pyfda Documentation

Release v0.9.0b2

Christian Muenker

May 15, 2024

CONTENTS:

CHAPTER

ONE

PYFDA

1.1 Python Filter Design Analysis Tool

pyfda is a tool written in Python / Qt for analyzing and designing discrete time filters with a user-friendly GUI. Fixpoint filter implementations (for some filter types) can be simulated and tested for overflow and quantization behaviour in the time and frequency domain.

1.2 License

pyfda source code ist distributed under a permissive MIT license, binaries / bundles come with a GPLv3 license due to bundled components with stricter licenses.

1.3 Installing, running and uninstalling pyfda

For details, see INSTALLATION.md.

1.3.1 Binaries

Binaries can be downloaded under Releases for versioned releases and for a latest release, automatically created for each push to the main branch.

Self-extracting archives for **64 bit Windows**, **OS X** and **Ubuntu Linux** are created with **`pyInstaller <https://www.pyinstaller.org/>`_**. The archives self-extract to a temporary directory that is automatically deleted when pyfda is terminated (except when it crashes), they don't modify the system except for two ASCII configuration files and a log file. No additional software / libraries need to be installed, there is no interaction with existing python installations and you can simply overwrite or delete the executables when updating. After downloading the Linux archive, you need to make it executable (`chmod 775 pyfda_linux`).

Binaries for **Linux** are created as Flatpaks as well **(currently defunct)** which can also be downloaded from **`Flathub <https://flathub.org/apps/details/com.github.chipmuenk.pyfda>`_**. Many Linux distros have built-in flatpak support, for others it's easy to install with e.g. `sudo apt install flatpak`. For details check the Flatpak home page.

1.3.2 pip

Python 3.8 and above is supported, there is only one version of pyfda for all operating systems at PyPI. As pyfda is a pure Python project (no compilation required), you can install pyfda the usual way, required libraries are downloaded automatically if missing:

```
> pip install pyfda
```

Upgrade:

```
> pip install pyfda -U
```

Uninstall:

```
> pip uninstall pyfda
```

Starting pyfda

A pip installation creates a start script `pyfdax` in `<python>/Scripts` which should be in your path. So, simply start pyfda using

```
> pyfdax
```

The following libraries are required and installed automatically by pip when missing.

- **`PyQt <https://www.riverbankcomputing.com/software/pyqt/>`_** and **`Qt5 <https://qt.io/>`_**
- **`numpy <https://numpy.org/>`_**
- **`numexpr <https://github.com/pydata/numexpr>`_**
- **`scipy <https://scipy.org/>`_**: **1.2.0** or higher
- **`matplotlib <https://matplotlib.org/>`_**: **3.1** or higher
- **`Markdown <https://github.com/Python-Markdown/markdown>`_**

Optional libraries:

- **`mplcursors <https://mplcursors.readthedocs.io/>`_** for annotating cursors
- **`docutils <https://docutils.sourceforge.io>`_** for rich text in documentation
- **xlwt** and / or **XlsxWriter** for exporting filter coefficients as *.xls(x) files

1.3.3 conda

If you're working with Anaconda's packet manager conda, there is a recipe for pyfda on `conda-forge` since July 2023:

```
> conda install --channel=conda-forge pyfda
```

You should install pyfda into a new environment to avoid unwanted interaction with other installations.

1.3.4 git

If you want to contribute to pyfda (great idea!), fork the latest version from https://github.com/chipmuenk/pyfda.git and create a local copy using

```
> git clone https://github.com/<your_username>pyfda
```

This command creates a new folder `pyfda` at your current directory level and copies the complete pyfda project into it. This Github tutorial provides a good starting point for working with git repos.

pyfda can then be installed (i.e. creating local config files and the `pyfdax` starter script) from local files using

```
> pip install -e <YOUR_PATH_TO_PYFDA_setup.py>
```

Now you can edit the code and test it. If you're happy with it, push it to your repo and create a Pull Request so that the code can be reviewed and merged into the `chipmuenk/pyfda` repo.

1.4 Building pyfda

For details on how to publish pyfda to PyPI, how to create pyInstaller and Flatpak bundles, see BUILDING.md.

1.5 Customization

The location of the following two configuration files (copied to user space) can be checked via the tab `Files -> About`:

- Logging verbosity can be controlled via the file `pyfda_log.conf`
- Widgets and filters can be enabled / disabled via the file `pyfda.conf`. You can also define one or more user directories containing your own widgets and / or filters.

Layout and some default paths can be customized using the file `pyfda/pyfda_rc.py`, at the moment you have to edit that file at its original location.

1.6 Features

1.6.1 Filter design

- **Design methods**: Equiripple, Firwin, Moving Average, Bessel, Butterworth, Elliptic, Chebyshev 1 and 2 (from scipy.signal and custom methods)
- **Second-Order Sections** are used in the filter design when available for more robust filter design and analysis
- **Fine-tune** manually the filter order and corner frequencies calculated by minimum order algorithms
- **Compare filter designs** for a given set of specifications and different design methods
- **Filter coefficients and poles / zeroes** can be displayed, edited and quantized in various formats

- **Fixpoint filters** based on the integrated `Fixed()` class or on the Amaranth hardware description language.

1.6.2 User Interface

- enhanced Matplotlib NavigationToolbar (nicer icons, additional functions)
- tooltips for all UI widgets and help files
- specify frequencies as absolute values or normalized to sampling or Nyquist frequency
- specify ripple and attenuations in dB, as voltage or as power ratios
- enter values as expressions like `exp(-pi/4 * 1j)` using numexpr syntax

1.6.3 Graphical Analyses

- Magnitude response (lin / power / log) with optional display of specification bands, phase and an inset plot
- Phase response (wrapped / unwrapped) and group delay
- Pole / Zero plot
- Transient response (impulse, step and various stimulus signals) in the time and frequency domain. Define your own stimuli like `abs(sin(2*pi*n*f1))` using numexpr syntax and the UI.
- 3D-Plots (H|(f)|, mesh, surface, contour) with optional pole / zero display

1.6.4 Import / Export

- Filter designs in JSON, pickled and in numpy's NPZ-format
- Coefficients and poles/zeros as comma-separated values (CSV), in CMSIS format in numpy's NPY- and NPZ-formats, in Excel (R), as a Matlab (R) workspace or in FPGA vendor specific formats like Xilinx (R) COE-format
- Transient stimuli (y[n] tab) as wav and csv files

1.7 Target group

- **Educators and students:** Provide an easy-to-use FOSS tool for demonstrating DSP stuff and interactive filter design that also works with the limited resolution of a beamer.
- **Fixpoint filter designer:** Recursive fixpoint filter design has become a niche for experts. Convenient design and simulation support (round-off noise, stability under different quantization options and topologies) could attract more designers to these filters that are easier on hardware resources and much more suitable especially for uCs and low-budget FPGAs.

1.8 Release History

For details, see CHANGELOG.md.

1.9 Planned features (help wanted)

- Dark mode
- Use Amaranth to simulate fixpoint filters and export them in HDL format
- Graphical modification of poles / zeros
- Document filter designs in PDF / HTML format
- Design, analysis and export of filters as second-order sections, display and edit them in the P/Z widget
- Multiplier-free filter designs (CIC, GCIC, LDI, ΣΔ, ...) for fixpoint filters with a low number of multipliers (or none at all)
- Analysis of different fixpoint filter topologies (direct form, cascaded form, parallel form, ...) concerning overflow and quantization noise

CHAPTER

TWO

USER MANUAL

2.1 UI Overview

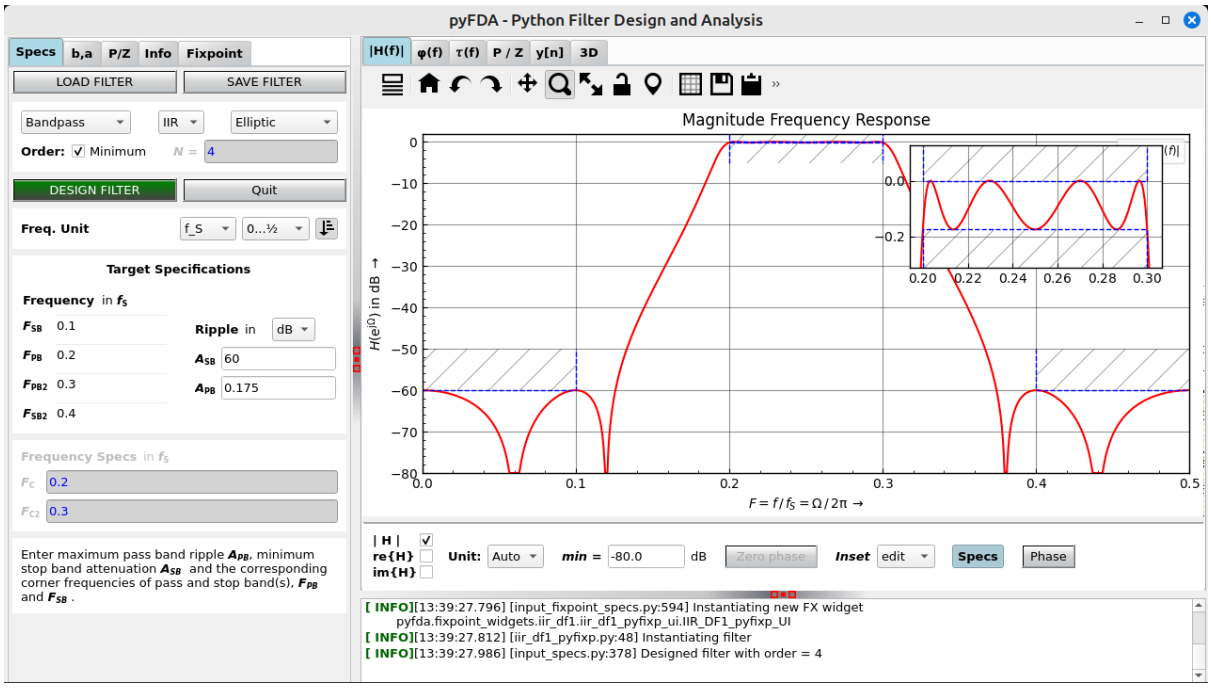

Fig. 2.1: Screenshot of pyfda

Fig. 2.1 shows the main pyfda screen with three subwindows that can be resized with the handles (red dots).

The tabs on the left-hand side access widgets to enter and view various specification and parameters for the filter / system to be designed resp. analyzed.

Subwindow for Input Widgets

2.1.1 Input Specs

Introduction

Fig. 2.2 shows a typical view of the **Specs** tab.

"Load" and "Save" … well, loads and saves complete filter designs. Coefficients and poles / zeros can be imported and exported in the "b,a" resp. the "P/Z" tab.

For the actual filter design, you can specify the kind of filter to be designed and its specifications in the frequency domain:

- **Response type** (low pass, band pass, ...)
- **Filter type** (IIR for a recursive filter with infinite impulse response or FIR for a non-recursive filter with finite impulse response)
- **Filter class** (elliptic, ...) allowing you to select the filter design algorithm

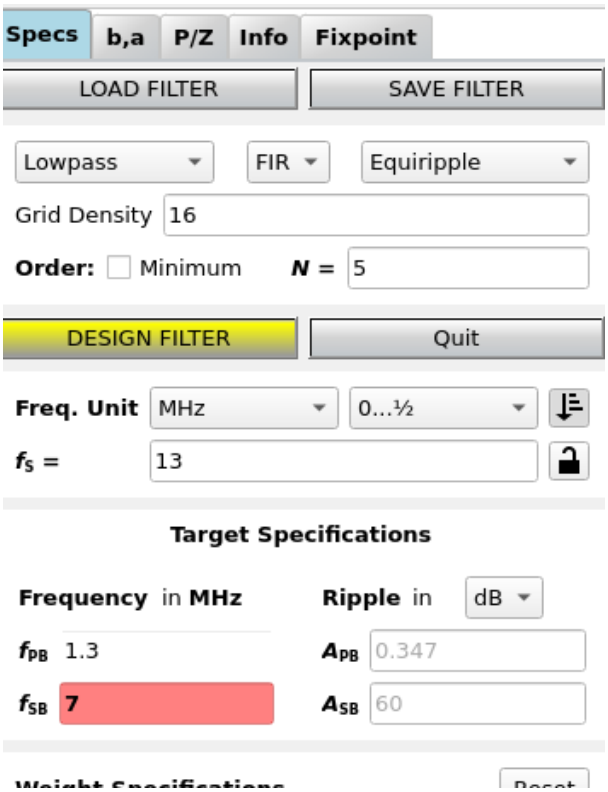

Fig. 2.2: Screenshot of specs input window

Not all combinations of design algorithms and response types are available - you won't be offered unavailable combinations and some fields may be greyed out.

A nice description of the design of FIR filters (also with pyfda) can be found at [Designing Generic FIR Filters with pyFDA and NumPy] (https://tomverbeure.github.io/2020/10/11/Designing-Generic-FIR-Filters-with-pyFDA-and-Numpy.html)

Order

The **order** of the filter, i.e. the number of poles / zeros / delays is either specified manually or the minimum order can be estimated for many filter algorithms to fulfill a set of given specifications.

Frequency Unit

In DSP, specifications and frequencies are expressed in different ways:

$$F = \frac{f}{f_S} \text{ or } \Omega = \frac{2\pi f}{f_S} = 2\pi F$$

In pyfda, you can enter parameters as absolute frequency f, as normalized frequency F w.r.t. to the *Sampling Frequency* f_S or to the *Nyquist Frequency* $f_{Ny} = f_S/2$ (Fig. 2.3):

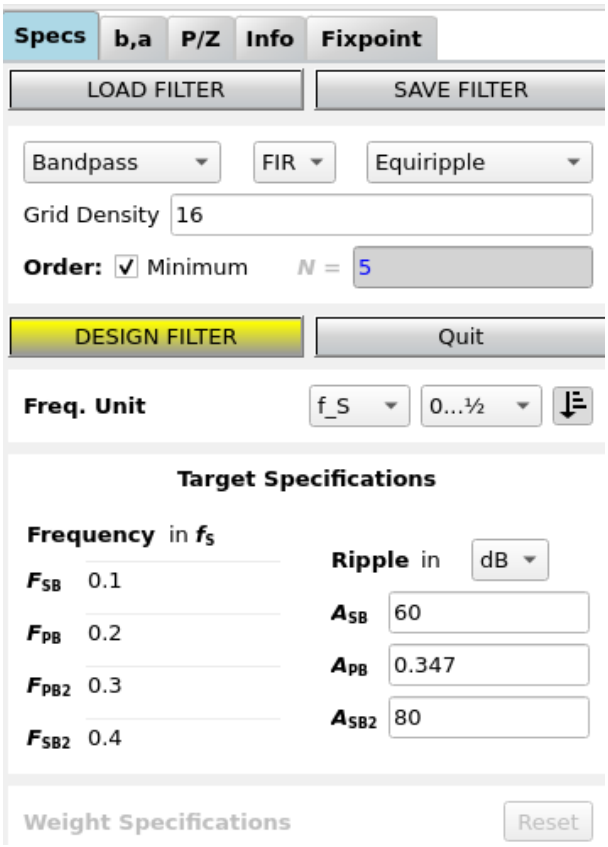

Fig. 2.3: Displaying normalized frequencies

Amplitude Unit

Amplitude specification can be entered as V, dB or W; they are converted automatically. Conversion depends on the filter type (IIR vs. FIR) and whether pass or stop band are specified. For details see the conversion functions `pyfda.libs.pyfda_lib.unit2lin()` and `pyfda.libs.pyfda_lib.lin2unit()`.

Background Info

Sampling Frequency

One of the most important parameters in a digital signal processing system is the **sampling frequency** f_S, defining the clock frequency with which the registers (flip-flops) in the system are updated. In a simple DSP system, the clock frequency of ADC, digital filter and DAC might be identical:

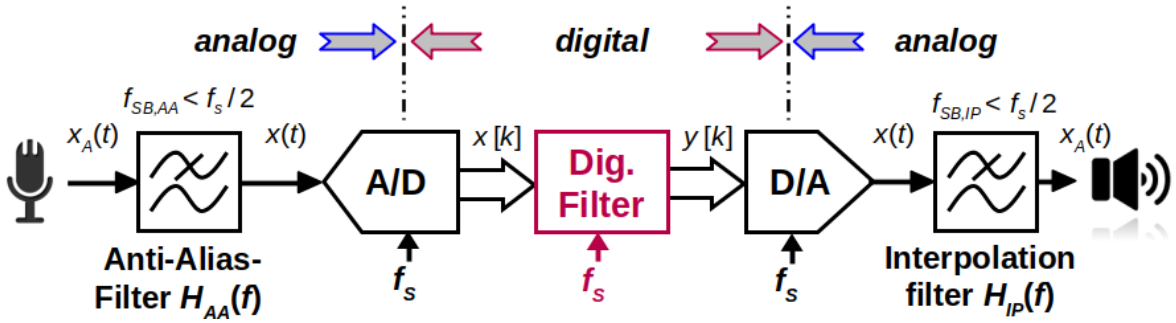

Fig. 2.4: A simple signal processing system

Sometimes it makes sense to change the sampling frequency in the processing system e.g. to reduce the sampling rate of an oversampling ADC or to increase the clocking frequency of an DAC to ease and improve reconstruction of the analog signal.

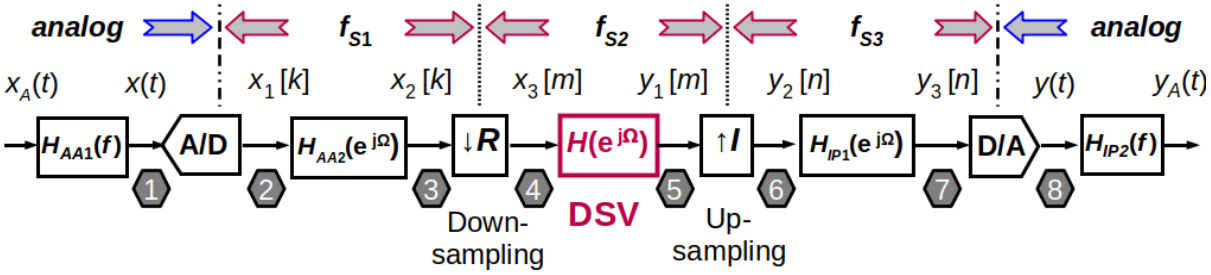

Fig. 2.5: A signal processing system with multiple sampling frequencies

Aliasing and Nyquist Frequency

When the sampling frequency is too low, significant information is lost in the process and the signal cannot be reconstructed without errors (forth image in Fig. 2.6) [Smith99]. This effect is called *aliasing*.

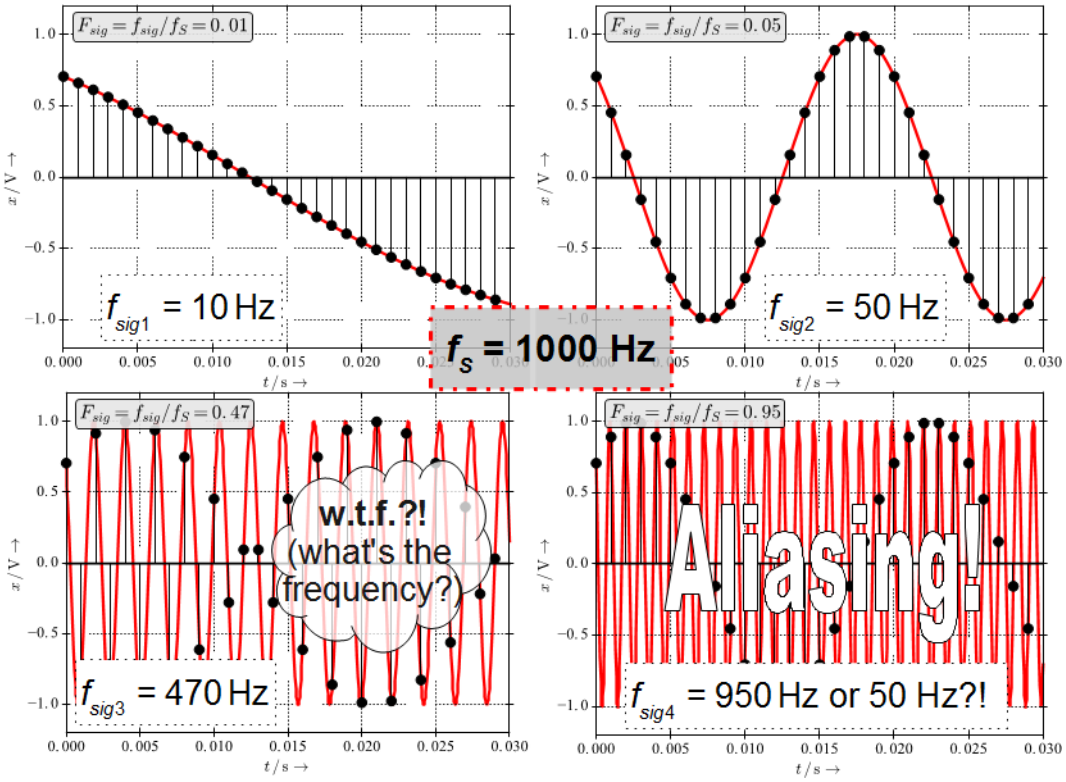

Fig. 2.6: Sampling with $f_S = 1000$ Hz of sinusoids with 4 different frequencies

When sampling with f_S, the maximum signal bandwidth B that can represented and reconstructed without errors is given by $B < f_S/2 = f_{Ny}$. This is also called the *Nyquist frequency* or *bandwidth* f_{Ny}. Some filter design tools and algorithms normalize frequencies w.r.t. to f_{Ny} instead of f_S.

Half-Band Filters

Explanation of half-band filters and how to design them with pyfda can be found at [Half-Band Filters, a Workhorse of Decimation Filters] (https://tomverbeure.github.io/2020/12/15/Half-Band-Filters-A-Workhorse-of-Decimation-Filters.html#designing-a-half-band-fir-filter-with-pyfda)

Development

More info on this widget can be found under dev_input_specs.

2.1.2 Input Coeffs

Fig. 2.7 shows a typical view of the **b,a** tab where you can view and edit the filter coefficients. Coefficient values are updated every time you design a new filter or update the poles / zeros.

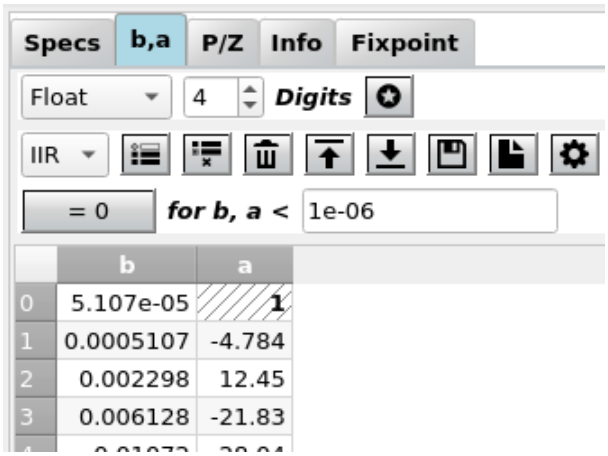

Fig. 2.7: Screenshot of the coefficients tab for floating point coefficients

In the top row, the display of the coefficients can be disabled as a coefficient update can be time consuming for high order filters ($N > 100$).

Quantization format

By default, coefficients are displayed in float quantization format, the format returned by the filter design algorithm, with a selectable number of decimal places. Internally, full precision is always used.

However, many hardware platforms with limited computing resources like uCs can only perform fixpoint arithmetics. Here, scaling and wordlength have a strong influence on the obtainable accuracy.

It is important to understand that the quantization format only influences the *display* of the coefficients, the frequency response etc. is only updated when the quantize icon (the staircase) is clicked. Only when you do a *fixpoint simulation* or generate Verilog code from the fixpoint tab, the selected word format is used for the coefficients.

Fixpoint

When the format is set to fractional or integer, the fixpoint options are displayed as in Fig. 2.8. Here, the format *Binary* has been set.

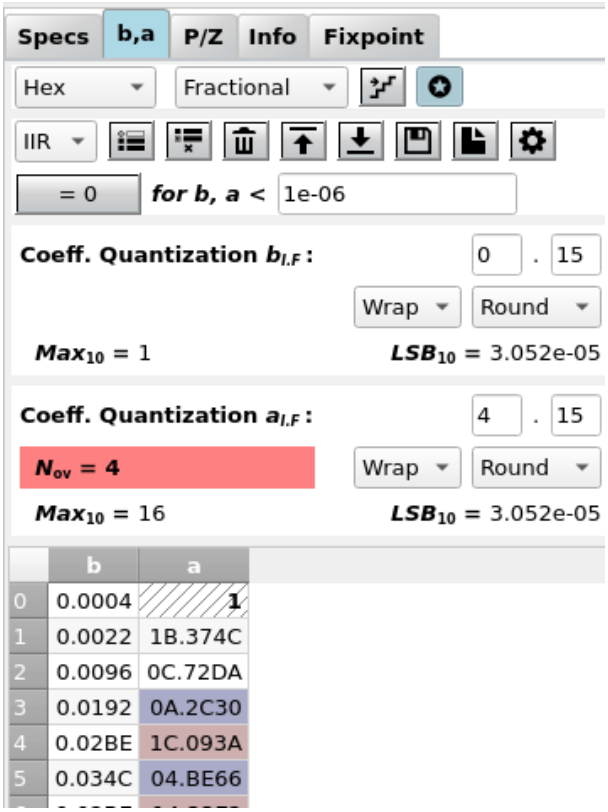

Fig. 2.8: Screenshot of the coefficients tab for fixpoint formats (binary display)

Fixpoint Formats

Any other format (Binary, Hex, Decimal, CSD) is a fixpoint format with a fixed number of binary places which activates further display options. These formats (except for CSD) are based on the integer value i.e. by simply interpreting the bits as an integer value `INT` with the MSB as the sign bit.

The scale between floating ("Real World Value", RWV) and fixpoint format is determined by partitioning the word length `W` into integer and fractional places `WI` and `WF` with total word length `W = WI + WF + 1` where the "`+ 1`" accounts for the sign bit.

Three kinds of partioning can be selected in a combo box:

- **The integer format has no fractional bits, `WF = 0` and**
  `W = WI + 1`. This is the format used by amaranth as well, `RWV = INT`
- **The normalized fractional format has no integer bits, `WI = 0` and**
  `W = WF + 1`.
- **The general fractional format has an arbitrary number of fractional**
  and integer bits, `W = WI + WF + 1`.

In any case, scaling is determined by the number of fractional bits:

$$RWV = INT \cdot 2^{-WF}$$

In addition to setting the position of the binary point you can select the behaviour for:

- **Quantization: The very high precision of the floating point format**
  needs to be reduced for the fixpoint representation. Here you can select between `floor` (truncate the LSBs), `round` (classical rounding) and `fix` (always round to the next smallest magnitude value)
- **Saturation: When the floating point number is outside the range of**
  the fixpoint format, either two's complement overflow occurs (`wrap`) or the value is clipped to the maximum resp. minimum ("saturation", `sat`)

More info on fixpoint arithmetics can be found under *Fixpoint Arithmetics*.

Development

More info on this widget can be found under dev_input_coeffs.

2.1.3 Input P/Z

Fig. 2.9 shows a typical view of the **P/Z** tab where you can view and edit the filter poles and zeros. Pole / zero values are updated every time you design a new filter. After editing poles or zeros by hand, the changes have to be applied via the (highlighted) button "Apply P/Z to filter".

In real-valued systems (i.e. systems with a real-valued impulse response and real-valued coefficients) poles and zeros are real-valued or come in conjugate complex pairs. This means they have the same real part and positive / negative imaginary part, e.g. $p_1 = 0.5 + 0.5j$ and $p_2 = 0.5 - 0.5j$ or $z_1 = 1\angle + 0.25\pi$ and $z_2 = 1\angle - 0.25\pi$. Otherwise, you end up with a complex-valued system with complex-valued coefficients which is not what you want in most cases.

Cartesian format

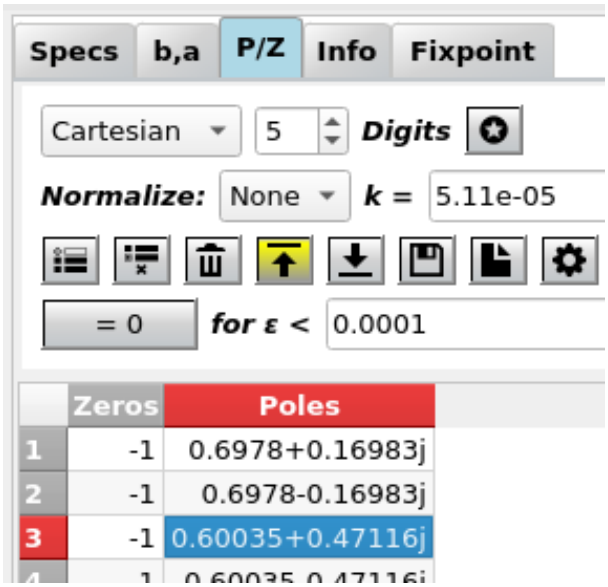

Fig. 2.9: Screenshot of the pole/zero tab in cartesian format

Poles and zeros are displayed and can be edited in cartesian format (x and y) by default as shown in Fig. 2.9.

Polar format

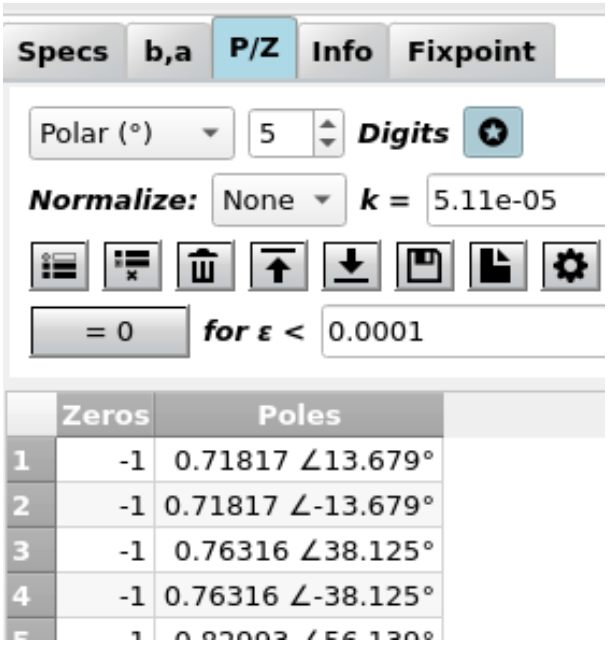

Fig. 2.10: Screenshot of the pole/zero tab in polar format with activated "Format" button

Alternatively, poles and zeros can be displayed and edited in polar format (radius and angle) as shown in Fig. 2.10. Especially for zeros which often are placed on the unit circle ($r = 1$) this format may be more suitable.

During editing, use the '>' character to separate radius and phase. The phase can be displayed and entered in the following formats:

- **Degrees** with a range of $\pm -180 \ldots +180$, terminate the phase with an 'o' or '°' to indicate degrees.
- **Rad** with a range of $\pm -\pi \ldots +\pi$, simply enter the value or terminate the phase with an 'r' or with 'rad' to indicate rads.
- Multiples of **pi** with a range of $\pm -1 \ldots +1$, terminate the phase with a 'p' or 'pi' to specifiy multiples of pi.

When entering poles or zeros, the format is chosen automatically, depending on which special characters (like '<', 'o', 'r' or 'pi') have been found in the text field.

You can "misuse" this feature as a converter between different number formats:

- '3<0.7854' or '3<0.7854r' or '3<0.7854rad'
- '3<0.25p' or '3<0.25pi'
- '3<45°' or '3<45o'
- 2.12132+2.12132j

all represent the same value. You can omit the radius if $r = 1$, simply enter '<45°' instead of '1<45°'.

Use the corresponding icons to enter a new row or delete one. The trash can deletes the whole table.

Saving and Loading

Poles and zeros can be saved in various file formats (CSV, MAT, NPZ, NPY). CSV file format options (row or column, delimiter, ...) are selected in the CSV pop-up menu (the 'cog' icon). Independent of the table display format, coefficients are saved with full precision in complex (cartesian) number format when the format button (the "star") is deactivated.

When the format button *is* activated, values are saved *exactly as displayed*. This means, cells may be saved with reduced number of digits and in polar number format, containing special characters like '<'.

Development

More info on this widget can be found under dev_input_pz.

2.1.4 Input Info

The **Info** tab (Fig. 2.11) displays infos on the current filter design and design algorithm.

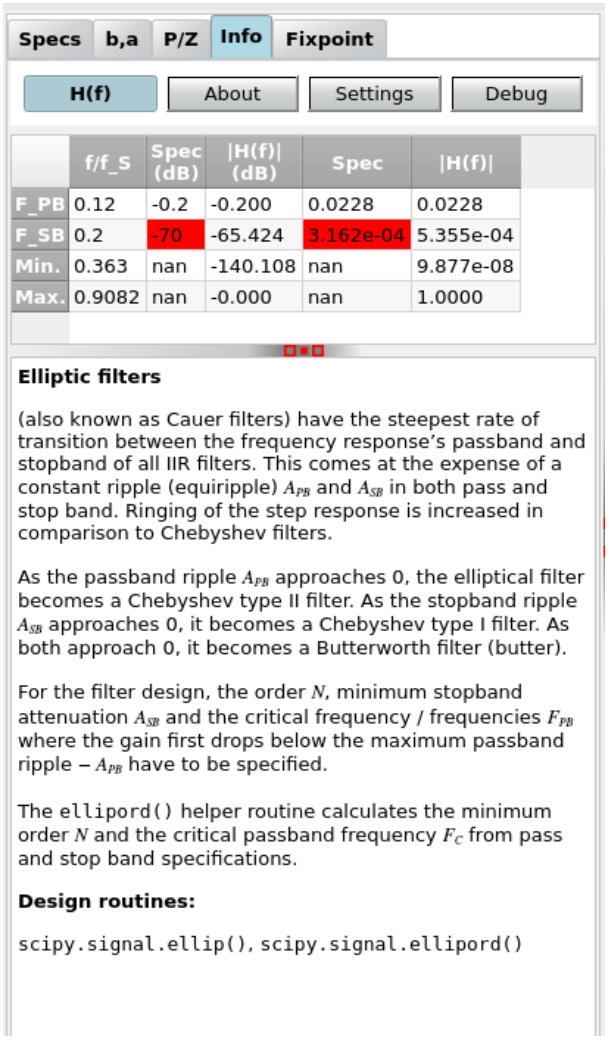

Fig. 2.11: Screenshot of the info tab

The buttons in the top row select which information is displayed:

The **H(f)** button activates the display of specifications in the frequency domain and how well they are met. Failed specifications are highlighted in red.

The **About** button opens a pop-up window with general infos about the software, licensing and module versions (Fig. 2.12).

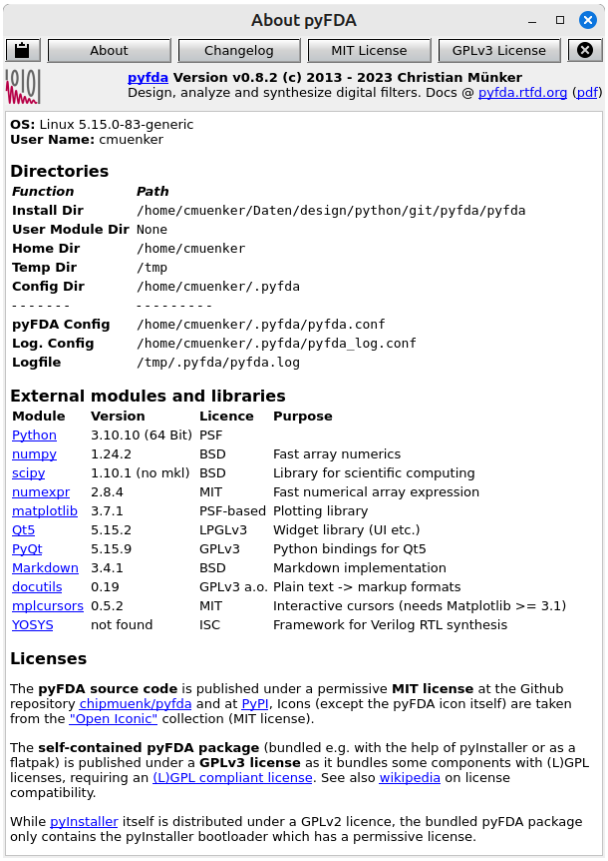

Fig. 2.12: Screenshot of the “About” pop-up window

The **Debug** button enables some debugging options:

- **Doc$**: Show docstring info from the corresponding python (usually scipy) module.
- **RTF**: Use Rich Text Format for documentation.
- **FiltDict**: Display the dictionary containing all current settings of the software. This dictionary is saved and restored when saving / loading a filter.
- **FiltTree**: Display the hierarchical tree with all filter widgets that have been detected during the start of the software

Development

More info on this widget can be found under dev_input_info.

2.1.5 Fixpoint Specs

Overview

The **Fixpoint** tab (Fig. 2.13) provides options for generating and simulating discrete-time filters that can be implemented in hardware. Hardware implementations for discrete-time filters usually imply fixpoint arithmetics but this could change in the future as floating point arithmetics can be implemented on FPGAs using dedicated floating point units (FPUs).

Order and the coefficients have been calculated by a filter design algorithm from the *pyfda.filter_widgets* package to meet target filter specifications (usually in the frequency domain).

In this tab, a fixpoint implementation can be selected in the upper left corner (fixpoint filter implementations are available only for a few filter design algorithms at the moment, most notably IIR filters are missing).

The fixpoint format of input word Q_X and output word Q_Y can be adjusted for all fixpoint filters, pressing the "lock" button makes the format of input and output word identical. Depending on the fixpoint filter, other formats (coefficients, accumulator) can be set as well.

In general, **Ovfl.** combo boxes determine overflow behaviour (Two's complement wrap around or saturation), **Quant.** combo boxes select quantization behaviour between rounding, truncation ("floor") or round-towards-zero ("fix"). These methods may not all be implemented for each fixpoint filter. Truncation is easiest to implement but has an average bias of -1/2 LSB, in contrast, rounding has no bias but requires an additional adder. Only rounding-towards-zero guarantees that the magnitude of the rounded number is not larger than the input, thus preventing limit cycles in recursive filters.

See also [Yates_2020] and [Lyons]

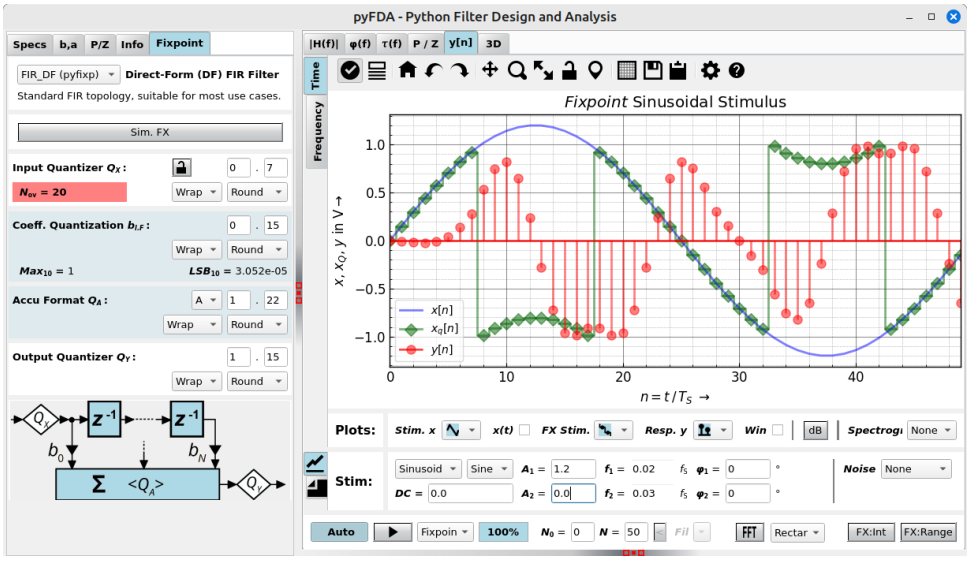

Fig. 2.13: Fixpoint parameter entry widget (overflow = wrap)

Typical simulation results are shown in Fig. 2.14 (time domain) and Fig. 2.15 (frequency domain).

