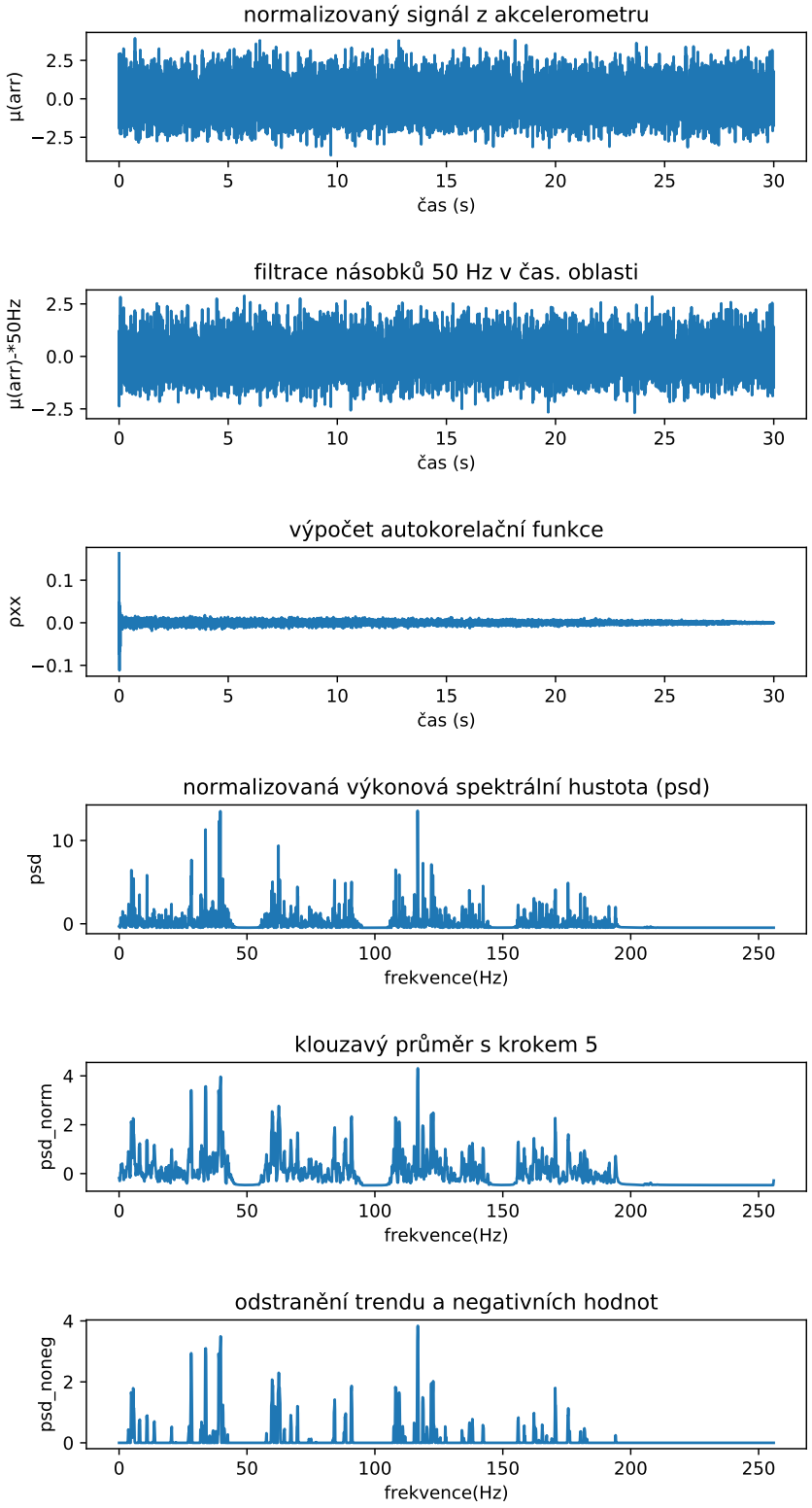
Obsah

1 Předzpracování dat



Obrázek 1: Přehled kroků předzpracování vstupního signálu.

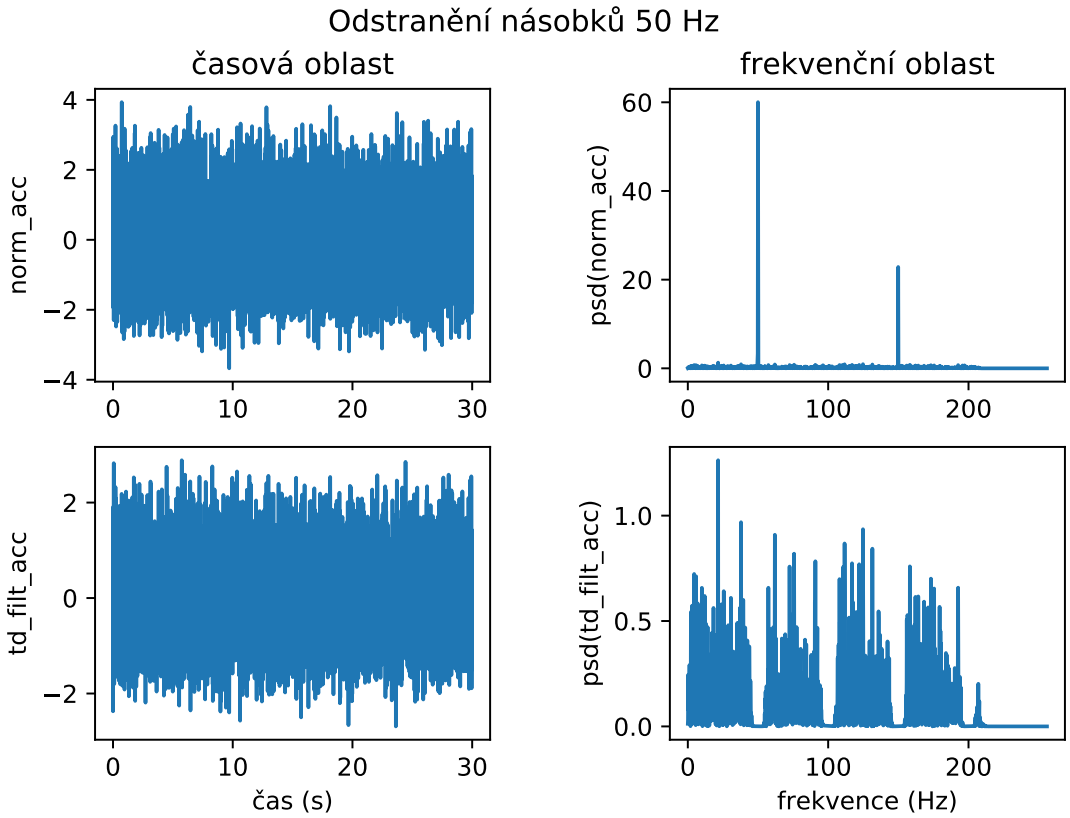
Pro předzpracování vstupních signálů (vibrací naměřených akcelerometry umístěnými na lampě) byla vytvořena třída *Preprocessor*, jejíž stěžejní úryvky kódu lze nalézt v příloze A. Na obrázku 1 je grafický přehled sekvence jednotlivých operací, které jsou na signál aplikovány. V tabulce 1 je uveden podrobný výčet proměnných, jejich rozměrů a metod, které k jejich výpočtu byly využity.

