pyfda Documentation

Release 0.3.1

Christian Muenker

Apr 24, 2020

Contents:

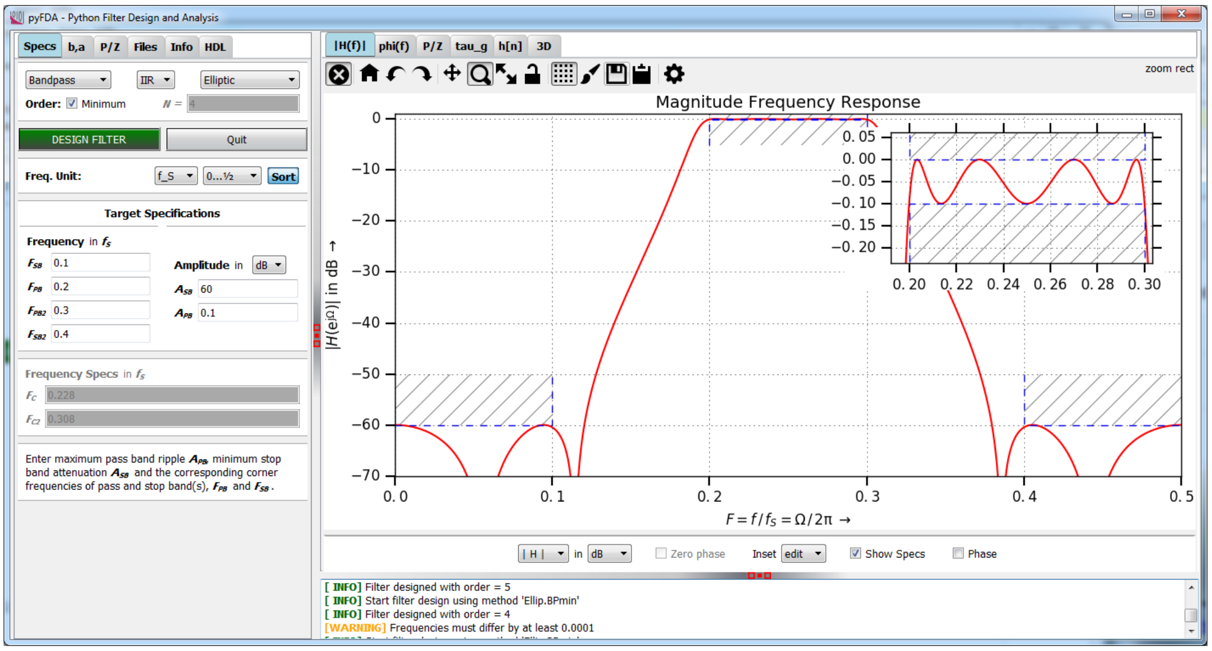
Version: 0.2.1

CHAPTER 1

User Manual

This part of the documentation is intended to describe the features of pyFDA that are relevant to a user (i.e. non-developer).

Once you have started up pyFDA, you'll see a screen similar to the following figure:



pyFDA - Python Filter Design and Analysis
Specs
b,a
P/Z
Files
Info
HDL
|H(f)|
phi(f)
P/Z
tau_g
h[n]
3D
zoom rect
Magnitude Frequency Response
$|H(e^{j\Omega})|$ in dB →
$F = f / f_S = \Omega / 2\pi$ →
DESIGN FILTER
Quit
Sort

Bandpass | IIR | Elliptic

Order: ✓ Minimum N = 4

Freq. Unit: f_S | 0...½

Target Specifications

Frequency in f_S

F_{SB}	0.1
F_{PB}	0.2
F_{PB2}	0.3
F_{SB2}	0.4

Amplitude in dB

A_{SB}	60
A_{PB}	0.1

Frequency Specs in f_S

F_C	0.228
F_{C2}	0.308

Enter maximum pass band ripple A_{PB}, minimum stop band attenuation A_{SB} and the corresponding corner frequencies of pass and stop band(s), F_{PB} and F_{SB} .

| H | in dB | Zero phase | Inset edit | ✓ Show Specs | Phase

```
[ INFO] Filter designed with order = 5
[ INFO] Start filter design using method 'Ellip.BPmin'
[ INFO] Filter designed with order = 4
[WARNING] Frequencies must differ by at least 0.0001
```

Fig. 1.1: Screenshot of pyfda

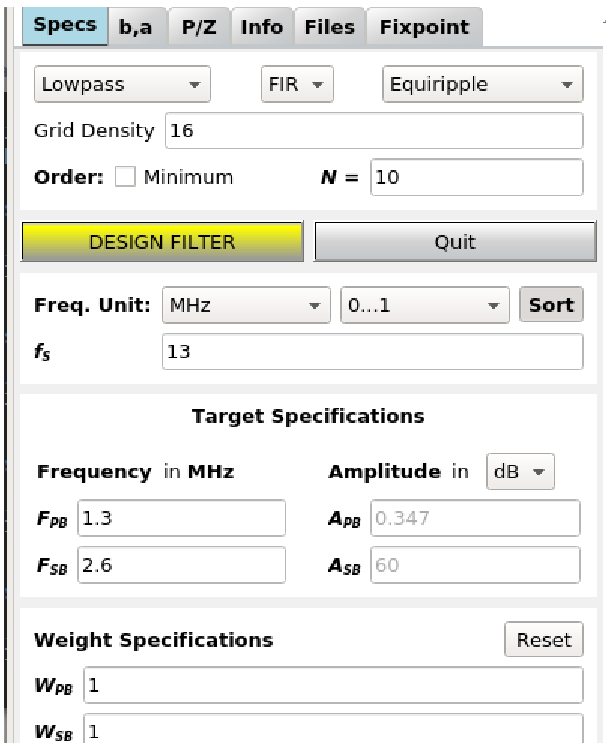
- **Inputs widgets:** On the left-hand side you see tabs for different input widgets, i.e. where you can enter and modify parameters for the filter to be designed
- **Plotting widgets** can be selected on the right hand side of the application.
- **Logger window is in the lower part of the plotting window, it can be resized** or completely closed.
    The content of the logger window can be selected, copied or cleared with a right mouse button context menu.

The invidual windows can be resized using the handles (red dots).

1.1 Input Specs

Fig. 1.2 shows a typical view of the **Specs** tab where 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



Lowpass | FIR | Equiripple

Grid Density 16

Order: ☐ Minimum N = 10

Freq. Unit: MHz | 0...1

f_S 13

Target Specifications

Frequency in MHz

F_{PB} 1.3

F_{SB} 2.6

Amplitude in dB

A_{PB} 0.347

A_{SB} 60

Weight Specifications

W_{PB} 1

W_{SB} 1

Fig. 1.2: 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.

1.1.1 Order

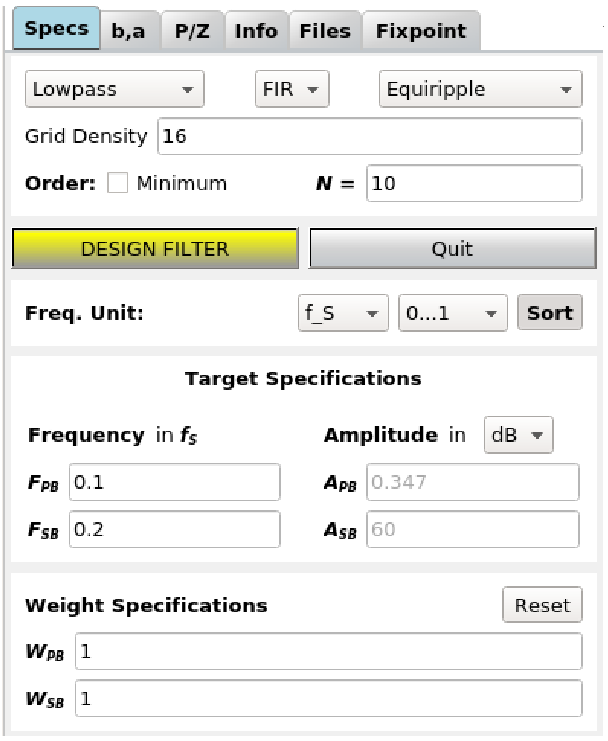
The **order** of the filter, i.e. the number of poles / zeros / delays is either specified manually or the minimum order can be estimated for many filter algorithms to fulfill a set of given specifications.

1.1.2 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. 1.3):


Specs | b,a | P/Z | Info | Files | Fixpoint

Lowpass | FIR | Equiripple

Grid Density: 16

Order: ☐ Minimum   N = 10

DESIGN FILTER | Quit

Freq. Unit: f_S | 0...1 | Sort

Target Specifications

Frequency in f_S		Amplitude in dB	
F_{PB}	0.1	A_{PB}	0.347
F_{SB}	0.2	A_{SB}	60

Weight Specifications (Reset)

W_{PB}	1
W_{SB}	1

Fig. 1.3: Displaying normalized frequencies

1.1.3 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.pyfda_lib.unit2lin()` and `pyfda.pyfda_lib.lin2unit()`.

1.1.4 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:

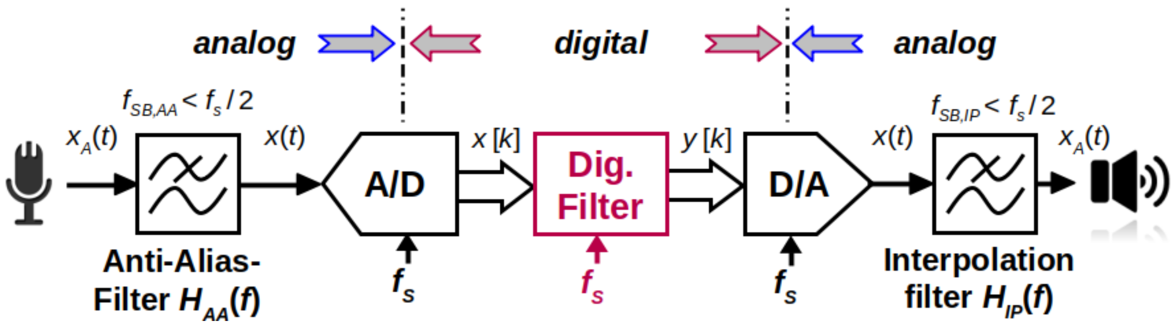


Fig. 1.4: A simple signal processing system

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.

