ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institute for Dynamic Systems and Control
IDSC
Institut für Dynamische Systeme und Regelungstechnik

151-0575-01 **Signals and Systems** (Fall 2022)

Programming Exercise

Issued: November 25, 2022 Due: Jan 25, 2023, 23:59 (CET)

Overview

- This programming exercise contains three parts, each consisting of programming questions as well as written-only questions. Please write all your solutions in the provided Jupyter Notebook files.
- The submissions will be evaluated by restarting the notebook kernel and running all cells successively. Please keep this in mind and make sure that the notebook you submit is runnable in this way.
- Your submission should consist of a .zip file containing the same, but accordingly edited notebooks as in the hand-out. Please submit your solutions using the "Programming Exercise Submission" option under "Programming Exercise" tab on Moodle.
- Up to three students may work together on the programming exercise and submit a group solution. All members of the group will receive the same grade. At least one member of the group must submit the .zip file and name it "ssprogex_<identity numbers separated by underscores>.zip". Make sure that you only submit the code that we ask for and remove any debugging functionalities.
- The programming exercise is optional and acts as a bonus towards your final grade. The bonus is either 0.0, 0.125, or 0.25-grade points, depending on your performance in the exercise, i.e., if you receive a bonus of 0.25-grade points, this will be added to the grade of your final exam, e.g. $5.25 + 0.25 = 5.5$.

Best Practices

We strongly suggest using Jupyter Hub available on the course Moodle page as the environment for solving the programming exercise. Furthermore, since most of the Python native functions have multiple options, we provide some specifications on how you should use these functions. The required packages have already been provided in the template notebook files. The tasks must be solved without additional packages.

- `scipy.fftpack.dct(x, norm='ortho')`
- `scipy.fftpack.idct(x, norm='ortho')`
- `numpy.reshape(a, newshape, order='F')`

Exercise 1: Filtering of an Audio Signal (20 points)

Introduction

In this exercise, we consider a specific case of signal discretization, from the audio signal perspective. In Figure 1, we show the steps of a process that represents recording a continuous-time audio signal $x(t)$ using a microphone. The original signal $x(t)$, which represents a waveform of a singer's voice in time is processed by three different blocks and affected by noise $w[n]$ in the process. In the remainder of this section, we take a look at each stage of this pipeline separately.

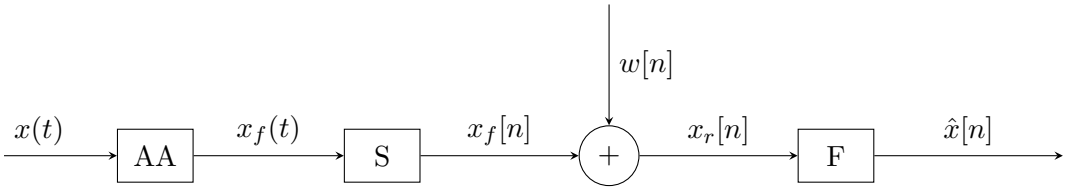


Figure 1: Schema of a recording process for a continuous-time audio signal $x(t)$

Anti-aliasing Filter "AA": An anti-aliasing filter is applied to ensure that the discretization process does not result in aliasing. Remember that, when sampling a continuous-time signal, it is important to ensure it has minimal frequency content (ideally none) at or above the Nyquist frequency, as otherwise the content above this frequency gets mapped (and hence aliased) into the spectrum range below it. This results in corruption of the discrete-time signal, making reconstruction of the original continuous-time signal impossible. To avoid this, an anti-aliasing filter can be performed before sampling to ensure the frequencies above the Nyquist one in the filtered signal $x_f(t)$ are attenuated and aliasing is hence prevented. One way to design such filter is to consider an analogue Butterworth low-pass filter with an appropriate cut-off frequency and this will be your task in the first part of this exercise.