Fixpoint filters are inherently non-linear due to quantization and saturation effects, that's why frequency characteristics can only be derived by running a transient simulation and calculating the Fourier response afterwards:

The following shows an example of a coefficient in Q2.4 and Q0.3 format using wrap-around and truncation. It's easy to see that for simple wrap-around logic, the sign of the result may change.

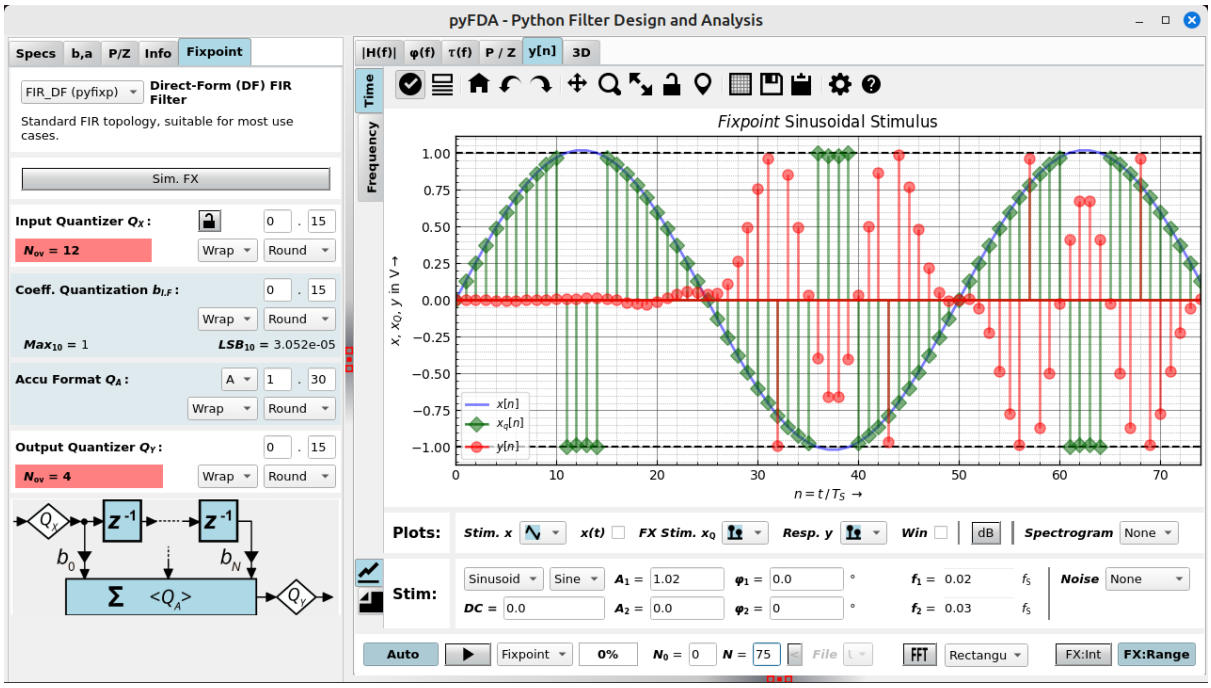

Fig. 2.14: Fixpoint simulation results (time domain)

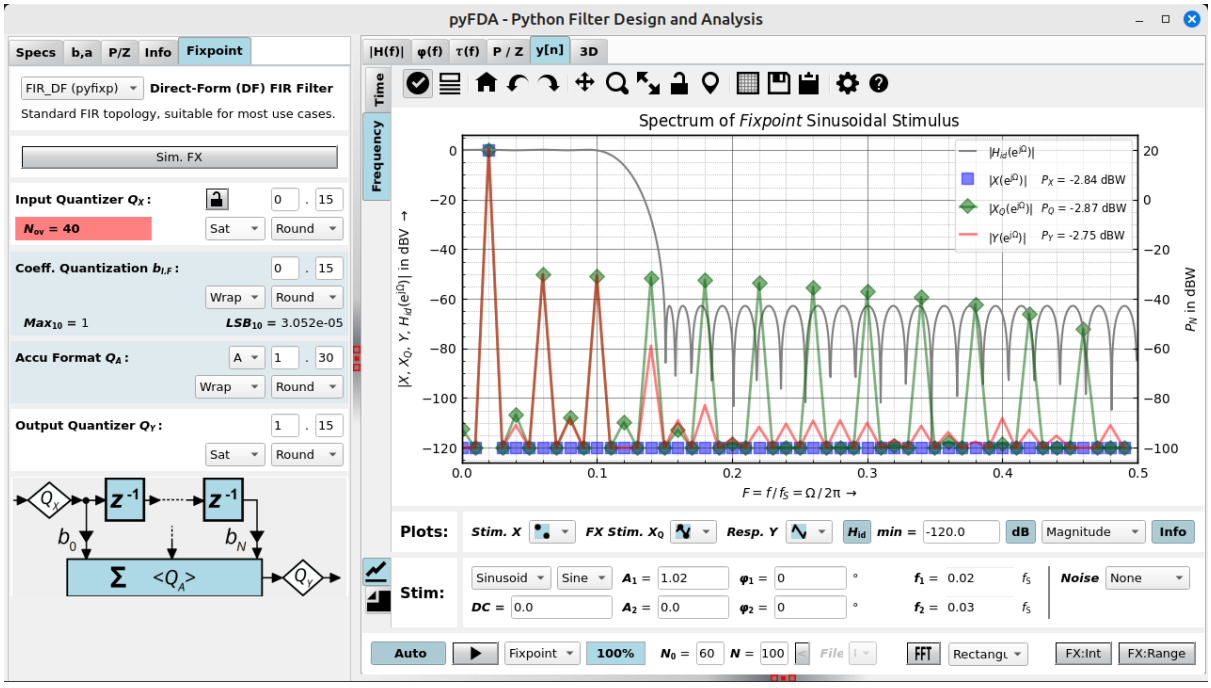

Fig. 2.15: Fixpoint simulation results (frequency domain)

```
S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  0  .  1  |  0  |  1  |  1   =  43 (INT) or 43/16 = 2 + 11/16␣
↪(RWV)
                .
          |  S  . WF0 | WF1 | WF2       :  WI = 0, WF = 3, W = 4
             0  .  1  |  0  |  1        =  5 (INT) or 5/8 (RWV)
```

Summation

Before adding two fixpoint numbers with a different number of integer and/or fractional bits, integer and fractional word lenghts need to equalized:

- the fractional parts are padded with zeros
- the integer parts need to be sign extended, i.e. with zeros for positive numbers and with ones for negative numbers
- adding numbers can require additional integer places due to word growth

For this reason, the position of the binary point needs to be

```
S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  0  .  1  |  0  |  1  |  1   =  43 (INT) or 43/16 = 2 + 11/16␣
↪(RWV)

                +

S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  0  |  0  .  1  |  0  |  1  |  0   =  10 (INT) or 10/16 (RWV)

=================================================================

S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  1  .  0  |  1  |  0  |  1   =  53 (INT) or 53/16 = 3 + 5/16 (RWV)
```

More info on fixpoint numbers and arithmetics can be found under *Fixpoint Arithmetics*.

Configuration

The configuration file `pyfda.conf` lists the fixpoint classes to be used, e.g. `DF1` and `DF2`. *`pyfda.libs.tree_builder.Tree_Builder`* parses this file and writes all fixpoint modules into the list `fb.fixpoint_widgets_list`. The input widget `pyfda.input_widgets.input_fixpoint_specs.Input_Fixpoint_Specs` constructs a combo box from this list with references to all successfully imported fixpoint modules. The currently selected fixpoint widget (e.g. `DF1`) is imported from *`pyfda.fixpoint_widgets`* together with the referenced image.

Development

More info on this widget can be found under dev_input_fixpoint_specs.

The subwidgets on the right-hand side allow for graphical analyses of the system.

2.1.6 Subwindow for Plotting Widgets

Plot H(f)

Fig. 2.16 shows a typical view of the **|H(f)|** tab for plotting the magnitude frequency response.

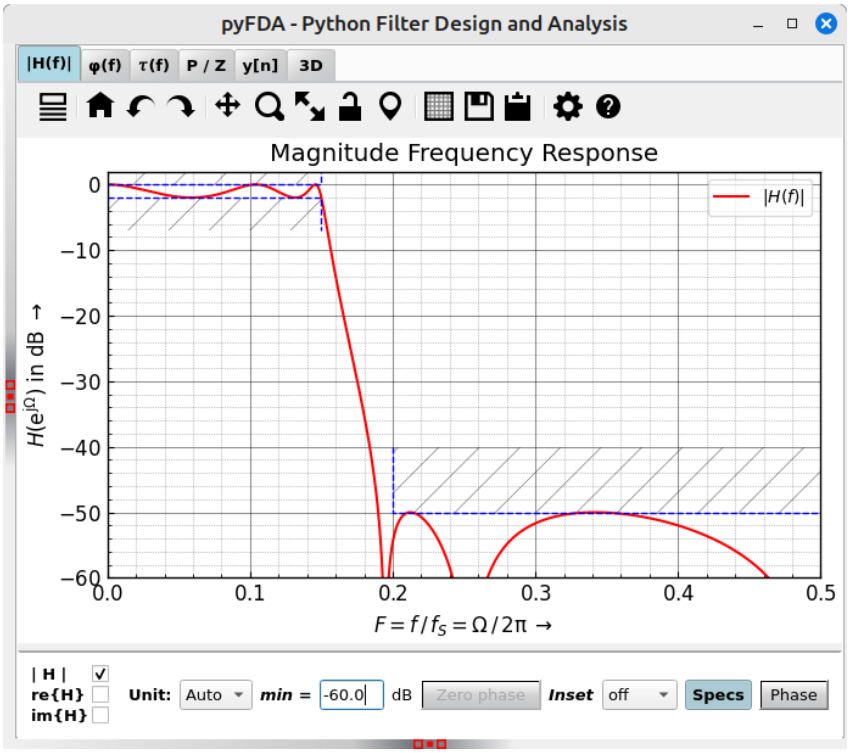

Fig. 2.16: Screenshot of the *|H(f)|* tab

You can plot magnitude, real or imaginary part in V (linear), W (squared) or dB (log. scale).

Zero phase removes the linear phase as calculated from the filter order. There is no check whether the design actually is linear phase, that's why results may be nonsensical. When the unit is dB or W , this option makes no sense and is not available. It also makes no sense when the magnitude of H(f) is plotted, but it might be interesting to look at the resulting phase.

Depending on the **Inset** combo box, a small inset plot of the frequency reponse is displayed, changes of zoom, unit etc. only have an influence on the main plot ("fixed") or the inset plot ("edit"). This way, you can e.g. zoom into pass band and stop band in the same plot. The handling still has some rough edges.

Show specs displays the specifications; the display makes little sense when re(H) or im(H) is plotted.

Phase overlays a plot of the phase, the unit can be set in the phase tab.

Development

More info on this widget can be found under dev_plot_hf.

Plot Phi(f)

Fig. 2.17 shows a typical view of the $\varphi(f)$ tab for plotting the phase response of an elliptical filter (IIR).

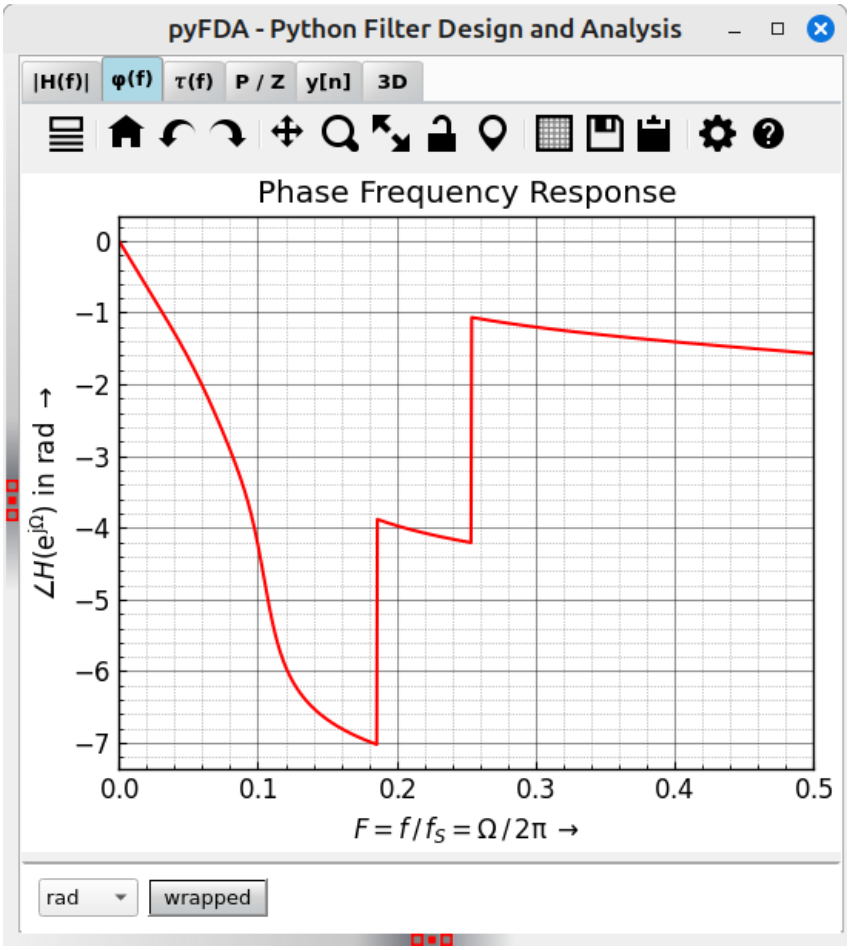

Fig. 2.17: Screenshot of the $\varphi(f)$ tab

You can select the unit for the phase and whether the phase will be wrapped between $-\pi \ldots \pi$ or not.

Development

More info on this widget can be found under dev_plot_phi.

Plot tau(f)

Fig. 2.18 shows a typical view of the $\tau(f)$ tab for plotting the group delay, here, an elliptical filter (IIR) is shown.

There are no user servicable parts on this tab.

The algorithm for calculating the group delay is explained in detail in
`pyfda.libs.pyfda_sig_lib.group_delay()`.

Show `group_delay()`

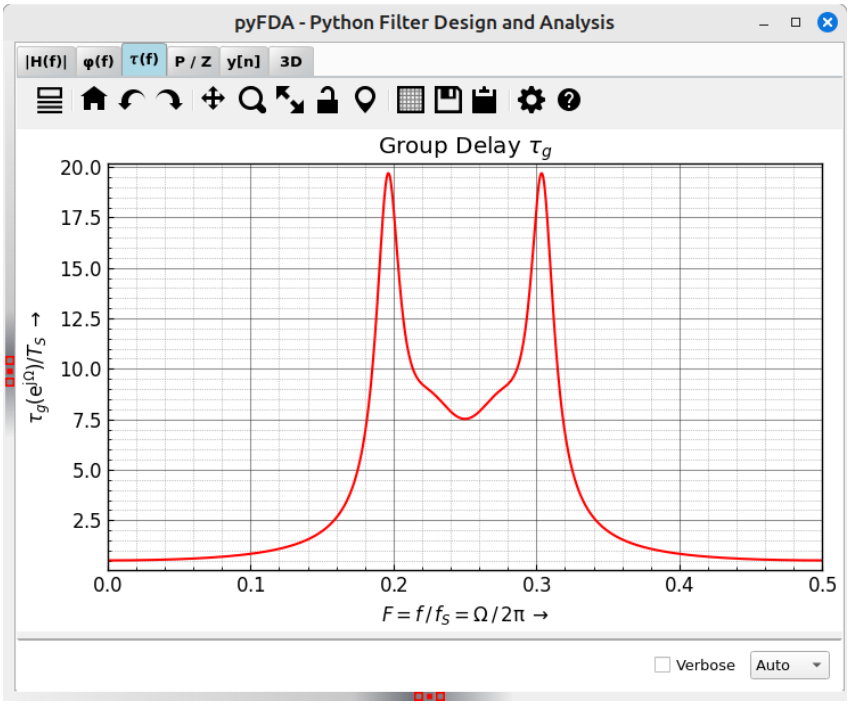

Fig. 2.18: Screenshot of the $\tau(f)$ tab

Development

More info on this widget can be found under dev_plot_tau_g.

Plot P/Z

Fig. 2.19 shows a typical view of the **P/Z** tab for plotting poles and zeros, here, an elliptical filter (IIR) is shown.

Optionally, the magnitude frequency response can be plotted around the unit circle to show the influence of poles and zeros (Fig. 2.20).

Development

More info on this widget can be found under dev_plot_pz.

Plot y[n]

Fig. 2.21 shows a typical view of the **y[n]** tab for plotting the transient response and its Fourier transformation, here, for a Chebychev filter (IIR).

This tab is split into several subwindows:

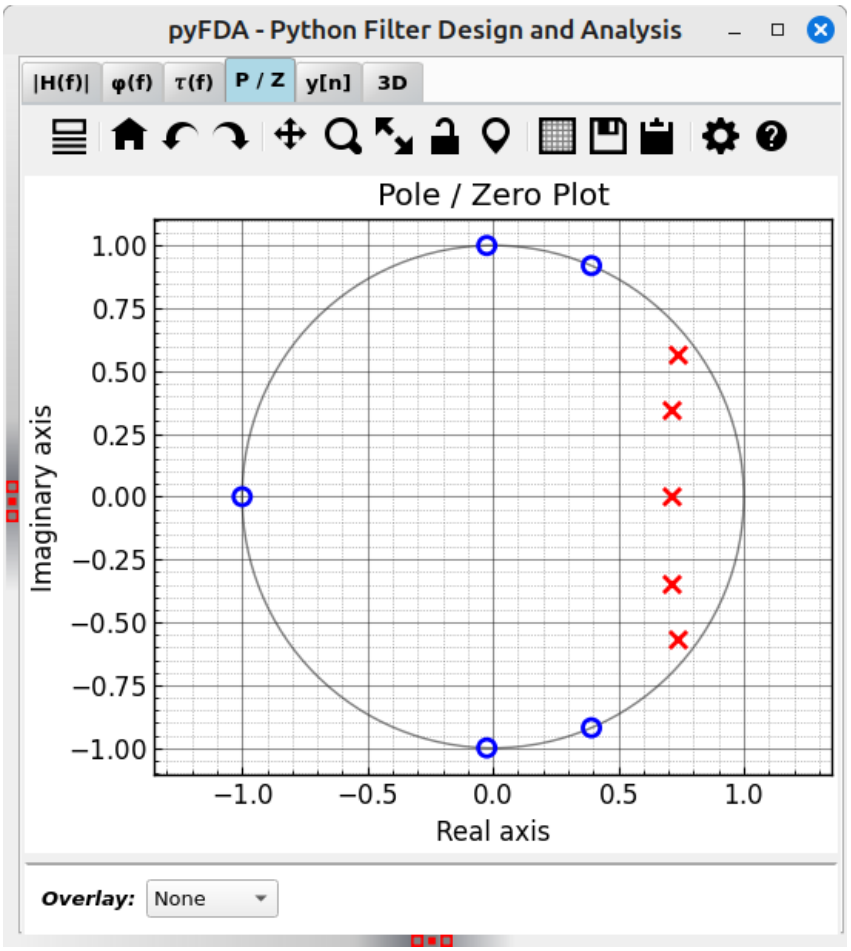

Fig. 2.19: Screenshot of the P/Z tab

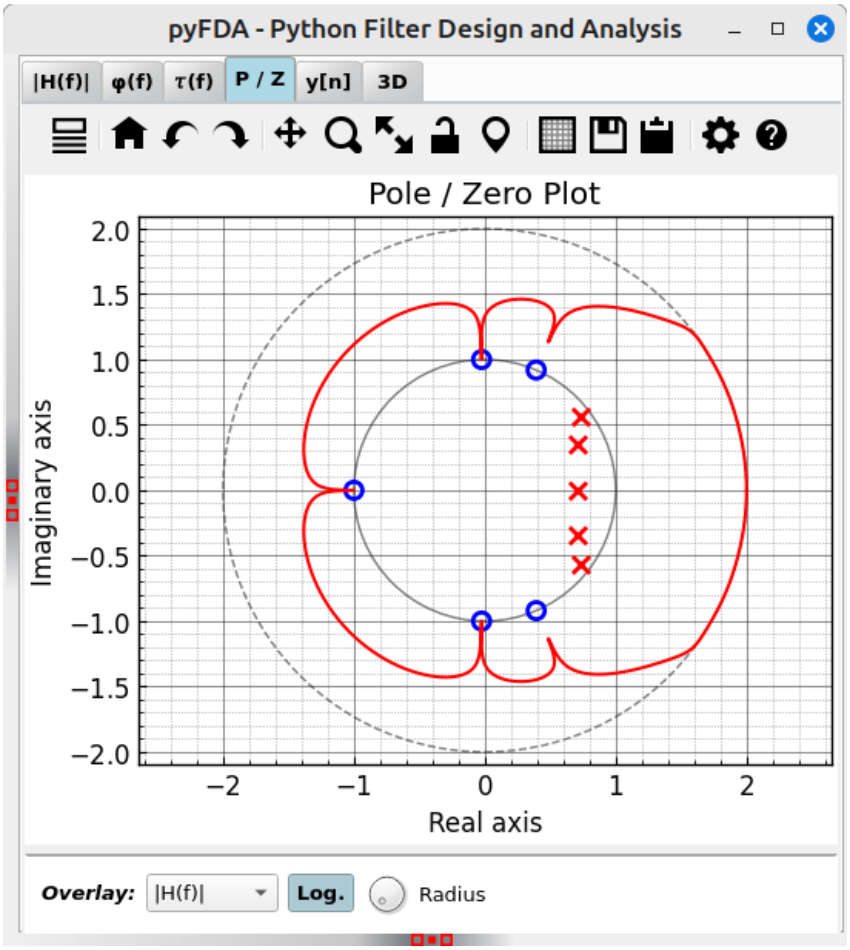

Fig. 2.20: Screenshot of the P/Z tab with overlayed H(f) plot

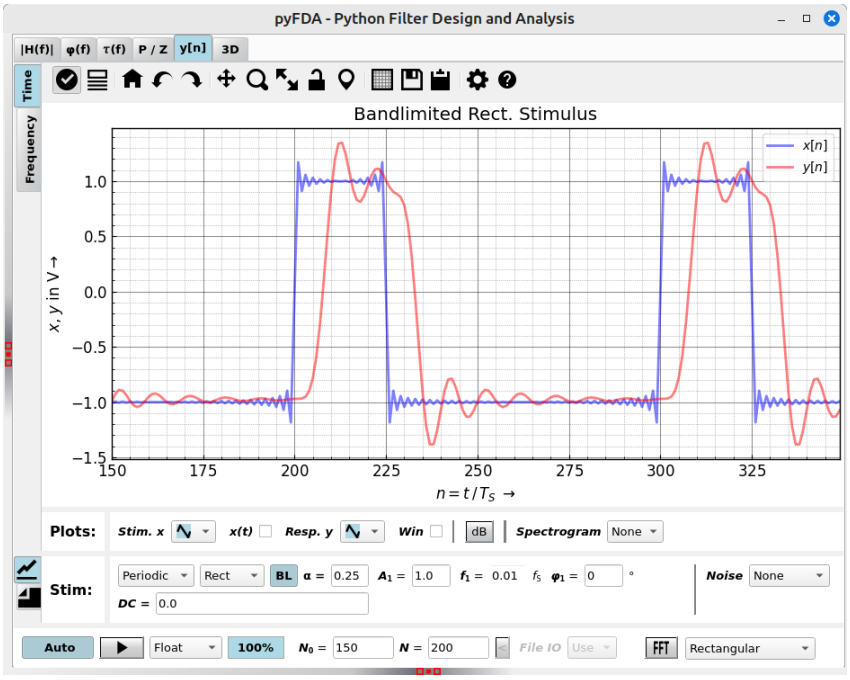

Fig. 2.21: Screenshot of the y[n] tab (time domain)

Time / Frequency (main plotting area)

These vertical tabs select between the time (transient) and frequency (spectral) domain. Signals are calculated in the time domain and then transformed using Fourier transform.

Time

Frequency

The Fourier transform of the transient signal can be viewed in the vertical tab "Frequency" (Fig. 2.22). This is especially important for fixpoint simulations where the frequency response cannot be calculated analytically.

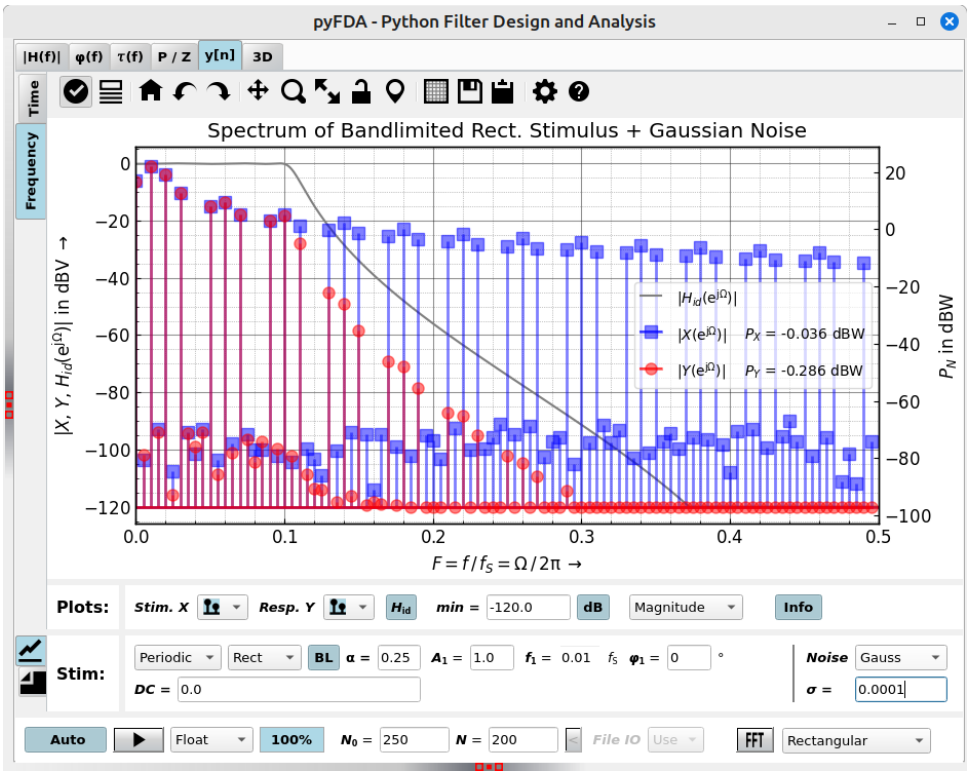

Fig. 2.22: Screenshot of the y[n] tab (frequency domain)

For an transform of periodic signals without leakage effect, ("smeared" spectral lines) take care that:

- The filter has settled sufficiently. Select a suitable value of **N0**.
- Choose the number of data points **N** in such a way that an integer number of periods is displayed (and transformed).
- The FFT window is set to rectangular. Other windows work as well but they distribute spectral lines over several bins. When it is not possible to capture an integer number of periods, use another window as the rectangular window has the worst leakage effect.

Plots

What will be plotted and how.

Stim.

Select the stimulus, its frequency, DC-content, noise … When the BL checkbox is checked, the signal is bandlimited to the Nyquist frequency. Some signals have strong harmonic content which produces aliasing. This can be seen best in the frequency domain (e.g. for a sawtooth signal with f = 0.15).

DC and Different sorts of noise can be added.

Run

Usually, plots are updated as soon as an option has been changed. This can be disabled with the **Auto** checkbox for cases where the simulation takes a long time (e.g. for some fixpoint simulations).

Development

More info on this widget can be found under dev_plot_impz.

Plot 3D

Fig. 2.23 shows a typical view of the **3D** tab for 3D visualizations of the magnitude frequency response and poles / zeros. Fig. 2.23 is a surface plot which looks nice but takes the longest time to compute.

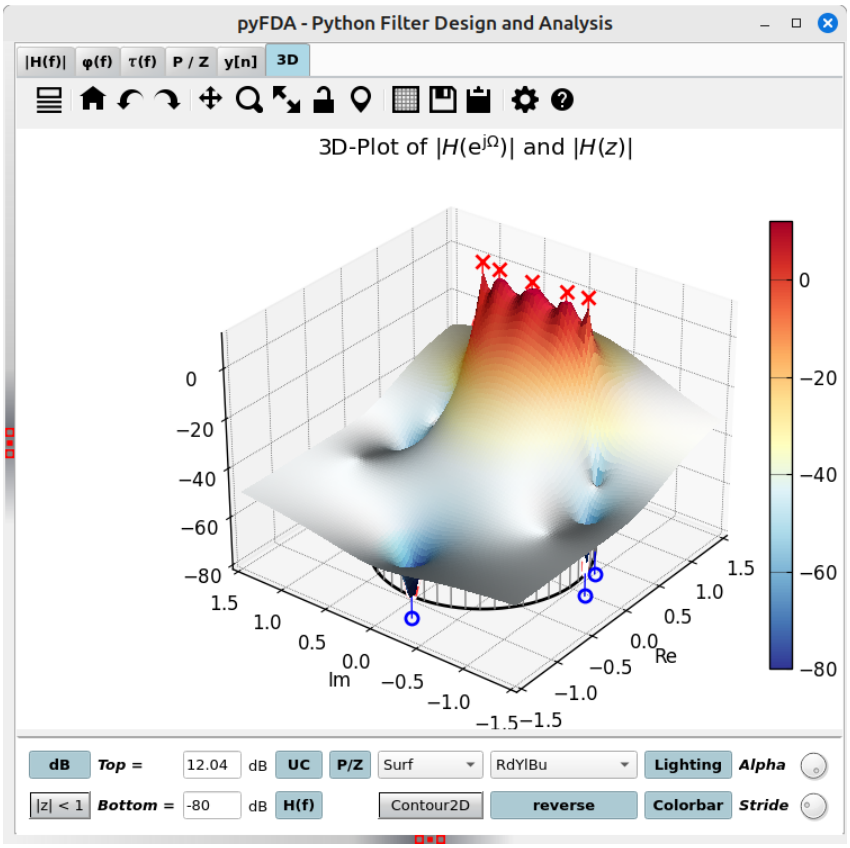

Fig. 2.23: Screenshot of the 3D tab (surface plot)

You can plot 3D visualizations of $|H(z)|$ as well as $|H(e^{j\omega})|$ along the unit circle (UC).

For faster visualizations, start with a mesh plot (Fig. 2.24) or a contour plot and switch to a surface plot when you are pleased with scale and view.

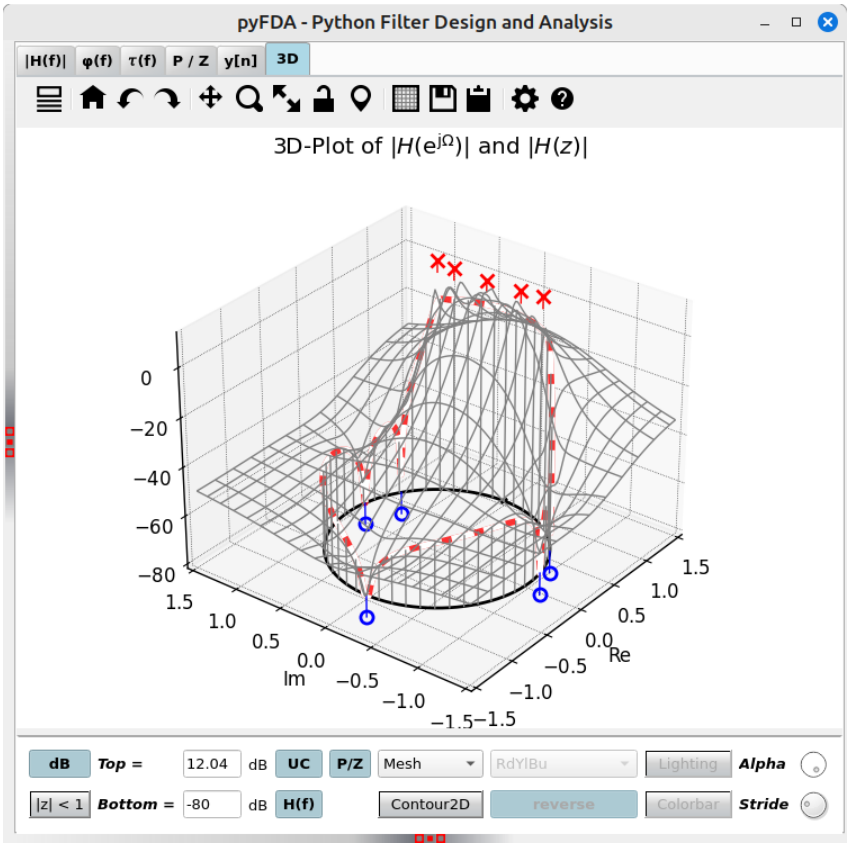

Fig. 2.24: Screenshot of the 3D tab (mesh plot)

Development

More info on this widget can be found under dev_plot_3d.

Some documentation treats general filter design and fixpoint arithmetics stuff.

General Documentation

Fixpoint Arithmetics

Overview

In contrast to floating point numbers, **fixpoint** numbers have a fixed scaling, requiring more care to avoid over- or underflows. The same binary word can represent an integer (Fig. 2.25) or a fractional (Fig. 2.26) number, signed or unsigned. The position of the binary point and whether the MSB represents the sign bit or not, it is all in the designer's head ...

The fixpoint format of input word Q_X and output word Q_Y can be adjusted for all fixpoint filters, pressing the "lock" button makes the format of input and output word identical. Depending on the fixpoint filter, other formats (coefficients, accumulator) can be set as well.

In general, **Ovfl.** combo boxes determine overflow behaviour (Two's complement wrap around or saturation), **Quant.** combo boxes select quantization behaviour between rounding, truncation ("floor") or round-towards-zero ("fix"). These methods may not all be implemented for each fixpoint filter. Truncation is easiest to

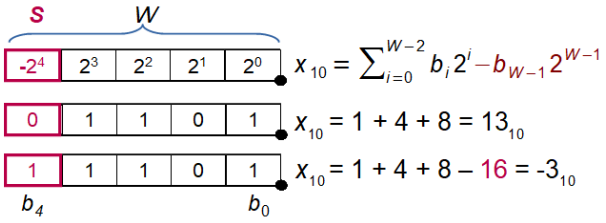

Fig. 2.25: Signed integer number in twos-complement format

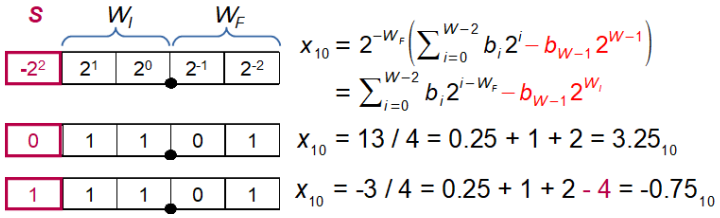

Fig. 2.26: Signed fractional number in twos-complement format

implement but has an average bias of -1/2 LSB, in contrast, rounding has no bias but requires an additional adder. Only rounding-towards-zero guarantees that the magnitude of the rounded number is not larger than the input, thus preventing limit cycles in recursive filters.

Typical simulation results are shown in Fig. 2.27, where first the input signal exceeds the numeric range and then the output signal. The overflow behaviour is set to 'wrap', resulting in twos-complement wrap around with changes in the sign.

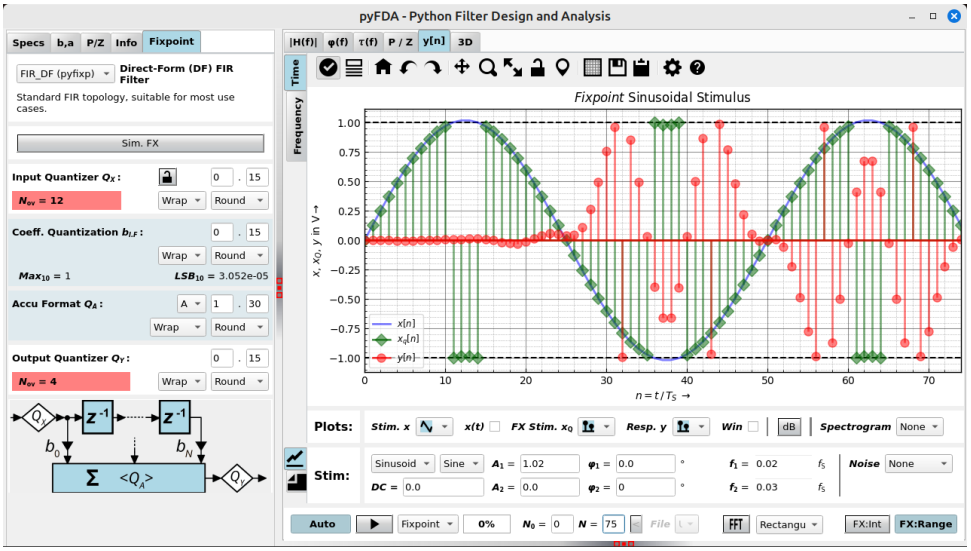

Fig. 2.27: Fixpoint filter response with overflows

Sign extension

When increasing the number of integer bits, numbers need to be sign extended, i.e. the new leading bits need to be filled with the sign bit (Fig. 2.28). Extending the number of fractional bits just requires zero padding.

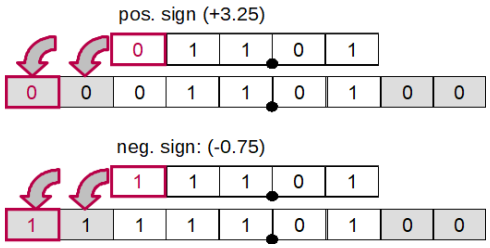

Fig. 2.28: Sign extension of integer and fractional numbers

Overflow behaviour

After summation or when reducing the number of integer bits, the result may not fit in the numeric range.

Discarding one or more leading bits to obtain the desired wordlength is easy but may produce wrap-arounds. The resulting sign changes can introduce instability and limit-cycle oscillations to the system (Fig. 2.29, left-hand side).

Saturation (Fig. 2.29, right-hand side) is much more benign but requires a little more effort: Before adding two numbers, both need to be sign extended by one bit to enable overflow detection. As shown in Fig. 2.29, when the two leading bits (sign and carry) are *01* or *10*, the result exceeds the numeric range and needs to be replaced by the maximum resp. minimum representable value. When reducing the number of integer bits, similar checks need to be performed to test for overflows.

