 [ 1. +0.j   0.3+1.j  -0.8+0.6j -0.8-0.6j  0.3-1.j   1. +0.j   0.3+1.j
```

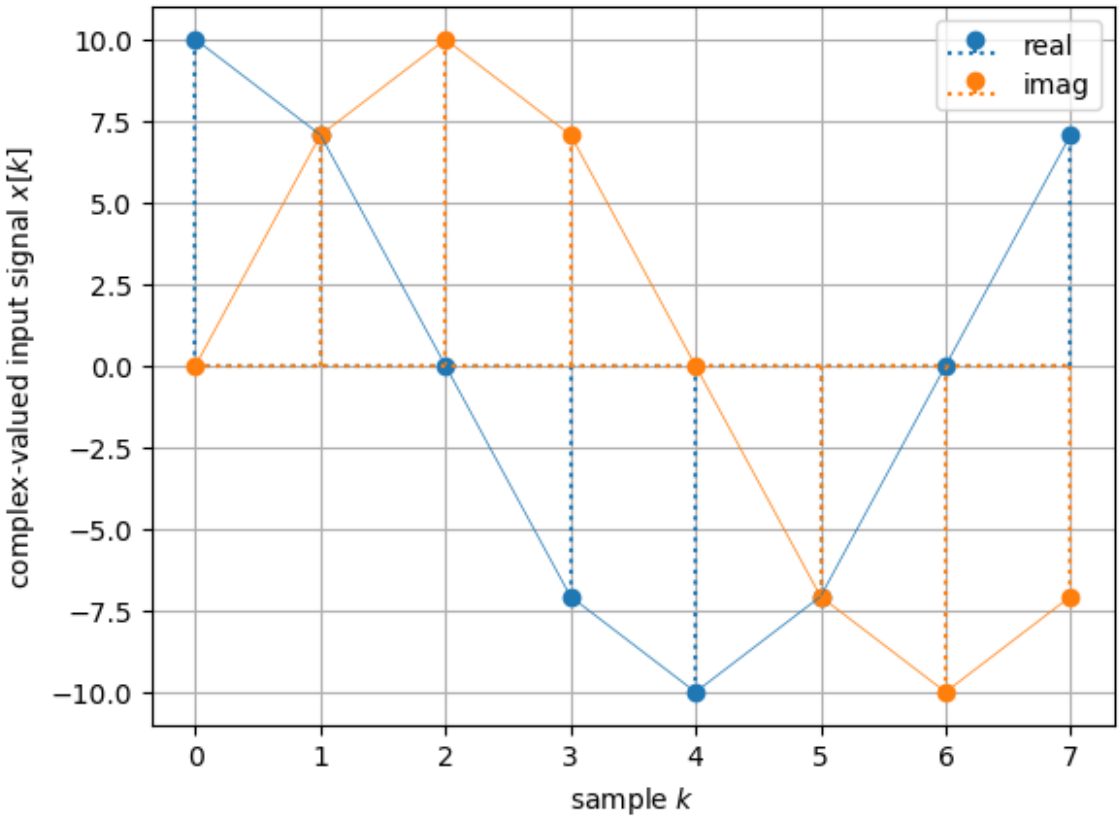
```
  -0.8+0.6j -0.8-0.6j  0.3-1.j ]
 [ 1. +0.j   0.8+0.6j  0.3+1.j  -0.3+1.j  -0.8+0.6j -1. +0.j  -0.8-
0.6j
  -0.3-1.j   0.3-1.j   0.8-0.6j]]

Reconstructed signal x (IDFT):
 [ 6.00000000e+00+1.24344979e-15j  2.00000000e+00+7.54951657e-16j
  4.00000000e+00+1.43218770e-15j  4.00000000e+00-2.18713936e-15j
  4.00000000e+00-1.11022302e-17j  5.00000000e+00-1.66651009e-15j
 -1.42108547e-15-2.77555756e-16j -3.90798505e-15-2.53130850e-15j
 -1.77635684e-15+3.68594044e-15j -1.24344979e-15-1.70419234e-15j]
```






```
#LAB1 DSP Synthesize a discrete-time signal by using the IDFT in
matrix notation for different values of N.Show the matrices W and K.
Plot the signal synthesized.