Tabulka 1: Přehled proměnných, jejich rozměrů a metod využitých pro jejich výpočet.

název	**rozměry obecně**	**rozměry konkrétně**	**popis**	**metoda**
arr	**[nsamples, nmeas]**	**[15360, :]**	signál vibrací z akcelerometru	-
$\mu(arr)$	[nsamples, nmeas]	[15360, :]	normalizace signálu vibrací	A.2
$\mu(arr)$-*50 Hz	[nsamples, nmeas]	[15360, :]	odstranění násobků 50 Hz	A.3
ρ_{xx}	**[nsamples - 1, nmeas]**	**[15359, :]**	autokorelační funkce signálu	A.4
psd	**[Nfft/2; nmeas]**	**[2560, :]**	výkonová spektrální hustota signálu	A.5
psd_{norm}	[Nfft/2; nmeas]	[2560, :]	normalizace psd	A.2
psd_{cg}	[Nfft/2; nmeas]	[2560, :]	ošetření psd plovoucím průměrem	A.6
$psd_{detrended}$	[Nfft/2; nmeas]	[2560, :]	odstranění absolutního trendu z psd	A.7
psd_{noneg}	[Nfft/2; nmeas]	[2560, :]	oříznutí negativních hodnot	A.8

1.1 Normalizace signálu a filtrace násobků 50 Hz

Naměřené vibrace jsou nejprve normalizovány (v obr. 1 značeno $\mu(arr)$, dále však pouze x) na nulovou střední hodnotu ($\mu = 0$) a jednotkový rozptyl ($\sigma = 1$) pomocí metody A.2. Na normalizovaný signál jsou následně aplikovány Butterworthovy bandstop (pásmová zádrž) filtry, které odstraňují násobky 50 Hz $(50, 100, 150, 200)$ a jejich okolí (± 5 Hz). Příslušné parametry filtrů jsou nalezeny metodou A.3.1 a aplikovány na normalizovaný signál metodou A.3.2 (viz obr. 2).



Obrázek 2: Porovnání normalizovaného signálu v časové oblasti (vlevo) a frekvenční oblasti (výkonová spektrální hustota, vpravo) před filtrací (nahoře) a po filtraci (dole) Butterworthovými filtry.

1.2 Výpočet autokorelační funkce

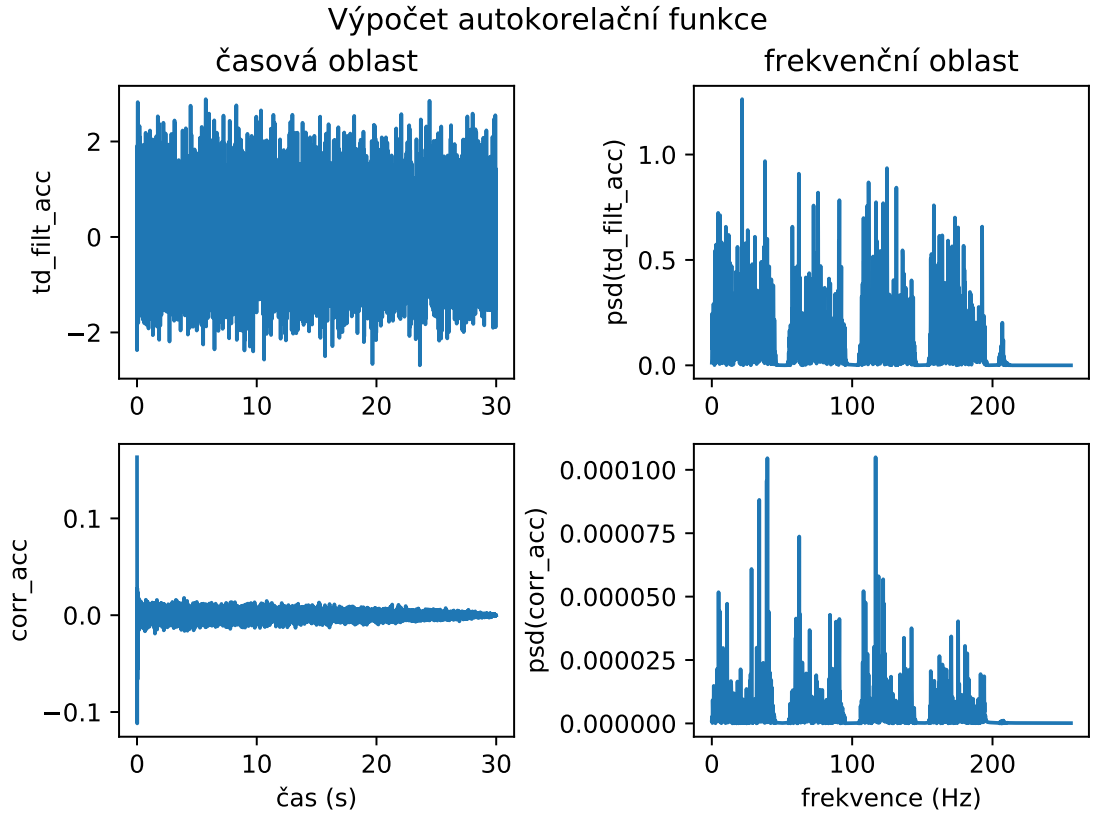
Ze signálu je následně vypočtena jeho autokorelační funkce, která zvyšuje odstup signálu od šumu (signal-to-noise ratio). Výpočet je dle [1] proveden následovně:

$$\rho_{xx}(m) = \frac{1}{N} \sum_{n=0}^{N-1-m} (x(n) \cdot x(n+m)), \text{kde } m \in (0, N-1) \tag{1}$$

kde x je vstupní signál a N je počet vzorků signálu.

Aby však ρ_{xx} nebyla závislá na střední hodnotě vstupního signálu [1] je navíc od autokorelační funkce odečtena druhá mocnina střední hodnoty signálu ($\mu(x)$):

$$\rho_{xx}(m) = \rho_{xx}(m) - (\mu(x))^2 \tag{2}$$



Obrázek 3: Normalizovaný signál bez násobků 50 Hz (nahoře) a jeho autokorelační funkce (dole) v časové oblasti (vlevo) a frekvenční oblasti (výkonová spektrální hustota, vpravo)

Výpočet autokorelační funkce je realizován maticově metodou A.4 a její porovnání s dosavadním signálem x je na obr. 3.

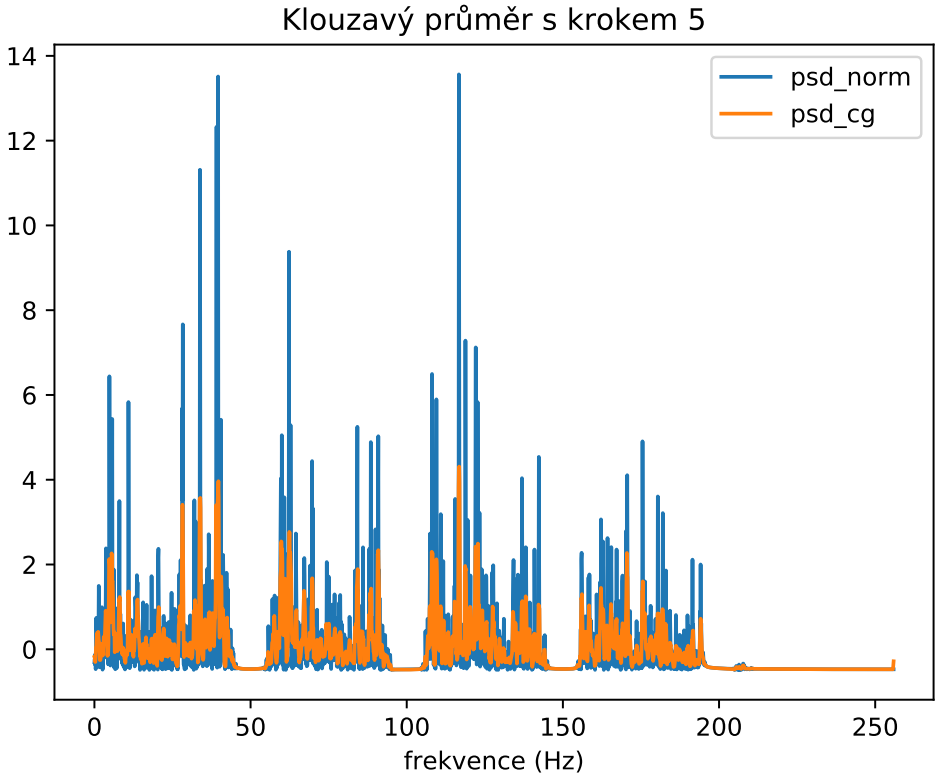
1.3 Převod na normalizovanou výkonovou spektrální hustotu (psd_{norm}) a ošetření plovoucím průměrem