Sampler "S": Once the spectrum of the original audio signal is appropriately limited in the frequencies it contains, it is subjected to sampling. The uniform sampler "S" samples $x_f(t)$ for $t = nT_s$, where n is an integer and T_s is the sampling time, resulting in a discrete-time signal $x_f[n]$.

The Effect of Noise: Depending on the acquisition process of the audio recording (e.g. the quality of the microphone, the amount of background noise, etc.), the resulting digital audio signal could be corrupted in different ways. The obtained signal can be viewed as a superposition of the discretized original analogue source sound wave, and the unwanted component that we refer to as "noise". The latter can take different forms in terms of its frequency content: in some cases, it spans across all frequencies with a different amplitude profile (different options are shown in Figure 2), while in others it only affects single frequencies. The latter is often the case for the signal spectrum at 50 Hz, due to the frequency of the electric network and is therefore often referred to as "electric hum". Knowing the cause of noise and hence also its characteristics can help immensely in mitigating the deteriorating effect it has on the original signal. In this exercise, the sampler is powered by a faulty power-grid socket, and, as a result,

the sampled signal $x_f[n]$ is disturbed by electric hum $w[n]$. However, the provided signal was recorded on a remote island that operates on its own power grid with an unknown frequency and hence also the frequency of the ensuing electric hum in $x_r[n]$ is unknown.

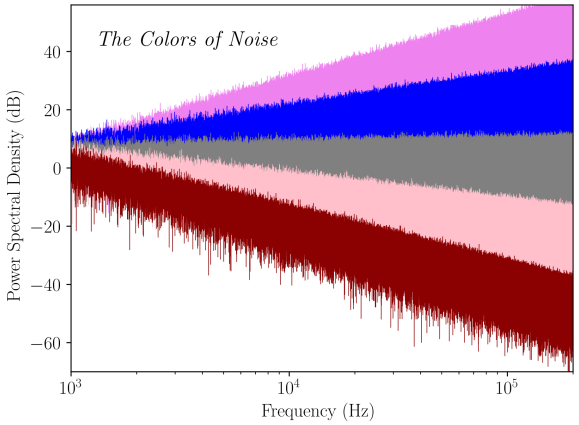


Figure 2: Simulated power spectral densities as a function of frequency for various colors of noise (violet, blue, white - denoted in gray, pink, brown/red). Source: Colors of noise

Filter "F": In order to mitigate the effect of noise on the recording, a filter is applied to attenuate the frequency corresponding to the electric hum. One way to do this is to completely remove the signal content at this frequency by setting the corresponding spectral coefficients to zero. To achieve this, one needs to first know the frequency of the noise and this will be your task in the second part of this exercise. Once you identify it, you will remove the spurious frequency content in the part three, where you will look into the design of the filter "F". Finally, in part four, you will compare filters "AA" and "F".

Provided Files

- `exercise1_task.ipynb` The main script for this part. Always run your code sections from here.

Tasks

1. **(5 points)** In this task, you should design the "AA" filter, so you avoid aliasing when sampling the input with a frequency $F_s = 44.1$kHz. You can use `scipy.signal.butter` function with the appropriate arguments, but ensure you can preserve as much of the signal content as possible. Provide the filter coefficients and the plot of the filter transfer function.

2. **(5 points)** In this task, you should identify the frequency of the electric hum in the recorded signal $x_r[n]$. To obtain this signal, use your student number (in case there are more people in your group, you can use any of your student numbers) as an input to the function `generate_recorded_signal()`. Then, compute the frequency spectrum of the noisy signal $x_r[n]$ and plot it. Based on the plot, identify the frequency of the noise. You

can assume that the noise is only present at a single frequency and that its amplitude is smaller than the amplitude of the original signal.

3. **(5 points)** In this task, you should perform the filtering in the block "F". To achieve this, set the spectral coefficient (in the frequency domain) corresponding to the above-identified noise frequency to zero. Once you do this, plot the spectrum and time-domain representation of the filtered signal. To check if you correctly cancelled the noise, also plot your results against the original noisy function (on the same plot). You can use the helper function `save_audio()` to save the audio file corresponding to the filtered signal.