import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv
from numpy.fft import fft, ifft
#from scipy.fft import fft, ifft

N = 2**3  # signal block length
k = np.arange(N)  # all required sample/time indices
A = 10  # signal amplitude

tmpmu = 2-1/2  # DFT eigenfrequency worst case
tmpmu = 1  # DFT eigenfrequency best case

x = A * np.exp(tmpmu * +1j*2*np.pi/N * k)

# plot
plt.stem(k, np.real(x), markerfmt='C0o',
         basefmt='C0:', linefmt='C0:', label='real')
plt.stem(k, np.imag(x), markerfmt='C1o',
         basefmt='C1:', linefmt='C1:', label='imag')
# note that connecting the samples by lines is actually wrong, we
```

```
# use it anyway for more visual convenience:
plt.plot(k, np.real(x), 'C0-', lw=0.5)
plt.plot(k, np.imag(x), 'C1-', lw=0.5)
plt.xlabel(r'sample $k$')
plt.ylabel(r'complex-valued input signal $x[k]$')
plt.legend()
plt.grid(True)
```



theory &own actions on 1st lab

.

.

.

.

We will now perform an DFT of $x[k]$ since we are interested in the frequency spectrum of it.

DFT Definition

The discrete Fourier transform pair for a discrete-time signal $x[k]$ with sample index k and the corresponding DFT spectrum $X[\mu]$ with frequency index μ is given as \begin{align} \text{DFT}: X[\mu]=&\sum_{k=0}^{N-1}x[k]\cdot\mathrm{e}^{-\mathrm{j}\frac{2\pi}{N}k\mu}\ \text{IDFT}:

x[k]=\frac{1}{N}&\sum_{\mu=0}^{N-1}X[\mu]\cdot\mathrm{e}^{+\mathrm{j}\frac{2\pi}{N}k\mu} \end{align}

Note the sign reversal in the exp()-function and the $1/N$ normalization in the IDFT. This convention is used by the majority of DSP text books and also in Python's `numpy.fft.fft()`, `numpy.fft.ifft()` and Matlab's `fft()`, `ifft()` routines.

DFT and IDFT with For-Loops

We are now going to implement the DFT and IDFT with for-loop handling. While this might be helpful to validate algorithms in its initial development phase, this should be avoided for practical used code in the field: for-loops are typically slow and very often more complicated to read than appropriate set up matrices and vectors. Especially for very large N the computation time is very long.

Anyway, the for-loop concept is: the DFT can be implemented with an outer for loop iterating over μ and an inner for loop summing over all k for a specific μ.

We use variable with _ subscript here, in order to save nice variable names for the matrix based calculation.

```
# DFT with for-loop:
X_ = np.zeros((N, 1), dtype=complex)  # alloc RAM, init with zeros
for mu_ in range(N):  # do for all DFT frequency indices
    for k_ in range(N):  # do for all sample indices
        X_[mu_] += x[k_] * np.exp(-1j*2*np.pi/N*k_*mu_)
```

IDFT with outer and inner looping reads as follows.

