pyfda Documentation

Release v0.9.0b1

Christian Muenker

Apr 09, 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.

docs/source/screenshots/pyfda_screenshot.png

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

Supported Python versions are 3.7 … 3.11, 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** and **Qt5**
- **numpy**
- **numexpr**
- **scipy**: **1.2.0** or higher
- **matplotlib**: **3.1** or higher
- **Markdown**

Optional libraries:

- **mplcursors** for annotating cursors
- **docutils** 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.

Start pyfda with

```
> pyfdax
```

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

1.6.2 User Interface

- only widgets needed for the currently selected design method are visible
- specifications are remembered when switching between filter design methods
- 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 Modular Architecture

Facilitate the implementation of new filter design / analysis / display methods. Generate your own

- Filter design widgets with your algorithm
- Plotting widgets
- Input widgets
- Fixpoint filter widgets, using the integrated `Fixed()` class

1.6.5 Import / Export

- Filter designs in pickled and in numpy's NPZ-format
- Coefficients and poles/zeros as comma-separated values (CSV) 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 Why yet another filter design tool?

- **Education:** Provide an easy-to-use FOSS tool for demonstrating basic digital stuff and filter design interactively that also works with the limited resolution of a beamer.
- **Show-off:** Demonstrate that Python is a potent tool for digital signal processing as well.
- **Fixpoint filter design:** 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 / Roadmap

For details, see CHANGELOG.md.

1.9 Planned features

1.9.1 Started

- Dark mode
- HDL filter implementation: Implementing a fixpoint filter in VHDL / Verilog without errors requires some experience, verifying the correct performance in a digital design environment with very limited frequency domain simulation options is even harder.

1.9.2 Ideas (help wanted)

- Keep multiple designs in memory, switch between them, compare results and store the whole set
- 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

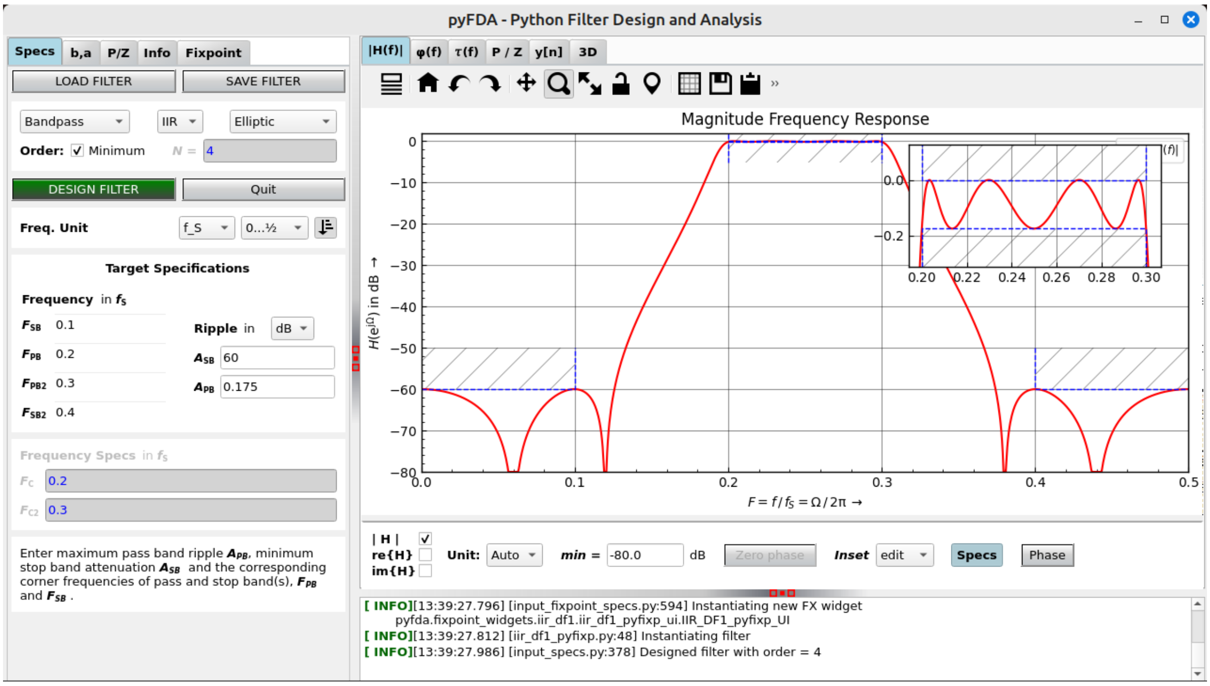

Fig. 2.1: Screenshot of pyfda

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.

2.1 Input Specs

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

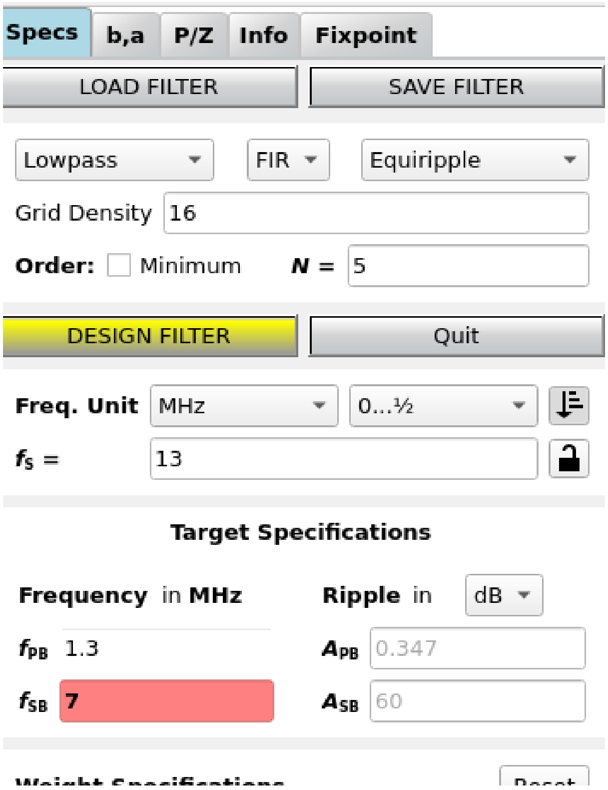

Fig. 2.2: Screenshot of specs input window

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.

2.1.1 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.

2.1.2 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):

2.1.3 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()`.

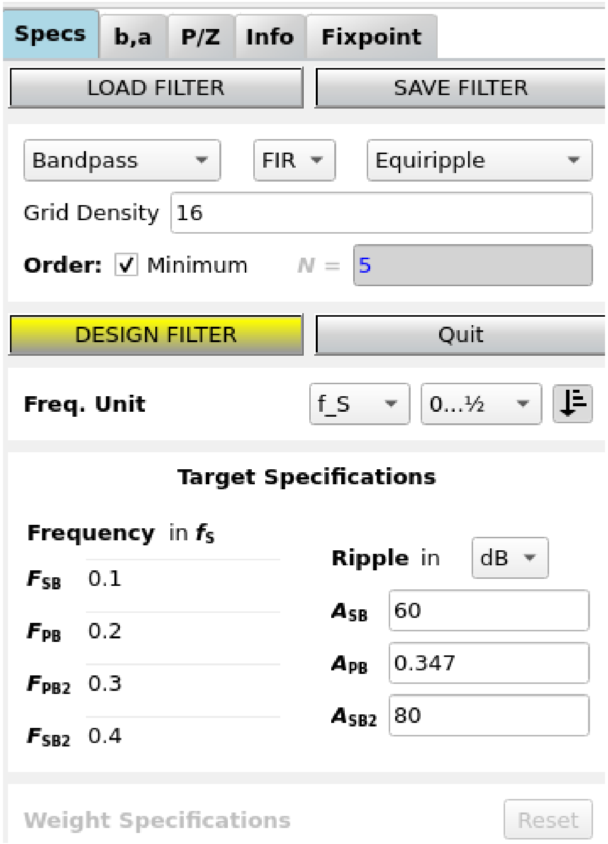

Fig. 2.3: Displaying normalized frequencies

2.1.4 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:

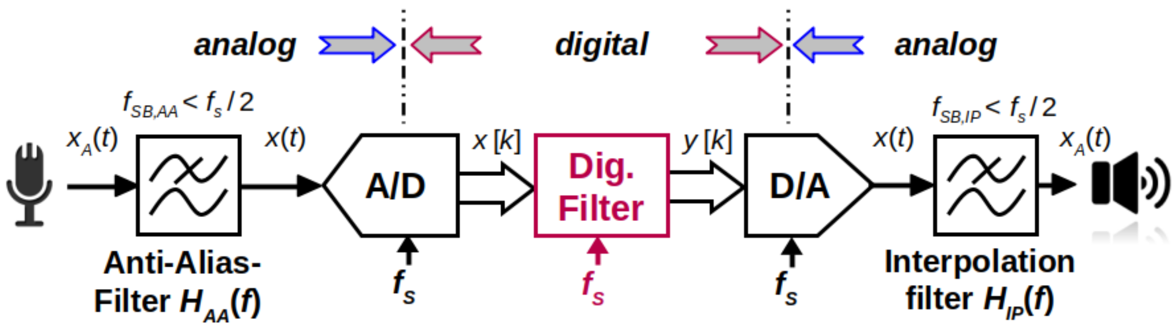

Fig. 2.4: A simple signal processing system

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.