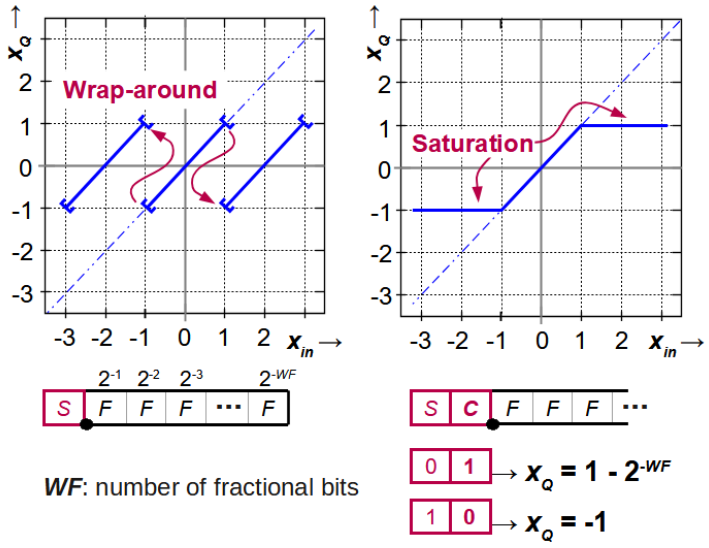

Fig. 2.29: Overflow behaviour with wrap-around or saturation

Truncation and rounding

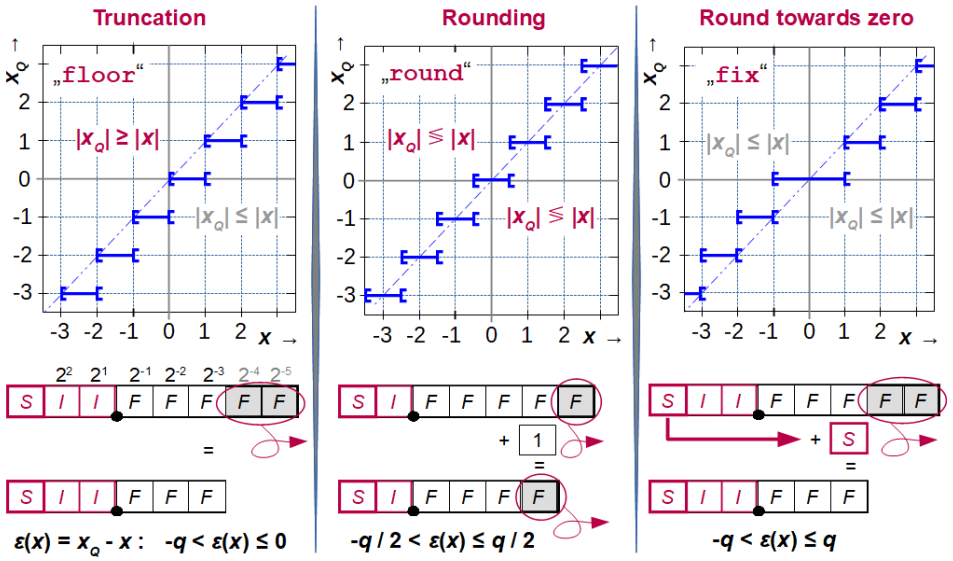

Fig. 2.30: Reducing fractional word length using truncation, rounding and round-towards-zero

The following shows an example of a positive number in Q2.4 that is converted to Q1.3 format using truncation. It's easy to see that for simple wrap-around logic, the sign of the result may change.

```
S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  0  .  1  |  0  |  1  |  1   =  43 (QINT) or 43/16 = 2 + 11/16 (QFRAC)
                |
                v
    S   | WI0 . WF0 | WF1 | WF2        :  WI = 1, WF = 3, W = 5
    1     0  .  1  |  0  |  1         = -32 + 21 = -11 (subtract -2Ŵ for sign␣
↪bit)
                                       = -16 + 5  = -11 (sign bit as -2^(W -1) )
                                       or -2 + 5/8 = -11 / 8
```

Summation

Before adding two fixpoint numbers with a different number of integer and/or fractional bits, integer and fractional word lenghts need to equalized:

- the fractional parts are padded with zeros
- the integer parts need to be sign extended, i.e. with zeros for positive numbers and with ones for negative numbers
- adding numbers can require additional integer places due to word growth

For this reason, the position of the binary point needs to be respected when summing fixpoint numbers.

```
S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  0  .  1  |  0  |  1  |  1   =  43 (INT) or 43/16 = 2 + 11/16 (RWV)

                +

S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  0  |  0  .  1  |  0  |  1  |  0   =  10 (INT) or 10/16 (RWV)
```

(continues on next page)

(continued from previous page)

```
              =

S | WI1 | WI0 . WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  1  .  0  |  1  |  0  |  1   =  53 (INT) or 53/16 = 3 + 5/16 (RWV)
```

Products

2.1.7 Logger Subwindow

The logging window in the lower part of the plotting window can be resized or completely closed. Its content can be selected, copied or cleared with a right mouse button context menu.

2.2 Customization

You can customize pyfda behaviour in some configuration files:

2.2.1 pyfda.conf

A copy of `pyfda/pyfda.conf` is created in `<USER_HOME>/.pyfda/pyfda.conf` where it can be edited by the user to choose which widgets and filters will be included. Fixpoint widgets can be assigned to filter designs and one or more user directories can be defined if you want to develop and integrate your own widgets (it's not so hard!):

```
# This file configures filters and plotting routines for pyFDA
# ----------------------------------------------------------------------
# - Encoding should be either UTF-8 without BOM or standard ASCII
# - All lines starting with # or ; are regarded as comments,
#   inline comments are not allowed
# - [Section] starts a new section
# - Options and values are separated by a ":" or "=" (e.g. dir1 : /home),
#   values are optional
# - Values are "sanitized" by removing [], ' and "
# - Values are split at commas, semicolons or CRs into a list of values
# - Values starting with a { are converted to a dict
# - "Interpolation" i.e. referencing values within the config file via e.g. ${dir1}
#   or ${Common:user_dir1} can be used


###################
[Common]
###################
# Stop pyfda when the parsed conf file has a lower version than required

version = 4

#-------------------------------------------------------------------------------
# Define variables than can be referenced in other sections by preceding the
# section name, e.g. fir_df1 = ${Common:FIR} is resolved to
# fir_df1 = [Equiripple, Firwin, Manual, MA]
#-------------------------------------------------------------------------------

#
IIR = [Bessel, Butter, Cheby1, Cheby2, Ellip]
```

(continues on next page)

(continued from previous page)

```
FIR = [Equiripple, Firwin, Manual, MA]

#------------------------------------------------------------------------------
# Add paths for special tools (optional):
#------------------------------------------------------------------------------
# yosys = "D:\Programme\yosys-win32-mxebin-0.9\yosys.exe"


#------------------------------------------------------------------------------
# Add user directory(s) to sys.path (optional):
#------------------------------------------------------------------------------
#
# Specify relative or absolute path(s) to one or more user directories. These
# directories are searched for the following subdirectories which must be named
# like the corresponding pyfda directories:
#
# input_widgets    # widgets for specifying filter parameters
# plot_widgets     # widgets for plotting filter properties
# filter_widgets   # widgets for controlling filter design algorithms
# fixpoint_widgets # widgets for specifying fixpoint filters
#
# These subdirectories need to contain an (usually empty)
# __init__.py file to be recognized as python modules.
#
# When a specified directory cannot be found, only a warning is issued.
#------------------------------------------------------------------------------
# Uncomment and specify your user directory (optional):
#
#user_dirs = "D:\Daten\design\python\git\pyfda\pyfda\widget_templates",
#             "/home/muenker/Daten/design/python/user_pyfda"

################################################################################
# The following sections define which classes will be imported by specifying
# the module names (= file names without .py suffix). The actual class names are
# obtained from a module level attribute "classes" in each module which can be a:
#
# - String, e.g. classes = "MyClassName"
# - List, e.g.   classes = ["MyClassName1", "MyClassName2"]
# - Dict, e.g.   classes = {"MyClassName1":"DisplayName1", "MyClassName2":
↪"DisplayName2"}
#
# When no display name is given, the class name is used for tab labels, combo boxes␣
↪etc.
#
# Modules are searched in all directories defined in sys.path and the user dir(s)
# and their subdirectories containing __init__.py files (subpackages) with the
# names listed above ("input_widgets" etc.)
#
# In addition to specifying only the module name, options can be passed as key-
# value combinations. Unknown options just raise a warning.


########################
[Input Widgets]
########################
# Try to import from the following input widget modules (files) from sys.path
```

(continues on next page)

(continued from previous page)

```
# and subdirectories / subpackages named "input_widgets".

input_specs
input_coeffs
input_pz
input_info
input_fixpoint_specs

########################
[Plot Widgets]
########################
# Try to import from the following plot widget modules (files) from sys.path
# and subdirectories / subpackages named "plot_widgets".

plot_hf : {'opt1':'aaa', 'opt2':'bbb'}
plot_phi
plot_tau_g
plot_pz
plot_impz
plot_3d
# myplot # this could be the name of your user module

########################
[Filter Widgets]
########################
# The specified filter design modules (files) are searched for in sys.path
# and in subdirectories / subpackages named "filter_widgets".
#
# The optional 'fix' argument defines one or more fixpoint implementations for
# the filter design. Unknown fixpoint implementations only raise a warning.
# In the "Fixpoint Widgets" section, fixpoint implementation can be assigned
# to filter designs as well.

# --- IIR ---
# super_filter : {'fix':['iir_cascade', 'iir_df1']}
# bessel : {'fix':['iir_cascade', 'iir_df1']}
bessel
butter
# cheby1 : "yet another option"
cheby1
# cheby2 : {'fix':'iir_special'}
cheby2
ellip
# ellip_zero # too specialized for general usage

# --- FIR ---
equiripple
firwin
ma
# delay # still buggy
# savitzky_golay # not implemented yet

# --- Manual (both FIR and IIR) ---
manual
```

(continues on next page)

(continued from previous page)

```
########################
[Fixpoint Widgets]
########################
# Try to import from the following fixpoint widget modules (files) from sys.path
# and subdirectories / subpackages named "fixpoint_widgets".
#
# Value is a filter design or a list of filter designs for which the fixpoint
# widget can be used.

fir_df.fir_df_pyfixp_ui = ${Common:FIR}
iir_df1.iir_df1_pyfixp_ui = ${Common:IIR}
# fir_df.fir_df_amaranth_ui = ${Common:FIR}
# fx_delay = ['Equiripple', 'Delay'] # need to fix fx_delay and Delay modules
```

2.2.2 pyfda_log.conf

A copy of `pyfda/pyfda_log.conf` is created in `<USER_HOME>/.pyfda/pyfda_log.conf` where it can be edited to control logging behaviour:

```
[loggers]
# List of loggers:
# - root logger has to be present
# - section name is "logger_" + name specified in the keys below. The logger
#   name is derived automatically in the files-to-be-logged from their
#  __name__ attribute (i.e. the file name without suffix)
# When a file doesn't exist (e.g. no_existo.py)
#
keys=root, pyfdax, pyfda_class, filter_factory, filterbroker,
     pyfda_lib, pyfda_sig_lib, pyfda_fix_lib, pyfda_qt_lib, pyfda_io_lib,
     pyfda_fft_windows_lib, tree_builder, csv_option_box,
     amplitude_specs, freq_specs, freq_units, input_coeffs, input_coeffs_ui,
     input_fixpoint_specs, input_info, input_pz, input_pz_ui, input_specs,
     input_tab_widgets, select_filter, target_specs,
     bessel, equiripple, firwin,
     fir_df_pyfixp, fir_df_pyfixp_ui, iir_df1_pyfixp, iir_df1_pyfixp_ui,
     mpl_widget, plot_3d, plot_fft_win, plot_hf, plot_impz, plot_impz_ui,
     plot_phi, plot_pz, plot_tab_widgets, plot_tau_g,
     plot_tran_stim, plot_tran_stim_ui, tran_io, tran_io_ui,
     no_existo

[handlers]
# List of handlers
keys=consoleHandler,fileHandler,QHandler

[formatters]
# List of formatters
keys=simpleFormatter,noDateFormatter,ezFormatter

# ============================================
[logger_root]
level=NOTSET
handlers=consoleHandler, QHandler

[logger_pyfdax]
level=INFO
```

(continues on next page)

(continued from previous page)

```
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.pyfdax
propagate=0

[logger_pyfda_class]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.pyfda_class
propagate=0

[logger_filter_factory]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.filter_factory
propagate=0

[logger_filterbroker]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.filterbroker
propagate=0

#-------------------- libs -------------------
[logger_pyfda_lib]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.pyfda_lib
propagate=0

[logger_pyfda_sig_lib]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.pyfda_sig_lib
propagate=0

[logger_pyfda_fix_lib]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.pyfda_fix_lib
propagate=0

[logger_pyfda_qt_lib]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.pyfda_qt_lib
propagate=0

[logger_pyfda_io_lib]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.pyfda_io_lib
propagate=0

[logger_pyfda_fft_windows_lib]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
```

(continues on next page)

(continued from previous page)

```
qualname=pyfda.libs.pyfda_fft_windows_lib
propagate=0

[logger_tree_builder]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.tree_builder
propagate=0

[logger_csv_option_box]
level=INFO
handlers=fileHandler,consoleHandler, QHandler
qualname=pyfda.libs.csv_option_box
propagate=0

#-------------------- input_widgets -------------------
[logger_amplitude_specs]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.amplitude_specs
propagate=0

[logger_freq_specs]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.freq_specs
propagate=0

[logger_freq_units]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.freq_units
propagate=0

[logger_input_coeffs]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_coeffs
propagate=0

[logger_input_coeffs_ui]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_coeffs
propagate=0

[logger_input_fixpoint_specs]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_fixpoint_specs
propagate=0

[logger_input_info]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_info
```

(continues on next page)

(continued from previous page)

```
propagate=0

[logger_input_pz]
level=WARNING
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_pz
propagate=0

[logger_input_pz_ui]
level=WARNING
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_pz_ui
propagate=0

[logger_input_specs]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_specs
propagate=0

[logger_input_tab_widgets]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.input_tab_widgets
propagate=0

[logger_select_filter]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.select_filter
propagate=0

[logger_target_specs]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.input_widgets.target_specs
propagate=0

#------------------- filter_widgets ---------------------
[logger_bessel]
level=INFO
handlers=fileHandler, consoleHandler,QHandler
qualname=pyfda.filter_widgets.bessel
propagate=0

[logger_equiripple]
level=INFO
handlers=fileHandler, consoleHandler,QHandler
qualname=pyfda.filter_widgets.equiripple
propagate=0

[logger_firwin]
level=INFO
handlers=fileHandler, consoleHandler,QHandler
qualname=pyfda.filter_widgets.firwin
propagate=0
```

(continues on next page)

(continued from previous page)

```
#------------------- fixpoint_widgets ----------------
[logger_fir_df_pyfixp]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.fixpoint_widgets.fir_df.fir_df_pyfixp
propagate=0

[logger_fir_df_pyfixp_ui]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.fixpoint_widgets.fir_df.fir_df_pyfixp_ui
propagate=0

[logger_iir_df1_pyfixp]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.fixpoint_widgets.iir_df1.iir_df1_pyfixp
propagate=0

[logger_iir_df1_pyfixp_ui]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.fixpoint_widgets.iir_df1.iir_df1_pyfixp_ui
propagate=0
#-------------------- plot_widgets -------------------
[logger_mpl_widget]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.mpl_widget
propagate=0

[logger_plot_3d]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_3d
propagate=0

[logger_plot_fft_win]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.logger_plot_fft_win
propagate=0

[logger_plot_hf]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_hf
propagate=0

[logger_plot_impz]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_impz
propagate=0

[logger_plot_impz_ui]
```

(continues on next page)

(continued from previous page)

```
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_impz_ui
propagate=0

[logger_plot_phi]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_phi
propagate=0

[logger_plot_pz]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_pz
propagate=0

[logger_plot_tab_widgets]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_tab_widgets
propagate=0

[logger_plot_tau_g]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.plot_tau_g
propagate=0

[logger_plot_tran_stim]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.tran.plot_tran_stim
propagate=0

[logger_plot_tran_stim_ui]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.tran.plot_tran_stim_ui
propagate=0

[logger_tran_io]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.tran.tran_io
propagate=0

[logger_tran_io_ui]
level=INFO
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.tran.tran_io_ui
propagate=0

#------ Test Case, file doesn't exist -----
[logger_no_existo]
level=INFO
```

(continues on next page)

(continued from previous page)

```
handlers=fileHandler,consoleHandler,QHandler
qualname=pyfda.plot_widgets.no_existo
propagate=0
#---------------------------------------

# specify how to log to:  text console / logging file / GUI logging window
#
# For each handler, define the class (implementation), formatting (see next section)
# and the minimum logging level (defined by the higher of global and individual level,
# e.g. level=INFO prevents all DEBUG level messages).
#---- Console
[handler_consoleHandler]
class=StreamHandler
level=INFO
formatter=noDateFormatter
args=(sys.stdout,)
#---- File
[handler_fileHandler]
class=DynFileHandler # FileHandler is default
level=INFO
formatter=simpleFormatter
args=('pyfda.log', 'w', 'utf-8') # overwrites log file
#args=('pyfda.log','a', 'utf-8') # appends to log file
#---- GUI
[handler_QHandler]
class=QEditHandler
level=INFO
formatter=ezFormatter
args=()

#---------------------------------------

[formatter_simpleFormatter]
format=[%(asctime)s.%(msecs).03d] [%(levelname)7s] [%(name)s:%(lineno)s] %(message)s
# for linebreaks simply make one!
datefmt=%Y-%m-%d %H:%M:%S

[formatter_noDateFormatter]
format=[%(levelname)7s] [%(name)s:%(lineno)s] %(message)s

[formatter_ezFormatter]
format=[%(levelname)7s][%(asctime)s.%(msecs).03d] [%(filename)s:%(lineno)d]
↪%(message)s
datefmt=%H:%M:%S
```

2.2.3 pyfda_rc.py

Layout and some parameters can be customized with the file `pyfda/pyfda_rc.py` (within the install directory right now, no user copy).

CHAPTER

THREE

DEVELOPMENT

This part of the documentation describes the features of pyFDA that are relevant for developers.

3.1 Software Organization

The software is organized as shown in the following figure

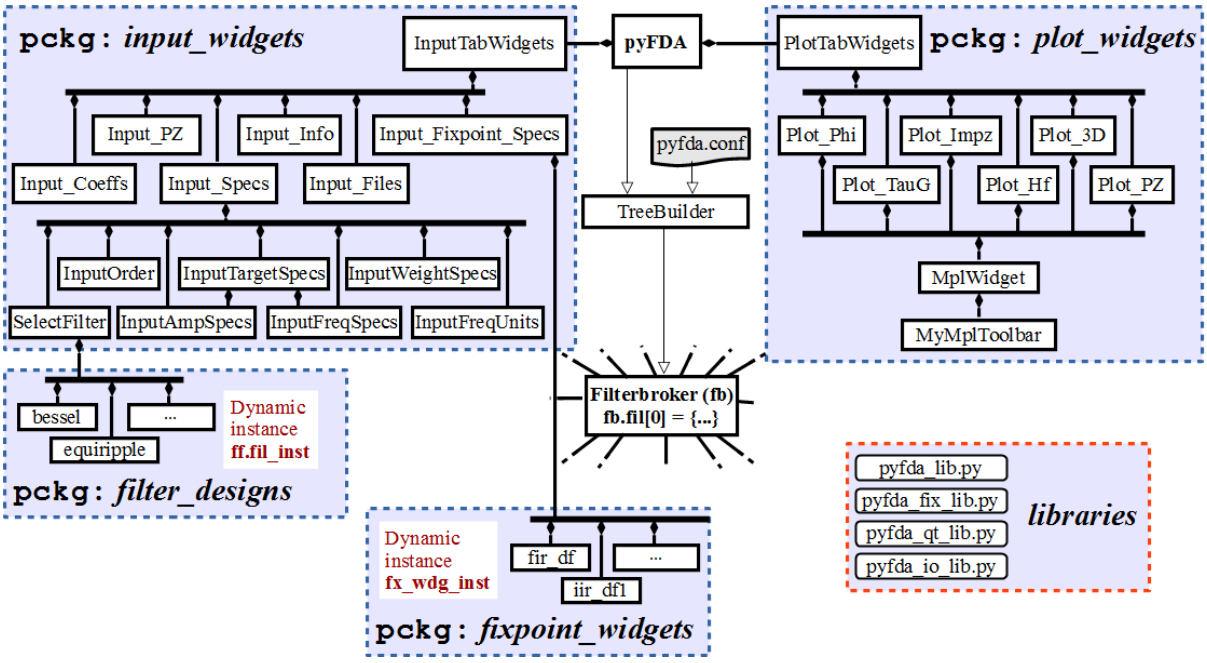

Fig. 3.1: pyfda Organization

Communication:
The modules communicate via Qt's signal-slot mechanism (see: *Signalling: What's up?*).

Data Persistence:
Common data is stored in dicts that can be accessed globally (see: *Persistence: Where's the data?*).

Customization:
The software can be customized a.o. via the file `conf.py` (see: *Customization*).

3.2 Signalling: What's up?

The figure above shows the general pyfda hierarchy. When parameters or settings are changed in a widget, a Qt signal is emitted that can be processed by other widgets with a `sig_rx` slot for receiving information. The dict `dict_sig` is attached to the signal as a "payload", providing information about the sender and the type of event . `sig_rx` is connected to the `process_sig_rx()` method that processes the dict.

Many Qt signals can be connected to one Qt slot and one signal to many slots, so signals of input and plot widgets are collected in `pyfda.input_widgets.input_tab_widgets` and `pyfda.plot_widgets.plot_tab_widgets` respectively and connected collectively.

When a redraw / calculations can take a long time, it makes sense to perform these operations only when the widget is visible and store the need for a redraw in a flag.

```
class MyWidget(QWidget):
    sig_resize = pyqtSignal()   # emit a local signal upon resize
    sig_rx = pyqtSignal(object) # incoming signal
    sig_tx = pyqtSignal(object) # outgoing signal
    from pyfda.libs.pyfda_qt_lib import emit

    def __init__(self, parent):
        super(MyWidget, self).__init__(parent)
        self.data_changed = True # initialize flags
        self.view_changed = True
        self.filt_changed = True
        self.sig_rx.connect(self.process_sig_rx)
        # usually done in method ``_construct_UI()``

    def process_sig_rx(self, dict_sig=None):
    """
    Process signals coming in via subwidgets and sig_rx
    """
    if dict_sig['id'] == id(self):
        logger.warning("Stopped infinite loop:\n{0}".format(pprint_log(dict_sig)))
        return
    if self.isVisible():
        if 'data_changed' in dict_sig or self.data_changed:
            self.recalculate_some_data() # this may take time ...
            self.data_changed = False
        if 'view_changed' in dict_sig and dict_sig['view_changed'] == 'new_limits'\
            or self.view_changed:
            self._update_my_plot()       # ... while this just updates the display
            self.view_changed = False
        if 'filt_changed' in dict_sig or self.filt_changed:
            self.update_wdg_UI()         # new filter needs new UI options
            self.filt_changed = False
    else:
        if 'data_changed' in dict_sig or 'view_changed' in dict_sig:
            self.data_changed = True
            self.view_changed = True
        if 'filt_changed' in dict_sig:
            self.filt_changed = True
```

Data can be transmitted via the global `sig_tx` signal (referenced by the imported `emit()` method):

```
dict_sig = {'fx_sim':'update_data', 'fx_results':some_new_data}
self.emit(dict_sig)
```

The following dictionary keys are generally used, individual ones can be created as needed.

'id'
Python `id(self)` reference to the sending widget instance, needed a.o. to prevent infinite loops which may occur when the rx event is connected to the tx signal. **Automatically added by ``emit()`` if not in ``dict_sig``.**

'class'
Class name of the sending widget, usually given as `self.__class__.__name__`. This can be used for debugging purposes. **Automatically added by ``emit()`` if not in ``dict_sig``.**

'ttl'
Optional, defines the "time-to-live". The integer value given at definition is decreased every time `emit()` is called. When zero is reached, the signal is terminated.

'filt_changed'
A different filter type (response type, algorithm, ...) has been selected or loaded, requiring an update of the UI in some widgets.

'data_changed'
A filter has been designed and the actual data (e.g. coefficients) has changed, you can add the (short) name or a data description as the dict value. When this key is sent, most widgets have to be updated.

'specs_changed'
Filter specifications have changed - this will influence only a few widgets like the dev_plot_hf widget that plots the filter specifications as an overlay or the dev_input_info widget that compares filter performance to filter specifications.

'view_changed'
When e.g. the range of the frequency axis is changed from $0 \ldots f_S/2$ to $-f_S/2 \ldots f_S/2$, this information can be propagated with the `'view_changed'` key.

'ui_local_changed'
Propagate a change of the UI to the containing widget but not to other widgets, examples are: - `'ui_local_changed': self.sender().objectName()'` to propagate the name of the emitting subwidget

'ui_global_changed'
Propagate a change of the UI to other widgets, examples are:

- `'ui_global_changed':'csv'` for a change of CSV import / export options
- `'ui_global_changed':'resize'` when the parent window has been resized
- `'ui_global_changed':'tab'` when a different tab has been selected

'fx_sim'
Signal the phase / status of a fixpoint simulation ('finished', 'error')

3.3 Persistence: Where's the data?

At startup, a dictionary is constructed with information about the filter classes and their methods. The central dictionary `fb.dict` is initialized.

3.4 Main Routines

3.4.1 pyfda.libs.pyfda_dirs

Handle directories in an OS-independent way, create logging directory etc. Upon import, all the variables are set. This is imported first by pyfdax, logger cannot be used yet. Hence, messages are printed to the console.

pyfda.libs.pyfda_dirs.**CONF_FILE = 'pyfda.conf'**

name for general configuration file

pyfda.libs.pyfda_dirs.**HOME_DIR = '/home/docs'**

Home dir and user name

pyfda.libs.pyfda_dirs.**LOG_CONF_FILE = 'pyfda_log.conf'**

name for logging configuration file

pyfda.libs.pyfda_dirs.**LOG_DIR_FILE = '/tmp/.pyfda/pyfda.log'**

Name of the log file, can be changed in pyfdax.py

pyfda.libs.pyfda_dirs.**TEMP_DIR = '/tmp'**

Temp directory for constructing logging dir

pyfda.libs.pyfda_dirs.**USER_DIRS = []**

Placeholder for user widgets directory list, set by treebuilder

pyfda.libs.pyfda_dirs.**USER_NAME = ''**

Home dir and user name

pyfda.libs.pyfda_dirs.**copy_conf_files**(*force_copy=False*, *logger=None*)

If they don't exist, create *pyfda.conf* und *pyfda_log.conf* from template files. in the user directory where they can be edited by the user without admin rights. If they exist and *force_copy=True*, make a backup of the old files and then overwrite them.

Parameters

- **force_copy** (*bool*) – When True, make a backup and overwrite existing config files.
- **logger** (*logger instance*) – Write info and error messages to *logger* when it exists, otherwise use *print()*. When called during the initial phase, loggers have not been created yet and *print()* has to be used.

Return type
None.

pyfda.libs.pyfda_dirs.**env**(*name*)

Get value for environment variable name from the OS.

Parameters
name (*str*) – environment variable

Returns
value of environment variable

Return type
str

pyfda.libs.pyfda_dirs.**get_conf_dir**()

Return the user's configuration directory

pyfda.libs.pyfda_dirs.**get_home_dir**()

Return the user's home directory and name

pyfda.libs.pyfda_dirs.**get_log_dir**()

Try different OS-dependent locations for creating log files and return the first suitable directory name. Only called once at startup.

see https://stackoverflow.com/questions/847850/cross-platform-way-of-getting-temp-directory-in-python

pyfda.libs.pyfda_dirs.**get_yosys_dir**()

Try to find YOSYS path and version from environment variable or path:

pyfda.libs.pyfda_dirs.**last_file_dir = '/home/docs'**

Place holder for file type selected (e.g. "csv") in last file dialog

pyfda.libs.pyfda_dirs.**last_file_name = ''**

Place holder for storing the directory location of the last file

pyfda.libs.pyfda_dirs.**last_file_type = ''**

Global handle to pop-up window for CSV options - this window must be closed before opening another pop-up window! Otherwise, the second window becomes unaccessible (?) and pyfda becomes unresponsive.

pyfda.libs.pyfda_dirs.**update_conf_files**(*logger*)

Copy templates to user config and logging config files, making backups of the old versions.

pyfda.libs.pyfda_dirs.**valid**(*path*)

Check whether path exists and is valid

3.4.2 pyfda.libs.tree_builder

Create the tree dictionaries containing information about filters, filter implementations, widgets etc. in hierarchical form

exception pyfda.libs.tree_builder.**ParseError**

class pyfda.libs.tree_builder.**Tree_Builder**

Read the config file and construct dictionary trees with

- all filter combinations
- valid combinations of filter widgets and fixpoint implementations

build_class_dict(*section*, *subpackage=''*)

- Try to dynamically import the modules (= files) parsed in *section* reading their module level attribute *classes* listing the classes contained in the module.

  When *classes* is a dictionary, e.g. *{"Cheby":"Chebyshev 1"}* where the key is the class name in the module and the value the corresponding display name (used for the combo box).

- When *classes* is a string or a list, use the string resp. the list items for both class and display name.
- Try to import the filter classes

Parameters

- **section** (*str*) – Name of the section in the configuration file to be parsed by `self.parse_conf_section`.
- **subpackage** (*str*) – Name of the subpackage containing the module to be imported. Module names are prepended successively with *['pyfda.' + subpackage + '.', '', subpackage + '.']*

Returns

- **classes_dict** (*dict*)
- *A dictionary with the classes as keys; values are dicts which define*

- *the options (like display name, module path, fixpoint implementations etc).*
- *Each entry has the form e.g.*
- **{<class name>** *({'name':<display name>, 'mod':<full module name>}} e.g.*)
- *.. code-block:: python –*

  {'Cheby1':{'name':'Chebyshev 1',
  'mod':'pyfda.filter_design.cheby1', 'fix': 'IIR_cascade', 'opt': ["option1", "option2"]}

build_fil_tree(*fc*, *rt_dict*, *fil_tree=None*)

Read attributes (ft, rt, rt:fo) from filter class fc) Attributes are stored in the design method classes in the format (example from `common.py`)

```
self.ft = 'IIR'
self.rt_dict = {
         'LP': {'man':{'fo':     ('a','N'),
                       'msg':    ('a', r"<br /><b>Note:</b> Read this!"),
                       'fspecs': ('a','F_C'),
                       'tspecs': ('u', {'frq':('u','F_PB','F_SB'),
                                        'amp':('u','A_PB','A_SB')})
                      },
                'min':{'fo':     ('d','N'),
                       'fspecs': ('d','F_C'),
                       'tspecs': ('a', {'frq':('a','F_PB','F_SB'),
                                        'amp':('a','A_PB','A_SB')})
                      }
               },
        'HP': {'man':{'fo':     ('a','N'),
                       'fspecs': ('a','F_C'),
                       'tspecs': ('u', {'frq':('u','F_SB','F_PB'),
                                        'amp':('u','A_SB','A_PB')})
                      },
                'min':{'fo':     ('d','N'),
                       'fspecs': ('d','F_C'),
                       'tspecs': ('a', {'frq':('a','F_SB','F_PB'),
                                        'amp':('a','A_SB','A_PB')})
                      }
               }
        }
```

Build a dictionary of all filter combinations with the following hierarchy:

response types -> filter types -> filter classes -> filter order rt (e.g. 'LP') ft (e.g. 'IIR') fc (e.g. 'cheby1') fo ('min' or 'man')

All attributes found for fc are arranged in a dict, e.g. for `cheby1.LPman` and `cheby1.LPmin`, listing the parameters to be displayed and whether they are active, unused, disabled or invisible for each subwidget:

```
'LP':{
'IIR':{
     'Cheby1':{
         'man':{'fo':     ('a','N'),
                'msg':    ('a', r"<br /><b>Note:</b> Read this!"),
                'fspecs': ('a','F_C'),
                'tspecs': ('u', {'frq':('u','F_PB','F_SB'),
                                 'amp':('u','A_PB','A_SB')})
                },
```

(continues on next page)

(continued from previous page)

```
        'min':{'fo':     ('d','N'),
               'fspecs': ('d','F_C'),
               'tspecs': ('a', {'frq':('a','F_PB','F_SB'),
                                'amp':('a','A_PB','A_SB')})
               }
            }
      }
}, ...
```

Finally, the whole structure is frozen recursively to avoid inadvertedly changing the filter tree.

For a full example, see the default filter tree `fb.fil_tree` defined in `filterbroker.py`.

Parameters
None

Returns
filter tree

Return type
dict

`init_filters()`

Run at startup to populate global dictionaries and lists:

- Read attributes (*ft*, *rt*, *fo*) from all valid filter classes (*fc*) in the global dict `fb.filter_classes` and store them in the filter tree dict `fil_tree` with the hierarchy

  rt-ft-fc-fo-subwidget:params .

Parameters
None

Returns

- *fb.fil_tree* :

Return type
None, but populates the following global attributes

`parse_conf_file()`

Parse the configuration file *pyfda.conf* (specified in `dirs.USER_CONF_DIR_FILE`). This is run only once at instantiation.

This is performed using *build_class_dict()* which calls *parse_conf_section()*:

- Try to find and import the modules specified in the corresponding sections
- Extract and import the classes defined in each module and give back an OrderedDict with the successfully imported classes and their options (like fully qualified module names, display name, associated fixpoint widgets etc.).
- Information for each section is stored in globally accessible OrderdDicts like`fb.filter_classes`.

The following sections are analyzed:

[Commons]
Try to find user directories; if they exist add them to *dirs.USER_DIRS* and *sys.path*

For the other sections, OrderedDicts are returned with the class names as keys and dictionaries with options as values.

[Input Widgets]
Store (user) input widgets in *fb.input_classes*

[Plot Widgets]
Store (user) plot widgets in *fb.plot_classes*

[Filter Widgets]
Store (user) filter widgets in *fb.filter_classes*

[Fixpoint Widgets]
Store (user) fixpoint widgets in *fb.fixpoint_classes*

Parameters
`None`

Return type
None, but *self.conf* contains the parsed configuration file.

`parse_conf_section`(*section*)

Parse `section` in config file *conf* and return an OrderedDict with the elements `{key:<OPTION>}` where *key* and <OPTION> have been read from the config file. <OPTION> has been sanitized and converted to a list or a dict.

Parameters
section (*str*) – name of the section to be parsed

Returns
section_conf_dict – Ordered dict with the keys of the config files and corresponding values

Return type
dict

pyfda.libs.tree_builder.**merge_dicts_hierarchically**(*d1*, *d2*, *path=None*, *mode='keep1'*)

Merge the hierarchical dictionaries `d1` and `d2`. The dict `d1` is modified in place and returned

Parameters

- **d1** (*dict*) – hierarchical dictionary 1
- **d2** (*dict*) – hierarchical dictionary 2
- **mode** (*str*) – Select the behaviour when the same key is present in both dictionaries:
  - **'keep1'**
    keep the entry from `d1` (default)
  - **'keep2'**
    keep the entry from `d2`
  - **'add1'**
    merge the entries, putting the values from `d2` first (important for lists)
  - **'add2'**
    merge the entries, putting the values from `d1` first (")
- **path** (*str*) – internal parameter for keeping track of hierarchy during recursive calls, it should not be set by the user

Returns
d1 – a reference to the first dictionary, merged-in-place.

Return type
dict

Example

```
>>> merge_dicts_hierarchically(fil_tree, fil_tree_add, mode='add1')
```

Notes

If you don't want to modify d1 in place, call the function using:

```
>>> new_dict = merge_dicts_hierarchically(dict(d1), d2)
```

If you need to merge more than two dicts use:

```
>>> from functools import reduce   # only for py3
>>> reduce(merge, [d1, d2, d3...]) # add / merge all other dicts into d1
```

Taken with some modifications from:

http://stackoverflow.com/questions/7204805/dictionaries-of-dictionaries-merge

3.4.3 pyfda.filter_factory

Dynamic parameters and settings are exchanged via the dictionaries in this file. Importing `filterbroker.py` runs the module once, defining all module variables which have a global scope like class variables and can be imported like

```
>>> import filter_factory as ff
>>> myfil = ff.fil_factory
```

class pyfda.filter_factory.**FilterFactory**

This class implements a filter factory that (re)creates the globally accessible filter instance `fil_inst` from module path and class name, passed as strings.

call_fil_method(*method*, *fil_dict*, *fc=None*)

Instantiate the filter design class passed as string `fc` with the globally accessible handle `fil_inst`. If `fc = None`, use the previously instantiated filter design class.

Next, call the design method passed as string `method` of the instantiated filter design class.

Parameters

- **method** (*string*) – The name of the design method to be called (e.g. 'LPmin')
- **fil_dict** (*dictionary*) – A dictionary with all the filter specs that is passed to the actual filter design routine. This is usually a copy of `fb.fil[0]` The results of the filter design routine are written back to the same dict.
- **fc** (*string (optional, default: None)*) – The name of the filter design class to be instantiated. When nothing is specified, the last filter selection is used.

Returns

err_code –

one of the following error codes:

-1
filter design operation has been cancelled by user

0
filter design method exists and is callable

16
passed method name is not a string

17
filter design method does not exist in class

18
filter design error containing "order is too high"

19
filter design error containing "failure to converge"

99
unknown error

Return type
int

Examples

```
>>> call_fil_method("LPmin", fil[0], fc="cheby1")
```

The example first creates an instance of the filter class 'cheby1' and then performs the actual filter design by calling the method 'LPmin', passing the global filter dictionary `fil[0]` as the parameter.

create_fil_inst(*fc*, *mod=None*)

Create an instance of the filter design class passed as a string `fc` from the module found in `fb.filter_classes[fc]`. This dictionary has been collected by `tree_builder.py`.

The instance can afterwards be globally referenced as `fil_inst`.

Parameters

- **fc** (*str*) – The name of the filter design class to be instantiated (e.g. 'cheby1' or 'equiripple')
- **mod** (*str (optional, default = None)*) – Fully qualified name of the filter module. When not specified, it is read from the global dict `fb.filter_classes[fc]['mod']`

Returns

err_code –

one of the following error codes:

-1
filter design class was instantiated successfully

0
filter instance exists, no re-instantiation necessary

1
filter module not found by FilterTreeBuilder

2
filter module found by FilterTreeBuilder but could not be imported

3
filter class could not be instantiated

4
unknown error during instantiation

Return type
int

Examples

```
>>> create_fil_instance('cheby1')
>>> fil_inst.LPmin(fil[0])
```

The example first creates an instance of the filter class 'cheby1' and then performs the actual filter design by calling the method 'LPmin', passing the global filter dictionary fil[0] as the parameter.

pyfda.filter_factory.**fil_factory = <pyfda.filter_factory.FilterFactory object>**

Class instance of FilterFactory that can be accessed in other modules

pyfda.filter_factory.**fil_inst = None**

Instance of current filter design class (e.g. "cheby1"), globally accessible

```
>>> import filter_factory as ff
>>> ff.fil_factory.create_fil_instance('cheby1') # create instance of dynamic␣
↪class
>>> ff.fil_inst.LPmin(fil[0]) # design a filter
```

3.4.4 pyfda.filterbroker

Dynamic parameters and settings are exchanged via the dictionaries in this file. Importing `filterbroker.py` runs the module once, defining all module variables which have a global scope like class variables and can be imported like

```
>>> import filterbroker as fb
>>> myfil = fb.fil[0]
```

The entries in this file are only used as initial / default entries and to demonstrate the structure of the global dicts and lists. These initial values are also handy for module-level testing where some useful settings of the variables is required.

Notes

Alternative approaches for data persistence could be the packages *shelve* or pickleshare More info on data persistence and storing / accessing global variables:

- http://stackoverflow.com/questions/13034496/using-global-variables-between-files-in-python
- http://stackoverflow.com/questions/1977362/how-to-create-module-wide-variables-in-python
- http://pymotw.com/2/articles/data_persistence.html
- http://stackoverflow.com/questions/9058305/getting-attributes-of-a-class
- http://stackoverflow.com/questions/2447353/getattr-on-a-module

pyfda.filterbroker.**base_dir = ''**

Project base directory

pyfda.filterbroker.**clipboard = None**

Handle to central clipboard instance

pyfda.filterbroker.**filter_classes** = {'Bessel': {'mod': 'pyfda.filter_widgets.bessel', 'name': 'Bessel'}, 'Butter': {'mod': 'pyfda.filter_widgets.butter', 'name': 'Butterworth'}, 'Cheby1': {'mod': 'pyfda.filter_widgets.cheby1', 'name': 'Chebyshev 1'}, 'Cheby2': {'mod': 'pyfda.filter_widgets.cheby2', 'name': 'Chebyshev 2'}, 'Ellip': {'mod': 'pyfda.filter_widgets.ellip', 'name': 'Elliptic'}, 'EllipZeroPhz': {'mod': 'pyfda.filter_widgets.ellip_zero', 'name': 'EllipZeroPhz'}, 'Equiripple': {'mod': 'pyfda.filter_widgets.equiripple', 'name': 'Equiripple'}, 'Firwin': {'mod': 'pyfda.filter_widgets.firwin', 'name': 'Windowed FIR'}, 'MA': {'mod': 'pyfda.filter_widgets.ma', 'name': 'Moving Average'}, 'Manual_FIR': {'mod': 'pyfda.filter_widgets.manual', 'name': 'Manual'}, 'Manual_IIR': {'mod': 'pyfda.filter_widgets.manual', 'name': 'Manual'}}

The keys of this dictionary are the names of all found filter classes, the values are the name to be displayed e.g. in the comboboxes and the fully qualified name of the module containing the class.

pyfda.filterbroker.**redo**()

Store current filter to undo memory *fil_undo*

pyfda.filterbroker.**undo**()

Restore current filter from undo memory *fil_undo*

CHAPTER

FOUR

PYFDA

4.1 pyfda package

4.1.1 Subpackages

pyfda.filter_widgets package

Submodules

pyfda.filter_widgets.bessel module

Design Bessel filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) format

This class is re-instantiated dynamically every time the filter design method is selected, reinitializing instance attributes.