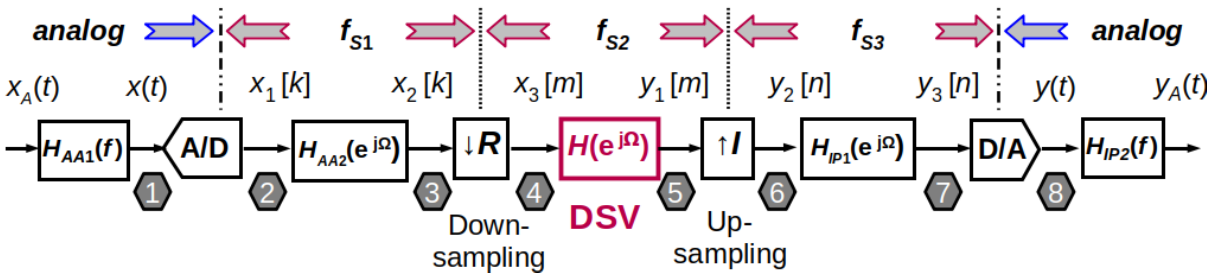


Fig. 1.5: A signal processing system with multiple sampling frequencies

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. 1.6) [Smith99]. This effect is called *aliasing*.

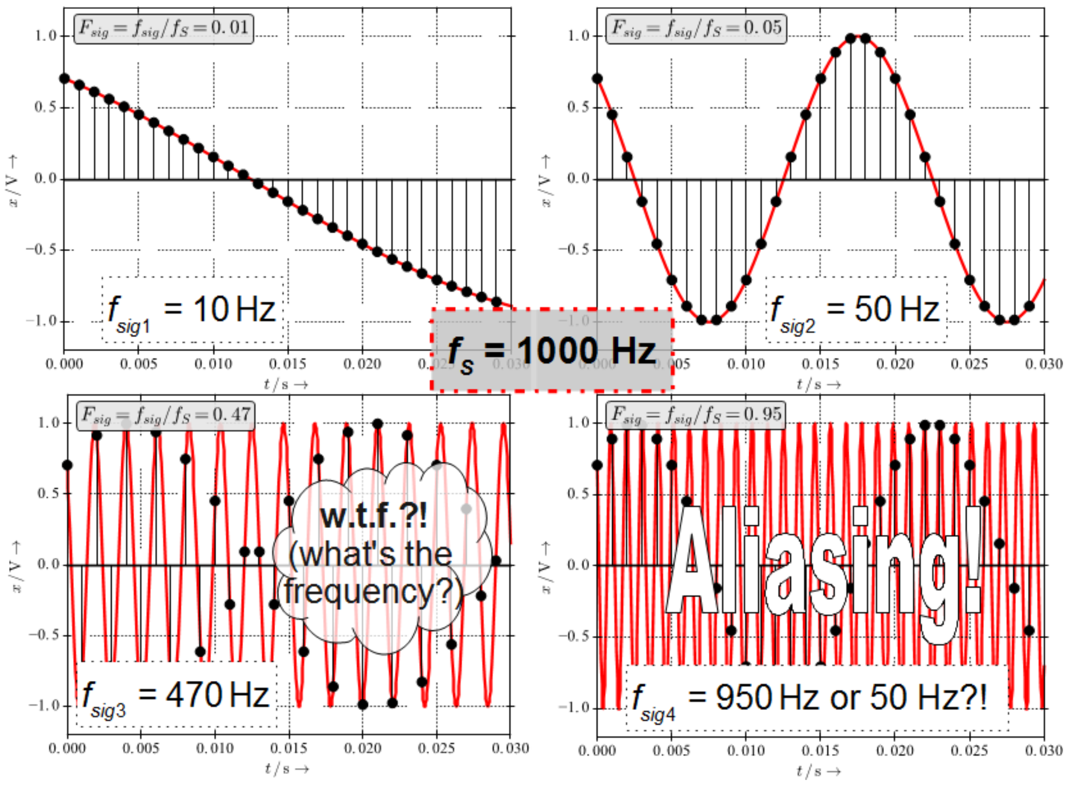


Fig. 1.6: Sampling with $f_S = 1000$ Hz of sinusoids with 4 different frequencies

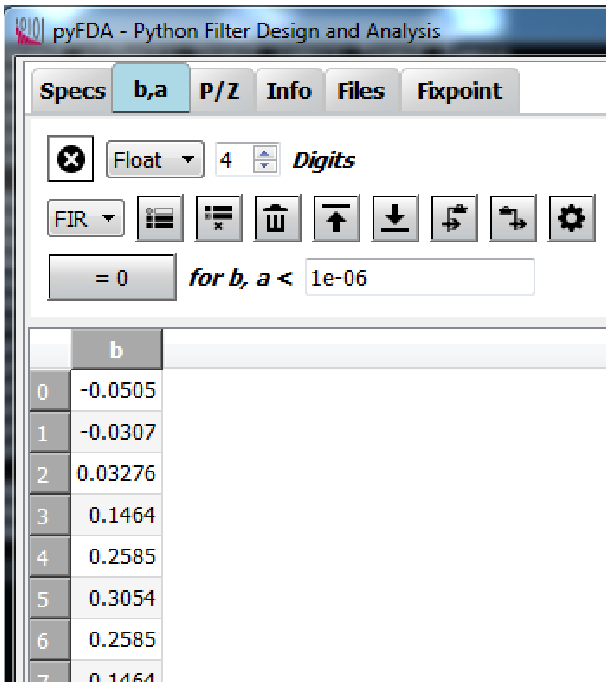
When sampling with f_S, the maximum signal bandwidth B that can represented and reconstructed without errors is given by $B < f_S/2 = f_{Ny}$. This is also called the *Nyquist frequency* or *bandwidth* f_{Ny}. Some filter design tools and algorithms normalize frequencies w.r.t. to f_{Ny} instead of f_S.

1.1.5 Development

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

1.2 Input Coeffs

Fig. 1.7 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.



	b
0	-0.0505
1	-0.0307
2	0.03276
3	0.1464
4	0.2585
5	0.3054
6	0.2585

Fig. 1.7: 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 ($N > 100$).

1.2.1 Fixpoint

When the format is not float, the fixpoint options are displayed as in Fig. 1.8. Here, the format *Binary* has been set.

Fixpoint Formats

Coefficients can be displayed in float format (the format returned by the filter design algorithm) with the maximum precision. This is also called "Real World Value" (RWV).

Any other format (Binary, Hex, Decimal, CSD) is a fixpoint format with a fixed number of binary places which triggers the display of further 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

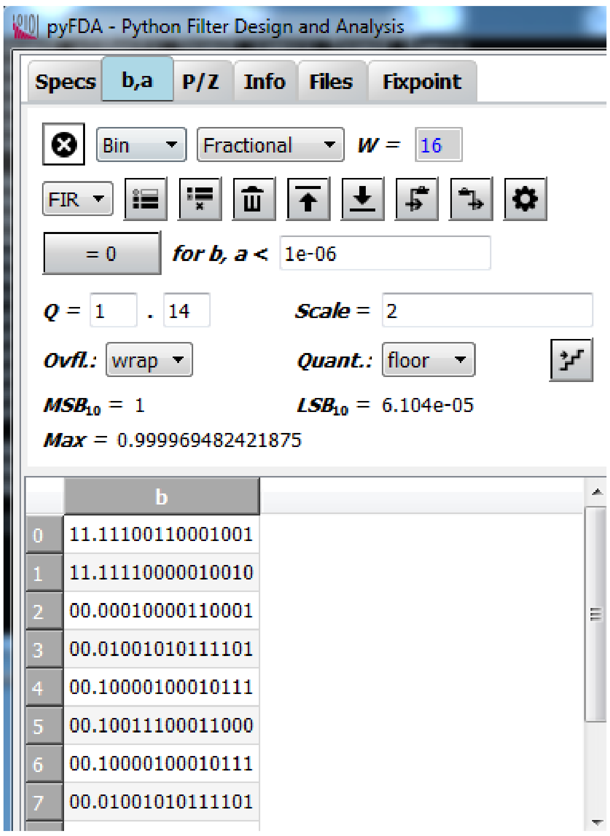
The scale between floating and fixpoint format is determined by partitioning of the word length `W` into integer and fractional places `WI` and `WF`. In general, `W = WI + WF + 1` where the "`+ 1`" accounts for the sign bit.

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

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



Bin | Fractional | *W* = 16

FIR

= 0 *for b, a <* 1e-06

Q = 1 . 14 *Scale* = 2

Ovfl.: wrap *Quant.:* floor

MSB$_{10}$ = 1 *LSB*$_{10}$ = 6.104e-05

Max = 0.999969482421875

	b
0	11.11100110001001
1	11.11110000010010
2	00.00010000110001
3	00.01001010111101
4	00.10000100010111
5	00.10011100011000
6	00.10000100010111
7	00.01001010111101

Fig. 1.8: Screenshot of the coefficients tab for fixpoint formats

It is important to understand that these settings only influence the *display* of the coefficients, the frequency response etc. is only updated when the quantize icon (the staircase) is clicked AND afterwards the changed coefficients are saved to the dict (downwards arrow). However, when you do a fixpoint simulation or generate Verilog code from the fixpoint tab, the selected word format is used for the coefficients.

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`)

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.

```
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       =  7 (INT) or 7/8 (RWV)
```

1.2.2 Development

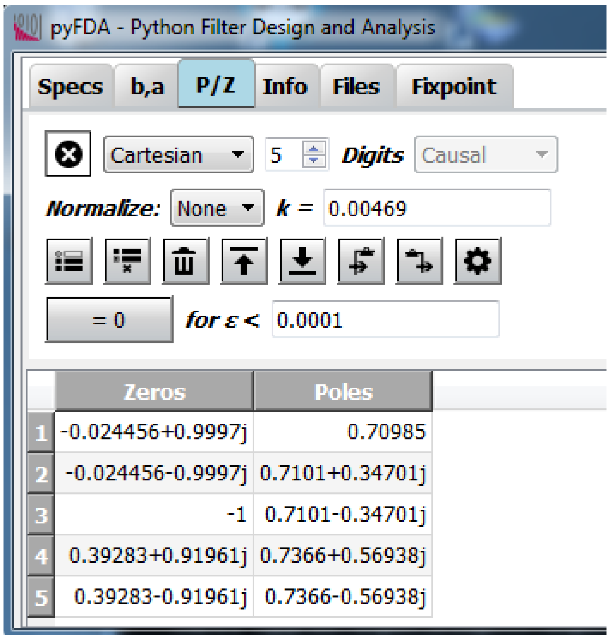
More info on this widget can be found under *input_coeffs*.

