signalanalysis

Release 0.1

Philip Gemmell

Oct 08, 2021

CONTENTS:

signalanalysis is a library including various tools for the reading, analysis and plotting of ECG and VCG data. It is designed to be as agnostic as possible for the types of data that it can read. Currently, it can read ECG data from:

1. CARP simulations of whole torso activity, using existing projects from *CARPutil <https://git.opencarp.org/openCARP/carputils>*;
2. .csv and .dat records;
3. wfdb file formats, using *wfdb-python <https://github.com/MIT-LCP/wfdb-python>*

Futher details of how to use these functions are in *Usage*, with the finer points within the files themselves.

CHAPTER

ONE

USAGE

1.1 Installation & Getting Started

To use signalanalysis, it is highly recommended to install using `pipenv`, the virtual environment, to ensure that dependencies are where possible maintained. Clone the repository, then install the requirements as follows:

```
user@home:~$ git clone git@github.com:philip-gemmell/signalanalysis.git
user@home:~$ cd signalanalysis
user@home:~/signalanalysis$ pipenv install
```

Once the repository is cloned, it is currently the case that all work must be done within the Python3 environment. However, it is recommended to use the virtual environment from pipenv rather than the system-wide Python3 (after entering a pipenv shell, it is quit using the `exit` command as shown)

```
user@home:~/signalanalysis$ pipenv shell
(signalanalysis) user@home:~/signalanalysis$ python3
>>> import signalanalysis
>>> quit()
(signalanalysis) user@home:~/signalanalysis$ exit
user@home:~/signalanalysis$
```

The project is arranged into various subdivisions. The required analysis/plotting packages for the ECG/VCG are separated out, and require separate importing. See the next section, and individual help files for further details.

- signalanalysis
  - signalanalysis.general
  - signalanalysis.ecg
  - signalanalysis.vcg
- signalplot
- tools

1.2 Reading ECG/VCG data

Currently, only ECG reading is supported; VCG data is calculated from ECG data using the Kors method (see method). ECG files are read upon the instantiation of the ECG class, though it is possible to re-read data if required for some reason.

```
>>> import signalanalysis.ecg
>>> import signalanalysis.vcg
>>> ecg_example = signalanalysis.ecg.Ecg("filename")
>>> vcg_example = signalanalysis.vcg.Vcg(ecg_example)
```

1.3 Shifting to classes from methods

Previously, all ECG/VCG data was extracted and stored in DataFrames, and most of the modules in this code currently support this format. However, it is planned to shift the main focus of the project to use classes, which allow encapsulation of linked data in one data structure. While the focus of this documentation will be future-facing, and look at using the classes, note that sometimes access to the raw, underlying DataFrames will still be required. To that end, the raw data can be accessed as the .data attribute:

```
>>> ecg_class = signalanalysis.ecg.Ecg("filename") # Returns an Ecg class
>>> vcg_class = signalanalysis.vcg.Vcg(ecg_class)  # Returns a Vcg class
>>> ecg_data = ecg_class.data                      # Returns a Pandas DataFrame of the␣
↪underlying data
>>> vcg_data = vcg_class.data                      # Returns a Pandas DataFrame of the␣
↪underlying data
```

CHAPTER

TWO

DOCUTILS DOCUMENTATION

This is the index of all modules and their associated methods and classes, documenting their use, parameters and return values. See *Usage* for a more step-by-step introduction to the intended use cases.

Broadly speaking, the modules are split thus:

- `signalanalysis` covers the analysis scripts for ECG/VCG analysis (e.g. calculating QRS duration)
- `signalplot` covers plotting methods (e.g. plotting the ECG leads on a single figure with annotation, plotting a 3D plot of VCG (including animation!))
- `tools` covers more general use tools that are not limited to ECG/VCG analysis.

signalanalysis
signalplot
tools

2.1 signalanalysis

signalanalysis.ecg
signalanalysis.general
signalanalysis.vcg

2.1.1 signalanalysis.ecg

Functions

get_ecg_from_electrodes	Converts electrode phi_e data to ECG lead data
get_electrode_phie	Extract phi_e data corresponding to ECG electrode locations
get_qrs_start	Calculates start of QRS complex using method of Hermans et al. (2017).

continues on next page

Table 3 – continued from previous page

read_ecg_from_csv	Extract ECG data from CSV file exported from St Jude Medical ECG recording
read_ecg_from_dat	Read ECG data from .dat file
read_ecg_from_igb	Translate the phie.igb file(s) to 10-lead, 12-trace ECG data

signalanalysis.ecg.get_ecg_from_electrodes

signalanalysis.ecg.**get_ecg_from_electrodes**(*electrode_data: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Converts electrode phi_e data to ECG lead data

Takes dictionary of phi_e data for 10-lead ECG, and converts these data to standard ECG trace data

Parameters **electrode_data** (*pd.DataFrame*) – Dictionary with keys corresponding to lead locations

Returns **ecg** – Dictionary with keys corresponding to the ECG traces

Return type pd.DataFrame

signalanalysis.ecg.get_electrode_phie

signalanalysis.ecg.**get_electrode_phie**(*phie_data: numpy.ndarray*, *electrode_file: Optional[str] = None*) → pandas.core.frame.DataFrame

Extract phi_e data corresponding to ECG electrode locations

Parameters

- **phie_data** (*np.ndarray*) – Numpy array that holds all phie data for all nodes in a given mesh
- **electrode_file** (*str, optional*) – File containing entries corresponding to the nodes of the mesh which determine the location of the 10 leads for the ECG. Will default to very project specific location. The input text file has each node on a separate line (zero-indexed), with the node locations given in order: V1, V2, V3, V4, V5, V6, RA, LA, RL, LL. Will default to '12LeadElectrodes.dat', but this is almost certainly not going to right for an individual project

Returns **electrode_data** – Dataframe of phie data for each node, with the dictionary key labelling which node it is.

Return type pd.DataFrame

signalanalysis.ecg.get_qrs_start

signalanalysis.ecg.**get_qrs_start**(*ecgs: Union[pandas.core.frame.DataFrame, List[pandas.core.frame.DataFrame]]*, *unipolar_only: bool = True*, *plot_result: bool = False*) → List[float]

Calculates start of QRS complex using method of Hermans et al. (2017)

Calculates the start of the QRS complex by a simplified version of the work presented in[1], wherein the point of maximum second derivative of the ECG RMS signal is used as the start of the QRS complex

[1] Hermans BJM, Vink AS, Bennis FC, Filippini LH, Meijborg VMF, Wilde AAM, Pison L, Postema PG, Delhaas T, "The development and validation of an easy to use automatic QT-interval algorithm," PLoS ONE, 12(9), 1–14 (2017), https://doi.org/10.1371/journal.pone.0184352

Parameters

- **ecgs** (*pd.DataFrame or list of pd.DataFrame*) – ECG data to analyse
- **unipolar_only** (*bool, optional*) – Whether to use only unipolar leads to calculate RMS, default=True
- **plot_result** (*bool, optional*) – Whether to plot the results for error-checking, default=False

Returns **qrs_starts** – QRS start times

Return type list of float

Notes

For further details of the action of unipolar_only, see general_analysis.get_signal_rms

It is faster to use scipy.ndimage.laplace() rather than np.gradient(np.gradient)), but preliminary checks indicated some edge problems that might throw off the results.

References

signalanalysis.ecg.read_ecg_from_csv

`signalanalysis.ecg.`**`read_ecg_from_csv`**(*filename: str*, *normalise: bool = False*) → pandas.core.frame.DataFrame

Extract ECG data from CSV file exported from St Jude Medical ECG recording

Parameters

- **filename** (*str*) – Name/location of the .dat file to read
- **normalise** (*bool, optional*) – Whether or not to normalise the ECG signals on a per-lead basis, default=True

Returns **ecg** – Extracted data for the 12-lead ECG

Return type list of pd.DataFrame

signalanalysis.ecg.read_ecg_from_dat

`signalanalysis.ecg.`**`read_ecg_from_dat`**(*filename: str*, *normalise: bool = False*) → pandas.core.frame.DataFrame

Read ECG data from .dat file

Parameters

- **filename** (*str*) – Name/location of the .dat file to read
- **normalise** (*bool, optional*) – Whether or not to normalise the ECG signals on a per-lead basis, default=False

Returns **ecg** – Extracted data for the 12-lead ECG

Return type pd.DataFrame

signalanalysis.ecg.read_ecg_from_igb

`signalanalysis.ecg.read_ecg_from_igb`(*filename: str*, *electrode_file: Optional[str] = None*, *normalise: bool = False*, *dt: float = 0.002*) → pandas.core.frame.DataFrame

Translate the phie.igb file(s) to 10-lead, 12-trace ECG data

Extracts the complete mesh data from the phie.igb file using CARPutils, which contains the data for the body surface potential for an entire human torso, before then extracting only those nodes that are relevant to the 12-lead ECG, before converting to the ECG itself https://carpentry.medunigraz.at/carputils/generated/carputils.carpio.igb.IGBFile.html#carputils.carpio.igb.IGBFile

Parameters

- **filename** (*str*) – Filename for the phie.igb data to extract
- **electrode_file** (*str, optional*) – File which contains the node indices in the mesh that correspond to the placement of the leads for the 10-lead ECG. Default given in get_electrode_phie function.
- **normalise** (*bool, optional*) – Whether or not to normalise the ECG signals on a per-lead basis, default=False
- **dt** (*float, optional*) – Time interval from which to construct the time data to associate with the ECG, default=0.002s (2ms)

Returns **ecgs** – DataFrame with Vm data for each of the labelled leads (the dictionary keys are the names of the leads)

Return type pd.DataFrame

Classes

Ecg	Base ECG class to encapsulate data regarding an ECG recording, inheriting from signalanalysis.general.Signal

signalanalysis.ecg.Ecg

class `signalanalysis.ecg.Ecg`(*filename: str*, ***kwargs*)

Bases: *signalanalysis.general.Signal*

Base ECG class to encapsulate data regarding an ECG recording, inheriting from signalanalysis.general.Signal

data

Raw ECG data for the different leads

Type pd.DataFrame

filename

Filename for the location of the data

Type str

normalised

Whether or not the data for the leads have been normalised

Type bool

n_beats

Number of beats recorded in the trace. Set to 0 if not calculated

Type int

`qrs_start`
Times calculated for the start of the QRS complex

Type list of float

`qrs_end`
Times calculated for the end of the QRS complex

Type end

`data_source`
Source for the data, if known e.g. Staff III database, CARP simulation, etc.