API version info:

1.0
initial working release

1.1

- copy `A_PB` -> `A_PB2` and `A_SB` -> ``A_SB2` for BS / BP designs
- mark private methods as private

1.2
new API using fil_save (enable SOS features)

1.3
new public methods `destruct_UI` and `construct_UI` (no longer called by `__init__`)

1.4

- module attribute `filter_classes` contains class name and combo box name instead of class attribute `name`
- `FRMT` is now a class attribute

2.0
Specify the parameters for each subwidget as tuples in a dict where the first element controls whether the widget is visible and / or enabled. This dict is now called `self.rt_dict`. When present, the dict `self.rt_dict_add` is read and merged with the first one.

2.1
Remove empty methods `construct_UI` and `destruct_UI` and attributes `self.wdg` and `self.hdl`

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.bessel.**Bessel**

Bases: object

Design Bessel filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) format

BPman(*fil_dict*)

BPmin(*fil_dict*)

BSman(*fil_dict*)

BSmin(*fil_dict*)

FRMT = 'sos'

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

ft
filter type

info
filter variants

pyfda.filter_widgets.bessel.**classes = {'Bessel': 'Bessel'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.butter module

Design Butterworth filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) or second-order sections (sos) format

Attention: This class is re-instantiated dynamically every time the filter design method is selected, calling its __init__ method.

API version info:

1.0: initial working release 1.1: - copy A_PB -> A_PB2 and A_SB -> A_SB2 for BS / BP designs

- mark private methods as private

1.2: new API using fil_save (enable SOS features when available) 1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name FRMT* is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the
first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove empty methods construct_UI and destruct_UI and attributes
self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.butter.**Butter**

Bases: object

BPman(*fil_dict*)

BPmin(*fil_dict*)

BSman(*fil_dict*)

BSmin(*fil_dict*)

FRMT = 'sos'

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

pyfda.filter_widgets.cheby1 module

Design Chebyshev 1 filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zpk (zeros, poles, gain) or second-order sections (sos) format.

Attention: This class is re-instantiated dynamically every time the filter design method is selected, calling its __init__ method.

API version info:

1.0: initial working release 1.1: - copy A_PB -> A_PB2 and A_SB -> A_SB2 for BS / BP designs

- mark private methods as private

1.2: new API using fil_save (enable SOS features when available) 1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name FRMT* is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the
first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove empty methods construct_UI and destruct_UI and attributes
self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.cheby1.**Cheby1**

Bases: object

BPman(*fil_dict*)

BPmin(*fil_dict*)

BSman(*fil_dict*)

BSmin(*fil_dict*)

FRMT = 'sos'

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

pyfda.filter_widgets.cheby1.**classes = {'Cheby1': 'Chebyshev 1'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.cheby2 module

Design Chebyshev 2 filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) or second-order sections (sos) format.

Attention: This class is re-instantiated dynamically everytime the filter design method is selected, calling the __init__ method.

API version info:

1.0: initial working release 1.1: - copy A_PB -> A_PB2 and A_SB -> A_SB2 for BS / BP designs

- mark private methods as private

1.2: new API using fil_save (enable SOS features when available) 1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name FRMT* is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the
first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove empty methods construct_UI anddestruct_UI and attributes
self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.cheby2.**Cheby2**

Bases: object

BPman(*fil_dict*)

BPmin(*fil_dict*)

BSman(*fil_dict*)

BSmin(*fil_dict*)

FRMT = 'sos'

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

pyfda.filter_widgets.cheby2.**classes = {'Cheby2': 'Chebyshev 2'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.common module

Common settings and some helper functions for filter design

class pyfda.filter_widgets.common.**Common**

Bases: object

pyfda.filter_widgets.common.**remezord**(*freqs*, *amps*, *rips*, *fs=1*, *alg='ichige'*)

Filter parameter selection for the Remez exchange algorithm.

Calculate the parameters required by the Remez exchange algorithm to construct a finite impulse response (FIR) filter that approximately meets the specified design.

Parameters

- **freqs** (*list*) – A monotonic sequence of band edges specified in Hertz. All elements must be non-negative and less than 1/2 the sampling frequency as given by the *fs* parameter. The band edges "0" and "f_S / 2" do not have to be specified, hence 2 * number(amps) - 2 freqs are needed.
- **amps** (*list*) – A sequence containing the amplitudes of the signal to be filtered over the various bands, e.g. 1 for the passband, 0 for the stopband and 0.42 for some intermediate band.
- **rips** (*list*) – A list with the peak ripples (linear, not in dB!) for each band. For the stop band this is equivalent to the minimum attenuation.
- **fs** (*float*) – Sampling frequency
- **alg** (*string*) – Filter length approximation algorithm. May be either 'herrmann', 'kaiser' or 'ichige'. Depending on the specifications, some of the algorithms may give better results than the others.

Return type
numtaps, bands, desired, weight – See help for the remez function.

Examples

We want to design a lowpass with the band edges of 40 resp. 50 Hz and a sampling frequency of 200 Hz, a passband peak ripple of 10% and a stop band ripple of 0.01 or 40 dB.

```
>>> (L, F, A, W) = remezord([40, 50], [1, 0], [0.1, 0.01], fs = 200)
```

pyfda.filter_widgets.common.**remlplen_herrmann**(*fp*, *fs*, *dp*, *ds*)

Determine the length of the low pass filter with passband frequency fp, stopband frequency fs, passband ripple dp, and stopband ripple ds. fp and fs must be normalized with respect to the sampling frequency. Note that the filter order is one less than the filter length.

Uses approximation algorithm described by Herrmann et al.:

O. Herrmann, L.R. Raviner, and D.S.K. Chan, Practical Design Rules for Optimum Finite Impulse Response Low-Pass Digital Filters, Bell Syst. Tech. Jour., 52(6):769-799, Jul./Aug. 1973.

pyfda.filter_widgets.common.**remlplen_ichige**(*fp*, *fs*, *dp*, *ds*)

Determine the length of the low pass filter with passband frequency fp, stopband frequency fs, passband ripple dp, and stopband ripple ds. fp and fs must be normalized with respect to the sampling frequency. Note that the filter order is one less than the filter length. Uses approximation algorithm described by Ichige et al.: K. Ichige, M. Iwaki, and R. Ishii, Accurate Estimation of Minimum Filter Length for Optimum FIR Digital Filters, IEEE Transactions on Circuits and Systems, 47(10):1008-1017, October 2000.

This seems tol give the most accurate results of the three approximations.

pyfda.filter_widgets.common.**remlplen_kaiser**(*fp*, *fs*, *dp*, *ds*)

Determine the length of the low pass filter with passband frequency fp, stopband frequency fs, passband ripple dp, and stopband ripple ds. fp and fs must be normalized with respect to the sampling frequency. Note that the filter order is one less than the filter length.

Uses approximation algorithm described by Kaiser:

J.F. Kaiser, Nonrecursive Digital Filter Design Using I_0-sinh Window function, Proc. IEEE Int. Symp. Circuits and Systems, 20-23, April 1974.

pyfda.filter_widgets.delay module

Design a simple delay for demonstrating the effect of latency and for debugging

Attention: This class is re-instantiated dynamically every time the filter design method is selected, calling the __init__ method.

API version info:
1.0: initial working release

class pyfda.filter_widgets.delay.**Delay**

Bases: QWidget

APman(*fil_dict*)

FRMT = 'zpk'

construct_UI()

Create additional subwidget(s) needed for filter design: These subwidgets are instantiated dynamically when needed in select_filter.py using the handle to the filter instance, fb.fil_inst.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

info = '\n Delay widget**\n\nallows entering the number of **delays** :math:`N` :math:`T_S`. It is treated as a FIR filter,\nthe number of delays is directly translated to a number of poles (:math:`N > 0`) \nor zeros (:math:`N < 0`).\n\nObviously, there is no minimum design algorithm or no design algorithm at all :-)\n\n '**

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the

name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

`pyfda.filter_widgets.delay.classes = {'Delay': 'Delay'}`

display name

Type
Dict containing class name

pyfda.filter_widgets.ellip module

Design ellip-Filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) format

Attention: This class is re-instantiated dynamically every time the filter design method is selected, calling its __init__ method.

API version info:

1.0: initial working release 1.1: - copy A_PB -> A_PB2 and A_SB -> A_SB2 for BS / BP designs

- mark private methods as private

1.2: new API using fil_save (enable SOS features when available) 1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name FRMT* is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the
first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove empty methods construct_UI and destruct_UI and attributes
self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

`class pyfda.filter_widgets.ellip.Ellip`

Bases: `object`

BPman(*fil_dict*)

Elliptic BP filter, manual order

BPmin(*fil_dict*)

Elliptic BP filter, minimum order

BSman(*fil_dict*)

Elliptic BS filter, manual order

BSmin(*fil_dict*)

Elliptic BP filter, minimum order

FRMT = 'sos'

HPman(*fil_dict*)

Elliptic HP filter, manual order

HPmin(*fil_dict*)

Elliptic HP filter, minimum order

LPman(*fil_dict*)

Elliptic LP filter, manual order

LPmin(*fil_dict*)

Elliptic LP filter, minimum order

info = '\n Elliptic filters**\n\n(also known as Cauer filters) have the steepest rate of transition between the\nfrequency response's passband and stopband of all IIR filters. This comes\nat the expense of a constant ripple (equiripple) :math:`A_PB` and :math:`A_SB`\nin both pass and stop band. Ringing of the step response is increased in\ncomparison to Chebyshev filters.\n\nAs the passband ripple :math:`A_PB` approaches 0, the elliptical filter becomes\na Chebyshev type II filter. As the stopband ripple :math:`A_SB` approaches 0,\nit becomes a Chebyshev type I filter. As both approach 0, it becomes a Butterworth\nfilter (butter).\n\nFor the filter design, the order :math:`N`, minimum stopband attenuation\n:math:`A_SB` and the critical frequency / frequencies :math:`F_PB` where the\ngain first drops below the maximum passband ripple :math:`-A_PB` have to be specified.\n\nThe ``ellipord()`` helper routine calculates the minimum order :math:`N` and the\ncritical passband frequency :math:`F_C` from pass and stop band specifications.\n\n**Design routines:**\n\n``scipy.signal.ellip()``, ``scipy.signal.ellipord()``\n\n '**

pyfda.filter_widgets.ellip.**classes = {'Ellip': 'Elliptic'}**

display name

Type

Dict containing class name

pyfda.filter_widgets.ellip_zero module

Design elliptic Filters (LP, HP, BP, BS) with zero phase in fixed or minimum order, return the filter design in zeros, poles, gain (zpk) format

Attention: This class is re-instantiated dynamically every time the filter design method is selected, calling its __init__ method.

API version info:

2.0: initial working release

2.1: Remove method destruct_UI and attributes self.wdg and self.hdl

2.2

Rename *filter_classes* -> *classes*

class pyfda.filter_widgets.ellip_zero.**EllipZeroPhz**

Bases: QWidget

BPman(*fil_dict*)

Elliptic BP filter, manual order

BPmin(*fil_dict*)

Elliptic BP filter, minimum order

BSman(*fil_dict*)

Elliptic BS filter, manual order

BSmin(*fil_dict*)

Elliptic BP filter, minimum order

FRMT = 'zpk'

HPman(*fil_dict*)

Elliptic HP filter, manual order

HPmin(*fil_dict*)

Elliptic HP filter, minimum order

LPman(*fil_dict*)

Elliptic LP filter, manual order

LPmin(*fil_dict*)

Elliptic LP filter, minimum order

construct_UI()

Create additional subwidget(s) needed for filter design: These subwidgets are instantiated dynamically when needed in select_filter.py using the handle to the filter instance, fb.fil_inst.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

file_dump(*fOut*)

Dump file out in custom text format that apply tool can read to know filter coef's

info = '\n Elliptic filters with zero phase**\n\n(also known as Cauer filters) have the steepest rate of transition between the \nfrequency response's passband and stopband of all IIR filters. This comes\nat the expense of a constant ripple (equiripple) :math:`A_PB` and :math:`A_SB`\nin both pass and stop band. Ringing of the step response is increased in\ncomparison to Chebyshev filters.\n \nAs the passband ripple :math:`A_PB` approaches 0, the elliptical filter becomes\na Chebyshev type II filter. As the stopband ripple :math:`A_SB` approaches 0,\nit becomes a Chebyshev type I filter. As both approach 0, becomes a Butterworth\nfilter (butter).\n\nFor the filter design, the order :math:`N`, minimum stopband attenuation\n:math:`A_SB` and the critical frequency / frequencies :math:`F_PB` where the \ngain first drops below the maximum passband ripple :math:`-A_PB` have to be specified.\n\nThe ``ellipord()`` helper routine calculates the minimum order :math:`N` and \ncritical passband frequency :math:`F_C` from pass and stop band specifications.\n\nThe Zero Phase Elliptic Filter squares an elliptic filter designed in\na way to produce the required Amplitude specifications. So initially the\namplitude specs design an elliptic filter with the square root of the amp specs.\nThe filter is then squared to produce a zero phase filter.\nThe filter coefficients are applied to the signal data in a backward and forward\ntime fashion. This filter can only be applied to stored signal data (not\nreal-time streaming data that comes in a forward time order).\n\nWe are forcing the order N of the filter to be even. This simplifies the poles/zeros\nto be complex (no real values).\n\n**Design routines:**\n\n``scipy.signal.ellip()``, ``scipy.signal.ellipord()``\n\n '**

save_filter()

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.filter_widgets.ellip_zero.**classes = {'EllipZeroPhz': 'EllipZeroPhz'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.equiripple module

Design equiripple-Filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in coefficients format ('ba')

Attention: This class is re-instantiated dynamically every time the filter design method is selected, calling the __init__ method.

API version info:

1.0: initial working release 1.1: mark private methods as private 1.2: new API using fil_save 1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name FRMT* is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the
first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove method destruct_UI and attributes self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.equiripple.**Equiripple**(*objectName='equiripple_inst'*)

Bases: QWidget

BPman(*fil_dict*)

BPmin(*fil_dict*)

BSman(*fil_dict*)

BSmin(*fil_dict*)

DIFFman(*fil_dict*)

FRMT = 'ba'

HILman(*fil_dict*)

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

construct_UI()

Create additional subwidget(s) needed for filter design: These subwidgets are instantiated dynamically when needed in select_filter.py using the handle to the filter instance, fb.fil_inst.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

info = "\n Equiripple filters**\n\nhave the steepest rate of transition between the frequency response's passband\nand stopband of all FIR filters. This comes at the expense of a constant ripple\n(equiripple) :math:`A_PB` and :math:`A_SB` in both pass and stop band.\n\nThe filter-coefficients are calculated in such a way that the transfer function\nminimizes the maximum error (**Minimax** design) between the desired gain and the\nrealized gain in the specified frequency bands using the **Remez** exchange algorithm.\nThe filter design algorithm is known as **Parks-McClellan** algorithm, in\nMatlab (R) it is called ``firpm``.\n\nManual filter order design requires specifying the frequency bands (:math:`F_PB`,\n:math:`f_SB` etc.), the filter order :math:`N` and weight factors :math:`W_PB`,\n:math:`W_SB` etc.) for individual bands.\n\nThe minimum order and the weight factors needed to fulfill the target specifications\nis estimated from frequency and amplitude specifications using Ichige's algorithm.\n\n**Design routines:**\n\n``scipy.signal.remez()``, ``pyfda_lib.remezord()``\n "**

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.filter_widgets.equiripple.**classes = {'Equiripple': 'Equiripple'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.firwin module

Design windowed FIR filters (LP, HP, BP, BS) with fixed order, return the filter design in coefficient ('ba') format

Attention: This class is re-instantiated dynamically everytime the filter design method is selected, calling the __init__ method.

API version info:

1.0: initial working release 1.1: mark private methods as private 1.2: new API using fil_save 1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name FRMT* is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove method destruct_UI and attributes self.wdg and self.hdl

2.2

Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.firwin.**Firwin**(*objectName='firwin_inst'*)

Bases: QWidget

BPman(*fil_dict*)

BPmin(*fil_dict*)

BSman(*fil_dict*)

BSmin(*fil_dict*)

FRMT = 'ba'

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

construct_UI()

Create additional subwidget(s) needed for filter design: These subwidgets are instantiated dynamically when needed in select_filter.py using the handle to the filter object, fb.filObj .

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

firwin(*numtaps*, *cutoff*, *window=None*, *pass_zero=True*, *scale=True*, *nyq=1.0*, *fs=None*)

FIR filter design using the window method. This is more or less the same as *scipy.signal.firwin* with the exception that an ndarray with the window values can be passed as an alternative to the window name.

The parameters "width" (specifying a Kaiser window) and "fs" have been omitted, they are not needed here.

This function computes the coefficients of a finite impulse response filter. The filter will have linear phase; it will be Type I if *numtaps* is odd and Type II if *numtaps* is even. Type II filters always have zero response at the Nyquist rate, so a ValueError exception is raised if firwin is called with *numtaps* even and having a passband whose right end is at the Nyquist rate.

Parameters

- **numtaps** (*int*) – Length of the filter (number of coefficients, i.e. the filter order + 1). *numtaps* must be even if a passband includes the Nyquist frequency.
- **cutoff** (*float or 1D array_like*) – Cutoff frequency of filter (expressed in the same units as *nyq*) OR an array of cutoff frequencies (that is, band edges). In the latter case, the frequencies in *cutoff* should be positive and monotonically increasing between 0 and *nyq*. The values 0 and *nyq* must not be included in *cutoff*.
- **window** (*ndarray or string*) – string: use the window with the passed name from scipy.signal.windows

  ndarray: The window values - this is an addition to the original firwin routine.
- **pass_zero** (*bool, optional*) – If True, the gain at the frequency 0 (i.e. the "DC gain") is 1. Otherwise the DC gain is 0.
- **scale** (*bool, optional*) – Set to True to scale the coefficients so that the frequency response is exactly unity at a certain frequency. That frequency is either: - 0 (DC) if the first passband starts at 0 (i.e. pass_zero

  is True)

  – *nyq* (the Nyquist rate) if the first passband ends at *nyq* (i.e the filter is a single band highpass filter); center of first passband otherwise
- **nyq** (*float, optional*) – Nyquist frequency. Each frequency in *cutoff* must be between 0 and *nyq*.

Returns

h – Coefficients of length *numtaps* FIR filter.

Return type

(numtaps,) ndarray

Raises

ValueError – If any value in *cutoff* is less than or equal to 0 or greater than or equal to *nyq*, if the values in *cutoff* are not strictly monotonically increasing, or if *numtaps* is even but a passband includes the Nyquist frequency.

See also:

`scipy.firwin`

hide_fft_wdg()

The closeEvent caused by clicking the "x" in the FFT widget is caught there and routed here to only hide the window

process_sig_rx(*dict_sig=None*)

Process local signals from / for - FFT window widget - qfft_win_select

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

sig_tx_local

int = . . . , arguments: Sequence = . . .) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

toggle_fft_wdg()

Show / hide FFT widget depending on the state of the corresponding button When widget is shown, trigger an update of the window function.

pyfda.filter_widgets.firwin.**classes = {'Firwin': 'Windowed FIR'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.firwin.**main()**

pyfda.filter_widgets.ma module

Design Moving-Average-Filters (LP, HP) with fixed order, return the filter design in coefficients format ('ba') or as poles/zeros ('zpk')

Attention: This class is re-instantiated dynamically everytime the filter design method is selected, calling the __init__ method.

API version info:

1.0: initial working release 1.1: mark private methods as private 1.2: - new API using fil_save & fil_convert (allow multiple formats,

save 'ba' _and_ 'zpk' precisely)

- include method _store_entries in _update_UI

1.3: new public methods destruct_UI + construct_UI (no longer called by __init__) 1.4: module attribute *filter_classes* contains class name and combo box name

instead of class attribute *name*

FRMT is now a class attribute

2.0: Specify the parameters for each subwidget as tuples in a dict where the first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1: Remove method destruct_UI and attributes self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.ma.**MA**(*objectName='ma_inst'*)

Bases: QWidget

BPman(*fil_dict*)

BSman(*fil_dict*)

FRMT = ('zpk', 'ba')

HPman(*fil_dict*)

HPmin(*fil_dict*)

LPman(*fil_dict*)

LPmin(*fil_dict*)

calc_ma(*fil_dict*, *rt='LP'*)

Calculate coefficients and P/Z for moving average filter based on filter length L = N + 1 and number of cascaded stages and save the result in the filter dictionary.

construct_UI()

Create additional subwidget(s) needed for filter design: These subwidgets are instantiated dynamically when needed in select_filter.py using the handle to the filter instance, fb.fil_inst.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

info = '\n Moving average filters**\n\ncan only be specified via their length and the number of cascaded sections.\n\nThe minimum order to obtain a certain attenuation at a given frequency is\ncalculated via the si function.\n\nMoving average filters can be implemented very efficiently in hard- and software\nas they require no multiplications but only addition and subtractions. Probably\nonly the lowpass is really useful, as the other response types only filter out resp.\nleave components at ``f_S/4`` (bandstop resp. bandpass) resp. leave components\nnear ``f_S/2`` (highpass).\n\n**Design routines:**\n\n``ma.calc_ma()``\n '**

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.filter_widgets.ma.**classes = {'MA': 'Moving Average'}**

display name

Type
Dict containing class name

pyfda.filter_widgets.manual module

Dummy / template file for manual filter designs by entering P/Z or b/a. Targets for LP, HP, BP, BS are provided. Returns nothing.

Attention: This class is re-instantiated dynamically everytime the filter design method is selected, calling the __init__ method.

API version info:

1.0
initial working release

1.1
mark private methods as private

1.2
new API using fil_save

1.3
new public methods destruct_UI + construct_UI (no longer called by __init__)

1.4
module attribute *filter_classes* contains class name and combo box name instead of class attribute *name*

2.0
Specify the parameters for each subwidget as tuples in a dict where the first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1
Remove empty methods construct_UI and destruct_UI and attributes self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.manual.**Manual_FIR**

Bases: object

BPman(*fil_dict*)

BSman(*fil_dict*)

DIFFman(*fil_dict*)

HILman(*fil_dict*)

HPman(*fil_dict*)

LPman(*fil_dict*)

class pyfda.filter_widgets.manual.**Manual_IIR**

Bases: object

BPman(*fil_dict*)

BSman(*fil_dict*)

DIFFman(*fil_dict*)

HILman(*fil_dict*)

HPman(*fil_dict*)

LPman(*fil_dict*)

pyfda.filter_widgets.manual.**classes = {'Manual_FIR': 'Manual', 'Manual_IIR': 'Manual'}**

display name

Type
Dict containing class name

Module contents

pyfda.filter_widgets.bessel

Design Bessel filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) format

This class is re-instantiated dynamically every time the filter design method is selected, reinitializing instance attributes.

API version info:

1.0
initial working release

1.1

- copy A_PB -> A_PB2 and A_SB -> ``A_SB2 for BS / BP designs
- mark private methods as private

1.2
new API using fil_save (enable SOS features)

1.3
new public methods destruct_UI and construct_UI (no longer called by __init__)

1.4

- module attribute filter_classes contains class name and combo box name instead of class attribute name
- FRMT is now a class attribute

2.0
Specify the parameters for each subwidget as tuples in a dict where the first element controls whether the widget is visible and / or enabled. This dict is now called self.rt_dict. When present, the dict self.rt_dict_add is read and merged with the first one.

2.1
Remove empty methods construct_UI and destruct_UI and attributes self.wdg and self.hdl

2.2
Rename *filter_classes* -> *classes*, remove Py2 compatibility

class pyfda.filter_widgets.bessel.**Bessel**

Design Bessel filters (LP, HP, BP, BS) with fixed or minimum order, return the filter design in zeros, poles, gain (zpk) format

ft

filter type

info

filter variants

pyfda.filter_widgets.bessel.**classes = {'Bessel': 'Bessel'}**

display name

Type
Dict containing class name

pyfda.fixpoint_widgets package

Subpackages

pyfda.fixpoint_widgets.fir_df package

Submodules

pyfda.fixpoint_widgets.fir_df.fir_df_amaranth module

pyfda.fixpoint_widgets.fir_df.fir_df_amaranth_mod module

pyfda.fixpoint_widgets.fir_df.fir_df_amaranth_ui module

pyfda.fixpoint_widgets.fir_df.fir_df_pyfixp module

Fixpoint class for calculating direct-form DF1 FIR filter using pyfixp routines

class pyfda.fixpoint_widgets.fir_df.fir_df_pyfixp.**FIR_DF_pyfixp**(*p*)

Bases: object

Construct fixed point object with parameter dict *p*

Usage:

filt = FIR_DF(p) # Instantiate fixpoint filter object with parameter dict

param p
Dictionary with coefficients and quantizer settings with a.o. the following keys : values

- 'b', value: array of coefficients as floats, scaled as *WI:WF*
- 'QACC', value: dict with quantizer settings for the accumulator
- **'q_mul', value: dict with quantizer settings for the partial products**
  optional, 'quant' and 'sat' are both set to 'none' if there is none

type p
dict

fxfilter(*x: iterable = None*, *zi: iterable = None*) → ndarray

Calculate FIR filter (direct form) response via difference equation with quantization. Registers can be initialized with *zi*.

Parameters

- **x** (*array of float or float or None*) – input value(s) scaled and quantized according to the setting of *p['QI']* and fb.fil[0]['qfrmt'] - When x is a scalar, calculate impulse response with the

  amplitude defined by the scalar.

  - When *x == None*, calculate impulse response with amplitude = 1.

- **zi** (*array-like*) – initial conditions for filter memory; when *zi == None*, register contents from last run are used.

Returns
yq – The quantized input value(s) as an ndarray of np.float64 and the same shape as *x* resp. *b* (impulse response).

Return type
ndarray

init(*p*, *zi: iterable = None*) → None

Initialize filter with parameter dict *p* by initialising all registers and quantizers. This needs to be done every time quantizers or coefficients are updated.

Parameters

- **p** (*dict*) – dictionary with coefficients and quantizer settings (see docstring of *__init__()* for details)
- **zi** (*array-like*) – Initialize *L = len(b)* filter registers. Strictly speaking, *zi[0]* is not a register but the current input value. When *len(zi) != len(b)*, truncate or fill up with zeros. When *zi == None*, all registers are filled with zeros.

Return type
None.

reset()

Reset register and overflow counters of quantizers (but don't reset coefficient quantizers)

pyfda.fixpoint_widgets.fir_df.fir_df_pyfixp_ui module

Widget for specifying the parameters of a direct-form FIR filter

class pyfda.fixpoint_widgets.fir_df.fir_df_pyfixp_ui.**FIR_DF_pyfixp_UI**

Bases: QWidget

Widget for entering word formats & quantization, also instantiates fixpoint filter class FilterFIR.

dict2ui()

Update all parts of the UI that need to be updated when specs have been changed outside this class, e.g. coefficients and coefficient wordlength. This also provides the initial setting for the widgets when the filter has been changed.

This is called from one level above by pyfda.input_widgets.input_fixpoint_specs.Input_Fixpoint_Specs.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

fxfilter(*stimulus*)

Provide wrapper around fixpoint filter simulation method: * takes stimulus (iterable or float or None) as parameter * returns fixpoint response (ndarray of float)

process_sig_rx(*dict_sig=None*)

- For locally generated signals (key = 'ui_local_changed'), emit *{'fx_sim': 'specs_changed'}* with local id. Update accu wordlengths for 'auto' or 'full' settings
- For external changes, i.e. *{'fx_sim': 'specs_changed'}* or *{'data_changed': xxx}* update the UI via *self.dict2ui*.

Ignore all other signals

Note: If coefficient / accu quantization settings have been changed in the UI, the referenced dicts *fb.fil[0]['fxq']['QCB']* and *...['QACC']* have already been updated by the corresponding subwidgets *FX_UI_WQ*

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_accu_settings()

Calculate required number of fractional bits for the accumulator from the sum of coefficient and input fractional bits.

Calculate number of extra integer bits for the accumulator (guard bits) depending on the coefficient area (sum of absolute coefficient values) for *cmbW == 'auto'* or depending on the number of coefficients for *cmbW == 'full'*. The latter works for arbitrary coefficients but requires more bits.

The new values are written to the fixpoint coefficient dict *fb.fil[0]['fxq']['QACC']* and the UI is updated.

update_ovfl_cnt_all()

Update all overflow counters of the UI after simulation has finished (except for coefficient quantizers).

This is usually called from one level above by `pyfda.input_widgets.input_fixpoint_specs.Input_Fixpoint_Specs`.

Module contents

Submodules

pyfda.fixpoint_widgets.fx_ui_wq module

Helper classes and functions for generating and simulating fixpoint filters

class pyfda.fixpoint_widgets.fx_ui_wq.**FX_UI_WQ**(*q_dict: dict*, *objectName: str = 'fx_ui_wq_inst'*, ***kwargs*)

Bases: `QWidget`

Subwidget for selecting and displaying fixpoint quantization / overflow options.

When ui subwidgets are modified, the dictionary specified during the construction is modified.

Subwidgets can be modified by calling the instance method *dict2ui(q_dict)*. q_dict is an optional quantization dict, when omitted, the dictionary passed during construction is used.

Constructor parameters

q_dict: dict
A dictionary containing the quantization settings that can be modified via the UI of this widget. This is usually a global quantization dict like *fb.fil[0]['fxq']['QCB']*, it can also be a local dict.

Attention: The dict is passed by reference, its values are modified via the UI. The quantizer dict *self.Q.q_dict* contains a copy of these keys / values.

objectName: str
The string is used to set the objectName of the Qt widget.

returns

- *None*
- *The values for the following keys can be modified via the UI*
- **- `quant`** (*quantization behaviour*)
- **- `ovfl`** (*overflow behaviour*)
- **- `WI`** (*number of integer bits*)
- **- `WF`** (*number of fractional bits*)
- *Programmatically, the values for the following keys can be modified.*
- **- `w_a_m`** (*automatic or manual update of word format*)

Widget (UI) settings are stored in the local *ui_dict* dictionary with the keys and their default settings described below.

Key : Default value # Comment

————————-:————————#————————————- 'label' : '' # widget text label, usually set by the 'label_q' : 'Quant.' # subwidget text label 'cmb_q' : List with tooltip and combo box choices (default: 'round', 'fix',

'floor'), see *pyfda_qt_lib.qcmb_box_populate()* or code below

'label_ov' : 'Ovfl.' # subwidget text label 'cmb_ov' : List with tooltip and combo box choices (default: 'wrap', 'sat')

'fractional' : True # Display WF, otherwise WF=0 'lbl_sep' : '.' # label between WI and WF field 'max_led_width' : 30 # max. length of lineedit field 'WI_len' : 2 # max. number of integer *digits* 'tip_WI'

: 'Number of integer bits' # Mouse-over tooltip 'WF_len' : 2 # max. number of frac. *digits* 'tip_WF' : 'Number of frac. bits' # Mouse-over tooltip

'lock_vis' : 'off" # Pushbutton for locking visible 'tip_lock' : 'Sync input/output quant.'# Tooltip for lock push button

'cmb_w_vis'
['off' # Is Auto/Man. selection visible?] # ['a', 'm', 'f']

'cmb_w_items' : List with tooltip and combo box choices 'count_ovfl_vis': 'on' # Is overflow counter visible?

['on', 'off', 'auto']

'MSB_LSB_vis' : 'off" # Are MSB / LSB settings visible?

All labels support HTML formatting.

When instantiating the widget, these settings can be modified by setting keyword parameters, e.g.:

```

**self.wdg_wq_accu = FX_UI_WQ(**
fb.fil[0]['fxq']['QACC'], objectName='wdg_wq_accu_inst', label='<b>Accu Quantizer <i>Q<sub>A&amp;nbsp;</sub></i>:</b>')

```

but_lock_checked(*checked: bool*) → None

Update the icon of the push button depending on its state (checked or not) and fire the signal {'ui_local_changed': 'butLock'}

but_lock_update_icon(*checked: bool*) → None

Update the icon of the push button depending on its state

dict2ui(*q_dict: dict = None*) → None

Use the passed quantization dict *q_dict* to update:

- UI subwidgets *WI*, *WF quant*, *ovfl*, *cmbW*
- the instance quantizer object *self.Q.q_dict*
- overflow counters need to be updated from calling instance

If *q_dict is None*, use data from the quantizer dict *self.Q.q_dict* instead, this can be used to update the UI.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

enable_subwidgets()

Enable integer and fractional part of the quantization format, depending on 'w_a_m' settings.

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the

signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

ui2dict() → None

The subwidgets for *ovfl*, *quant*, *WI*, *WF*, *w_a_m* trigger this method when modified.

Update the quantization dict *self.Q.q_dict* and the global quantization dict *self.q_dict* from the UI.

Emit a signal with *{'ui_local_changed': <objectName of the sender>}*.

update_WI_WF()

Update display, visibility / writability of integer and fractional part of the quantization format. depending on *fb.fil[0]['fx_sim']* ...['qfrmt'] and ...['w_a_m'] settings

update_ovfl_cnt()

Update the overflow counter and MSB / LSB display (if visible)

Module contents

pyfda.input_widgets package

Submodules

pyfda.input_widgets.amplitude_specs module

Widget for entering amplitude specifications

Author: Christian Münker

class pyfda.input_widgets.amplitude_specs.**AmplitudeSpecs**(*parent=None*, *title='Amplitude Specs'*, *objectName=''*)

Bases: QWidget

Build and update widget for entering the amplitude specifications like A_SB, A_PB etc.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

eventFilter(*source*, *event*)

Filter all events generated by the QLineEdit widgets. Source and type of all events generated by monitored objects are passed to this eventFilter, evaluated and passed on to the next hierarchy level.

- When a QLineEdit widget gains input focus (QEvent.FocusIn`), display the stored value from filter dict with full precision
- When a key is pressed inside the text field, set the *spec_edited* flag to True.
- When a QLineEdit widget loses input focus (QEvent.FocusOut`), store current value in linear format with full precision (only if `spec_edited`== True) and display the stored value in selected format

load_dict()

Reload and reformat the amplitude textfields from filter dict when a new filter design algorithm is selected or when the user has changed the unit (V / W / dB):

- Reload amplitude entries from filter dictionary and convert to selected to reflect changed settings unit.
- Update the lineedit fields, rounded to specified format.

process_sig_rx(*dict_sig=None*)

Process signals coming in via subwidgets and sig_rx

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_UI(*new_labels=()*)

Called from filter_specs.update_UI() and target_specs.update_UI(). Set labels and get corresponding values from filter dictionary. When number of entries has changed, the layout of subwidget is rebuilt, using

- *self.qlabels*, a list with references to existing QLabel widgets,
- *new_labels*, a list of strings from the filter_dict for the current filter design
- 'num_new_labels`, their number
- *self.n_cur_labels*, the number of currently visible labels / qlineedit fields

pyfda.input_widgets.freq_specs module

Subwidget for entering frequency specifications

class pyfda.input_widgets.freq_specs.**FreqSpecs**(*parent=None*, *title='Frequency Specs'*, *objectName=''*)

Bases: QWidget

Build and update widget for entering the frequency specifications like F_sb, F_pb etc.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

eventFilter(*source*, *event*)

Filter all events generated by the QLineEdit widgets. Source and type of all events generated by monitored objects are passed to this eventFilter, evaluated and passed on to the next hierarchy level.

- When a QLineEdit widget gains input focus (QEvent.FocusIn`), display the stored value from filter dict with full precision
- When a key is pressed inside the text field, set the *spec_edited* flag to True.
- When a QLineEdit widget loses input focus (QEvent.FocusOut`), store current value normalized to f_S with full precision (only if `spec_edited`== True) and display the stored value in selected format

load_dict()

Triggered by FocusIn, FocusOut and ESC-Key in LineEdit fields and by `sort_dict_freqs():

***load_dict()* is called during init and when the frequency unit or the** sampling frequency have been changed via *filter_specs.update_UI()* -> *self.update_UI()* -> *self.sort_dict_freqs()*

- Reload textfields from filter dictionary
- Transform the displayed frequency spec input fields according to the units setting (i.e. f_S). Spec entries are always stored normalized w.r.t. f_S in the dictionary; when f_S or the unit are changed, only the displayed values of the frequency entries are updated, not the dictionary!
- Update the displayed frequency unit

It should be called when *specs_changed* or *data_changed* is emitted at another place, indicating that a reload is required.

process_sig_rx(*dict_sig=None*)

Process signals coming in via subwidgets and sig_rx

recalc_freqs()

Update normalized frequencies when absolute frequencies are locked and update frequency unit. This is called by via signal {'view_changed': 'f_S'}.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the

name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

`sig_tx`

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

`sort_dict_freqs()`

- **Sort visible filter dict frequency spec entries with ascending frequency if**
  the sort button is activated
- Update the visible QLineEdit frequency widgets

The method is called when: - update_UI has been called after changing the filter design algorithm

that the response type has been changed eg. from LP -> HP, requiring a different order of frequency entries

- a frequency spec field has been edited
- the sort button has been clicked (from filter_specs.py)

`update_UI(new_labels=())`

Called by *input_specs.update_UI()* and *target_specs.update_UI()* Set labels and get corresponding values from filter dictionary. When number of entries has changed, the layout of subwidget is rebuilt, using

- *self.qlabels*, a list with references to existing QLabel widgets,
- *new_labels*, a list of strings from the filter_dict for the current filter design
- 'num_new_labels`, their number
- *self.n_cur_labels*, the number of currently visible labels / qlineedit fields

`update_f_display(source)`

Update frequency display when frequency or sampling frequency has been updated. Depending on whether it has focus or not, the value is displayed with full precision or rounded.

Triggered by

pyfda.input_widgets.freq_units module

Subwidget for entering frequency units

class pyfda.input_widgets.freq_units.**FreqUnits**(*parent=None*, *title='Frequency Units'*, *objectName=''*)

Bases: QWidget

Build and update widget for entering frequency unit, frequency range and sampling frequency f_S

The following key-value pairs of the *fb.fil[0]* dict are modified:

- *'freq_specs_unit'* : The unit ('f_S', 'f_Ny', 'Hz' etc.) as a string
- ***'freqSpecsRange'***
  [A list with two entries for minimum and maximum frequency] values for labelling the frequency axis
- *'f_S'* : The sampling frequency for referring frequency values to as a float
- *'f_max'* : maximum frequency for scaling frequency axis
- *'plt_fUnit'*: frequency unit as string
- *'plt_tUnit'*: time unit as string
- *'plt_fLabel'*: label for frequency axis
- *'plt_tLabel'*: label for time axis

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

eventFilter(*source*, *event*)

Filter all events generated by the QLineEdit *f_S* widget. Source and type of all events generated by monitored objects are passed to this eventFilter, evaluated and passed on to the next hierarchy level.

- When a QLineEdit widget gains input focus (QEvent.FocusIn`), display the stored value from filter dict with full precision
- When a key is pressed inside the text field, set the *spec_edited* flag to True.
- When a QLineEdit widget loses input focus (QEvent.FocusOut`), store current value with full precision (only if `spec_edited`== True) and display the stored value in selected format. Emit 'view_changed':'f_S'
- When f_S has been changed, update *fb.fil[0]['f_S']*, emit *{'view_changed': 'f_S'}* to update other widgets and only *then* update {'f_S_prev': fb.fil[0]['f_S']} to allow correction of normalized frequency with the old value of f_S.

load_dict()

Reload comboBox settings and textfields from filter dictionary Block signals during update of combobox / lineedit widgets This is called from *input_specs.load_dict()*

process_sig_rx(*dict_sig=None*)

Process signals coming from - FFT window widget - qfft_win_select

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_UI(*emit=True*)

update_UI is called - during init (direct call) - when the unit combobox is changed (signal-slot) - when a signal {'view_changed': 'f_S'} or {'data_changed': ...} has been

received. In this case, the UI is updated from the fb.fil[0] dictionary and no signal is emitted (*emit==False*).

Set various scale factors and labels depending on the setting of the unit combobox.

