Signal and Image Processing

Old A3 Questions

Research Group Neuroinformatics
University of Vienna

Contents

1 Signals

1.1 Plotting Signals

In this exercise, you will define auxiliary functions to plot signals in the time and frequency domains. You are allowed to reuse these to plot signals in other exercises.

a) Time domain

Define a function `plot_time(x, fs)` to plot a signal in the time domain, where x is the signal (real-valued numpy array of arbitrary length), and fs is the sampling frequency in Hz (float). The x-axis should display the time in seconds and should be labeled accordingly.

b) Frequency domain

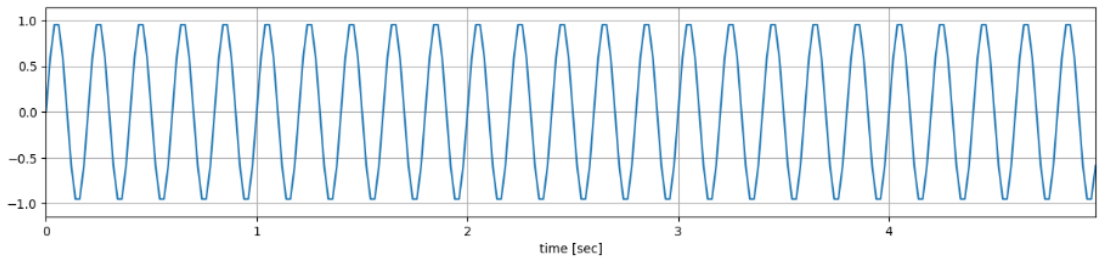
Define a function `plot_freq(X, fs)` to plot a signal in the frequency domain, where X is the signal in the frequency domain (complex-valued numpy array of arbitrary length), and fs is the sampling frequency in Hz (float). The function should plot two side-by-side plots, showing the amplitude spectrum on the left and the phase spectrum on the right. The x-axis should display the frequency in Hz, with 0Hz in the center of the plot. Make sure that your function is able to correctly plot signals of different lengths and sampling frequencies!

Hint: Use `numpy.fft.fftshift` to shift the arrays obtained by `numpy.fft.fft`.

c) Test the functions

Define a sine signal with a frequency of 5Hz, with a length of 5 seconds, and with a sampling frequency of 50Hz. Compute its DFT using `numpy.fft.fft`. Plot the signal in the time and frequency domain using your previously defined functions. The results should look like this:

```
In [7]: plot_time(x, fs)
```



```
In [8]: plot_freq(X, fs)
```

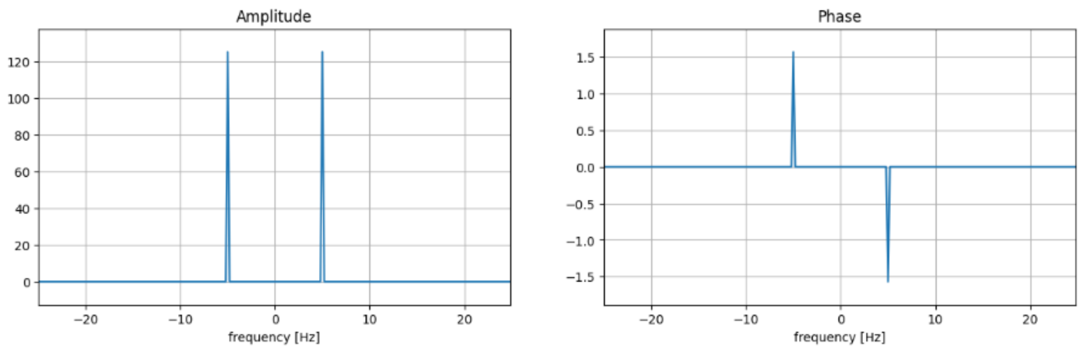


Figure 1: Time and frequency domain plots of a 5Hz sine wave (5s duration, 50Hz sampling rate).

1.2 Convolution

Write a function `convolution(x,h,domain,overlap,periodic)` to compute 1D convolution in the temporal and spectral domain. The function should take two-time signals `'x'` and `'h'` as input, call other sub-functions (described below), and return a convoluted signal. In addition, the function should suffice the following requirements:

- Be able to handle signals of any length.
- Parameter `domain` has two possible options; `‘temporal’` or `‘spectral’`.
- Sub-functions:
  - (i) Implement inverse, and discrete fourier transform (DFT and iDFT) functions to transform signals from temporal into spectra domain.
  - (ii) When `domain==‘temporal’`, call sub-function `conv(x,h,overlap,period)` to perform both linear and circular convolutions in time domain. Define the function as described below:
    - Parameters `overlap` specifies the condition for the mode of convolution while `periodic` specifies either linear or circular convolution.
    - When `overlap==True`, the function should return the output signal which is same as `numpy.convolve(mode='valid')`, otherwise returns result similar to `numpy.convolve(mode='full')`.
    - Compute all circular convolutions with period `N = max(len(x),len(h))`.
    - For all cases of linear convolution, set `overlap` as required, and `periodic==False`; while for all cases of circular convolution, set `overlap==True`.

    Hint: Use the convolution property of the Fourier Transform.
    Find attached in the assignment resources, `Circular Convolution.pdf` as a guide to writing the circular convolution function.
  - (iii) When `domain==‘spectral’`, computes both linear and circular convolution in frequency domain. Define function `CONV(x,h,overlap,period)`. The conditions for overlap and periodic as stated above holds for this function as well.

$$\text{Hint: } y[n] = x[n] * h[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n-k]$$

(a) Let x = [1, 2, 0, 1] and h = [2, 2, 1, 1]. Use the 2 signals to test your written function against that of `numpy.convolve`. Apply your written DFT function and plot the amplitude spectrum on each. [**2 points**]

(b) Perform convolution of `x` and `h` in both the temporal (set overlap==False) and spectral domain. **Please stem-plot the resulting signals, label and title each plots appropriately.** [**6 points**]
Note: Assume your defined signals `x` and `h` to be 0 outside of range.