Type str

`comments`
Any further details known about the data, e.g. sex, age, etc.

Type str

`read`(*filename*)
Reads in the data from the original file. Called upon initialisation

`read_ecg_from_wfdb`(*filename*, *normalise=False*)
Reads data from a WFDB type series of files, e.g. from the Lobachevsky ECG database (https://physionet.org/content/ludb/1.0.1/)

`get_n_beats`(*threshold=0.5*, *min_separation=0.2*)
Calculates the number of beats given in the recording

`get_qrs_start`()
Calculates the start of the QRS complex

Methods

get_n_beats	Calculate the number of beats in an ECG trace, and save the individual beats to file for later use
get_qrs_start	Calculates start of QRS complex using method of Hermans et al. (2017).
get_rms	Returns the RMS of the combined signal
read	
read_ecg_from_wfdb	
reset	Reset all properties of the class

`get_n_beats`(*threshold: float = 0.5*, *min_separation: float = 0.2*, *unipolar_only: bool = True*, *plot: bool = False*)
Calculate the number of beats in an ECG trace, and save the individual beats to file for later use

When given the raw data of an ECG trace, will estimate the number of beats recorded in the trace based on the RMS of the ECG signal exceeding a threshold value. The estimated individual beats will then be saved in a list in a lossless manner, i.e. saved as [ECG1, ECG2, ..., ECG(n)], where ECG1=[0:peak2], ECG2=[peak1:peak3], ..., ECGn=[peak(n-1):end]

threshold [float {0<1}] Minimum value to search for for a peak in RMS signal to determine when a beat has occurred, default=0.5

min_separation [float] Minimum time (in s) that should be used to separate separate beats, default=0.2s

unipolar_only [bool, optional] Whether to use only unipolar ECG leads to calculate RMS, default=True

plot [bool] Whether to plot results of beat detection, default=False

self.n_beats [int] Number of beats detected in signal

The scalar RMS is calculated according to

rac{1}{n}sum_{i=1}^n (extnormal{ECG}_i^2(t))}

for all leads available from the signal (12 for ECG, 3 for VCG). If unipolar_only is set to true, then ECG RMS is calculated using only 'unipolar' leads. This uses V1-6, and the non-augmented limb leads (VF, VL and VR)

..math:: VF = LL-V_{WCT} =

rac{2}{3}aVF ..math:: VL = LA-V_{WCT} =

rac{2}{3}aVL ..math:: VR = RA-V_{WCT} =

rac{2}{3}aVR

`get_qrs_start`(*unipolar_only: bool = True*, *min_separation: float = 0.05*, *plot_result: bool = False*)
Calculates start of QRS complex using method of Hermans et al. (2017)

Calculates the start of the QRS complex by a simplified version of the work presented in[1], wherein the point of maximum second derivative of the ECG RMS signal is used as the start of the QRS complex

Parameters
- `self` (`Ecg`) – ECG data to analyse
- `unipolar_only` (*bool, optional*) – Whether to use only unipolar leads to calculate RMS, default=True
- `min_separation` (*float, optional*) – Minimum separation from the peak used to detect various beats, default=0.05s
- `plot_result` (*bool, optional*) – Whether to plot the results for error-checking, default=False

Returns **self.qrs_start** – QRS start times

Return type list of float

Notes

For further details of the action of unipolar_only, see general_analysis.get_signal_rms

It is faster to use scipy.ndimage.laplace() rather than np.gradient(np.gradient)), but preliminary checks indicated some edge problems that might throw off the results.

[1] Hermans BJM, Vink AS, Bennis FC, Filippini LH, Meijborg VMF, Wilde AAM, Pison L, Postema PG, Delhaas T, "The development and validation of an easy to use automatic QT-interval algorithm," PLoS ONE, 12(9), 1–14 (2017), https://doi.org/10.1371/journal.pone.0184352

References

get_rms(*preprocess_data: Optional[pandas.core.frame.DataFrame] = None*, *drop_columns: Optional[List[str]] = None*, *unipolar_only: bool = True*)

Returns the RMS of the combined signal

Parameters

- **preprocess_data** (`pd.DataFrame, optional`) – Only passed if there is some extant data that is to be used for getting the RMS (for example, if the unipolar data only from ECG is being used, and the data is thus preprocessed in a manner specific for ECG data in the ECG routine)
- **drop_columns** (`list of str, optional`) – List of any columns to drop from the raw data before calculating the RMS. Can be used in conjunction with preprocess_data
- **unipolar_only** (*#*) –
- **RMS** (`# Whether to use only unipolar ECG leads to calculate`) –
- **default=True** –

reset()

Reset all properties of the class

Function called when reading in new data into an existing class (for some reason), which would make these properties and attributes clash with the other data

2.1.2 signalanalysis.general

Functions

get_signal_rms	Calculate the ECG(RMS) of the ECG as a scalar
get_twave_end	Return the time point at which it is estimated that the T-wave has been completed

signalanalysis.general.get_signal_rms

signalanalysis.general.**get_signal_rms**(*signal: pandas.core.frame.DataFrame*, *unipolar_only: bool = True*) → List[float]

Calculate the ECG(RMS) of the ECG as a scalar

signal: pd.DataFrame ECG or VCG data to process

unipolar_only [bool, optional] Whether to use only unipolar ECG leads to calculate RMS, default=True

signal_rms [list of float] Scalar RMS ECG or VCG data

The scalar RMS is calculated according to

rac{1}{n}sum_{i=1}^n (extnormal{ECG}_i^2(t))}

for all leads available from the signal (12 for ECG, 3 for VCG). If unipolar_only is set to true, then ECG RMS is calculated using only 'unipolar' leads. This uses V1-6, and the non-augmented limb leads (VF, VL and VR)

..math:: VF = LL-V_{WCT} =

rac{2}{3}aVF ..math:: VL = LA-V_{WCT} =

rac{2}{3}aVL ..math:: VR = RA-V_{WCT} =

rac{2}{3}aVR

The development and validation of an easy to use automatic QT-interval algorithm Hermans BJM, Vink AS, Bennis FC, Filippini LH, Meijborg VMF, Wilde AAM, Pison L, Postema PG, Delhaas T PLoS ONE, 12(9), 1–14 (2017) https://doi.org/10.1371/journal.pone.0184352

signalanalysis.general.get_twave_end

`signalanalysis.general.get_twave_end`(*ecgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*, *leads: Union[str, List[str]] = 'LII'*, *i_distance: int = 200*, *filter_signal: Optional[str] = None*, *baseline_adjust: Optional[Union[float, List[float]]] = None*, *return_median: bool = True*, *remove_outliers: bool = True*, *plot_result: bool = False*) → List[pandas.core.frame.DataFrame]

Return the time point at which it is estimated that the T-wave has been completed

Parameters

- **ecgs** (*pd.DataFrame or list of pd.DataFrame*) – Signal data, either ECG or VCG
- **leads** (*str, optional*) – Which lead to check for the T-wave - usually this is either 'LII' or 'V5', but can be set to a list of various leads. If set to 'global', then all T-wave values will be calculated. Will return all values unless return_median flag is set. Default 'LII'
- **i_distance** (*int, optional*) – Distance between peaks in the gradient, i.e. will direct that the function will only find the points of maximum gradient (representing T-wave, etc.) with a minimum distance given here (in terms of indices, rather than time). Helps prevent being overly sensitive to 'wobbles' in the ecg. Default=200
- **filter_signal** (*{'butterworth', 'savitzky-golay'}, optional*) – Whether or not to apply a filter to the data prior to trying to find the actual T-wave gradient. Can pass either a Butterworth filter or a Savitzky-Golay filter, in which case the required kwargs for each can be provided. Default=None (no filter applied)
- **baseline_adjust** (*float or list of float, optional*) – Point from which to calculate the adjusted baseline for calculating the T-wave, rather than using the zeroline. In line with Hermans et al., this is usually the start of the QRS complex, with the baseline calculated as the median amplitude of the 30ms before this point.
- **return_median** (*bool, optional*) – Whether or not to return an average of the leads requested, default=True
- **remove_outliers** (*bool, optional*) – Whether to remove T-wave end values that are greater than 1 standard deviation from the mean from the data. Only has an effect if more than one lead is provided, and return_average is True. Default=True
- **plot_result** (*bool, optional*) – Whether to plot the results or not, default=False

Returns **twave_ends** – Time value for when T-wave is estimated to have ended.

Return type list of pd.DataFrame

Notes

Calculates the end of the T-wave as the time at which the T-wave's maximum gradient tangent returns to the baseline. The baseline is either set to zero, or set to the median value of 30ms prior to the start of the QRS complex (the value of which has to be passed in the *baseline_adjust* variable).

References

Classes

`Signal`	Base class for general signal, either ECG or VCG

signalanalysis.general.Signal

class `signalanalysis.general.Signal(**kwargs)`

Bases: `object`

Base class for general signal, either ECG or VCG

data
Raw ECG data for the different leads
Type pd.DataFrame

filename
Filename for the location of the data
Type str

normalised
Whether or not the data for the leads have been normalised
Type bool

n_beats
Number of beats recorded in the trace. Set to 0 if not calculated
Type int

qrs_start
Times calculated for the start of the QRS complex
Type list of float

qrs_end
Times calculated for the end of the QRS complex
Type end

data_source
Source for the data, if known e.g. Staff III database, CARP simulation, etc.
Type str

comments
Any further details known about the data, e.g. sex, age, etc.
Type str

get_rms(*unipolar_only=True*)
Returns the RMS of the combined signal

Methods

get_n_beats	Calculate the number of beats in an ECG trace, and save the individual beats to file for later use
get_rms	Returns the RMS of the combined signal
reset	Reset all properties of the class

get_n_beats(*threshold: float = 0.5*, *min_separation: float = 0.2*, *unipolar_only: bool = True*, *plot: bool = False*)

Calculate the number of beats in an ECG trace, and save the individual beats to file for later use

