PyDynamic Documentation

S. Eichstädt, M. Gruber, B. Ludwig, T. Bruns, M. Weber, I. Smith

Apr 22, 2022

GETTING STARTED:

CHAPTER

ONE

PYTHON LIBRARY FOR THE ANALYSIS OF DYNAMIC MEASUREMENTS

1.1 Table of content

1.2 Quickstart

To dive right into it, install PyDynamic and execute one of the examples:

```
(my_PyDynamice_venv) $ pip install PyDynamic
Collecting PyDynamic
[...]
Successfully installed PyDynamic-[...]
(my_PyDynamice_venv) $ python
Python 3.9.7 (default, Aug 31 2021, 13:28:12)
[GCC 11.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> from PyDynamic.examples.uncertainty_for_dft.deconv_DFT import DftDeconvolutionExample
>>> DftDeconvolutionExample()
```

(continues on next page)

(continued from previous page)

```
Propagating uncertainty associated with measurement through DFT
Propagating uncertainty associated with calibration data to real and imag part
Propagating uncertainty through the inverse system
Propagating uncertainty through the low-pass filter
Propagating uncertainty associated with the estimate back to time domain
```

You will see a couple of plots opening up to observe the results. For further information just read on and visit our *tutorial section*.

1.3 Features

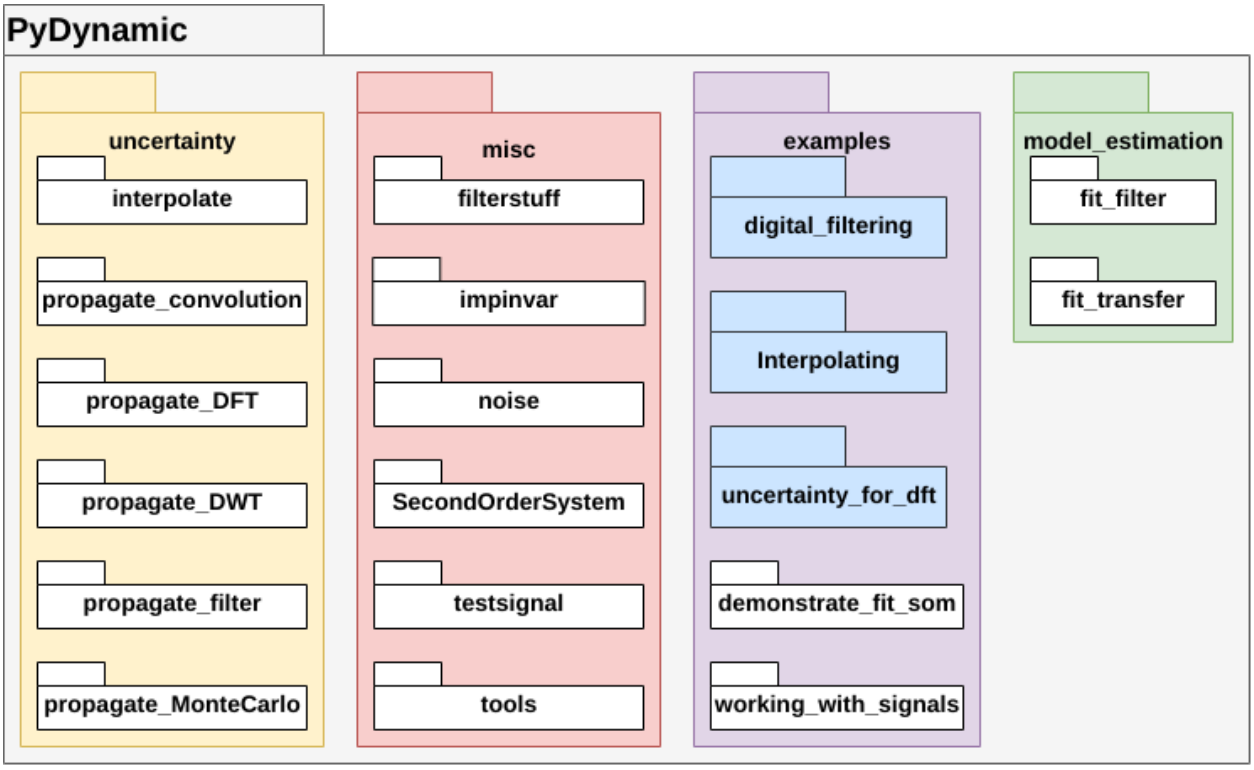
PyDynamic offers propagation of *uncertainties* for