(c) Compare your result above with that of `numpy.convolve`. Since `numpy.convolve` does not have circular convolution implemented, please only check for both overlap cases of your linear convolution in the temporal domain. [**2 points**]

(d) Compare results derived from temporal and spectral convolutions, are they the same? If not why? [**2 points**]

2 Spectral Analysis

2.1 Discrete Fourier Transform

Instructions:

- Label the x-axis in the unit of Hz and should range between $\left[-\frac{f_s}{2}, \frac{f_s}{2}\right]$
- You are allowed to use `numpy.fft.fft` and `numpy.fft.fftshift`.

(a) Let $x_1[n] = \sum_{k\in\{5,30,80,150,250\}} \sin(2\pi \frac{k}{N} n)$ of length $N = 1500$ and a sampling frequency $f_s = 600\text{Hz}$. Plot $x_1[n]$ and its frequency spectrum $X_1(f)$. (**Hint:** $X_1(f)$ is a complex-valued signal, you can either plot its real- and imaginary part separately or use 3D plot.)

(b) From $X_1(f)$ obtained in (a), plot the amplitude $|X_1(f)|$ and phase spectrum $\angle X_1(f)$. (**Hint:** Use `numpy.unwrap` to bound the angular frequency ω between $[-\pi, \pi]$; $\omega = \frac{2\pi f}{f_s}$)

(c) **Effect of the sampling frequency:** Let $x_2[n]$ be identical to $x_1[n]$ but with different sampling frequency $f_s = 1000$ Hz. Plot $x_2[n]$ and the amplitude spectrum $|X_2(f)|$. Compare your results with $x_1[n]$ and $|X_1(f)|$, are they identical? **Explain your result.**

(d) **Effect of the number of samples:** Let $x_3[n]$ be identical to $x_1[n]$ but with different number of samples, i.e., $n \in [0, 149]$. Plot $x_3[n]$ and the amplitude spectrum $|X_3(f)|$. Compare your results with $x_1[n]$ and $|X_1(f)|$, are they identical? **Explain your result.**

(e) Define $y[n]$ with same equation as $x_1[n]$, with the same sampling frequencies and number of samples as $x_1[n]$ but with **cosine** function instead of sine. With plots, compare the amplitude and phase spectrum, real and imaginary part of $y[n]$ with that of $x_1[n]$. **Explain the differences between using cosine and sine as the signal function.**

2.2 The Hilbert transform

Whereas the Fourier transform removes the temporal information, the Hilbert transform allows you to analyze the dynamics of a signal, i.e., how the amplitude, the phase, and the frequency change over time.

1. Generate a sine wave according to

$$\mathbf{x[n]} = A \cdot \sin\left(\frac{2\pi f}{f_s} \cdot n\right),$$

with an amplitude $A = 3$ (a.u.), a frequency $f = 2$ Hz, a sampling frequency $f_s = 1$ kHz, and a vector n with length $N = 2000$. **Plot** the signal $\mathbf{x[n]}$ and set the unit of the x-axis to **seconds**. [**1 point**]

2. Write a function `analytic_signal(x)`, which returns the analytic signal $\mathbf{z[n]}$ and is given the signal $\mathbf{x[n]}$ as parameter `x`. The analytic signal $\mathbf{z[n]}$ is calculated via the iDFT$\{\mathbf{Z}(\omega)\}$, with **[2 points]**

$$\mathbf{Z}(\omega) = \begin{cases} 2\mathbf{X}(\omega), & 0 \leq \omega \leq \pi \\ 0, -\pi \leq \omega \leq 0 \end{cases}$$

   Note: Use your self-written functions to calculate the DFT and iDFT.

3. Calculate the instantaneous amplitude, instantaneous phase and instantaneous frequency from your analytic signal $\mathbf{z[n]}$, which are defined as: **[3 points]**

   - Instantaneous amplitude: $|\mathbf{z[n]}|$
   - Instantaneous phase: $\angle\mathbf{z[n]}$
   - Instantaneous frequency: $\dfrac{d\angle\mathbf{z[n]}}{dn}$

   (i) **Plot** the instantaneous amplitude together with the real part $Re\{\mathbf{z[n]}\}$ and the imaginary part $Im\{\mathbf{z[n]}\}$ of the analytic signal in the same plot window.

   (ii) **Plot** the instantaneous phase and the instantaneous frequency. How are the frequency bursts related to the instantaneous phase?

   (iii) **Calculate and plot** the instantaneous phase and the instantaneous frequency again by applying the function `numpy.unwrap()` after you have calculated the angle of the analytic signal $\mathbf{z[n]}$. What differences can you observe compared to task (ii)?

   Hints:

   - Use the function `numpy.angle()` to calculate $\angle\mathbf{z[n]}$
   - In order to calculate the instantaneous frequency in Hz, you have to convert the instantaneous phase from radian.
   - Set the limits of the y-axis to 0 and $2f$ to display the instantaneous frequency properly.

   Note: For all your plots, carefully label all axes and set the unit of the x-axis to seconds.

4. Generate a signal $\mathbf{x\_2[n]}$ with two frequency components, by adding a second sine wave (with $f = 3$ Hz) to your signal $\mathbf{x[n]}$ **[2 points]**

   (i) **Calculate** the analytic signal $\mathbf{z\_2[n]}$ from your signal $\mathbf{x\_2[n]}$

   (ii) **Calculate and plot** the instantaneous amplitude together with the real part $Re\{\mathbf{z\_2[n]}\}$ and the imaginary part $Im\{\mathbf{z\_2[n]}\}$ of your new analytic signal $\mathbf{z\_2[n]}$ in the same plot window.

   (iii) **Calculate and plot** the instantaneous phase and the instantaneous frequency by applying the function `numpy.unwrap()` after you have calculated the angle of the analytic signal $\mathbf{z\_2[n]}$. How is the instantaneous frequency related to the two frequency components of your signal $\mathbf{x\_2[n]}$?