When given the raw data of an ECG trace, will estimate the number of beats recorded in the trace based on the RMS of the ECG signal exceeding a threshold value. The estimated individual beats will then be saved in a list in a lossless manner, i.e. saved as [ECG1, ECG2, . . . , ECG(n)], where ECG1=[0:peak2], ECG2=[peak1:peak3], . . . , ECGn=[peak(n-1):end]

threshold [float {0<1}] Minimum value to search for for a peak in RMS signal to determine when a beat has occurred, default=0.5

min_separation [float] Minimum time (in s) that should be used to separate separate beats, default=0.2s

unipolar_only [bool, optional] Whether to use only unipolar ECG leads to calculate RMS, default=True

plot [bool] Whether to plot results of beat detection, default=False

self.n_beats [int] Number of beats detected in signal

The scalar RMS is calculated according to

rac{1}{n}sum_{i=1}^n (extnormal{ECG}_i^2(t))}

for all leads available from the signal (12 for ECG, 3 for VCG). If unipolar_only is set to true, then ECG RMS is calculated using only 'unipolar' leads. This uses V1-6, and the non-augmented limb leads (VF, VL and VR)

..math:: VF = LL-V_{WCT} =

rac{2}{3}aVF ..math:: VL = LA-V_{WCT} =

rac{2}{3}aVL ..math:: VR = RA-V_{WCT} =

rac{2}{3}aVR

get_rms(*preprocess_data: Optional[pandas.core.frame.DataFrame] = None*, *drop_columns: Optional[List[str]] = None*)

Returns the RMS of the combined signal

Parameters

- **preprocess_data** (*pd.DataFrame, optional*) – Only passed if there is some extant data that is to be used for getting the RMS (for example, if the unipolar data only from ECG is being used, and the data is thus preprocessed in a manner specific for ECG data in the ECG routine)
- **drop_columns** (*list of str, optional*) – List of any columns to drop from the raw data before calculating the RMS. Can be used in conjunction with preprocess_data
- **unipolar_only** (*#*) –
- **RMS** (*# Whether to use only unipolar ECG leads to calculate*) –
- **default=True** –

reset()

Reset all properties of the class

Function called when reading in new data into an existing class (for some reason), which would make these properties and attributes clash with the other data

2.1.3 signalanalysis.vcg

Functions

calculate_delta_dipole_angle	Calculates the angular difference between two VCGs based on difference in azimuthal and elevation angles.
compare_dipole_angles	Calculates the angular differences between two VCGs at multiple points during their evolution
get_azimuth_elevation	Calculate azimuth and elevation angles for a specified section of the VCG.
get_dipole_magnitudes	Calculates metrics relating to the magnitude of the weighted dipole of the VCG
get_qrs_start_end	Calculate the extent of the VCG QRS complex on the basis of max derivative
get_single_vcg_azimuth_elevation	Get the azimuth and elevation data for a single VCG trace, along with the average dipole magnitude.
get_spatial_velocity	Calculate spatial velocity
get_vcg_area	Calculate area under VCG curve for a given section (e.g.
get_vcg_from_ecg	Convert ECG data to vectorcardiogram (VCG) data using the Kors matrix method
get_weighted_dipole_angles	Calculate metrics relating to the angles of the weighted dipole of the VCG.
plot_density_effect	Plot the effect of density on metrics.
plot_metric_change	Function to plot all the various figures for trend analysis in one go.
plot_metric_change_barplot	Plots a bar chart for the observed metrics.

signalanalysis.vcg.calculate_delta_dipole_angle

signalanalysis.vcg.**calculate_delta_dipole_angle**(*azimuth1: List[float]*, *elevation1: List[float]*, *azimuth2: List[float]*, *elevation2: List[float]*, *convert_to_degrees: bool = False*) → List[float]

Calculates the angular difference between two VCGs based on difference in azimuthal and elevation angles.

Useful for calculating difference between weighted averages.

Parameters

- **azimuth1** (*list of float*) – Azimuth angles for the first dipole
- **elevation1** (*list of float*) – Elevation angles for the first dipole
- **azimuth2** (*list of float*) – Azimuth angles for the second dipole
- **elevation2** (*list of float*) – Elevation angles for the second dipole

- **convert_to_degrees** (*bool, optional*) – Whether to convert the angle from radians to degrees, default=False

Returns dt – List of angles between a series of dipoles, either in radians (default) or degrees depending on input argument

Return type list of float

signalanalysis.vcg.compare_dipole_angles

`signalanalysis.vcg.compare_dipole_angles`(*vcg1: pandas.core.frame.DataFrame*, *vcg2: pandas.core.frame.DataFrame*, *t_start1: float = 0*, *t_end1: Optional[float] = None*, *t_start2: float = 0*, *t_end2: Optional[float] = None*, *n_compare: int = 10*, *convert_to_degrees: bool = False*, *matlab_match: bool = False*) → List[float]

Calculates the angular differences between two VCGs at multiple points during their evolution

To compensate for the fact that the two VCG traces may not be of the same length, the comparison does not occur at every moment of the VCG; rather, the dipoles are calculated for certain fractional points during the VCG.

Parameters

- **vcg1** (*pd.DataFrame*) – First VCG trace to consider
- **vcg2** (*pd.DataFrame*) – Second VCG trace to consider
- **t_start1** (*float, optional*) – Time from which to consider the data from the first VCG trace, default=0
- **t_end1** (*float, optional*) – Time until which to consider the data from the first VCG trace, default=end
- **t_start2** (*float, optional*) – Time from which to consider the data from the second VCG trace, default=0
- **t_end2** (*float, optional*) – Time until which to consider the data from the second VCG trace, default=end
- **n_compare** (*int, optional*) – Number of points during the VCGs at which to calculate the dipole angle. If set to -1, will calculate at every point during the VCG, but requires VCG traces to be the same length, default=10
- **convert_to_degrees** (*bool, optional*) – Whether to convert the angles from radians to degrees, default=False
- **matlab_match** (*bool, optional*) – Whether to extract the data segment to match Matlab output or to use simpler Python, default=False

Returns dt – Angle between two given VCGs at n points during the VCG, where n is given as input

Return type list of float

signalanalysis.vcg.get_azimuth_elevation

signalanalysis.vcg.**get_azimuth_elevation**(*vcgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*, *t_start: Optional[List[float]] = None*, *t_end: Optional[List[float]] = None*) → Tuple[List[Iterable[float]], List[Iterable[float]]]

Calculate azimuth and elevation angles for a specified section of the VCG.

Will calculate the azimuth and elevation angles for the VCG at each recorded point, potentially within specified limits (e.g. start/end of QRS)

Parameters

- **vcgs** (*pd.DataFrame or list of pd.DataFrame*) – VCG data to calculate
- **t_start** (*list of float, optional*) – Start time from which to calculate the angles, default=0
- **t_end** (*list of float, optional*) – End time until which to calculate the angles, default=end

Returns

- **azimuth** (*list of list of float*) – List (one entry for each passed VCG) of azimuth angles (in radians) for the dipole for every time point during the specified range
- **elevation** (*list of list of float*) – List (one entry for each passed VCG) of elevation angles (in radians) for the dipole for every time point during the specified range

signalanalysis.vcg.get_dipole_magnitudes

signalanalysis.vcg.**get_dipole_magnitudes**(*vcgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*, *t_start: Union[float, List[float]] = 0*, *t_end: Union[float, List[float]] = - 1*) → Tuple[List[numpy.ndarray], List[float], List[float], List[List[float]], List]

Calculates metrics relating to the magnitude of the weighted dipole of the VCG

Returns the mean weighted dipole, maximum dipole magnitude,(x,y.z) components of the maximum dipole and the time at which the maximum dipole occurs

Parameters

- **vcgs** (*pd.DataFrame or list of pd.DataFrame*) – VCG data to calculate
- **t_start** (*list of float, optional*) – Start time from which to calculate the magnitude, default=0 (for any other value to be recognisable, time variable must be given)
- **t_end** (*list of float, optional*) – End time until which to calculate the magnitudes, default=end (for any other value to be recognisable, time variable must be given)

Returns

- **dipole_magnitude** (*list of np.ndarray*) – Magnitude time courses for each VCG

- **weighted_magnitude** (*list of float*) – Mean magnitude of the VCG
- **max_dipole_magnitude** (*list of float*) – Maximum magnitude of the VCG
- **max_dipole_components** (*list of list of float*) – x, y, z components of the dipole at is maximum value
- **max_dipole_time** (*list of float*) – Time at which the maximum magnitude of the VCG occurs

signalanalysis.vcg.get_qrs_start_end

`signalanalysis.vcg.`**`get_qrs_start_end`**(*vcgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*, *velocity_offset: int = 2*, *low_p: float = 40*, *order: int = 2*, *threshold_frac_start: float = 0.22*, *threshold_frac_end: float = 0.54*, *filter_sv: bool = True*, *qrs_window: float = 180*, *ecgs: Optional[Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]] = None*) → Tuple[List[float], List[float], List[float]]

Calculate the extent of the VCG QRS complex on the basis of max derivative

TODO: Check whether i_qrs_start variable is needed, or can be simplified using DataFrame function

Calculate the start and end points, and hence duration, of the QRS complex of a list of VCGs. It does this by finding the time at which the spatial velocity of the VCG exceeds a threshold value (the start time), then searches backwards from the end of the VCG to find when this threshold is exceeded (the end time); the start and end thresholds do not necessarily have to be the same.

Parameters

- **vcgs** (*list of pd.DataFrame or pd.DataFrame*) – List of VCG data to get QRS start and end points for
- **velocity_offset** (*int, optional*) – Offset between values in VCG over which to calculate spatial velocity, i.e. 1 will use neighbouring values to calculate the gradient/velocity. Default=2
- **low_p** (*float, optional*) – Low frequency for bandpass filter, default=40
- **order** (*int, optional*) – Order for Butterworth filter, default=2
- **threshold_frac_start** (*float, optional*) – Fraction of maximum spatial velocity to trigger start of QRS detection, default=0.15
- **threshold_frac_end** (*float, optional*) – Fraction of maximum spatial velocity to trigger end of QRS detection, default=0.15
- **filter_sv** (*bool, optional*) – Whether or not to apply filtering to spatial velocity prior to finding the start/end points for the threshold
- **qrs_window** (*float, optional*) – Default size of 'window' in which to search for end of QRS complex, default=180ms
- **ecgs** (*list of pd.DataFrame or pd.DataFrame, optional*) – ECG data associated with VCG data. Only used if having trouble establishing QRS start, in which case will be used to plot ECG data to allow user to determine whether or not the QRS is occurring at the start of the simulation, or whether there is a more deep-seated issue with the data.