1.3 Input P/Z

Fig. 1.9 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 or update the coefficients.

In the top row, the display of poles and zeros can be disabled as an update can be time consuming for high order filters ($N > 100$).

1.3.1 Cartesian format



Cartesian, 5 Digits

Normalize: None, k = 0.00469

= 0 for ε < 0.0001

	Zeros	Poles
1	-0.024456+0.9997j	0.70985
2	-0.024456-0.9997j	0.7101+0.34701j
3	-1	0.7101-0.34701j
4	0.39283+0.91961j	0.7366+0.56938j
5	0.39283-0.91961j	0.7366-0.56938j

Fig. 1.9: Screenshot of the pole/zero tab in cartesian format

Poles and zeros are displayed in cartesian format (x and y) by default as shown in Fig. 1.9.

1.3.2 Polar format

Alternatively, poles and zeros can be displayed and edited in polar format (radius and angle) as shown in Fig. 1.10. Especially for zeros which typically sit on the unit circle ($r = 1$) this format may be more suitable.

1.3.3 Development

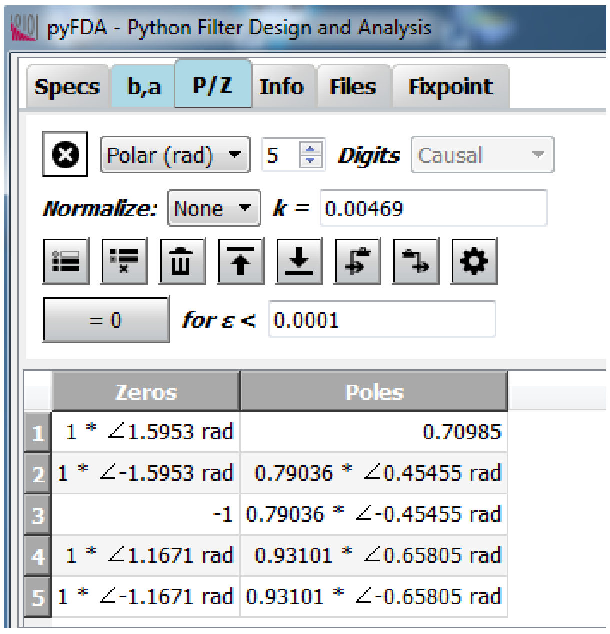
More info on this widget can be found under *input_pz*.

1.4 Input Info

Fig. 1.11 shows a typical view of the **Info** tab where information on the current filter design and design algorithm is displayed.

In the top row, checkboxes select which information is displayed.

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



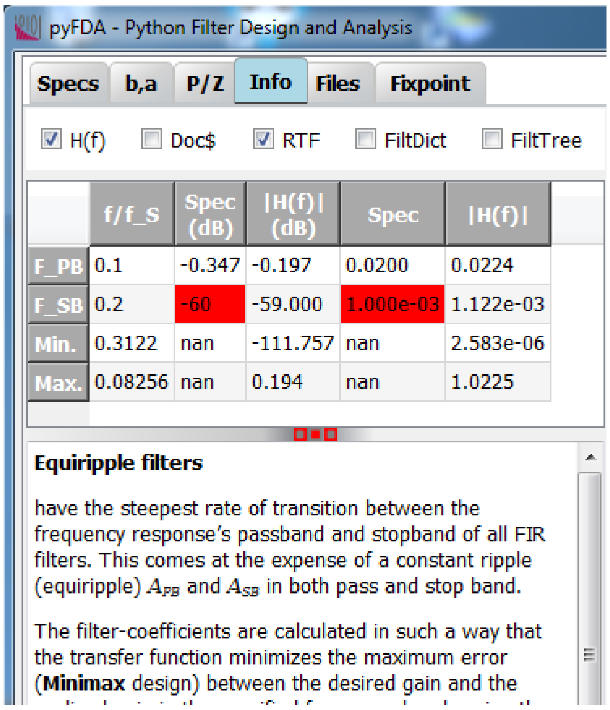
Polar (rad) 5 *Digits* Causal

Normalize: None *k* = 0.00469

= 0 *for ε <* 0.0001

	Zeros	Poles
1	1 * ∠1.5953 rad	0.70985
2	1 * ∠-1.5953 rad	0.79036 * ∠0.45455 rad
3	-1	0.79036 * ∠-0.45455 rad
4	1 * ∠1.1671 rad	0.93101 * ∠0.65805 rad
5	1 * ∠-1.1671 rad	0.93101 * ∠-0.65805 rad

Fig. 1.10: Screenshot of the pole/zero tab in polar format



☑ H(f) ☐ Doc$ ☑ RTF ☐ FiltDict ☐ FiltTree

	f/f_S	Spec (dB)	|H(f)| (dB)	Spec	|H(f)|
F_PB	0.1	-0.347	-0.197	0.0200	0.0224
F_SB	0.2	-60	-59.000	1.000e-03	1.122e-03
Min.	0.3122	nan	-111.757	nan	2.583e-06
Max.	0.08256	nan	0.194	nan	1.0225

Equiripple filters

have the steepest rate of transition between the frequency response's passband and stopband of all FIR filters. This comes at the expense of a constant ripple (equiripple) A_{PB} and A_{SB} in both pass and stop band.

The filter-coefficients are calculated in such a way that the transfer function minimizes the maximum error (**Minimax** design) between the desired gain and the

Fig. 1.11: Screenshot of the info tab

The **Doc$** checkbox selects whether docstring info from the corresponding python module is displayed. The **RTF** checkbox selects Rich Text Format for the documentation.

The **FiltDict** and **FiltTree** checkboxes are for debugging purposes only.

1.4.1 Development

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

1.5 Input Files

Fig. 1.12 shows a typical view of the **Files** tab where filter designs can be saved and loaded.

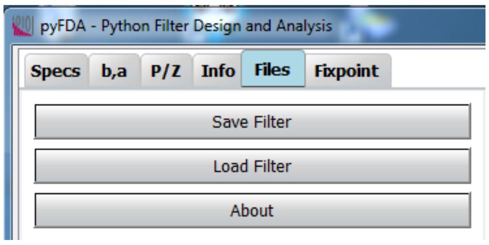
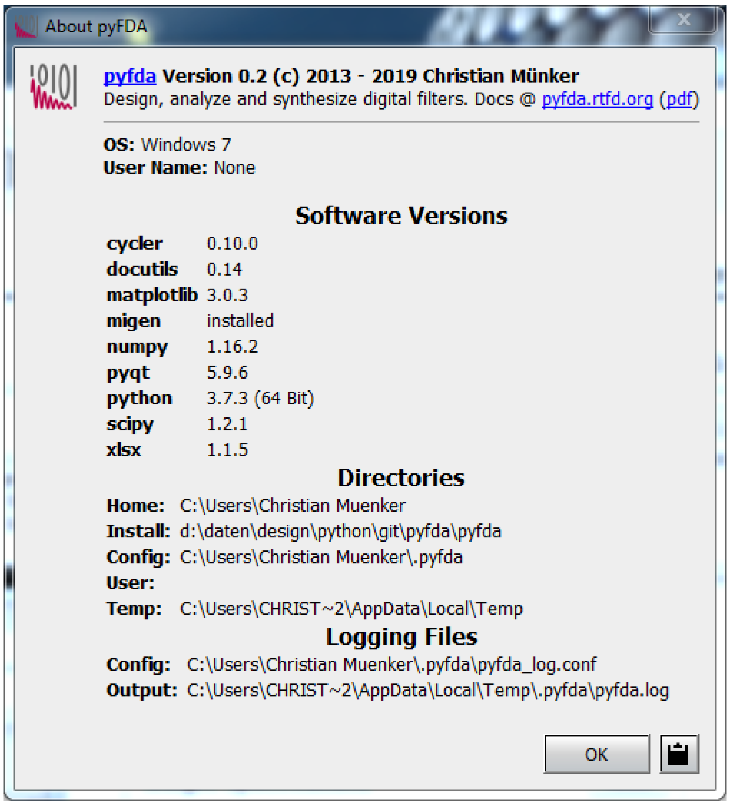


Fig. 1.12: Screenshot of the files tab

Additionally, you can view the python version, paths etc. in the **About** popup window:



pyfda Version 0.2 (c) 2013 - 2019 Christian Münker
Design, analyze and synthesize digital filters. Docs @ pyfda.rtfd.org (pdf)

OS: Windows 7
User Name: None

Software Versions

cycler	0.10.0
docutils	0.14
matplotlib	3.0.3
migen	installed
numpy	1.16.2
pyqt	5.9.6
python	3.7.3 (64 Bit)
scipy	1.2.1
xlsx	1.1.5

Directories

Home:	C:\Users\Christian Muenker
Install:	d:\daten\design\python\git\pyfda\pyfda
Config:	C:\Users\Christian Muenker\.pyfda
User:	
Temp:	C:\Users\CHRIST~2\AppData\Local\Temp

Logging Files

Config:	C:\Users\Christian Muenker\.pyfda\pyfda_log.conf
Output:	C:\Users\CHRIST~2\AppData\Local\Temp\.pyfda\pyfda.log

Fig. 1.13: Screenshot of the “About pyfda” popup window

1.5.1 Development

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

1.6 Fixpoint Specs

1.6.1 Overview

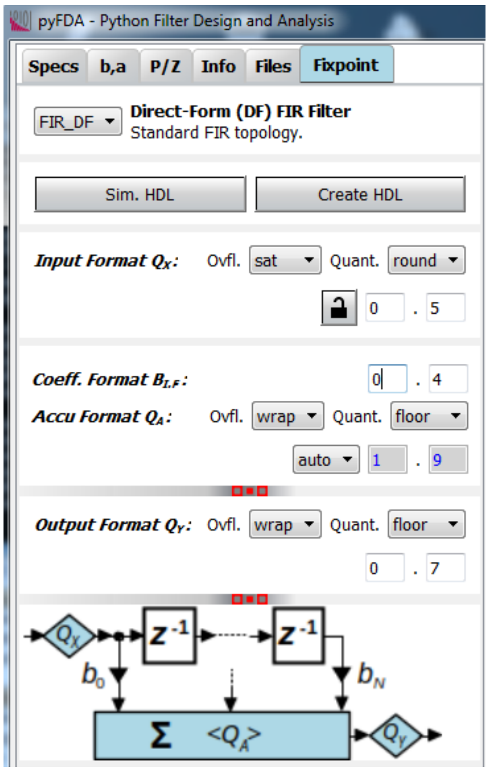
The **Fixpoint** tab (Fig. 1.14) provides options for generating and simulating discrete-time filters that can be implemented in hardware. Hardware implementations for discrete-time filters usually imply fixpoint arithmetics but this could change in the future as floating point arithmetics can be implemented on FPGAs using dedicated floating point units (FPUs).

