Cue-distractor package
DRAFT

May 4, 2015

1 Interface

- the framework has its own namespace `cdf`, this prefix is omited below

1.1 Structures

- experimental data are kept in `hRun` and `hTrial` structures
- structure `hConfig` holds tuning parameters and controls framework bevaviour
- all structures are handle classes (derived from `matlab.mixin.Copyable`), so they are passed by reference(!), clones can be created by invoking their `copy()` method

Run

- for each subject and recording there is a single `hRun` structure
- all of its fields need to be set by application
- format of `hRun.audiodata` is normalized, two-channel (response in first), time along rows as returned by matlab's `wavread` function
- `hRun.audiolen` needs to be specified in units of samples

field	type	default	description
`hRun.id`	scalar numeric	`NaN`	subject identifier
`hRun.audiofile`	row char	`''`	audio filename
`hRun.audiodata`	matrix numeric	`[]`	audio data
`hRun.audiolen`	scalar numeric	`NaN`	audio length
`hRun.audiorate`	scalar numeric	`NaN`	sampling rate
`hRun.trials`	row object	`[]`	vector of trials

Trial

- `hTrial` structure holds information about trial conditions and subject's response
- trial conditions need to be set by application
- there a two sets of response information: `hTrial.detected` and `hTrial.labeled`
- former one is set by framework during detection passes
- latter one needs to be set by application if annotation data are available
- time values (ranges, positions, etc.) need to be specified in units of samples and are used as global(!) indices of audio data
- empty labels (`''`) are invalid ones, none labels should be encoded as `'none'`

field	type	default	description
`hTrial.id`	scalar numeric	`NaN`	trial identifier
`hTrial.range`	row numeric	`[NaN, NaN]`	trial range
`hTrial.cuelabel`	row char	`''`	cue label
`hTrial.distlabel`	row char	`''`	distractor label
`hTrial.cue`	scalar numeric	`NaN`	cue position
`hTrial.soa`	scalar numeric	`NaN`	stimulus-onset asynchrony
`hTrial.distbo`	scalar numeric	`NaN`	distractor burst-onset
`hTrial.distvo`	scalar numeric	`NaN`	distractor voice-onset
`hTrial.detected.range`	row numeric	`[NaN, NaN]`	response range
`hTrial.detected.label`	row char	`''`	response label
`hTrial.detected.bo`	scalar numeric	`NaN`	response burst-onset
`hTrial.detected.vo`	scalar numeric	`NaN`	response voice-onset
`hTrial.detected.vr`	scalar numeric	`NaN`	response voice-release
`hTrial.detected.featfile`	row char	`''`	feature filename
`hTrial.labeled`			same as `hTrial.detected`

Configuration

- `hConfig` structure holds everything related to controlling framework behaviour
- default values are tuned and should work out of box
- refer to internals section for specific meaning and effects
- time values need to be specified in milliseconds(!), powers in decibels and frequencies in hertz

field	type	default	description
`hConfig.sync_mrklen`	scalar numeric	`1`	sync marker length
`hConfig.sync_thresh`	scalar numeric	`3`	sync detection threshold
`hConfig.sync_range`	scalar numeric	`[-25, 5]`	sync detection range
`hConfig.sta_frame`	row numeric	`[15, 5]`	short-time frame
`hConfig.sta_wnd`	scalar object	`@hann`	window function
`hConfig.sta_band`	row numeric	`[100, 8000]`	frequency band limits
`hConfig.glottis_band`	row numeric	`[100, 500]`	glottis band limits
`hConfig.glottis_rordt`	scalar numeric	`10`	rate-of-rise delta
`hConfig.glottis_rorpeak`	scalar numeric	`6`	ror peak power threshold
`hConfig.schwa_length`	scalar numeric	`20`	schwa vowel length
`hConfig.schwa_power`	scalar numeric	`-20`	relative schwa power
`hConfig.plosion_threshs`	row numeric	`[20, 10]`	plosion thresholds
`hConfig.plosion_delta`	scalar numeric	`1`	plosion delta
`hConfig.plosion_width`	scalar numeric	`10`	plosion width
`hConfig.feat_intcount`	scalar numeric	`10`	subsequences interval count
`hConfig.feat_intlen`	scalar numeric	`5`	minimum interval length