Returns

- **qrs_start** (*list of float*) – List of start time of QRS complexes of provided VCGs
- **qrs_end** (*list of float*) – List of end time of QRS complex of provided VCGs
- **qrs_duration** (*list of float*) – List of duration of QRS complex of provided VCGs

signalanalysis.vcg.get_single_vcg_azimuth_elevation

`signalanalysis.vcg.`**`get_single_vcg_azimuth_elevation`**(*vcg: pandas.core.frame.DataFrame*, *t_start: float*, *t_end: float*, *weighted: bool = True*) → Tuple[List[float], List[float], numpy.ndarray]

Get the azimuth and elevation data for a single VCG trace, along with the average dipole magnitude.

Returns the azimuth and elevation angles for a single given VCG trace. Can analyse only a segment of the VCG if required, and can weight the angles according to the dipole magnitude. Primarily designed as a helper function for get_azimuth_elevation and get_weighted_dipole_angles.

Parameters

- **vcg** (*pd.DataFrame*) – VCG data to calculate
- **t_start** (*float*) – Start time from which to calculate the angles
- **t_end** (*float*) – End time until which to calculate the angles
- **weighted** (*bool, optional*) – Whether or not to weight the returned angles by the magnitude of the dipole at the same moment, default=True

Returns

- **theta** (*list of float*) – List of the azimuth angles for the VCG dipole, potentially weighted according to the dipole magnitude at the associated time
- **phi** (*list of float*) – List of the elevation above xy-plane angles for the VCG dipole, potentially weighted according to the dipole magnitude at the associated time
- **dipole_magnitude** (*np.ndarray*) – Array containing the dipole magnitude at all points throughout the VCG

signalanalysis.vcg.get_spatial_velocity

`signalanalysis.vcg.`**`get_spatial_velocity`**(*vcgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*, *velocity_offset: int = 2*, *filter_sv: bool = True*, *low_p: float = 40*, *order: int = 2*) → List[pandas.core.frame.DataFrame]

Calculate spatial velocity

Calculate the spatial velocity of a VCG, in terms of calculating the gradient of the VCG in each of its x, y and z components, before combining these components in a Euclidian norm. Will then find the point at which the spatial velocity exceeds a threshold value, and the point at which it declines below another threshold value.

Parameters

- **vcgs** (*list of pd.DataFrame or pd.DataFrame*) – VCG data to analyse
- **velocity_offset** (*int, optional*) – Offset between values in VCG over which to calculate spatial velocity, i.e. 1 will use neighbouring values to calculate the gradient/velocity. Default=2
- **filter_sv** (*bool, optional*) – Whether or not to apply filtering to spatial velocity, default=True
- **low_p** (*float, optional*) – Low frequency for bandpass filter, default=40
- **order** (*int, optional*) – Order for Butterworth filter, default=2

Returns **sv** – Spatial velocity data, filtered according to input parameters

Return type list of pd.DataFrame

Notes

Calculation of spatial velocity based on[1],[2],[3]

References

signalanalysis.vcg.get_vcg_area

signalanalysis.vcg.**get_vcg_area**(*vcgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*, *limits_start: Optional[List[float]] = None*, *limits_end: Optional[List[float]] = None*, *method: str = 'pythag'*, *matlab_match: bool = False*) → List[float]

Calculate area under VCG curve for a given section (e.g. QRS complex).

Calculate the area under the VCG between two intervals (usually QRS start and QRS end). This is calculated in two ways: a 'Pythagorean' method, wherein the area under each of the VCG(x), VCG(y) and VCG(z) curves are calculated, then combined in a Euclidean norm, or a '3D' method, wherein the area of the arc traced in 3D space between successive timepoints is calculated, then summed.

Parameters

- **vcgs** (*pd.DataFrame or list of pd.DataFrame*) – VCG data from which to get area
- **limits_start** (*list of float, optional*) – Start times (NOT INDICES) for where to calculate area under curve from, default=0
- **limits_end** (*list of float, optional*) – End times (NOT INDICES) for where to calculate are under curve until, default=end
- **method** (*{'pythag', '3d'}, optional*) – Which method to use to calculate the area under the VCG curve, default='pythag'

[1] Kors JA, van Herpen G, "Methodology of QT-interval measurement in the modular ECG analysis system (MEANS)" Ann Noninvasive Electrocardiol. 2009 Jan;14 Suppl 1:S48-53. doi: 10.1111/j.1542-474X.2008.00261.x.

[2] Xue JQ, "Robust QT Interval Estimation—From Algorithm to Validation" Ann Noninvasive Electrocardiol. 2009 Jan;14 Suppl 1:S35-41. doi: 10.1111/j.1542-474X.2008.00264.x.

[3] Sörnmo L, "A model-based approach to QRS delineation" Comput Biomed Res. 1987 Dec;20(6):526-42.

- **matlab_match** (*bool, optional*) – Whether to alter the calculation for start and end indices to match the original Matlab output, from which this module is based, default=False

Returns

- **qrs_area_3d** (*list of float*) – Values for the area under the curve (as defined by the 3D method) between the provided limits for each of the VCGs
- **qrs_area_pythag** (*list of float*) – Values for the area under the curve (as defined by the Pythagorean method) between the provided limits for each of the VCGs
- **qrs_area_components** (*list of list of float*) – Areas under the individual x, y, z curves of the VCG, for each of the supplied VCGs

signalanalysis.vcg.get_vcg_from_ecg

`signalanalysis.vcg.`**`get_vcg_from_ecg`**`(`*ecgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame]*`)` → List[pandas.core.frame.DataFrame]

Convert ECG data to vectorcardiogram (VCG) data using the Kors matrix method

Parameters **ecgs** (*list of pd.DataFrame or pd.DataFrame*) – List of ECG dataframe data, or ECG dataframe data directly, with dict keys corresponding to ECG outputs

Returns **vcgs** – List of VCG output data

Return type list of pd.DataFrame

References

Kors JA, van Herpen G, Sittig AC, van Bemmel JH. Reconstruction of the Frank vectorcardiogram from standard electrocardiographic leads: diagnostic comparison of different methods Eur Heart J. 1990 Dec;11(12):1083-92.

signalanalysis.vcg.get_weighted_dipole_angles

`signalanalysis.vcg.`**`get_weighted_dipole_angles`**`(`*vcgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame], t_start: Optional[List[float]] = None, t_end: Optional[List[float]] = None*`)` → Tuple[List[float], List[float], List[List[float]]]

Calculate metrics relating to the angles of the weighted dipole of the VCG. Usually used with QRS limits.

Calculates the weighted averages of both the azimuth and the elevation (inclination above the xy-plane) for a given section of the VCG. Based on these weighted averages of the angles, the unit weighted dipole for that section of the VCG is returned as well.

Parameters

- **vcgs** (*pd.DataFrame or list of pd.DataFrame*) – VCG data to calculate

- **t_start** (*list of float, optional*) – Start time from which to calculate the angles, default=0
- **t_end** (*list of float, optional*) – End time until which to calculate the angles, default=end

Returns

- **waa** (*list of float*) – List of Weighted Average Azimuth angles (in radians) for each given VCG
- **wae** (*list of float*) – List of Weighted Average Elevation (above xy-plane) angles (in radians) for each given VCG
- **uwd** (*list of list of float*) – x, y, z coordinates for the unit mean weighted dipole for the given (section of) VCGs

signalanalysis.vcg.plot_density_effect

signalanalysis.vcg.**plot_density_effect**(*metrics, metric_name, metric_labels=None, density_labels=None, linestyles=None, colours=None, markers=None*)

Plot the effect of density on metrics.

signalanalysis.vcg.plot_metric_change

signalanalysis.vcg.**plot_metric_change**(*metrics, metrics_phi, metrics_rho, metrics_z, metric_name, metrics_lv=None, labels=None, scattermarkers=None, linemarkers=None, colours=None, linestyles=None, layout=None, axis_match=True, no_labels=False*)

Function to plot all the various figures for trend analysis in one go.

signalanalysis.vcg.plot_metric_change_barplot

signalanalysis.vcg.**plot_metric_change_barplot**(*metrics_cont, metrics_lv, metrics_sept, metric_labels, layout=None*)

Plots a bar chart for the observed metrics.

Classes

Vcg	Base class to encapsulate data from VCG

signalanalysis.vcg.Vcg

class signalanalysis.vcg.**Vcg**(*ecg:* signalanalysis.ecg.Ecg, ***kwargs*)

Bases: *signalanalysis.general.Signal*

Base class to encapsulate data from VCG

Methods

get_from_ecg	Convert ECG data to vectorcardiogram (VCG) data using the Kors matrix method
get_n_beats	Calculate the number of beats in an ECG trace, and save the individual beats to file for later use
get_rms	Returns the RMS of the combined signal
reset	Reset all properties of the class

get_from_ecg(*ecg:* signalanalysis.ecg.Ecg)

Convert ECG data to vectorcardiogram (VCG) data using the Kors matrix method

Parameters **ecg** (signalanalysis.ecg.Ecg) – List of ECG dataframe data, or ECG dataframe data directly, with dict keys corresponding to ECG outputs

References

Kors JA, van Herpen G, Sittig AC, van Bemmel JH. Reconstruction of the Frank vectorcardiogram from standard electrocardiographic leads: diagnostic comparison of different methods Eur Heart J. 1990 Dec;11(12):1083-92.

get_n_beats(*threshold: float = 0.5*, *min_separation: float = 0.2*, *unipolar_only: bool = True*, *plot: bool = False*)

Calculate the number of beats in an ECG trace, and save the individual beats to file for later use

When given the raw data of an ECG trace, will estimate the number of beats recorded in the trace based on the RMS of the ECG signal exceeding a threshold value. The estimated individual beats will then be saved in a list in a lossless manner, i.e. saved as [ECG1, ECG2, …, ECG(n)], where ECG1=[0:peak2], ECG2=[peak1:peak3], …, ECGn=[peak(n-1):end]

threshold [float {0<1}] Minimum value to search for for a peak in RMS signal to determine when a beat has occurred, default=0.5

min_separation [float] Minimum time (in s) that should be used to separate separate beats, default=0.2s

unipolar_only [bool, optional] Whether to use only unipolar ECG leads to calculate RMS, default=True

plot [bool] Whether to plot results of beat detection, default=False

self.n_beats [int] Number of beats detected in signal

The scalar RMS is calculated according to

rac{1}{n}sum_{i=1}^n (extnormal{ECG}_i^2(t))} for all leads available from the signal (12 for ECG, 3 for VCG). If unipolar_only is set to true, then ECG RMS is calculated using only 'unipolar' leads. This uses V1-6, and the non-augmented limb leads (VF, VL and VR)