- application of the discrete Fourier transform and its inverse
- filtering with an FIR or IIR filter with uncertain coefficients
- design of a FIR filter as the inverse of a frequency response with uncertain coefficients
- design on an IIR filter as the inverse of a frequency response with uncertain coefficients
- deconvolution in the frequency domain by division
- multiplication in the frequency domain
- transformation from amplitude and phase to a representation by real and imaginary parts
- 1-dimensional interpolation

For the validation of the propagation of uncertainties, the Monte-Carlo method can be applied using a memory-efficient implementation of Monte-Carlo for digital filtering.

1.4 Module diagram

The fundamental structure of PyDynamic is shown in the following figure.



However, imports should generally be possible without explicitly naming all packages and modules in the path, so that for example the following import statements are all equivalent.

```
from PyDynamic.uncertainty.propagate_filter import FIRuncFilter
from PyDynamic.uncertainty import FIRuncFilter
from PyDynamic import FIRuncFilter
```

1.5 Documentation

The documentation for PyDynamic can be found on ReadTheDocs[1]

1.6 Installation

The installation of PyDynamic is as straightforward as the Python ecosystem suggests. Detailed instructions on different options to install PyDynamic you can find in the installation section of the docs[2].

[1] http://pydynamic.readthedocs.io
[2] https://pydynamic.readthedocs.io/en/latest/INSTALL.html

1.7 Contributing

Whenever you are involved with PyDynamic, please respect our Code of Conduct [3]. If you want to contribute back to the project, after reading our Code of Conduct, take a look at our open developments in the project board [4], pull requests [5] and search the issues [6]. If you find something similar to your ideas or troubles, let us know by leaving a comment or remark. If you have something new to tell us, feel free to open a feature request or bug report in the issues. If you want to contribute code or improve our documentation, please check our contribution advices and tips[7].

If you have downloaded this software, we would be very thankful for letting us know. You may, for instance, drop an email to one of the authors [8] (e.g. Sascha Eichstädt[9], Björn Ludwig [10] or Maximilian Gruber [11])

1.8 Examples

We have collected extended material for an easier introduction to PyDynamic in the package *examples*. Detailed assistance on getting started you can find in the corresponding sections of the docs:

- examples[12]
- tutorials[13]

In various Jupyter Notebooks and scripts we demonstrate the use of the provided methods to aid the first steps in PyDynamic. New features are introduced with an example from the beginning if feasible. We are currently moving this supporting collection to an external repository on GitHub. They will be available at github.com/PTB-M4D/PyDynamic_tutorials[14] in the near future.

1.9 Roadmap

1. Implementation of robust measurement (sensor) models
2. Extension to more complex noise and uncertainty models
3. Introducing uncertainty propagation for Kalman filters

For a comprehensive overview of current development activities and upcoming tasks, take a look at the project board[15], issues[16] and pull requests[17].

[3] https://github.com/PTB-M4D/PyDynamic/blob/master/CODE_OF_CONDUCT.md
[4] https://github.com/PTB-M4D/PyDynamic/projects/1
[5] https://github.com/PTB-M4D/PyDynamic/pulls
[6] https://github.com/PTB-M4D/PyDynamic/issues
[7] https://pydynamic.readthedocs.io/en/latest/CONTRIBUTING.html
[8] https://github.com/PTB-M4D/PyDynamic/graphs/contributors
[9] sascha.eichstaedt@ptb.de
[10] bjoern.ludwig@ptb.de
[11] maximilian.gruber@ptb.de
[12] https://pydynamic.readthedocs.io/en/latest/Examples.html
[13] https://pydynamic.readthedocs.io/en/latest/Tutorials.html
[14] https://github.com/PTB-M4D/PyDynamic_tutorials
[15] https://github.com/PTB-M4D/PyDynamic/projects/1
[16] https://github.com/PTB-M4D/PyDynamic/issues
[17] https://github.com/PTB-M4D/PyDynamic/pulls

1.10 Citation

If you publish results obtained with the help of PyDynamic, please use the above linked Zenodo DOI[18] for the code itself or cite

Sascha Eichstädt, Clemens Elster, Ian M. Smith, and Trevor J. Esward *Evaluation of dynamic measurement uncertainty – an open-source software package to bridge theory and practice* **J. Sens. Sens. Syst.**, 6, 97-105, 2017, DOI: 10.5194/jsss-6-97-2017 [19]

1.11 Acknowledgement

Part of this work is developed as part of the Joint Research Project 17IND12 Met4FoF [20] of the European Metrology Programme for Innovation and Research (EMPIR).