1.2 Workflow

- the following passes are proposed in order of processing
- passes may be processed on single or multi-run basis, but for each single run the given order has to be followed
- it is suitable to write data to disk after each pass to enable pass-wise testing and debugging

Raw conversion

- this pass is totally application-specific and depends on experimental data format

- at the end of this pass application should provide a valid `hRun` structure containing proper `hTrial` structures

Syncing

- as there is a still not understood asynchrony between trigger and recording devices timings need to be synced
- in fact this pass is optional and not needed for short recordings, for large recordings it is crucial
- upon completion all trial timings are adjusted to fit with sync marker positions

Response extraction

- the first pass which actually involves detection techniques is the coarse extraction of subject's speech from recording
- this is the slowest pass as it works on complete trial audio data and full bandwidth spectrum
- after processing trial response ranges (`hTrial.detected.range`) are set properly
- its primary goal is data reduction for further processing

Landmark detection

- this pass is internally split into glottis activity and burst detection
- it processes rather fast since it works only on partial audio data and a spectral subband
- after execution response landmarks (`hTrial.detected.bo`, `.vo` and `.vr`) are set

Babbling spectrum

- this pass can be performed on both detected and labeled data
- it estimates the average power spectrum of response speech parts

1.3 Functions

Syncing

```
offs = sync( run, cfg, sync_resp )
```

- this function syncs trial timings to fit with sync markers positions
- experimental data are specified through input `run` of type `hRun`
- marker detection parameters are given through input `cfg` of type `hConfig` (using `.sync_*`)
- if `sync_resp` is set to `false` response timings will not be synced (useful if annotation data are wrongly synced)
- after execution trial timings are adjusted and sync marker offsets in number of samples are returned in vector `offs`

Response extraction

```
extract( run, cfg )
```

- this function extracts subject's speech parts
- experimental data are specified via input `run` of type `hRun`
- extraction is controlled by input `cfg` of type `hConfig` (using `.sta_*`)
- after execution detected response ranges (`hTrial.detected.range`) are properly set
- if no valid range is found corresponding range is set to default `[NaN, NaN]`

Landmark detection

`landmark( run, cfg )`

- this function detects landmarks (+b, ±g) in response
- experimental data are specified via input `run` of type `hRun`
- extraction is controlled by input `cfg` of type `hConfig` (using `.glottis_*`, `.schwa_*` and `.plosion_*`)
- after execution response landmarks (`hTrial.detected.bo`, `.vo` and `.vr`) are set
- landmarks which are not detected are set to default `NaN`

Babbling spectrum

`[pows, freqs] = babbling( run, cfg, labeled, landmarks )`

- this function estimates the average power spectrum of response speech parts
- experimental data are specified via input `run` of type `hRun`
- if input flag `labeled` is set to `true` annotated data will be analyzed
- input flag `landmarks` controls whether to use landmarks or extraction ranges for spectral analysis
- analysis is controlled by input `cfg` of type `hConfig` (using `.sta_*`)
- after execution the function returns vectors `pows` and `freqs` containing spectral powers and frequency bins of the average spectrum

1.4 Plots

- for testing and debugging framework functionality there are some prepared plot functions residing in namespace `cdf.plot`
- all of these functions do not show any plots, instead they write images to disk

`plot.sync( run, offs, plotfile )`

- this function plots sync marker offsets to `plotfile`
- experimental data are given in input `run` of type `hRun`
- input vector `offs` needs to be specified as returned by `sync` function

`plot.trial_range( run, cfg, trial, range, rzp, plotfile )`

- this function plots a trial range to `plotfile`
- the plot includes two-channel audio data, full bandwidth spectrogram, response ranges and landmarks
- experimental data are given by input `run` of type `hRun` using `trial` of type `hTrial`
- plot range can be adjusted by input `range` with zero point `rzp`
- input configuration `cfg` of type `hConfig` is used for spectrogram generation (`.sta_*`)