Order and the coefficients have been calculated by a filter design algorithm from the *pyfda.filter_designs* 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.



FIR_DF **Direct-Form (DF) FIR Filter**
Standard FIR topology.

Input Format Q_X: Ovfl. sat Quant. round
0 . 5

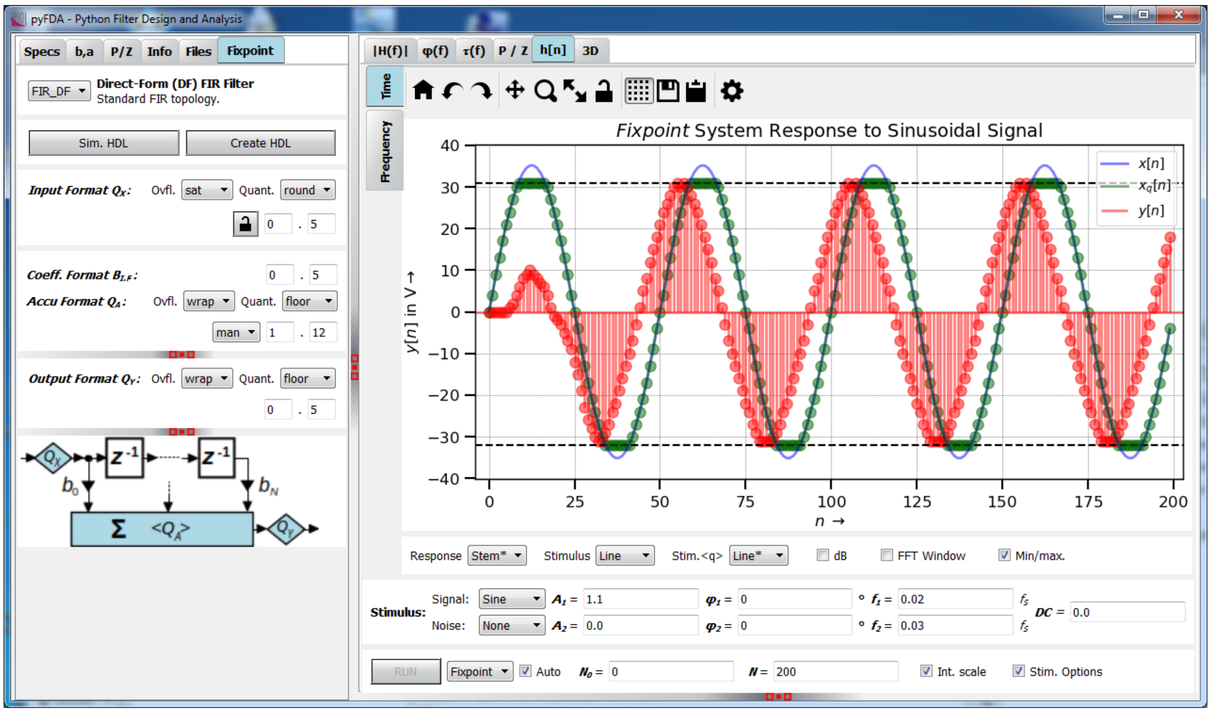
Coeff. Format $B_{I,F}$: 0 . 4

Accu Format Q_A: Ovfl. wrap Quant. floor
auto 1 . 9

Output Format Q_Y: Ovfl. wrap Quant. floor
0 . 7

Fig. 1.14: Fixpoint parameter entry widget

Typical simulation results are shown in Fig. 1.15 (time domain) and Fig. 1.16 (frequency domain).



FIR_DF Direct-Form (DF) FIR Filter
Standard FIR topology.

Input Format Q_X: Ovfl. sat Quant. round 0 . 5

Coeff. Format B_I.F: 0 . 5

Accu Format Q_A: Ovfl. wrap Quant. floor man 1 . 12

Output Format Q_Y: Ovfl. wrap Quant. floor 0 . 5

Response Stem* Stimulus Line Stim.<q> Line* ☐ dB ☐ FFT Window ☑ Min/max.

Stimulus: Signal: Sine A_1 = 1.1 φ_1 = 0 ° f_1 = 0.02 f_S DC = 0.0
Noise: None A_2 = 0.0 φ_2 = 0 ° f_2 = 0.03 f_S

Fixpoint ☑ Auto N_0 = 0 N = 200 ☑ Int. scale ☑ Stim. Options

Fig. 1.15: Fixpoint simulation results (time 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:

1.6.2 Configuration

The configuration file `pyfda.conf` lists the fixpoint classes to be used, e.g. `DF1` and `DF2`. `pyfda.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.

1.6.3 Development

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

1.7 Plot H(f)

Fig. 1.17 shows a typical view of the **|H(f)|** tab for plotting the magnitude frequency response.

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.

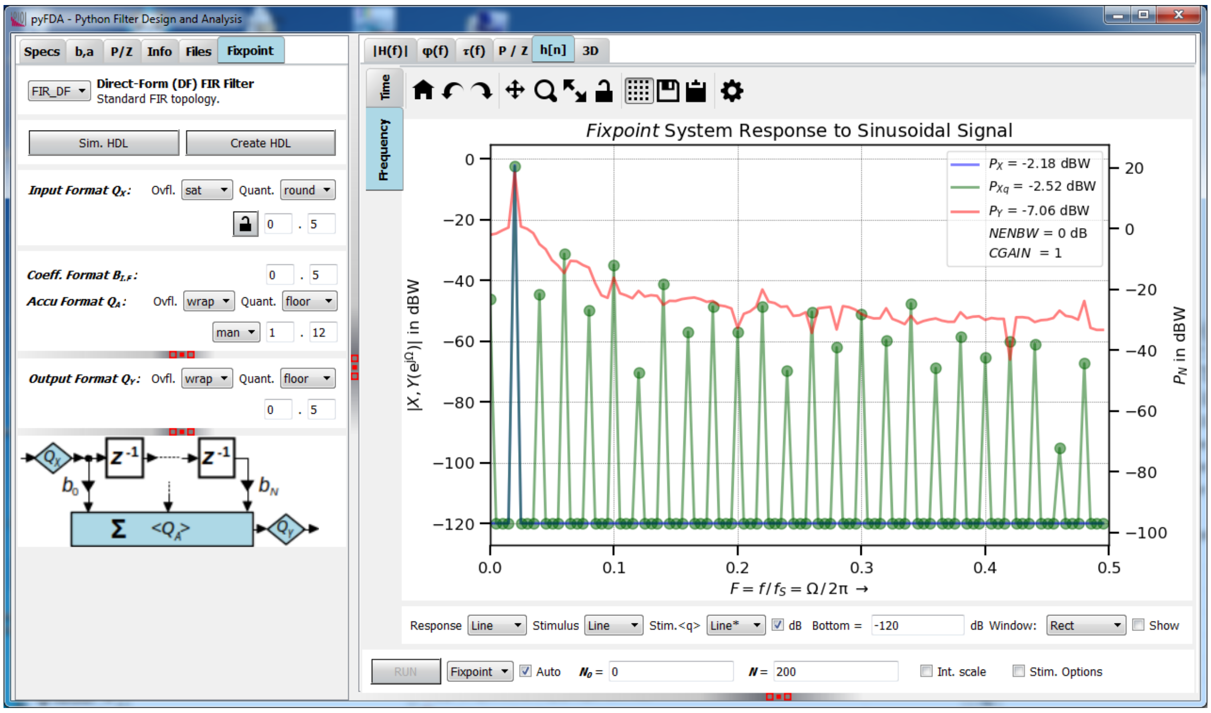

pyFDA - Python Filter Design and Analysis
Specs
b,a
P/Z
Info
Files
Fixpoint
|H(f)|
φ(f)
τ(f)
P / Z
h[n]
3D
Time
Frequency
FIR_DF
Direct-Form (DF) FIR Filter
Standard FIR topology.
Sim. HDL
Create HDL
Input Format Q_X: Ovfl. sat Quant. round 0 . 5
Coeff. Format $B_{I,F}$: 0 . 5
Accu Format Q_A: Ovfl. wrap Quant. floor man 1 . 12
Output Format Q_Y: Ovfl. wrap Quant. floor 0 . 5
Fixpoint System Response to Sinusoidal Signal
P_X = -2.18 dBW
P_{Xq} = -2.52 dBW
P_Y = -7.06 dBW
NENBW = 0 dB
CGAIN = 1
$|X, Y(e^{j\Omega})|$ in dBW
P_N in dBW
$F = f / f_S = \Omega / 2\pi \rightarrow$
Response Line Stimulus Line Stim.<q> Line* dB Bottom = -120 dB Window: Rect Show
RUN Fixpoint Auto N_0 = 0 N = 200 Int. scale Stim. Options

Fig. 1.16: Fixpoint simulation results (frequency domain)