Autokorelační funkce ρ_{xx} je ošetřena Hammingovou okénkovou funkcí (viz metoda A.5.1) a převedena pomocí diskrétní fourierovy transformace (DFT, resp. FFT) do frekvenční oblasti a z její absolutní hodnoty vypočtena výkonová spektrální hustota (viz metoda A.5.2) dle vztahu:

$$psd = \frac{1}{N}(FFT(\rho_{xx}))^2 \tag{3}$$

kde $N = 5120$ je délka FFT. Nyquistova délka je tedy 2560 vzork, což odpovídá frekvenci 250 Hz.

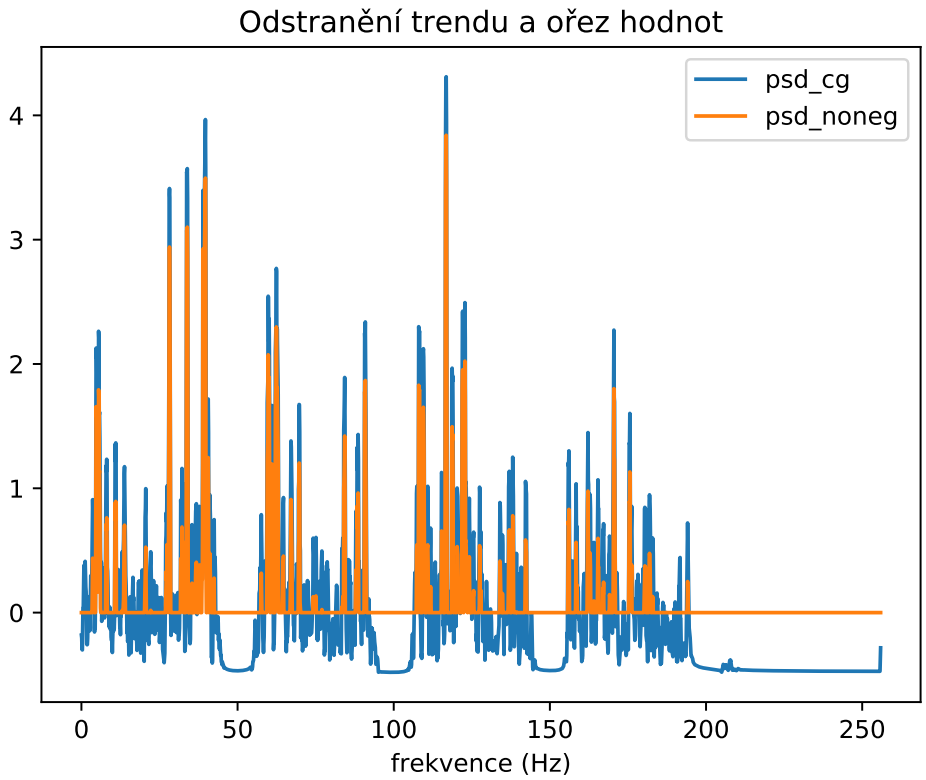
Výkonová spektrální hustota (psd) je dále normalizována na nulovou střední hodnotu a jednotkový rozptyl (psd_{norm}) a v metodě A.6 ošetřena plovoucím (klouzavým) průměrem se základním krokem 5 (psd_{cg}). Porovnání výkonových spektrálních hustot psd_{norm} a psd_{cg} lze pozorovat na obrázku 4



Obrázek 4: Aplikace metody A.6 na normalizovanou výkonovou spektrální hustotu (modrá, psd_{norm}) pro výpočet plovoucího (klouzavého) průměru s krokem 5 (oranžová, psd_{cg}).

1.4 Odstranění trendu a negativních hodnot

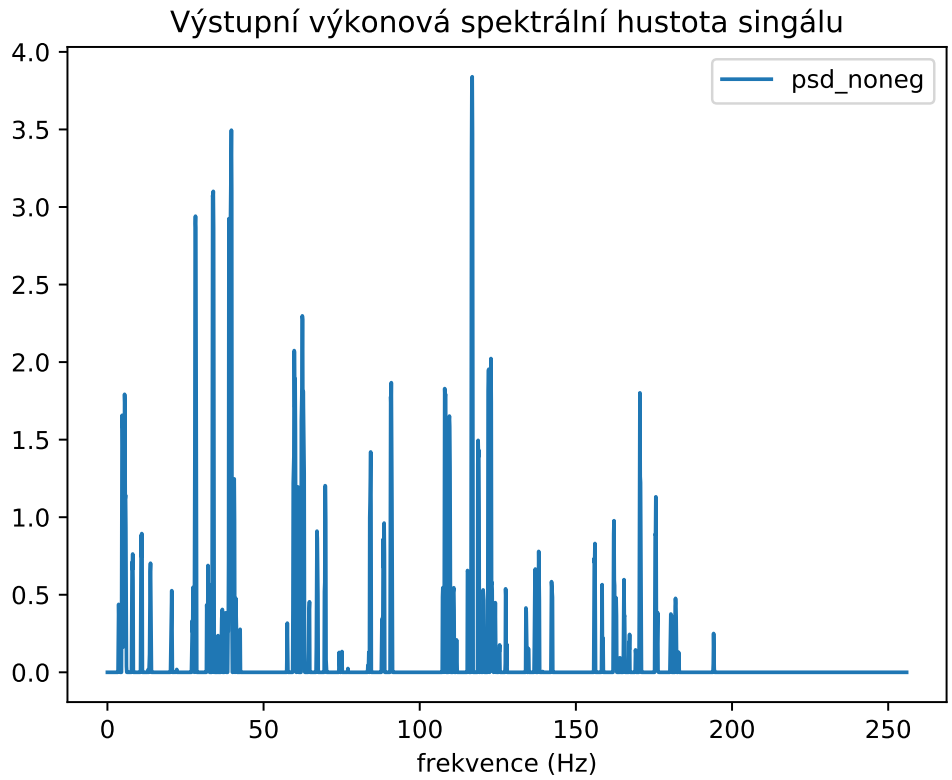
Závěrem je v psd_{cg} nalezen a odstraněn absolutní trend (viz metoda A.7). Nevýznamné části spektra jsou tímto uvrženy do záporných hodnot, které jsou posléze odstraněny (viz metoda A.8). Výsledkem je nezáporná výkonová spektrální hustota (psd_{noneg}). Porovnání psd_{cg} a psd_{noneg} se nachází na obr. 5.



Obrázek 5: Výkonová spektrální hustota před (modrá, psd_{cg}) a po odstranění absolutního trendu a oříznutí záporných hodnot (oranžová, psd_{noneg}).

1.5 Výstupní spektrum fáze předzpracování

V předchozí kapitole získaná psd_{noneg} je finálním výstupem fáze předzpracování (třída *Preprocessor*), který se v následujících kapitolách bude pro jednoduchost nazývat pojmem výkonová spektrální hustota nebo označením *psd*. Graf této výstupní hodnoty je na obrázku 6.



Obrázek 6: Příklad výstupní výkonové spektrální hustoty z fáze předzpracování.