4. **(5 points)** Compare the filters "AA" and "F". In particular, contrast them in terms of causality and what you can say regarding their benefits and drawbacks.

Exercise 2: Audio Compression (40 points)

Introduction

The MP3 audio coding format is one of the most popular audio formats used today. It is mainly used to compress audio files in order to allow efficient storage and sending. The compression method used by MP3 is lossy, i.e., some parts of the original audio signal are removed and cannot be restored. The main idea behind this compression method is to remove frequencies, that are not audible, e.g. frequencies that are outside the frequency band detectable by our ears or frequencies that are masked by other louder frequencies.

At the heart of the MP3 compression method lies the discrete cosine transform (DCT), which is a special case of the discrete Fourier transform (DFT). There are four types of DCT, but the second type (DCT-II) is commonly referred to as "the" DCT, and is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right].$$

The MP3 compression method does not directly use the DCT-II, but rather a modified version of it, which allows overlapping signal windows. For this exercise we will restrict ourselves to the DCT-II and use it as a rudimentary approximation of the MP3 compression method.

Provided Files

We provide a zip-file containing the following files:

- `exercise2_task.ipynb` The main script for this part. Always run your code sections from here.
- `audio_read.ipynb` This function reads an audio file into a multidimensional array depending on the number of channels. It also returns the sampling rate.
- `audio_save.ipynb` This function writes a multidimensional array to an audio file at a given sampling rate.
- `compress.ipynb` Template for the compression function.
- `compress_with_windowing.ipynb` Template for the modified compression function, which handles windowing.
- `decompress.ipynb` Template for the decompression function.
- `decompress_with_windowing.ipynb` Template for the modified decompression function, which handles windowing.
- `load_from_disk.ipynb` This function loads a json file in frequency domain.

- `mydct.ipynb` Template function for your implementation of the DCT.
- `save_to_disk.ipynb` This function saves your compressed audio signal in the frequency domain. The function returns the file size and produces a .json file. The output `filesize` is expressed in Bytes.
- `visualize_time_frequency.p` This function visualizes the energy of a 1D time domain signal in the time-frequency domain.
- `Pete_Murphy_Aside.flac` Uncompressed audio file. Note that FLAC uses a non-lossy compression method, hence the file size is smaller than the uncompressed audio file size of 3.9 MB.

Tasks

1. **(10 points)** In the first task, we want to derive the discrete cosine transform (DCT) - in particular the second version, DCT-II - from the discrete Fourier transform (DFT).

   As seen in the lecture, the discrete Fourirer transform (DFT) is given by:

   $$X[k] = \sum_{n=0}^{N-1} x[n] \, e^{-jk\frac{2\pi \cdot n}{N}} \tag{1}$$

   The second discrete cosine transform (DCT-II) is defined as:

   $$X[k] = \sum_{n=0}^{N-1} x[n] \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right]. \tag{2}$$

   (a) **(8 points)** Formally derive the expression for the DCT-II from the DFT. State all the assumptions that you make.

   (b) **(2 points)** Describe in a few sentences how the DCT can be computed using the DFT, i.e., provide some insight to the formal derivations in (a).

   Note: For more information you can refer to `https://en.wikipedia.org/wiki/MP3`

2. **(6 points)** Based on the results from Task 1, implement your own DCT function. Use the native Python function `fft` and follow the steps you detailed in Task 1, to compute the DCT.

   Note: Naively implementing the DCT function, i.e., using two for-loops, will result in an execution time of several hours for the audio file provided. Use an even extension of the original signal in order to extend the signal length from N to $2N$.

   (a) **(5 points)** Implement your own DCT, in `mydct`. Using the native Python function `dct` in your code will result in zero points.

   (b) **(1 point)** Comparing the results of your implementation with the ones of the native Python `dct`, you will notice differences in the frequency coefficients obtained. Determine the `normalization_constant` used by Python and store it in `normalization_constant`. Make sure your final `mydct` function outputs normalized coefficients.