This work was part of the Joint Support for Impact project 14SIP08[21] of the European Metrology Programme for Innovation and Research (EMPIR). The EMPIR[22] is jointly funded by the EMPIR participating countries within EURAMET and the European Union.

1.12 Disclaimer

This software is developed at Physikalisch-Technische Bundesanstalt (PTB). The software is made available "as is" free of cost. PTB assumes no responsibility whatsoever for its use by other parties, and makes no guarantees, expressed or implied, about its quality, reliability, safety, suitability or any other characteristic. In no event will PTB be liable for any direct, indirect or consequential damage arising in connection with the use of this software.

1.13 License

PyDynamic is distributed under the LGPLv3 license [23] except for the module `impinvar.py` [24] in the package `misc` [25], which is distributed under the GPLv3 license [26].

[18] https://doi.org/10.5281/zenodo.1489877
[19] https://doi.org/10.5194/jsss-6-97-2017
[20] http://met4fof.eu
[21] https://www.euramet.org/research-innovation/search-research-projects/details/project/standards-and-software-to-maximise-end-user-uptake-of-nmi-calibrations-o
[22] http://msu.euramet.org
[23] https://github.com/PTB-M4D/PyDynamic/blob/master/licence.txt
[24] https://github.com/PTB-M4D/PyDynamic/blob/master/src/PyDynamic/misc/impinvar.py
[25] https://pydynamic.readthedocs.io/en/master/PyDynamic.misc.html
[26] https://github.com/PTB-M4D/PyDynamic/blob/master/src/PyDynamic/misc/impinvar_license.txt

CHAPTER

TWO

INSTALLATION

There is a *quick way* to get started, but we advise setting up a virtual environment and guide through the process in the section *Proper Python setup with virtual environment*

2.1 Quick setup (not recommended)

If you just want to use the software, the easiest way is to run from your system's command line

```
pip install --user PyDynamic
```

This will download the latest version from the Python package repository and copy it into your local folder of third-party libraries. Note that PyDynamic runs with **Python versions 3.7 to 3.10**. Usage in any Python environment on your computer is then possible by

```
import PyDynamic
```

or, for example, for the module containing the Fourier domain uncertainty methods:

```
from PyDynamic.uncertainty import propagate_DFT
```

2.1.1 Updating to the newest version

Updates can then be installed via

```
pip install --user --upgrade PyDynamic
```

2.2 Proper Python setup with virtual environment (recommended)

The setup described above allows the quick and easy use of PyDynamic, but it also has its downsides. When working with Python we should rather always work in so-called virtual environments, in which our project specific dependencies are satisfied without polluting or breaking other projects' dependencies and to avoid breaking all our dependencies in case of an update of our Python distribution.

2.2.1 Set up a virtual environment

If you are not familiar with Python virtual environments [27] you can get the motivation and an insight into the mechanism in the official docs[28].

You have the option to set up PyDynamic using *Anaconda*, if you already have it installed, or use the Python built-in tool `venv`. The commands differ slightly between *Windows* and *Mac/Linux* or if you use *Anaconda* .

Create a `venv` Python environment on Windows

In your Windows PowerShell execute the following to set up a virtual environment in a folder of your choice.

```
PS C:> cd C:\LOCAL\PATH\TO\ENVS
PS C:\LOCAL\PATH\TO\ENVS> py -3 -m venv PyDynamic_venv
PS C:\LOCAL\PATH\TO\ENVS> PyDynamic_venv\Scripts\activate
```

Proceed to *the next step*.

Create a `venv` Python environment on Mac & Linux

In your terminal execute the following to set up a virtual environment in a folder of your choice.

```
$ cd /LOCAL/PATH/TO/ENVS
$ python3 -m venv PyDynamic_venv
$ source PyDynamic_venv/bin/activate
```

Proceed to *the next step*.

Create an Anaconda Python environment

To get started with your present *Anaconda* installation just go to *Anaconda prompt* and execute

```
$ cd /LOCAL/PATH/TO/ENVS
$ conda env create --file /LOCAL/PATH/TO/PyDynamic/requirements/environment.yml
```

That's it!