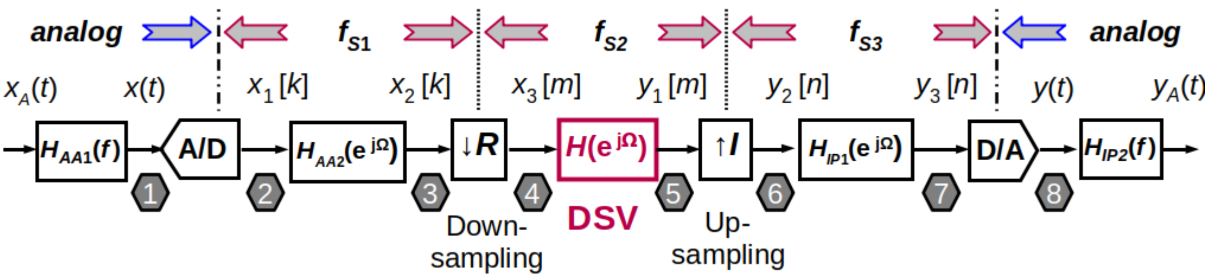

Fig. 2.5: A signal processing system with multiple sampling frequencies

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*.

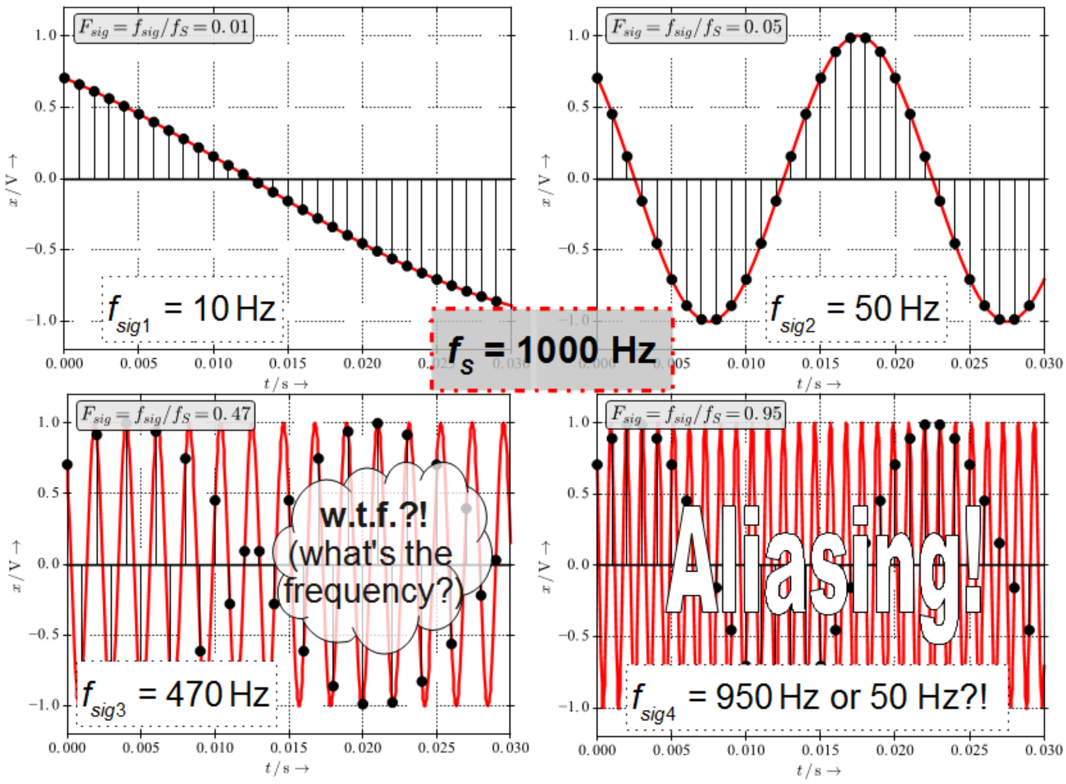

Fig. 2.6: Sampling with $f_S = 1000$ Hz of sinusoids with 4 different frequencies

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.

2.1.5 Development

More info on this widget can be found under *input_specs*.

2.2 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.

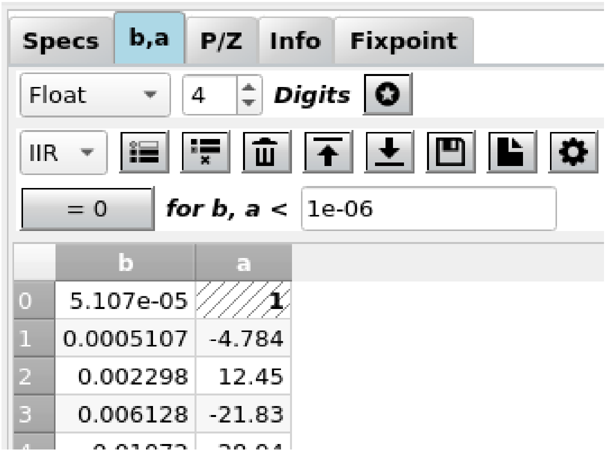

Fig. 2.7: Screenshot of the coefficients tab for floating point coefficients

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

2.2.1 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.

2.2.2 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.

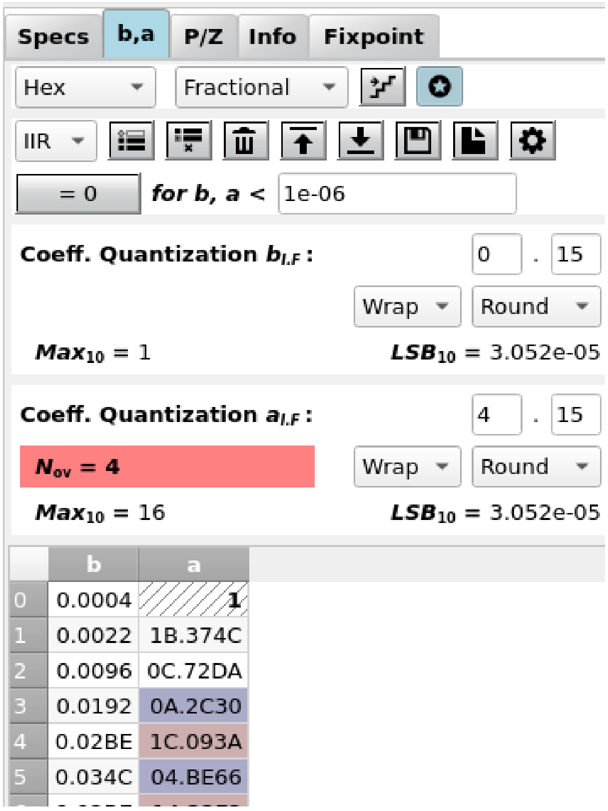

Fig. 2.8: Screenshot of the coefficients tab for fixpoint formats (binary display)

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

$$c^2 = a^2 + b^2$$

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*.

2.2.3 Development

More info on this widget can be found under *input_coeffs*.

2.3 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.

