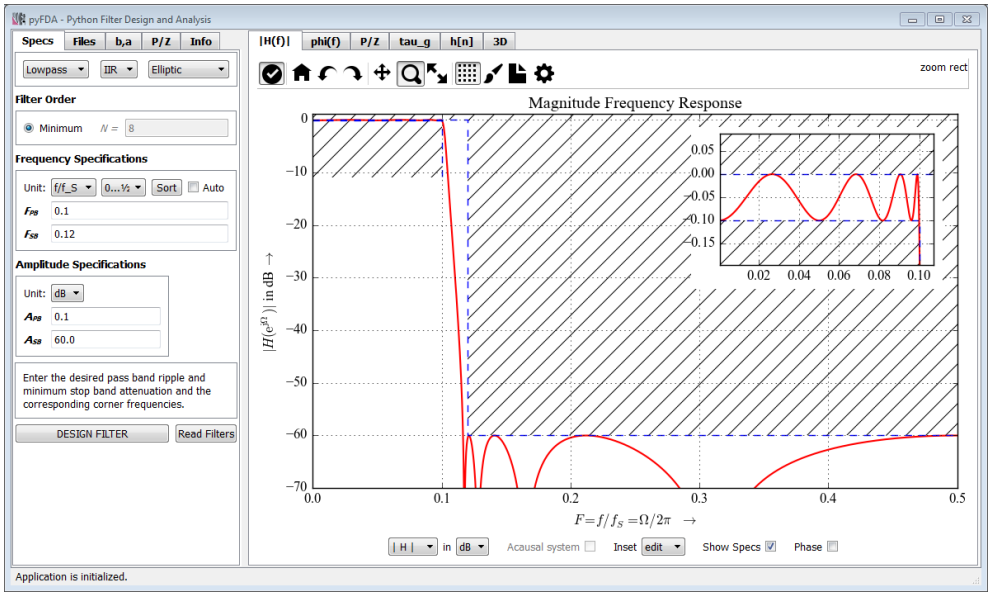
pyFDA: Software Architecture and Filter API

Christian Münker



pyFDA screenshot

3. Juni 2015

Christian.Muenker@hm.edu

Inhaltsverzeichnis

1 Overview

pyFDA has been written to be extensible and modular, easing the addition of own filter and analysis modules.

This has (hopefully) been achieved with two central ideas:

A central global dictionary: The file **`filterbroker.py`** with module-level attributes (dictionaries) is imported by all files that need to store and exchange parameters, filter designs etc. (`https://docs.python.org/3/faq/programming.html#how-do-i-share-global-variables-across-modules`)

Dynamically imported design files: A tree with all available filter design classes and characteristics is built at the start of the program from all files in the `filter_widgets` directory. The actual classes with the design algorithms are imported dynamically when needed, the GUI is adapted according to the parameters defined in each filter design class. Additional widgets can be defined in the design class.

1.1 Class Structure and Hierarchy

The following graphics have been created from the top directory using **`pyreverse`** and some post-processing with LibreOffice (and they are only readable when zoomed in):

```
pyreverse -o pdf -k -ignore=simpleeval.py,input_target_spec.py -p pyFDA .
```

where `-k` only shows the class names (not the attributes and methods) `-p` sets the project name that is the base for the filenames