2.2.2 Install PyDynamic via `pip`

Once you activated your virtual environment, you can install PyDynamic via:

```
pip install PyDynamic
```

```
Collecting PyDynamic
[...]
Successfully installed PyDynamic-[...] [...]
```

That's it!

[27] https://docs.python.org/3/glossary.html#term-virtual-environment
[28] https://docs.python.org/3/tutorial/venv.html

2.2.3 Optional Jupyter Notebook dependencies

If you are familiar with Jupyter Notebooks, you find some examples in the *examples* subfolder of the source code repository. To execute these you need additional dependencies which you get by appending `[examples]` to PyDynamic in all the above, e.g.

```
(PyDynamic_venv) $ pip install PyDynamic[examples]
```

2.2.4 Install known to work dependencies' versions

In case errors arise within PyDynamic, the first thing you can try is installing the known to work configuration of dependencies against which we run our test suite. This you can easily achieve with our version specific requirements files. First you need to install our dependency management package *pip-tools*, then find the Python version you are using with PyDynamic. Finally, you install the provided dependency versions for your specific Python version. This is all done with the following sequence of commands after activating. Change the suffix `-py38` according to the Python version you find after executing `(PyDynamic_venv) $ python --version`:

```
(PyDynamic_venv) $ pip install --upgrade pip-tools
Collecting pip-tools
[...]
Successfully installed pip-tools-5.2.1
(PyDynamic_venv) $ python --version
Python 3.8.8
(PyDynamic_venv) $ pip-sync requirements/dev-requirements-py38.txt requirements/
↪requirements-py38.txt
Collecting [...]
[...]
Successfully installed [...]
(PyDynamic_venv) $
```

CHAPTER

THREE

CHANGELOG

3.1 v2.1.3 (2022-04-19)

3.1.1 Fix

- **test_ARMA:** Increase closeness tolerance (`e35e536`[29])

See all commits in this version[30]

3.2 v2.1.2 (2022-02-07)

3.2.1 Fix

- **tools:** Switch to eigs import from scipy.sparse.linalg for scipy>=1.8.0 (`6618278`[31])

See all commits in this version[32]

3.3 v2.1.1 (2021-12-18)

3.3.1 Fix

- **LSIIR:** Proper init of final_tau (`29f2eef`[33])

[29] https://github.com/PTB-M4D/PyDynamic/commit/e35e536c42966da81930aca05166abe1e04c906a
[30] https://github.com/PTB-M4D/PyDynamic/compare/v2.1.2...v2.1.3
[31] https://github.com/PTB-M4D/PyDynamic/commit/6618278a1f9dda069d433a4a469bbe865a3d54df
[32] https://github.com/PTB-M4D/PyDynamic/compare/v2.1.1...v2.1.2
[33] https://github.com/PTB-M4D/PyDynamic/commit/29f2eefadc05d7cf8affd9727d8afb9b56259737

3.3.2 Documentation

- **Signal:** Introduce Signal class into docs (0da9b9d[34])
- **Python 3.10:** Introduce Python 3.10 to the docs (a20384a[35])

See all commits in this version[36]

3.4 v2.1.0 (2021-12-03)

3.4.1 Feature

- **tools:** Provide convenience functions to prepare input vectors for DFT and filtering (6d15922[37])

3.4.2 Documentation

- **examples:** Add reference to hydrophone paper (3c7880a[38])
- **examples:** Add regularization example inside DFT best practice (75f6dcc[39])

See all commits in this version[40]

3.5 v2.0.0 (2021-11-05)

3.5.1 Feature

- Weighted least-squares IIR or FIR filter fit to freq. resp. or reciprocal with uncertainties (8aca955[41])
- **DWT:** Add wavelet transform with online-support (aed3deb[42])
- **propagate_DWT:** Add prototype of wave_rec_realtime (76ca8df[43])
- **misc:** Add buffer-class for realtime applications (d105de2[44])
- **propagate_DWT:** Return the internal state (31fdb19[45])
- **IIRuncFilter:** Always return internal state (175357a[46])