`plot.extract( run, detected, labeled, plotfile )`

- this function plots response extraction accuracies to `plotfile`
- the plot includes range start and stop deltas and range overlap, pointless without any annotation data
- experimental data are given by input `run` of type `hRun` and matrices `detected` and `labeled` holding detected and annotated response ranges

`plot.trial_extract( run, cfg, trial, plotfile )`

- this function plots extraction internals to `plotfile`

- the plot includes response audio data, denoised total power and voice activity
- experimental data are given by input `run` of type `hRun` using `trial` of type `hTrial`
- actually this function re-extracts response range from a single trial using input configuration `cfg` of type `hConfig`

`plot.landmark( run, detected, labeled, plotfile )`

- this function plots landmark detection accuracies to `plotfile`
- the plot includes burst-onset, voice-onset and voice-release deltas, pointless without any annotation data
- experimental data are given by input `run` of type `hType` and matrices `detected` and `labeled` holding detected and annotated landmarks

`plot.trial_glottis( run, cfg, trial, plotfile )`

- this function plots glottis landmark detection internals to `plotfile`
- the plot includes response subband spectrogram, denoised maximum power and rate-of-rises with peaks
- experimental data are given by input `run` of type `hRun` using `trial` of type `hTrial`
- actually this function re-detects glottis landmarks in a single trial using input configuration `cfg` of type `hConfig`

`plot.trial_burst( run, cfg, trial, plotfile )`

- this function plots burst landmark detection internals to `plotfile`
- the plot includes response audio data and plosion index
- experimental data are given by input `run` of type `hRun` using `trial` of type `hTrial`
- actually this function re-detects burst landmark in a single trial using input configuration `cfg` of type `hConfig`

`plot.timing( run, detected, labeled, plotfile )`

- this function plots landmark timings to `plotfile`
- the plot includes voice-onset time, vowel and syllable lengths
- experimental data are given by input `run` of type `hType` and matrices `detected` and `labeled` holding detected and optional annotated landmarks

`plot.babbling( pows, freqs, plotfile )`

- this function plots babbling power spectrum to `plotfile`
- spectrum data are specified by input vectors `pows` and `freqs` as returned by `babbling`

2 Example

- shown below is a minimal example of how to use framework's detection functions and plot corresponding tests
- audio data are supposed to be in file `'audio.wav'`, trial data in file `'log.txt'` and manually annotated data in file `'labels.xlsx'`
- application-specific functions `read_audio`, `read_trials` and `read_labels` are supposed to read data properly into specified `run` structure

```
cfg = cdf.hConfig(); % default configuration

run = cdf.hRun(); % read raw data
read_audio( run, 'audio.wav' );
read_trials( run, 'log.txt' );
read_labels( run, 'labels.xlsx' );

offs = cdf.sync( run, cfg, false ); % sync timings
cdf.plot.sync( run, offs, 'sync.png' ); % plot test

cdf.extract( run, cfg ); % extract responses
detected = [run.trials.detected];
detected = cat( 1, detected.range );
labeled = [run.trials.labeled];
labeled = cat( 1, labeled.range );
cdf.plot.extract( run, detected, labeled, 'extract.png' ); % plot test

cdf.landmark( run, cfg ); % detect landmarks
detected = [run.trials.detected];
detected = cat( 2, [detected.bo]', [detected.vo]', [detected.vr]' );
labeled = [run.trials.labeled];
labeled = cat( 2, [labeled.bo]', [labeled.vo]', [labeled.vr]' );
cdf.plot.landmark( run, detected, labeled, 'landmark.png' ); % plot tests
cdf.plot.timing( run, detected, labeled, 'timing.png' );

... % whatever needs to be done with detected data
```