..math:: VF = LL-V_{WCT} =

rac{2}{3}aVF ..math:: VL = LA-V_{WCT} =
rac{2}{3}aVL ..math:: VR = RA-V_{WCT} =

rac{2}{3}aVR

get_rms(*preprocess_data: Optional[pandas.core.frame.DataFrame] = None*, *drop_columns: Optional[List[str]] = None*)

Returns the RMS of the combined signal

Parameters

- **preprocess_data** (*pd.DataFrame, optional*) – Only passed if there is some extant data that is to be used for getting the RMS (for example, if the unipolar data only from ECG is being used, and the data is thus preprocessed in a manner specific for ECG data in the ECG routine)
- **drop_columns** (*list of str, optional*) – List of any columns to drop from the raw data before calculating the RMS. Can be used in conjunction with preprocess_data
- **unipolar_only** (*#*) –
- **RMS** (*# Whether to use only unipolar ECG leads to calculate*) –
- **default=True** –

reset()

Reset all properties of the class

Function called when reading in new data into an existing class (for some reason), which would make these properties and attributes clash with the other data

2.2 signalplot

signalplot.ecg
signalplot.general
signalplot.vcg

2.2.1 signalplot.ecg

Functions

plot	Plot and label the ECG data from simulation(s).

signalplot.ecg.plot

`signalplot.ecg.plot(`*ecgs: Union[List[pandas.core.frame.DataFrame], pandas.core.frame.DataFrame], legend_ecg: Optional[List[str]] = None, linewidths_ecg: float = 2, limits: Optional[Union[list, float]] = None, legend_limits: Optional[List[str]] = None, plot_sequence: Optional[List[str]] = None, single_fig: bool = True, colours_ecg: Optional[Union[List[str], List[List[float]], List[Tuple[float]]]] = None, linestyles_ecg: Optional[List[str]] = '-', colours_limits: Optional[Union[List[str], List[List[float]], List[Tuple[float]]]] = None, linestyles_limits: Optional[List[str]] = None, fig: Optional[matplotlib.pyplot.figure] = None, ax=None*`)` → tuple

Plot and label the ECG data from simulation(s). Optional to add in QRS start/end boundaries for plotting

Parameters

- **ecgs** (*pd.DataFrame or list of pd.DataFrame*) – Dataframe or list of dataframes for ECG data, with keys corresponding to the trace name and index to the time data
- **legend_ecg** (*list of str, optional*) – List of names for each given set of ECG data e.g. ['BCL=300ms', 'BCL=600ms'], default=None
- **linewidths_ecg** (*float, optional*) – Width to use for plotting lines, default=3
- **limits** (*float or list of float or pd.DataFrame, optional*) – Optional temporal limits (e.g. QRS limits) to add to ECG plots. Can add multiple limits, which will be plotted identically on all axes. If provided as a dataframe, will plot the limits on the relevant axis
- **legend_limits** (*list of str, optional*) – List of names for each given set of limits e.g. ['QRS start', 'QRS end'], default=None
- **plot_sequence** (*list of str, optional*) – Sequence in which to plot the ECG traces. Will default to: V1, V2, V3, V4, V5, V6, LI, LII, LIII, aVR, aVL, aVF
- **single_fig** (*bool, optional*) – If true, will plot all axes on a single figure window. If false, will plot each axis on a separate figure window. Default is True
- **colours_ecg** (*str or list of str or list of list/tuple of float, optional*) – Colours to be used to plot ECG traces. Can provide as either string (e.g. 'b') or as RGB values (floats). Will default to ca.get_plot_colours()
- **linestyles_ecg** (*str or list, optional*) – Linestyles to be used to plot ECG traces. Will default to ca.get_plot_lines()
- **colours_limits** (*str or list of str or list of list/tuple of float, optional*) – Colours to be used to plot limits. Can provide as either string (e.g. 'b') or as RGB values (floats). Will default to ca.get_plot_colours()
- **linestyles_limits** (*str or list, optional*) – Linestyles to be used to plot limits. Will default to ca.get_plot_lines()
- **fig** (*optional*) – If given, will plot data on existing figure window

- **ax** (*optional*) – If given, will plot data using existing axis handles

Returns

- *fig* – Handle to output figure window, or dictionary to several handles if traces are all plotted in separate figure windows (if single_fig=False)
- **ax** (*dict*) – Dictionary to axis handles for ECG traces

Raises

- **AssertionError** – Checks that various list lengths are the same
- **TypeError** – If input argument is given in an unexpected format

2.2.2 signalplot.general

2.2.3 signalplot.vcg

Functions

add_unit_sphere	Add a unit sphere to a 3D plot
animate_3d	Animate the evolution of the VCG in 3D space, saving that animation to a file.
plot_2d	Plot x vs y (or y vs z, or other combination) for VCG trace, with line colour shifting to show time progression.
plot_3d	Plot the evolution of VCG in 3D space
plot_arc3d	Plot arc between two given vectors in 3D space.
plot_density_effect	Plot the effect of density on metrics.
plot_metric_change	Function to plot all the various figures for trend analysis in one go.
plot_metric_change_barplot	Plots a bar chart for the observed metrics.
plot_spatial_velocity	Plot the spatial velocity for given VCG data
plot_xyz_components	Plot x, y, z components of VCG data
plot_xyz_vector	Plots a specific vector in 3D space (e.g.

signalplot.vcg.add_unit_sphere

signalplot.vcg.**add_unit_sphere**(*ax*) → None

Add a unit sphere to a 3D plot

Parameters **ax** – Handles to axes

signalplot.vcg.animate_3d

`signalplot.vcg.animate_3d`(*vcg: numpy.ndarray, limits: Optional[Union[float, List[float], List[List[float]]]] = None, linestyle: Optional[str] = '-', colourmap: Optional[str] = 'viridis', linewidth: Optional[float] = 3, output_file: Optional[str] = 'vcg_xyz.mp4'*) → None

Animate the evolution of the VCG in 3D space, saving that animation to a file.

Parameters

- **vcg** (*np.ndarray*) – VCG data
- **limits** (*float or list of float or list of list of floats, optional*) –

  Limits for the axes. If none, will set to the min/max values of the provided data. Can provid

  1) a single value (+/- of that value applied to all axes)
  2) [min, max] to be applied to all axes
  3) [[xmin, xmax], [ymin, ymax], [zmin, zmax]]
- **linestyle** (*str, optional*) – Linestyle for the data, default='-'
- **colourmap** (*str, optional*) – Colourmap to use when plotting, default='viridis'
- **linewidth** (*float, optional*) – Linewidth when used to plot VCG, default=3
- **output_file** (*str, optional*) – Name of the file to save the animation to, default='vcg_xyz.mp4'

signalplot.vcg.plot_2d

`signalplot.vcg.plot_2d`(*vcg: pandas.core.frame.DataFrame, x_plot: str = 'x', y_plot: str = 'y', linestyle: str = '-', colourmap: str = 'viridis', linewidth: float = 3, axis_limits: Optional[Union[float, List[float]]] = None, fig: Optional[matplotlib.pyplot.figure] = None*) → matplotlib.pyplot.figure

Plot x vs y (or y vs z, or other combination) for VCG trace, with line colour shifting to show time progression.

Plot a colour-varying course of a VCG in 2D space

Parameters

- **vcg** (*pd.DataFrame*) – VCG data to be plotted
- **x_plot** (*str, optional*) – Which components of VCG to plot, default='x', 'y'
- **y_plot** (*str, optional*) – Which components of VCG to plot, default='x', 'y'
- **linestyle** (*str, optional*) – Linestyle to apply to the plot, default='-'

- **colourmap** (*str, optional*) – Colourmap to use for the line, default='viridis'
- **linewidth** (*float, optional*) – Linewidth to use, default=3
- **axis_limits** (*list of float or float, optional*) – Limits to apply to the axes, default=None
- **fig** (*plt.figure, optional*) – Handle to pre-existing figure (if present) on which to plot data, default=None

Returns **fig** – Handle to output figure window

Return type plt.figure

signalplot.vcg.plot_3d

signalplot.vcg.**plot_3d**(*vcg: pandas.core.frame.DataFrame*, *linestyle: str = '-'*, *colourmap: str = 'viridis'*, *linewidth: float = 3.0*, *axis_limits: Optional[Union[float, List[float]]] = None*, *unit_min: bool = True*, *sig_fig: Optional[int] = None*, *fig: Optional[matplotlib.pyplot.figure] = None*) → matplotlib.pyplot.figure

Plot the evolution of VCG in 3D space

Parameters

- **vcg** (*pd.DataFrame*) – VCG data
- **linestyle** (*str, optional*) – Linestyle to plot data, default='-'
- **colourmap** (*str, optional*) – Colourmap to use when plotting data, default='viridis'
- **linewidth** (*float, optional*) – Linewidth to use, default=3
- **axis_limits** (*list of float or float, optional*) – Limits to apply to the axes, default=None
- **unit_min** (*bool, optional*) – Whether to have the axes set to, as a minimum, unit length, default=True
- **sig_fig** (*int, optional*) – Maximum number of decimal places to be used on the axis plots (e.g., if set to 2, 0.12345 will be displayed as 0.12). Used to avoid floating point errors, default=None (no adaption made)
- **fig** (*plt.figure, optional*) – Handle to existing figure (if exists)

Returns **fig** – Figure handle

Return type plt.figure

signalplot.vcg.plot_arc3d

`signalplot.vcg.plot_arc3d`(*vector1: List[float]*, *vector2: List[float]*, *radius: float = 0.2*, *fig: Optional[matplotlib.pyplot.figure] = None*, *colour: str = 'C0'*) → matplotlib.pyplot.figure

Plot arc between two given vectors in 3D space.

Parameters

- **vector1** (*list of float*) – First vector
- **vector2** (*list of float*) – Second vector
- **radius** (*float, optional*) – Radius of arc to plot on figure
- **fig** (*plt.figure, optional*) – Handle of figure on which to plot the arc. If not given, will produce new figure
- **colour** (*str, optional*) – Colour in which to display the arc

Returns **fig** – Handle for figure on which arc has been plotted

Return type plt.figure

signalplot.vcg.plot_density_effect

`signalplot.vcg.plot_density_effect`(*metrics: List[List[float]]*, *metric_name: str*, *metric_labels: Optional[List[str]] = None*, *density_labels: Optional[List[str]] = None*, *linestyles: Optional[List[str]] = None*, *colours: Optional[List[str]] = None*, *markers: Optional[List[str]] = None*)

Plot the effect of density on metrics.