[34] https://github.com/PTB-M4D/PyDynamic/commit/0da9b9d928688460953ffa3e4b92185c5b45f633
[35] https://github.com/PTB-M4D/PyDynamic/commit/a20384abf20cf576d5da6b6f0a5960b78d046093
[36] https://github.com/PTB-M4D/PyDynamic/compare/v2.1.0...v2.1.1
[37] https://github.com/PTB-M4D/PyDynamic/commit/6d15922d14d934467710cc0466ad1a21b4d6a066
[38] https://github.com/PTB-M4D/PyDynamic/commit/3c7880a977cc35f76bfcf33e60ea3ebc95d56ab5
[39] https://github.com/PTB-M4D/PyDynamic/commit/75f6dccc923d084e68872c0afbfce9cc536c15a0
[40] https://github.com/PTB-M4D/PyDynamic/compare/v2.0.0...v2.1.0
[41] https://github.com/PTB-M4D/PyDynamic/commit/8aca9554165b805aee82d6081db967d7947b5c1e
[42] https://github.com/PTB-M4D/PyDynamic/commit/aed3deb40f2fa85376ba18a1b9c45b1ffd090036
[43] https://github.com/PTB-M4D/PyDynamic/commit/76ca8df9e9f4778a8b6b57cefd28523d167cda89
[44] https://github.com/PTB-M4D/PyDynamic/commit/d105de2228fee1459c38c2a6ee7596a080496bc4
[45] https://github.com/PTB-M4D/PyDynamic/commit/31fdb191ea0d49d9b71f824c6733639c3b16edf6
[46] https://github.com/PTB-M4D/PyDynamic/commit/175357a564c7e00a2aa4341eb1e3346c3fe774c0

3.5.2 Fix

- **propagate_filter:** Avoid floating point issues with small negative uncertainties via clipping (bbe9d13[47])
- **FIRuncFilter:** Actually perform shifting for fast computation cases (14345c6[48])
- **FIRuncFilter:** Output shifting returns expected covariance matrix (3c6ca41[49])
- **propagate_DWT:** Adjust renamed function (7978c26[50])
- **imports:** Make DWT-methods available from top-level (85165a6[51])
- **examples:** Remove unsed imports (f32d975[52])
- **examples:** Remove unused buffer from speed-comparison-filter (d02a9f3[53])
- **IIRuncFilter:** Take sqrt(Ux[0]) in case of kind=corr (38bdb99[54])
- **IIRuncFilter:** Warn user if Ux is float but kind not diag (47e01f5[55])
- **IIRuncFilter:** Use None as default for Uab (0e7fd18[56])
- **propagate_filter:** Refine error messages (038ef72[57])
- **example:** Remove validate_FIRuncFilter (76d09a2[58])
- **example:** Adjust validate_FIRuncFilter (7469c91[59])
- **examples:** Review validate_DWT_monte_carlo- sort imports- add docstring- fix renamed functions- fix changed signatures\n- apply black (0199dfe[60])
- **example:** Enhance realtime_dwt (14f54fd[61])
- **model_estimation:** Introduce new package *model_estimation* in preparation of deprecations (627575c[62])
- **IIRuncFilter:** Match default kind with FIRuncFilter (0a0fdfe[63])
- **propagate_filter:** Fix correlated uncertainty formula (70e9375[64])
- **FIRuncFilter:** Set internal state of lfilter (1f60e76[65])
- **validate_DWT_monte_carlo:** Adjust return values of dwt/idwt (4dd601b[66])
- **test_decomposition_realtime:** Adjust concat statement (947ed21[67])
- **wave_dec_realtime:** Missing argument in np.empty (583a7b5[68])