   Note: For all your plots, carefully label all axes and set the unit of the x-axis to seconds.

2.3 Multitaper Spectral Analysis

In this exercise, you will implement a function for multitaper spectral analysis, a different way of estimating the spectrum of a signal, which is typically less noisy.

a) Power spectral density

Instead of examining the frequency spectrum that results from the D(T)FT, one often instead looks at the **power spectral density (PSD)**. It shows how much of the signal's energy is concentrated on each frequency band.

Let $X(\omega)$ be the DFT of a signal $x \cdot w$ of length N, where w is a window function. Then, its PSD is defined as

$$\mathrm{PSD}\{x\}(\omega) = \frac{2|X(\omega) \cdot 1_{\{\omega \geq 0\}}(\omega)|^2}{f_s \cdot S}, \text{ where } S = \sum_{i=1}^{N} w[n]^2,$$

f_s is the sampling frequency and $\mathbf{1}_A$ is the characteristic function, i.e.,

$$\mathbf{1}_A(x) = \begin{cases} 1 \text{ if } x \in A \\ 0 \text{ otherwise.} \end{cases}$$

Implement a function `psd(x, w, fs)` to calculate the PSD from a signal x and window function w of the same length (real-valued numpy arrays) and the sampling frequency fs (float). You may use `np.fft.fft` to compute the DFT.

b) The Slepian sequences

The idea of multitapering is to apply mutually orthogonal windows (tapers) to the signal and average their PSDs. Typically, the Slepian sequences, also called discrete prolate spheroidal sequences (DPSS), are used.

- Use `scipy.signal.windows.dpss` to create the first three tapers of length $M = 500$ (use $NW = 1$).
- Plot all three tapers in one plot. Make sure to include a legend showing the labels of the different lines.
- Verify computationally that the tapers are indeed pairwise orthogonal to each other. Explain your computation in a short comment.

c) Multitaper spectrum

The multitaper spectrum is computed by applying the tapers to the signal and computing the PSD for each of these signals separately. Then, these different estimations are averaged. Implement a function `multitaper(x, fs, k)`, where x is the signal (real-valued numpy array of arbitrary length), fs is the sampling frequency in Hz and k is the number of tapers that should be used. Within the function, set the parameter NW of `scipy.signal.windows.dpss` to the length of the signal in seconds.
Hint: use the function `psd(x, w, fs)` from part (a) to compute the PSD of each windowed signal.

d) Testing the function

- Import the audio file `speech.wav` using `scipy.io.wavfile.read`.
- Print the signal's sampling frequency.
- Plot the signal in the time and frequency domain. You may use `np.fft.fft` to compute the DFT.
- Compute and plot the PSD and the multitaper spectrum using 20 tapers.
- Describe your observations in a comment.

2.4 The Short-Time Fourier Transform

a) STFT implementation

Implement a function `stft(x, fs, wlen, win_func=False)` computing the STFT of a given signal x (numpy array of arbitrary length), fs is the sampling frequency (float) and wlen is the window length in seconds (float). The function should compute the DFT (`numpy.fft.fft`) for each segment of length wlen separately (no overlap) and return a complex-valued array of the shape `(n_windows, n_frequencies)`. When the boolean parameter `win_func` is set to `True`, multiply each segment with the *Hann* window function.

Hint: Use scipy.signal.windows.hann to create the Hann window function.

b) Plotting the STFT

- Define the signal shown below with a sampling frequency of 100Hz.

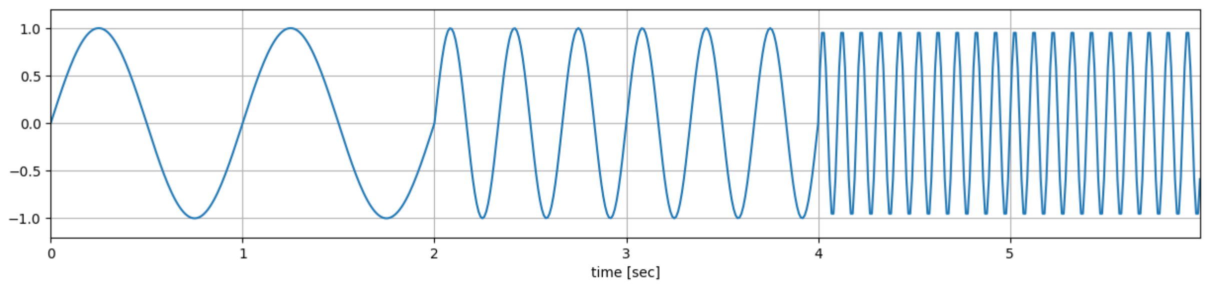


Figure 2: Test signal with different frequency components.

- Compute the signal's STFT using your previously defined function, using a window size of 1 second.
- Define a function `plot_stft(x_stft, fs, fmax=None, log=False, colormap_min=None, colormap_max=None)` that plots the amplitude of the STFT as an image. The function should:
  - Plot only the **positive frequencies**. If `fmax` is specified, limit the plot to frequencies up to `fmax`.
  - Plot the two amplitude spectra on a logarithmic scale (log-scaled, `numpy.log`) if `log=True`.

- Use `plt.imshow` to plot, with:
  * `extent` make sure the axis ticks are labeled correctly in seconds and Hz,
  * `vmin=colormap_min` and `vmax=colormap_max` optionally adjust the color scale,
  * and use `aspect='auto'` and `origin='lower'`.
- Add a **colorbar** to the plot.