3. **(24 points)** You are provided with an uncompressed audio file *Peter_Murphy_Aside.flac*, containing two audio channels - left and right. Your task is to compress the audio file in a MP3-like manner with the goal of reaching a high compression rate, while maintaining a high audio quality.

   (a) **(3 points)** Implement the `compress` function which computes the DCT and compresses the signal by removing unnecessary frequencies. Use `n_comp` to specify the

number of frequencies to keep. What happens if you vary `n_comp`? Plot the frequencies that you keep after the compression.

Note: You may use the native Python function `dct` function.

(b) **(3 points)** Implement the `decompress` function that transforms the compressed frequency domain signal back to the time domain. Despite not being able to recover the discarded frequencies, the time domain signal must have the same length `N` as the original one. You may use the Python native `idct` function for the inverse transform.

Note: For comparison, you can play your compressed and uncompressed audio signals using a media player (e.g. VLC media player).

(c) **(5 points)** In order to account for time varying frequency intensities as seen in the previous figures, one can apply a window function. *Windowing* involves multiplying the timesteps of a signal by a window function of finite length whose amplitude varies smoothly and gradually towards zero at the edges. This allows to keep certain frequencies, while others are discarded. Your task is to modify the original `compress` function to allow windowing of the audio signal. Start by defining a suitable window size `win_size`. The number of implemented windows `num_win` is determined such that at each timestep, only one of the windows is applied. Choose the number of components to keep `n_comp` to be 10% of `win_size`. For every window in `num_win`, you need to determine which `n_comp` of the original signal you want to keep. What is a suitable criteria to determine which frequency components to keep? Implement your solution in the `compress_with_windowing` function.

The output of `compress_with_windowing` includes:

i. `X_lc` and `X_rc`: the frequency channels that you keep after the compression of size `n_comp` x `num_win` + 1

ii. `i_lc` and `i_rc`: the corresponding indices of the original frequencies

Note: Make sure you take care of the last window, meaning the last part of the audio signal which doesn't fill a whole window. Any `win_size` that yields a suitable compressed signal can be used for this task. You may use the native Python `dct` function.

Hint: Use a rectangular window function.

(d) **(6 points)** Following your implementation in (c), we now want to decompress the signal that was previously compressed with windowing. We can use the indices of the original frequencies along with the frequency channels of the compressed signal to reconstruct the original signal. Implement your solution in the `decompress_with_windowing` function. Your output includes the reconstructed (decompressed) signal in the time domain.

Note: You may use the native Python `idct` function. However you should apply it to the array for each individual window, and not to the entire matrix containing all of the windows directly.

(e) **(4 points)** To determine the optimal window size `opt_win_size`, we try to minimize the error on our reconstructed audio signal in the time domain, i.e. `y_lc_win` and `y_rc_win`. Compare the reconstructed signal with the original signal, `x_lc` and `x_rc` using the root mean square. Do this for several window sizes and plot the error. As in the previous tasks, set `n_comp` to 10% of the window size for all your tries. What do you observe when you increase the window size? Can you find a global optimum?

Write an optimizer loop that searches for the optimal window size. Limit your search to window sizes smaller than 70000. A grid with 50 tries should be sufficient for your solution. Indicate the optimal window size in your plot and state the value in the title of the plot.

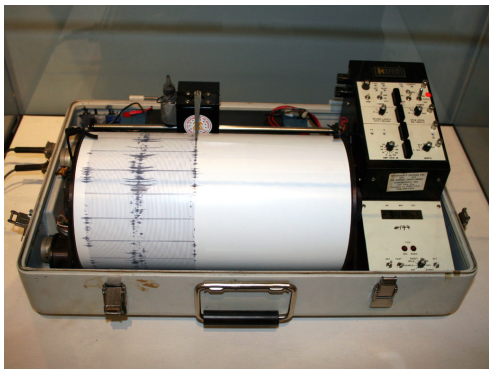
(f) **(3 points)** Having an optimal window size, try to compress the signal to achieve a small file size while maintaining good audio quality. Use `save_to_disk` to save your frequency domain signal to a json file. Use the root mean square error to assess the audio quality. In the end, you must submit one compressed *audio.json*.