```
# IDFT with for-loop:
x_ = np.zeros((N, 1), dtype=complex)  # alloc RAM, init with zeros
for k_ in range(N):
    for mu_ in range(N):
        x_[k_] += X_[mu_] * np.exp(+1j*2*np.pi/N*k_*mu_)
x_ *= 1/N  # normalization in the IDFT stage
```

Besides exchanged variables, main differences are sign reversal in exp() and the $1/N$ normalization. This is expected due to the DFT/IDFT equation pair given above.

DFT and IDFT with Matrix Multiplication

Now we do a little better: We should think of the DFT/IDFT in terms of a matrix operation setting up a set of linear equations.

For that we define a column vector containing the samples of the discrete-time signal $x[k]$ \ begin{equation} \mathbf{x}_k = (x[k=0], x[k=1], x[k=2], \dots , x[k=N-1])^\mathrm{T} \ end{equation}

and a column vector containing the DFT coefficients $X[\mu]$

\begin{equation} \mathbf{x}_\mu = (X[\mu=0], X[\mu=1], X[\mu=2], \dots, X[\mu=N-1])^\mathrm{T} \end{equation}

Then, the matrix operations

\begin{align} \text{DFT: } & \mathbf{x}_\mu = \mathbf{W}^* \mathbf{x}_k\ \text{IDFT: } & \mathbf{x}*k = \frac{1}{N} \mathbf{W} \mathbf{x}*\mu \end{align}

hold.

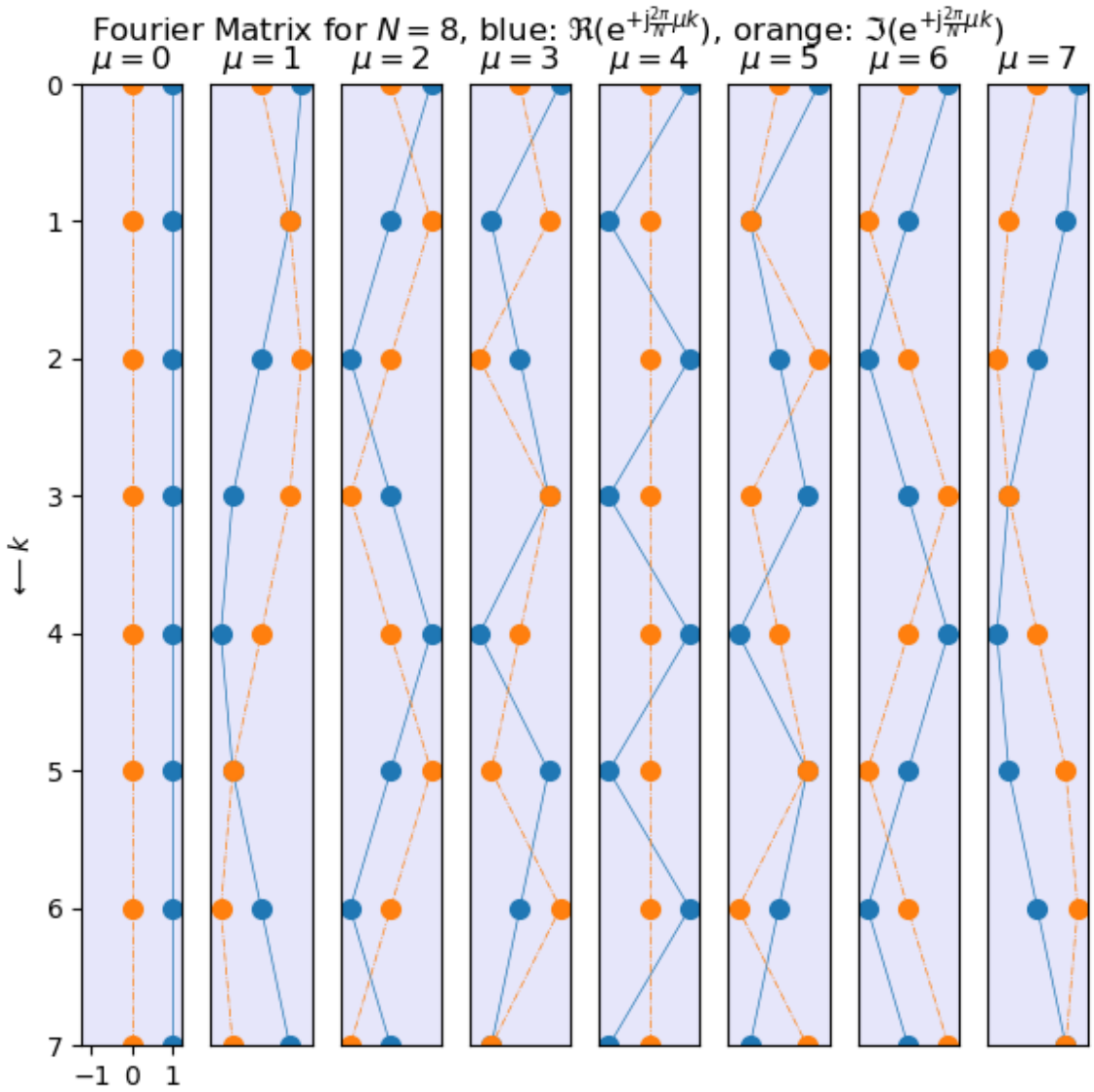
$()^T$ is the transpose, $()^*$ is the conjugate complex.

The $N \times N$ Fourier matrix is defined as (element-wise operation $\odot$) \begin{equation} \mathbf{W} = \mathrm{e}^{+\mathrm{j}\frac{2\pi}{N} \odot \mathbf{K}} \end{equation} using the so called twiddle factor (note that the sign in the exp() is our convention) \begin{equation} W_N = \mathrm{e}^{+\mathrm{j}\frac{2\pi}{N}} \end{equation} and the outer product \begin{equation} \mathbf{K} = \begin{bmatrix} 0\ 1\ 2\ \vdots\ N-1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 2 & \cdots & N-1 \end{bmatrix} \end{equation} containing all possible products $k\ \mu$ in a suitable arrangement.

For the simple case $N=4$ these matrices are \begin{align} \mathbf{K} = \begin{bmatrix} 0 & 0 & 0 & 0\ 0 & 1 & 2 & 3\ 0 & 2 & 4 & 6\ 0 & 3 & 6 & 9 \end{bmatrix} \rightarrow \mathbf{W} = \begin{bmatrix} 1 & 1 & 1 & 1\ 1 & +\mathrm{j} & -1 & -\mathrm{j}\ 1 & -1 & 1 & -1\ 1 & -\mathrm{j} & -1 & +\mathrm{j} \end{bmatrix} \end{align}