Update the freqSpecsRange and finally, emit 'view_changed':'f_S' signal

pyfda.input_widgets.input_coeffs module

Widget for displaying and modifying filter coefficients

class pyfda.input_widgets.input_coeffs.**Input_Coeffs**(*parent=None*)

Bases: QWidget

Create widget with a (sort of) model-view architecture for viewing / editing / entering data contained in *self.ba* which is a list of two numpy arrays:

- *self.ba[0]* contains the numerator coefficients ("b")
- *self.ba[1]* contains the denominator coefficients ("a")

The lists don't neccessarily have the same length but they are always defined. For FIR filters, *self.ba[1][0] = 1*, all other elements are zero.

The length of both lists can be egalized with *self._equalize_ba_length()*.

Views / formats are handled by the ItemDelegate() class.

clear_table()

Clear self.ba: Initialize coeff for a poles and a zero @ origin, a = b = [1; 0].

Refresh QTableWidget

dict2ui()

- update the UI from the dictionary
- Update the fixpoint quant. object
- Update the quantized coefficient view and the overflow counter
- Refresh the table

Triggered by:

- ***process_sig_rx()*: self.fx_specs_changed == True or**
  dict_sig['fx_sim'] == 'specs_changed'
- *self.qfrmt2dict()*
- *self.fx_base2dict()*

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

export_table()

Export data from coefficient table *self.tblCoeff* to clipboard / file in CSV format.

fx_base2dict()

Read out the UI settings of *self.ui.cmb_fx_base* (triggering this method) which specifies the fx number base (dec, bin, ...) for display and store it in *fb.fil[0]['fx_base']*.

Refresh the table and update quantization widgets. Don't emit a signal because this only influences the view not the data itself.

load_dict()

- Copy filter dict array *fb.fil[0]['ba']* to the coefficient list *self.ba*
- Set quantization UI from dict, update quantized coeff. display / overflow counter
- Update the display via *self.refresh_table()*.

The filter dict is a "normal" 2D-numpy float array for the b and a coefficients while the coefficient list *self.ba* is a list of two float ndarrays to allow for different lengths of b and a subarrays while adding / deleting items.

process_sig_rx(*dict_sig=None*)

Process signals coming from sig_rx

qfrmt2dict()

Read out the UI settings of *self.ui.cmb_qfrmt* (triggering this method) and store it under the 'qfrmt' key if it is a fixpoint format. Set the *fb.fil[0]['fx_sim']* flag accordingly.

Refresh the table and update quantization widgets, finally emit a signal *{'fx_sim': 'specs_changed'}*.

quant_coeffs_save()

Triggered by pushing "Quantize button":

- Store selected / all quantized coefficients in *self.ba*
- Refresh table (for the case that anything weird happens during quantization)
- Reset Overflow flags *self.ba_q[2]* and *self.ba_q[3]*
- Save quantized *self.ba* to filter dict (in *_save_dict()*). This emits {'data_changed': 'input_coeffs'}

quant_coeffs_view()

This method only creates a view on the quantized coefficients and stores it in *self.ba_q*, the actual coefficients in *self.ba* remain unchanged!

- Reset overflow counters
- Quantize filter coefficients *self.ba* with quantizer objects *self.Q[0]* and *self.Q[1]* for *b* and *a* coefficients respectively and store them in the array *self.ba_q*. Depending on the number base (float, dec, hex, ...) the result can be of type float or string.
- Store pos. / neg. overflows in the 3rd and 4th column of *self.ba_q* as 0 or +/- 1.

refresh_table()

Update *self.ba_q* from *self.ba* (list with 2 one-dimensional numpy arrays), i.e. requantize displayed values (not *self.ba*) and overflow counters.

Refresh the table from it. Data is displayed via *ItemDelegate.displayText()* in the number format set by *fb.fil[0]['fx_base']*.

- self.ba[0] -> b coefficients
- self.ba[1] -> a coefficients

The table dimensions are set according to the filter type set in *fb.fil[0]['ft']* which is either 'FIR' or 'IIR' and by the number of rows in *self.ba*.

Called at the end of nearly every method.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

class pyfda.input_widgets.input_coeffs.**ItemDelegate**(*parent*)

Bases: QStyledItemDelegate

The following methods are subclassed to replace display and editor of the QTableWidget.

- *displayText()* displays the data stored in the table in various number formats
- *createEditor()* creates a line edit instance for editing table entries
- *setEditorData()* pass data with full precision and in selected format to editor
- *setModelData()* pass edited data back to model (*self.ba*)

Editing the table triggers *setModelData()* but does not emit a signal outside this class, only the *ui.butSave* button is highlighted. When it is pressed, a signal with *'data_changed':'input_coeffs'* is produced in class *Input_Coeffs*. Additionally, a signal is emitted with *'fx_sim': 'specs_changed'*

createEditor(*parent*, *options*, *index*)

Neet to set editor explicitly, otherwise QDoubleSpinBox instance is created when space is not sufficient?! editor: instance of e.g. QLineEdit (default) index: instance of QModelIndex options: instance of QStyleOptionViewItemV4

displayText(*text*, *locale*) → str

Display *text* with selected fixpoint base and number of places

text: string / QVariant from QTableWidget to be rendered locale: locale for the text

The instance parameter *Q[c].ovr_flag* is set to +1 or -1 for
positive / negative overflows, else it is 0.

initStyleOption(*option*, *index*)

Initialize *option* with the values using the *index* index. When the item (0,1) is processed, it is styled especially. All other items are passed to the original *initStyleOption()* which then calls *displayText()*. Afterwards, check whether an fixpoint overflow has occured and color item background accordingly.

setEditorData(*editor*, *index*)

Pass the data to be edited to the editor: - retrieve data with full accuracy from self.ba (in float format) - requantize data according to settings in fixpoint object - represent it in the selected format (int, hex, ...)

editor: instance of e.g. QLineEdit index: instance of QModelIndex

setModelData(*editor*, *model*, *index*) → None

When editing has finished, read the updated data from the editor (= QTableWidget), and store it in *self.ba* as float / complex for *fb.fil[0]['fx_sim'] == False*.

For all other formats, convert data back to floating point format via *frmt2float()* and store it in *self.ba* as float / complex. Next, use *float2frmt()* to quantize data and and store it in *parent.ba_q*. Finally, refresh the table item to display it in the selected format via *_refresh_table_item()*.

editor: instance of e.g. QLineEdit model: instance of QAbstractTableModel index: instance of QModelIndex

text(*item*) → str

Return item text as string transformed by self.displayText()

This is used a.o. by *pyfda_io_lib.qtable2csv()* and *libs.pyfda_fix_lib* to read out a table in text mode, e.g. *text = table.itemDelegate().text(item)*

pyfda.input_widgets.input_coeffs.**classes = {'Input_Coeffs': 'b,a'}**

display name

Type
Dict containing class name

pyfda.input_widgets.input_coeffs_ui module

Create the UI for the FilterCoeffs class

class pyfda.input_widgets.input_coeffs_ui.**Input_Coeffs_UI**(*parent=None*)

Bases: QWidget

Create the UI for the FilterCoeffs class

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

process_sig_rx(*dict_sig=None*)

Process signals coming from the CSV pop-up window

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.input_widgets.input_fixpoint_specs module

Widget for simulating fixpoint filters and generating Verilog Code

class pyfda.input_widgets.input_fixpoint_specs.**Input_Fixpoint_Specs**(*parent=None*, *objectName='input_fixpoint_spec_inst'*)

Bases: QWidget

Create the widget that holds the dynamically loaded fixpoint filter UI

dict2ui()

Called during *__init__()* and from *process_sig_rx()*.

Update UI from *fb.fil[0]['fx_sim']*, *fb.fil[0]['qfrmt']* and the fx filter dict *fb.fil[0]['fxq']*. This affects the visibility and the fx settings of input, output and dyn. filter widget via their *dict2ui()* methods. The setting of the *self.cmb_qfrmt* combobox influencing float / fixpoint number format is updated as well.

embed_fixp_img(*img_file: str*) → QPixmap

Embed *img_file* in png format as *self.img_fixp*

Parameters
img_file (*str*) – path and file name to image file

Returns
self.img_fixp – pixmap containing the passed img_file

Return type
QPixmap object

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

exportHDL()

Synthesize HDL description of filter

fx_filt_init()

Wrapper around *self.fx_filt_ui.init_filter()* to catch errors. Initialize fix-point filter, reset registers and overflow counters

TODO: - Update the *fxqc_dict* containing all quantization information

Returns
error – 0 for sucessful fx widget construction, -1 for error

Return type
int

fx_sim_calc_response(*dict_sig*) → None

- Read fixpoint stimulus from *dict_sig* in integer format
- Pass it to the fixpoint filter which calculates the fixpoint response
- Store the result in *fb.fx_results* and return. In case of an error, *fb.fx_results == None*

Return type
None

load_fx_filter() → None

A new filter has been loaded, create fixpoint filter from scratch.

(Re-)Read list of available fixpoint filters for a given filter class every time a new filter has been designed or loaded.

Then try to import the fixpoint designs in the list and populate the fixpoint implementation combo box *self.cmb_fx_wdg* with successfull imports.

process_sig_rx(*dict_sig: dict = None*) → None

Process signals coming in via *sig_rx* from other widgets.

Trigger fx simulation:

1. *fx_sim': 'init'*: Start fixpoint simulation by sending *'fx_sim':'start_fx_response_calculation'*
2. Store fixpoint response in *fb.fx_result* and return to initiating routine

process_sig_rx_local(*dict_sig: dict = None*) → None

Process signals coming in from input and output quantizer subwidget and emit {'fx_sim': 'specs_changed'} in the end.

qfrmt2ui()

Triggered by by a change of index of the combo box *self.cmb_qfrmt*.

- Update UI (fixpoint format, visibility of fixpoint widgets) from combobox *self.cmb_qfrmt* to *fb.fil[0]['fx_sim']* and *fb.fil[0]['qfrmt']*.
- Update fixpoint widget settings via *self.dict2ui()*
- Emit {'fx_sim': 'specs_changed'}.

resize_img() → None

Triggered when *self* (the widget) is selected or resized. The method resizes the image inside QLabel to completely fill the label while keeping the aspect ratio.

The parent *InputTabWidget* defines the available width (minus some offset due to margins etc.), unfortunately *self.width()* cannot be used as a measure as it expands with the parent but doesn't shrink.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_rx_local

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

sig_tx

int = . . . , arguments: Sequence = . . .) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

pyfda.input_widgets.input_fixpoint_specs.classes = {'Input_Fixpoint_Specs': 'Fixpoint'}

display name

Type
Dict with class name

pyfda.input_widgets.input_info module

Widget for displaying infos about filter and filter design method and debugging infos

class pyfda.input_widgets.input_info.**Input_Info**(*parent=None*)

Bases: QWidget

Create widget for displaying infos about filter specs and filter design method

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

load_dict()

update docs and filter performance

process_sig_rx(*dict_sig=None*)

Process signals coming from sig_rx

sig_rx

int = . . . , arguments: Sequence = . . .) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

sig_tx

int = . . . , arguments: Sequence = . . .) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

pyfda.input_widgets.input_info.**classes = {'Input_Info': 'Info'}**

display name

Type
Dict containing class name

pyfda.input_widgets.input_info_about module

Widget for exporting / importing and saving / loading filter data

class pyfda.input_widgets.input_info_about.**AboutWindow**(*parent=None*)

Bases: QDialog

Create a pop-up widget for the About Window.

collect_info()

Collect information about version, imported modules in strings:

self.info_str
[General info, copyright, version, link to readthedocs] This info is always visible.

self.about_str: OS, user name, directories, versions of installed software

display_GPL_lic()

Display GPL license

display_MIT_lic()

Display MIT license

display_about_str()

Display general "About" info

display_changelog()

Display changelog

to_clipboard(*my_string*, *html=False*)

Copy version info to clipboard TODO: This is stupid: md -> html -> md ?!

pyfda.input_widgets.input_pz module

Widget for displaying and modifying filter Poles and Zeros

class pyfda.input_widgets.input_pz.**Input_PZ**(*parent=None*)

Bases: QWidget

Create the window for entering exporting / importing and saving / loading data

cmplx2frmt(*text*, *places=-1*)

Convert number "text" (real or complex or string) to the format defined by cmbPZFrmt.

Returns
string

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

eventFilter(*source*, *event*)

Filter all events generated by the QLineEdit widgets. Source and type of all events generated by monitored objects are passed to this eventFilter, evaluated and passed on to the next hierarchy level.

- When a QLineEdit widget gains input focus (*QEvent.FocusIn*), display the stored value from filter dict with full precision
- When a key is pressed inside the text field, set the *spec_edited* flag to True.
- When a QLineEdit widget loses input focus (*QEvent.FocusOut*), store current value in linear format with full precision (only if *spec_edited == True*) and display the stored value in selected format

export_table()

Export data from coefficient table *self.tblCoeff* to clipboard in CSV format or to file using a selected format

frmt2cmplx(*string: str*, *default: float = 0.0*) → complex

Convert string to real or complex, try to find out the format (cartesian, polar with various angle formats)

load_dict()

Load all entries from filter dict fb.fil[0]['zpk'] into the Zero/Pole/Gain list self.zpk and update the display via *self._refresh_table()*. The explicit np.array(...) statement enforces a deep copy of fb.fil[0], otherwise the filter dict would be modified inadvertedly. *dtype=object* needs to be specified to create a numpy array from the nested lists with differing lengths without creating the deprecation warning

"Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated."

The filter dict fb.fil[0]['zpk'] is a list of numpy float ndarrays for z / p / k values *self.zpk* is an array of float ndarrays with different lengths of z / p / k subarrays to allow adding / deleting items.

Format is: [array[zeros, ...], array[poles, ...], k]

process_sig_rx(*dict_sig=None*)

Process signals coming from sig_rx

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

class pyfda.input_widgets.input_pz.**ItemDelegate**(*parent*)

Bases: QStyledItemDelegate

The following methods are subclassed to replace display and editor of the QTableWidget.

- *displayText()* displays the data stored in the table in various number formats
- *createEditor()* creates a line edit instance for editing table entries
- *setEditorData()* pass data with full precision and in selected format to editor
- *setModelData()* pass edited data back to model (*self.zpk*)

createEditor(*parent*, *options*, *index*)

Neet to set editor explicitly, otherwise QDoubleSpinBox instance is created when space is not sufficient?! editor: instance of e.g. QLineEdit (default) index: instance of QModelIndex options: instance of QStyleOptionViewItemV4

displayText(*text*, *locale*)

Display *text* with selected format (cartesian / polar) and number of places

text: string / QVariant from QTableWidget to be rendered locale: locale for the text

initStyleOption(*option*, *index*)

Initialize *option* with the values using the *index* index. All items are passed to the original *initStyleOption()* which then calls *displayText()*.

Afterwards, check whether a pole (index.column() == 1)is outside the UC and color item background accordingly (not implemented yet).

setEditorData(*editor*, *index*)

Pass the data to be edited to the editor: - retrieve data with full accuracy (*places=-1*) from *zpk* (in float format) - represent it in the selected format (Cartesian, polar, ...)

editor: instance of e.g. QLineEdit index: instance of QModelIndex

setModelData(*editor*, *model*, *index*)

When editor has finished, read the updated data from the editor, convert it to complex format and store it in both the model (= QTableWidget) and in *zpk*. Finally, refresh the table item to display it in the selected format (via *to be defined*) and normalize the gain.

editor: instance of e.g. QLineEdit model: instance of QAbstractTableModel index: instance of QModelIndex

text(*item*)

Return item text as string transformed by self.displayText()

pyfda.input_widgets.input_pz.**classes = {'Input_PZ': 'P/Z'}**

display name

Type
Dict containing class name

pyfda.input_widgets.input_pz_ui module

Create the UI for the FilterPZ class

class pyfda.input_widgets.input_pz_ui.**Input_PZ_UI**(*parent=None*)

Bases: QWidget

Create the UI for the InputPZ class

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

process_sig_rx(*dict_sig=None*)

Process signals coming from the CSV pop-up window

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the

signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.input_widgets.input_specs module

Widget stacking all subwidgets for filter specification and design. The actual filter design is started here as well.

class pyfda.input_widgets.input_specs.**Input_Specs**(*parent=None*, *objectName='input_specs_inst'*)

Bases: QWidget

Build widget for entering all filter specs

color_design_button(*state*)

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

load_dict()

Reload info text from global dict *fb.fil[0]* and reset 'DESIGN' button

process_sig_rx(*dict_sig*, *propagate=False*)

Process signals coming in via subwidgets and sig_rx

All signals terminate here unless the flag *propagate=True*.

The sender name of signals coming in from local subwidgets is changed to its parent widget (*input_specs*) to prevent infinite loops.

process_sig_rx_local(*dict_sig=None*)

Signals coming in from local subwidgets need to be propagated, so set *propagate=True* and proceed with processing in *process_sig_rx*.

quit_program()

When <QUIT> button is pressed, send 'quit_program'

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_rx_local

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

start_design_filt()

Start the actual filter design process:

- store the entries of all input widgets in the global filter dict.
- call the design method, passing the whole dictionary as the argument: let the design method pick the needed specs
- update the input widgets in case weights, corner frequencies etc. have been changed by the filter design method
- the plots are updated via signal-slot connection

update_UI(*dict_sig={}*) → None

update_UI is called every time the filter design method or order (min / man) has been changed as this usually requires a different set of frequency and amplitude specs.

At this time, the actual filter object instance has been created from the name of the design method (e.g. 'cheby1') in select_filter.py. Its handle has been stored in fb.fil_inst.

fb.fil[0] (currently selected filter) is read, then general information for the selected filter type and order (min/man) is gathered from the filter tree [fb.fil_tree], i.e. which parameters are needed, which widgets are visible and which message shall be displayed.

Then, the UIs of all subwidgets are updated using their "update_UI" method.

update_info()

Update the info field of the filter selection

pyfda.input_widgets.input_specs.**classes = {'Input_Specs': 'Specs'}**

display name

Type
Dict containing class name

pyfda.input_widgets.input_tab_widgets module

Tabbed container for all input widgets

class pyfda.input_widgets.input_tab_widgets.**InputTabWidgets**(*parent=None*, *objectName='input_tab_widgets_inst'*)

Bases: QWidget

Create a tabbed widget for all input subwidgets in the list `fb.input_widgets_list`. This list is compiled at startup in *pyfda.libs.tree_builder.Tree_Builder*.

current_tab_changed()

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

log_rx(*dict_sig=None*)

Enable *self.sig_rx.connect(self.log_rx)* above for debugging.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.input_widgets.select_filter module

Subwidget for selecting the filter, consisting of combo boxes for: - Response Type (LP, HP, Hilbert, ...) - Filter Type (IIR, FIR, CIC ...) - Filter Class (Butterworth, ...)

class pyfda.input_widgets.select_filter.**SelectFilter**(*parent=None*, *objectName='select_filter_inst'*)

Bases: QWidget

Construct and read combo boxes for selecting the filter, consisting of the following hierarchy:

1. Response Type rt (LP, HP, Hilbert, ...)
2. Filter Type ft (IIR, FIR, CIC ...)
3. Filter Class (Butterworth, ...)

Every time a combo box is changed manually, the filter tree for the selected response resp. filter type is read and the combo box(es) further down in the hierarchy are populated according to the available combinations.

sig_tx({'filt_changed'}) is emitted and propagated to input_filter_specs.py where it triggers the recreation of all subwidgets.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

load_dict()

Reload comboboxes from filter dictionary to update changed settings after loading a filter design from disk. *load_dict* uses the automatism of _set_response_type etc. of checking whether the previously selected filter design method is also available for the new combination.

load_filter_order(*enb_signal=False*)

Called by set_design_method or from InputSpecs (with enb_signal = False),
load filter order setting from fb.fil[0] and update widgets

process_sig_rx(*dict_sig*)

Process signals coming in via sig_rx

All signals terminate here.

The sender name of signals coming in from local subwidgets is changed to its parent widget to prevent infinite loops.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.input_widgets.target_specs module

Widget collecting subwidgets for the target filter specifications (currently only amplitude and frequency specs.)

class pyfda.input_widgets.target_specs.**TargetSpecs**(*parent=None*, *title='Target Specs'*, *objectName=''*)

Bases: QWidget

Build and update widget for entering the target specifications (frequencies and amplitudes) like F_SB, F_PB, A_SB, etc.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

process_sig_rx(*dict_sig=None*)

Process signals coming in via subwidgets and sig_rx

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the

signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

`sig_tx_local`

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

`update_UI`(*new_labels=()*)

Called when a new filter design algorithm has been selected - Pass new frequency and amplitude labels to the amplitude and frequency

spec widgets. The first element of the 'amp' and the 'freq' tuple is the state with 'u' for 'unused' and 'd' for disabled

- The *filt_changed* signal is emitted already by *select_filter.py*

pyfda.input_widgets.weight_specs module

Widget for entering weight specifications

`class pyfda.input_widgets.weight_specs.`**`WeightSpecs`**(*parent=None*, *objectName=''*)

Bases: `QWidget`

Build and update widget for entering the weight specifications like W_SB, W_PB etc.

`emit`(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

`eventFilter`(*source*, *event*)

Filter all events generated by the QLineEdit widgets. Source and type of all events generated by monitored objects are passed to this eventFilter, evaluated and passed on to the next hierarchy level.

- When a QLineEdit widget gains input focus (QEvent.FocusIn`), display the stored value from filter dict with full precision
- When a key is pressed inside the text field, set the *spec_edited* flag to True.

- When a QLineEdit widget loses input focus (QEvent.FocusOut`), store current value in linear format with full precision (only if `spec_edited`== True) and display the stored value in selected format

load_dict()

Reload textfields from filter dictionary to update changed settings

process_sig_rx(*dict_sig=None*)

Process signals coming in via subwidgets and sig_rx

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_UI(*new_labels=[]*)

Called from filter_specs.update_UI() Set labels and get corresponding values from filter dictionary. When number of entries has changed, the layout of subwidget is rebuilt, using

- *self.qlabels*, a list with references to existing QLabel widgets,
- ***new_labels*, a list of strings from the filter_dict for the current**
  filter design
- 'num_new_labels`, their number
- *self.n_cur_labels*, the number of currently visible labels / qlineedit fields

Module contents

pyfda.libs package

Submodules

pyfda.libs.compat module

Was: Compatibility wrapper to obtain same syntax for both Qt4 and 5, PyQt4 has been removed

class pyfda.libs.compat.**QPushButtonRT**(*parent=None*, *text=None*, *margin=10*)

Bases: QPushButton

Subclass QPushButton to render rich text

setText(*self*, *text: str | None*)

sizeHint(*self*) → QSize

pyfda.libs.csv_option_box module

Library with classes and functions for file and text IO

class pyfda.libs.csv_option_box.**CSV_option_box**(*parent*)

Bases: QDialog

Create a pop-up widget for setting CSV options. This is needed when storing / reading Comma-Separated Value (CSV) files containing coefficients or poles and zeros.

closeEvent(*event*)

Override closeEvent (user has tried to close the window) and send a signal to parent where window closing is registered before actually closing the window.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

load_settings()

Load settings of CSV options widget from pyfda_rc.params.

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

store_settings()

Store settings of CSV options widget in `pyfda_rc.params`.

pyfda.libs.frozendict module

Create an immutable dictionary for the filter tree. The eliminates the risk that a filter design routine inadvertedly modifies the dict e.g. via a shallow copy. Used by filterbroker.py and filter_tree_builder.py

Taken from http://stackoverflow.com/questions/2703599/what-would-a-frozen-dict-be

class pyfda.libs.frozendict.**FrozenDict**(*orig={}*, ***kw*)

Bases: frozenset

Behaves in most ways like a regular dictionary, except that it's immutable.

It differs from other implementations because it doesn't subclass "dict". Instead it subclasses "frozenset" which guarantees immutability. FrozenDict instances are created with the same arguments used to initialize regular dictionaries, and has all the same methods.

```
>>> f = FrozenDict(x=3,y=4,z=5)
>>> f['x']
>>> 3
>>> f['a'] = 0
>>> TypeError: 'FrozenDict' object does not support item assignment
```

FrozenDict can accept un-hashable values, but FrozenDict is only hashable if its values are hashable.

```
>>> f = FrozenDict(x=3, y=4, z=5)
>>> hash(f)
>>> 646626455
>>> g = FrozenDict(x=3,y=4,z=[])
>>> hash(g)
>>> TypeError: unhashable type: 'list'
```

FrozenDict interacts with dictionary objects as though it were a dict itself:

```
>>> original = dict(x=3, y=4, z=5)
>>> frozen = FrozenDict(x=3, y=4, z=5)
>>> original == frozen
>>> True
```

FrozenDict supports bi-directional conversions with regular dictionaries:

```
>>> original = {'x': 3, 'y': 4, 'z': 5}
>>> FrozenDict(original)
>>> FrozenDict({'x': 3, 'y': 4, 'z': 5})
>>> dict(FrozenDict(original))
>>> {'x': 3, 'y': 4, 'z': 5}
```

copy()

Return a shallow copy of a set.

classmethod fromkeys(*keys*, *value*)

get(*key*, *default=None*)

items()

keys()

values()

class pyfda.libs.frozendict.**Item**(*iterable=()*, /)

Bases: tuple

Designed for storing key-value pairs inside a FrozenDict, which itself is a subclass of frozenset. The __hash__ is overloaded to return the hash of only the key. __eq__ is overloaded so that normally it only checks whether the Item's key is equal to the other object, HOWEVER, if the other object itself is an instance of Item, it checks BOTH the key and value for equality.

WARNING: Do not use this class for any purpose other than to contain key value pairs inside FrozenDict!!!!

The __eq__ operator is overloaded in such a way that it violates a fundamental property of mathematics. That property, which says that a == b and b == c implies a == c, does not hold for this object. Here's a demonstration:

```
>>> x = Item(('a',4))
>>> y = Item(('a',5))
>>> hash('a')
>>> 194817700
>>> hash(x)
>>> 194817700
>>> hash(y)
>>> 194817700
>>> 'a' == x
>>> True
>>> 'a' == y
>>> True
>>> x == y
>>> False
```

property key

property value

pyfda.libs.frozendict.**col**(*i*)

For binding named attributes to spots inside subclasses of tuple.

pyfda.libs.frozendict.**freeze_hierarchical**(*hier_dict*)

Return the argumenent as a FrozenDict where all nested dicts have also been converted to FrozenDicts recursively. When the argument is not a dict, return the argument unchanged.

pyfda.libs.pyfda_dirs module

Handle directories in an OS-independent way, create logging directory etc. Upon import, all the variables are set. This is imported first by pyfdax, logger cannot be used yet. Hence, messages are printed to the console.

pyfda.libs.pyfda_dirs.**CONF_FILE = 'pyfda.conf'**

name for general configuration file

pyfda.libs.pyfda_dirs.**HOME_DIR = '/home/docs'**

Home dir and user name

pyfda.libs.pyfda_dirs.**LOG_CONF_FILE = 'pyfda_log.conf'**

name for logging configuration file

pyfda.libs.pyfda_dirs.**LOG_DIR_FILE = '/tmp/.pyfda/pyfda.log'**

Name of the log file, can be changed in pyfdax.py

pyfda.libs.pyfda_dirs.**TEMP_DIR = '/tmp'**

Temp directory for constructing logging dir

pyfda.libs.pyfda_dirs.**USER_DIRS = []**

Placeholder for user widgets directory list, set by treebuilder

pyfda.libs.pyfda_dirs.**USER_NAME = ''**

Home dir and user name

pyfda.libs.pyfda_dirs.**copy_conf_files**(*force_copy=False*, *logger=None*)

If they don't exist, create *pyfda.conf* und *pyfda_log.conf* from template files. in the user directory where they can be edited by the user without admin rights. If they exist and *force_copy=True*, make a backup of the old files and then overwrite them.

Parameters

- **force_copy** (*bool*) – When True, make a backup and overwrite existing config files.
- **logger** (*logger instance*) – Write info and error messages to *logger* when it exists, otherwise use *print()*. When called during the initial phase, loggers have not been created yet and *print()* has to be used.

Return type
None.

pyfda.libs.pyfda_dirs.**env**(*name*)

Get value for environment variable `name` from the OS.

Parameters
name (*str*) – environment variable

Returns
value of environment variable

Return type
str

pyfda.libs.pyfda_dirs.**get_conf_dir**()

Return the user's configuration directory

pyfda.libs.pyfda_dirs.**get_home_dir**()

Return the user's home directory and name

pyfda.libs.pyfda_dirs.**get_log_dir**()

Try different OS-dependent locations for creating log files and return the first suitable directory name. Only called once at startup.

see https://stackoverflow.com/questions/847850/cross-platform-way-of-getting-temp-directory-in-python

pyfda.libs.pyfda_dirs.**get_yosys_dir**()

Try to find YOSYS path and version from environment variable or path:

pyfda.libs.pyfda_dirs.**last_file_dir = '/home/docs'**

Place holder for file type selected (e.g. "csv") in last file dialog

pyfda.libs.pyfda_dirs.**last_file_name = ''**

Place holder for storing the directory location of the last file

pyfda.libs.pyfda_dirs.**last_file_type = ''**

Global handle to pop-up window for CSV options - this window must be closed before opening another pop-up window! Otherwise, the second window becomes unaccessible (?) and pyfda becomes unresponsive.

pyfda.libs.pyfda_dirs.**update_conf_files**(*logger*)

Copy templates to user config and logging config files, making backups of the old versions.

pyfda.libs.pyfda_dirs.**valid**(*path*)

Check whether path exists and is valid

pyfda.libs.pyfda_fft_windows_lib module

class pyfda.libs.pyfda_fft_windows_lib.**QFFTWinSelector**(*win_dict*, *objectName=''*)

Bases: QWidget

dict2ui()

The *win_dict* dictionary has been updated somewhere else, now update the window selection widget and make corresponding parameter widgets visible if *self.win_dict['cur_win_name']* is different from current combo box entry:

- set FFT window type combobox from *self.win_dict['cur_win_name']*
- use *ui2dict_win()* to update parameter widgets for new window type from *self.win_dict* without emitting a signal

dict2ui_params()

Set parameter values from *win_dict*

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

get_window(*N: int*, *win_name: str = None*, *sym: bool = False*, *clear_cache: bool = False*) → array

Calculate or retrieve from cache the selected window function with *N* points.

Parameters

- **N** (*int*) – Number of data points
- **win_name** (*str, optional*) – Name of the window. If specified (default is None), this will be used to obtain the window function, its parameters and tool tipps etc. via *set_window_name()*. If not, the previous setting are used. If window and number of data points are unchanged, the window is retrieved from *self.win_dict['win']* instead of recalculating it.

  If some kind of error occurs during calculation of the window, a rectangular window is used as a fallback and the class attribute *self.err* is set to *True*.
- **sym** (*bool, optional*) – When True, generate a symmetric window for filter design. When False (default), generate a periodic window for spectral analysis.
- **clear_cache** (*bool, optional*) – Clear the window cache

Returns

win_fnct – The window function with *N* data points (should be normalized to 1) This is also stored in *self.win_dict['win']*. Additionally, the normalized equivalent noise bandwidth is calculated and stored as *self.win_dict['nenbw']* as well as the correlated gain *self.win_dict['cgain']*.

Return type

ndarray

process_sig_rx(*dict_sig=None*)

Process signals coming from the widget one hierarchy higher to update the widgets from the dictionary

set_window_name(*win_name: str = ''*) → bool

Select and set a window function object from its name *win_name* and update the *win_dict* dictionary correspondingly with:

win_dict['cur_win_name'] # win_name: new current window name (str) win_dict['win'] # []: clear window function array (empty list) win_dict[win_name]['win_fnct'] # function object win_dict[win_name]['n_par'] # number of parameters (int)

The above is only updated when the window type has been changed compared to *win_dict['cur_win_name']* !

Parameters
win_name (*str*) – Name of the window, which will be looked up in *all_windows_dict*. If it is "", use *self.win_dict['cur_win_name']* instead

Returns
win_err – Error flag; *True* when *win_name* could not be resolved

Return type
bool

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

ui2dict_params()

Read out window parameter widget(s) when editing is finished and update win_dict.

Emit 'view_changed': 'fft_win_par'

ui2dict_win() → None

- read FFT window type combo box and update win_dict using *set_window_name()*
- determine number of parameters and make lineedit or combobox fields visible
- set tooltipps and parameter values from dict

ui2dict_win_emit(*arg=None*) → None

- triggered by the window type combo box
- read FFT window type combo box and update win_dict using *set_window_name()*, update parameter widgets accordingly
- emit 'view_changed': 'fft_win_type'

class pyfda.libs.pyfda_fft_windows_lib.**UserWindows**(*parent*)

Bases: object

pyfda.libs.pyfda_fft_windows_lib.**blackmanharris**(*N*, *L*, *sym*)

pyfda.libs.pyfda_fft_windows_lib.**calc_cosine_window**(*N*, *sym*, *a*)

Return window based on cosine functions with amplitudes specified by the list *a*.

pyfda.libs.pyfda_fft_windows_lib.**get_valid_windows_list**(*win_names_list=[]*, *win_dict={}*)

Extract the list of all the keys from *win_dict* that define a window and are contained in the list of window names 'win_names_list'. This is verified by checking whether the key in *win_dict* has a dict as a value with the key *fn_name* and the window function as a value. When *win_dict* is empty, use the global *all_windows_dict*.

When *win_names_list* is empty, return all valid window names from *all_windows_dict*.

All window names in 'win_names_list' without a corresponding key in *all_windows_dict* raise a warning.

The result is a alphabetically sorted (on the lower-cased names) list containing the valid window names (strings).

This list can be used e.g. for initialization of a combo box.

Parameter

win_names_list: list of str
A list of window names defining the windows available in the constructed instance, a subset of all the windows defined in *all_windows_dict*

win_dict: dict

rtype
A validated list of window names

pyfda.libs.pyfda_fft_windows_lib.**get_windows_dict**(*win_names_list=[]*, *cur_win_name='Rectangular'*)

Return a subdictionary of a deep copy of *all_windows_dict* containing all valid windows for the names passed in *win_names_list*. When the latter is empty, put all valid windows into the returned subdictionary (which should be more or less a mutable deep copy of *all_windows_dict* in this case.).

cur_win_name determines the initial value of the *cur_win_name* key in the returned dictionary.

Parameters

- **win_names_list** (*list of window names (str), optional*)
- **cur_win_name** (*str, optional*) – Name of initial setting for *cur_win_name* value (current window name), default: "Rectangular"

Returns
A dictionary with windows, window functions, docstrings etc

Return type
dict

pyfda.libs.pyfda_fft_windows_lib.**logger = <Logger pyfda.libs.pyfda_fft_windows_lib (INFO)>**

Dictionary with available FFT windows, their function names and their properties.

When the function name *fn_name* is just a string, it is taken from *scipy.signal.windows*, otherwise it has to be fully qualified name.

pyfda.libs.pyfda_fft_windows_lib.**ultraspherical**(*N*, *alpha=0.5*, *x_0=1*, *sym=True*)

The window does not work yet! More info: https://www.recordingblogs.com/wiki/ultraspherical-window and https://www.ece.uvic.ca/~andreas/RLectures/UltraSpherWinJASP.pdf

pyfda.libs.pyfda_fix_lib module

Fixpoint library for converting numpy scalars and arrays to quantized numpy values and formatting reals in various formats

class pyfda.libs.pyfda_fix_lib.**Fixed**(*q_dict*)

Bases: object

Implement binary quantization of signed scalar or array-like objects in the form WI.WF where WI and WF are the wordlength of integer resp. fractional part; total wordlength is $W = WI + WF + 1$ due to the sign bit.

Examples

Define a dictionary with the format options and pass it to the constructor:

```
>>> q_dict = {'WI':1, 'WF':14, 'ovfl':'sat', 'quant':'round'}
>>> myQ = Fixed(q_dict)  # instantiate fixpoint quantizer
>>> WI = myQ.q_dict['WI']  # access quantizer parameters
>>> myQ.set_qdict({'WF': 13, 'WI': 2})  # update quantizer parameters
```

Parameters

- **q_dict** (*dict*) – define quantization options with the following keys
- **'WI'** (*)
- **'WF'** (*)
- **'quant'** (*) –
  - **'floor': (default) largest integer *I* such that** $I \leq x$ (= binary truncation)
  - 'round': (binary) rounding
  - 'fix': round to nearest integer towards zero ('Betragsschneiden')
  - 'ceil': smallest integer *I*, such that $I \geq x$
  - 'rint': round towards nearest int
  - 'none': no quantization
- **'ovfl'** (*) –
  - 'wrap': do a two's complement wrap-around
  - 'sat' : saturate at minimum / maximum value
  - 'none': no overflow; the integer word length is ignored
- **N_over** (*) – total number of overflows (should be considered as read-only except for when an external quantizer is used)

- **Additionally**
- **the** (*the following keys from global dict fb.fil[0] define*)
- **numbers** (*number base and quantization/overflow behaviour for fixpoint*)
- **``'fx_sim'``** (*)
- **``'fx_base'``** (*) –
  - 'dec' : decimal (base = 10)
  - 'bin' : binary (base = 2)
  - 'hex' : hexadecimal (base = 16)
  - 'oct' : octal (base = 8)
  - 'csd' : canonically signed digit (base = "3")
- **``'qfrmt'``** (*) –
  - 'qint' : fixpoint integer format
  - 'qfrac' : fractional fixpoint format

q_dict

A reference to the quantization dictionary passed during construction (see above). This dictionary is updated here and can be accessed from outside.

Type
dict

LSB

value of LSB (smallest quantization step), *self.LSB = 2 ** -q_dict['WF']*

Type
float

MSB

value of most significant bit (MSB), *self.MSB = 2 ** (q_dict['WI'] - 1)*

Type
float

MIN

most negative representable value, *self.MIN = -2 * self.MSB*

Type
float

MAX

largest representable value, *self.MAX = 2 * self.MSB - self.LSB*

Type
float

N

total number of simulation data points

Type
integer

N_over_neg

number of negative overflows (commented out)

Type
integer

N_over_pos

number of positive overflows (commented out)

Type
integer

ovr_flag

overflow flag, meaning:

0 : no overflow

+1: positive overflow

-1: negative overflow

has occured during last fixpoint conversion.

Type
integer or integer array (same shape as input argument)

places

number of places required for printing in the selected 'fx_base' format. For binary formats, this is the same as the wordlength. Calculated from the numeric base 'fx_base' and the total word length WI + WF + 1.

Type
integer

Overflow flags and counters are set in `self.fixp()` and reset in `self.reset_N()`

Example

class *Fixed()* can be used like the Matlab *quantizer()* object / function from the fixpoint toolbox, see (Matlab) 'help round' and 'help quantizer/round' e.g.

MATLAB

```
>>> q_dsp = quantizer('fixed', 'round', [16 15], 'wrap');
>>> yq = quantize(q_dsp, y)
```

PYTHON >>> q_dsp = {'WI':0, 'WF': 15, 'quant':'round', 'ovfl':'wrap'} >>> my_q = Fixed(q_dsp) >>> yq = my_q.fixp(y)

fixp(*y*, *in_frmt: str = 'qfrac'*, *out_frmt: str = 'qfrac'*)

Return a quantized copy *yq* for *y* (scalar or array-like) with the same shape as *y*. The returned data is always in float format, use float2frmt() to obtain different number formats.

This is used a.o. by the following methods / classes:

- *frmt2float()*: always returns a float with RWV
- *float2frmt()*: starts with RWV, passes on the scaling argument
- input_coeffs: uses both methods above when quantizing coefficients

Saturation / two's complement wrapping happens outside the range +/- MSB, requantization (round, floor, fix, ...) is applied on the ratio *y / LSB*.

Fractional number format WI.WF (*fb.fil[0]['qfrmt'] = 'qfrac'*): *LSB = 2 ** -WF*

- Multiply float input by *1 / self.LSB = 2**WF*, obtaining integer scale
- Quantize
- Scale back by multiplying with *self.LSB* to restore fractional point

- Find pos. and neg. overflows and replace them by wrapped or saturated values

Integer number format W = 1 + WI + WF (*fb.fil[0]['qfrmt'] = 'qint'*): *LSB = 1*

- Multiply float input by *2 ** WF* to obtain integer scale
- Quantize and treat overflows in integer scale

Parameters

- **y** (*scalar or array-like object of float*) – input value (floating point format) to be quantized
- **in_frmt** (*str*) – Determine the scaling *before* quantizing / saturation *'qfrac'* (default): fractional float input, *y* is multiplied by *2 ** WF before* quantizing / saturating.