- Plot the STFT of the above signal with `fmax=30` and default parameters. The result should look like this:

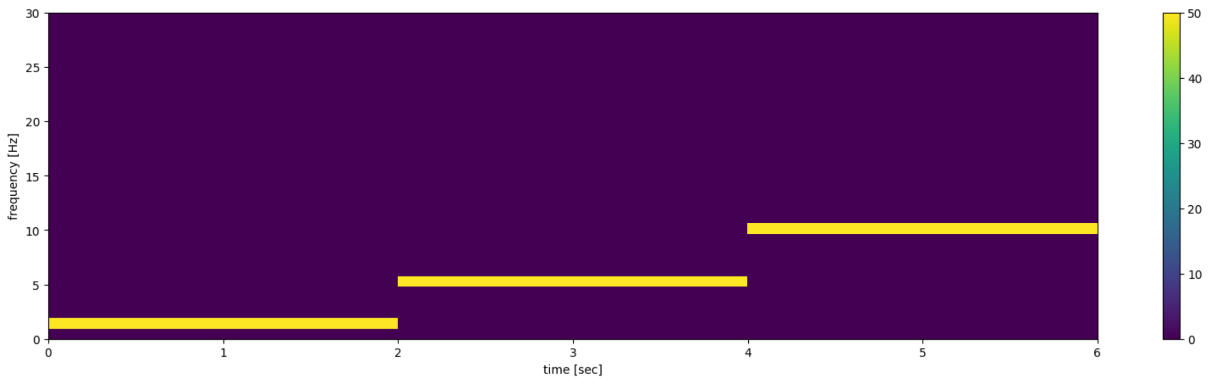


Figure 3: STFT of the test signal with different frequency components.

c) Real-world example

- For the signal `speech.wav`, compute the STFT with a window length of 0.5 seconds.
- Plot the STFT with a suitable `fmax`.
- There are two different speakers. One has a higher voice than the other. Can you tell them apart without listening to the file? If so, who is speaking when, and how do you know? Write your observations in a comment.

2.5 Discrete Cosine Transform

For the N-points one-dimensional discrete cosine transform (1D-DCT), its $k-$th basis vector $\overrightarrow{\mathbf{a}_k}$ is defined as

$$a_k[n] = c_k \cos\left[\frac{\pi}{2N} k\,(2n+1)\right] \in \mathbb{R}^{N\times 1}, n\&k \in [0, ..., N-1]\,, \tag{1}$$

where c_k is a constant coefficient and can be defined differently upon requirement, and these basis vectors can be visualized as below in Figure 4 (N=8, and $c_k \equiv 1, \forall k$).

For the 1D input $\mathbf{x} \in \mathbb{R}^{N\times 1}$, the k-th coefficient of its 1D-DCT (type II) X_k^{1D-DCT} can be simplified as

$$X_k^{1D-DCT} = \sum_{n=0}^{N-1} x[n] a_k[n] = \overrightarrow{\mathbf{x}}^T \overrightarrow{\mathbf{a}_k}, k = 0, ..., N-1, \tag{2}$$

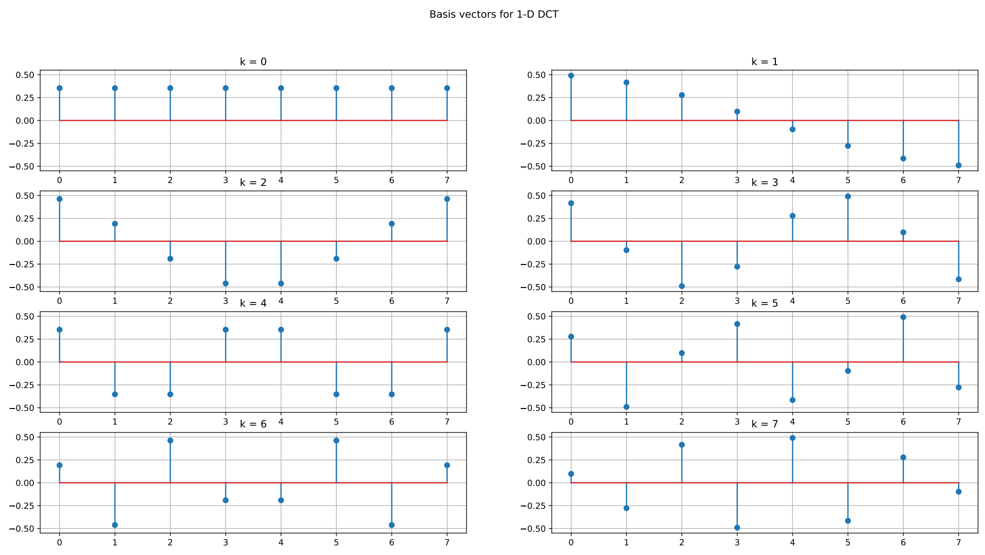


Figure 4: Basis vectors of DCT, each subplot represents one component of the basis vectors

where c_k can be defined differently depending on the requirement of orthonormality; hence, the 1D-DCT coefficients are the inner products between the input signal and each component of the basis vectors. These coefficients can be interpreted as the scalar projection of the 1D signal in the direction of different components of the basis vectors. Similarly, the 2D-DCT coefficients can also be interpreted as the scalar projection of the input image in the direction of different components of the *basis vectors* on the 2D space named basis image.

Note: A basis image should be composed of a set of sub-images, and each sub-image is namely the *component of the basis image.*

In the lecture, we have introduced how to apply 2D-DCT in image compression using the whole image as input, which is convenient while facing low computational efficiency. In practice, e.g., in the standard JPEG, a *block transform* strategy is adopted for 2D-DCT to balance the trade-off between computational efficiency and transformation quality. The *block transform* means slicing the large input image into several small image blocks first and applying 2D-DCT to each block, which you need to implement as below.

a) Basis vector

1. Write a function

```
a_k = basis_vector(N,k)
```

which takes N and k as input and returns the $\overrightarrow{\mathbf{a}_k}$, i.e., the k-th basis vector of N points DCT which is defined as below:

$$\overrightarrow{\mathbf{a}_k} = \left[\cdots, c_k \cos\left[\frac{\pi}{2N} k\,(2n+1)\right], \cdots\right]^\intercal \in \mathbb{R}^{N\times 1}, k\&n \in [0, \cdots, N-1] \qquad (3)$$

$$c_k = \begin{cases} \sqrt{\frac{1}{N}} & \text{, if } k = 0 \\ \sqrt{\frac{2}{N}} & \text{, otherwise,} \end{cases}$$

where c_k is the coefficient defined differently here from the lecture note for preserving the orthonormality, i.e.,

$$\overrightarrow{\mathbf{a}_k}^\intercal \overrightarrow{\mathbf{a}_l} = \begin{cases} 1 & , k = l \\ 0 & , k \neq l \end{cases}, \forall k, l \in [0, \cdots, N-1]$$