3 Internals

- TODO: `dsp`, `sta`, `k15`, `xis`

4 Algorithms

- TODO: voice activity
- TODO: denoising
- TODO: burst/glottis detection

4.1 Essentials

Time series/Multi-dimensional arrays

- a time series is a N-tuple $x = (x_0, \dots, x_{N-1})$ where $x_i \in X$
- let its components be real-valued, d-dimensional arrays $X = \mathbb{R}^{n_1 \times \cdots \times n_d}$
- some specific arrays are scalars $\mathbb{R}$, vectors $\mathbb{R}^n$ and matrices $\mathbb{R}^{n \times m}$
- use shorthand notation $\mathbb{R}^{(d)} = \mathbb{R}^{n_1 \times \cdots \times n_d}$ if size of dimensions is not important
- time series itself are multi-dimensional arrays: $x \in \mathbb{R}^{N \times n_1 \times \cdots \times n_d} = \mathbb{R}^{(d+1)}$

Basic operations

- operations reducing dimensionality (like mean, min, etc.) act on first dimension, e.g.

$$\text{mean} : \mathbb{R}^{N \times n_1 \times \cdots} \to \mathbb{R}^{n_1 \times \cdots}, (x_0, \dots, x_{N-1}) \mapsto \frac{1}{N} \sum_{i=0}^{N-1} x_i$$

- successive application might reduce dimensionality down to scalars

- other unary operations act component-by-component and recursively, e.g.

$$\log : \mathbb{R}^{N\times\cdots} \to \mathbb{R}^{N\times\cdots}, (x_0, \ldots, x_{N-1}) \mapsto (\log x_0, \ldots \log x_{N-1})$$

- basic arithmetic operations also act element-wise
- in general: mutli-dimensional arrays form a vector space and therefor inherit properties of their underlying field X as usual

Short-time framing

- let $x = (x_0, \ldots, x_{N-1})$ be a time series with $x \in X^{(d)}$ and $x_i \in X^{(d-1)}$
- denote ${}^{L}_{S}y = (y_0, \ldots, y_{M-1})$ as a series of sliding frames with length L and stride S, where
  - individual frames are given by $y_i = (x_{iS}, \ldots, x_{iS+L-1})$
  - x is zero padded appropriately, means $\forall i \geq N : x_i = 0$
  - the number of frames is given by $M = \lceil N/S \rceil$ (ceiling)
- $y \in X^{(d+1)}$ itself is a multi-dimensional array of short-time frames
- apodization with window $w \in \mathbb{R}^L$ is done by ordinary scalar multiplication

$$y^w = (y_0^w, \ldots, y_{M-1}^w), \quad y_i^w = (w_0 y_{i,0}, \ldots, w_{L-1} y_{i,L-1})$$

Short-time fourier tranform/Spectrogram

- given a time series $x \in X^{(d)}$ with short-time representation ${}^{L}_{S}y = (y_0, \ldots, y_{M-1}) \in X^{(d+1)}$
- denote ${}^{L}_{S}\tilde{x} = (\tilde{x}_0 \ldots, \tilde{x}_{M-1}) \in X^{(d+1)}$ as the short-time fourier transform/spectrogram of x, where
  - spectral frames $\tilde{x}_i$ are given by the fourier transform of temporal frames y_i:

$$\tilde{x}_i = (\tilde{x}_{i,0}, \ldots, \tilde{x}_{i,K-1}), \quad \tilde{x}_{i,j} = \sum_{k=0}^{K-1} y_{i,k} e^{-2\pi \mathrm{i} \frac{jk}{K}}$$

  - K is chosen to be the next power of two greater or equal than L, temporal frames are zero padded accordingly $\forall k \geq L : y_{i,k} = 0$
- exploiting the symmetry for real-valued times series $\tilde{x}_{i,j} = \overline{\tilde{x}_{i,K-j}}$ leads to one-sided spectrogram
- spectral powers are given by $(\tilde{x}/K)^2$, TODO: shouldn't it be $(\tilde{x}/\sqrt{K})^2$?

Short-time unframing/Smoothing

- let ${}^{L}_{S}y = (y_0, \ldots, y_{M-1}) \in X^{(d)}$ be an arbitrary short-time frame series
- denote ${}^{L}_{S}\hat{y} = (\hat{y}_0, \ldots, \hat{y}_{N-1}) \in X^{(d)}$ as the unframed and smoothed version of y
- TODO...