```
# k = np.arange(N)  # all required sample/time indices, already
defined above

# all required DFT frequency indices, actually same entries like in k
mu = np.arange(N)

# set up matrices
K = np.outer(k, mu)  # get all possible entries k*mu in meaningful
arrangement
W = np.exp(+1j * 2*np.pi/N * K)  # analysis matrix for DFT

# visualize the content of the Fourier matrix
# we've already set up (use other N if desired):
# N = 8
# k = np.arange(N)
# mu = np.arange(N)
# W = np.exp(+1j*2*np.pi/N*np.outer(k, mu))  # set up Fourier matrix

fig, ax = plt.subplots(1, N)
fig.set_size_inches(6, 6)
fig.suptitle(
    r'Fourier Matrix for $N=$%d, blue: $\Re(\mathrm{e}^{+\mathrm{j} \
frac{2\pi}{N} \mu k})$, orange: $\Im(\mathrm{e}^{+\mathrm{j} \frac{2\
pi}{N} \mu k})$' % N)

for tmp in range(N):
```

```
    ax[tmp].set_facecolor('lavender')
    ax[tmp].plot(W[:, tmp].real, k, 'C0o-', ms=7, lw=0.5)
    ax[tmp].plot(W[:, tmp].imag, k, 'C1o-.', ms=7, lw=0.5)
    ax[tmp].set_ylim(N-1, 0)
    ax[tmp].set_xlim(-5/4, +5/4)
    if tmp == 0:
        ax[tmp].set_yticks(np.arange(0, N))
        ax[tmp].set_xticks(np.arange(-1, 1+1, 1))
        ax[tmp].set_ylabel(r'$\longleftarrow k$')
    else:
        ax[tmp].set_yticks([], minor=False)
        ax[tmp].set_xticks([], minor=False)
    ax[tmp].set_title(r'$\mu=$%d' % tmp)
fig.tight_layout()
fig.subplots_adjust(top=0.91)

fig.savefig('fourier_matrix.png', dpi=300)