Hint: You may take the mean of the errors of both channels as the ground truth error.

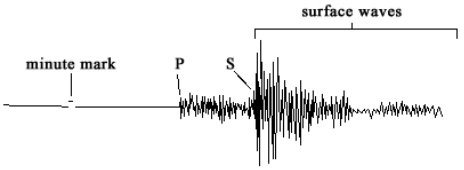
Exercise 3: System Identification (40 points)

Introduction

A seismograph consists of a seismometer and a recording device. The seismometer measures ground noises and shaking (caused by earthquakes, volcanic eruptions or even large music festivals). The recorded history of shocks and vibrations is called a seismogram, which is the output of a seismograph.



(a) Analog seismograph. Source: Wikipedia



(b) Seismogram. Source: Wiki Educator

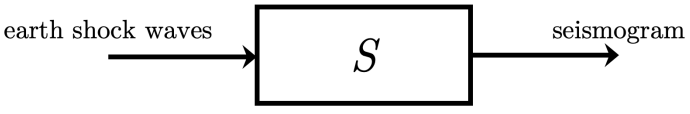
Provided Files

- `exercise3_task` The main script for this part. Always run your code sections from here.
- `seismograph.py` contains the functions `seismograph(u)`, `noisy_seismograph(u)`, `stabilized_seismograph(u, k_1, k_2)` and `noisy_stabilized_seismograph(u, k_1, k_2)` used in part 1 and 2 of the exercise.

  Use for example `from seismograph import noisy_seismograph` to be able to use a function in your notebook.
- `measurements.py` contains the functions `measurement_1`, `measurement_2` and `measurement_3` for the bonus exercise.

Part 1: Open-Loop System Identification (20 points)

The seismograph dynamics can be captured by a transfer function S.



Ideally, we want S to be a constant, i. e. $S = s$ where $s \in \mathbb{R}$. This way we can simply scale up the measurements to obtain the magnitude of an earthquake. Unfortunately, transfer function S cannot be modeled as a constant due to many non-idealities, such as the quality

of the seismograph, the rigidity of its mounting, the foundation of the surrounding building, the dynamics of the recording device, and numerous other factors. Therefore, we model the seismograph dynamics as a second-order system, i.e.,

$$S(s) = \frac{\omega^2}{s^2 + 2\delta\omega s + \omega^2} ,$$

where $\omega, \delta \in \mathbb{R}$ are unknown.

The system is discretized with some discretization time T_s, resulting in the following transfer function

$$S_{\text{DT}}(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} ,$$

where the parameters $b_0, b_1, b_2, a_1, a_2 \in \mathbb{R}$ are unknown.

Tasks

1. **(4 points)** Briefly explain how you would identify the unknown parameters of the discretized system using the two methods taught in the lectures.

2. **(12 points)** Implement the chosen method and identify the parameters b_0, b_1, b_2 and a_1, a_2 for measurements both with and without additive white noise.

   - Use `seismograph(u)` to simulate a seismograph.
   - Use `noisy_seismograph(u)` to simulate a seismograph affected by measurement noise.

   Suggestion: Write functions that are reusable for signals of various length and systems of different order.

3. **(3 points)** Compare your results. Is your method feasible to identify both the system with and without noise? Make a qualitative statement using the root mean square error (RMSE) between the *identified* frequency responses of the systems with and without noise. Compute the RMSE separately for the magnitude and phase of the transfer function.