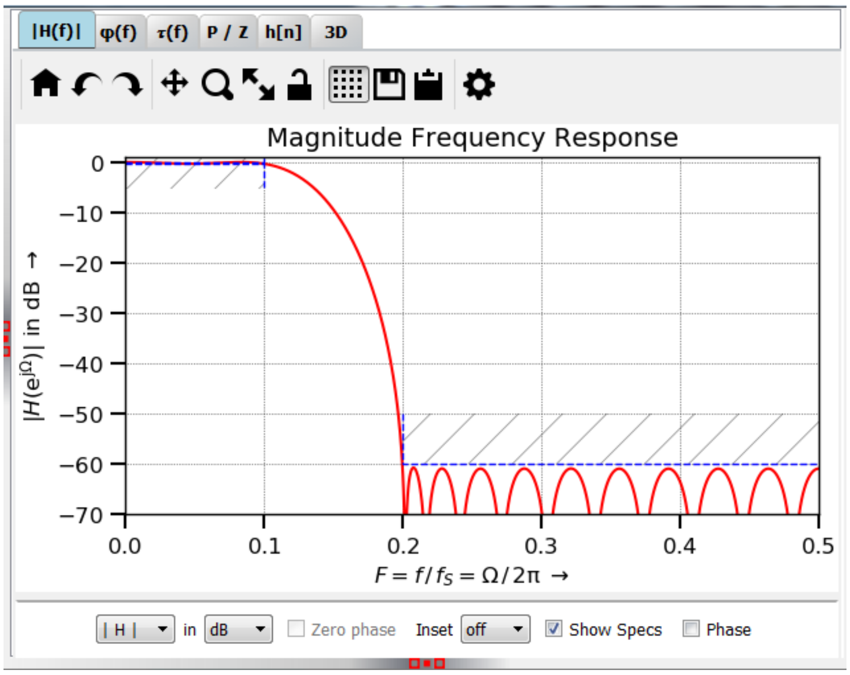

|H(f)|
φ(f)
τ(f)
P / Z
h[n]
3D
Magnitude Frequency Response
$|H(e^{j\Omega})|$ in dB $\rightarrow$
$F = f / f_S = \Omega / 2\pi \rightarrow$
| H | in dB Zero phase Inset off Show Specs Phase

Fig. 1.17: Screenshot of the **|H(f)|** tab

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.

1.7.1 Development

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

1.8 Plot Phi(f)

Fig. 1.18 shows a typical view of the $\varphi(f)$ tab for plotting the phase response of an elliptical filter (IIR).

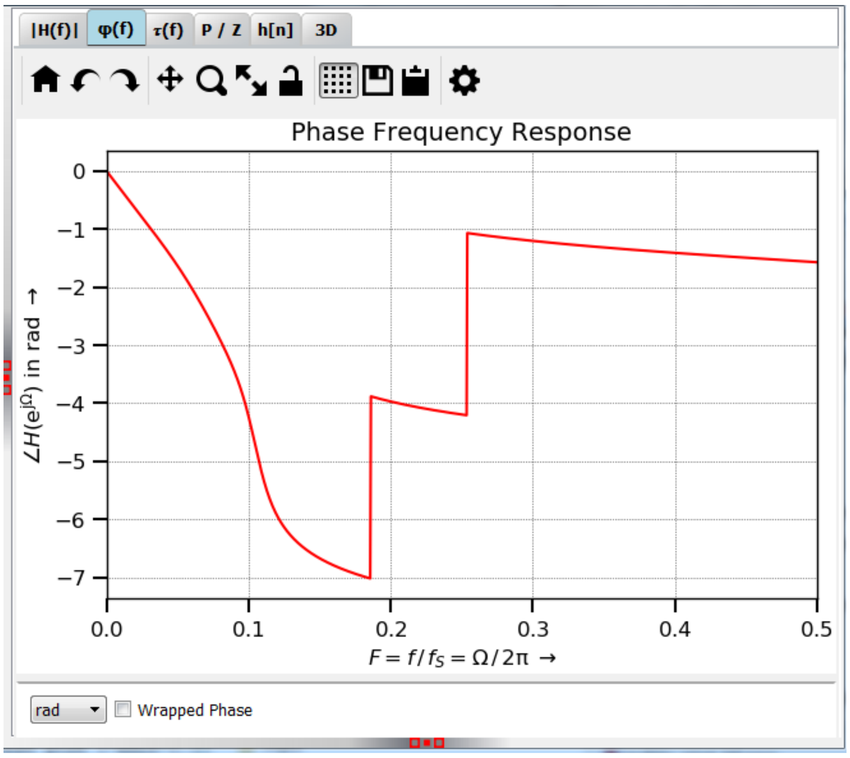


Fig. 1.18: Screenshot of the $\varphi(f)$ tab

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

1.8.1 Development

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

1.9 Plot tau(f)

Fig. 1.19 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.

1.9.1 Development

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

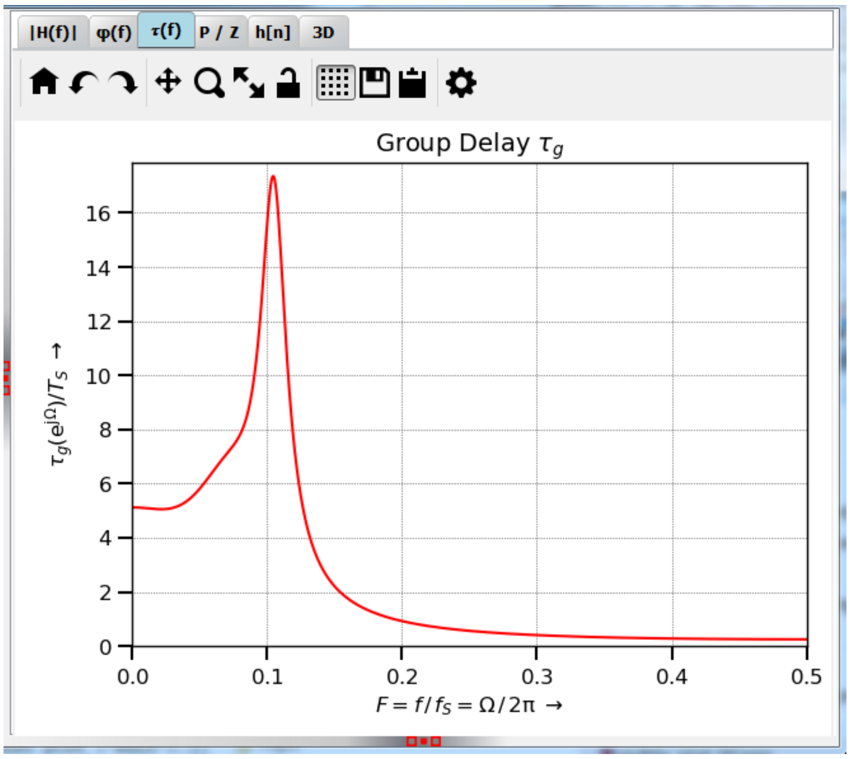


Fig. 1.19: Screenshot of the $\tau(f)$ tab

1.10 Plot P/Z

Fig. 1.20 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. 1.21).

1.10.1 Development

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

1.11 Plot h[n]

Fig. 1.22 shows a typical view of the **h[n]** tab for plotting the transient response and its Fourier transformation, here, an elliptical filter (IIR) is shown.

There are a lot of options in this tab:

Time / Frequency 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. 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**.
- The number of data points **N** is chosen in such a way that an integer number of periods is transformed.
- The FFT window is set (in the Frequency tab) 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.

View What will be plotted and how.