# TBD: row version for analysis
```



Fourier Matrix Properties

The DFT and IDFT basically solve two sets of linear equations, that are linked as forward and inverse problem.

This is revealed with the important property of the Fourier matrix

\begin{equation} \mathbf{W}^{-1} = \frac{\mathbf{W}^\mathrm{H}}{N} = \frac{\mathbf{W}^\mathrm{*}}{N}, \end{equation}

the latter holds since the matrix is symmetric.

Thus, we see that by our convention, the DFT is the inverse problem (signal analysis) and the IDFT is the forward problem (signal synthesis)

\begin{align} \text{DFT: } & \mathbf{x}_\mu = \mathbf{W}^* \mathbf{x}*k |rightarrow | *mathbf{x}*\mu = N \mathbf{W}^{-1} , \mathbf{x}_k\ \text{IDFT: } & \mathbf{x}*k = |*frac{1}{N} | *mathbf{W} |*mathbf{x}*\mu. \end{align}

The occurrence of the N, $1/N$ factor is due to the prevailing convention in signal processing literature.

If the matrix is normalised as $\frac{W}{\sqrt{N}}$, a so called unitary matrix results, for which the important property \begin{equation} (\frac{\mathbf{W}}{\sqrt{N}})^\mathrm{H} , (\frac{\mathbf{W}}{\sqrt{N}}) = \mathbf{I} = (\frac{\mathbf{W}}{\sqrt{N}})^{-1} , (\frac{\mathbf{W}}{\sqrt{N}}) \end{equation} holds, i.e. the complex-conjugate, transpose is equal to the inverse $\left(\frac{W}{\sqrt{N}}\right)^H = \left(\frac{W}{\sqrt{N}}\right)^{-1}$ and due to the matrix symmetry also $\left(\frac{W}{\sqrt{N}}\right)^{\dot{c}} = \left(\frac{W}{\sqrt{N}}\right)^{-1}$ is valid.

This tells that the matrix $\frac{W}{\sqrt{N}}$ is **orthonormal**, i.e. the matrix spans a orthonormal vector basis (the best what we can get in linear algebra world to work with) of N normalized DFT eigensignals.

So, DFT and IDFT is transforming vectors into other vectors using the vector basis of the Fourier matrix.

Check DFT Eigensignals and -Frequencies

The columns of the Fourier matrix W contain the eigensignals of the DFT. These are \ begin{align} w_\mu[k] = \cos(\frac{2\pi}{N} k \mu) + \mathrm{j} \sin(\frac{2\pi}{N} k \mu) \ end{align} since we have intentionally set up the matrix this way.

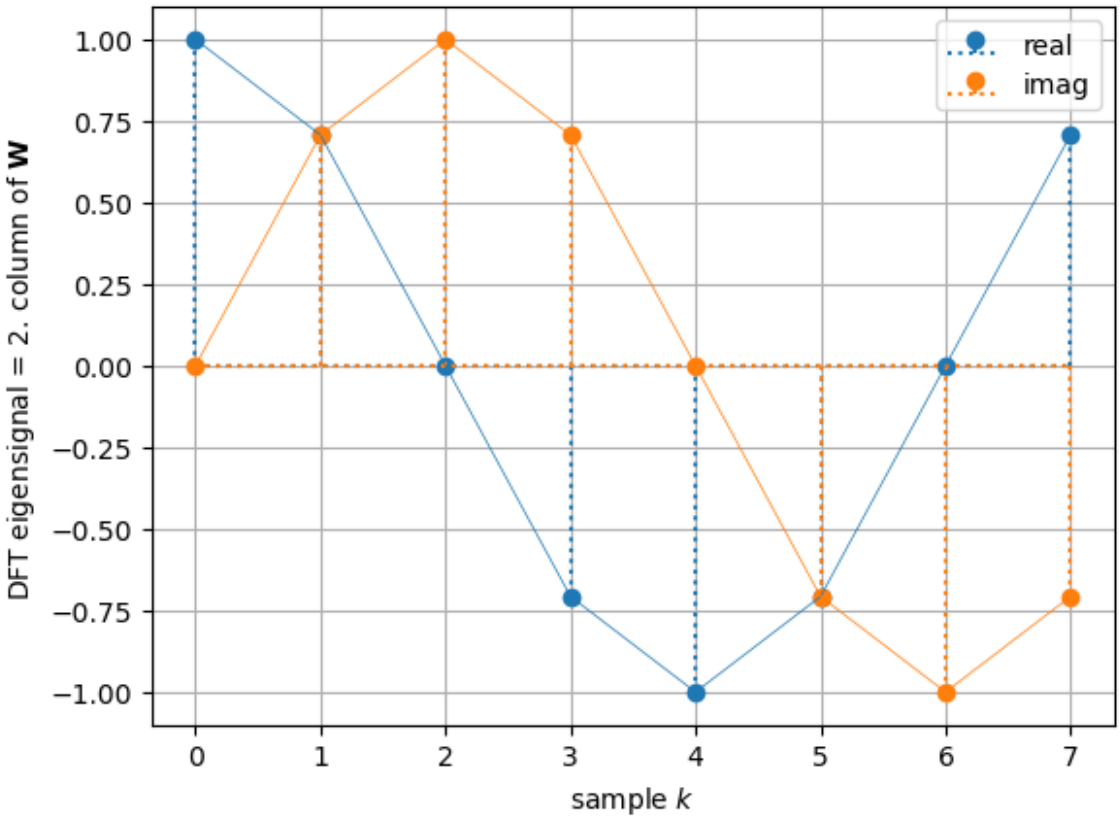
The plot below shows the eigensignal for $\mu = 1$, which fits again one signal period in the block length N. For $\mu = 2$ we obtain two periods in one block.

The eigensignals for $0 \leq \mu \leq N-1$ therefore exhibit a certain digital frequency, the so called DFT eigenfrequencies.

What eigensignal corresponds to $\mu = 0$?...