2. Plot the basis vectors for $N = 8$ to recreate the same plots as in Figure 4.

b) Basis image

1. Based on the DCT basis vectors obtained from `basis_vector(N,k)`, write a function

   `A_BI = basis_image(N)`

   which takes the N as input and returns the corresponding basis image with shape $N^2 \times N^2$.

$$\mathbf{A}_{BI} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \cdots & \cdots & \mathbf{A}_{0,N-1} \\ \mathbf{A}_{1,0} & \ddots & \cdots & \cdots & \mathbf{A}_{1,N-1} \\ \vdots & \vdots & \mathbf{A}_{k,l} & \cdots & \cdots \\ \vdots & \vdots & \vdots & \ddots & \cdots \\ \mathbf{A}_{N-1,0} & \mathbf{A}_{N-1,1} & \cdots & \cdots & \mathbf{A}_{N-1,N-1} \end{bmatrix} \in \mathbb{R}^{N^2 \times N^2}, \tag{4}$$

where $\mathbf{A}_{k,l} = \overrightarrow{\mathbf{a}_k} \overrightarrow{\mathbf{a}_l}^\intercal \in \mathbb{R}^{N \times N}, k\&l \in [0, 1, \cdots, N-1]$,

2. Plot the 2D-DCT basis image $\mathbf{A}_{BI}$ for $N = 8$.
   - Compare the four corners of the image and discuss what you can see.

c) DCT matrix

1. Based on the basis vectors of 2D-DCT, write a function

   `A_DCT = dct_mat(N)`

   which takes the number of DCT points N as input and returns the corresponding DCT matrix with shape $N \times N$ which defined as:

$$\mathbf{A}_{DCT} = \begin{bmatrix} \overrightarrow{\mathbf{a}_0}^\intercal \\ \vdots \\ \overrightarrow{\mathbf{a}_k}^\intercal \\ \vdots \\ \overrightarrow{\mathbf{a}_{N-1}}^\intercal \end{bmatrix} \in \mathbb{R}^{N \times N}, \tag{5}$$

   Hint: For calculating the DCT matrix, you only need the basis vectors.

2. Plot the DCT matrix $\mathbf{A}_{DCT}$ for $N = 8$.

d) Block transform

1. Write the blockwise 2D-DCT function

$$\mathbf{X}_{block}, \mathbf{Y}_{block}, \mathbf{Y} = \texttt{block\_dct2}(\mathbf{X}, N),$$

which takes the whole image $\mathbf{X} \in \mathbb{R}^{M\times M}$ as well as the number of DCT points (i.e., the size of each block) N as input and returns the sub-images $\mathbf{X}_{block}$, block transformed sub-images $\mathbf{Y}_{block}$ (both are with shape $\frac{M}{N} \times \frac{M}{N} \times N \times N$) as well as the 2D-DCT image $\mathbf{Y}$ which is "assembled" from $\mathbf{Y}_{block}$ as illustrated in Figure 5.

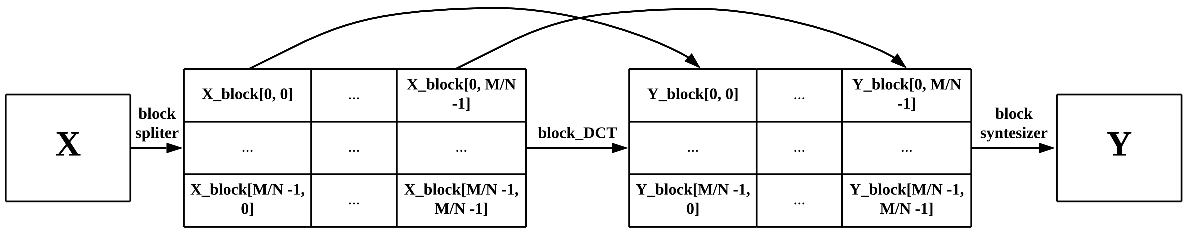


Figure 5: Pipeline of the block transform.

- Only consider the case where M is the multiple of N, i.e., $\frac{M}{N}$ is an integer larger than 1, and both input images and blocks are squared.

  Note: You are not allowed to use `scipy.fftpack.dct()`.
- You first need to split the image into blocks, i.e., sub-images, and then apply the DCT to each block.
- Use the DCT matrix computed from **Task 3 (c)**, then implement the function based on the given formula as shown below:

$$\mathbf{Y}_{block}[k, l] = \mathbf{A}_{DCT}\mathbf{X}_{block}[k, l]\mathbf{A}_{DCT}^T, \forall k\&l \in [0, 1, \cdots, \frac{M}{N} - 1] \tag{6}$$

2. Apply the blockwise 2D-DCT transform to an image.
   - Use the file *christoph512.jpg* as input image.
   - Plot the block-DCT image $\mathbf{Y}$ for $N = [2, 4, 8, 16]$.

e) Reordering

A block-based transform's resulting image is meaningless for a human observer. However, when reordering the pixels, all coefficients from the same basis function are grouped. By doing this, we can grasp the properties of the transform much better.

1. Write a reordering function

$$\mathbf{Y}_{\mathbf{reorder}} = \texttt{reorder}(\mathbf{Y}, N)$$

that groups together the pixels of the block-DCT image, i.e., $\mathbf{Y}$, as shown in Figure 6.