2 Metoda vyhodnocení rozdílu křížové entropie multiškálově binarizovaného spektra

Metoda navržená ve třídě $M2$ ($Methods.M2$) slouží k vyhodnocení mechanického opotřebení sloupů sledované lampy na základě rozdílu v hodnotách informační křížové entropie (4) mezi multiškálově binarizovanými spektrálními výkonovými hustotami.

$$H(p,q) = \sum_{i=1}^{N} (p(x_i) \cdot \log_2(q(x_i))) \tag{4}$$

kde N je počet vzorků signálů.

2.1 Metoda *get_multiscale_distributions*

Výpočet binarizovaného spektra probíhá v následujících krocích:

1. rozdělení *psd* na frekvenční intervaly (košíky) o šířce *bs* (metoda *_split_to_bins* C.3)
2. zjištění celkového počtu hodnot, které v rámci každého košíku dosahují hodnoty vyšší než zvolený práh *th* (metoda *_binarize_and_softmax* C.4)
3. převod hodnot na distribuci pravděpodobnosti jednotlivých frekvencí (každý košík odpovídá střední frekvenci z rozsahu frekvencí, které zahrnuje) za pomocí funkce Softmax (metoda *_binarize_and_softmax* C.4)

Multiškálovost binarizace spočívá v definici několika prahových hodnot

$$ths = (th_1, th_2, ..., th_{Nths})$$

kde $Nths$ je počet definovaných prahů a několika velikostí košíků

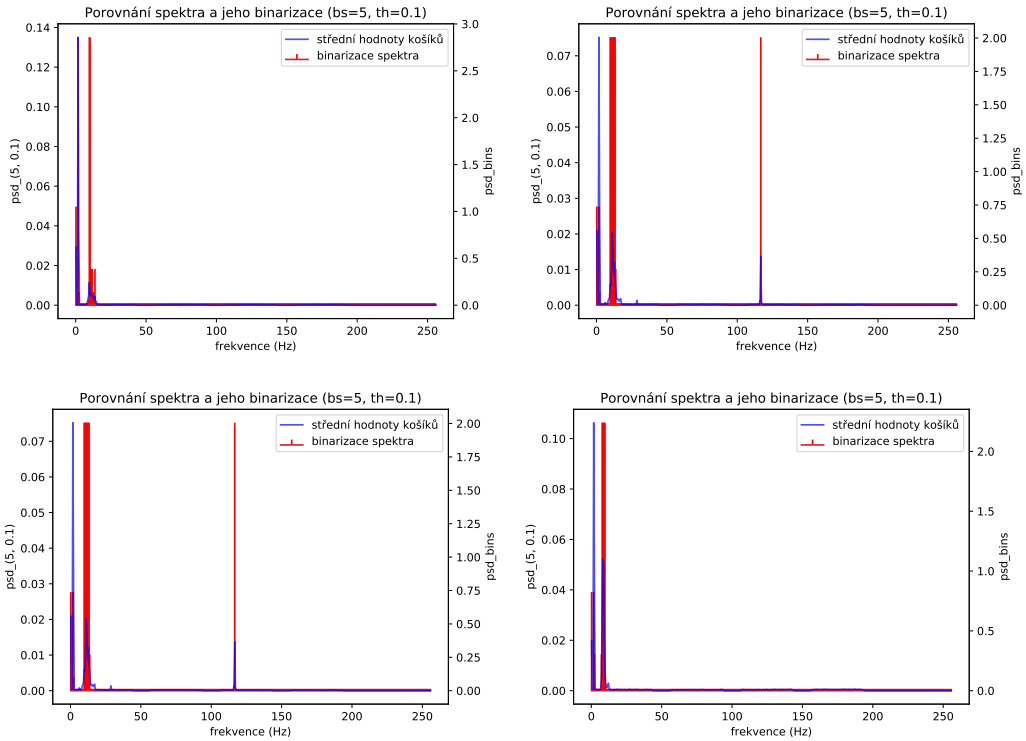
$$bsz = (bs_1, bs_2, ..., bs_{Nbs})$$

kde a Nbs je počet definovaných velikostí košíků. Např lze definovat:

$$\begin{aligned} bsz &= (5, 10, 20) \\ ths &= (0.1, 0.2, 0.4, 0.8) \end{aligned} \tag{5}$$

Hodnoty *bs* jsou definovány jakožto počty vzorků v košíku. Rozsah frekvencí, které košík zahrnuje je tedy závislý na periodě vzorkování vstupního pole *psd* (tzn. že pro periodu vzorkování $0,1\,\text{Hz}$ a např. $bs = 20$ odpovídá rozsah jednoho košíku $2\,\text{Hz}$).

Pro každou kombinaci *bs* a *th* z *bsz* a *ths* je poté vypočteno binarizované spektrum ($psd_{bs,th}$) dle výše zmíněných kroků. Takto binarizovaná spektra jsou uložena pro další zpracování. Celý tento proces výpočtu multiškálově binarizovaných spekter je souhrnně proveden metodou *get_multiscale_distributions* (C.2). Příklady binarizovaných spekter pro $bs = 5$ $(0,5\,\text{Hz})$ a $th = 0,1$ jsou na obrázku 7.



Obrázek 7: Příklad binarizace výkonové spektrální hustoty na binarizované spektrum s velikostí košíku $bs = 5$ a prahovou hodnotou $th = 0,1$.

2.2 Metoda *train*

Metoda *train* (C.5) spočívá ve výpočtu multiškálových binarizovaných spekter $psd_{bs,th}^{train}$ z dostatečně dlouhé trénovací datové sady signálů vibrací neporušené lampy (např. 2 měsíce). Hodnoty $psd_{bs,th}^{train}$ jsou uloženy jako proměnné instance třídy, aby mohly být využity k výpočtu křížové entropie mezi trénovacími a testovacími daty.

2.3 Chráněná metoda *_cross_entropy*

Chráněná metoda *_cross_entropy* (C.6) slouží k výpočtu informační křížové entropie mezi dvěma binarizovanými spektry dle vztahu (4).

2.4 Metoda *compare*

Metoda *compare* (C.7) využívá natrénovaných hodnot binarizovaných spekter $psd_{bs,th}^{train}$ jakožto jeden vstup do chráněné metody *_cross_entropy*. Druhým vstupem jsou cesty k *psd* souborům (dále značeno psd^{test}), které mají být s trénovacími daty

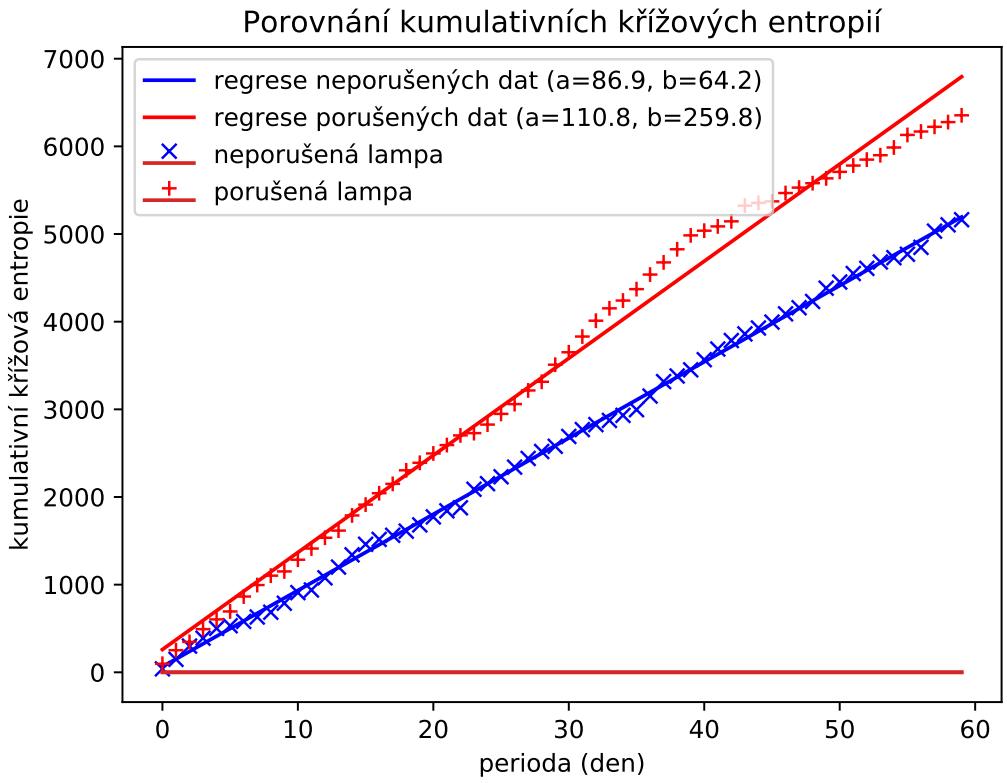
srovnány (pomocí křížové entropie). psd^{test} může být nejprve periodicky rozděleno do dnů či jiných časových intervalů a následně převedeno na binarizovaná spektra $psd^{test}_{bs,th}$ se stejnými prahovými hodnotami a velikostmi košíků, jaké byly definovány při trénování instance třídy (metoda *train*).