   Hint: The RMSE for two scalar data sets $\{x_1, x_2, ..., x_N\}, \{\tilde{x}_1, \tilde{x}_2, ..., \tilde{x}_N\}$ is defined as

   $$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \tilde{x}_i)^2}{N}} .$$

4. **(1 point)** Assuming a discretization frequency of 1000Hz, approximate the continuous-time parameters ω and δ for the system without noise using the discrete-time parameters you just computed.

   Hint: Use the bi-linear transform. You can check your results using Python's `control.damp` function.

Part 2: Active Stabilization (20 points)

The fact that the previously identified seismograph has a second-order transfer function is not ideal for the use case of measuring the earth's vibration. In fact, second-order transfer function can have a distinct resonance frequency. Signals that excite the resonance frequency will result in a somewhat unexpected seismogram.

As the owner of a second-order seismograph, you decide to buy an active stabilization device that promises to minimize any unwanted resonance and damping effects. The way your seismograph S interacts with the stabilization device C is shown in 4.

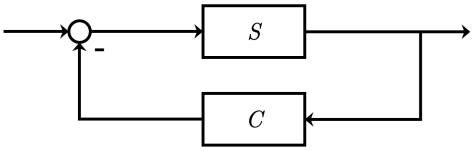


Figure 4: Block diagram of the actively stabilized seismograph.

Unfortunately, the device was developed for a different seismograph and therefore doesn't work straight out-of-the-box. To make matters even worse, the stabilizer's software is proprietary and the manufacturer has done a poor job on documentation. The only thing you know is, that the stabilizer has two "magic" tuning variables k_1 and k_2 with $k_1, k_2 \in [0, 1]$ that you can control.

Tasks

1. **(2 points)** Derive the transfer function of the actively stabilized (closed-loop) seismograph T as a function of S and C. Of what order is T assuming both S and C are second-order systems?

2. **(1 point)** Using your previous result, express C as a function of S and T.

3. **(5 points)** For $k_1 = 0.5$ and $k_2 = 0.2$, identify the unknown parameters of the closed-loop system T.

   - Use `stabilized_seismograph(u, k_1, k_2)` to simulate the closed loop system (without noise).
   - Approximate T by the fourth order system

   $$T(z) = \frac{b_{T,0} + b_{T,1}z^{-1} + b_{T,2}z^{-2} + b_{T,3}z^{-3} + b_{T,4}z^{-4}}{1 + a_{T,1}z^{-1} + a_{T,2}z^{-2} + a_{T,3}z^{-3} + a_{T,4}z^{-4}} \,.$$

4. **(3 points)** Use your results from the previous exercises (where you identified S and T without noise) to approximate the parameters of C (for $k_1 = 0.5$, $k_2 = 0.2$), where

   $$C(z) = \frac{b_{C,0} + b_{C,1}z^{-1} + b_{C,2}z^{-2}}{1 + a_{C,1}z^{-1} + a_{C,2}z^{-2}} \,.$$

5. **(6 points)** Now approximate the parameters of C for the system with noise.

   - Use `noisy_stabilized_seismograph(u, k_1, k_2)` to simulate the closed-loop system with noise.

- For S use the transfer function you identified for the seismograph *with* noise.
- The manufacturer claims C to be of second-order. Is the second-order assumption still valid for C? If not, what is the reason for that?

Hint: if you didn't solve the previous exercise, use $a_{S,1} = -1.7$, $a_{S,2} = 0.76$, $b_{S,0} = 0.015$, $b_{S,1} = 0.03$, $b_{S,2} = 0.015$ for task 4 and 5.

6. **(3 points)** For a fixed $k_1 = 0.8$, find k_2^* such that

$$k_2^* = \arg\min_{k_2} \|\Theta\|_1 \,,$$

where $\Theta = [b_{T,1}, b_{T,2}, b_{T,3}, b_{T,4}, a_{T,1}, a_{T,2}, a_{T,3}, a_{T,4}]^T$ and $\|v\|_1 = \sum_{i=1}^{N} |v_i|$.