2.3.1 Cartesian format

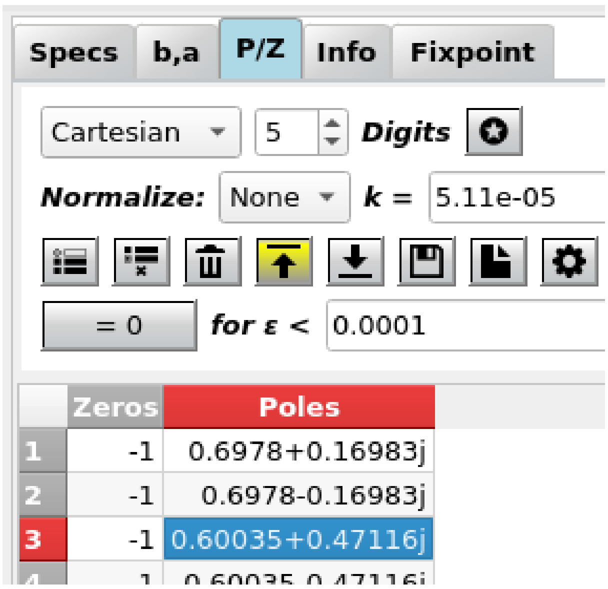

Fig. 2.9: Screenshot of the pole/zero tab in cartesian format

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

2.3.2 Polar format

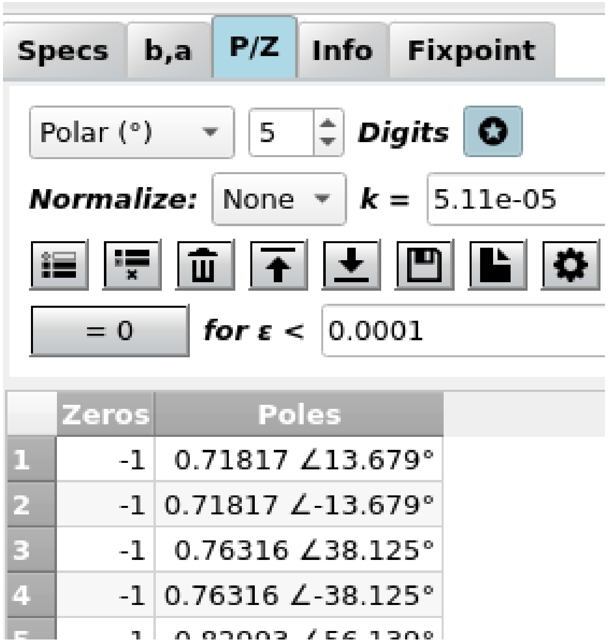

Fig. 2.10: Screenshot of the pole/zero tab in polar format with activated "Format" button

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 '<'.

2.3.3 Development

More info on this widget can be found under *input_pz*.

2.4 Input Info

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

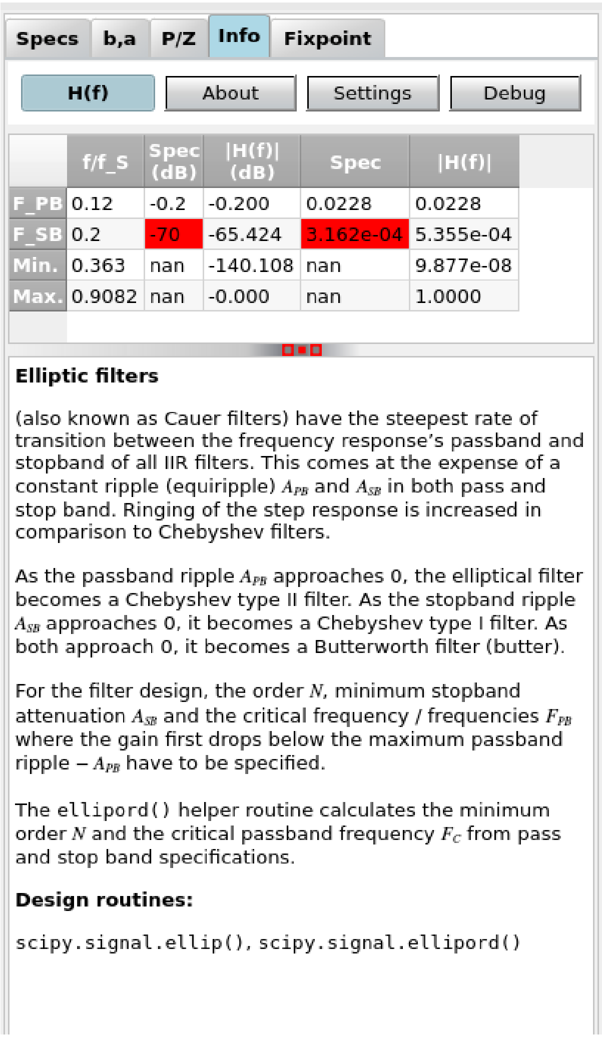

Fig. 2.11: Screenshot of the info tab

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).

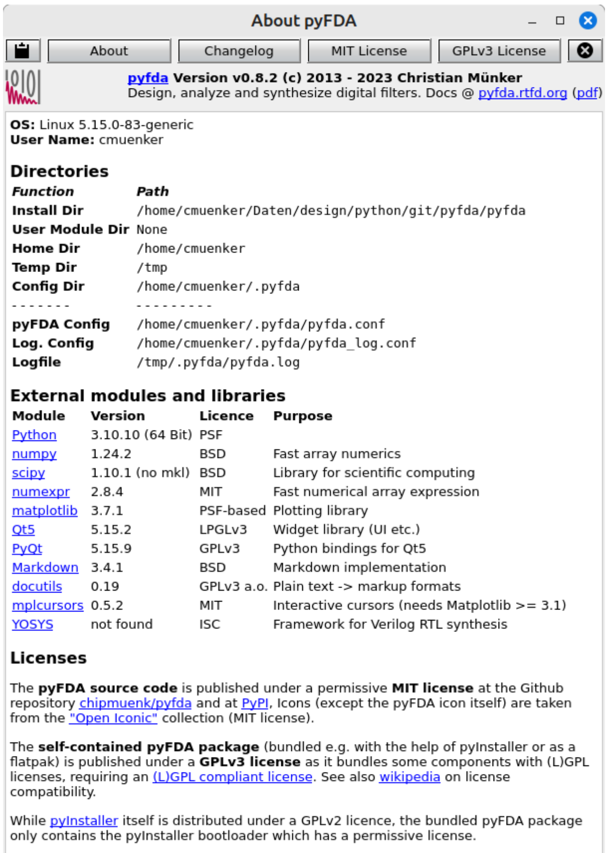

Fig. 2.12: Screenshot of the "About" pop-up window

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

2.4.1 Development

More info on this widget can be found under *input_info*.

2.5 Fixpoint Specs

2.5.1 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.

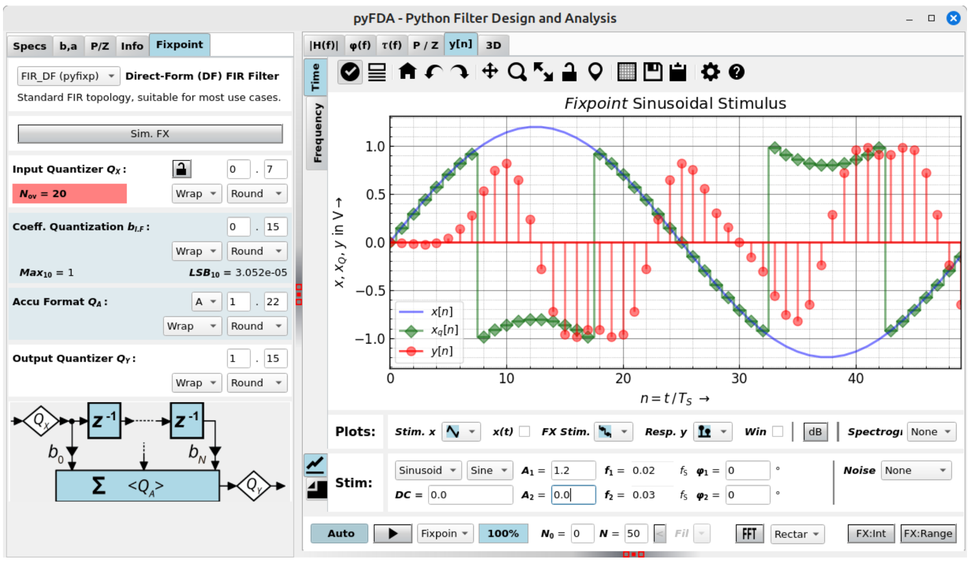

Fig. 2.13: Fixpoint parameter entry widget (overflow = wrap)

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.