Metoda *compare* tedy dle 4 vypočte hodnotu křížové entropie (*ce*) mezi $psd^{train}_{bs,th}$ a $psd^{test}_{bs,th}$ a vrátí hodnotu *ce* z každého dne testovacích dat (pokud je zvolená perioda 1 den) pro každou kombinaci *bs* a *th*. Tyto hodnoty *ce* již mohou být využity pro další vyhodnocení mechanického poškození stožáru lampy např. vyhodnocením kumulativní křížové entropie.

2.5 Kumulativní křížová entropie

Z hodnot *ce* mezi trénovacími a testovacími daty lze vyhodnotit jakou rychlostí (s jakou směrnicí) se hodnoty *ce* kumulují (sčítají) v čase (ce_{cummul}).

Pro ukázku byl v souboru *main.py* vytvořen příklad kontinuálního vyhodnocení poškození stožáru lampy na základě kumulativní křížové entropie s periodou 1 den (viz příloha D). Zde je nejprve instance třídy $M2$ natrénována metodou *train* na 2 měsících dat z neporušené lampy (psd^{train}) pro $ths = (.001, .01, .1, .2, .5,)$ a $bsz = (80,)$. Následně jsou využity dvě další testovací datové sady. První sada odpovídá 2 dalším měsícům dat z neporušené lampy ($psd^{neporuseno}$). Druhá sada poté zahrnuje 2 měsíce, ve kterých byla lampa již mechanicky poškozena ($psd^{poruseno}$). Pomocí metody *compare* byly vypočteny hodnoty $ce^{neporuseno}$ a $ce^{poruseno}$ s periodou 1 den. Pro každý den je v *ce* k dispozici $Nths \cdot Nbs$ (konkrétně 5) hodnot křížové entropie z nichž byla vybrána vždy ta nejvyšší jak v případě $ce^{neporuseno}$ tak $ce^{poruseno}$. Kumulativním součtem těchto nejvyšších hodnot v jednotlivých dnech byly získány hodnoty $ce^{neporuseno}_{cummul}$ a $ce^{poruseno}_{cummul}$, které jsou vykresleny v grafu na obrázku 8 společně s aproximací jejich průběhu za pomocí lineární regrese.



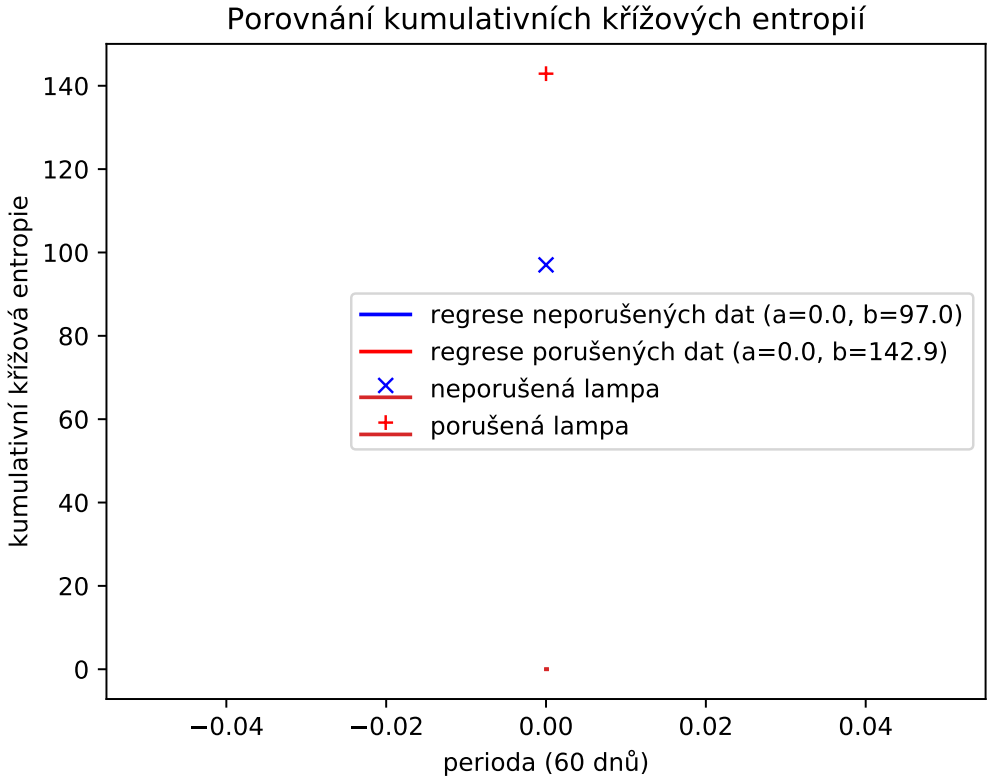
Obrázek 8: Porovnání kumulativních křížových entropií s neporušenými daty a porušenými daty a aproximace jejich průběhu lineární regresí.

Z obr. 8 je zřejmé, že regresní přímka aproximující $ce_{cummul}^{poruseno}$ stoupá rychleji (směrnice $a^{poruseno} = 110,8$) nežli regresní přímka aproximující $ce_{cummul}^{neporuseno}$ (směrnice $a^{neporuseno} = 86,9$). Křížová entropie porušených dat je tudíž v průměru vyšší nežli křížová entropie neporušených dat.

Pro průběžné vyhodnocení poškození stožáru lampy lze tedy kontinuálně pozorovat směrnici regresní přímky kumulativní křížové entropie za určité období (zde 60 dnu) a pokud naroste o určité procento od původní hodnoty, označit lampu ke kontrole porušení.

2.6 Porovnání křížových entropií agregovaných spektrálních hustot

Alternativní metodou vyhodnocení poškození by mohl být výpočet samotné multiškálové křížové entropie z agregace (průměru) testovaných výkonových spektrálních hustot za určité období a pozorování její hodnoty. Toho lze docílit v souboru *main.py* nastavením proměnné *period* na stejnou hodnotu jako *ndays*. Výstupem je v tomto případě pouze jedna hodnota křížové entropie pro porušená data a jedna pro neporušená data za *ndays* (zde konkrétně 60 dnu) (viz obr. 9).



Obrázek 9: Hodnoty křížových entropií mezi 2 měsíci trénovacích dat a 2 měsíci testovacích neporušených (modrá) a 2 měsíci testovacích porušených (červená) dat pro $ths = (.001, .01, .1, .2, .5,)$ a $bsz = (80,)$.