  For all other settings, *y* is transformed unscaled.
- **out_frmt** (*str*) – Determine the scaling *after* quantizing / saturation **'qfrac'** (default): fractional fixpoint output format, *y* is divided by *2 ** WF after* quantizing / saturating.

  For all other settings, *y* is transformed unscaled.

Returns

yq – with the same shape as *y*, in the range *-2 * self.MSB ... 2 * self.MSB - self.LSB*

Return type

float scalar or ndarray

Examples

```
>>> q_obj_a = {'WI':1, 'WF':6, 'ovfl':'sat', 'quant':'round'}
>>> myQa = Fixed(q_obj_a) # instantiate fixed-point object myQa
>>> myQa.resetN()  # reset overflow counter
>>> a = np.arange(0,5, 0.05) # create input signal
```

```
>>> aq = myQa.fixp(a) # quantize input signal
>>> plt.plot(a, aq) # plot quantized vs. original signal
>>> print(myQa.q_dict('N_over'), "overflows!") # print number of overflows
```

```
>>> # Convert output to same format as input:
>>> b = np.arange(200, dtype = np.int16)
>>> btype = np.result_type(b)
>>> # MSB = 2**7, LSB = 2**(-2):
>>> q_obj_b = {'WI':7, 'WF':2, 'ovfl':'wrap', 'quant':'round'}
>>> myQb = Fixed(q_obj_b) # instantiate fixed-point object myQb
>>> bq = myQb.fixp(b)
>>> bq = bq.astype(btype) # restore original variable type
```

float2frmt(*y*) → str

Convert an array or single value of float / complex / string to a quantized representation in one of the formats float / int / bin / hex / csd.

Called a.o. by *itemDelegate.displayText()* for on-the-fly number conversion. Returns fixpoint representation for *y* (scalar or array-like) with numeric format *self.frmt* and a total wordlength of *W = self.q_dict['WI'] + self.q_dict['WF'] + 1* bits. The result has the same shape as *y*.

The float is always quantized / saturated using *self.fixp()* before it is converted to different fixpoint number bases.

Parameters
y (*scalar or array-like*) – y has to be an integer, float or complex decimal number

Returns
The numeric format is set in *fb.fil[0]['fx_base'])*. It has the same shape as *y*. For all formats except *float* a fixpoint representation with a total number of W = WI + WF + 1 binary digits is returned.

Return type
str, float or an ndarray of float or string

`frmt2float`(*y*)

Return floating point representation for fixpoint *y* (scalar or array) given in format *fb.fil[0]['fx_base']*.

When input format is float, return unchanged.

Else:

- Remove illegal characters and leading '0's
- Count number of fractional places *frc_places* and remove radix point
- Calculate decimal, fractional representation *y_dec* of string, using the base and the number of fractional places
- Calculate two's complement for *W* bits (only for negative bin and hex numbers)
- Calculate fixpoint float representation *y_float = fixp(y_dec, out_frmt='qfrmt')*, dividing the result by *2**WF*.

Parameters
y (*scalar or string or array of scalars or strings in number format float or*) – *fb.fil[0]['fx_base']* ('dec', 'hex', 'oct', 'bin' or 'csd')

Returns
- Quantized floating point (*dtype=np.float64*) representation of input string
- of same shape as *y*.

`frmt2float_scalar`(*y: str*) → float

Convert a string in 'dec', 'bin', 'oct', 'hex', 'csd' numeric format to float.

- format is taken from the global *fb.fil[0]['fx_base']*
- maximum wordlength is determined from the local quantization dict keys *self.q_dict['WI']* and *self.q_dict['WF']*
- negative numbers can be represented by a '-' sign or in two's complement
- represented numbers may be fractional and / or complex.
- the result is divided by 2**WF for *fb.fil[0]['qfrmt'] == 'qint'* in *fixp()*

Parameters
y (*str*) – A string formatted as a decimal, binary, octal, hex or csd number representation. The number string may contain a '.' or ',' to represent fractal numbers. When the string contains a 'j'm it is tried to split the string into real and imaginary part.

Returns
The float / complex representation of the string

Return type
float or complex

requant(*x_i*, *QI*)

Change word length of input signal *x_i* with fractional and integer widths defined by 'QI' to the word format defined by *self.q_dict* using the quantization and saturation methods specified by *self.q_dict['quant']* and *self.q_dict['ovfl']*.

Input and output word are aligned at their binary points.

Parameters

- **self** (*Fixed() object*) – *self.qdict()* is the quantizer dict that specifies the output word format and the requantizing / saturation methods to be used.
- **x_i** (*int, float or complex scalar or array-like*) – signal to be requantized with quantization format defined in quantizer QI
- **QI** (*quantizer*) – Quantizer for input word, only the keys 'WI' and 'WF' for integer and fractional wordlength are evaluated. *QI.q_dict['WI'] = 2* and *QI.q_dict['WF'] = 13* e.g. define Q-Format '2.13' with 2 integer, 13 fractional bits and 1 implied sign bit = 16 bits total.

Returns

y – requantized output data with same shape as input data, quantized as specified in *self.qdict*.

Return type

any

Example

The following shows an example of rescaling an input word from Q2.4 to Q0.3 using wrap-around and truncation. It's easy to see that for simple wrap-around logic, the sign of the result may change.

```
S | WI1 | WI0 * WF0 | WF1 | WF2 | WF3  :  WI = 2, WF = 4, W = 7
0 |  1  |  0  *  1  |  0  |  1  |  1   =  43 (dec) or 43/16 = 2 + 11/16␣
↪(float)
            *
        |  S  * WF0 | WF1 | WF2        :  WI = 0, WF = 3, W = 4
        0  *  1  |  0  |  1         =  7 (dec) or 7/8 (float)
```

When the input is integer format, the fractional value is calculated as an intermediate representation by multiplying the integer value by 2 ** (-WF). Integer and fractional part are truncated / extended to the output quantization specifications.

Changes in the number of integer bits *dWI* and fractional bits *dWF* are handled separately.

When operating on bit level in hardware, the following operations are used:

Fractional Bits

- For reducing the number of fractional bits by *dWF*, simply right-shift the integer number by *dWF*. For rounding, add '1' to the bit below the truncation point before right-shifting.
- Extend the number of fractional bits by left-shifting the integer by *dWF*, LSB's are filled with zeros.

Integer Bits

- For reducing the number of integer bits by *dWI*, simply right-shift the integer by *dWI*.
- The number of fractional bits is SIGN-EXTENDED by filling up the left-most bits with the sign bit.

resetN()

Reset counters and overflow-flag of Fixed object

set_qdict(*d: dict*) → None

Update the instance quantization dict *self.q_dict* from passed dict *d*:

- Sanitize *WI* and *WF*
- Calculate attributes *MSB*, *LSB*, *MIN* and *MAX*
- Calculate number of places needed for printing from *qfrmt*,`fx_base` and *W* and store it as attribute *self.places*

Check the docstring of class *Fixed()* for details on quantization

verify_q_dict_keys(*q_dict: dict*) → None

Check against *self.q_dict_default* dictionary whether all keys in the passed *q_dict* dictionary are valid.

Unknown keys throw an error message.

pyfda.libs.pyfda_fix_lib.**bin2hex**(*bin_str: str*, *WI=0*) → str

Convert number *bin_str* in binary format to hex formatted string. *bin_str* is prepended / appended with zeros until the number of bits before and after the radix point (position given by *WI*) is a multiple of 4.

pyfda.libs.pyfda_fix_lib.**bin2oct**(*bin_str: str*, *WI=0*) → str

Convert number *bin_str* in binary format to octal formatted string. *bin_str* is prepended / appended with zeros until the number of bits before and after the radix point (position given by *WI*) is a multiple of 3.

pyfda.libs.pyfda_fix_lib.**csd2dec**(*csd_str*)

Convert the CSD string *csd_str* to a decimal, *csd_str* may contain '+' or '-', indicating whether the current bit is meant to positive or negative. All other characters are simply ignored (= replaced by zero).

csd_str has to be an integer CSD number.

Parameters

csd_str (*string*) – A string with the CSD value to be converted, consisting of '+', '-', '.' and '0' characters.

Return type

decimal (integer) value of the CSD string

Examples

+00- = $+2^3$ - 2^0 = +7

-0+0 = -2^3 + 2^1 = -6

pyfda.libs.pyfda_fix_lib.**dec2csd**(*dec_val*, *WF=0*)

Convert the argument *dec_val* to a string in CSD Format.

Parameters

- **dec_val** (*scalar (integer or real)*) – decimal value to be converted to CSD format
- **WF** (*integer*) – number of fractional places. Default is WF = 0 (integer number)

Returns

- *string* – containing the CSD value
- **Original author** (*Harnesser*)
- **https** (*//sourceforge.net/projects/pycsd/*)
- **License** (*GPL2*)

pyfda.libs.pyfda_fix_lib.**dec2hex**(*val*, *nbits*, *WF=0*)

— currently not used, no unit test —

Return *val* in hex format with a wordlength of *nbits* in two's complement format. The built-in hex function returns args < 0 as negative values. When val >= 2**nbits, it is "wrapped" around to the range 0 ... 2**nbits-1

Parameters

- **val** (*integer*) – The value to be converted in decimal integer format.
- **nbits** (*integer*) – The wordlength

Return type
A string in two's complement hex format

pyfda.libs.pyfda_fix_lib.**qstr**(*text*)

carefully replace qstr() function - only needed for Py2 compatibility

pyfda.libs.pyfda_fix_lib.**quant_coeffs**(*coeffs: iterable*, *Q*, *recursive: bool = False*, *out_frmt: str = ''*) → ndarray

Quantize the coefficients, scale and convert them to a list of integers, using the quantization settings of *Fixed()* instance *Q* and global setting *fb.fil[0]['qfrmt']* (*'qfrac'* or *'qint'*) and *fb.fil[0]['fx_sim']* (*True* or *False*)

Parameters

- **coeffs** (*iterable*) – An iterable of coefficients to be quantized
- **Q** (*object*) – Instance of Fixed object containing quantization dict *q_dict*
- **recursive** (*bool*) – When *False* (default), process all coefficients. When *True*, The first coefficient is ignored (must be 1)
- **out_frmt** (*str*) – Output quantization format ("qint" or "qfrac"). When nothing is specified, use the global setting from *fb.fil[0]['qfrmt']*.

Returns

- *A numpy array of integer coeffcients, quantized and scaled with the*
- *settings of the quantization object dict.*

pyfda.libs.pyfda_fix_lib_amaranth module

Helper classes and functions for amaranth fixpoint filters

pyfda.libs.pyfda_io_lib module

Library with classes and functions for file and text IO

class pyfda.libs.pyfda_io_lib.**NumpyEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: JSONEncoder

Special json encoder for numpy and other non-supported types, building upon https://stackoverflow.com/questions/26646362/numpy-array-is-not-json-serializable

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

pyfda.libs.pyfda_io_lib.**coe_header**(*title: str*) → str

Generate a file header (comment) for various FPGA FIR coefficient export formats with information on the filter type, corner frequencies, ripple etc

Parameters
title (*str*) – A string that is written in the top of the comment section of the exported file.

Returns
header – The string with all the gathered information

Return type
str

pyfda.libs.pyfda_io_lib.**create_file_filters**(*file_types: tuple*, *file_filters: str = ''*)

Create a string with file filters for QFileDialog object from *file_types*, a tuple of file extensions and the global *file_filters_dict*.

When the file extension stored after last QFileDialog operation is in the tuple of file types, return this file extension for e.g. preselecting the file type in QFileDialog.

Parameters

- **file_types** (*tuple of str*) – list of file extensions which are used to create a file filter.
- **file_filters** (*str*) – String with file filters for QFileDialog object with the form *"Comma / Tab Separated Values (*.csv);; Audio (*.wav *.mp3)"*. By default, this string is empty, but it can be used to add file filters not contained in the global *file_filters_dict*.

Returns

- **file_filters** (*str*) – String containing file filters for a QFileDialog object
- **last_file_filter** (*str*) – Single file filter to setup the default file extension in QFileDialog

pyfda.libs.pyfda_io_lib.**csv2array**(*f: TextIO*)

Convert comma-separated values from file or text to numpy array, taking into accout the settings of the CSV dict.

Parameters

- **f** (*TextIO*) – handle to file or file-like object, e.g.
- **open(file_name** (*>>> f =*)
- **or** (*'r') #*)
- **io.StringIO(text)** (*>>> f =*)

Returns

- **data_arr** (*ndarray*) – numpy array of str with table data from file or *None* when import was unsuccessful
- *Read data as it is, splitting each row into the column items when*

- – *CSV_dict['orientation'] == cols* or
- – *CSV_dict['orientation'] == auto* and cols <= rows
- *Transpose data when*
- – *CSV_dict['orientation'] == rows* or
- – *CSV_dict['orientation'] == auto* and cols > rows
- *np.shape(data)* returns rows, columns
- While opening a file, the *newline* parameter can be used to
- *control how universal newlines works (it only applies to text mode).*
- It can be None, '', '*n*', '*r*', and '*rn*'. It works as follows
- **- Input** (If *newline == None*, universal newlines mode is enabled. Lines in) – the input can end in 'n', 'r', or 'rn', and these are translated into 'n' before being returned to the caller. If it is '', universal newline mode is enabled, but line endings are returned to the caller untranslated. If it has any of the other legal values, input lines are only terminated by the given string, and the line ending is returned to the caller untranslated.
- *- On output, if newline is None, any 'n' characters written are translated* – to the system default line separator, os.linesep. If newline is '', no translation takes place. If newline is any of the other legal values, any 'n' characters written are translated to the given string.
- **Example** (*convert from Windows-style line endings to Linux:*)
- .. *code-block:: python* – fileContents = open(filename,"r").read() f = open(filename,"w", newline="n") f.write(fileContents) f.close()
- **https** (*//pythonconquerstheuniverse.wordpress.com/2011/05/08/newline-conversion-in-python-3/*)

pyfda.libs.pyfda_io_lib.**data2array**(*parent: object*, *fkey: str*, *title: str = 'Import'*, *as_str: bool = False*)

Copy tabular data from clipboard or file to a numpy array

Parameters

- **parent** (*object*) – parent instance with a QFileDialog attribute.
- **fkey** (*str*) – Key for accessing data in *.npz file or Matlab workspace (*.mat)
- **title** (*str*) – title string for the file dialog box
- **as_str** (*bool*) – When True, return ndarray in raw str format, otherwise convert to float or complex

Returns
table data

Return type
ndarray of str or None

The following keys from the global dict `params['CSV']` are evaluated:

'delimiter'
str (default: <tab>), character for separating columns

'lineterminator'
str (default: As used by the operating system), character for terminating rows. By default, the character is selected depending on the operating system:

- Windows: Carriage return + line feed
- MacOS : Carriage return

- *nix : Line feed

'orientation'
str (one of 'auto', 'horiz', 'vert') determining with which orientation the table is read.

'header'
str (**'auto'**, 'on', 'off'). When `header=='on'`, treat first row as a header that will be discarded.

'clipboard'
bool (default: True). When `clipboard == True`, copy data from clipboard, else use a file

Parameters that are 'auto', will be guessed by `csv.Sniffer()`.

`pyfda.libs.pyfda_io_lib.`**`export_coe_TI`**(*f: TextIO*) → None

Save FIR filter coefficients in TI coefficient format Coefficient have to be specified by an identifier 'b0 ... b191' followed by the coefficient in normalized fractional format, e.g.

b0 .053647 b1 -.27485 b2 .16497 ...

** not implemented yet **

`pyfda.libs.pyfda_io_lib.`**`export_coe_cmsis`**(*f: TextIO*) → None

Get coefficients in SOS format and delete 4th column containing the '1.0' of the recursive parts.

See https://www.keil.com/pack/doc/CMSIS/DSP/html/group__BiquadCascadeDF1.html https://dsp.stackexchange.com/questions/79021/iir-design-scipy-cmsis-dsp-coefficient-format https://github.com/docPhil99/DSP/blob/master/MatlabSOS2CMSIS.m

TODO: check *scipy.signal.zpk2sos* for details concerning sos paring

`pyfda.libs.pyfda_io_lib.`**`export_coe_microsemi`**(*f: TextIO*) → bool

Save FIR filter coefficients in Microsemi coefficient format as file '*.txt'. Coefficients have to be in integer format, the last line has to be empty. For (anti)symmetric filter only one half of the coefficients must be specified?

`pyfda.libs.pyfda_io_lib.`**`export_coe_vhdl_package`**(*f: TextIO*) → bool

Save FIR filter coefficients as a VHDL package '*.vhd', specifying the number base and the quantized coefficients (decimal or hex integer).

`pyfda.libs.pyfda_io_lib.`**`export_coe_xilinx`**(*f: TextIO*) → bool

Save FIR filter coefficients in Xilinx coefficient format as file '*.coe', specifying the number base and the quantized coefficients (decimal or hex integer).

Returns error status (False if the file was saved successfully)

`pyfda.libs.pyfda_io_lib.`**`export_fil_data`**(*parent: object*, *data: str*, *fkey: str = ''*, *title: str = 'Export'*, *file_types: Tuple[str, ...] = ('csv', 'mat', 'npy', 'npz')*)

Export filter coefficients or pole/zero data in various formats, file name and type are selected via the ui.

Parameters

- **parent** (*handle to calling instance for creating file dialog instance*)
- **data** (*str*) – formatted as CSV data, i.e. rows of elements separated by 'delimiter', terminated by 'lineterminator'. Some data formats
- **fkey** (*str*) – Key for accessing data in `*.npz` or Matlab workspace (`*.mat`) file. When fkey == 'ba', exporting to FPGA coefficients format is enabled.
- **title** (*str*) – title string for the file dialog box (e.g. "filter coefficients ")
- **file_types** (*tuple of strings*) – file extension (e.g. *(csv)* or list of file extensions (e.g. *(csv, txt)* which are used to create a file filter.

pyfda.libs.pyfda_io_lib.**extract_file_ext**(*file_type: str*, *return_list: bool = False*) → str

Extract list with file extension(s), e.g. '.vhd' from type description 'VHDL (*.vhd)' returned by QFileDialog. Depending on the OS, this may be the full file type description or just the extension like '(*.vhd)'.

When *file_type* contains no '(', the passed string is returned unchanged.

For an explanation of the RegEx, see the docstring for *prune_file_ext*.

Parameters

- **file_type** (*str*)
- **return_list** (*bool (default = False)*) – When True, return a list with file extensions (possibly empty or with only one element), when False (default) only return the first element (a string)

Returns

The file extension between (...), e.g. 'csv' or the list of file extension or the unchanged input argument *file_type* when no '(' was contained.

Return type

str or list of str

pyfda.libs.pyfda_io_lib.**load_data_np**(*file_name: str*, *file_type: str*, *fkey: str = ''*, *as_str: bool = False*) → ndarray

Import data from a file and convert it to a numpy array.

Parameters

- **file_name** (*str*) – Full path and name of the file to be imported
- **file_type** (*str*) – File type, currently supported are 'csv', 'mat', 'npy', 'npz, 'txt', 'wav'.
- **fkey** (*str*) – Key for accessing data in *.npz or Matlab workspace (*.mat) files with multiple entries.
- **as_str** (*bool*) – When False (default), try to convert results to ndarray of float or complex. Otherwise, return an ndarray of str.

Returns

Data from the file (ndarray) or None (error)

Return type

ndarray of float / complex / int or str

pyfda.libs.pyfda_io_lib.**load_filter**(*self*) → int

Load filter from JSON, zipped binary numpy array or (c)pickled object to filter dictionary

pyfda.libs.pyfda_io_lib.**prune_file_ext**(*file_type: str*) → str

Prune file extension, e.g. 'Text file' from 'Text file (*.txt)' returned by QFileDialog file type description.

Pruning is achieved with the following regular expression:

```
return = re.sub('\([^\)]+\)', '', file_type)
```

Parameters

file_type (*str*)

Returns

The pruned file description

Return type

str

Notes

Syntax of python regex: `re.sub(pattern, replacement, string)`

This returns the string obtained by replacing the leftmost non-overlapping occurrences of `pattern` in `string` by `replacement`.

- '.' means any character
- '+' means one or more
- '[^a]' means except for 'a'
- '([^)]+)' : match '(', gobble up all characters except ')' till ')'
- '(' must be escaped as '\('

pyfda.libs.pyfda_io_lib.**qtable2csv**(*table: object*, *data: ndarray*, *zpk=False*, *formatted: bool = False*) → str

Transform QTableWidget data to CSV formatted text

Parameters

- **table** (*object*) – Instance of QTableWidget
- **data** (*object*) – Instance of the numpy variable shadowing table data
- **zpk** (*bool*) – when True, append the gain (*data[2]*) to the table
- **formatted** (*bool*) – When True, copy data as formatted in the table, otherwise copy from the model ("shadow").

The following keys from the global dict dict `params['CSV']` are evaluated:

'delimiter'
str (default: ","), character for separating columns

'lineterminator'
str (default: As used by the operating system), character for terminating rows. By default, the character is selected depending on the operating system:

- Windows: Carriage return + line feed
- MacOS : Carriage return
- *nix : Line feed

'orientation'
str (one of 'auto', 'horiz', 'vert') determining with which orientation the table is written. 'vert' means a line break after each entry or pair of entries which usually is not what you want. 'auto' doesn't make much sense when writing, 'horiz' is used in this case.

'header'
str (default: 'auto'). When `header='on'`, write the first row with 'b, a'.

'clipboard'
bool (default: True), when `clipboard == True`, copy data to clipboard, else use a file.

Returns
Nothing, text is exported to clipboard or to file via `export_fil_data`

Return type
None

pyfda.libs.pyfda_io_lib.**read_csv_info_old**(*filename*)

DON'T USE ANYMORE! Get infos about the size of a csv file without actually loading the whole file into memory.

See https://stackoverflow.com/questions/64744161/best-way-to-find-out-number-of-rows-in-csv-without-loading-the-full-thi

pyfda.libs.pyfda_io_lib.**read_wav_info**(*file*)

Get infos about the following properties of a wav file without actually loading the whole file into memory. This is achieved by reading the header.

pyfda.libs.pyfda_io_lib.**save_data_np**(*file_name: str*, *file_type: str*, *data: ndarray*, *f_S: int = 1*, *fmt: str = '%f'*) → int

Save numpy ndarray data to a file in wav or csv format

Parameters

- **file_name** (*str*) – Full path and name of the file to be imported
- **file_type** (*str*) – File type, currently supported are 'csv' or 'wav'
- **data** (*np.ndarray*) – Data to be saved to a file. The data dtype (uint8, int16, int32, float32) determines the bits-per-sample and PCM/float of the WAV file
- **f_S** (*int (optional)*) – Sampling frequency (only used for WAV file format), only integer sampling frequencies are supported by the WAV format.
- **fmt** (*str (optional)*) – Optional, default '%f'. Format string, only used for exporting data in CSV format. Other options are e.g. '%1.2f' for reduced number of digits, '%d' for integer format or '%s' for strings.

Return type

0 for success, -1 for file cancel or error

pyfda.libs.pyfda_io_lib.**save_filter**(*self*)

Save filter as JSON formatted textfile, zipped binary numpy array or pickle object

pyfda.libs.pyfda_io_lib.**select_file**(*parent: object*, *title: str = ''*, *mode: str = 'r'*, *file_types: Tuple[str, ...] = ('csv', 'txt')*) → Tuple[str, str]

Select a file from a file dialog box for either reading or writing and return the selected file name and type.

Parameters

- **title** (*str*) – title string for the file dialog box (e.g. "Filter Coefficients"),
- **mode** (*str*) – file access mode, must be either "r" or "w" for read / write access
- **file_types** (*tuple of str*) – supported file types, e.g. *('txt', 'npy', 'mat') which need to be keys of `file_filters_dict*

Returns

- **file_name** (*str*) – Fully qualified name of selected file. *None* when operation has been cancelled.
- **file_type** (*str*) – File type, e.g. 'wav'. *None* when operation has been cancelled.

pyfda.libs.pyfda_io_lib.**write_wav_frame**(*parent*, *file_name*, *data: array*, *f_S=1*, *title: str = 'Export'*)

Export a frame of data in wav format

Parameters

- **parent** (*handle to calling instance for creating file dialog instance*)
- **data** (*np.array*) – data to be exported
- **title** (*str*) – title string for the file dialog box (e.g. "audio data ")

pyfda.libs.pyfda_lib module

Library with various general functions and variables needed by the pyfda routines

pyfda.libs.pyfda_lib.**H_mag**(*num*, *den*, *z*, *H_max*, *H_min=None*, *log=False*, *div_by_0='ignore'*)

Calculate |*H(z)*| at the complex frequency(ies) z (scalar or array-like). The function *H(z)* is given in polynomial form with numerator and denominator. When `log == True`, $20 \log_{10}(|H(z)|)$ is returned.

The result is clipped at H_min, H_max; clipping can be disabled by passing None as the argument.

Parameters

- **num** (*float or array-like*) – The numerator polynome of H(z).
- **den** (*float or array-like*) – The denominator polynome of H(z).
- **z** (*float or array-like*) – The complex frequency(ies) where *H(z)* is to be evaluated
- **H_max** (*float*) – The maximum value to which the result is clipped
- **H_min** (*float, optional*) – The minimum value to which the result is clipped (default: None)
- **log** (*boolean, optional*) – When true, return 20 * log10 (|H(z)|). The clipping limits have to be given as dB in this case.
- **div_by_0** (*string, optional*) – What to do when division by zero occurs during calculation (default: 'ignore'). As the denomintor of H(z) becomes 0 at each pole, warnings are suppressed by default. This parameter is passed to numpy.seterr(), hence other valid options are 'warn', 'raise' and 'print'.

Returns

H_mag – The magnitude |*H(z)*| for each value of z.

Return type

float or ndarray

pyfda.libs.pyfda_lib.**calc_Hcomplex**(*fil_dict*, *worN*, *wholeF*, *fs=6.283185307179586*)

A wrapper around *signal.freqz()* for calculating the complex frequency response H(f) for antiCausal systems as well. The filter coefficients are are extracted from the filter dictionary.

Parameters

- **fil_dict** (*dict*) – dictionary with filter data (coefficients etc.)
- **worN** (*{None, int or array-like}*) – number of points or frequencies where the frequency response is calculated
- **wholeF** (*bool*) – when True, calculate frequency response from 0 ... f_S, otherwise calculate between 0 ... f_S/2
- **fs** (*float*) – sampling frequency, used for calculation of the frequency vector. The default is 2*pi

Returns

- **w** (*ndarray*) – The frequencies at which h was computed, in the same units as fs. By default, w is normalized to the range [0, pi) (radians/sample).
- **h** (*ndarray*) – The frequency response, as complex numbers.

Examples

pyfda.libs.pyfda_lib.**ceil_even**(*x*) → int

Return the smallest even integer not less than x. x can be integer or float.

pyfda.libs.pyfda_lib.**ceil_odd**(*x*) → int

Return the smallest odd integer not less than x. x can be integer or float.

pyfda.libs.pyfda_lib.**clean_ascii**(*arg*)

Remove non-ASCII-characters (outside range 0 ... x7F) from *arg* when it is a *str*. Otherwise, return *arg* unchanged.

Parameters
arg (*str*) – This is a unicode string under Python 3

Returns

arg – Input string, cleaned from non-ASCII characters when *arg* is a string

or

Unchanged parameter *arg* when not a string

Return type
str

pyfda.libs.pyfda_lib.**cmp_version**(*mod: str*, *version: str*) → int

Compare version number of installed module *mod* against value *version* (str) and return 1, 0 or -1 if the installed version is greater, equal or less than the number in *version*. If *mod* is not installed, return -2.

Parameters

- **mod** (*str*) – name of the module to be compared
- **version** (*str*) – version number in the form e.g. "0.1.6"

Returns

result –

one of the following error codes:

-2
module is not installed

-1
version of installed module is lower than the specified version

0
version of installed module is equal to specied version

1
version of installed module is higher than specified version

Return type
int

pyfda.libs.pyfda_lib.**cmplx_sort**(*p*)

sort roots based on magnitude.

pyfda.libs.pyfda_lib.**croundd**(*x*, *n_dig=0*)

Round complex number to n_dig digits. If n_dig == 0, don't round at all, just convert complex numbers with an imaginary part very close to zero to real.

pyfda.libs.pyfda_lib.**dB**(*lin: float*, *power: bool = False*) → float

Calculate dB from linear value. If power = True, calculate 10 log ..., else calculate 20 log ...

pyfda.libs.pyfda_lib.**expand_lim**(*ax*, *eps_x: float*, *eps_y: float = None*) → None

Expand the xlim and ylim-values of passed axis by eps

Parameters

- **ax** (*axes object*)
- **eps_x** (*float*) – factor by which x-axis limits are expanded
- **eps_y** (*float*) – factor by which y-axis limits are expanded. If eps_y is None, eps_x is used for eps_y as well.

Return type
None

pyfda.libs.pyfda_lib.**fil_convert**(*fil_dict: dict*, *format_in*) → None

Convert between poles / zeros / gain, filter coefficients (polynomes) and second-order sections and store all formats not generated by the filter design routine in the passed dictionary `fil_dict`.

Parameters

- **fil_dict** (*dict*) – filter dictionary containing a.o. all formats to be read and written.
- **format_in** (*string or set of strings*) – format(s) generated by the filter design routine. Must be one of

  'sos'
  a list of second order sections - all other formats can easily be derived from this format

  'zpk'
  [z,p,k] where z is the array of zeros, p the array of poles and k is a scalar with the gain - the polynomial form can be derived from this format quite accurately

  'ba'
  [b, a] where b and a are the polynomial coefficients - finding the roots of the a and b polynomes may fail for higher orders

Returns

- *None*
- *Exceptions*
- ———-
- *ValueError for Nan / Inf elements or other unsuitable parameters*

pyfda.libs.pyfda_lib.**fil_save**(*fil_dict: dict*, *arg*, *format_in: str*, *sender: str*, *convert: bool = True*) → None

Save filter design `arg` given in the format specified as `format_in` in the dictionary `fil_dict`. The format can be either poles / zeros / gain, filter coefficients (polynomes) or second-order sections.

Convert the filter design to the other formats if `convert` is True.

Parameters

- **fil_dict** (*dict*) – The dictionary where the filter design is saved to.
- **arg** (*various formats*) – The actual filter design
- **format_in** (*string*) – Specifies how the filter design in 'arg' is passed:

  'ba'
  Coefficient form: Filter coefficients in FIR format (b, one dimensional) are automatically converted to IIR format (b, a).

'zpk'
Zero / pole / gain format: When only zeroes are specified, poles and gain are added automatically.

'sos'
Second-order sections

- **sender** (*string*) – The name of the method that calculated the filter. This name is stored in `fil_dict` together with `format_in`.
- **convert** (*boolean*) – When `convert = True`, convert arg to the other formats.

Return type
None

`pyfda.libs.pyfda_lib.floor_even`(*x*) → int

Return the largest even integer not larger than x. x can be integer or float.

`pyfda.libs.pyfda_lib.floor_odd`(*x*) → int

Return the largest odd integer not larger than x. x can be integer or float.

`pyfda.libs.pyfda_lib.format_ticks`(*ax*, *xy: str*, *scale: float = 1.0*, *format: str = '%.1f'*) → None

Reformat numbers at x or y - axis. The scale can be changed to display e.g. MHz instead of Hz. The number format can be changed as well.

Parameters

- **ax** (*axes object*)
- **xy** (*string, either 'x', 'y' or 'xy'*) – select corresponding axis (axes) for reformatting
- **scale** (*float (default: 1.)*) – rescaling factor for the axes
- **format** (*string (default: %.1f)*) – define C-style number formats

Return type
None

Examples

Scale all numbers of x-Axis by 1000, e.g. for displaying ms instead of s.

```
>>> format_ticks('x',1000.)
```

Two decimal places for numbers on x- and y-axis

```
>>> format_ticks('xy',1., format = "%.2f")
```

`pyfda.libs.pyfda_lib.lin2unit`(*lin_value: float*, *filt_type: str*, *amp_label: str*, *unit: str = 'dB'*) → float

Convert linear amplitude specification to dB or W, depending on filter type ('FIR' or 'IIR') and whether the specifications belong to passband or stopband. This is determined by checking whether amp_label contains the strings 'PB' or 'SB' :

- **Passband:**

$$\text{IIR:} \quad A_{dB} = -20\log_{10}(1 - lin\_value)$$
$$\text{FIR:} \quad A_{dB} = 20\log_{10}\frac{1 + lin\_value}{1 - lin\_value}$$

- **Stopband:**

$$A_{dB} = -20\log_{10}(lin\_value)$$

Returns the result as a float.

pyfda.libs.pyfda_lib.**mod_version**(*mod: str = ''*) → str

Return the version of the module 'mod'. If the module is not found, return empty string. When no module is specified, return a string with all modules and their versions sorted alphabetically.

pyfda.libs.pyfda_lib.**qstr**(*text*)

Convert text (QVariant, QString, string) or numeric object to plain string.

In Python 3, python Qt objects are automatically converted to QVariant when stored as "data" (itemData) e.g. in a QComboBox and converted back when retrieving to QString. In Python 2, QVariant is returned when itemData is retrieved. This is first converted from the QVariant container format to a QString, next to a "normal" non-unicode string.

Parameters

text (*QVariant, QString, string or numeric data type that can be converted*) – to string

Return type

The current *text* data as a unicode (utf8) string

pyfda.libs.pyfda_lib.**round_even**(*x*) → int

Return the nearest even integer from x. x can be integer or float.

pyfda.libs.pyfda_lib.**round_odd**(*x*) → int

Return the nearest odd integer from x. x can be integer or float.

pyfda.libs.pyfda_lib.**safe_eval**(*expr*, *alt_expr=0*, *return_type: str = 'float'*, *sign: str = None*)

Try ... except wrapper around numexpr to catch various errors When evaluation fails or returns *None*, try evaluating *alt_expr*. When this also fails, return 0 to avoid errors further downstream.

Parameters

- **expr** (*str or scalar*) – Expression to be evaluated, is cast to a string
- **alt_expr** (*str or scalar*) – Expression to be evaluated when evaluation of first string fails, is cast to a string.
- **return_type** (*str*) – Type of returned variable ['float' (default) / 'cmplx' / 'int' / '' or 'auto']
- **sign** (*str*) –

  enforce positive / negative sign of result ['pos', 'poszero' / None (default) 'negzero' / 'neg']

Returns

the evaluated result or 0 when both arguments fail

Return type

float (default) / complex / int

Function attribute *err* contains number of errors that have occurred during evaluation (0 / 1 / 2)

pyfda.libs.pyfda_lib.**set_dict_defaults**(*d: dict*, *default_dict: dict*) → None

Add the key:value pairs of *default_dict* to dictionary *d* in-place for all missing keys.

pyfda.libs.pyfda_lib.**sos2zpk**(*sos*)

Taken from scipy/signal/filter_design.py - edit to eliminate first order section

Return zeros, poles, and gain of a series of second-order sections

Parameters

sos (*array_like*) – Array of second-order filter coefficients, must have shape `(n_sections, 6)`. See *sosfilt* for the SOS filter format specification.

Returns

- **z** (*ndarray*) – Zeros of the transfer function.
- **p** (*ndarray*) – Poles of the transfer function.
- **k** (*float*) – System gain.

Notes

Added in version 0.16.0.

pyfda.libs.pyfda_lib.**to_html**(*text: str*, *frmt: str = None*) → str

Convert text to HTML format:

- pretty-print logger messages
- convert "n" to "
- convert "< " and "> " to "<" and ">"
- format strings with italic and / or bold HTML tags, depending on parameter *frmt*. When *frmt=None*, put the returned string between tags to enforce HTML rendering downstream
- replace '_' by HTML subscript tags. Numbers 0 … 9 are never set to italic format

Parameters

- **text** (*str*) – Text to be converted
- **frmt** (*str*) – define text style
  - 'b' : bold text
  - 'i' : italic text
  - 'bi' or 'ib' : bold and italic text

Returns
HTML - formatted text

Return type
str

Examples

```
>>> to_html("F_SB", frmt='bi')
"<b><i>F<sub>SB</sub></i></b>"
>>> to_html("F_1", frmt='i')
"<i>F</i><sub>1</sub>"
```

pyfda.libs.pyfda_lib.**unique_roots**(*p*, *tol: float = 0.001*, *magsort: bool = False*, *rtype: str = 'min'*, *rdist: str = 'euclidian'*)

Determine unique roots and their multiplicities from a list of roots.

Parameters

- **p** (*array_like*) – The list of roots.
- **tol** (*float, default tol = 1e-3*) – The tolerance for two roots to be considered equal. Default is 1e-3.
- **magsort** (*Boolean, default = False*) – When magsort = True, use the root magnitude as a sorting criterium (as in the version used in numpy < 1.8.2). This yields false results for roots with similar magniudes (e.g. on the unit circle) but is signficantly faster for a large number of roots (factor 20 for 500 double roots.)

- **rtype** (*{'max', 'min', 'avg'}, optional*) – How to determine the returned root if multiple roots are within *tol* of each other. - 'max' or 'maximum': pick the maximum of those roots (magnitude ?). - 'min' or 'minimum': pick the minimum of those roots (magnitude?). - 'avg' or 'mean' : take the average of those roots. - 'median' : take the median of those roots
- **dist** (*{'manhattan', 'euclid'}, optional*) – How to measure the distance between roots: 'euclid' is the euclidian distance. 'manhattan' is less common, giving the sum of the differences of real and imaginary parts.

Returns

- **pout** (*list*) – The list of unique roots, sorted from low to high (only for real roots).
- **mult** (*list*) – The multiplicity of each root.

Notes

This utility function is not specific to roots but can be used for any sequence of values for which uniqueness and multiplicity has to be determined. For a more general routine, see *numpy.unique*.

Examples

```
>>> vals = [0, 1.3, 1.31, 2.8, 1.25, 2.2, 10.3]
>>> uniq, mult = unique_roots(vals, tol=2e-2, rtype='avg')
```

Check which roots have multiplicity larger than 1:

```
>>> uniq[mult > 1]
array([ 1.305])
```

Find multiples of complex roots on the unit circle: >>> vals = np.roots(1,2,3,2,1) uniq, mult = unique_roots(vals, rtype='avg')

pyfda.libs.pyfda_lib.**unit2lin**(*unit_value: float*, *filt_type: str*, *amp_label: str*, *unit: str = 'dB'*) → float

Convert amplitude specification in dB or W to linear specs:

- **Passband:**

$$\text{IIR:} \quad A_{PB,lin} = 1 - 10^{-unit\_value/20}$$

$$\text{FIR:} \quad A_{PB,lin} = \frac{10^{unit\_value/20} - 1}{10^{unit\_value/20} + 1}$$

- **Stopband:**

$$A_{SB,lin} = -10^{-unit\_value/20}$$

Returns the result as a float.

pyfda.libs.pyfda_qt_lib module

Library with various helper functions for Qt widgets

class pyfda.libs.pyfda_qt_lib.**EventTypes**

Bases: object

https://stackoverflow.com/questions/62196835/how-to-get-string-name-for-qevent-in-pyqt5 Events in Qt5: https://doc.qt.io/qt-5/qevent.html

Stores a string name for each event type.

With PySide2 str() on the event type gives a nice string name, but with PyQt5 it does not. So this method works with both systems.