```
tmpmu = 1  # column index

plt.stem(k, np.real(W[:, tmpmu]), label='real',
         markerfmt='C0o', basefmt='C0:', linefmt='C0:')
plt.stem(k, np.imag(W[:, tmpmu]), label='imag',
         markerfmt='C1o', basefmt='C1:', linefmt='C1:')
# note that connecting the samples by lines is actually wrong, we
# use it anyway for more visual convenience
plt.plot(k, np.real(W[:, tmpmu]), 'C0-', lw=0.5)
plt.plot(k, np.imag(W[:, tmpmu]), 'C1-', lw=0.5)
plt.xlabel(r'sample $k$')
plt.ylabel(r'DFT eigensignal = '+str(tmpmu+1)+ r'. column of $\
mathbf{W}$')
```

```
plt.legend()
plt.grid(True)
```



The nice thing about the chosen eigenfrequencies, is that the eigensignals are **orthogonal**.

This choice of the vector basis is on purpose and one of the most important ones in linear algebra and signal processing.

We might for example check orthogonality with the **complex** inner product of some matrix columns.

```
np.dot(np.conj(W[:, 0]), W[:, 0])  # same eigensignal, same
eigenfrequency
# np.vdot(W[:,0],W[:,0])  # this is the suitable numpy function
```

```
(8+0j)
```

```
np.dot(np.conj(W[:, 0]), W[:, 1])  # different eigensignals
# np.vdot(W[:,0],W[:,1])  # this is the suitable numpy function
# result should be zero, with numerical precision close to zero:
```