LITERATURA

[1] Pavlík, Radomír a Poláček, Vladimír. *Detekce užitečného signálu v aplikaci harmonického radaru s využitím MATLAB*. 2009. URL: `http://dsp.vscht.cz/konference_matlab/MATLAB09/prispevky/079_pavlik.pdf`.

PŘÍLOHY

A Výběr metod ze třídy *Preprocessor* (předzpracování)

A.1 Inicializační metoda třídy *Preprocessor*

```
def __init__(self,
             fs=512,
             ns_per_hz=10,
             freq_range=(0, 256),
             tdf_order=5,
             tdf_ranges=((45, 55), (95, 105), (145, 155), (195, 205)),
             use_autocorr = True,
             noise_f_rem=(0, ),
             noise_df_rem=(0, ),
             mov_filt_size=5,
             rem_neg=True):
    """
    :param fs: (int) sampling frequency of the acquisition
    :param ns_per_hz: (int) desired number of samples per Hertz in FFT
    :param freq_range: (list/tuple) (minimum frequency, maximum frequency)
    rest is thrown away
    :param tdf_order: (int) order of the time domain bandreject filter
    :param tdf_ranges: (List/Tuple[List/Tuple]) time domain filter
    bandreject frequency areas ((lb1, ub1), (lb2, ub2), ...)
    :param use_autocorr: (bool) if True, calculate autocorrelation
    function before transforming to psd
    :param noise_f_rem: (list/tuple) frequencies that should be removed (
    zeroed out) from the power spectrum
    :param noise_df_rem: (list/tuple) range around f_rem that should also
    be removed
```

```
21      :param mov_filt_size: (int) length of the rectangular filter for
        moving average application
        :param rem_neg: (bool) if True, remove negative values after the final
         preprocessing stage
23      """
        self.fs = fs
25      self.ns_per_hz = ns_per_hz
        self.freq_range = freq_range
27      self.tdf_order = tdf_order
        self.tdf_ranges = np.array(tdf_ranges)  # 2D array
29      self.use_autocorr = use_autocorr
        self.noise_f_rem = noise_f_rem
31      self.noise_df_rem = noise_df_rem
        self.mov_filt_size = mov_filt_size
33      self.rem_neg = rem_neg

35      # calculate numerators and denominators of time domain frequency
        filters
        self.nums, self.denoms = self._make_bandstop_filters()
37
        # initialize counter for conditional plot
39      self.cplot_call = 0

41      # initialize dict for semiresults
        self.semiresults = dict()
```

A.2 Normalizace vstupního signálu

```
@staticmethod
2 def _calc_zscore(arr):
      """ calculate zscore normalized array from arr across the 0th
      dimension
4
```

```
    :param arr: 2D array to be normalized (expected dimensions: [15360,
    :])
    :return zscore of arr [15360, :]:
    """
    assert len(arr.shape) == 2, "arr must be 2D array of [time samples,
    number of measurements]"
    return (arr - np.mean(arr, 0)) / np.std(arr, 0)
```

A.3 Filtrace pásmovou zádrží v časové oblasti

A.3.1 Vytvoření Butterworthových filtrů

```
def _make_bandstop_filters(self):
    """ calculate a bandstop filter params from given input parameters """
    f_nyquist = self.fs/2
    Wh = self.tdf_ranges/f_nyquist

    nfilts = Wh.shape[0]

    nums = np.empty((nfilts, 2*self.tdf_order + 1))
    denoms = np.empty((nfilts, 2*self.tdf_order + 1))

    for i in range(nfilts):
        b, a, *_ = butter(self.tdf_order, Wh[i, :], 'bandstop')
        nums[i, :] = b
        denoms[i, :] = a

    return nums, denoms
```

A.3.2 Aplikace Butterworthových filtrů na signál

```
def _apply_time_domain_filters(self, arr):
    """ apply all calculated bandstop filteres on arr """
    nfilts = self.nums.shape[0]
```

```
4
    for i in range(arr.shape[-1]):
6       for num, denom in zip(self.nums, self.denoms):
            arr[:, i] = filtfilt(num, denom, arr[:, i])  # updating in
    place
8   return arr
```

A.4 Výpočet autokorelační funkce

```
@staticmethod
2 def _autocorr(arr):
    """ calculate autocorrelation function (ACF) from the array (arr) to
    reduce noise
4       inspired by http://dsp.vscht.cz/konference_matlab/MATLAB09/
    prispevky/079_pavlik.pdf
    """
6   N = arr.shape[0]  # number of signal samples
    nmeas = arr.shape[1]  # number of measurements
8
    Rrr = np.zeros((N-1, nmeas))
10
    for i in range(nmeas):
12      a = arr[:, i]
        XX = hankel(a[1:])  # create hankel matrix from a[1] to a[N-1] (
    upper left triangular matrix)
14      vX = a[:-1]  # vector a[0] to a[N-2]
        Rrr[:, i] = np.matmul(XX, vX) / N - a.mean() ** 2  # calculate
    normalized ACF
16  return Rrr
```

A.5 Výkonová spektrální hustota (psd)

A.5.1 Aplikace Hammingova okna

```
@staticmethod
def _hamming(arr):
    """ Apply Hamming window across the 0th dimension in arr """
    return (arr.T*np.hamming(arr.shape[0])).T
```

A.5.2 Výpočet výkonové spektrální hustoty

```
def _calc_psd(self, arr):
    """ calculate power spectral density from arr across the 0th dimension

    :param arr: 2D array of time signals to be FFTed and PSDed (expected dimensions: [15360, :])
    :return: FIRST HALF OF THE freq_vals and psd (rest is redundant)
        :var freq_vals: 1D array of frequency values in self.freq_range (x axis) [fs*ns_per_hz//2],
        :var psd: 2D array of power spectral densities of singals in arr [fs*ns_per_hz//2, :]
    """
    assert len(arr.shape) == 2, "arr must be 2D array of [time samples, number of measurements]"

    nfft = self.fs * self.ns_per_hz
    arr = self._hamming(arr)
    arr_fft = np.fft.fft(arr, nfft, 0)
    arr_psd = np.abs(arr_fft)**2 / nfft

    freq_vals = np.arange(0, self.fs, 1.0 / self.ns_per_hz)

    # restrict frequency values to the range of self.freq_range
    freq_slice = slice(self.freq_range[0]*self.ns_per_hz, self.freq_range[1]*self.ns_per_hz)

```

```
    return freq_vals[freq_slice], arr_psd[freq_slice, :]
```

A.6 Ošetření plovoucím průměrem

```
def _coarse_grain(self, psd_arr):
    """
    Calculating moving average using rectangular filter of length "
    filt_len"

    :param psd_arr: array of power spectral densities [nfft, :]

    :return psd_arr_cg: coarse grained psd_arr using moving average
    """
    psd_arr_cg = np.zeros(psd_arr.shape)
    filt = np.ones(self.mov_filt_size) / self.mov_filt_size

    for i in range(psd_arr.shape[1]):
        psd_arr_cg[:, i] = np.convolve(psd_arr[:, i], filt, 'same')

    return psd_arr_cg
```

A.7 Odstranění trendu

```
def _detrend(self, freqs, psd_arr):
    """
    Remove linear trend from y=psd_arr(x) based on Least Squares optimized
     curve fitting

    :param freqs: x
    :param psd_arr: y
    :return: psd_arr_detrended
    """
    nfs, nms = psd_arr.shape
```

```
    psd_arr_detrended = np.zeros((nfs, nms))
    min_idx, max_idx = (int(self.ns_per_hz*f) for f in self.freq_range)

    for i in range(nms):
        par_init = (1., )

        x = freqs[min_idx:max_idx]
        y = psd_arr[min_idx:max_idx, i]

        par, success = leastsq(self.__error_func, par_init, args=(x, y,
    self.__linear_func))

        y_trend = self.__linear_func(par, x)

        psd_arr_detrended[:, i] = np.concatenate((psd_arr[:min_idx, i], y
    - y_trend, psd_arr[max_idx:, i]))

    return psd_arr_detrended
```