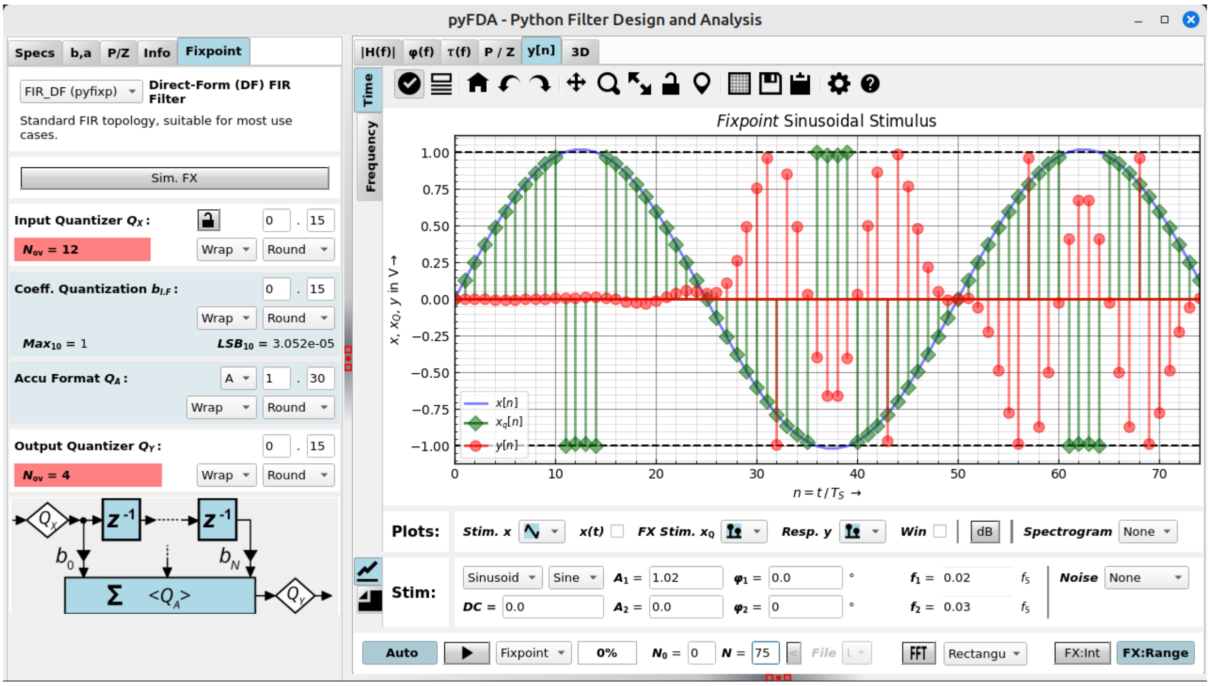

Fig. 2.14: Fixpoint simulation results (time domain)