Example usage (simultaneous initialization and method call / translation) > event_str = EventTypes().as_string(QEvent.UpdateRequest) > assert event_str == "UpdateRequest"

Example usage, separate initialization and method call > event types = EventTypes() > event_str = event_types.as_string(event.type())

as_string(*event: Type*) → str

Return the string name for this event.

class pyfda.libs.pyfda_qt_lib.**PushButton**(*txt: str = ''*, *icon: QIcon = None*, *N_x: int = 8*, *checkable: bool = True*, *checked: bool = False*, *objectName=''*)

Bases: QPushButton

Create a QPushButton with a width fitting the label with bold font as well

Parameters

- **txt** (*str*) – Text for button (optional)
- **icon** (*QIcon*) – Icon for button. Either *txt* or *icon* must be defined.
- **N_x** (*int*) – Width in number of "x"
- **checkable** (*bool*) – Whether button is checkable
- **checked** (*bool*) – Whether initial state is checked

minimumSizeHint(*self*) → QSize

sizeHint(*self*) → QSize

class pyfda.libs.pyfda_qt_lib.**QHLine**(*width=1*)

Bases: QFrame

Create a thin horizontal line utilizing the HLine property of QFrames Usage:

> myline = QHLine() > mylayout.addWidget(myline)

class pyfda.libs.pyfda_qt_lib.**QLabelVert**(*text*, *orientation='west'*, *forceWidth=True*)

Bases: QLabel

Create a vertical label

Adapted from https://pyqtgraph.readthedocs.io/en/latest/_modules/pyqtgraph/widgets/VerticalLabel.html

https://stackoverflow.com/questions/34080798/pyqt-draw-a-vertical-label

check https://stackoverflow.com/questions/29892203/draw-rich-text-with-qpainter

minimumSizeHint(*self*) → QSize

paintEvent(*self*, *a0: QPaintEvent | None*)

sizeHint(*self*) → QSize

class pyfda.libs.pyfda_qt_lib.**QVLine**(*width=2*)

Bases: QFrame

Create a thin vertical line utilizing the HLine property of QFrames Usage:

> myline = QVLine() > mylayout.addWidget(myline)

class pyfda.libs.pyfda_qt_lib.**RotatedButton**

Bases: QPushButton

Create a rotated QPushButton

Taken from

https://forum.qt.io/topic/9279/moved-how-to-rotate-qpushbutton-63/7

getSyleOptions()

init(*text*, *parent*, *orientation='west'*)

minimumSizeHint(*self*) → QSize

paintEvent(*self*, *a0: QPaintEvent | None*)

sizeHint(*self*) → QSize

pyfda.libs.pyfda_qt_lib.**emit**(*self*, *dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

pyfda.libs.pyfda_qt_lib.**popup_warning**(*self*, *param1: int = 0*, *param2: str = ''*, *message: str = ''*) → bool

Pop-up a warning box and require a user prompt. When *message == ""*, warn of very large filter orders, otherwise display the passed message

pyfda.libs.pyfda_qt_lib.**qcmb_box_add_item**(*cmb_box*, *item_list*, *data=True*, *fireSignals=False*, *caseSensitive=False*)

Add an entry in combobox with text / data / tooltipp from *item_list*. When the item is already in combobox (searching for data or text item, depending *data*), do nothing. Signals are blocked during the update of the combobox unless *fireSignals* is set *True*. By default, the search is case insensitive, this can be changed by passing *caseSensitive=False*.

Parameters

- **item_list** (*list*) – List with *["new_data", "new_text", "new_tooltip"]* to be added.
- **data** (*bool (default: False)*) – Whether the string refers to the data or text fields of the combo box
- **fireSignals** (*bool (default: False)*) – When True, fire a signal if the index is changed (useful for GUI testing)
- **caseInsensitive** (*bool (default: False)*) – When true, perform case sensitive search.

Returns

- *The index of the found item with string / data. When not found in the*
- *combo box, return index -1.*

pyfda.libs.pyfda_qt_lib.**qcmb_box_add_items**(*cmb_box: QComboBox*, *items_list: list*) → None

Add items to combo box *cmb_box* with text, data and tooltip from the list *items_list*.

Text and tooltip are prepared for translation via *self.tr()*

Parameters

- **cmb_box** (*instance of QComboBox*) – Combobox to be populated
- **items_list** (*list*) –

  List of combobox entries, in the format
  ("data 1st item", "text 1st item", "tooltip for 1st item" # [optional]), ("data 2nd item", "text 2nd item", "tooltip for 2nd item")]

Return type
None

pyfda.libs.pyfda_qt_lib.**qcmb_box_del_item**(*cmb_box: QComboBox*, *string: str*, *data: bool = False*, *fireSignals: bool = False*, *caseSensitive: bool = False*) → int

Try to find the entry in combobox corresponding to *string* in a text field (*data = False*) or in a data field (*data=True*) and delete the item. When *string* is not found,do nothing. Signals are blocked during the update of the combobox unless *fireSignals* is set *True*. By default, the search is case insensitive, this can be changed by passing *caseSensitive=False*.

Parameters

- **string** (*str*) – The label in the text or data field to be deleted.
- **data** (*bool (default: False)*) – Whether the string refers to the data or text fields of the combo box
- **fireSignals** (*bool (default: False)*) – When True, fire a signal if the index is changed (useful for GUI testing)
- **caseInsensitive** (*bool (default: False)*) – When true, perform case sensitive search.

Returns

- *The index of the item with string / data. When not found in the combo box,*
- *return index -1.*

pyfda.libs.pyfda_qt_lib.**qcmb_box_populate**(*cmb_box: QComboBox*, *items_list: list*, *item_init: str*) → int

Clear and populate combo box *cmb_box* with text, data and tooltip from the list *items_list* with initial selection of *init_item* (data).

Text and tooltip are prepared for translation via *self.tr()*

Parameters

- **cmb_box** (*instance of QComboBox*) – Combobox to be populated
- **items_list** (*list*) – List of combobox entries, in the format ["Tooltip for Combobox", ("data 1st item", "text 1st item", "tooltip for 1st item"), ("data 2nd item", "text 2nd item", "tooltip for 2nd item")]

  Tooltipps are optional.
- **item_init** (*str*) – data for initial setting of combobox. When data is not found, set combobox to first item.

Returns
ret – Index of *item_init* in combobox. If index == -1, *item_init* was not in *items_list*

Return type
int

pyfda.libs.pyfda_qt_lib.**qget_cmb_box**(*cmb_box: QComboBox*, *data: bool = True*) → str

Get current itemData or Text of comboBox and convert it to string.

In Python 3, python Qt objects are automatically converted to QVariant when stored as "data" e.g. in a QComboBox and converted back when retrieving. In Python 2, QVariant is returned when itemData is retrieved. This is first converted from the QVariant container format to a QString, next to a "normal" non-unicode string.

Returns:

The current text or data of combobox as a string

pyfda.libs.pyfda_qt_lib.**qget_selected**(*table*, *select_all=False*, *reverse=True*)

Get selected cells in `table` and return a dictionary with the following keys:

'idx': indices of selected cells as an unsorted list of tuples

'sel': list of lists of selected cells per column, by default sorted in reverse

'cur': current cell selection as a tuple

Parameters

- **select_all** (*bool*) – select all table items and create a list when True
- **reverse** (*bool*) – return selected fields upside down when True

pyfda.libs.pyfda_qt_lib.**qset_cmb_box**(*cmb_box: QComboBox*, *string: str*, *data: bool = False*, *fireSignals: bool = False*, *caseSensitive: bool = False*) → int

Set combobox to the index corresponding to *string* in a text field (*data = False*) or in a data field (*data=True*). When *string* is not found in the combobox entries, select the first entry. Signals are blocked during the update of the combobox unless *fireSignals* is set *True*. By default, the search is case insensitive, this can be changed by passing *caseSensitive=False*.

Parameters

- **string** (*str*) – The label in the text or data field to be selected. When the string is not found, select the first entry of the combo box.
- **data** (*bool (default: False)*) – Whether the string refers to the data or text fields of the combo box
- **fireSignals** (*bool (default: False)*) – When True, fire a signal if the index is changed (useful for GUI testing)
- **caseSensitive** (*bool (default: False)*) – When true, perform case sensitive search.

Returns

- *The index of the string. When the string was not found in the combo box,*
- *select first entry of combo box and return index -1.*

pyfda.libs.pyfda_qt_lib.**qstyle_widget**(*widget*, *state*)

Apply the "state" defined in pyfda_rc.py to the widget, e.g.: Color the >> DESIGN FILTER << button according to the filter design state.

This requires setting the property, "unpolishing" and "polishing" the widget and finally forcing an update.

- 'normal' : default, no color styling
- 'ok': green, filter has been designed, everything ok
- 'changed': yellow, filter specs have been changed
- 'running': orange, simulation is running

- 'error' : red, an error has occurred during filter design
- 'failed' : pink, filter fails to meet target specs (not used yet)
- 'u' or 'unused' : grey text color
- 'd' or 'disabled': background color darkgrey
- 'a' or 'active' : no special style defined

pyfda.libs.pyfda_qt_lib.**qtext_height**(*text: str = 'X'*, *font=None*) → int

Calculate size of *text* in points`.

The actual size of the string is calculated using fontMetrics and the default or the passed font

Parameters
test (*str*) – string to calculate the height for (default: "X")

Returns
lineSpacing – The height of the text (line spacing) in points

Return type
int

Notes

This is based on https://stackoverflow.com/questions/27433165/how-to-reimplement-sizehint-for-bold-text-in-a-delegate-qt

and

https://stackoverflow.com/questions/47285303/how-can-i-limit-text-box-width-of- # qlineedit-to-display-at-most-four-characters/47307180#47307180

https://stackoverflow.com/questions/56282199/fit-qtextedit-size-to-text-size-pyqt5

pyfda.libs.pyfda_qt_lib.**qtext_width**(*text: str = ''*, *N_x: int = 17*, *bold: bool = True*, *font=None*) → int

Calculate width of *text* in points`. When *text*= `, calculate the width of number *N_x* of characters 'x'.

The actual width of the string is calculated by creating a QTextDocument with the passed text and retrieving its *idealWidth()*

Parameters
- **test** (*str*) – string to calculate the width for
- **N_x** (*int*) – When *text* == '', calculate the width from *N_x * width('x')*
- **bold** (*bool (default: True)*) – When *True*, determine width based on bold font

Returns
width – The width of the text in points

Return type
int

Notes

This is based on https://stackoverflow.com/questions/27433165/how-to-reimplement-sizehint-for-bold-text-in-a-delegate-qt

and

https://stackoverflow.com/questions/47285303/how-can-i-limit-text-box-width-of- # qlineedit-to-display-at-most-four-characters/47307180#47307180

pyfda.libs.pyfda_qt_lib.**qwindow_stay_on_top**(*win: QDialog*, *top: bool*) → None

Set flags for a window such that it stays on top (True) or not

On Windows 7 the new window stays on top anyway. Additionally setting WindowStaysOnTopHint blocks the message window when trying to close pyfda.

On Windows 10 and Linux, *WindowStaysOnTopHint* needs to be set.

pyfda.libs.pyfda_sig_lib module

Library with various signal processing related functions

pyfda.libs.pyfda_sig_lib.**angle_zero**(*X*, *n_eps=1000.0*, *mode='auto'*, *wrapped='auto'*)

Calculate angle of argument *X* when abs(X) > *n_eps* * machine resolution. Otherwise, zero is returned.

pyfda.libs.pyfda_sig_lib.**div_safe**(*num*, *den*, *n_eps: float = 1.0*, *i_scale: float = 1.0*, *verbose: bool = False*)

Perform elementwise array division after treating singularities, meaning:

- check whether denominator (*den*) coefficients approach zero
- check whether numerator (*num*) or denominator coefficients are non-finite, i.e. one of *nan*, *inf* or *ninf*.

At each singularity, replace denominator coefficient by *1* and numerator coefficient by *0*

Parameters

- **num** (*array_like*) – numerator coefficients
- **den** (*array_like*) – denominator coefficients
- **n_eps** (*float*) – n_eps * machine resolution is the limit for the denominator below which the ratio is set to zero. The machine resolution in numpy is given by *np.spacing(1)*, the distance to the nearest number which is equivalent to matlab's "eps".
- **i_scale** (*float*) – The scale for the index *i* for *num, den* for printing the index of the singularities.
- **verbose** (*bool, optional*) – whether to print The default is False.

Returns

ratio – The ratio of num and den (zero at singularities)

Return type

array_like

pyfda.libs.pyfda_sig_lib.**group_delay**(*b*, *a=1*, *nfft=512*, *whole=False*, *analog=False*, *verbose=True*, *fs=6.283185307179586*, *sos=False*, *alg='scipy'*, *n_eps=100*)

Calculate group delay of a discrete time filter, specified by numerator coefficients *b* and denominator coefficients *a* of the system function *H* (*z*).

When only *b* is given, the group delay of the transversal (FIR) filter specified by *b* is calculated.

Parameters

- **b** (*array_like*) – Numerator coefficients (transversal part of filter)
- **a** (*array_like (optional, default = 1 for FIR-filter)*) – Denominator coefficients (recursive part of filter)
- **whole** (*boolean (optional, default : False)*) – Only when True calculate group delay around the complete unit circle (0 … 2 pi)
- **verbose** (*boolean (optional, default : True)*) – Print warnings about frequency points with undefined group delay (amplitude = 0) and the time used for calculating the group delay

- **nfft** (*integer (optional, default: 512)*) – Number of FFT-points
- **fs** (*float (optional, default: fs = 2*pi)*) – Sampling frequency.
- **alg** (*str (default: "scipy")*) –

  The algorithm for calculating the group delay:

  - "scipy" The algorithm used by scipy's grpdelay,
  - "jos": The original J.O.Smith algorithm; same as in "scipy" except that the frequency response is calculated with the FFT instead of polyval
  - "diff": Group delay is calculated by differentiating the phase
  - "Shpakh": Group delay is calculated from second-order sections
- **n_eps** (*integer (optional, default : 100)*) – Minimum value in the calculation of intermediate values before tau_g is set to zero.

Returns

- **tau_g** (*ndarray*) – group delay
- **w** (*ndarray*) – angular frequency points where group delay was computed

Notes

The following explanations follow [JOS].

Definition and direct calculation ('diff')

The group delay $\tau_g(\omega)$ of discrete time (DT) and continuous time (CT) systems is the rate of change of phase with respect to angular frequency. In the following, derivative is always meant w.r.t. ω:

$$\tau_g(\omega) = -\frac{\partial}{\partial\omega}\angle H(\omega) = -\frac{\partial\phi(\omega)}{\partial\omega} = -\phi'(\omega)$$

With numpy / scipy, the group delay can be calculated directly with

```
w, H = sig.freqz(b, a, worN=nfft, whole=whole)
tau_g = -np.diff(np.unwrap(np.angle(H)))/np.diff(w)
```

The derivative can create numerical problems for e.g. phase jumps at zeros of frequency response or when the complex frequency response becomes very small e.g. in the stop band.

This can be avoided by calculating the group delay from the derivative of the *logarithmic* frequency response in polar form (amplitude response and phase):

$$\begin{aligned}\ln(H(\omega)) = \ln\left(H_A(\omega)e^{j\,\phi(\omega)}\right) &= \ln\left(H_A(\omega)\right) + j\phi(\omega) \\ \Rightarrow\ \frac{\partial}{\partial\omega}\ln(H(\omega)) &= \frac{H'_A(\omega)}{H_A(\omega)} + j\phi'(\omega)\end{aligned}$$

where $H_A(\omega)$ is the amplitude response. $H_A(\omega)$ and its derivative $H'_A(\omega)$ are real-valued, therefore, the group delay can be calculated by separating real and imginary components (and discarding the real part):

$$\Re\left\{\frac{\partial}{\partial\omega}\ln(H(\omega))\right\} = \frac{H'_A(\omega)}{H_A(\omega)}\ \Im\left\{\frac{\partial}{\partial\omega}\ln(H(\omega))\right\} \qquad = \phi'(\omega) \tag{4.1}$$

and hence

$$\tau_g(\omega) = -\phi'(\omega) = -\Im\left\{\frac{\partial}{\partial\omega}\ln(H(\omega))\right\} = -\Im\left\{\frac{H'(\omega)}{H(\omega)}\right\}$$

Note: The last term contains the complex response $H(\omega)$, not the amplitude response $H_A(\omega)$!

In the following, it will be shown that the derivative of birational functions (like DT and CT filters) can be calculated very efficiently and from this the group delay.

J.O. Smith's basic algorithm for FIR filters ('scipy')

An efficient form of calculating the group delay of FIR filters based on the derivative of the logarithmic frequency response has been described in [JOS] and [Lyons08] for discrete time systems.

A FIR filter is defined via its polyome $H(z) = \sum_k b_k z^{-k}$ and has the following derivative:

$$\frac{\partial}{\partial\omega} H(z = e^{j\omega T}) = \frac{\partial}{\partial\omega} \sum_{k=0}^{N} b_k e^{-jk\omega T} = -jT \sum_{k=0}^{N} k b_k e^{-jk\omega T} = -jT H_R(e^{j\omega T})$$

where H_R is the "ramped" polynome, i.e. polynome H multiplied with a ramp k, yielding

$$\tau_g(e^{j\omega T}) = -\Im\left\{\frac{H'(e^{j\omega T})}{H(e^{j\omega T})}\right\} = -\Im\left\{-jT\frac{H_R(e^{j\omega T})}{H(e^{j\omega T})}\right\} = T\,\Re\left\{\frac{H_R(e^{j\omega T})}{H(e^{j\omega T})}\right\}$$

scipy's grpdelay directly calculates the complex frequency response $H(e^{j\omega T})$ and its ramped function at the frequency points using the polyval function.

When zeros of the frequency response are on or near the data points of the DFT, this algorithm runs into numerical problems. Hence, it is neccessary to check whether the magnitude of the denominator is less than e.g. 100 times the machine eps. In this case, τ_g is set to zero.

J.O. Smith's basic algorithm for IIR filters ('scipy')

IIR filters are defined by

$$H(z) = \frac{B(z)}{A(z)} = \frac{\sum b_k z^k}{\sum a_k z^k},$$

their group delay can be calculated numerically via the logarithmic frequency response as well.

The derivative of $H(z)$ w.r.t. ω is calculated using the quotient rule and by replacing the derivatives of numerator and denominator polynomes with their ramp functions:

$$\frac{H'(e^{j\omega T})}{H(e^{j\omega T})} = \frac{\left(B(e^{j\omega T})/A(e^{j\omega T})\right)'}{B(e^{j\omega T})/A(e^{j\omega T})} = \frac{B'(e^{j\omega T})A(e^{j\omega T}) - A'(e^{j\omega T})B(e^{j\omega T})}{A(e^{j\omega T})B(e^{j\omega T})} \tag{4.2}$$

$$= \frac{B'(e^{j\omega T})}{B(e^{j\omega T})} - \frac{A'(e^{j\omega T})}{A(e^{j\omega T})} = -jT\left(\frac{B_R(e^{j\omega T})}{B(e^{j\omega T})} - \frac{A_R(e^{j\omega T})}{A(e^{j\omega T})}\right) \tag{4.3}$$

This result is substituted once more into the log. derivative from above:

$$\tau_g(e^{j\omega T}) = -\Im\left\{\frac{H'(e^{j\omega T})}{H(e^{j\omega T})}\right\} = -\Im\left\{-jT\left(\frac{B_R(e^{j\omega T})}{B(e^{j\omega T})} - \frac{A_R(e^{j\omega T})}{A(e^{j\omega T})}\right)\right\} \tag{4.4}$$

$$= T\Re\left\{\frac{B_R(e^{j\omega T})}{B(e^{j\omega T})} - \frac{A_R(e^{j\omega T})}{A(e^{j\omega T})}\right\} \tag{4.5}$$

If the denominator of the computation becomes too small, the group delay is set to zero. (The group delay approaches infinity when there are poles or zeros very close to the unit circle in the z plane.)

J.O. Smith's algorithm for CT filters

The same process can be applied for CT systems as well: The derivative of a CT polynome $P(s)$ w.r.t. ω is calculated by:

$$\frac{\partial}{\partial\omega} P(s = j\omega) = \frac{\partial}{\partial\omega} \sum_{k=0}^{N} c_k (j\omega)^k = j \sum_{k=0}^{N-1} (k+1) c_{k+1} (j\omega)^k = j P_R(s = j\omega)$$

where P_R is the "ramped" polynome, i.e. its k th coefficient is multiplied by the ramp $k + 1$, yielding the same form as for DT systems (but the ramped polynome has to be calculated differently).

$$\tau_g(\omega) = -\Im\left\{\frac{H'(\omega)}{H(\omega)}\right\} = -\Im\left\{j\frac{H_R(\omega)}{H(\omega)}\right\} = -\Re\left\{\frac{H_R(\omega)}{H(\omega)}\right\}$$

J.O. Smith's improved algorithm for IIR filters ('jos')

J.O. Smith gives the following speed and accuracy optimizations for the basic algorithm:

- convert the filter to a FIR filter with identical phase and group delay (but with different magnitude response)
- use FFT instead of polyval to calculate the frequency response

The group delay of an IIR filter $H(z) = B(z)/A(z)$ can also be calculated from an equivalent FIR filter $C(z)$ with the same phase response (and hence group delay) as the original filter. This filter is obtained by the following steps:

- The zeros of $A(z)$ are the poles of $1/A(z)$, its phase response is $\angle A(z) = -\angle 1/A(z)$.
- Transforming $z \rightarrow 1/z$ mirrors the zeros at the unit circle, correcting the negative phase response. This can be performed numerically by "flipping" the order of the coefficients and multiplying by z^{-N} where N is the order of $A(z)$. This operation also conjugates the coefficients (?) which mirrors the zeros at the real axis. This effect has to be compensated, yielding the polynome $\tilde{A}(z)$. It is the "flip-conjugate" or "Hermitian conjugate" of $A(z)$.

  Frequently (e.g. in the scipy and until recently in the Matlab implementation) the conjugate operation is omitted which gives wrong results for complex coefficients.
- Finally, $C(z) = B(z)\tilde{A}(z)$:

$$C(z) = B(z)\left[z^{-N}A^*(1/z)\right] = B(z)\tilde{A}(z)$$

where

$$\tilde{A}(z) = z^{-N}A^*(1/z) = a_N^* + a_{N-1}^* z^{-1} + \ldots + a_1^* z^{-(N-1)} + z^{-N} \tag{4.6}$$

$$\Rightarrow \tilde{A}(e^{j\omega T}) = e^{-jN\omega T}A^*(e^{-j\omega T}) \tag{4.7}$$

$$\Rightarrow \angle\tilde{A}(e^{j\omega T}) = -\angle A(e^{j\omega T}) - N\omega T \tag{4.8}$$

In Python, the coefficients of $C(z)$ are calculated efficiently by convolving the coefficients of $B(z)$ and $\tilde{A}(z)$:

```
c = np.convolve(b, np.conj(a[::-1]))
```

where b and a are the coefficient vectors of the original numerator and denominator polynomes. The actual group delay is then calculated from the equivalent FIR filter as described above.

Calculating the frequency response with the *np.polyval(p,z)* function at the *NFFT* frequency points along the unit circle, $z = \exp(-j\omega)$, seems to be numerically less robust than using the FFT for the same task, it is also much slower.

This measure fixes already most of the problems described for narrowband IIR filters in scipy issues [Scipy_9310] and [Scipy_1175]. In my experience, these problems occur for all narrowband IIR response types.

Shpak algorithm for IIR filters

The algorithm described above is numerically efficient but not robust for narrowband IIR filters. Especially for filters defined by second-order sections, it is recommended to calculate the group delay using the D. J. Shpak's algorithm.

Code is available at [Endolith_5828333] (GPL licensed) or at [SPA] (MIT licensed).

This algorithm sums the group delays of the individual sections which is much more robust as only second-order functions are involved. However, converting *(b,a)* coefficients to SOS coefficients introduces inaccuracies.

Examples

```
>>> b = [1,2,3] # Coefficients of H(z) = 1 + 2 z^2 + 3 z^3
>>> tau_g, td = pyfda_lib.grpdelay(b)
```

pyfda.libs.pyfda_sig_lib.**group_delayz**(*b*, *a*, *w*, *plot=None*, *fs=6.283185307179586*)

Compute the group delay of digital filter.

Parameters

- **b** (*array_like*) – Numerator of a linear filter.
- **a** (*array_like*) – Denominator of a linear filter.
- **w** (*array_like*) – Frequencies in the same units as *fs*.
- **plot** (*callable*) – A callable that takes two arguments. If given, the return parameters *w* and *gd* are passed to plot.
- **fs** (*float, optional*) – The angular sampling frequency of the digital system.

Returns

- **w** (*ndarray*) – The frequencies at which *gd* was computed, in the same units as *fs*.
- **gd** (*ndarray*) – The group delay in seconds.

pyfda.libs.pyfda_sig_lib.**impz**(*b*, *a=1*, *FS=1*, *N=0*, *step=False*)

Calculate impulse response of a discrete time filter, specified by numerator coefficients b and denominator coefficients a of the system function H(z).

When only b is given, the impulse response of the transversal (FIR) filter specified by b is calculated.

Parameters

- **b** (*array_like*) – Numerator coefficients (transversal part of filter)
- **a** (*array_like (optional, default = 1 for FIR-filter)*) – Denominator coefficients (recursive part of filter)
- **FS** (*float (optional, default: FS = 1)*) – Sampling frequency.
- **N** (*float (optional)*) – Number of calculated points. Default: N = len(b) for FIR filters, N = 100 for IIR filters
- **step** (*bool (optional, default False)*) – return the step response instead of the impulse response

Returns

- **hn** (*ndarray*) – impulse or step response with length N (see above)
- **td** (*ndarray*) – contains the time steps with same length as hn

Examples

```
>>> b = [1,2,3] # Coefficients of H(z) = 1 + 2 z^2 + 3 z^3
>>> h, n = dsp_lib.impz(b)
```

pyfda.libs.pyfda_sig_lib.**impz_len**(*system*, *zpk: bool = False*, *level: float = -40*) → int

Calculate length of impulse response for FIR and IIR filters.

Parameters

- **system** (*array-like*) – The discrete-time system, either specified by its coefficients or its poles and zeros.
- **zpk** (*bool*) – When False (default), the system is specified by its coefficients.

- **level** (*float*) –

  The relative level in dB to which the impulse response has decayed (relevant IIR filters only).

Returns

N – The length of the impulse response.

Return type

int

Notes

For FIR filters, the lenngth of the impulse response is equal to the number of filter taps (= filter order + 1).

For IIR filters, it is only possible to approximate the number of steps until the relative level of the impulse response is below the specified *level*.

The most simple approximation for IIR filters is to only regard the pole p_{max} nearest to the unit circle. For many real-world filters and systems, this pole dominates the transient response ("dominant pole approximation") [JOS_time_constant] with a time constant

$$\tau \approx \frac{T_S}{1 - p_{max}}$$

When calculating the number of samples instead of an absolute time, *T_S = 1*.

The number of samples required to reach *level* in dBs is calculated with

$$N \approx -level/20 * \ln(10)\tau$$

When poles are specified (*zpk == True*), it is of course easy to find the dominant pole. When coefficients of the system are specified, poles can be calculated as the roots of the system

Alternatively, partial fraction expansion (PFE) yields poles and residues. These can be used for a more general solution, see [dsp_stackexchange_2021], [dsp_stackexchange_2022].

pyfda.libs.pyfda_sig_lib.**quadfilt_group_delayz**(*b*, *w*, *fs=6.283185307179586*)

Compute group delay of 2nd-order digital filter.

Parameters

- **b** (*array_like*) – Coefficients of a 2nd-order digital filter.
- **w** (*array_like*) – Frequencies in the same units as *fs*.
- **fs** (*float, optional*) – The sampling frequency of the digital system.

Returns

- **w** (*ndarray*) – The frequencies at which *gd* was computed.
- **gd** (*ndarray*) – The group delay in seconds.

pyfda.libs.pyfda_sig_lib.**sos_group_delayz**(*sos*, *w*, *plot=None*, *fs=6.283185307179586*)

Compute group delay of digital filter in SOS format.

Parameters

- **sos** (*array_like*) – Array of second-order filter coefficients, must have shape `(n_sections, 6)`. Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.
- **w** (*array_like*) – Frequencies in the same units as *fs*.
- **plot** (*callable, optional*) – A callable that takes two arguments. If given, the return parameters *w* and *gd* are passed to plot.

- **fs** (*float, optional*) – The sampling frequency of the digital system.

Returns

- **w** (*ndarray*) – The frequencies at which *gd* was computed.
- **gd** (*ndarray*) – The group delay in seconds.

pyfda.libs.pyfda_sig_lib.**validate_sos**(*sos*)

Helper to validate a SOS input

Copied from *scipy.signal._filter_design._validate_sos()*

pyfda.libs.pyfda_sig_lib.**zeros_with_val**(*N: int*, *val: float = 1.0*, *pos: int = 0*)

Create a 1D array of *N* zeros where the element at position *pos* has the value *val*.

Parameters

- **N** (*int*) – number of elements
- **val** (*scalar*) – value to be inserted at position *pos* (default: 1)
- **pos** (*int*) – Position of *val* to be inserted (default: 0)

Returns
arr – Array with zeros except for element at position *pos*

Return type
np.ndarray

pyfda.libs.pyfda_sig_lib.**zorp_group_delayz**(*zorp*, *w*, *fs=1*)

Compute group delay of digital filter with a single zero/pole.

Parameters

- **zorp** (*complex*) – Zero or pole of a 1st-order linear filter
- **w** (*array_like*) – Frequencies in the same units as *fs*.
- **fs** (*float, optional*) – The sampling frequency of the digital system.

Returns

- **w** (*ndarray*) – The frequencies at which *gd* was computed.
- **gd** (*ndarray*) – The group delay in seconds.

pyfda.libs.pyfda_sig_lib.**zpk2array**(*zpk*)

Test whether Z = zpk[0] and P = zpk[1] have the same length, if not, equalize the lengths by adding zeros.

Test whether gain = zpk[2] is a scalar or a vector and whether it (or the first element of the vector) is != 0. If the gain is 0, set gain = 1.

Finally, convert the gain into an vector with the same length as P and Z and return the the three vectors as one array.

Parameters
zpk (*list, tuple or ndarray*) – Zeros, poles and gain of the system

Return type
zpk as an array or an error string

pyfda.libs.pyfda_sig_lib.**zpk_group_delay**(*z*, *p*, *k*, *w*, *plot=None*, *fs=6.283185307179586*)

Compute group delay of digital filter in zpk format.

Parameters

- **z** (*array_like*) – Zeroes of a linear filter
- **p** (*array_like*) – Poles of a linear filter
- **k** (*scalar*) – Gain of a linear filter

- **w** (*array_like*) – Frequencies in the same units as *fs*.
- **plot** (*callable, optional*) – A callable that takes two arguments. If given, the return parameters *w* and *gd* are passed to plot.
- **fs** (*float, optional*) – The sampling frequency of the digital system.

Returns

- **w** (*ndarray*) – The frequencies at which *gd* was computed.
- **gd** (*ndarray*) – The group delay in seconds.

pyfda.libs.tree_builder module

Create the tree dictionaries containing information about filters, filter implementations, widgets etc. in hierarchical form

exception pyfda.libs.tree_builder.**ParseError**

Bases: Exception

class pyfda.libs.tree_builder.**Tree_Builder**

Bases: object

Read the config file and construct dictionary trees with

- all filter combinations
- valid combinations of filter widgets and fixpoint implementations

build_class_dict(*section*, *subpackage=''*)

- Try to dynamically import the modules (= files) parsed in *section* reading their module level attribute *classes* listing the classes contained in the module.

  When *classes* is a dictionary, e.g. *{"Cheby":"Chebyshev 1"}* where the key is the class name in the module and the value the corresponding display name (used for the combo box).
- When *classes* is a string or a list, use the string resp. the list items for both class and display name.
- Try to import the filter classes

Parameters

- **section** (*str*) – Name of the section in the configuration file to be parsed by `self.parse_conf_section`.
- **subpackage** (*str*) – Name of the subpackage containing the module to be imported. Module names are prepended successively with *['pyfda.' + subpackage + '.', '', subpackage + '.']*

Returns

- **classes_dict** (*dict*)
- *A dictionary with the classes as keys; values are dicts which define*
- *the options (like display name, module path, fixpoint implementations etc).*
- *Each entry has the form e.g.*
- **{<class name>** (*{'name':<display name>, 'mod':<full module name>}} e.g.*)
- *.. code-block:: python –*

  {'Cheby1':{'name':'Chebyshev 1',
  'mod':'pyfda.filter_design.cheby1', 'fix': 'IIR_cascade', 'opt': ["option1", "option2"]}

build_fil_tree(*fc*, *rt_dict*, *fil_tree=None*)

Read attributes (ft, rt, rt:fo) from filter class fc) Attributes are stored in the design method classes in the format (example from `common.py`)

```
self.ft = 'IIR'
self.rt_dict = {
         'LP': {'man':{'fo':     ('a','N'),
                       'msg':    ('a', r"<br /><b>Note:</b> Read this!"),
                       'fspecs': ('a','F_C'),
                       'tspecs': ('u', {'frq':('u','F_PB','F_SB'),
                                        'amp':('u','A_PB','A_SB')})
                      },
                'min':{'fo':     ('d','N'),
                       'fspecs': ('d','F_C'),
                       'tspecs': ('a', {'frq':('a','F_PB','F_SB'),
                                        'amp':('a','A_PB','A_SB')})
                      }
               },
        'HP': {'man':{'fo':     ('a','N'),
                       'fspecs': ('a','F_C'),
                       'tspecs': ('u', {'frq':('u','F_SB','F_PB'),
                                        'amp':('u','A_SB','A_PB')})
                      },
                'min':{'fo':     ('d','N'),
                       'fspecs': ('d','F_C'),
                       'tspecs': ('a', {'frq':('a','F_SB','F_PB'),
                                        'amp':('a','A_SB','A_PB')})
                      }
               }
        }
```

Build a dictionary of all filter combinations with the following hierarchy:

response types -> filter types -> filter classes -> filter order rt (e.g. 'LP') ft (e.g. 'IIR') fc (e.g. 'cheby1') fo ('min' or 'man')

All attributes found for fc are arranged in a dict, e.g. for `cheby1.LPman` and `cheby1.LPmin`, listing the parameters to be displayed and whether they are active, unused, disabled or invisible for each subwidget:

```
'LP':{
'IIR':{
     'Cheby1':{
         'man':{'fo':     ('a','N'),
                'msg':    ('a', r"<br /><b>Note:</b> Read this!"),
                'fspecs': ('a','F_C'),
                'tspecs': ('u', {'frq':('u','F_PB','F_SB'),
                                 'amp':('u','A_PB','A_SB')})
                },
         'min':{'fo':     ('d','N'),
                'fspecs': ('d','F_C'),
                'tspecs': ('a', {'frq':('a','F_PB','F_SB'),
                                 'amp':('a','A_PB','A_SB')})
                }
               }
        }
 }, ...
```

Finally, the whole structure is frozen recursively to avoid inadvertedly changing the filter tree.

For a full example, see the default filter tree `fb.fil_tree` defined in `filterbroker.py`.

Parameters
`None`

Returns
filter tree

Return type
dict

`init_filters()`

Run at startup to populate global dictionaries and lists:

- Read attributes (*ft*, *rt*, *fo*) from all valid filter classes (*fc*) in the global dict `fb.filter_classes` and store them in the filter tree dict `fil_tree` with the hierarchy

  rt-ft-fc-fo-subwidget:params .

Parameters
`None`

Returns

- *fb.fil_tree* :

Return type
None, but populates the following global attributes

`parse_conf_file()`

Parse the configuration file *pyfda.conf* (specified in `dirs.USER_CONF_DIR_FILE`). This is run only once at instantiation.

This is performed using *build_class_dict()* which calls *parse_conf_section()*:

- Try to find and import the modules specified in the corresponding sections
- Extract and import the classes defined in each module and give back an OrderedDict with the successfully imported classes and their options (like fully qualified module names, display name, associated fixpoint widgets etc.).
- Information for each section is stored in globally accessible OrderdDicts like`fb.filter_classes`.

The following sections are analyzed:

[Commons]
Try to find user directories; if they exist add them to *dirs.USER_DIRS* and *sys.path*

For the other sections, OrderedDicts are returned with the class names as keys and dictionaries with options as values.

[Input Widgets]
Store (user) input widgets in *fb.input_classes*

[Plot Widgets]
Store (user) plot widgets in *fb.plot_classes*

[Filter Widgets]
Store (user) filter widgets in *fb.filter_classes*

[Fixpoint Widgets]
Store (user) fixpoint widgets in *fb.fixpoint_classes*

Parameters
`None`

Return type
None, but *self.conf* contains the parsed configuration file.

parse_conf_section(*section*)

Parse `section` in config file *conf* and return an OrderedDict with the elements `{key:<OPTION>}` where *key* and <OPTION> have been read from the config file. <OPTION> has been sanitized and converted to a list or a dict.

Parameters
section (*str*) – name of the section to be parsed

Returns
section_conf_dict – Ordered dict with the keys of the config files and corresponding values

Return type
dict

pyfda.libs.tree_builder.**merge_dicts_hierarchically**(*d1*, *d2*, *path=None*, *mode='keep1'*)

Merge the hierarchical dictionaries `d1` and `d2`. The dict `d1` is modified in place and returned

Parameters

- **d1** (*dict*) – hierarchical dictionary 1
- **d2** (*dict*) – hierarchical dictionary 2
- **mode** (*str*) – Select the behaviour when the same key is present in both dictionaries:
  - **'keep1'**
    keep the entry from `d1` (default)
  - **'keep2'**
    keep the entry from `d2`
  - **'add1'**
    merge the entries, putting the values from `d2` first (important for lists)
  - **'add2'**
    merge the entries, putting the values from `d1` first (")
- **path** (*str*) – internal parameter for keeping track of hierarchy during recursive calls, it should not be set by the user

Returns
d1 – a reference to the first dictionary, merged-in-place.

Return type
dict

Example

```
>>> merge_dicts_hierarchically(fil_tree, fil_tree_add, mode='add1')
```

Notes

If you don't want to modify `d1` in place, call the function using:

```
>>> new_dict = merge_dicts_hierarchically(dict(d1), d2)
```

If you need to merge more than two dicts use:

```
>>> from functools import reduce   # only for py3
>>> reduce(merge, [d1, d2, d3...]) # add / merge all other dicts into d1
```

Taken with some modifications from:

http://stackoverflow.com/questions/7204805/dictionaries-of-dictionaries-merge

Module contents

pyfda.plot_widgets package

Subpackages

pyfda.plot_widgets.tran package

Submodules

pyfda.plot_widgets.tran.plot_tran_stim module

Widget for plotting impulse and general transient responses

class pyfda.plot_widgets.tran.plot_tran_stim.**Plot_Tran_Stim**

Bases: QWidget

Construct a widget for defining transient signals

calc_stimulus_frame(*x: ndarray = array([1.19508723, -0.39702915, -0.25077886, 0.25518419, 0.87133791, -1.08344499, 0.31419319, -0.82324527, 1.32089771, -1.39399192])*, *N_first: int = 0*, *N_frame: int = 10*, *N_end: int = 10*) → ndarray

Calculate a data frame of stimulus *x* with a length of *N_frame* samples, starting with index *N_first*

Parameters

- **x** (*ndarray of float or complex*) – empty array that is filled in place frame by frame
- **N_first** (*int*) – index of first data point of the current frame
- **N_frame** (*int*) – current frame length; the last frame can be shorter than the rest
- **N_end** (*int*) – last sample of total stimulus to be generated (needed for scaling for some stimuli)

Returns

x is filled with data in place

Return type

None

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

init_labels_stim()

intialize title string, y-axis label and some variables

process_sig_rx(*dict_sig=None*) → None

Process signals coming from - the navigation toolbars (time and freq.) - local widgets (impz_ui) and - plot_tab_widgets() (global signals)

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.plot_widgets.tran.plot_tran_stim_ui module

Create the UI for the Plot_Tran_Impz class

class pyfda.plot_widgets.tran.plot_tran_stim_ui.**Plot_Tran_Stim_UI**(*objectName='plot_tran_stim_ui_inst'*)

Bases: QWidget

Create the UI for the PlotImpz class

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

eventFilter(*source*, *event*)

Filter all events generated by the monitored frequency / time related widgets led_f1 and 2, T1 / T2 and TW1 / TW2. Source and type of all events generated by monitored objects are passed

to this eventFilter, evaluated and passed on to the next hierarchy level.

- When a QLineEdit widget gains input focus (QEvent.FocusIn), display the stored value from filter dict with full precision
- When a key is pressed inside the text field, set the *spec_edited* flag to True.

- When a QLineEdit widget loses input focus (`QEvent.FocusOut`) or when the Return key is pressed, store current value normalized to f_S with full precision and display the de-normalized value in selected format or full precision when *spec_edited* == *True*. Emit 'ui_local_changed':'stim'.

normalize_freqs()

Update normalized frequencies and periods if required.

normalize_freqs() is called when sampling frequency has been changed via signal ['view_changed':'f_S'] from plot_impz.process_sig_rx

Frequency and time related entries are always stored normalized w.r.t. f_S which is loaded from filter dictionary and stored as *self.f_scale* (except when the frequency unit is k when *f_scale* = *self.N_FFT*).