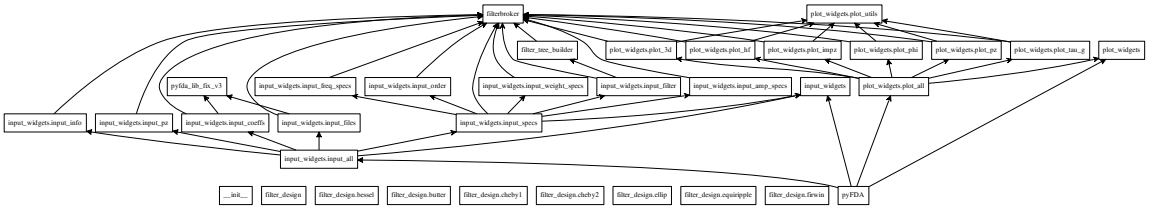


Abbildung 1: Packages in pyFDA

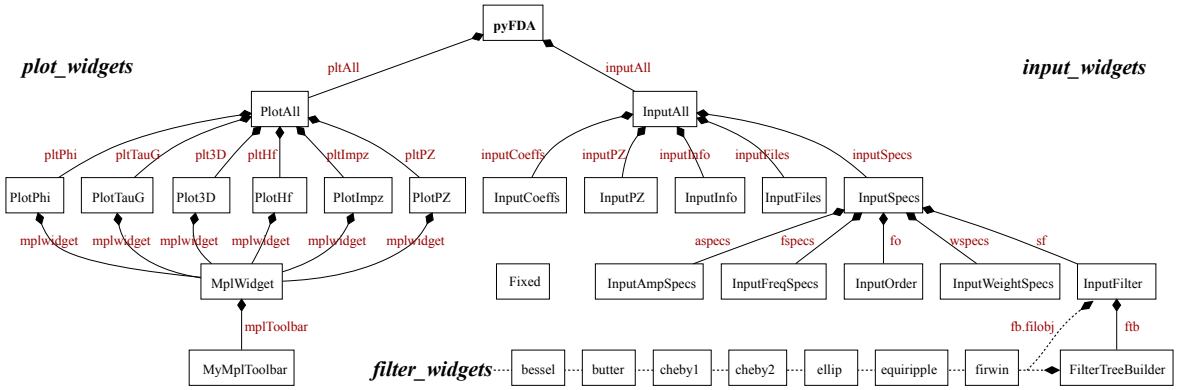


Abbildung 2: Class hierarchy in pyFDA

1.2 Libraries and Testing

No proper testing strategy has been implemented so far (sorry!). However, all files / custom widgets can be run independently to test for syntactic correctness and basic functionality, especially of GUI elements. This has been achieved by the technique described in http://stackoverflow.com/questions/11536764/attempted-relative-import-in-non-package-even-with-init-py :

> „The python import mechanism works relative to the `__name__` of the current file. When you execute a file directly, it doesn't have it's usual name, but has `"__main__"` as its name instead. So relative imports don't work.
>
> You can use `import some_library` directly if you have this above your imports:

```
if __name__ == '__main__' and __package__ is None:
    from os import sys, path
    sys.path.append(path.dirname(path.dirname(path.abspath(__file__))))
```

> You can use the `__package__` attribute to ensure that an executable script files in a package can relatively import other modules from within the same package. The `__package__` attribute tells that file what name it's supposed to have in the package hierarchy. See http://www.python.org/dev/peps/pep-0366/ for details.“

In this project, the following libraries and common files from the top level directory are used:

filterbroker.py : This is the central file used as the data exchange hub where global dictionaries are defined (see section xxx).

pyfda_lib.py : This library contains some DSP and general helper functions.

pyfda_fixlib.py : This library contains the fast fixpoint classes and methods.

simpleeval.py : With the help of this library simple expressions can be evaluated in line edit fields (see section xxx).

> „If you have a script `script.py` in package `pack.subpack`, then setting it's `__package__` to `pack.subpack` will let you do `from ..module import something` to import something from `pack.module`. Note that, as the documentation says, you still have to have the top-level package on the system path. This is already the way things work for imported modules. The only thing `__package__` does is let you use that behavior for directly-executed scripts as well.“

Another option is using the `-m` option of the python interpreter. However, you can't run `python -m core_test` from within the `tests` subdirectory - it has to be from the parent, or you have to add the parent to the path.

1.3 Naming Conventions

The following conventions have been adopted for naming instance names of UI widgets and layouts:

QtGui Widgets	
lblXXX:	QLabels
cmbXXX:	QComboBox
chkXXX:	QCheckBox
butXXX:	QPushButton
tblXXX:	QTableWidget
frmXXX:	QFrame
ledXXX:	QLineedit
tabXXX:	QTabWidget
spcXXX:	QSpacerItem
QtGui Layouts	
layVXXX:	QVBoxLayout
layHXXX:	QHBoxLayout
layGXXX:	QGridLayout

1.4 simpleeval

2 Input Widgets

3 Plot Widgets

4 Communication

When a filter design has been changed, this information is propagated through the hierarchy to various input and plot widgets using Qt's signal-slot mechanism.