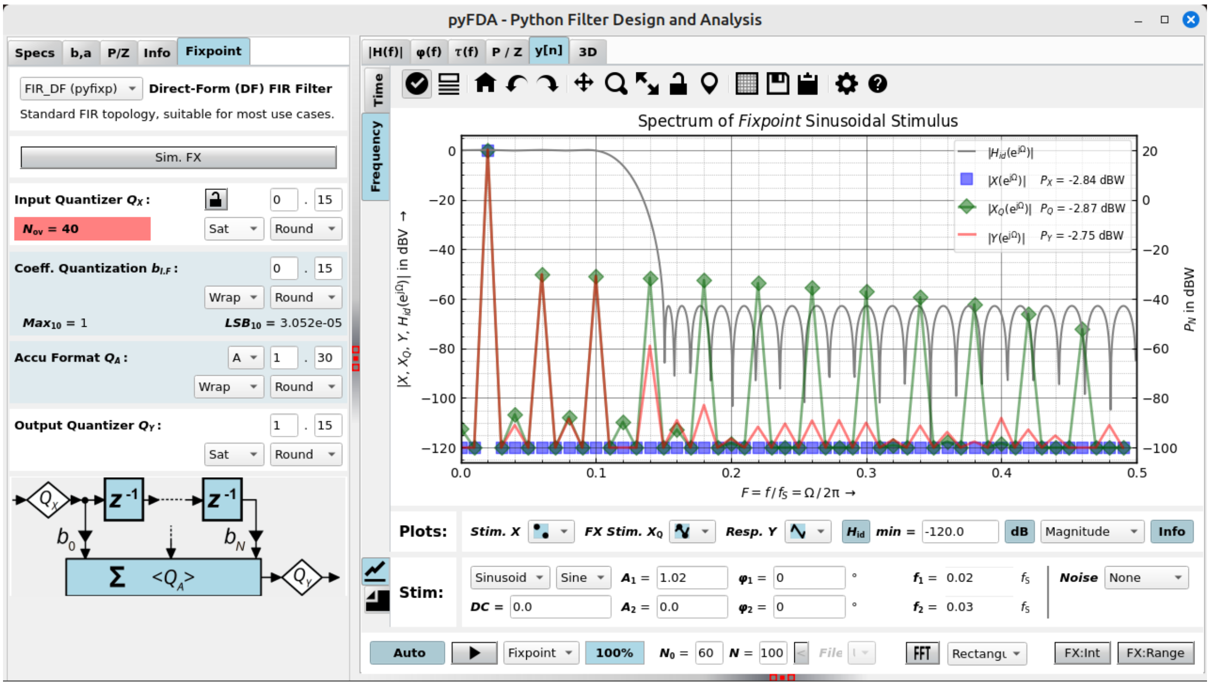

Fig. 2.15: Fixpoint simulation results (frequency domain)

```
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*.

2.5.2 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.

2.5.3 Development

More info on this widget can be found under *input_widgets.input_fixpoint_specs*.

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

2.6 Plot H(f)

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

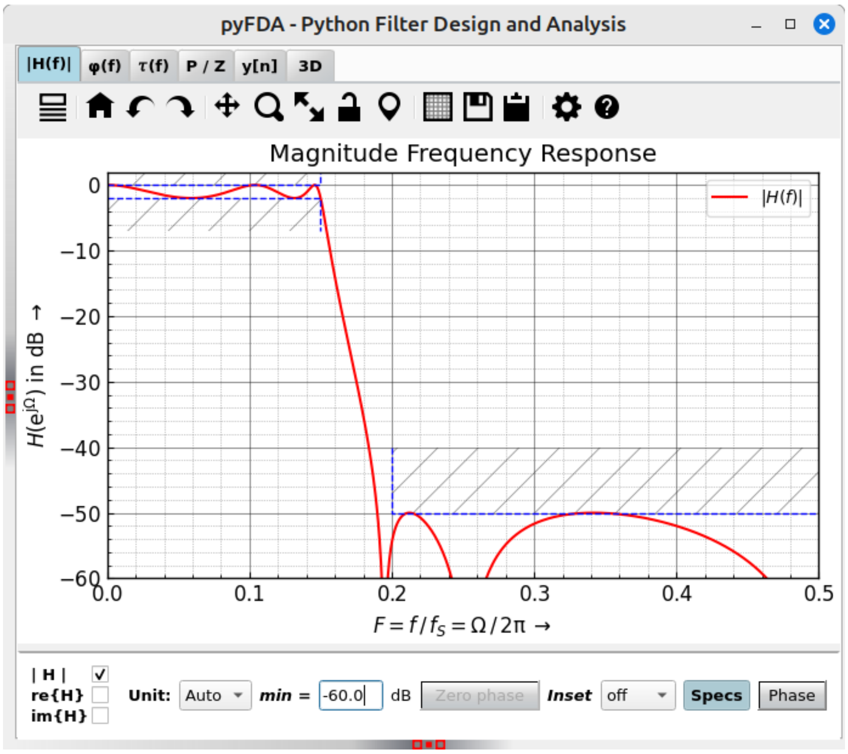

Fig. 2.16: Screenshot of the |*H(f)*| tab

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.

2.6.1 Development

More info on this widget can be found under *plot_hf*.

2.7 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).

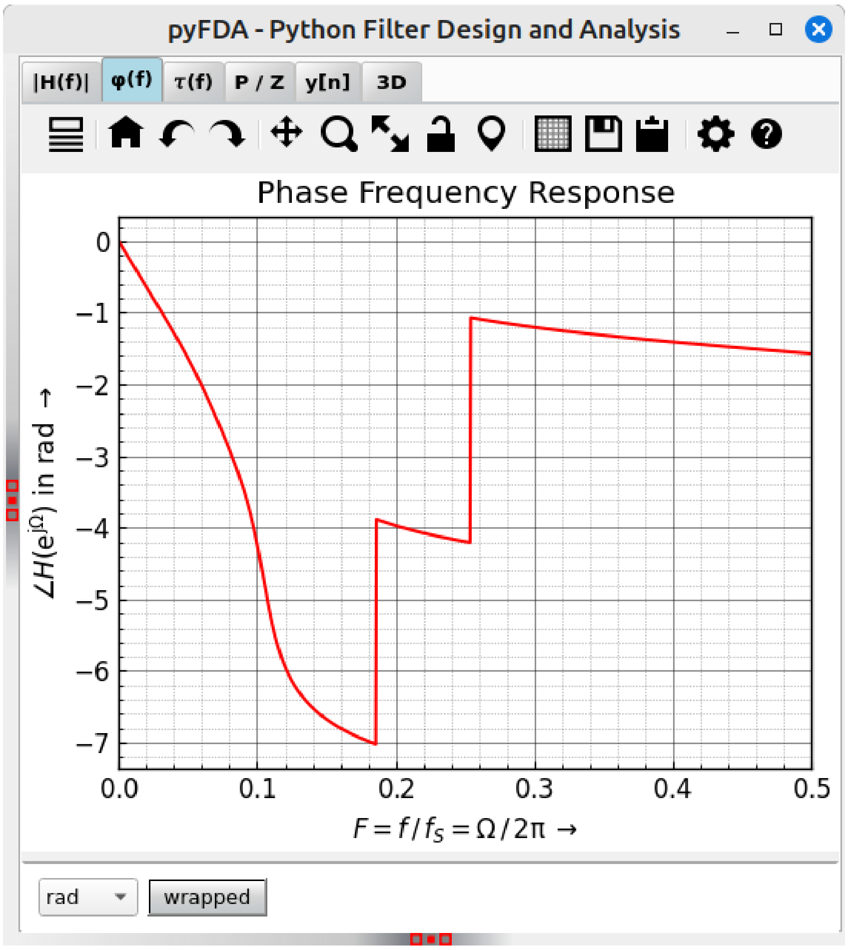

Fig. 2.17: Screenshot of the $\varphi(f)$ tab

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

2.7.1 Development

More info on this widget can be found under *plot_phi*.

2.8 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.

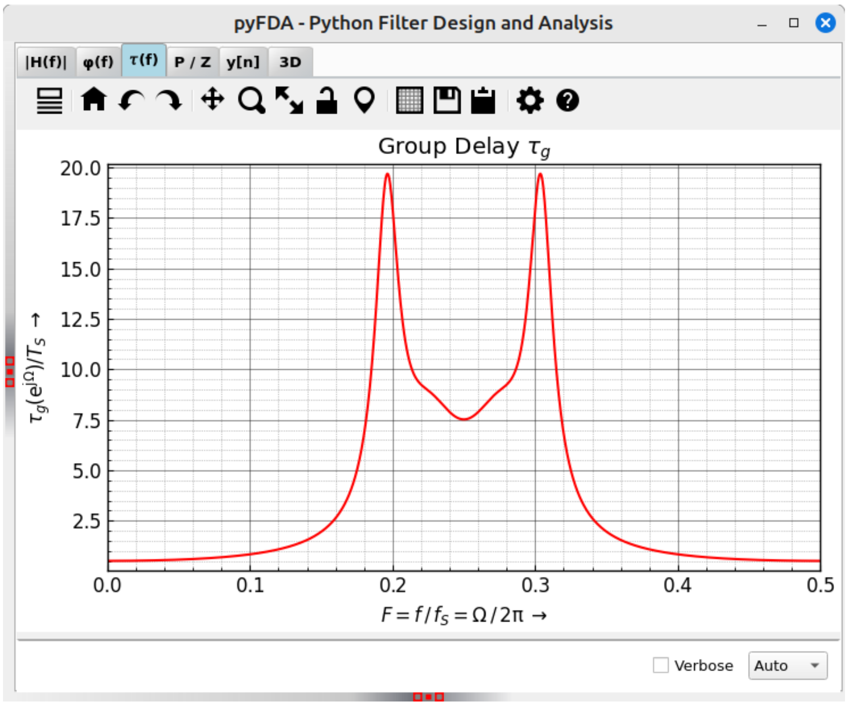

Fig. 2.18: Screenshot of the $\tau(f)$ tab

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()`

2.8.1 Development

More info on this widget can be found under *plot_tau_g*.

2.9 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).

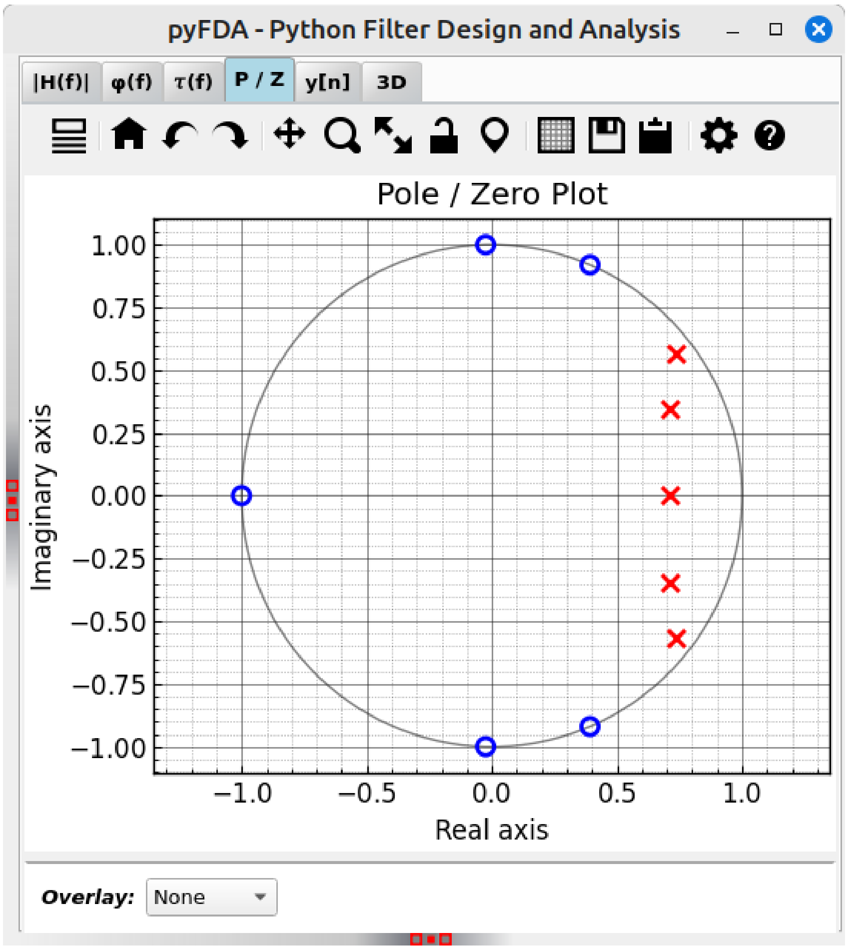

Fig. 2.19: Screenshot of the P/Z tab