```
(-5.551115123125783e-16-2.220446049250313e-16j)
```

Initial Example: IDFT Signal Synthesis for N=8

Let us synthesize a discrete-time signal by using the IDFT in matrix notation for $N = 8$.

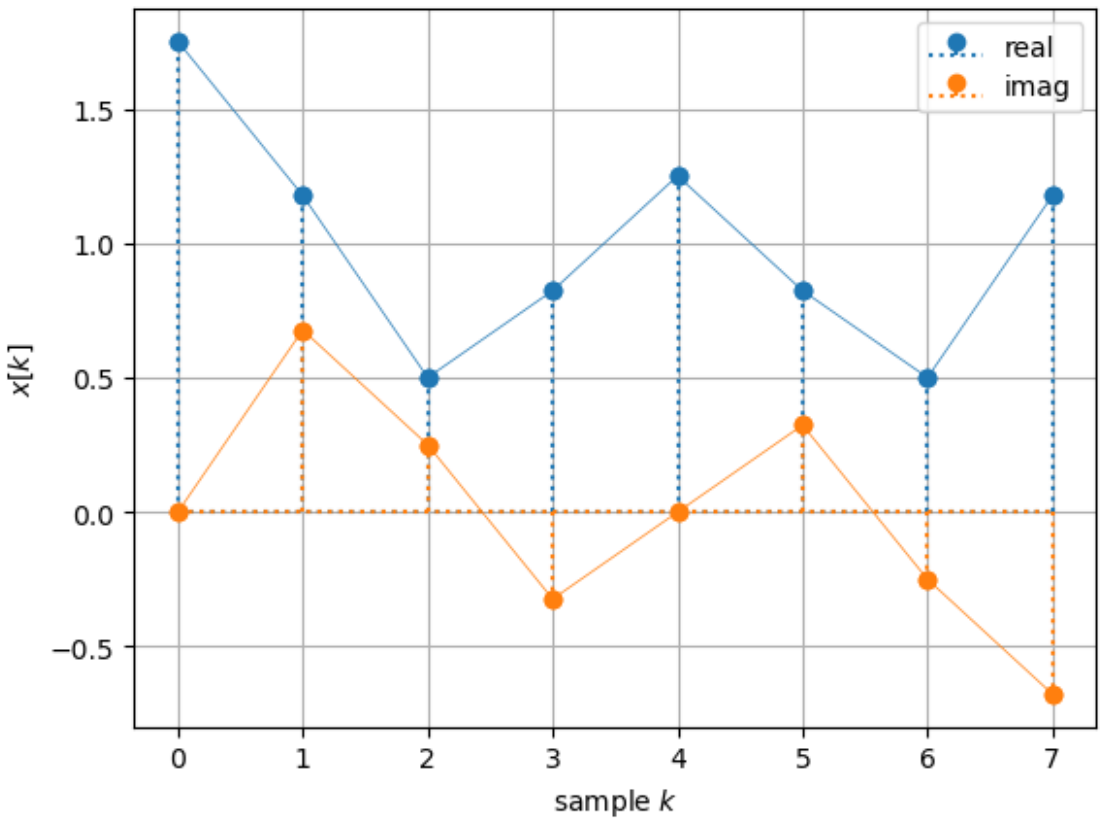
The signal should contain a DC value, the first and second eigenfrequency with different amplitudes, such as

\begin{equation} \mathbf{x}_\mu = [8, 2, 4, 0, 0, 0, 0, 0]^\text{T} \end{equation}

using large `X_test` in code.

```
if N == 8:
    X_test = np.array([8, 2, 4, 0, 0, 0, 0, 0])
    # x_test = 1/N*W@X_test  # >= Python3.5
    x_test = 1/N * np.matmul(W, X_test)

    plt.stem(k, np.real(x_test), label='real',
             markerfmt='C0o', basefmt='C0:', linefmt='C0:')
    plt.stem(k, np.imag(x_test), label='imag',
             markerfmt='C1o', basefmt='C1:', linefmt='C1:')
    # note that connecting the samples by lines is actually wrong, we
    # use it anyway for more visual convenience
    plt.plot(k, np.real(x_test), 'C0o-', lw=0.5)
    plt.plot(k, np.imag(x_test), 'C1o-', lw=0.5)
    plt.xlabel(r'sample $k$')
    plt.ylabel(r'$x[k]$')
    plt.legend()
    plt.grid(True)

    # check if results are identical with numpy ifft package
    print(np.allclose(ifft(X_test), x_test))
    print('DC is 1 as expected: ', np.mean(x_test))
```

```
True
DC is 1 as expected:  (1+1.3877787807814457e-17j)
```



This is a linear combination of the Fourier matrix columns, which are the DFT eigensignals, as

```
if N == 8:
    x_test2 = X_test[0] * W[:, 0] + X_test[1] * W[:, 1] + X_test[2] *
W[:, 2]
```

We don't need summing the other columns, since their DFT coefficients in `X_test` are zero.

Finally, normalizing yields the IDFT.

```
if N == 8:
    x_test2 *= 1/N
    print(np.allclose(x_test, x_test2))  # check with result before
```

```
True
```

Initial Example: DFT Spectrum Analysis for N=8

Now, let us calculate the DFT of the signal `x_test`. As result, we'd expect the DFT vector

\begin{equation} \mathbf{x}_\mu = [8, 2, 4, 0, 0, 0, 0, 0]^\text{T} \end{equation}

that we started from.