A.8 Odstranění negativních hodnot

```
def _remove_negative(self, psd_arr):
    """
    Remove negative values from psd

    :param psd_arr: array of power spectral densities [nfft, :]
    :return: psd_arr_positive
    """
    return psd_arr.clip(min=0)
```

B Třída *Method*, která je rodičem metody *M2*

```
class Method:

    def __init__(self, preprocessor=Preprocessor(), from_existing_file=
    True):
        """
        :param preprocessor: (obj) Preprocessor class instance which
    determines how to preprocess PSD files
        :param from_existing_file: (bool) whether to try to load from
    existing files or ignore them
        """
        self.preprocessor = preprocessor
        self.from_existing_file = from_existing_file

    @staticmethod
    def _calc_mean_and_var(psd, period=None, nmeas=144):
        naccs, nfft, ndm = psd.shape
        if period:
            nsamples = ndm//nmeas//period
        else:
            nsamples = 1

        mean = np.empty((nsamples, naccs, nfft))
        var = np.empty((nsamples, naccs, nfft))

        psd = np.array_split(psd, nsamples, axis=-1)  # split to cca "
    period" long entries
#         print(f"period: {period}, nsamples: {nsamples}, len(psd): {len(
    psd)}")
        # calculate mean and var from each entry
        for i in range(nsamples):
            mean[i, :, :] = psd[i].mean(axis=2)
            var[i, :, :] = psd[i].var(axis=2)
```

```
28
        return mean, var
30
    def _get_PSD(self, path, period=None, remove_0th_dim=False):
32      """ load freqs and PSD if the preprocessed files already exist,
    otherwise calculate freqs and PSD and save them

34      :param path: (string) path to files with data
        :param period: (int) number of days from which to aggregate
36
        :return (freqs(1Darray), mean(1Darray), var(1Darray)):
38      """
        if ".npy" in os.path.splitext(path)[-1]:
40          folder_path = os.path.split(path)[0]
        else:
42          folder_path = path
        path_to_freqs = folder_path + "/freqs.npy"
44      path_to_PSD = folder_path + "/PSD.npy"
        path_to_vars = folder_path + "/PSDvar.npy"
46
        try:
48          if self.from_existing_file and "X.npy" in path:
                nlamps = FLAGS.nlamps
50              freqs, X = self.preprocessor.run([path], return_as="
    ndarray")

52              X = np.split(X, nlamps, axis=0)  # reshape from (ndays.
    nmeas.nlamps, nfft//2, naccs//nlamps)
                X = np.dstack(X)                 # to (ndays.nmeas, nfft
    //2, naccs)
54              psd = X.transpose((2, 1, 0))     # and then to (naccs,
    nfft//2, ndays.nmeas)
```

```
56                mean, var = self._calc_mean_and_var(psd, period)

58            elif self.from_existing_file:
                  freqs = np.load(path_to_freqs)
60                mean = np.load(path_to_PSD)
                  var = np.load(path_to_vars)
62            else:
                  print("\nIgnoring existing files!")
64                raise FileNotFoundError
      except FileNotFoundError:
66            freqs, psd = self.preprocessor.run([path], return_as="ndarray"
)
              ndays, naccs, nfft, nmeas = psd.shape
68            # reshape from (ndays, naccs, nfft/2, nmeas) to (naccs, nfft
//2, ndays.nmeas)
              psd = psd.transpose((1, 2, 0, 3))
70            psd = psd.reshape((naccs, nfft, ndays*nmeas))

72            # calculate PSD (== long term average values of psd)
              mean, var = self._calc_mean_and_var(psd, period=None, nmeas=
nmeas)
74
              # save freqs and PSD files
76            if ".mat" not in path:
                  np.save(path_to_freqs, freqs)
78                np.save(path_to_PSD, mean)
                  np.save(path_to_vars, var)
80
      if not period and remove_0th_dim:
82            mean = mean.reshape(mean.shape[1:])
              var = var.reshape(var.shape[1:])
```

```
84
        return freqs, mean, var
```

C Výběr metod ze třídy $M2$ (křížová entropie)

C.1 Inicializace metody $M2$

```
1  class M2(Method):
2
3      def __init__(self, preprocessor=Preprocessor(), from_existing_file=
       True, var_scaled_PSD=False):
4          super(M2, self).__init__(preprocessor, from_existing_file=
       from_existing_file)
5
6          self.bin_sizes = None
7          self.thresholds = None
8          self.var_scaled_PSD = var_scaled_PSD
9
10         self.trained_distributions = None
```

C.2 Výpočet multiškálově binarizovaných spekter

```
def get_multiscale_distributions(self, path, bin_sizes=(10, ), thresholds
    =(0.1, ), period=None):
2       """ Calculate multiscale distributions from files given in path for
    each combination of values in
        bin_sizes and thresholds.
4
        :param path: path to files with psd files
6       :param bin_sizes: tuple with desired bin sizes
        :param thresholds: tuple with desired thresholds
8       :param period: period with which to calculate the distributions
```

```
    :return multiscale_distributions: List[(1, 1), 1D array[nbins], 2D
array[naccs, nbins]]
    """

    freqs, PSD, PSD_var = self._get_PSD(path, period)

    multiscale_distributions = list()

    for i, (mean, var) in enumerate(zip(PSD, PSD_var)):
        if self.var_scaled_PSD:
            var = (var - var.min())/(var.max() - var.min())  # normalize
to interval (0, 1)
            mean = mean*var

        md = list()

        # multiscale (grid search way)
        for bin_size in bin_sizes:
            for threshold in thresholds:
                freq_bins, PSD_bins = self._split_to_bins(freqs, mean,
bin_size)

                freq_binarized_mean = freq_bins.mean(axis=-1)
                PSD_binarized_softmax = self._binarize_and_softmax(
PSD_bins, threshold)

                md.append([(bin_size, threshold), freq_binarized_mean,
PSD_binarized_softmax])

        multiscale_distributions.append(md)

```

```
    return multiscale_distributions if period else
    multiscale_distributions[0]
```

C.3 Rozdělení na frekvenční intervaly (košíky)

```
@staticmethod
def _split_to_bins(freqs, psd_array, bin_size):
    """Take psd_array and split it into bins (frequency-wise).

    :param freqs: array of input frequencies [number of frequencies, ]
    :param psd_array: array of input psd values [number of measurements,
    number of frequencies]
    :param bin_size: (int) desired bin size in number of values (not
    frequencies)

    :return freq_bins (2D array) [number of bins, size of one bin]
            psd_bins: (3D array) [number of measurements, number of bins,
    size of one bin]
    """
    naccs, nfreqs = psd_array.shape

    nbins = nfreqs // bin_size + (nfreqs % bin_size > 0)

    freq_bins = np.zeros((nbins, bin_size), dtype=np.float32)
    psd_bins = np.zeros((naccs, nbins, bin_size), dtype=np.float32)

    for i in range(nbins):
        area = slice(i*bin_size, (i + 1)*bin_size)
        freq_bins[i, :] = freqs[area]
        psd_bins[:, i, :] = psd_array[:, area]

    return freq_bins, psd_bins
```