TODO: look into decorator for the LaTeX preamble?

Parameters

- **metrics** (*list of list of float*) – Effects of scar density on given metrics, presented as e.g. [metric_LV, metric_septum]
- **metric_name** (*str*) – Name of metric being assessed
- **metric_labels** (*list of str, optional*) – Labels for the metrics being plotted, default=['LV', 'Septum']
- **density_labels** (*list of str, optional*) – Labels for the different scar densities being plotted
- **linestyles** (*list of str, optional*) – Linestyles for the density effect plots, default=['-' for _ in range(len(metrics))]
- **colours** (*list of str, optional*) – Colours to use for the plot, default=common_analysis.get_plot_colours(len(metrics))
- **markers** (*list of str, optional*) – Markers to use for the discrete data points in the plot, default=['o' for _ in range(len(metrics))]

signalplot.vcg.plot_metric_change

`signalplot.vcg.plot_metric_change`(*metrics: List[List[List[float]]]*, *metrics_phi: List[List[List[float]]]*, *metrics_rho: List[List[List[float]]]*, *metrics_z: List[List[List[float]]]*, *metric_name: str*, *metrics_lv: Optional[List[bool]] = None*, *labels: Optional[List[str]] = None*, *scattermarkers: Optional[List[str]] = None*, *linemarkers: Optional[List[str]] = None*, *colours: Optional[List[str]] = None*, *linestyles: Optional[List[str]] = None*, *layout: Optional[str] = None*, *axis_match: bool = True*, *no_labels: bool = False*) → Tuple

Function to plot all the various figures for trend analysis in one go.

TODO: labels parameter seems redundant - potentially remove

Parameters

- **metrics** (*list of list of list of float*) – Complete list of all metric data recorded [phi+rho+z+size+other]
- **metrics_phi** (*list of list of list of float*) – Metric data recorded for scar size variations in phi UVC
- **metrics_rho** (*list of list of list of float*) – Metric data recorded for scar size variations in rho UVC
- **metrics_z** (*list of list of list of float*) – Metric data recorded for scar size variations in z UVC
- **metric_name** (*str*) – Name of metric being plotted (for labelling purposes). Can incorporate LaTeX typesetting.
- **metrics_lv** (*list of bool, optional*) – Boolean to distinguish whether metrics being plotted are for LV or septal data, default=[True, False]
- **labels** (*list of str, optional*) – Labels for the data sets being plotted, default=['LV', 'Septum']
- **scattermarkers** (*list of str, optional*) – Markers to use to plot the data on the scatterplots, default=['+', 'o', 'D', 'v', '^', 's', '*', 'x']
- **linemarkers** (*list of str, optional*) – Markers to use on the line plots to indicate discrete data points, required to be at least as long as the longest line plot to be drawn (rho), default=['.' for _ in range(len(metrics_rho))]
- **colours** (*list of str, optional*) – Sequence of colours to plot data (if plotting LV and septal data, will require two different colours to allow them to be distinguished), default=common_analysis.get_plot_colours(len(metrics_rho))
- **linestyles** (*list of str, optional*) – Linestyles to be used for plotting the data on lineplots, default=['-' for _ in range(len(metrics_rho))]

- **layout** (*{'combined', 'figures'}, optional*) – String specifying the output, whether all plots should be combined into one figure window (default), or whether individual figure windows should be plotted for each plot
- **axis_match** (*bool, optional*) – Whether to make sure all plotted figures share the same axis ranges, default=True
- **no_labels** (*bool, optional*) – Whether to have labels on the figures, or not - having no labels can make it far easier to 'prettify' the figures manually later in Inkscape, default=False

Returns

- **fig** (*plt.figure or dict of plt.figure*) – Handle to figure(s)
- **ax** (*dict*) – Handles to axes

signalplot.vcg.plot_metric_change_barplot

signalplot.vcg.**plot_metric_change_barplot**(*metrics_cont: List[List[float]], metrics_lv: List[List[float]], metrics_sept: List[List[float]], metric_labels: List[str], layout: Optional[str] = None*) → Tuple

Plots a bar chart for the observed metrics.

Parameters

- **metrics_cont** (*list of list of float*) – Values of series of metrics for no scar
- **metrics_lv** (*list of list of float*) – Values of series of metrics for LV scar
- **metrics_sept** (*list of list of float*) – Values of series of metrics for septal scar
- **metric_labels** (*list of str*) – Names of metrics being plotted
- **layout** (*{'combined', 'fig'}, optional*) – Whether to plot bar charts on combined plot window, or in individual figure windows

Returns

- **fig** (*plt.figure or list of plt.figure*) – Handle(s) to figures
- **ax** (*list*) – Handles to axes

signalplot.vcg.plot_spatial_velocity

`signalplot.vcg.plot_spatial_velocity`(*vcg: Union[pandas.core.frame.DataFrame, List[pandas.core.frame.DataFrame]]*, *sv: Optional[List[List[float]]] = None*, *limits: Optional[List[List[float]]] = None*, *fig: Optional[matplotlib.pyplot.figure] = None*, *legend_vcg: Optional[Union[List[str], str]] = None*, *legend_limits: Optional[Union[List[str], str]] = None*, *limits_linestyles: Optional[List[str]] = None*, *limits_colours: Optional[List[str]] = None*, *filter_sv: bool = True*) → Tuple

Plot the spatial velocity for given VCG data

Plot the spatial velocity and VCG elements, with limits (e.g. QRS limits) if provided. Note that if spatial velocity is not provided, default values will be used to calculate it - if anything else is desired, then spatial velocity must be calculated first and provided to the function.

Parameters

- **vcg** (*pd.DataFrame or list of pd.DataFrame*) – VCG data
- **sv** (*list of list of float, optional*) – Spatial velocity data. Only required to be given here if special parameters wish to be given, otherwise it will be calculated using default parameters (default)
- **limits** (*list of list of float, optional*) – A series of 'limits' to be plotted on the figure with the VCG and spatial plot. Presented as a list of the same length of the VCG data, with the required limits within:

  e.g. [[QRS_start1, QRS_start2, …], [QRS_end1, QRS_end2, …], …]

  Default=None
- **fig** (*plt.figure, optional*) – Handle to existing figure, if data is wished to be plotted on existing plot, default=None
- **legend_vcg** (*str or list of str, optional*) – Labels to apply to the VCG/SV data, default=None
- **legend_limits** (*str or list of str, optional*) – Labels to apply to the limits, default=None
- **limits_linestyles** (*list of str, optional*) – Linestyles to apply to the different limits being supplied, default=None (will use varying linestyles based on tools.plotting.get_plot_lines)
- **limits_colours** (*list of str, optional*) – Colours to apply to the different limits being supplied, default=None (will use varying colours based on tools.plotting.get_plot_colours)
- **filter_sv** (*bool, optional*) – Whether or not to apply filtering to spatial velocity prior to finding the start/end points for the threshold, default=True

Returns Handles to the figure and axes generated

Return type fig, ax

signalplot.vcg.plot_xyz_components

`signalplot.vcg.plot_xyz_components`(*vcgs: Union[pandas.core.frame.DataFrame, List[pandas.core.frame.DataFrame]]*, *legend: Optional[List[str]] = None*, *colours: Optional[List[List[float]]] = None*, *linestyles: Optional[List[str]] = None*, *legend_location: Optional[str] = None*, *limits: Optional[List[List[float]]] = None*, *limits_legend: Optional[List[str]] = None*, *limits_colours: Optional[List[List[float]]] = None*, *limits_linestyles: Optional[List[str]] = None*, *limits_legend_location: str = 'lower right'*, *layout: str = 'grid'*) → tuple

Plot x, y, z components of VCG data

Multiple options given for layout of resulting plot

Parameters

- **vcgs** (*list of pd.DataFrame or pd.DataFrame*) – List of vcg data: [vcg_data1, vcg_data2, …]
- **legend** (*list of str, optional*) – Legend names for each VCG trace, default=None
- **colours** (*list of list of float or list of str, optional*) – Colours to use for plotting, default=common_analysis.get_plot_colours
- **linestyles** (*list of str, optional*) – Linestyles to use for plotting, default='-'
- **legend_location** (*str, optional*) – Location to plot the legend. Default=None, which will translate to 'best' if no legend is required for limits, or 'upper right' if legend is needed for limits
- **limits** (*list of list of float, optional*) – QRS limits to plot on axes, default=None To be presented in form [[qrs_start1, qrs_starts, …], [qrs_end1, qrs_end2, …], …]
- **limits_legend** (*list of str, optional*) – Legend to apply to the limits plotted, default=None
- **limits_colours** (*list of list of float or list of str, optional*) – Colours to use when plotting limits, default=common_analysis.get_plot_colours
- **limits_linestyles** (*list of str, optional*) – Linestyles to use when plotting limits, default='-'
- **limits_legend_location** (*str, optional*) – Location to use for the legend containing the limits data
- **layout** (*{'grid', 'figures', 'combined', 'row', 'column', 'best'}, optional*) –

  Layout of resulting plot grid x,y,z plots are arranged in a grid (like best, but more rigid grid) figures Each x,y,z plot is on a separate figure combined x,y,z plots are combined on a single set of axes row x,y,z plots are arranged on a horizontal row in one figure column x,y,z plots are arranged in a vertical column in one figure best x,y,z plots are arranged to try and optimise space (nb: figures not equal sizes…)

Returns Handle for resulting figure(s) and axes

Return type fig, ax

signalplot.vcg.plot_xyz_vector

signalplot.vcg.**plot_xyz_vector**(*vector: Optional[List[float]] = None*, *x: Optional[float] = None*, *y: Optional[float] = None*, *z: Optional[float] = None*, *fig: Optional[matplotlib.pyplot.figure] = None*, *linecolour: str = 'C0'*, *linestyle: str = '-'*, *linewidth: float = 2*)

Plots a specific vector in 3D space (e.g. to reflect maximum dipole)