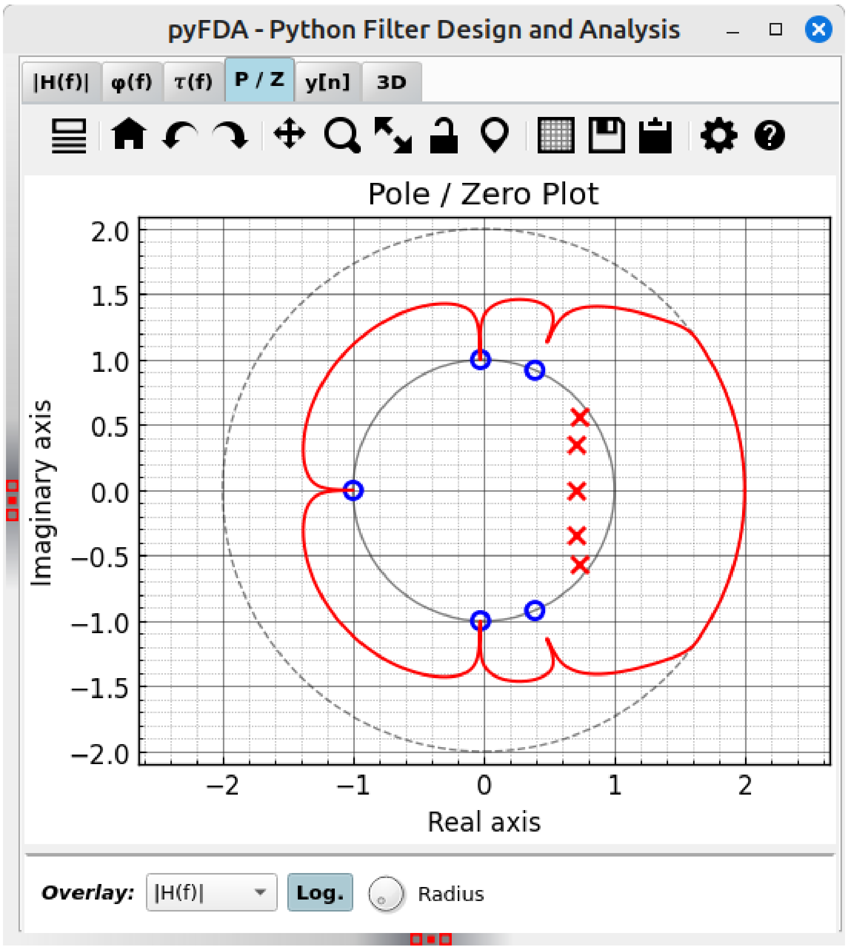

Fig. 2.20: Screenshot of the P/Z tab with overlayed H(f) plot

2.9.1 Development

More info on this widget can be found under *plot_pz*.

2.10 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).

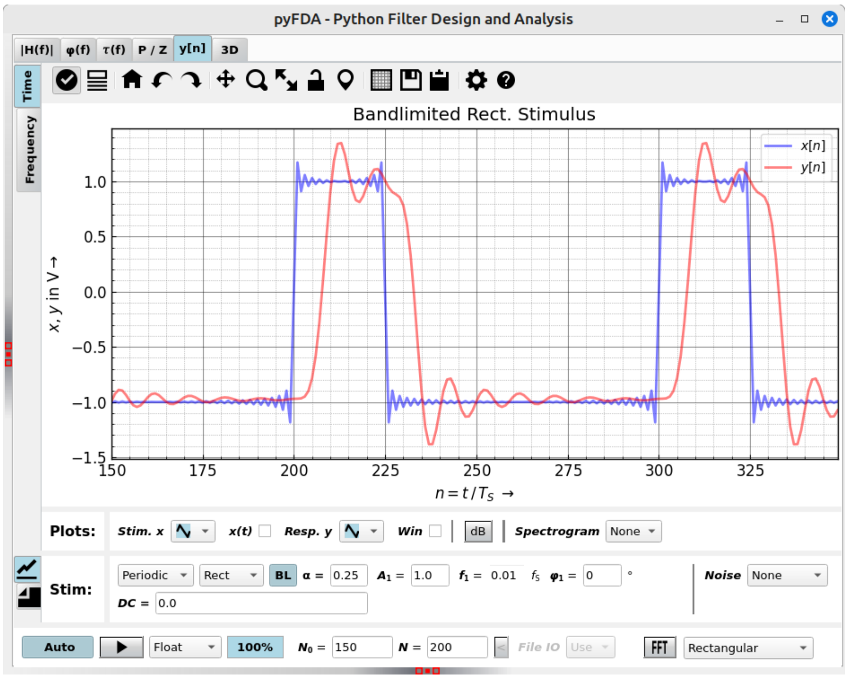

Fig. 2.21: Screenshot of the y[n] tab (time domain)

This tab is split into several subwindows:

2.10.1 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.

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).

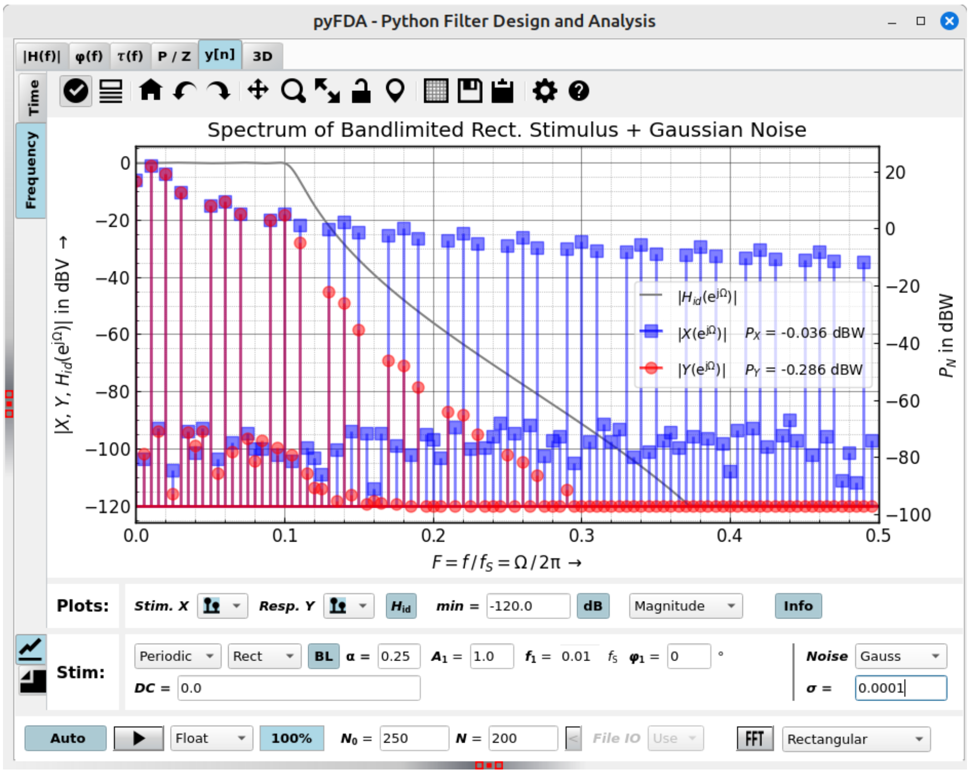

Fig. 2.22: Screenshot of the y[n] tab (frequency domain)

- 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.

2.10.2 Plots

What will be plotted and how.

2.10.3 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.

2.10.4 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).

2.10.5 Development

More info on this widget can be found under *plot_impz*.

2.11 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.

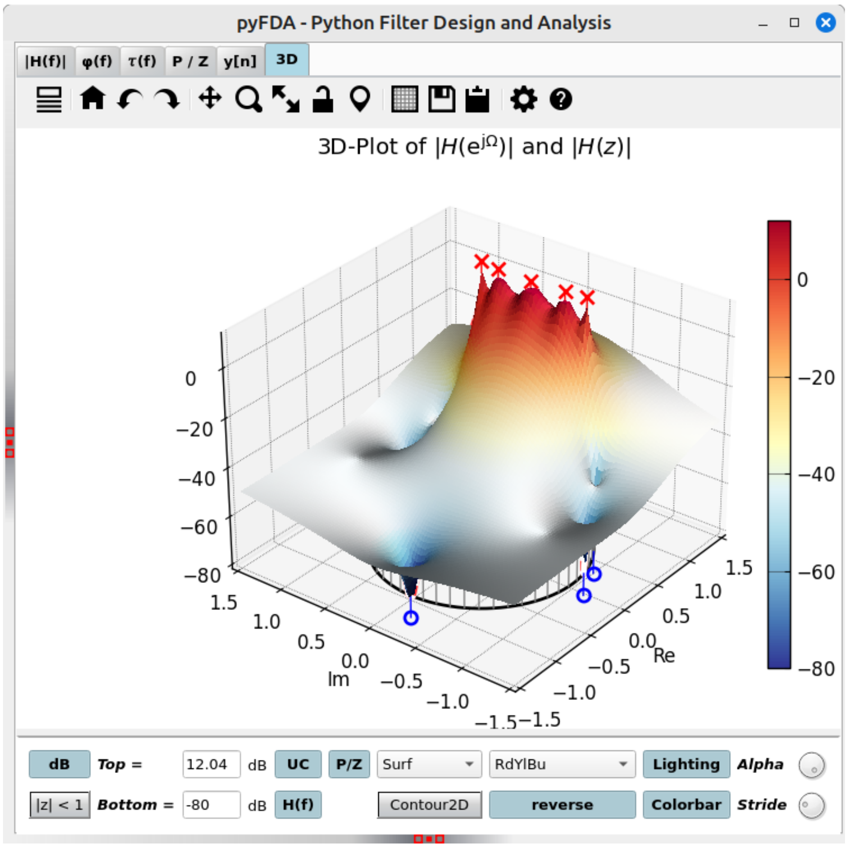

Fig. 2.23: Screenshot of the 3D tab (surface plot)

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.

2.11.1 Development

More info on this widget can be found under *plot_3d*.

Some documentation treats general filter design and fixpoint arithmetics stuff.

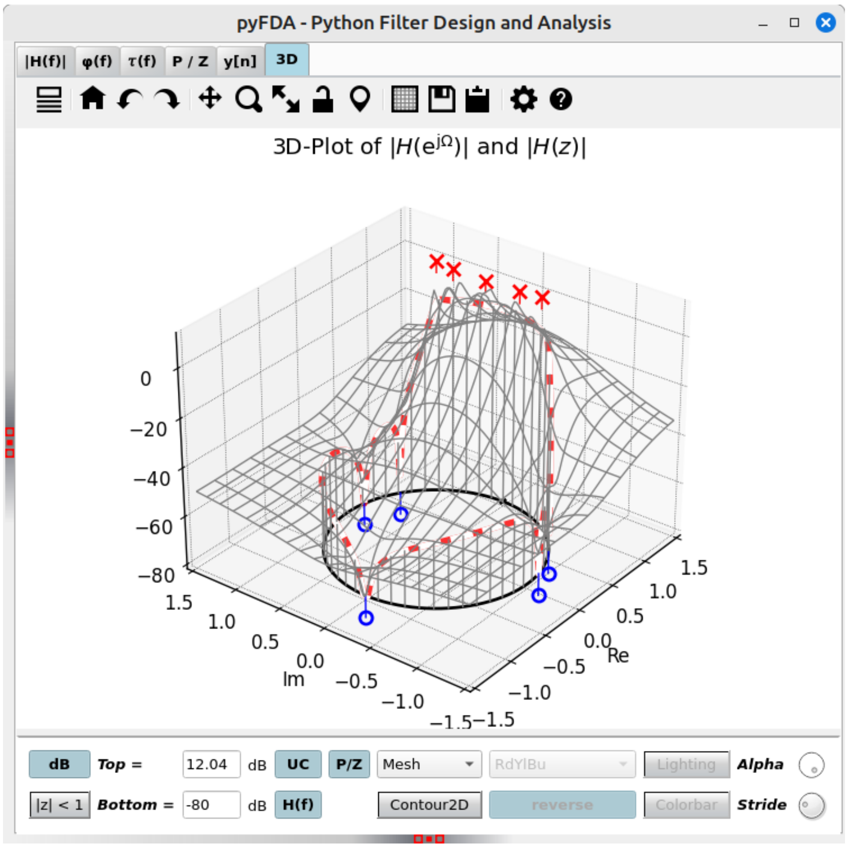

Fig. 2.24: Screenshot of the 3D tab (mesh plot)