Individual widgets generate signals when the filter specs have been changed or a new filter has been designed:

```
1  from PyQt4.QtCore import pyqtSignal
2  ...
3  class MyTopClass(QtGui.QWidget):
4
5            sigFilterChanged  = pyqtSignal()
6                  sigFilterDesigned = pyqtSignal()
7                  ...
8                  def myInputWidget(self):
9                                    self.sigFilterChanged.emit()
10                                   self.sigFilterDesigned.emit()
```

These

5 Filter Design Objects

The structure of a filter file and the attributes and methods that need to be provided are described in this section.

When starting pyFDA, `filter_tree_builder.py` is run, extracting the relevant information from all *.py files found in the subdirectory filter_design and building a hierachical tree in `filter_broker.py`.

When adding new filter objects `my_filter.py` to an pyFDA installation, two things need to be kept in mind:

- copying it to the `filter_widgets` directory
- adding a line with the filename to the list of filter files `Init.txt` in the same directory.

5.1 Who needs you?

A filter design object is instantiated dynamically every time the filter design method is changed in

`input_widgets/input_filter.py` in **`SelectFilter.setDesignMethod()`**

The handle to this object is stored in `filterbroker.py` in `filObj`.

The actual design methods (LP, HP, ...) are called dynamically in `input_widgets/input_specs.py` in **`InputSpecs.startDesignFilt()`**.

An example for a design method is

```
def LPman(self, fil_dict):
        self.get_params(fil_dict)
        self.save(fil_dict, sig.ellip(self.N, self.A_PB, self.A_SB, self.F_PB,
                btype='low', analog = False, output = frmt))
```

with the single parameter `fil_dict`, that supplies the global filter dictionary containing all parameters and the designed filter as well.

The local helper function `get_params()` extracts parameters from the global filter dictionary and scales the parameters if required (as in the case for ellip routines):

```
def get_params(self, fil_dict):
 """
 Translate parameters from the passed dictionary to instance
 parameters, scaling / transforming them if needed.
 """
 self.N     = fil_dict['N']
 self.F_PB  = fil_dict['F_PB'] * 2 # Frequencies are normalized to f_Nyq
```

The local helper function `save()` saves the filter design back to the dictionary and the filter order and corner frequencies if they have been calculated by a minimum order algorithm.

```
def save(self, fil_dict, arg):
  """
  Store results of filter design in the global filter dictionary. Corner
  frequencies calculated for minimum filter order are also stored in the
  dictionary to allow for a smooth manual filter design.
  """
  pyfda_lib.save_fil(fil_dict, arg, frmt, __name__)

  if self.F_PBC is not None: # has corner frequency been calculated?
    fil_dict['N'] = self.N # yes, update filterbroker
    if np.isscalar(self.F_PBC): # HP or LP - a single corner frequency
      fil_dict['F_PBC'] = self.F_PBC / 2.
    else: # BP or BS - two corner frequencies (BP or BS)
      fil_dict['F_PBC'] = self.F_PBC[0] / 2.
      fil_dict['F_PBC2'] = self.F_PBC[1] / 2.
```

The method

5.2 Info strings

All information that is displayed in `input_widget/input_info.py` in a `QtGui.QTextBrowser()` widget is provided in the multi line strings **`self.info`** and **`self.info_doc`** in MarkDown format. They are analyzed and converted to HTML using `publish_string` from `docutils.core`. `self.info` contains self-written information on the filter design method, `self.info_doc` optionally collects python docstrings. See an excerpt from `ellip.py`:

```
1  self.info = """
2  **Elliptic filters**
3
4  (also known as Cauer filters) have a constant ripple :math:‘A_PB‘ resp.
5  :math:‘A_SB‘ in both pass- and stopband(s).
6
7  For the filter design, the order :math:‘N‘, minimum stopband attenuation
8  :math:‘A_SB‘, the passband ripple :math:‘A_PB‘ and
9  the critical frequency / frequencies :math:‘F_PB‘ where the gain drops below
10 :math:‘-A_PB‘ have to be specified.
11
12 **Design routines:**
13
14 ‘‘scipy.signal.ellip()‘‘
15 ‘‘scipy.signal.ellipord()‘‘
16 """
17
18 self.info_doc = []
19 self.info_doc.append('ellip()\n========')
20 self.info_doc.append(sig.ellip.__doc__)
21 self.info_doc.append('ellipord()\n==========')
22 self.info_doc.append(ellipord.__doc__)
```