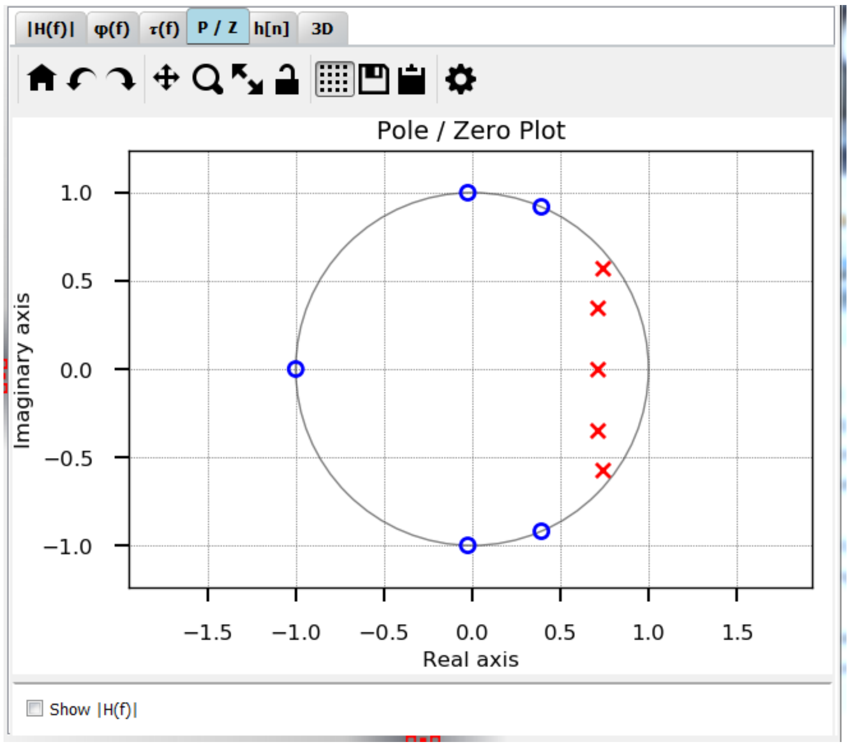


Fig. 1.20: Screenshot of the P/Z tab

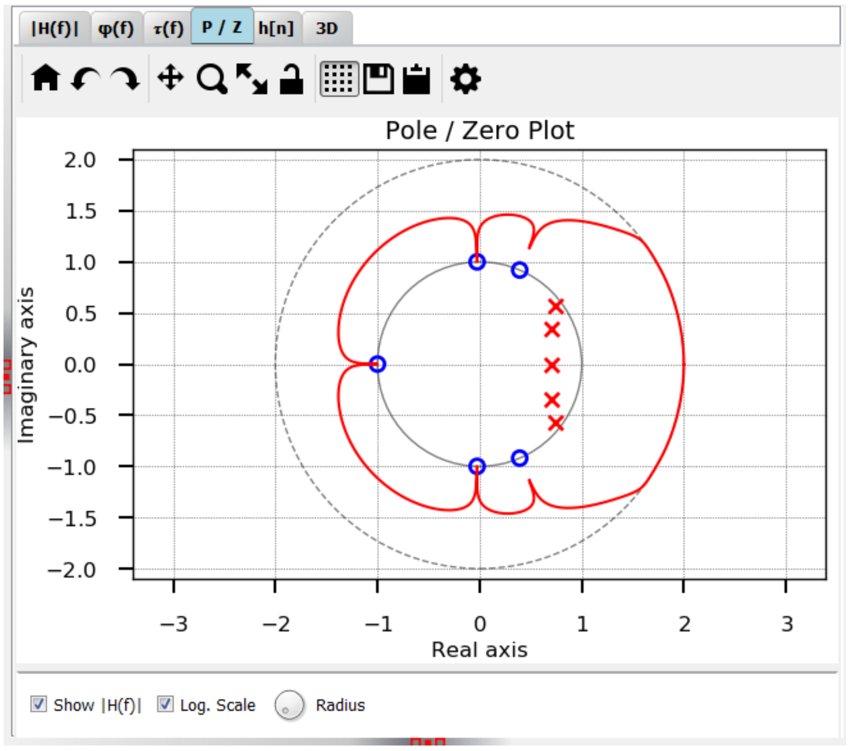
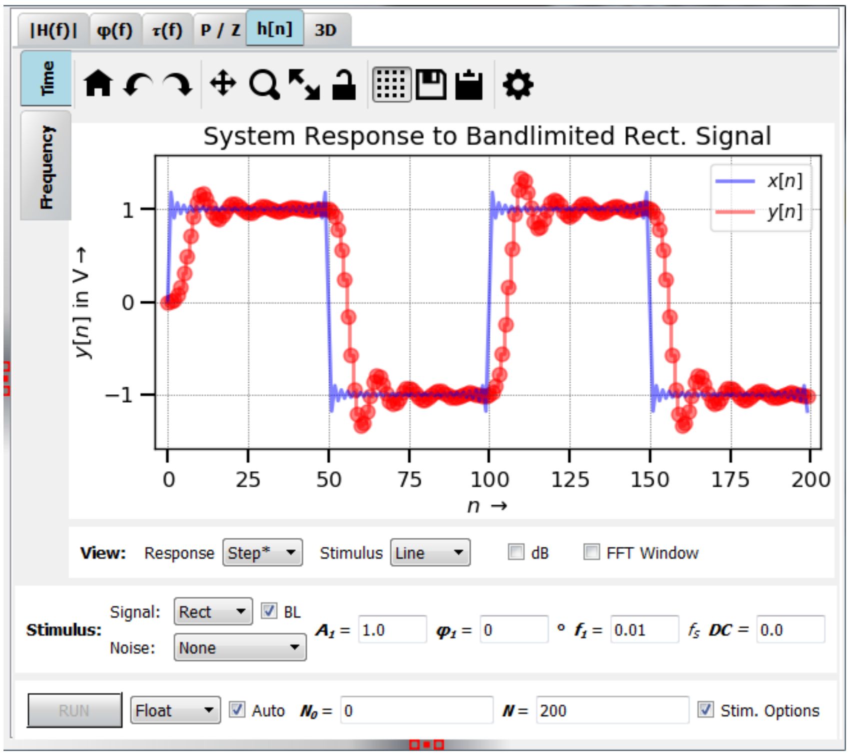


Fig. 1.21: Screenshot of the P/Z tab with overlayed H(f) plot



View: Response Step* Stimulus Line ☐ dB ☐ FFT Window

Stimulus: Signal: Rect ☑ BL A_1 = 1.0 φ_1 = 0 ° f_1 = 0.01 f_S DC = 0.0
Noise: None

Float ☑ Auto N_0 = 0 N = 200 ☑ Stim. Options

Fig. 1.22: Screenshot of the h[n] tab (time domain)

Stimulus 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). The stimulus options can be hidden with the checkbox **Stim. Options**.

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)

The Fourier transform of the transient signal can be viewed in the vertical tab “Frequency” (Fig. 1.23). This is especially important for fixpoint simulations where the frequency response cannot be calculated analytically.

1.11.1 Development

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

1.12 Plot 3D

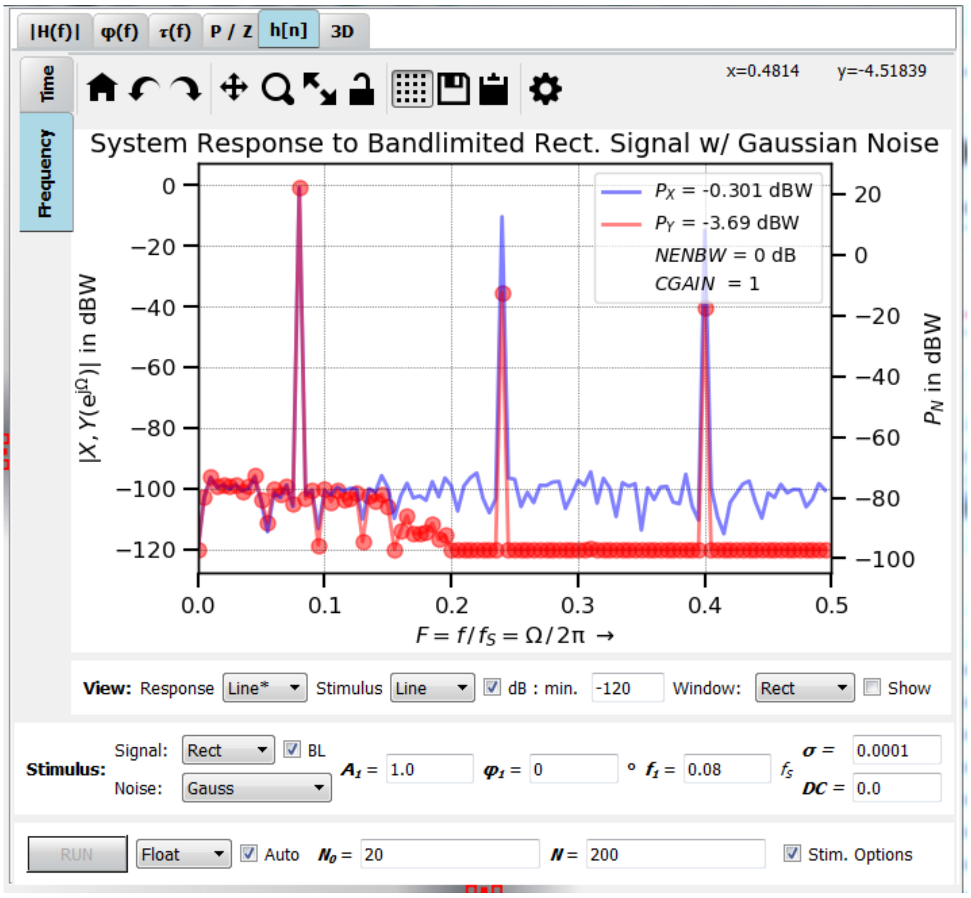
Fig. 1.24 shows a typical view of the **3D** tab for 3D visualizations of the magnitude frequency response and poles / zeros. Fig. 1.24 is a surface plot which looks nice but takes the longest time to compute.

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. 1.25) or a contour plot and switch to a surface plot when you are pleased with scale and view.

1.12.1 Development

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

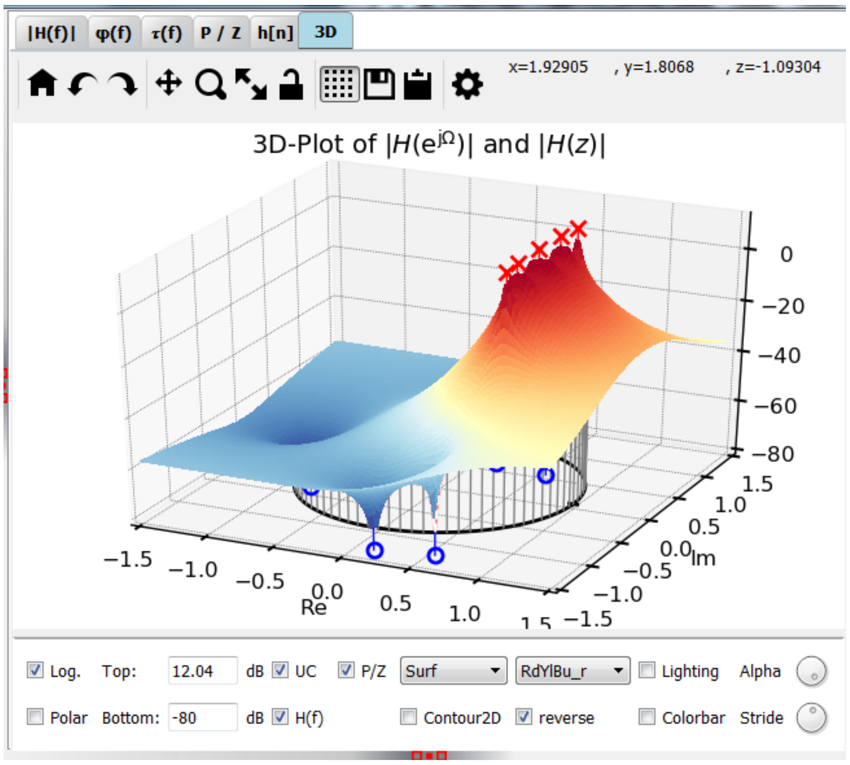


View: Response Line* Stimulus Line ☑ dB : min. -120 Window: Rect ☐ Show

Stimulus: Signal: Rect ☑ BL Noise: Gauss A_1 = 1.0 φ_1 = 0 ° f_1 = 0.08 f_S σ = 0.0001 DC = 0.0

RUN Float ☑ Auto N_0 = 20 N = 200 ☑ Stim. Options

Fig. 1.23: Screenshot of the h[n] tab (frequency domain)


|H(f)| φ(f) τ(f) P / Z h[n] 3D
x=1.92905 , y=1.8068 , z=-1.09304
3D-Plot of |H(e^{jΩ})| and |H(z)|
Re
Im

☑ Log. Top: 12.04 dB ☑ UC ☑ P/Z Surf RdYlBu_r ☐ Lighting Alpha

☐ Polar Bottom: -80 dB ☑ H(f) ☐ Contour2D ☑ reverse ☐ Colorbar Stride

Fig. 1.24: Screenshot of the 3D tab (surface plot)