Parameters

- **vector** (*list of float*) – [x, y, z] values of vector to plot, alternatively given as separate x, y, z variables
- **x** (*float*) – [x, y, z] values of vector to plot, alternatively given as vector variable
- **y** (*float*) – [x, y, z] values of vector to plot, alternatively given as vector variable
- **z** (*float*) – [x, y, z] values of vector to plot, alternatively given as vector variable
- **fig** (*plt.figure, optional*) – Existing figure handle, if desired to plot the vector onto an extant plot
- **linecolour** (*str, optional*) – Colour to plot the vector as
- **linestyle** (*str, optional*) – Linestyle to use to plot the body of the arrow
- **linewidth** (*float, optional*) – Width to plot the body of the arrow

Returns **fig** – Figure handle

Return type plt.figure

Raises **ValueError** – Exactly one of vertices and x,y,z must be given

Notes

Must provide either vector or [x,y,z]

2.3 tools

tools.maths	
tools.plotting	
tools.python	

2.3.1 tools.maths

Functions

acos2	Function to return the inverse cos function across the range (-pi, pi], rather than (0, pi]
asin2	Function to return the inverse sin function across the range (-pi, pi], rather than (-pi/2, pi/2]
filter_butterworth	Filter data using Butterworth filter
filter_savitzkygolay	Filter EGM data using a Savitzky-Golay filter
get_median	Add the median value of data to a dataframe
normalise_signal	Returns a normalised signal, such that the maximum value in the signal is 1, or the minimum is -1
simplex_volume	Return the volume of the simplex with given vertices or sides.

tools.maths.acos2

`tools.maths.`**`acos2`**(*x: float*, *y: float*) → float

Function to return the inverse cos function across the range (-pi, pi], rather than (0, pi]

Parameters

- **x** (*float*) – x coordinate of the point in 2D space
- **y** (*float*) – y coordinate of the point in 2D space

Returns **theta** – Angle corresponding to point in 2D space in radial coordinates, within range (-pi, pi]

Return type float

tools.maths.asin2

`tools.maths.asin2(x: float, y: float) → float`

Function to return the inverse sin function across the range (-pi, pi], rather than (-pi/2, pi/2]

Parameters

- **x** (*float*) – x coordinate of the point in 2D space
- **y** (*float*) – y coordinate of the point in 2D space

Returns theta – Angle corresponding to point in 2D space in radial coordinates, within range (-pi, pi]

Return type float

tools.maths.filter_butterworth

`tools.maths.filter_butterworth(data: Union[numpy.ndarray, pandas.core.frame.DataFrame], sample_freq: float = 500.0, freq_filter: float = 40, order: int = 2, filter_type: str = 'low') → Union[numpy.ndarray, pandas.core.frame.DataFrame]`

Filter data using Butterworth filter

Filter a given set of data using a Butterworth filter, designed to have a specific passband for desired frequencies. It is set up to use seconds, not milliseconds.

Parameters

- **data** (*np.ndarray or pd.DataFrame*) – Data to filter
- **sample_freq** (*int or float*) – Sampling rate of data (Hz), default=500. If data passed as dataframe, the sample_freq will be calculated from the dataframe index.
- **freq_filter** (*int or float*) – Cut-off frequency for filter, default=40
- **order** (*int*) – Order of the Butterworth filter, default=2
- **filter_type** (*{'low', 'high', 'band'}*) – Type of filter to use, default='low'

Returns filter_out – Output filtered data

Return type np.ndarray

tools.maths.filter_savitzkygolay

`tools.maths.filter_savitzkygolay(data: pandas.core.frame.DataFrame, window_length: int = 0.05, order: int = 2, deriv: int = 0, delta: float = 1.0)`

Filter EGM data using a Savitzky-Golay filter

Filter a given set of data using a Savitzky-Golay filter, designed to smooth data using a convolution process fitting to a low-degree polynomial within a given window. Default values are either taken from scipy documentation (not all options are provided here), or adapted to match Hermans et al.

Parameters

- **data** (*pd.DataFrame*) – Data to filter
- **window_length** (*float, optional*) – The length of the filter window in seconds. When passed to the scipy filter, will be converted to a positive odd integer (i.e. the number of coefficients). Default=0.05
- **order** (*int, optional*) – The order of the polynomial used to fit the samples. polyorder must be less than window_length. Default=2
- **deriv** (*int, optional*) – The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.
- **delta** (*float, optional*) – The spacing of the samples to which the filter will be applied. This is only used if deriv > 0. Default=1.0

Returns **data** – Output filtered data

Return type pd.DataFrame

References

The development and validation of an easy to use automatic QT-interval algorithm
Hermans BJM, Vink AS, Bennis FC, Filippini LH, Meijborg VMF, Wilde AAM, Pison L, Postema PG, Delhaas T PLoS ONE, 12(9), 1–14 (2017) https://doi.org/10.1371/journal.pone.0184352

tools.maths.get_median

`tools.maths.`**`get_median`**`(`*data: pandas.core.frame.DataFrame*, *remove_outliers: bool = True*`)` → pandas.core.frame.DataFrame

Add the median value of data to a dataframe

TODO: Complete this code if required (currently only potentially useful for T-wave analysis)

tools.maths.normalise_signal

`tools.maths.`**`normalise_signal`**`(`*data: Union[numpy.ndarray, pandas.core.frame.DataFrame]*`)` → Union[numpy.ndarray, pandas.core.frame.DataFrame]

Returns a normalised signal, such that the maximum value in the signal is 1, or the minimum is -1

Parameters **data** (*np.ndarray*) – Signal to be normalised

Returns **normalised_data** – Normalised signal

Return type np.ndarray or pd.DataFrame

tools.maths.simplex_volume

tools.maths.**simplex_volume**(*, *vertices=None*, *sides=None*) → float

Return the volume of the simplex with given vertices or sides.

If vertices are given they must be in a NumPy array with shape (N+1, N): the position vectors of the N+1 vertices in N dimensions. If the sides are given, they must be the compressed pairwise distance matrix as returned from scipy.spatial.distance.pdist.

Raises a ValueError if the vertices do not form a simplex (for example, because they are coplanar, colinear or coincident).

Warning: this algorithm has not been tested for numerical stability.

2.3.2 tools.plotting

Functions

add_colourbar	Add arbitrary colourbar to a figure, for instances when an automatic colorbar isn't available
add_xyz_axes	Plot dummy axes (can't move splines in 3D plots)
get_plot_colours	Return iterable list of RGB colour values that can be used for custom plotting functions
get_plot_lines	Returns different line-styles for plotting
set_axis_limits	Set axis limits (not automatic for line collections, so needs to be done manually)
set_symmetrical_axis_limits	Sets symmetrical limits for a series of axes
write_colourmap_to_xml	Create a Paraview friendly colourmap useful for highlighting a particular range

tools.plotting.add_colourbar

tools.plotting.**add_colourbar**(*limits: List[float]*, *fig: Optional[matplotlib.pyplot.figure] = None*, *colourmap: str = 'viridis'*, *n_elements: int = 100*) → None

Add arbitrary colourbar to a figure, for instances when an automatic colorbar isn't available

Parameters

- **limits** (*list of float*) – Numerical limits to apply
- **fig** (*plt.figure, optional*) – Figure on which to plot the colourbar. If not provided (default=None), then will pick up the figure most recently available
- **colourmap** (*str, optional*) – Colourmap to be used, default='viridis'
- **n_elements** (*int, optional*) – Number of entries to be made in the colourmap index, default=100

Notes

This is useful for instances such as when LineCollections are used to plot line that changes colour during the plotting process, as LineCollections do not enable an automatic colorbar to be added to the plot. This function adds a dummy colorbar to replace that.

tools.plotting.add_xyz_axes

`tools.plotting.add_xyz_axes`(*fig: matplotlib.pyplot.figure*, *ax: mpl_toolkits.mplot3d.axes3d.Axes3D*, *axis_limits: Optional[Union[float, List[float], List[List[float]]]] = None*, *symmetrical_axes: bool = False*, *equal_limits: bool = False*, *unit_axes: bool = False*, *sig_fig: Optional[int] = None*) → None

Plot dummy axes (can't move splines in 3D plots)

Parameters

- **fig** (*plt.figure*) – Figure handle
- **ax** (*Axes3D*) – Axis handle
- **axis_limits** (*float or list of float or list of list of float, optional*) – Axis limits, either same for all dimensions (min=-max), or individual limits ([min, max]), or individual limits for each dimension
- **symmetrical_axes** (*bool, optional*) – Apply same limits to x, y and z axes
- **equal_limits** (*bool, optional*) – Set axis minimum to minus axis maximum (or vice versa)
- **unit_axes** (*bool, optional*) – Apply minimum of -1 -> 1 for axis limits
- **sig_fig** (*int, optional*) – Maximum number of decimal places to be used on the axis plots (e.g., if set to 2, 0.12345 will be displayed as 0.12). Used to avoid floating point errors, default=None (no adaption made)

tools.plotting.get_plot_colours

`tools.plotting.get_plot_colours`(*n: int = 10*, *colourmap: Optional[str] = None*) → List[Tuple[float]]

Return iterable list of RGB colour values that can be used for custom plotting functions

Returns a list of RGB colours values, potentially according to a specified colourmap. If n is low enough, will use the custom 'tab10' colourmap by default, which will use alternating colours as much as possible to maximise visibility. If n is too big, then the default setting is 'viridis', which should provide a gradation of colour from first to last.