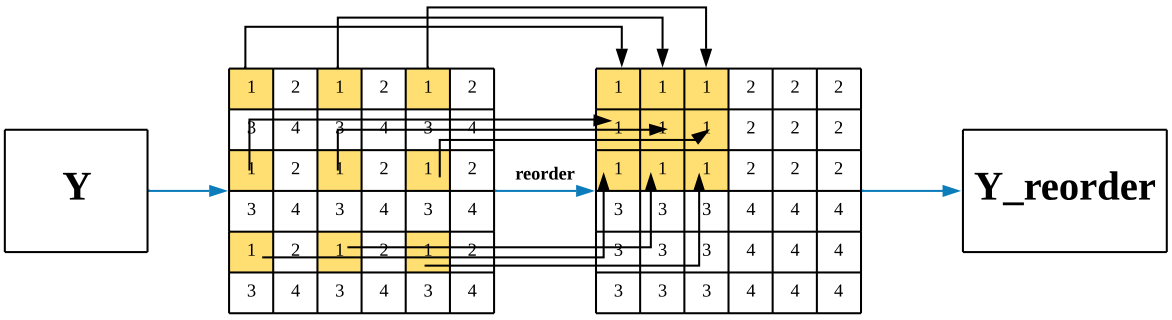


Figure 6: Reordering pixels of the block-DCT image for $M = 6$ and $N = 2$.

2. Plot the reordered block-DCT image $\mathbf{Y}_{\mathbf{reorder}}$ for $N = [2, 4, 8, 16]$.
   - The top-left sub-image of the reordered block-DCT image should be a smaller version of the original image. Why does this phenomenon appear?

2.6 Discrete Wavelet Transform

In the lecture, we introduced the discrete wavelet transform (DWT) from the perspective of multi-resolution analysis and compared it with another time-frequency analysis method, the short-time Fourier transform (STFT). Apart from that, DWT, in particular 2D-DWT, can also be derived from the perspective of a recursive dyadic decomposition of the low- and high-pass filter, i.e., sub-band decomposition. In this way, the blockwise DCT can be interpreted as a special case of DWT because this linear block transform, i.e., DCT, is equivalent to the sub-band decomposition when the number of bands is equal to the transform length N, and the decimation factor in the sub-band decomposition is also equal to N.

Nevertheless, as the technique applied in the standard JPEG, the blockwise DCT method possesses many advantages while also facing the unpleasant block artifact. To alleviate this problem, in the next generation of standard JPEG, i.e., JPEG 2000, DWT is adopted as the new compression method. Instead of replicating the original processing pipeline in JPEG2000, we choose the Haar transform, which is known as the simplest wavelet transform.

The smallest Haar matrix is defined as $\mathbf{A}_2^{Haar}$ and the corresponding Haar transform is similar to DCT, which is:

$$\mathbf{A}_2^{Haar} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{7}$$

$$\mathbf{Y} = \mathbf{A}_2^{Haar} \mathbf{X} \left(\mathbf{A}_2^{Haar}\right)^{\intercal} \tag{8}$$

Assuming input image is $\mathbf{X} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, apparently, the transformed image is

$$\mathbf{Y} = \tfrac{1}{2} \begin{bmatrix} a+b+c+d & a-b+c-d \\ a+b-c-d & a-b-c+d \end{bmatrix}$$

These operations correspond to:

1. $a+b+c+d$: 4-point average or 2-D lowpass (Lo-Lo) filter
2. $a-b+c-d$: average horizontal gradient or horizontal highpass and vertical lowpass (Hi-Lo) filter
3. $a+b-c-d$: average vertical gradient or horizontal lowpass and vertical highpass (Lo-Hi) filter
4. $a-b-c+d$: diagonal curvature or 2-D highpass (Hi-Hi) filter

a) Haar matrix

A general definition of the Haar matrix is shown below:

$$\mathbf{A}_{2^{n+1}}^{Haar} = 2^{-\frac{n}{2}} \begin{bmatrix} \mathbf{A}_{2^n}^{Haar} \otimes [1 \quad 1] \\ 2^{\frac{n}{2}} \mathbf{I}_{2^n} \otimes [1 \quad -1] \end{bmatrix} \in Re^{2^{n+1} \times 2^{n+1}}, n \geq 1 \text{ and } n \in \mathcal{Z}, \tag{9}$$

where $\otimes$ represents the Kronecker product (`https://en.wikipedia.org/wiki/Kronecker_product`).

1. Write a function

   `A_H = haar_mat(`n`)`

   which takes n as input and returns the Haar matrix with size $2^{n+1} \times 2^{n+1}$.

   Note: You are allowed to use `numpy.kron()`.
2. Plot the Haar matrix A_{Haar} for $n = [1, 2, 3, 4]$].

b) Haar transform with linear structure

1. Now, write a function

   $\mathbf{Y}$ `= haar_trans(`$\mathbf{X}, N$`)`

   which takes a squared image $\mathbf{X} \in Re^{M \times M}$ and the block size N as input and return the Haar-transformed image $\mathbf{Y} \in Re^{M \times M}$ using the corresponding Haar matrix $\mathbf{A}_N^{Haar}$ based on equation (8).

   - Only consider the case where M is multiple of N and N is to the power of 2, i.e., $N = 2, 4, 8, \ldots$.
   - This function should contain three major parts: splitting the image into blocks, applying the Haar transform to each block, and reordering the transformed image, as illustrated in Figures 5 and 6.

   Hint: You can reuse functions from **Task 3**.
2. Plot the Haar-transformed images.
   - Use the image file *christoph512.jpg* as input.
   - Plot the Haar-transformed image for $N = [2, 4, 8, 16]$.
   - Compare the Haar-transformed images with the DCT-transformed images with the same block size and describe the differences you can find.

c) Haar transform with pyramidal structure

In the previous question, you implemented the Haar transform like a basic linear transform. You should be able to notice when using Haar transform with $N = 2$, the transformed image can be divided into four regions, Lo-Lo, Lo-Hi, Hi-Lo, and Hi-Hi, i.e., each region contains a different frequency component along either row or column, i.e., after the row-/column- filtering with either low- or high-pass filter.