C.4 Výpočet binarizovaného spektra s využitím funkce softmax

```
@staticmethod
def _binarize_and_softmax(psd_bins, threshold):
    """ Calculate binarized values in bins scaled by softmax to
    probability distribution with sum of 1

    :param psd_bins: (3D array) psd array which is split to bins [:, nbins
    , bin_size]
    :param threshold: (int) desired cutoff value for binarization
    :return psd_binarized_softmaxed: (2D array) [:, nbins]
    """
    psd_binarized = np.array(psd_bins > threshold, dtype=np.float32)

    psd_binarized_sum = psd_binarized.sum(axis=-1)
    psd_sum_of_exp = np.exp(psd_binarized_sum).sum(axis=-1)

    psd_binarized_softmaxed = np.exp(psd_binarized_sum)/np.expand_dims(
    psd_sum_of_exp, 1)

    return psd_binarized_softmaxed
```

C.5 Natrénování třídy M2 na trénovacích signálech vibrací

```
def train(self, path, bin_sizes, thresholds):
    """ calculate binarized distributions of PSD from path for each
    combination of bin_size and threshold nd then save them for further
    use

    :param path: path to file(s) containing psd arrays for training
    :param bin_sizes: tuple of desired bin sizes
    :param thresholds: tuple of desired thresholds
    :return: None
    """
```

```
        self.bin_sizes = bin_sizes
        self.thresholds = thresholds

        self.trained_distributions = self.get_multiscale_distributions(path,
    self.bin_sizes, self.thresholds)

        print("Training complete!")
```

C.6 Výpočet křížové entropie

```
@staticmethod
def _cross_entropy(d1, d2):
    """ Calculate information cross-entropy between d1 and d2

    :param d1: distribution 1
    :param d2: distribution 2
    :return: -sum(d1(x).log2(d2(x)))
    """

    return -np.sum(d1*np.log2(d2))
```

C.7 Porovnání dvou sad binarizovaných spekter z hlediska křížové entropie

```
def compare(self, path, period=None, print_results=True):
    """ Calculate cross entropy between psd loaded from path and trained
    multiscale distributions (M2().train).
    If period is specified, the compared psd files will first be split by
    days (e.g. period=1 means that ce will be
    calculated for each day of psd_comp sepparately)
```

```
    :param path: path to file(s) containing psd arrays for comparison with
     training data
    :param period: (int) split the comparison into periods of days (None==
    period is same as length of the data)
    :param print_results: (int)
    :return ce: array of cross entropies for each threshold, bin_size and
    period
    """


    if not self.trained_distributions:
        raise(ValueError, "Nejdříve je třeba metodu natrénovat (M2().train
    ).")

    valid_distributions = self.get_multiscale_distributions(path, self.
    bin_sizes, self.thresholds, period)

    nperiods = len(valid_distributions)
    nbins = len(self.bin_sizes)
    nth = len(self.thresholds)

    ce = np.empty((nperiods, nbins*nth))

    if print_results:
        print("\n--CROSS ENTROPY VALUES--")
        print(f"|| period | bs |  th  || CEnt ||")
    for per, period_dist in enumerate(valid_distributions):
        for i, (((bin_sz, th), _, dtrain), ((_, _), _, dval)) in enumerate
    (zip(self.trained_distributions, period_dist)):
            ce[per, i] = self._cross_entropy(dtrain, dval)
            if print_results:
```

```
                print(f'|| {per:6d} | {bin_sz:2d} | {th:.2f} || {ce[per, i
    ]:4.0f} ||')
32
    return ce
```

D *main.py*: Příklad vyhodnocení poškození stožáru lampy na základě směrnice kumulativní křížové entropie

```
1  from collections import Counter
3  from flags import FLAGS
   from preprocessing import Preprocessor
5  from Methods import M2
7  import numpy as np
   from matplotlib import pyplot as plt
9
11 def calc_periodic_best(ce, bin_sizes, thresholds):
       nperiods, nparams = ce.shape
13     nth = len(thresholds)
15     best_params = [np.nan]*nperiods
       periodic_best = np.empty((nperiods,), dtype=np.float32)
17     periodic_best[:] = -np.inf
       for i, day in enumerate(ce):
19         best_j = 0
           for j, val in enumerate(day):
21             if val > periodic_best[i]:
                   periodic_best[i] = val
```

```
                best_j = j
        best_params[i] = divmod(best_j, nth)
        print(f"day: {i}, bs: {bin_sizes[best_params[i][0]]}, th: {
    thresholds[best_params[i][1]]} val: {periodic_best[i]}")
    return best_params, periodic_best


def linear_regression(y, x=None):
    """Calculate params of a line (a, b) using least squares algorithm to
    best fit the input data (x, y)"""
    ndata = len(y)
    if not all(x):
        x = np.arange(0, ndata, 1)
    else:
        assert len(x) == ndata, "x and y don't have the same length"
    A = np.vstack((x, np.ones(ndata))).T
    a, b = np.linalg.lstsq(A, y, rcond=None)[0]
    return a, b


if __name__ == "__main__":
    # Paths to files
    setting = "training"
    folder = FLAGS.paths[setting]["folder"]
    dataset = FLAGS.paths[setting]["dataset"]
    period = [FLAGS.paths[setting]["period"]]*len(dataset)
    filename = ["X.npy"]*len(dataset)
    paths = [f"./{folder}/{d}/{p}/{f}" for d, p, f in zip(dataset, period,
     filename)]
    from_existing_file = True

    # multiscale params
```

```
    bin_sizes = (80, )
    thresholds = (.001, .01, .1, .2, .5, )
    plot_distributions = False

    # periodic params
    ndays = 60
    period = 1

    # define instance of Preprocessor and initialize M2
    preprocessor = Preprocessor()
    m2 = M2(preprocessor, from_existing_file=from_existing_file)

    # Train the method on 2 months of neporuseno (trained)
    m2.train(paths[0], bin_sizes, thresholds)

    # Calculate cross entropy of
    ce2 = m2.compare(paths[1], period=period, print_results=False)  #
trained with neporuseno2
    ce3 = m2.compare(paths[2], period=period, print_results=False)  #
trained with poruseno
    ce23 = np.vstack((ce2, ce3))
    # print(ce2.shape)  # (nperiods, nbins*nthresholds)

    # Find the highest cross-entropy for each day
    ce2_best_params, ce2_periodic_best = calc_periodic_best(ce2, bin_sizes
, thresholds)
    ce3_best_params, ce3_periodic_best = calc_periodic_best(ce3, bin_sizes
, thresholds)

    # Count which and how many times have combinations of (bin_size,
threshold) been chosen as highest ce
    ce2_best_js = Counter(ce2_best_params)
```

```
    ce3_best_js = Counter(ce3_best_params)
    print(ce2_best_js)
    print(ce3_best_js)

    # Calculate cummulative sum of cross-entropies
    x = np.arange(0, len(ce2_periodic_best), 1)[:ndays]
    y2 = np.cumsum(ce2_periodic_best)[:ndays]
    y3 = np.cumsum(ce3_periodic_best)[:ndays]

    # Calculate params for linear regressions
    a2, b2 = linear_regression(y2, x)
    a3, b3 = linear_regression(y3, x)

    # plot the results of cummulative cross-entropies and their regression
    plt.plot(x, a2*x + b2, "b", label=f"regrese neporušených dat (a={a2:.1f}, b={b2:.1f})")
    plt.plot(x, a3*x + b3, "r", label=f"regrese porušených dat (a={a3:.1f}, b={b3:.1f})")
    plt.stem(y2, markerfmt="bx", linefmt="none", use_line_collection=True, label="neporušená lampa")
    plt.stem(y3, markerfmt="r+", linefmt="none", use_line_collection=True, label="porušená lampa")
    plt.xlabel(f"perioda ({period} " + ("den" if period == 1 else "dnů") + ")")
    plt.ylabel("kumulativní křížová entropie")
    plt.title("Porovnání kumulativních křížových entropií")
    plt.legend()

    # save the resulting plot
    plt.savefig(f"./images/M2/cummul_ce_nd-{ndays}_p-{period}.pdf")
    plt.show()
```