Parameters

- **n** (*int, optional*) – Number of distinct colours required, default=10
- **colourmap** (*str*) – Matplotlib colourmap to base the end result on. Will default to 'tab10' if n<11, 'viridis' otherwise

Returns **cmap** – List of RGB values

Return type list of tuple

tools.plotting.get_plot_lines

`tools.plotting.get_plot_lines(n: int = 4)` → Union[List[tuple], List[str]]

Returns different line-styles for plotting

Parameters **n** (*int, optional*) – Number of different line-styles required

Returns **lines** – List of different line-styles

Return type list of str or list of tuple

tools.plotting.set_axis_limits

`tools.plotting.set_axis_limits(ax, data: Optional[pandas.core.frame.DataFrame] = None, unit_min: bool = True, axis_limits: Optional[Union[float, List[float]]] = None, pad_percent: float = 0.01)` → None

Set axis limits (not automatic for line collections, so needs to be done manually)

Parameters

- **ax** – Handles to the axes that need to be adjusted
- **data** (*pd.DataFrame, optional*) – Data that has been plotted, default=None
- **unit_min** (*bool, optional*) – Whether to have the axes set to, as a minimum, unit length, default=True
- **axis_limits** (*list of float or float, optional*) – Min/max values for axes, either as one value (i.e. min=-max), or two separate values. Same axis limits will be applied to all dimensions
- **pad_percent** (*float, optional*) – Percentage 'padding' to add to the ranges, to try and ensure that the edges of linewidths are not cut off, default=0.01

tools.plotting.set_symmetrical_axis_limits

`tools.plotting.set_symmetrical_axis_limits(ax_min: float, ax_max: float, unit_axes: bool = False)` → Tuple[float, float]

Sets symmetrical limits for a series of axes

TODO: fold functionality into set_axis_limits to avoid redundant functions

Parameters

- **ax_min** (*float*) – Minimum value for axes
- **ax_max** (*float*) – Maximum value for axes
- **unit_axes** (*bool, optional*) – Whether to apply a minimum axis range of [-1,1]

Returns **ax_min, ax_max** – Symmetrical axis limits, where ax_min=-ax_max

Return type float

tools.plotting.write_colourmap_to_xml

`tools.plotting.`**`write_colourmap_to_xml`**`(`*start_data: float*, *end_data: float*, *start_highlight: float*, *end_highlight: float*, *opacity_data: float = 1*, *opacity_highlight: float = 1*, *n_tags: int = 20*, *colourmap: str = 'viridis'*, *outfile: str = 'colourmap.xml'*`)` → None

Create a Paraview friendly colourmap useful for highlighting a particular range

Creates a colourmap that is entirely gray, save for a specified region of interest that will vary according to the specified colourmap

start_data start value for overall data (can't just use data for region of interest -
Paraview will scale)

end_data end value for overall data start_highlight start value for region of interest end_highlight end value for region of interest

opacity_data 1.0 overall opacity to use for all data opacity_highlight 1.0 opacity for region of interest colourmap 'viridis' colourmap to use outfile 'colourmap.xml' filename to save .xml file under

None

2.3.3 tools.python

Functions

check_list_depth	Function to calculate the depth of nested loops
convert_index_to_time	Return 'real' time for a given index
convert_input_to_list	Convert a given input to a list of inputs of required length.
convert_time_to_index	Converts a given time point to the relevant index value
deprecated_convert_time_to_index	Return indices of QRS start and end points.
find_list_fraction	Find index corresponding to certain fractional length within a list, e.g.
get_i_colour	Get index appropriate to colour value to plot on a figure (will be 0 if brand new figure)
get_time	Returns variables for time, dt and t_end, depending on input.
recursive_len	Return the total number of elements with a potentially nested list

tools.python.check_list_depth

`tools.python.check_list_depth(input_list, depth_count=1, max_depth=0, n_args=0)`

Function to calculate the depth of nested loops

TODO: Finish this damn code

Parameters

- **input_list** (*list*) – Input argument to check
- **depth_count** (*int, optional*) – Depth of nested loops thus far
- **max_depth** (*int, optional*) – Maximum expected depth of list, default=0 (not checked)
- **n_args** (*int, optional*) – Required length of 'base' list, default=0 (not checked)

Returns **depth_count** – Depth of nested loops

Return type int

Notes

A list of form [a1, a2, a3, . . .] has depth 1. A list of form [[a1, a2, a3, . . .], [b1, b2, b3, . . .], . . .] has depth 2. And so forth. . .

If n_args is set to an integer greater than 0, it will check that the lowest level of lists (for all entries) will be of the required length

if depth=1 as above, len([a1, a2, a3, . . .]) == n_args if depth=2 as above, len([a1, a2, a3, . . .]) == n_args && len([b1, b2, b3, . . .]) == n_args

tools.python.convert_index_to_time

`tools.python.convert_index_to_time(idx: int, time: Optional[numpy.ndarray] = None, t_start: float = 0, t_end: float = 200, dt: float = 2) → float`

Return 'real' time for a given index

Parameters

- **idx** (*int*) – Index to convert
- **time** (*np.ndarray, optional*) – Time data; if not provided, will be assumed from t_start, t_end and dt variables, default=None
- **t_start** (*float, optional*) – Start time for overall data, default=0
- **t_end** (*float, optional*) – End time for overall data, default=200
- **dt** (*float, optional*) – Interval between time points, default=2

Returns **time** – The time value that corresponds to the given index

Return type float

tools.python.convert_input_to_list

tools.python.**convert_input_to_list**(*input_data: Any*, *n_list: int = 1*, *n_list2: int = -1*, *list_depth: int = 1*, *default_entry: Optional[Any] = None*) → list

Convert a given input to a list of inputs of required length. If already a list, will confirm that it's the right length.

Parameters

- **input_data** (*Any*) – Input argument to be checked
- **n_list** (*int, optional*) – Number of entries required in input; if set to -1, will not perform any checks beyond 'depth' of lists, default=1
- **n_list2** (*int, optional*) – Number of entries for secondary input; if set to -1, will not perform any checks
- **list_depth** (*int*) – Number of nested lists required. If just a simple list of e.g. VCGs, then will be 1 ([vcg1, vcg2,…]). If a list of lists (e.g. [[qrs_start1, qrs_start2,…], [qrs_end1, qrs_end2,…]), then 2.
- **default_entry** (*{'colour', 'line', None, Any}, optional*) – Default entry to put into list. If set to None, will just repeat the input data to match n_list. However, if set to either 'colour' or 'line', will return the default potential settings, default=None

Returns **output** – Formatted output

Return type list

Notes

If the data are already provided as a list and list_depth==1, function will simply check that the list is of the correct length. If list_depth==2, will check that deepest level of nesting has the correct length; if n_list2 is provided, it will check the top level of the list is of the correct length. This is used, for example, when several different limits are provided for several different VCGs, and a legend is needed. Thus, if there are n different VCGs to be plotted, and each has m different limits to be plotted, the legend can be checked to be of the form [[x11, x21,…,xn1], [x12, x22,…,xn2],…[x1m, x2m,…xnm]]

If the data are not in list form, will:

(a) if default_entry==None, will replicate input_data to match n_vcg, e.g. '-' becomes ['-', '-',…]

(b) if default_entry=='colour', will return list of RBG values for colours

(c) if default_entry=='line', will return list of line entries

(d) for any other value of default_entry, will reproduce that value

tools.python.convert_time_to_index

`tools.python.convert_time_to_index`(*time_point: float*, *time: Optional[Union[List[float], numpy.ndarray]] = None*, *t_start: Optional[float] = None*, *t_end: Optional[float] = None*, *dt: Optional[float] = None*) → int

Converts a given time point to the relevant index value

Parameters

- **time_point** (*float*) – Time point for which we wish to find the corresponding index. If set to -1, will return the final index
- **time** (*float or np.ndarray, optional*) – Time data from which we wish to extract the index. If set to None, the time will be constructed based on the assumed t_start, t_end and dt values
- **t_start** (*float, optional*) – Start point of time; only used if `time' variable not given, default=None
- **t_end** (*float, optional*) – End point of time; only used if `time' variable not given, default=None
- **dt** (*float, optional*) – Interval between time points; only used if time not given, default=None

Returns **i_time** – Index corresponding to the time point given

Return type int

Raises **AssertionError** – If insufficient data are provided to the function to enable it to function

tools.python.deprecated_convert_time_to_index

`tools.python.deprecated_convert_time_to_index`(*qrs_start: Optional[float] = None*, *qrs_end: Optional[float] = None*, *time: Optional[List[float]] = None*, *t_start: float = 0*, *t_end: float = 200*, *dt: float = 2*) → Tuple[int, int]

Return indices of QRS start and end points. NB: indices returned match Matlab output

..deprecated:: This function is depreacted, but is in use due to other functions still using it for the moment

Parameters

- **qrs_start** (*float or int, optional*) – Start time to convert to index. If not given, will default to the same as the start time of the entire list
- **qrs_end** (*float or int, optional*) – End time to convert to index. If not given, will default to the same as the end time of the entire list
- **time** (*float, optional*) – Time data to be used to calculate index. If given, will over-ride the values used for dt/t_start/t_end. Default=None

- **t_start** (*float or int, optional*) – Start time of overall data, default=0
- **t_end** (*float or int, optional*) – End time of overall data, default=200
- **dt** (*float or int, optional*) – Interval between time points, default=2

Returns

- **i_qrs_start** (*int*) – Index of start time
- **i_qrs_end** (*int*) – Index of end time

tools.python.find_list_fraction

`tools.python.find_list_fraction`(*input_list*, *fraction=0.5*, *interpolate=True*)

Find index corresponding to certain fractional length within a list, e.g. halfway along, a third along

If only looking for an interval halfway along the list, uses a simpler method that is computationally faster

input_list List to find the fractional value of

fraction 0.5 Fraction of length of list to return the value of interpolate True If fraction does not precisely specify a particular entry in the list, whether to return the

values on either side, or whether to interpolate between the two values (with weighting given to how close the fraction is to one value or the other)

tools.python.get_i_colour

`tools.python.get_i_colour`(*axis_handle*) → int

Get index appropriate to colour value to plot on a figure (will be 0 if brand new figure)

tools.python.get_time

`tools.python.get_time`(*time: Optional[numpy.ndarray] = None*, *dt: Optional[float] = None*, *t_end: Optional[float] = None*, *n_vcg: Optional[int] = 1*, *len_vcg: Optional[List[int]] = None*) → Tuple[List[numpy.ndarray], List[float], List[float]]

Returns variables for time, dt and t_end, depending on input.

Parameters

- **time** (*np.ndarray, optional*) – Time data for a given VCG, default=None
- **dt** (*float, optional*) – Interval between recording points for the VCG, default=None
- **t_end** (*float, optional*) – Total duration of the VCG recordings, default=None
- **n_vcg** (*int, optional*) – Number of VCGs being assessed, default=1

- **len_vcg** (*int, optional*) – Number of data points for each VCG being assessed, None

Returns

- **time** (*list of np.ndarray*) – Time data for a given VCG
- **dt** (*list of float*) – Mean time interval for a given VCG recording
- **t_end** (*list of float*) – Total duration of each VCG recording

Notes

Time OR t_end/dt/len_vcg must be passed to this function

tools.python.recursive_len

`tools.python.recursive_len(item: list)`

Return the total number of elements with a potentially nested list

New stuff7

CHAPTER

THREE

INDICES AND TABLES

Note that, at the moment, none of the following links actually work…

- genindex
- modindex
- search

PYTHON MODULE INDEX