- Use `noisy_stabilized_seismograph(u, k_1, k_2)` to simulate the closed-loop system T.
- Use a bisection algorithm (recall that $k_2 \in [0, 1]$). For each iteration of the algorithm, run system identification on T until your error in k_2^* is less than $\varepsilon = 0.05$.
  Note: When using a *different* algorithm you might not get partial points if the result is not correct. If the result and implementation is correct, you will also get full points when using a different algorithm.

Bonus: Earthquake Epicenter Triangulation (5 points)

Assume there are three identical seismographs

$$S_1(z) = S_2(z) = S_3(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}} \,,$$

with the parameters that were identified in exercise (no active stabilization) located at (1) the Swiss Seismological Service[1], (2) the Institute de Physique du Globe de Paris[2] and (3) the Black Forest Observatory[3]. Their parameters are well-known and tested.

The relationship between the earth's vibration at an earthquake's epicenter and a seismographs measurement can be modeled as

$$S_i(z) \cdot z^{-n} \cdot A_i \,,$$

with $0 < A_i < 1$ and $T_i = n_i T_s$.

In other words, the true earthquake is delayed and damped independent of its frequency solely due to the distance of the seismograph from the epicenter. Of course the measurement is also subject to the seismograph's dynamics.

[1] N 47.37857505586893, E 8.547217563936153
[2] N 48.844780319827855, E 2.3564190845681
[3] N 48.296104343412175, E 8.329414307713494

Tasks

1. **(2 points)** Approximate the times T_1, T_2 and T_3 at which the earthquake was measured at each seismographic station.

   - Use `measurement_1`, `measurement_2` and `measurement_3` to obtain the three seismographs' measurements. All measurements are evenly spaced with $T_s = 0.01$s
   - Apply a moving-median filter to remove measurement noise. Adjust the filter window as you see fit.
   - Read the time at which each seismograph measured the earthquake from a visualization of the respective filtered measurement.

2. **(2 points)** Approximate the distances d_1, d_2 and d_3 of each seismograph from the epicenter. Note, that you do not know when exactly the earthquake happened. Assume an earthquake travels at 12km/s.

   Note: you can neglect the earth's curvature.

3. **(1 point)** Triangulate the epicenter location on a map[4] and give an approximate location (in coordinates or give the name of the nearest town).

[4]This tool might be useful

Submission

Please hand in a zip-file named `ssprogex_<identity numbers>.zip` (please only submit zip-files, i.e., no 7z-files), where `<identity numbers>` is a list of the student identity numbers (Legi-Nummern) of all students who worked on the solution, separated by an underscore (e.g. `ssprogex_15-123-456_17-654-321.zip`). The zip-file should contain exactly the same files as the original handout. Please write your code in the provided template files. Up to three students are allowed to work together on the problem and all receive the same grade. At least one member of the group must submit the zip-file on Moodle before the due date and time. You are ultimately responsible that we receive your solution in time.

Jupyter Notebooks Before you submit any Jupyter Notebook please make sure that you only submit the code that we ask for. Remember to remove any debug functionality (e.g. plots, disp, etc.). For questions that require a written solution, we encourage you user the *Markdown* mode and directly write your answers in the provided Jupyter Notebook scripts.

PDF files You don't need to submit any Latex documents since you can directly write your analytical solutions in Jupyter Notebook. The only pdf that you must submit is a copy of the `DeclarationOfOriginality.pdf`, signed by **every student** who participated in the submission. Submissions that do not contain the document will not be graded. Group members who have not signed the form will receive 0 points.

Plagiarism

By signing the *Declaration of Originality*, students attest that (among other things): They have authored the work in question themselves; they have read the *Citation Etiquette* information sheet on plagiarism and adhered to the rules of citation standard in their disciplines. **Each submission will be electronically checked for plagiarism**.

Late and Faulty Submissions

Late submissions receive 0 points. Code or functions which fail to run (e.g. due to syntax errors, infinite loops, errors, etc) will receive 0 points; other sections of the submission will still be graded.