- When the *f_S* lock button is unchecked, only the displayed values for frequency entries are updated with f_S, not the normalized freqs.
- When the *f_S* lock button is checked, the absolute frequency values in the widget fields are kept constant, and the normalized freqs are updated.

process_sig_rx(*dict_sig=None*)

Process signals coming from - FFT window widget - qfft_win_select

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx_fft

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_freq_units()

Update labels for time / frequency related specs

pyfda.plot_widgets.tran.plot_tran_stim_ui.**main**()

pyfda.plot_widgets.tran.tran_io module

Widget for loading and storing stimulus data from / to transient plotting widget

class pyfda.plot_widgets.tran.tran_io.**Tran_IO**(*parent*)

Bases: QWidget

Construct a widget for reading data from file

close_csv_win()

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

load_button_clicked()

When load button was clicked, determine its state. When it was "ok" (loaded), unload data and reset button.

Continue with select_chan_normalize() otherwise.

load_data_raw()

Select a file in a UI dialog (CSV or WAV) and load it into *self.data_raw* Try to find the dimensions and some other infos.

When loading the file was successful, store the fully qualified file name and the file type in the attributes *self.file_name* and *self.file_type* and return 0. When an error occurred, return -1.

open_csv_win()

Pop-up window for CSV options, triggered by clicking the options button

process_sig_rx(*dict_sig=None*) → None

Process signals coming from - the navigation toolbars (time and freq.) - local widgets (impz_ui) and - plot_tab_widgets() (global signals)

save_data() → None

Save a file with UI dialog (CSV or WAV), using the data for left and right channel, selected in the UI.

WAV files can be exported in various int formats, the sampling frequency is read from the line edit field.

TODO: uint8 export doesn't work

save_nr_loops()

select_chan_normalize()

select_chan_normalize() is triggered by *load_data_np()* and by signal-slot connections

- self.ui.cmb_chan_import.currentIndexChanged
- self.ui.but_normalize.clicked
- self.ui.led_normalize.editingFinished

It processes *self.data_raw* and yields *self.x_file* as a result which is assigned as *self.stim_wdg.x_file* = *self.file_io_wdg.x_file* in the class *Plot_Impz()* when the signal *{'data_changed': 'file_io'}* is received.

- For two channel *self.data_raw*, assign one channel or the sum of both channels to *data*. Alternatively, assign one channel of *self.data_raw* as real and the other as imaginary component of *data*.
- **Scale *data* to the maximum specified by *self.ui.led_normalize* and**
  assign normalized result to *self.x_file*.

set_f_s_wav(*f_s_wav=None*)

Set sampling frequency for wav files, either from LineEdit (button *Auto f_s* unchecked) or from argument *f_s_wav* (button *Auto f_s* checked), passed either from loaded wav file or from updated f_S some other place in the app.

The sampling frequency needs to integer and at least 1.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

unload_data()

Enable load button and set to normal mode, replace label "Loaded" by "Load", clear loaded data, disable normalize button and emit 'data_changed' signal

pyfda.plot_widgets.tran.tran_io_ui module

Create the UI for the Tran_IO class

class pyfda.plot_widgets.tran.tran_io_ui.**Tran_IO_UI**(*parent=None*)

Bases: QWidget

Create the UI for the Tran_IO class

set_ui_visibility()

Update visiblity and accessibility of some widgets, depending on the settings of other widgets.

update_ui(*cmplx=False*, *fx=False*)

Update the combo boxes for file saving, depending on whether signals are complex and fixpoint simulation has been selected.

Module contents

Submodules

pyfda.plot_widgets.mpl_widget module

Construct a widget consisting of a matplotlib canvas and an improved Navigation toolbar.

class pyfda.plot_widgets.mpl_widget.**MplToolbar**(*canv*, *mpl_widget*, **args*, ***kwargs*)

Bases: NavigationToolbar2QT

Custom Matplotlib Navigationtoolbar, derived (subclassed) from Qt's NavigationToolbar with the following changes: - new icon set - new functions and icons for grid toggle, full view, screenshot - removed buttons for configuring subplots and editing curves - added an x,y location widget and icon

Signalling / communication works via the signal `sig_tx'

derived from http://www.python-forum.de/viewtopic.php?f=24&t=26437

http://pydoc.net/Python/pyQPCR/0.7/pyQPCR.widgets.matplotlibWidget/ !! http://matplotlib.org/users/navigation_toolbar.html !!

see also http://stackoverflow.com/questions/17711099/programmatically-change-matplotlib-toolbar-mode-in-qt4
http://matplotlib-users.narkive.com/C8XwIXah/need-help-with-darren-dale-qt-example-of-extending-toolbar
https://sukhbinder.wordpress.com/2013/12/16/simple-pyqt-and-matplotlib-example-with-zoompan/

Changing the info: http://stackoverflow.com/questions/15876011/add-information-to-matplotlib-navigation-toolbar-status-ba
https://stackoverflow.com/questions/53099295/matplotlib-navigationtoolbar-advanced-figure-options

Using Tool Manager https://matplotlib.org/3.1.1/gallery/user_interfaces/toolmanager_sgskip.html https://stackoverflow.com/questions/52971285/add-toolbar-button-icon-matplotlib

Documentation on QKeySequences: https://doc.qt.io/qt-6/qkeysequence.html

Construct a shortcut string: logger.info(QtGui.QKeySequence.toString(QtGui.QKeySequence(Qt.SHIFT|Qt.CTRL|Qt.Key_Z)

cycle_draw_grid(*cycle=True*, *axes=None*)

Cycle the grid of all axes through the states 'off', 'coarse' and 'fine' and redraw the figure.

Parameters

- **cycle** (*bool, optional*) – Cycle the grid display and redraw the canvas in the end when True. When false, only restore the grid settings.
- **axes** (*matplotlib axes, optional*) – When none is passed, use local *self.mpl_widget.fig.axes*

Return type
None.

cycle_ui_level(*ui_level: int = -1*) → None

Cycle the UI level: UI elements fully / partially / invisible)

Parameters
ui_level (*int, optional*) – Set the ui level and the icon accordingly when ui_level != -1, (was not passed as a parameter), cycle through the *self.ai_num_levels* and emit *{'mpl_toolbar': 'ui_level'}*

Return type
None

edit_parameters()

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

enable_plot(*state=None*)

Toggle the plot enable button, enable / disable other plot buttons accordingly and emit

help()

Open help page from https://pyfda.rtfd.org in browser

home()

Reset zoom to default settings (defined by plotting widget). This method shadows *home()* inherited from NavigationToolbar.

mpl2Clip(*key_event=False*)

Copy current figure to the clipboard, either directly as PNG file or as base64 encoded PNG file, with or without title.

Qt.ShiftModifier = 0x02000000 # Shift key pressed Qt.ControlModifier = 0x04000000 # Control key Qt.AltModifier = 0x08000000 # Alt key, doesn't work under Linux? Qt.MetaModifier = 0x10000000 # Meta key

When *key_event == True*, the trigger was a CTRL+C keypress and the Control modifier has to be blanked out.

ALT-key doesn't work as a mouse modifier because it shifts the focus from the
toolbar to the menubar (? not implemented here)

save_button_states()

set_message(*msg*)

Display a message on toolbar or in status bar.

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

`toggle_lock_zoom()`

Toggle the lock zoom settings and save the plot limits in any case:
when previously unlocked, settings need to be saved when previously locked, current settings can be saved without effect

`toolitems = ()`

`class pyfda.plot_widgets.mpl_widget.MplWidget(parent)`

Bases: `QWidget`

Construct a subwidget consisting of a Matplotlib canvas and a subclassed NavigationToolbar.

`eventFilter(source, event)`

Filter all events generated by the monitored widgets (self). Source and type of all events generated by monitored objects are passed

to this eventFilter, evaluated and passed on to the next hierarchy level.

`get_full_extent(ax, pad=0.0)`

Get the full extent of axes system *ax*, including axes labels, tick labels and titles.

Needed for inset plot in H(f)

`plt_full_view()`

Zoom to full extent of data if axes is set to "navigationable" by the navigation toolbar

`redraw()`

Redraw the figure with new properties (grid, linewidth) and restore the plot limits when *a_zo_locked* is True

When zoom lock is used and / or plot limits shall be pushed into the queue, you need to call redraw()

`save_limits()`

Save x- and y-limits of all axes in self.limits when zoom is unlocked

`toggle_cursor()`

Toggle the tracking cursor

`pyfda.plot_widgets.mpl_widget.no_plot(x, y, ax=None, bottom=0, label=None, **kwargs)`

Don't plot anything - needed for plot factory

`pyfda.plot_widgets.mpl_widget.scatter(x, y, ax=None, label=None, mkr_fmt=None, **kwargs)`

Create a copy of matplolibs scatter that can handle 'ms' and 'markersize' keys These keys are removed from the mkr_fmt dictionary and translated to s = ms * ms When neither ms nor markersize are provided, a default of ms = 10 is used.

Return a handle to the scatter plot.

`pyfda.plot_widgets.mpl_widget.stems(x, y, ax=None, label=None, mkr_fmt=None, **kwargs)`

Provide a faster replacement for stem plots under matplotlib < 3.1.0 using vlines (= LineCollection). LineCollection keywords are supported.

pyfda.plot_widgets.plot_3d module

Widget for plotting |H(z)| in 3D

class pyfda.plot_widgets.plot_3d.**Plot_3D**

Bases: QWidget

Class for various 3D-plots: - lin / log line plot of H(f) - lin / log surf plot of H(z) - optional display of poles / zeros

draw()

Main drawing entry point: perform the actual plot

draw_3d()

Draw various 3D plots

init_axes()

Initialize and clear the axes to get rid of colorbar The azimuth / elevation / distance settings of the camera are restored after clearing the axes. See http://stackoverflow.com/questions/4575588/matplotlib-3d-plot-with-pyqt4-in-qtabwidget-mplwidget

process_sig_rx(*dict_sig=None*)

Process signals coming from the navigation toolbar and from sig_rx

redraw()

Redraw the canvas when e.g. the canvas size has changed

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.plot_widgets.plot_3d.**classes = {'Plot_3D': '3D'}**

display name

Type
Dict containing class name

pyfda.plot_widgets.plot_fft_win module

Create a popup window with FFT window information

class pyfda.plot_widgets.plot_fft_win.**Plot_FFT_win**(*win_dict*, *sym=False*, *title='pyFDA Window Viewer'*, *ignore_close_event=True*)

Bases: QDialog

Create a pop-up widget for displaying time and frequency view of an FFT window.

Data is passed via the dictionary *win_dict* that is specified during construction. Available windows, parameters, tooltipps etc are provided by the widget *pyfda_fft_windows_lib.QFFTWinSelection*

Parameters

- **parent** (*class instance*) – reference to parent
- **win_dict** (*dict*) – dictionary derived from *pyfda_fft_windows_lib.all_windows_dict* with valid and available windows and their current settings (if applicable)
- **sym** (*bool*) – Passed to *get_window()*: When True, generate a symmetric window for use in filter design. When False (default), generate a periodic window for use in spectral analysis.
- **title** (*str*) – Title text for Qt Window
- **ignore_close_event** (*bool*) – Disable close event when True (Default)

- `self.calc_N()`

- `self.update_view()`:

- `self.draw()`: calculate window and FFT and draw both

- `get_win(N)` : Get the window array

calc_win_draw()

(Re-)Calculate the window, its FFT and some characteristic values and update the plot of the window and its FFT. This should be triggered when the window type or length or a parameters has been changed.

Return type
None

closeEvent(*event*)

Catch *closeEvent* (user has tried to close the FFT window) and send a signal to parent to decide how to proceed.

This can be disabled by setting *self.ignore_close_event = False* e.g. for instantiating the widget as a standalone window.

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

process_sig_rx(*dict_sig=None*)

Process signals coming from the navigation toolbar and from sig_rx:

- *self.calc_N*
- *self.update_view*:
- *self.draw*: calculate window and FFT and draw both

redraw()

Redraw the canvas when e.g. the canvas size has changed

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_bottom()

Update log bottom settings

update_fft_win(*dict_sig=None*)

Update FFT window when window or parameters have changed and pass thru 'view_changed':'fft_win_type' or 'fft_win_par'

update_info()

Update the text info box for the window

update_view()

Draw the figure with new limits, scale, lin/log etc without recalculating the window or its FFT.

pyfda.plot_widgets.plot_hf module

The `Plot_Hf` class constructs the widget to plot the magnitude frequency response |H(f)| of the filter either in linear or logarithmic scale. Optionally, the magnitude specifications and the phase can be overlayed.

class pyfda.plot_widgets.plot_hf.**Plot_Hf**

Bases: `QWidget`

Widget for plotting |H(f)|, frequency specs and the phase

align_y_axes(*ax1*, *ax2*)

Sets tick marks of twinx axes to line up with total number of ax1 tick marks

calc_hf()

(Re-)Calculate the complex frequency response H_cmplx(W) (complex) for W = 0 ... 2 pi:

draw()

Re-calculate |H(f)| and draw the figure

draw_inset()

Construct / destruct second axes for an inset second plot

draw_phase(*ax*)

Draw phase on second y-axis in the axes system passed as the argument

init_axes()

Initialize and clear the axes (this is run only once)

plot_spec_limits(*ax*)

Plot the specifications limits (F_SB, A_SB, ...) as hatched areas with borders.

process_sig_rx(*dict_sig=None*)

Process signals coming from the navigation toolbar and from sig_rx

redraw()

Redraw the canvas when e.g. the canvas size has changed

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_view()

Draw the figure with new limits, scale etc without recalculating H(f)

pyfda.plot_widgets.plot_hf.**classes = {'Plot_Hf': '|H(f)|'}**

display name

Type
Dict containing class name

pyfda.plot_widgets.plot_impz module

Widget for plotting impulse and general transient responses

class pyfda.plot_widgets.plot_impz.**Plot_Impz**(*objectName='plot_impz_inst'*)

Bases: QWidget

Construct a widget for plotting impulse and general transient responses

calc_auto(*autorun: bool = None*) → None

Triggered when checkbox "Autorun" is clicked or specs have been edited, requiring a recalculation.

When Autorun has been pushed (*but_auto_run.isChecked() == True*) and calculation is required, automatically run *impz_init()*.

calc_fft()

(Re-)calculate FFTs of stimulus *self.X*, quantized stimulus *self.X_q* and response *self.Y* using the window function from *self.ui.win_dict['win']*.

draw(*arg=None*)

(Re-)draw the figure without recalculation. When triggered by a signal- slot connection from a button, combobox etc., arg is a boolean or an integer representing the state of the widget. In this case, *needs_redraw* is set to True.

draw_data(*plt_style: str, ax: object, x: ndarray, y: ndarray, bottom: float = 0, label: str = '', plt_fmt: dict = {}, mkr_fmt: dict = {}, **args*)

Plot x, y data (numpy arrays with equal length) in a plot style defined by *plt_style*.

Parameters

- **plt_style** (*str*) – one of "line", "stem", "steps", "dots"
- **ax** (*matplotlib axis*) – Handle to the axis where signal is to be plotted
- **x** (*array-like*) – x-axis: time or frequency data
- **y** (*array-like*) – y-data
- **bottom** (*float*) – Bottom line y-coordinate for stem plot. The default is 0.
- **label** (*str*) – Plot label
- **plt_fmt** (*dict*) – Line styles (color, linewidth etc.) for plotting (default: None).
- **mkr_fmt** (*dict*) – Marker styles
- **args** (*dict*) – additional keys and values. As they might not be compatible with every plot style, they have to be added individually

Returns
handle – This provides a handle to the properties of line and marker (optionally) which are displayed by legend

Return type
A *lines.Line2D()* objects or tuple with two of them

draw_freq()

(Re-)draw the frequency domain mplwidget

draw_time(*N_start=0*, *N_end=0*)

(Re-)draw the time domain mplwidget

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

file_io() → None

Check status of file_io widget:

- if no file is loaded, do nothing and return 0, disable *cmb_file_io* and the option to transfer the number of samples to N
- **else map the file data to *self.stim_wdg.x_file* to make it accessible**
  from the stimulus widget. If *cmb_file_io == `use*, disable the widget to modify stimuli

impz()

Calculate floating point / fixpoint response and redraw it

Triggered by:

- *self.impz_init()* (floating point)
- **Fixpoint widget, requesting "start_fx_response_calculation"**
  via *process_rx_signal()* (fixpoint filter)

impz_finish()

Do some housekeeping, resetting and drawing when *self.impz()* has finished:

- Calculate step error if selected
- Check for complex stimulus or response
- Calculate simulation time

- Draw the signals
- Reset Run Icon to normal state, reset *needs_calc* flag
- Update File IO save combo boxes

impz_init(*arg=None*) → None

Initialize transient simulation.

Parameters
arg (*bool or None*)

Return type
None

Triggered by:

- *_construct_UI()* during initialization
- Pressing "Run" button, passing button state as a bool
- *self.ui.cmb_sim_select* when changing between fixpoint and float mode
- *self.calc_auto()* when activating "Autorun"
- Autorun (when something relevant in the UI has been updated)
- signal `{'fx_sim' : 'specs_changed'}`

The following tasks are performed:

- Enable energy scaling for impulse stimuli when requirements are met
- check for and enable fixpoint settings
- resize stimulus widget
- when triggered by *but_run* or when *Auto`== pressed and `self.needs_calc == True*, continue with calculating stimulus / response
- When in fixpoint mode, initialize quantized stimulus *x_q* and input quantizer and emit *{'fx_sim':'init'}*

process_sig_rx(*dict_sig=None*)

Process signals coming from - the navigation toolbars (time and freq.) - local widgets (impz_ui) and - plot_tab_widgets() (global signals)

process_sig_rx_f(*dict_sig=None*)

Special treatment for signals coming from FREQ plot navigation toolbar

process_sig_rx_t(*dict_sig=None*)

Special treatment for signals coming from TIME plot navigation toolbar

redraw()

Redraw the currently visible canvas (but not the plot!) when e.g. the canvas size has changed

resize_stim_tab_widget()

Resize active tab of stimulus Tab widget to fit the height of the contained widget. This is triggered by: - initialization in *_construct_UI()* - changed tab in the stimulus tab widget (signal-slot) - an 'ui-changed' - signal (*process_signal_rx()*)

set_N_to_file_len() → None

Check status of file_io widget: - if no file is loaded, do nothing. This shouldn't happen (check to be sure ...) - else set N_end = len(file_data) in the UI

set_ui_level(*ui_level*)

Sync time and frequency subwidget and set their ui display level

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

toggle_fx_settings(*arg=None*)

arg can be the following arguments, triggered by:

- arg *None*: from *__init__()*, *impz_init()* or *process_sig_rx()* when {dict_sig['fx_sim'] == 'specs_changed'} was received. Read the state of *fb.fil[0]['fx_sim']* and update combobox correspondingly
- arg int 0 or 1 from *self.ui.cmb_sim_select* when index was changed (signal-slot-connection), update *fb.fil[0]['fx_sim']* correspondingly, fire signal {'fx_sim': 'specs_changed'} and start simulation
- arg "fixpoint" or "float" from a direct call with (not used currently), update ui and combobox *self.ui.cmb_sim_select* correspondingly

When fixpoint simulation is selected, all corresponding widgets are made visible and *fb.fil[0]['fx_sim']* is set to True.

If *fb.fil[0]['fx_sim']* has been changed since last time, *self.needs_calc* is set to True and the run button is set to 'changed'.

toggle_stim_options()

Toggle visibility of stimulus options, depending on the state of the "Stimuli" button

zoom_home()

Zoom to home settings

pyfda.plot_widgets.plot_impz.**classes = {'Plot_Impz': 'y[n] / Y(f)'}**

display name

Type
Dict containing class name

pyfda.plot_widgets.plot_impz_ui module

Create the UI for the PlotImz class

class pyfda.plot_widgets.plot_impz_ui.**PlotImpz_UI**

Bases: QWidget

Create the UI for the PlotImpz class

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

hide_fft_wdg()

The closeEvent caused by clicking the "x" in the FFT widget is caught there and routed here to only hide the window

process_sig_rx(*dict_sig=None*)

Process signals coming from - FFT window widget - qfft_win_select

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx_fft

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

toggle_fft_wdg()

Show / hide FFT widget depending on the state of the corresponding button When widget is shown, trigger an update of the window function.

update_N(*emit=True*, *N_end=0*)

Update values for *self.N* and *self.win_dict['N']*, for *self.N_start* and *self.N_end* from the corresponding QLineEditWidgets.

Parameters

- **emit** (*bool*) – When *emit==True* (default), fire *{'ui_local_changed': 'N'}* to update the FFT window and the *plot_impz* widgets. In contrast to *view_changed*,

  this also forces a calculation of the transient response.

- **N_end** (*int*) – When *N_end* is specified, use the passed value for the total number of data points
- **by** (*This method is called*)
- **emit==False** (*- self._construct_ui() with*)
- **calculation** (*- plot_impz() with emit==False when the automatic*) – of N has to be updated (e.g. order of FIR filter) has changed
- **have** (*- signal-slot connection when N_start or N_end QLineEdit widgets*) – been changed (*emit==True*)

update_N_auto()

pyfda.plot_widgets.plot_impz_ui.**main**()

pyfda.plot_widgets.plot_phi module

Widget for plotting phase frequency response phi(f)

class pyfda.plot_widgets.plot_phi.**Plot_Phi**

Bases: QWidget

calc_resp()

(Re-)Calculate the complex frequency response H(f)

draw()

Main entry point: Re-calculate |H(f)| and draw the figure

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

init_axes()

Initialize and clear the axes - this is only called once

process_sig_rx(*dict_sig=None*)

Process signals coming from the navigation toolbar and from sig_rx

redraw()

Redraw the canvas when e.g. the canvas size has changed

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

unit_changed()

Unit for phase display has been changed, emit a 'view_changed' signal and continue with drawing.

update_view()

Draw the figure with new limits, scale etc without recalculating H(f)

pyfda.plot_widgets.plot_phi.**classes = {'Plot_Phi': '(f)'}**

display name

Type
Dict containing class name

pyfda.plot_widgets.plot_pz module

Widget for plotting poles and zeros

class pyfda.plot_widgets.plot_pz.**Plot_PZ**

Bases: QWidget

draw()

draw_Hf(*r=2*, *Hf_visible=True*)

Draw the magnitude frequency response around the UC

draw_contours(*overlay*)

draw_pz()

(re)draw P/Z plot

init_axes()

Initialize and clear the axes (this is only run once)

process_sig_rx(*dict_sig: dict = None*) → None

Process signals coming from the navigation toolbar and from sig_rx

redraw()

Redraw the canvas when e.g. the canvas size has changed

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

update_view()

Draw the figure with new limits, scale etcs without recalculating H(f) – not yet implemented, just use draw() for the moment

zplane(*b=None*, *a=1*, *z=None*, *p=None*, *k=1*, *pn_eps=0.001*, *analog=False*, *plt_ax=None*, *plt_poles=True*, *style='equal'*, *anaCircleRad=0*, *lw=2*, *mps=10*, *mzs=10*, *mpc='r'*, *mzc='b'*, *plabel=''*, *zlabel=''*)

Plot the poles and zeros in the complex z-plane either from the coefficients (*b, `a) of a discrete transfer function `H`( `z*) (zpk = False) or directly from the zeros and poles (z,p) (zpk = True).

When only b is given, an FIR filter with all poles at the origin is assumed.

Parameters

- **b** (*array_like*) – Numerator coefficients (transversal part of filter) When b is not None, poles and zeros are determined from the coefficients b and a
- **a** (*array_like (optional, default = 1 for FIR-filter)*) – Denominator coefficients (recursive part of filter)
- **z** (*array_like, default = None*) – Zeros When b is None, poles and zeros are taken directly from z and p
- **p** (*array_like, default = None*) – Poles
- **analog** (*boolean (default: False)*) – When True, create a P/Z plot suitable for the s-plane, i.e. suppress the unit circle (unless anaCircleRad > 0) and scale the plot for a good display of all poles and zeros.
- **pn_eps** (*float (default : 1e-2)*) – Tolerance for separating close poles or zeros
- **plt_ax** (*handle to axes for plotting (default: None)*) – When no axes is specified, the current axes is determined via plt.gca()
- **plt_poles** (*Boolean (default : True)*) – Plot poles. This can be used to suppress poles for FIR systems where all poles are at the origin.
- **style** (*string (default: 'scaled')*) – Style of the plot, for *style == 'scaled'* make scale of x- and y- axis equal, *style == 'equal'* forces x- and y-axes to be equal. This is passed as an argument to the matplotlib *ax.axis(style)*
- **mps** (*integer (default: 10)*) – Size for pole marker
- **mzs** (*integer (default: 10)*) – Size for zero marker

- **mpc** (*char (default: 'r')*) – Pole marker colour
- **mzc** (*char (default: 'b')*) – Zero marker colour
- **lw** (*integer (default: 2)*) – Linewidth for unit circle
- **plabel** (*string (default: '')*) – This string is passed to the plot command for poles and zeros and can be displayed by legend()
- **zlabel** (*string (default: '')*) – This string is passed to the plot command for poles and zeros and can be displayed by legend()

Returns
z, p, k

Return type
ndarray

Notes

pyfda.plot_widgets.plot_pz.**classes = {'Plot_PZ': 'P / Z'}**

display name

Type
Dict containing class name

pyfda.plot_widgets.plot_tab_widgets module

Create a tabbed widget for all plot subwidgets in the list `fb.plot_widgets_list`. This list is compiled at startup in `pyfda.tree_builder.Tree_Builder`, it is kept as a module variable in *pyfda.filterbroker*.

class pyfda.plot_widgets.plot_tab_widgets.**PlotTabWidgets**(*parent=None*, *objectName='plot_tab_widgets_inst'*)

Bases: `QTabWidget`

current_tab_changed()

current_tab_redraw()

emit(*dict_sig: dict = {}*, *sig_name: str = 'sig_tx'*) → None

Emit a signal *self.<sig_name>* (defined as a class attribute) with a dict *dict_sig* using Qt's *emit()*.

- Add the keys *'id'* and *'class'* with id resp. class name of the calling instance if not contained in the dict
- If key 'ttl' is in the dict and its value is less than one, terminate the signal. Otherwise, reduce the value by one.
- If the sender has passed an objectName, add it with the key "sender_name" to the dict.

eventFilter(*source*, *event*)

Filter all events generated by the QTabWidget. Source and type of all events generated by monitored objects are passed to this eventFilter, evaluated and passed on to the next hierarchy level.

This filter stops and restarts a one-shot timer for every resize event. When the timer generates a timeout after 500 ms, `current_tab_redraw()` is called by the timer.

log_rx(*dict_sig=None*)

Enable *self.sig_rx.connect(self.log_rx)* above for debugging.

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

sig_tx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

pyfda.plot_widgets.plot_tau_g module

Widget for plotting the group delay

class pyfda.plot_widgets.plot_tau_g.**Plot_tau_g**

Bases: QWidget

Widget for plotting the group delay

calc_tau_g()

(Re-)Calculate the complex frequency response H(f)

draw()

init_axes()

Initialize the axes and set some stuff that is not cleared by *ax.clear()* later on.

process_sig_rx(*dict_sig=None*)

Process signals coming from the navigation toolbar and from sig_rx

redraw()

Redraw the canvas when e.g. the canvas size has changed

sig_rx

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the

signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = ..., revision

`update_view()`

Draw the figure with new limits, scale etc without recalculating H(f)

Module contents

4.1.2 Submodules

4.1.3 pyfda.filter_factory module

Dynamic parameters and settings are exchanged via the dictionaries in this file. Importing `filterbroker.py` runs the module once, defining all module variables which have a global scope like class variables and can be imported like

```
>>> import filter_factory as ff
>>> myfil = ff.fil_factory
```

`class pyfda.filter_factory.FilterFactory`

Bases: `object`

This class implements a filter factory that (re)creates the globally accessible filter instance `fil_inst` from module path and class name, passed as strings.

`call_fil_method(method, fil_dict, fc=None)`

Instantiate the filter design class passed as string `fc` with the globally accessible handle `fil_inst`. If `fc = None`, use the previously instantiated filter design class.

Next, call the design method passed as string `method` of the instantiated filter design class.

Parameters

- **method** (*string*) – The name of the design method to be called (e.g. 'LPmin')
- **fil_dict** (*dictionary*) – A dictionary with all the filter specs that is passed to the actual filter design routine. This is usually a copy of `fb.fil[0]` The results of the filter design routine are written back to the same dict.
- **fc** (*string (optional, default: None)*) – The name of the filter design class to be instantiated. When nothing is specified, the last filter selection is used.

Returns

err_code –

one of the following error codes:

-1
filter design operation has been cancelled by user

0
filter design method exists and is callable

16
passed method name is not a string

17
filter design method does not exist in class

18
filter design error containing "order is too high"

19
filter design error containing "failure to converge"

99
unknown error

Return type
int

Examples

```
>>> call_fil_method("LPmin", fil[0], fc="cheby1")
```

The example first creates an instance of the filter class 'cheby1' and then performs the actual filter design by calling the method 'LPmin', passing the global filter dictionary `fil[0]` as the parameter.

create_fil_inst(*fc*, *mod=None*)

Create an instance of the filter design class passed as a string `fc` from the module found in `fb.filter_classes[fc]`. This dictionary has been collected by `tree_builder.py`.

The instance can afterwards be globally referenced as `fil_inst`.

Parameters

- **fc** (*str*) – The name of the filter design class to be instantiated (e.g. 'cheby1' or 'equiripple')
- **mod** (*str (optional, default = None)*) – Fully qualified name of the filter module. When not specified, it is read from the global dict `fb.filter_classes[fc]['mod']`

Returns

err_code –

one of the following error codes:

-1
filter design class was instantiated successfully

0
filter instance exists, no re-instantiation necessary

1
filter module not found by FilterTreeBuilder

2
filter module found by FilterTreeBuilder but could not be imported

3
filter class could not be instantiated

4
unknown error during instantiation

Return type
int

Examples

```
>>> create_fil_instance('cheby1')
>>> fil_inst.LPmin(fil[0])
```

The example first creates an instance of the filter class 'cheby1' and then performs the actual filter design by calling the method 'LPmin', passing the global filter dictionary fil[0] as the parameter.

pyfda.filter_factory.**fil_factory = <pyfda.filter_factory.FilterFactory object>**

Class instance of FilterFactory that can be accessed in other modules

pyfda.filter_factory.**fil_inst = None**

Instance of current filter design class (e.g. "cheby1"), globally accessible

```
>>> import filter_factory as ff
>>> ff.fil_factory.create_fil_instance('cheby1') # create instance of dynamic␣
↪class
>>> ff.fil_inst.LPmin(fil[0]) # design a filter
```

4.1.4 pyfda.filterbroker module

Dynamic parameters and settings are exchanged via the dictionaries in this file. Importing `filterbroker.py` runs the module once, defining all module variables which have a global scope like class variables and can be imported like

```
>>> import filterbroker as fb
>>> myfil = fb.fil[0]
```

The entries in this file are only used as initial / default entries and to demonstrate the structure of the global dicts and lists. These initial values are also handy for module-level testing where some useful settings of the variables is required.

Notes

Alternative approaches for data persistence could be the packages *shelve* or pickleshare More info on data persistence and storing / accessing global variables:

- http://stackoverflow.com/questions/13034496/using-global-variables-between-files-in-python
- http://stackoverflow.com/questions/1977362/how-to-create-module-wide-variables-in-python
- http://pymotw.com/2/articles/data_persistence.html
- http://stackoverflow.com/questions/9058305/getting-attributes-of-a-class
- http://stackoverflow.com/questions/2447353/getattr-on-a-module

pyfda.filterbroker.**base_dir = ''**

Project base directory

pyfda.filterbroker.**clipboard = None**

Handle to central clipboard instance

pyfda.filterbroker.**filter_classes** = {'Bessel': {'mod': 'pyfda.filter_widgets.bessel', 'name': 'Bessel'}, 'Butter': {'mod': 'pyfda.filter_widgets.butter', 'name': 'Butterworth'}, 'Cheby1': {'mod': 'pyfda.filter_widgets.cheby1', 'name': 'Chebyshev 1'}, 'Cheby2': {'mod': 'pyfda.filter_widgets.cheby2', 'name': 'Chebyshev 2'}, 'Ellip': {'mod': 'pyfda.filter_widgets.ellip', 'name': 'Elliptic'}, 'EllipZeroPhz': {'mod': 'pyfda.filter_widgets.ellip_zero', 'name': 'EllipZeroPhz'}, 'Equiripple': {'mod': 'pyfda.filter_widgets.equiripple', 'name': 'Equiripple'}, 'Firwin': {'mod': 'pyfda.filter_widgets.firwin', 'name': 'Windowed FIR'}, 'MA': {'mod': 'pyfda.filter_widgets.ma', 'name': 'Moving Average'}, 'Manual_FIR': {'mod': 'pyfda.filter_widgets.manual', 'name': 'Manual'}, 'Manual_IIR': {'mod': 'pyfda.filter_widgets.manual', 'name': 'Manual'}}

The keys of this dictionary are the names of all found filter classes, the values are the name to be displayed e.g. in the comboboxes and the fully qualified name of the module containing the class.

pyfda.filterbroker.**redo**()

Store current filter to undo memory *fil_undo*

pyfda.filterbroker.**undo**()

Restore current filter from undo memory *fil_undo*

4.1.5 pyfda.pyfda_class module

Mainwindow for the pyFDA app

class pyfda.pyfda_class.**DynFileHandler**(**args*)

Bases: FileHandler

subclass FileHandler with a customized handler for dynamic definition of the logging filepath and -name

class pyfda.pyfda_class.**QEditHandler**

Bases: Handler

subclass Handler to also log messages to textWidget on main display Overrides stdout to print messages to textWidget (XStream)

emit(*record*)

Do whatever it takes to actually log the specified logging record.

This version is intended to be implemented by subclasses and so raises a NotImplementedError.

class pyfda.pyfda_class.**XStream**

Bases: QObject

subclass for log messages on logger window Overrides stdout to print messages to textWidget

fileno()

flush()

messageWritten

int = ..., arguments: Sequence = ...) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type

pyqtSignal(*types, name

Type
str = . . . , revision

static stdout()

write(*msg*)

class pyfda.pyfda_class.**pyFDA**(*parent=None*)

Bases: QMainWindow

Create the main window consisting of a tabbed widget for entering filter specifications, poles / zeros etc. and another tabbed widget for plotting various filter characteristics

QMainWindow is used here as it is a class that understands GUI elements like toolbar, statusbar, central widget, docking areas etc.

closeEvent(*event*)

reimplement QMainWindow.closeEvent() to prompt the user

logger_win_context_menu(*point*)

Show right mouse button context menu

process_sig_rx(*dict_sig=None*)

Process signals coming from sig_rx: - trigger close event in response to 'quit_program' emitted in another subwidget:

sig_rx

int = . . . , arguments: Sequence = . . .) -> PYQT_SIGNAL

types is normally a sequence of individual types. Each type is either a type object or a string that is the name of a C++ type. Alternatively each type could itself be a sequence of types each describing a different overloaded signal. name is the optional C++ name of the signal. If it is not specified then the name of the class attribute that is bound to the signal is used. revision is the optional revision of the signal that is exported to QML. If it is not specified then 0 is used. arguments is the optional sequence of the names of the signal's arguments.

Type
pyqtSignal(*types, name

Type
str = . . . , revision

4.1.6 pyfda.pyfda_rc module

This file contains layout definitions for Qt and matplotlib widgets A dark and a light theme can be selected via a constant but this more a demonstration on how to set things than a finished layout yet.

Default parameters, paths etc. are also defined at the end of the file.

Importing pyfda_rc runs the module once, defining all module variables which are global (similar to class variables).

4.1.7 pyfda.pyfdax module

Mainwindow for the pyFDA app

pyfda.pyfdax.**main**()

entry point for the pyfda application see http://pyqt.sourceforge.net/Docs/PyQt4/qapplication.html :

"For any GUI application using Qt, there is precisely *one* QApplication object, no matter whether the application has 0, 1, 2 or more windows at any given time. ... Since the QApplication object does so much initialization, it must be created *before* any other objects related to the user interface are created."

Environment variables controlling Qt behaviour need to be set even before initializing the QApplication object

Scaling

- DPI: The resolution number of dots per inch in a digital print
- PPI: Pixel density of an electronic image device (e.g. computer monitor)
- Point: 1/72 Inch = 0.3582 mm, physical measure in typography
- em: Equal to font height. For e.g. a 12 pt font, 1 em = 12 pt
- Physical DPI: The PPI that a physical screen actually provides.
- **Logical DPI: The PPI that software claims a screen provides. This can be thought** of as the PPI provided by a virtual screen created by the operating system. Font sizes are specified in logical DPI
- **Screen scaling: High-Resolution screens have a very high physical DPI, resulting** in very small characters. Screen scaling by e.g. 125 ... 200% increases the logical DPI and hence the character size by the same amount.

MacOS

Early displays had 72 PPI, equaling 72 Pt/Inch, i.e. 1 Pixel = 1 Point. Print-out size was equal to screen size.

Windows

A 72-point font is defined to be one logical inch = 96 pixels tall.

12 pt = 12/72 = 1/6 logical inch = 96/6 pixels = 16 pixels @ 96 dpi

Enable automatic scaling based on the monitor's pixel density. This doesn't change the # size of point based fonts! # os.environ["QT_ENABLE_HIGHDPI_SCALING"] = "1" # os.environ["QT_AUTO_SCREEN_SCALE_FACTOR"] = "1" # replaced by QT_ENABLE_HIGHDPI_SCALING # Define global scale factor for the whole application, including point-sized fonts: # os.environ["QT_SCALE_FACTOR"] = "1"

4.1.8 pyfda.qrc_resources module

pyfda.qrc_resources.**qCleanupResources**()

pyfda.qrc_resources.**qInitResources**()

4.1.9 pyfda.version module

Store the version number here for setup.py and pyfdax.py

4.1.10 Module contents

CHAPTER

FIVE

LITERATURE

References

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[dsp_stackexchange_2021] "Estimating reverberant time (T60) of an IIR filter", 30.8. 2021, https://dsp.stackexchange.com/questions/77005/estimating-reverberant-time-t60-of-an-iir-filter

[dsp_stackexchange_2022] "IIR filter relaxation time computation", 30.6.2022 https://dsp.stackexchange.com/questions/68023/iir-filter-relaxation-time-computation

[Endolith_5828333] Endolith,"designtools.py", Github Gist https://gist.github.com/endolith/5828333

[JOS] Julius O. Smith III, "Numerical Computation of Group Delay" in "Introduction to Digital Filters with Audio Applications", Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, http://ccrma.stanford.edu/~jos/filters/Numerical_Computation_Group_Delay.html, referenced 2014-04-02 or https://www.dsprelated.com/freebooks/filters/Numerical_Computation_Group_Delay.html

[JOS_time_constant] Julius O. Smith III, "Time Constant of One Pole", from "Introduction to digital filters", https://ccrma.stanford.edu/~jos/fp/Time_Constant_One_Pole.html

[Lyons] Richard Lyons, "Understanding Digital Signal Processing", 3rd Ed., Prentice Hall, 2010.

[Lyons08] Richard Lyons, "Computing the Group Delay of a Filter", DSPrelated.com, 19. Nov. 2008, https://www.dsprelated.com/showarticle/69.php

[Scipy_1175] Scipy issue #1175 "Additional functions for scipy.signal", 25. Apr. 2013, https://github.com/scipy/scipy/issues/1175

[Scipy_9310] Scipy issue #9310 "signal.group_delay fails for narrow band filters", 26. Sep. 2018, https://github.com/scipy/scipy/issues/9310

[Smith99] Steven W. Smith, "The Scientist and Engineer's Guide to Digital Signal Processing", 3rd Ed., 1999, https://www.DSPguide.com

[SPA] https://github.com/spatialaudio/group-delay-of-filters

[Yates_2020] Randy Yates, "Fixed-Point Arithmetic: An Introduction", 15. Sep. 2020, http://www.digitalsignallabs.com/fp.pdf

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z