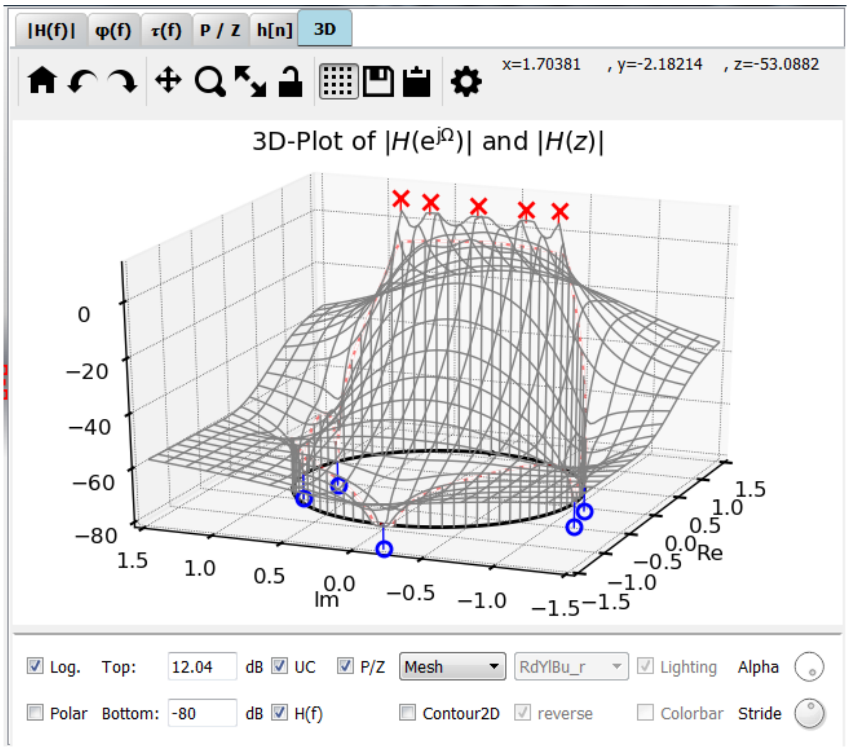


Fig. 1.25: Screenshot of the 3D tab (mesh plot)

1.13 Customization

You can customize pyfda behaviour in some configuration files:

1.13.1 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!):

1.13.2 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:

1.13.3 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 2

Development

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

2.1 Software Organization

The software is organized as shown in the following figure

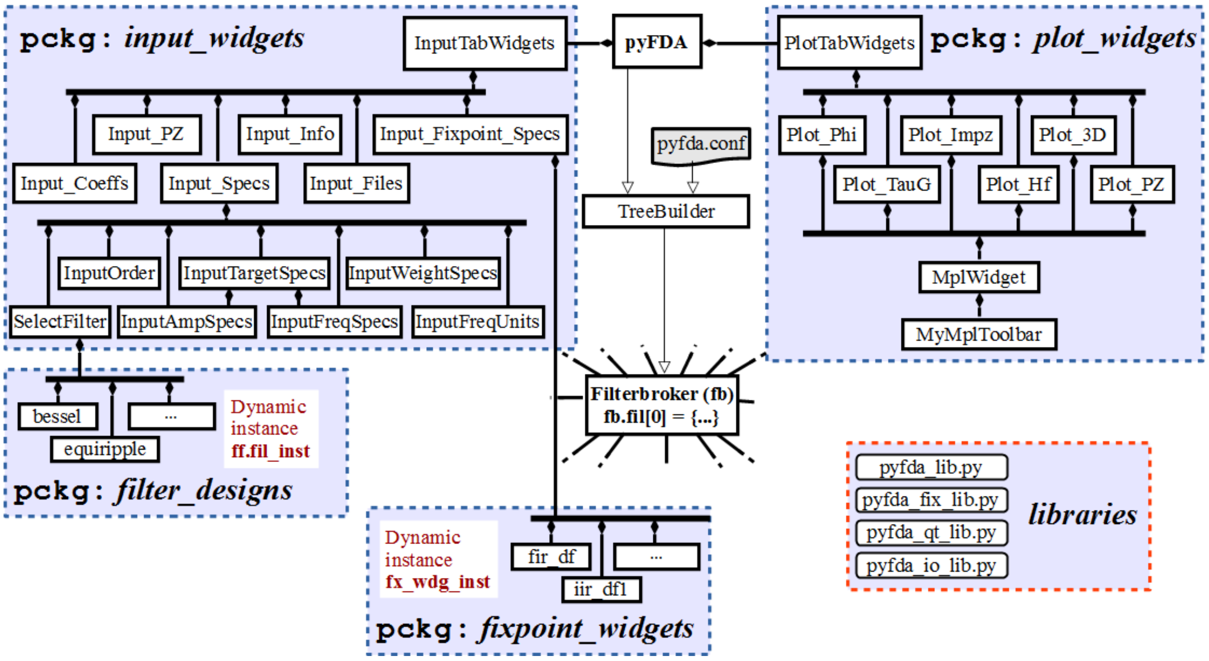


Fig. 2.1: pyfda Organization

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*).

2.2 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

    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['sender'] == __name__:   # only needed when a ``sig_tx signal`` is␣
↪fired
            logger.debug("Infinite loop detected")
            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
```

Information is transmitted via the global `sig_tx` signal:

```
dict_sig = {'sender':__name__, 'fx_sim':'set_results', 'fx_results':self.fx_
↪results}
self.sig_tx.emit(dict_sig)
```

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

'sender' Fully qualified name of the sending widget, usually given as `__name__`. The sender name is needed a.o. to prevent infinite loops which may occur when the rx event is connected to the tx signal.

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

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

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

2.3 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.

2.4 Main Routines

2.4.1 `pyfda.pyfda_dirs`

2.4.2 `pyfda.tree_builder`

2.4.3 `pyfda.pyfda_lib`

2.4.4 `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
```

2.4.5 `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.`**`design_filt_state = 'changed'`**
"ok", "changed", "error", "failed"

Type State of filter design

`pyfda.filterbroker.`**`filter_classes = {'Bessel': {'mod': 'pyfda.filter_designs.bessel',`**

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.

2.4.6 `pyfda.pyfda_io_lib`

2.5 Libraries

2.5.1 `pyfda.pyfda_fix_lib`

2.5.2 `pyfda.pyfda_fix_lib.Fixed`

2.6 Package input_widgets

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

2.6.1 input_tab_widgets

Tabbed container for all input widgets

`class` `pyfda.input_widgets.input_tab_widgets.`**`InputTabWidgets`**(*parent*)

Create a tabbed widget for all input subwidgets in the list `fb.input_widgets_list`. This list is compiled at startup in `pyfda.tree_builder.Tree_Builder`.

`log_rx`(*dict_sig=None*)

Enable *self.sig_rx.connect(self.log_rx)* above for debugging.

2.6.2 input_specs

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*)

Build widget for entering all filter specs

`load_dict`()

Reload all specs/parameters entries from global dict fb.fil[0], using the "load_dict" methods of the individual classes

`process_sig_rx`(*dict_sig=None, 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*)

Flag signals coming in from local subwidgets with *propagate=True* before proceeding with processing in *process_sig_rx*.

`quit_program`()

When <QUIT> button is pressed, send 'quit_program'

`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={}*)

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.

`pyfda.input_widgets.input_specs.`**`classes = {'Input_Specs': 'Specs'}`**

display name

Type Dict containing class name

2.6.3 select_filter

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*)

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.

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

2.6.4 input_coeffs

Widget for displaying and modifying filter coefficients

class `pyfda.input_widgets.input_coeffs.`**`Input_Coeffs`**(*parent*)

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 list 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.

`load_dict`()
: Load all entries from filter dict *fb.fil[0]['ba']* into the coefficient list *self.ba* and 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 register *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

`qdict2ui`()
: Triggered by: - process_sig_rx() if self.fx_specs_changed or dict_sig['fx_sim'] == 'specs_changed' - Set the UI from the quantization dict and update the fixpoint object. When neither WI == 0 nor WF == 0, set the quantization format to general fractional format qfrac.

`quant_coeffs`()
: Quantize selected / all coefficients in self.ba and refresh QTableWidget

`ui2qdict`()
: Triggered by modifying *ui.cmbFormat*, *ui.cmbQOvfl*, *ui.cmbQuant*, *ui.ledWF*, *ui.ledWI* or *ui.ledW* (via *_W_changed()*) or *ui.cmbQFrmt* (via *_set_number_format()*) or *ui.ledScale()* (via *_set_scale()*) or 'qdict2ui()' via *_set_number_format()*

  Read out the settings of the quantization comboboxes.

  - **Store them in the filter dict *fb.fil[0]['fxqc']['QCB']* and as class** attributes in the fixpoint object *self.myQ*
  - Emit a signal with *'view_changed':'q_coeff'*
  - Refresh the table

class `pyfda.input_widgets.input_coeffs.`**`ItemDelegate`**(*parent*)
: 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 *'view_changed':'q_coeff'* by *ui2qdict()*?!

  `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 fixpoint base and number of places

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

    The instance parameter myQ.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*)
When editor has finished, read the updated data from the editor, convert it back to floating point format and store it in both the model (= QTableWidget) and in self.ba. Finally, refresh the table item to display it in the selected format (via *float2frmt()*).

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_coeffs.`**`classes = {'Input_Coeffs': 'b,a'}`**
display name

Type Dict containing class name

2.6.5 input_pz

Widget for displaying and modifying filter Poles and Zeros

class `pyfda.input_widgets.input_pz.`**Input_PZ** (*parent*)
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

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

frmt2cmplx (*text*, *default=0.0*)
Convert format defined by cmbPZFrmt to real or complex

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.