2.12 Fixpoint Arithmetics

2.12.1 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 …

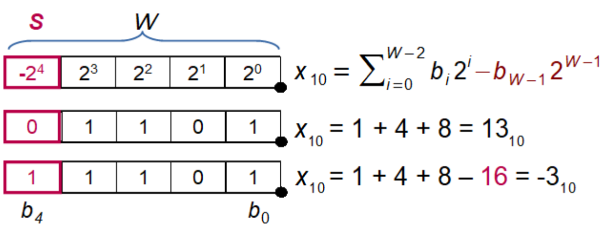

Fig. 2.25: Signed integer number in twos-complement format

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

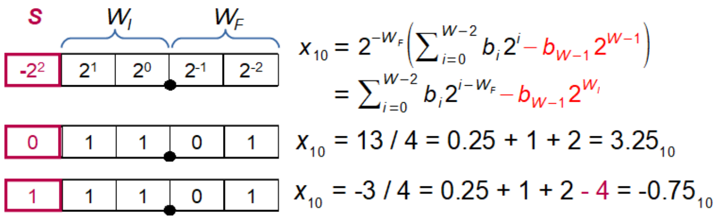

Fig. 2.26: Signed fractional number in twos-complement format

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.

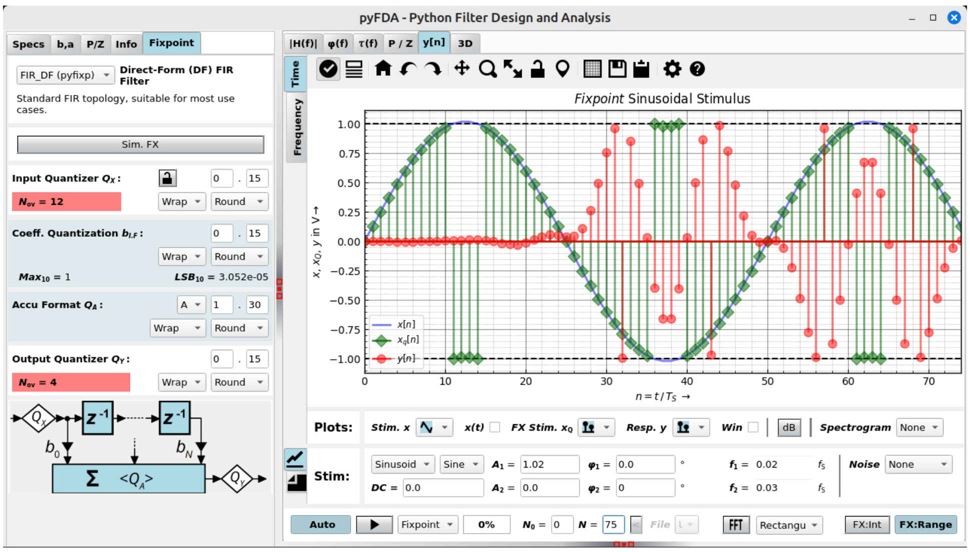

Fig. 2.27: Fixpoint filter response with overflows

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.

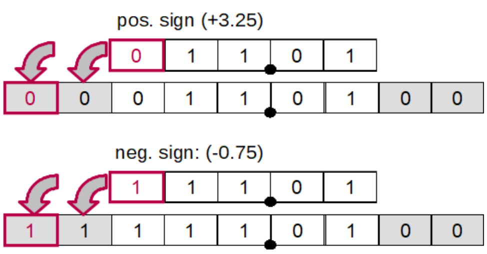

Fig. 2.28: Sign extension of integer and fractional numbers

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.

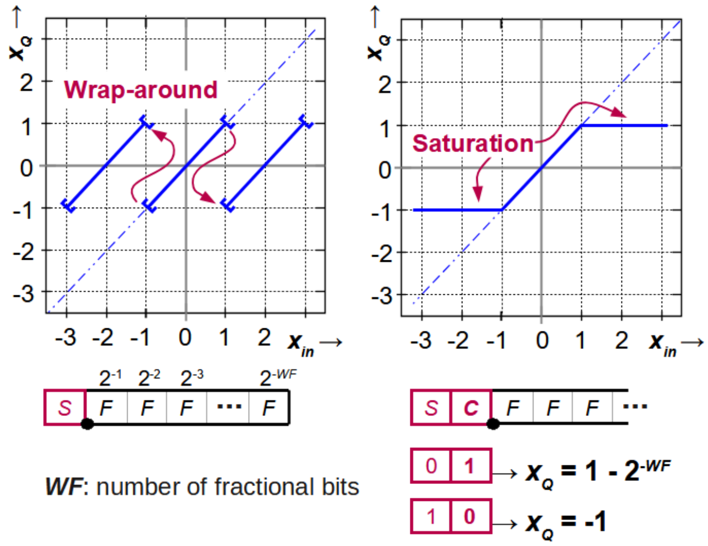

Fig. 2.29: Overflow behaviour with wrap-around or saturation

Truncation and rounding

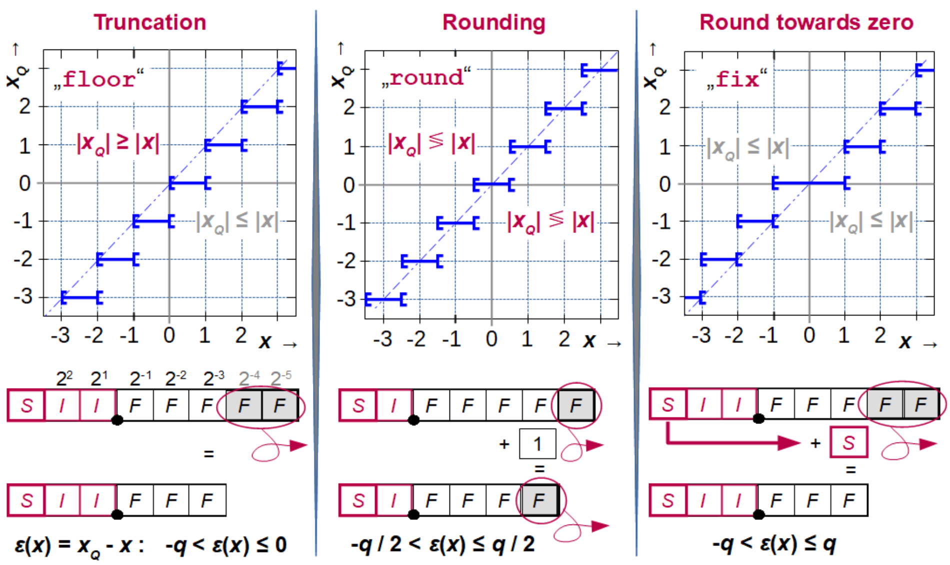

Fig. 2.30: Reducing fractional word length using truncation, rounding and round-towards-zero

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)


              =

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

2.13 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.

2.14 Customization

You can customize pyfda behaviour in some configuration files:

2.14.1 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]
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:
#
```