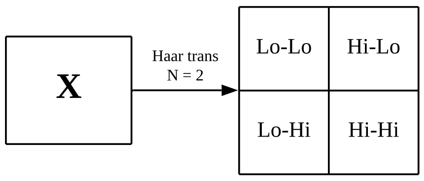


Figure 7: Illustration of block Haar transform ($N = 2$).

The 2D-DWT can also be implemented differently, where the transformed image is derived in a pyramid or tree structure, as shown below.

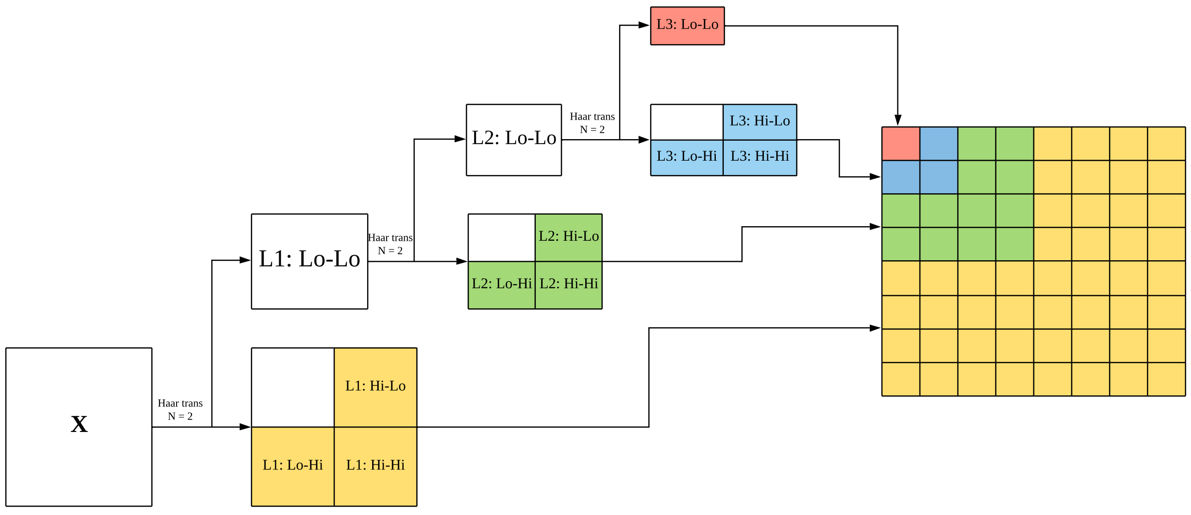


Figure 8: Illustration of the pyramid structured Haar transform (Decomposition level = 3).

1. Based on the idea in Figure 8, write a function

$$\mathbf{Y} = \texttt{pyramid\_haar}(\mathbf{X}, level)$$

   which takes a squared image $\mathbf{X} \in Re^{M\times M}$ and the decomposition level *level* as input and returns the pyramidal-structured Haar-transformed image $\mathbf{Y} \in Re^{M\times M}$.

   - Apply the linear Haar transform with block size $N = 2$ at each level.

2. Plot the pyramidal-structured Haar-transformed images.

   - Use the image file *christoph512.jpg* as input.

- Plot the images for the levels $level = [1, 2, 3, 4]$.
- Compare them to the linearly Haar-transformed images. How are the levels (at block size $N = 2$) related to the block sizes in the linear Haar transform?

3 Filter Design & Resampling

3.1 Digital filter design and resampling

In the first task, you will resample an audio signal. As shown in the lecture, when resampling (down- and/or upsampling) a signal, you also need to apply a low-pass filter. Therefore, We will first guide you through the steps of designing and applying a finite impulse response (FIR) filter via windowing:

1. Analytically specify the desired frequency response
2. Apply the inverse Fourier Transform to obtain the impulse response
3. Use a window function to obtain a finite-length filter

After that, you will apply the filter to an audio signal to perform a proper resampling.

a) Design an ideal zero-phase FIR low-pass filter

1. Write a function

   ```
   H_0, h_0 = ideal_lowpass(fc, fs, M)
   ```

   which takes the cut-off frequency `fc` in Hz, the sampling frequency `fs` in Hz, and the length of the filter `M` in samples as input parameters. The output `H_0` is the amplitude spectrum of the ideal low-pass filter in the frequency domain, and the output `h_0` is the impulse response in the time domain.

   - Specify the amplitude spectrum of your filter such that

     $$H_0(f) = \begin{cases} 1, |f| \leq f_c \\ 0, \text{otherwise} \end{cases}$$

     Hints:

     – Consider the frequency resolution of your filter: $f_{\text{res}} = \frac{f_s}{M}$.

     – Since you are designing a zero-phase filter, you don't have to specify a phase.

   - Apply the inverse Fourier Transform to $H_0(f)$ to obtain the impulse response $h_0[n]$.

     Hint: Use the functions `fftshift` and `ifft` from the library `numpy.fft`.

2. Design a *Hann* window, by writing another function `hann_window(M)` which takes the length of the window `M` in samples as an input parameter and returns the window `w_hann` in the time domain. The *Hann* window is defined as follows:

   $$w_{\text{hann}}[n] = \begin{cases} 0.5 - 0.5\cos(\frac{2\pi \cdot n}{M}), 0 \leq n \leq M-1 \\ 0, \text{otherwise} \end{cases}$$

3. Write a function to obtain an FIR filter

   ```
   H_fir, h_fir = fir_lowpass(fc, fs, M)
   ```

which takes the cut-off frequency `fc` in Hz, the sampling frequency `fs` in Hz, and the length of the filter `M` in samples as input parameters. The output `H_fir` is the amplitude spectrum of the FIR low-pass filter in the frequency domain, and the output `h_fir` is its impulse response in the time domain.

- Call your functions `ideal_lowpass` and `hann_window`.
- Multiply your window function $w_{\text{hann}}[n]$ with the impulse response of your filter $h_0[n]$ to obtain a finite-length (FIR) filter:

$$h_{\text{fir}}[n] = h_0[n] \cdot w_{\text{hann}}$$

- Apply the Fourier Transform to $h_{\text{fir}}[n]$ to obtain the amplitude spectrum $H_{\text{fir}}(\omega)$.
  Hint: Use the functions `fftshift` and `fft` from the library `numpy.fft`.