The filter dict is a "normal" numpy float array for z / p / k values The ZPK register *self.zpk* should be a list of float ndarrays to allow for different lengths of z / p / k subarrays while adding / deleting items.?

process_sig_rx(*dict_sig=None*)
Process signals coming from sig_rx

setup_signal_slot()
Setup setup signal-slot connections

class pyfda.input_widgets.input_pz.**ItemDelegate**(*parent*)
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 - to be implemented) 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()

class pyfda.input_widgets.input_pz.**ItemDelegateAnti**(*parent*)
The following methods are subclassed to replace display and editor of the QTableWidget.

displayText() displays number with n_digits without sacrificing precision of the data stored in the table.

displayText(*self*, *Any*, *QLocale*) → str

pyfda.input_widgets.input_pz.**classes = {'Input_PZ': 'P/Z'}**
display name

Type Dict containing class name

2.6.6 input_info

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

class pyfda.input_widgets.input_info.**Input_Info**(*parent*)
Create widget for displaying infos about filter specs and filter design method

load_dict()
update docs and filter performance

process_sig_rx(*dict_sig=None*)
Process signals coming from sig_rx

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

Type Dict containing class name

2.6.7 input_files

Widget for exporting / importing and saving / loading filter data

class pyfda.input_widgets.input_files.**Input_Files**(*parent*)
Create the widget for saving / loading data

about_window()
Display an "About" window with copyright and version infos

file_dump(*fOut*)
Dump file out in custom text format that apply tool can read to know filter coef's

load_filter()
Load filter from zipped binary numpy array or (c)pickled object to filter dictionary and update input and plot widgets

save_filter()
Save filter as zipped binary numpy array or pickle object

pyfda.input_widgets.input_files.**classes = {'Input_Files': 'Files'}**
display name

Type Dict containing class name

2.6.8 input_fixpoint_specs

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 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*.

2.7 Package plot_widgets

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

2.7.1 plot_tab_widgets

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*)

`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.

2.7.2 plot_hf

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`**(*parent*)
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(f)

`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

`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

2.7.3 plot_phi

Widget for plotting phase frequency response phi(f)

class pyfda.plot_widgets.plot_phi.**Plot_Phi**(*parent*)

calc_resp()
(Re-)Calculate the complex frequency response H(f)

draw()
Main entry point: Re-calculate |H(f)| and draw the figure

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

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': 'ϕ(f)'}**
display name

Type Dict containing class name

2.7.4 plot_tau_g

Widget for plotting the group delay

class pyfda.plot_widgets.plot_tau_g.**Plot_tau_g**(*parent*)
Widget for plotting the group delay

calc_tau_g()
(Re-)Calculate the complex frequency response H(f)

init_axes()
Initialize and clear the axes

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

update_view()
Draw the figure with new limits, scale etc without recalculating H(f)

pyfda.plot_widgets.plot_tau_g.**classes = {'Plot_tau_g': 'tau_g'}**
display name

Type Dict containing class name

2.7.5 plot_pz

Widget for plotting poles and zeros

class pyfda.plot_widgets.plot_pz.**Plot_PZ**(*parent*)

draw_Hf(*r=2*)
: Draw the magnitude frequency response around the UC

draw_pz()
: (re)draw P/Z plot

init_axes()
: Initialize and clear the axes (this is only run 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

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='square'*, *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: 'square')*) – Style of the plot, for style == 'square' make scale of x- and y- axis equal.
- **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
- **zlabel** (*plabel,*) – 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

2.7.6 plot_impz

Widget for plotting impulse and general transient responses

class `pyfda.plot_widgets.plot_impz.`**`Plot_Impz`**(*parent*)

Construct a widget for plotting impulse and general transient responses

`calc_auto`(*autorun=None*)

Triggered when checkbox "Autorun" is clicked. Enable or disable the "Run" button depending on the setting of the checkbox. When checkbox is checked (*autorun* == *True* passed via signal- slot connection), automatically run *impz()*.

`calc_fft`()

(Re-)calculate FFTs of stimulus *self.X*, quantized stimulus *self.X_q* and response *self.Y* using the window function *self.ui.win*.

`calc_response`()

(Re-)calculate float filter response *self.y* from stimulus *self.x*.

Split response into imag. and real components *self.y_i* and *self.y_r* and set the flag *self.cmplx*.

`calc_stimulus`()

(Re-)calculate stimulus *self.x*

`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_freq`()

(Re-)draw the frequency domain mplwidget

`draw_response_fx`(*dict_sig=None*)

Get Fixpoint results and plot them

`draw_time`()

(Re-)draw the time domain mplwidget

`fx_select`(*fx=None*)

Select between fixpoint and floating point simulation. Parameter *fx* can be:

- str "Fixpoint" or "Float" when called directly
- int 0 or 1 when triggered by changing the index of combobox *self.ui.cmb_sim_select* (signal-slot-connection)

In both cases, the index of the combobox is updated according to the passed argument. If the index has been changed since last time, *self.needs_calc* is set to True and the run button is set to "changed".

When fixpoint simulation is selected, all corresponding widgets are made visible. *self.fx_sim* is set to True.

If *self.fx_sim* has changed, *self.needs_calc* is set to True.

`impz`(*arg=None*)

Triggered by:

- construct_UI() [Initialization]
- Pressing "Run" button, passing button state as a boolean
- Activating "Autorun" via *self.calc_auto()*
- 'fx_sim' : 'specs_changed'
-

Calculate response and redraw it.

Stimulus and response are only calculated if *self.needs_calc == True*.

plot_fnc (*plt_style*, *ax*, *plt_dict=None*, *bottom=0*)
Return a plot method depending on the parameter *plt_style* (str) and the axis instance *ax*. An optional *plt_dict* is modified in place.

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)

redraw ()
Redraw the currently visible canvas when e.g. the canvas size has changed

`pyfda.plot_widgets.plot_impz.`**`classes = {'Plot_Impz': 'h[n]'}`**
display name

Type Dict containing class name

2.7.7 plot_3d

Widget for plotting |H(z)| in 3D

class `pyfda.plot_widgets.plot_3d.`**`Plot_3D`** (*parent*)
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

`pyfda.plot_widgets.plot_3d.`**`classes = {'Plot_3D': '3D'}`**
display name

Type Dict containing class name

2.8 Package filter_designs

Package providing various algorithms for FIR and IIR filter design.

2.8.1 `pyfda.filter_designs.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_designs.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 = None`

filter type

`info = None`

filter variants

`pyfda.filter_designs.bessel.`**`classes = {'Bessel': 'Bessel'}`**

display name

Type Dict containing class name

2.9 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_designs* 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_wdg_inst` (after the previous one has been destroyed).

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

The widget's methods

- `response = fx_wdg_inst.run_sim(stimulus)`
- `fx_wdg_inst.to_verilog()`

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

2.9.1 input_widgets.input_fixpoint_specs

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

Widget for simulating fixpoint filters and generating Verilog Code

class `pyfda.input_widgets.input_fixpoint_specs.`**`Input_Fixpoint_Specs`**(*parent*)

Create the widget that holds the dynamically loaded fixpoint filter ui

`embed_fixp_img`(*img_file*)

Embed image as self.img_fixp, either in png or svg format

Parameters

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

`eventFilter`(*source*, *event*)

Filter all events generated by monitored QLabel, only resize events are processed here, generating a *sig_resize* signal. All other events are passed on to the next hierarchy level.

`exportHDL`()

Synthesize HDL description of filter

`fx_sim_init`()

Initialize fix-point simulation:

- Update the *fxqc_dict* containing all quantization information
- Setup a filter instance for migen simulation
- Request a stimulus signal

`fx_sim_set_stimulus`(*dict_sig*)

- Get fixpoint stimulus from *dict_sig* in integer format
- Pass it to the fixpoint filter and calculate the fixpoint response

- Send the reponse to the plotting widget

process_sig_rx(*dict_sig=None*)
Process signals coming in via subwidgets and sig_rx

Play PingPong with a stimulus & plot widget:

2. `fx_sim_init()`: Request stimulus by sending 'fx_sim':'get_stimulus'
3. **`fx_sim_set_stimulus()`: Receive stimulus from widget in 'fx_sim':'send_stimulus'** and pass it to HDL object for simulation
4. Send back HDL response to widget via 'fx_sim':'set_response'

resize_img()
Triggered when self (the widget) is resized, consequently the image inside QLabel is resized to completely fill the label while keeping the aspect ratio.

This doesn't really work at the moment.

update_fxqc_dict()
Update the fxqc dictionary before simulation / HDL generation starts.

wdg_dict2ui()
Trigger an update of the fixpoint widget UI when view (i.e. fixpoint coefficient format) or data have been changed outside this class. Additionally, pass the fixpoint quantization widget to update / restore other subwidget settings.

Set the RUN button to "changed".

`pyfda.input_widgets.input_fixpoint_specs.`**`classes = {'Input_Fixpoint_Specs': 'Fixpoint'}`**
display name

Type Dict containing class name

2.9.2 `pyfda.fixpoint_widgets.fir_df`

Widget for specifying the parameters of a direct-form DF1 FIR filter

class `pyfda.fixpoint_widgets.fir_df.`**`FIR_DF_wdg`**(*parent*)
Widget for entering word formats & quantization, also instantiates fixpoint filter class `FilterFIR`.

construct_fixp_filter()
Construct an instance of the fixpoint filter object using the settings from the 'fxqc' quantizer dict

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`*.

run_sim(*stimulus*)
Pass stimuli and run filter simulation, see https://reconfig.io/2018/05/hello_world_migen https://github.com/m-labs/migen/blob/master/examples/sim/fir.py

tb_wdg_stim(*stimulus, inputs, outputs*)
use stimulus list from widget as input to filter

to_verilog()
Convert the migen description to Verilog

ui2dict()
Read out the quantization subwidgets and store their settings in the central fixpoint dictionary *fb.fil[0]['fxqc']* using the keys described below.

Coefficients are quantized with these settings in the subdictionary under the key 'b'.

Additionally, these subdictionaries are returned to the caller (`input_fixpoint_specs`) where they are used to update `fb.fil[0]['fxqc']`

Parameters **`None`** –

Returns

- **fxqc_dict** (*dict*) – containing the following keys and values:
- **- 'QCB'** (*dictionary with coefficients quantization settings*)
- **- 'QA'** (*dictionary with accumulator quantization settings*)
- **- 'b'** (*list of coefficients in integer format*)

`update_accu_settings``()`

Calculate number of extra integer bits needed in the accumulator (bit growth) 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]['fxqc']['QA']*.

`update_q_coeff``(`*dict_sig*`)`

Update coefficient quantization settings and coefficients.

The new values are written to the fixpoint coefficient dict as *fb.fil[0]['fxqc']['QCB']* and *fb.fil[0]['fxqc']['b']*.

`pyfda.fixpoint_widgets.fir_df.`**`classes = {'FIR_DF_wdg': 'FIR_DF'}`**

display name

Type Dict containing widget class name

CHAPTER 3

Literature

References

CHAPTER 4

API documentation

4.1 `pyfda` – Main package

CHAPTER 5

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

Index