(continues on next page)

(continued from previous page)

```
# 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
# 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
```

(continues on next page)

(continued from previous page)

```
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


########################
[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_nmigen_ui = ${Common:FIR}
# fx_delay = ['Equiripple', 'Delay'] # need to fix fx_delay and Delay modules
```

2.14.2 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
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
```

(continues on next page)

(continued from previous page)

```
[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
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 -------------------
```

(continues on next page)

(continued from previous page)

```
[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
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
```

(continues on next page)

(continued from previous page)

```
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
#------------------- 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
```

(continues on next page)

(continued from previous page)

```
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]
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
```

(continues on next page)

(continued from previous page)

```
[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
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,)
```

(continues on next page)

(continued from previous page)

```
#---- 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
```

2.14.3 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).

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

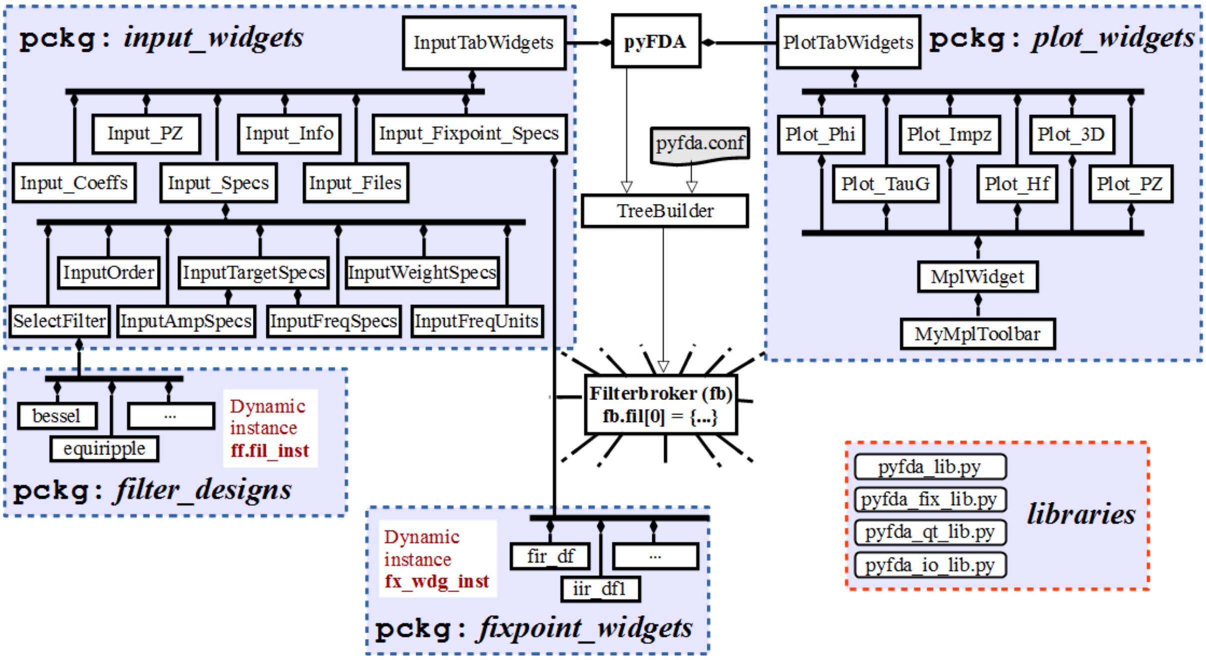

Fig. 3.1: pyfda Organization

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 *plot_hf* widget that plots the filter specifications as an overlay or the *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_20240409-112853.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.libs.pyfda_lib

3.4.4 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
```

3.4.5 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*

3.4.6 pyfda.libs.pyfda_io_lib

3.5 Libraries

pyfda contains the following libraries:

- *pyfda_lib*: General functions
- *pyfda_sig_lib*: Functions related to signal processing
- *pyfda_qt_lib*: Functions related to Qt
- *pyfda_io_lib*: Functions related to file I/O
- *pyfda_fix_lib*: Fixpoint classes and functions

3.5.1 pyfda_lib

3.5.2 pyfda_sig_lib

3.5.3 pyfda_qt_lib

3.5.4 pyfda_io_lib

3.5.5 pyfda_fix_lib

3.6 Package input_widgets

This package contains the widgets for entering / selecting parameters for the filter design.

3.6.1 input_tab_widgets

3.6.2 input_specs

3.6.3 select_filter

3.6.4 input_coeffs

3.6.5 input_pz

3.6.6 input_info

3.6.7 input_fixpoint_specs

The configuration file *libs.pyfda_template.conf* lists which fixpoint classes (e.g. `FIR_DF` and `IIR_DF1`) can be used with which filter design algorithm. *libs.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` constructs a combo box from this list with references to all successfully imported fixpoint modules. The currently selected fixpoint widget (e.g. *FIR_DF*) is imported from *Package fixpoint_widgets* together with the referenced picture.

Each fixpoint module / class contains a widget that is constructed using helper classes from *fixpoint_widgets.fixpoint_helpers.py*. The widgets allow entering fixpoint specifications like word lengths and formats for input, output and internal structures (like an accumulator) for each class. It also contains a reference to a picture showing the filter topology.

Details of the mechanism and the module are described in *input_widgets.input_fixpoint_specs*.

3.7 Package plot_widgets

Package providing widgets for plotting various time and frequency dependent filter properties

3.7.1 plot_tab_widgets

3.7.2 plot_hf

3.7.3 plot_phi

3.7.4 plot_tau_g

3.7.5 plot_pz

3.7.6 plot_impz

3.7.7 plot_3d

3.8 Package filter_widgets

Package providing various algorithms for FIR and IIR filter design.

3.8.1 `pyfda.filter_widgets.bessel`

3.9 Package fixpoint_widgets

This package contains widgets and fixpoint descriptions for simulating filter designs with fixpoint arithmetics and for converting filter designs to Verilog using the migen library. These Verilog netlists can be synthesized e.g. on an FPGA.

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).

Filter topologies are defined in the corresponding classes and can be implemented in hardware. The filter topologies use the order and the coefficients that have been determined by a filter design algorithm from the *pyfda.filter_widgets* package for a target filter specification (usually in the frequency domain). Filter coefficients are quantized according to the settings in the fixpoint widget.

Each fixpoint module / class contains a widget that is constructed using helper classes from `fixpoint_widgets.fixpoint_helpers`. The widgets allow entering fixpoint specifications like word lengths and formats for input, output and internal structures (like an accumulator) for each class. It also contains a reference to a picture showing the filter topology.

The configuration file *pyfda.conf* lists which fixpoint classes (e.g. `FIR_DF` and `IIR_DF1`) can be used with which filter design algorithm. *tree_builder* parses this file and writes all fixpoint modules into the list *fb.fixpoint_widgets_list*.

The widgets are selected and instantiated in the widget *input_widgets.input_fixpoint_specs*.

The input widget `pyfda.input_widgets.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. *FIR_DF*) is imported from *Package fixpoint_widgets* together with the referenced picture.

First, a filter widget is instantiated as `self.fx_filt_ui` (after the previous one has been destroyed).

Next, `fx_filt_ui.construct_fixp_filter()` constructs an instance `fixp_filter` of a fixpoint filter class (of e.g. *pyfda.fixpoint_widgets.fir_df*).

The widget's methods

- `response = fx_filt_ui.fx_filt.run_sim(stimulus)`
- `fx_filt_ui.fx_filt.to_verilog()`

are used for bit-true simulations and for generating Verilog code for the filter.

3.9.1 input_widgets.input_fixpoint_specs

A fixpoint filter for a given filter design is selected in this widget

3.9.2 `pyfda.fixpoint_widgets.fir_df`

CHAPTER

FOUR

LITERATURE

References

CHAPTER

FIVE

API DOCUMENTATION

5.1 `pyfda` – Main package

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[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,

[Lyons] Richard Lyons, “Understanding Digital Signal Processing”, 3rd Ed., Prentice Hall, 2010.

[Smith99] Steven W. Smith, “The Scientist and Engineer’s Guide to Digital Signal Processing”, 3rd Ed., 1999, https://www.DSPguide.com

PYTHON MODULE INDEX

p

INDEX

V