4. Define a filter of length $M = 501$, a cut-off frequency $f_c = 30$ Hz, and a sampling frequency $f_s = 1000$ Hz.
   - Plot your filter in the time and the frequency domain **before and after** you apply your window function.
   - Which differences can you find between before and after applying your window function (in the time and frequency domain)?

b) Test your filter on a signal with multiple frequencies

1. Define a signal

$$x[n] = \sum_{k \in \{5,20,50,100\}} \sin(2\pi \frac{k}{f_s} n)$$

   with length $N = 2000$ ($n \in [0, N-1]$) and sampling frequency $f_s = 1000$ Hz. Plot the signal in the time and frequency domain.

2. Apply your finite-length filter $h_{\text{fir}}[n]$ to your signal $x[n]$ by using convolution:

$$x_{\text{filt}}[n] = x[n] * h_{\text{fir}}[n]$$

   Hint: Use the function `numpy.convolve` with the parameter `mode='same'` .
   - Plot the filtered signal $x_{\text{filt}}[n]$ in the time and frequency domain.
   - Does your filter work properly? Discuss your results!

c) Downsample the audio signal

1. Inspection of the audio signal:
   - Use `scipy.io.wavfile.read` to load the audio file *beethoven.wav.*
   - Print the sampling rate and the audio signal's length (in samples and seconds).
   - The audio file contains a stereo signal; convert it to a mono signal.

- Use `sounddevice.play()` to play the audio signal.
- Plot the signal in the time and frequency domain.

2. Naively downsample the audio signal by the factor $c_{\text{down}} \in \{5, 10, 20\}$.
   - Take every c_{down}^{th} sample of your audio signal.
     Example: $c_{\text{down}} = 2$, $x[n] = [1, 2, 3, 4, 5, 6] \rightarrow x_{\text{down}}[n] = [1, 3, 5]$
   - Update the sampling frequency $f_{s_{\text{down}}} = \dfrac{f_s}{c_{\text{down}}}$
   - Print the sampling rate and length (in samples and seconds) of the naively downsampled signals.
   - Play the naively downsampled audio signals and compare them to the original audio signal.
     Hint: Divide your downsampled signal by a factor of $clip = \frac{f_s}{2}$ to prevent your signal from clipping.
3. Properly downsample your audio signal by the factor $c_{\text{down}} \in \{5, 10, 20\}$ by applying an FIR filter
   - Use the same filter length as in (b) and the sampling frequency of your audio signal.
   - Calculate the cut-off frequency $f_c = \dfrac{\frac{f_s}{2}}{c_{\text{down}}}$
   - Apply the FIR filter to your audio signal and then take every c_{down}^{th} sample of your audio signal.
   - Play the properly downsampled audio signals and compare them to the naively downsampled signals and the original audio signal.
   - Print the cut-off frequency and the length (in samples and seconds) of the properly downsampled signals.
4. Compare the amplitude spectrum of the original audio signal with the spectra of the naively and the properly downsampled signals for all three downsampling factors c_{down}.
   - Plot all amplitude spectra in a range of $[-\frac{f_{s_{\text{down}}}}{2}, \frac{f_{s_{\text{down}}}}{2}]$, with respect to the corresponding downsampling factor c_{down}.
   - What differences do you see between the original spectrum, the naively downsampled spectrum, and the properly downsampled spectrum and
   - What differences do you see between the different downsampling factors c_{down}?

d) Reconstruct the audio signal

1. Naively upsample the properly downsampled (FIR filtered) audio signals by the factor $c_{\text{up}} = [5, 10, 20]$ by using linear interpolation. Match the downsampled signals with the corresponding upsampling factors c_{up}, so that $f_{s_{\text{up}}} = f_s$.

- Repeat every sample of your signal c_{up} times.
  Example: $c_{\text{up}} = 2$, $x[n] = [1, 3] \rightarrow x_{\text{up}}[n] = [1, 1, 3, 3]$
  Hint: You are allowed to use the function `numpy.repeat`.
- Update the sampling frequency:

$$f_{s_{\text{up}}} = f_{s_{\text{down}} \cdot c_{\text{up}}} \tag{10}$$

- Play the naively upsampled audio signals and compare them to the original audio signal.
  Hint: Divide your upsampled signal by a factor of $clip = \frac{f_s}{2}$ to prevent your signal from clipping.
- Print the sampling rate and the length (in samples and seconds) of the naively upsampled signals.

2. Properly upsample your properly downsampled (FIR filtered) audio signals by the factor $c_{\text{up}} = [5, 10, 20]$ by using the *Spline Interpolation* algorithm. For a detailed explanation, please look at `https://en.wikipedia.org/wiki/Spline_interpolation`. Match the downsampled signals with the corresponding upsampling factors c_{up}, so that $f_{s_{\text{up}}} = f_s$.

   - Use the function `interp1d` from the library `scipy.interpolate` with the parameter `kind='cubic'`.
   - Play the properly upsampled audio signals and compare them to the original audio signal and the naively upsampled signals.
     Hint: Divide your upsampled signal by a factor of $clip = \frac{f_s}{2}$ to prevent your signal from clipping.
   - Print the sampling rate and the length (in samples and seconds) of the properly upsampled signal.

3. Compare the amplitude spectrum of the original audio signal with the spectra of the naively and the properly upsampled signals for all three upsampling factors c_{down}.

   - Plot all amplitude spectra in a range of $[-\frac{f_{s_{\text{up}}}}{2}, \frac{f_{s_{\text{up}}}}{2}]$, with respect to the corresponding upsampling factor c_{up}.
   - What differences do you see between the original spectrum, the naively upsampled spectrum, and the properly upsampled spectrum?
   - What differences do you see between the different upsampling factors c_{up}?