```
if N == 8:
    # X_test2 = np.conj(W)@x_test  # >= Python3.5
    X_test2 = np.matmul(np.conj(W), x_test)  # DFT, i.e. analysis
    print(np.allclose(X_test, X_test2))  # check with result before
```

```
True
```

This looks good. It is advisable also to check against the `numpy.fft` implementation:

```
if N == 8:
    print(np.allclose(fft(x_test), X_test))
```

```
True
```

Besides different quantization errors in range $10^{-15..-16}$ (which is prominent even with 64Bit double precision calculation) all results produce the same output.

The analysis stage for the discrete-time signal domain, i.e. the DFT can be reinvented by some intuition: How 'much' of the reference signal $w_{\text{column i}}$ (any column in W) is contained in the discrete-time signal x_k that is to be analysed.

In signal processing / statistic terms we look for the amount of correlation of the signals $w_{\text{column i}}$ and x_k.

In linear algebra terms we are interested in the projection of x_k onto $w_{\text{column i}}$, because the resulting length of this vector reveals the amount of correlation, which is precisely one DFT coefficient $X[\cdot]$.

The complex inner products $w_{\text{column i}}^{\text{H}} \cdot x_k$ reveals these searched quantities.

```
if N == 8:
    print(np.conj(W[:, 0])@x_test)
    print(np.conj(W[:, 1])@x_test)
    print(np.conj(W[:, 2])@x_test)
```

```
(8+5.551115123125783e-17j)
(1.9999999999999996-1.7998605749412785e-16j)
(4-2.5382063323692944e-16j)
```

Doing this for all columns of matrix W, all DFT coefficients are obtained, such as

\begin{align} X[\mu=0] =& \mathbf{w}_{\text{column 1}}^\text{H} \cdot \mathbf{x}*k| X[|mu=1] =& |mathbf{w}*{\text{column 2}}^\text{H} \cdot \mathbf{x}*k| X[|mu=2] =& |mathbf{w}*{\text{column 3}}^\text{H} \cdot \mathbf{x}*k| X[|mu=3] =& |mathbf{w}*{\text{column 4}}^\text{H} \cdot \mathbf{x}*k| &|vdots| X[|mu=N-1] =& |mathbf{w}*{\text{column N}}^\text{H} \cdot \mathbf{x}_k. \end{align}

Naturally, all operations can be merged to one single matrix multiplication using the conjugate transpose of W.

\begin{equation} \mathbf{x}_\mu = \mathbf{W}^\text{H} \cdot \mathbf{x}_k = \mathbf{W}^* \cdot \mathbf{x}_k \end{equation}

That's what we have performed with the single liner `X_test2 = np.matmul(np.conj(W), x_test)`

Example: Plot the DFT Magnitude Spectrum

We should now be familiar with the DFT and IDFT basic idea.

Now, let us **return to our initially created signal** `x` at the very beginning of this notebook. We want to explore and learn to interpret the DFT magnitude spectrum of it. So, we'd perform a DFT first.

```
X = fft(x)
# print(np.allclose(np.conj(W)@x, X))  # >=Python 3.5
print(np.allclose(np.matmul(np.conj(W), x), X))
```

```
True
```

Next, let us plot the magnitude of the spectrum over μ.

- We should play around with the variable `tmpmu` when defining the input signal at the very beginning of the notebook. For example we can check what happens for `tmpmu = 1`, `tmpmu = 2` and run the whole notebook to visualize the actual magnitude spectra.

We should recognize the link of the 'energy' at μ in the magnitude spectrum with the chosen `tmpmu`.

- We can apply any real valued `tmpmu` for creating the input signal, for example
  - `tmpmu = N+1`, `tmpmu = N+2`
  - `tmpmu = 1.5`

We should explain what happens in these cases. Recall periodicity and eigenfrequencies/-signals as fundamental concepts.

```
plt.stem(mu, np.abs(X)/N, markerfmt='C0o', basefmt='C0:',
linefmt='C0:')
# plt.plot(mu, np.abs(X)/N, 'C0', lw=1)  # this is here a misleading
plot and hence not used
plt.xlabel(r'DFT eigenfrequency $\mu$')
plt.ylabel(r'DFT spectrum magnitude $\frac{|X[\mu]|}{N}$')
plt.grid(True)
```