[47] https://github.com/PTB-M4D/PyDynamic/commit/bbe9d1334c6ec6c51489b8cb1a19c167ca8c7fa6
[48] https://github.com/PTB-M4D/PyDynamic/commit/14345c62c848a97df2f791fb99ee2162d17a9f7d
[49] https://github.com/PTB-M4D/PyDynamic/commit/3c6ca4172b1362dd9cd3b0e91ac374dd5f458f3f
[50] https://github.com/PTB-M4D/PyDynamic/commit/7978c26cc0dac9dece2f5518d47db6e180fd768a
[51] https://github.com/PTB-M4D/PyDynamic/commit/85165a6d034a8ae8ae858d6b791d48dd0e899692
[52] https://github.com/PTB-M4D/PyDynamic/commit/f32d975e23be75fa3387ba861e23ea6433472987
[53] https://github.com/PTB-M4D/PyDynamic/commit/d02a9f36ea67088baeeff0880c468768d38a70d6
[54] https://github.com/PTB-M4D/PyDynamic/commit/38bdb996b7d5fa427097be48ace01ac9896fdccd
[55] https://github.com/PTB-M4D/PyDynamic/commit/47e01f544b7497dee40c51bbc09fe7310066b624
[56] https://github.com/PTB-M4D/PyDynamic/commit/0e7fd18dd94d4610108976aee14322c7feb18531
[57] https://github.com/PTB-M4D/PyDynamic/commit/038ef72e4c38f268cbe4dfe645a69743499a4b49
[58] https://github.com/PTB-M4D/PyDynamic/commit/76d09a25c9ec4d1e12f592c2bbd802e819838cdb
[59] https://github.com/PTB-M4D/PyDynamic/commit/7469c913bd0f104fc00b9ddf38ff5ac01ff35e98
[60] https://github.com/PTB-M4D/PyDynamic/commit/0199dfe02ff8ae322e6304fa955e790739203d63
[61] https://github.com/PTB-M4D/PyDynamic/commit/14f54fd7eb72c5fabb3bd8d63f16a02ea8b2be73
[62] https://github.com/PTB-M4D/PyDynamic/commit/627575caf1e066e466b668f81ce019c5a4b59f7f
[63] https://github.com/PTB-M4D/PyDynamic/commit/0a0fdfe7e9bd06f499dd5f5059459c370d7d59e4
[64] https://github.com/PTB-M4D/PyDynamic/commit/70e9375992b6b85524ed80ac99ee0a7d94b4bec6
[65] https://github.com/PTB-M4D/PyDynamic/commit/1f60e76f03f808e7d20821c13a2a2b337ab6d084
[66] https://github.com/PTB-M4D/PyDynamic/commit/4dd601b4732260a9f621cc725e74c5ea3a085991
[67] https://github.com/PTB-M4D/PyDynamic/commit/947ed211041c3a12fbf060a72d14f20274145423
[68] https://github.com/PTB-M4D/PyDynamic/commit/583a7b591b3e32c14038ae267aad7a90fe6ea2fe

- **idwt:** Remove leftover from debugging (7cca19d[69])
- **idwt:** Adjust boundary conditions (b7788ff[70])
- **test_dwt:** Remove too many unpack values (4b52d67[71])

3.5.3 Breaking

- Combine *deconvolution.fit_filter* and *identification.fit_filter* into *model_estimation.fit_filter* and provide access to all functionality via according parameter sets for *model_estimation.fit_filter.LSFIR* and *model_estimation.fit_filter.LSIIR*. (8aca955[72])
- Rename input parameters t and t_new to x and x_new in *PyDynamic.uncertainty.interpolate* (918f5bb[73])
- Rename `fit_sos()` to `fit_som()` because it actually handles second-order models and not second-order-systems. (bc42fd1[74])

3.5.4 Documentation

- **README:** Restyle README and generally improve structure of docs (1409856[75])
- Fix some formatting issues resulting in strange looking or misleading info on ReadTheDocs (ab30b4b[76])
- **Design of a digital deconvolution filter (FIR type):** Introduce one more example notebook (c51b98b[77])
- **uncertainties:** Integrate DWT-module to docs (fb7a99a[78])
- **propagate_DWT:** Enhance/prettify docstrings (1fcfc43[79])
- **IIRuncFilter:** Minor adjustments to docstring (475a754[80])
- **propagate_DWT:** Extend module description (a007797[81])
- **README:** Document in README optional dependency installation for Jupyter Notebooks (a59f98d[82])
- **propagate_filter:** Fix IIRuncFilter docstring (e2bd085[83])
- **propagate_filter:** Mention FIR and IIR difference (f6dcd4e[84])
- **examples:** Move validation script to examples (abc0fd9[85])
- **examples:** Include errorbars instead of lines (76d978e[86])
- **examples:** Use latex font and adjust naming (57f4c83[87])

[69] https://github.com/PTB-M4D/PyDynamic/commit/7cca19d53919bb771267f8868535de942fe72db2
[70] https://github.com/PTB-M4D/PyDynamic/commit/b7788ffc7d6d713eeb7c995fd0f83f2ae78d3f23
[71] https://github.com/PTB-M4D/PyDynamic/commit/4b52d6750c307cabeebbcb0c70726534c0a73c00
[72] https://github.com/PTB-M4D/PyDynamic/commit/8aca9554165b805aee82d6081db967d7947b5c1e
[73] https://github.com/PTB-M4D/PyDynamic/commit/918f5bb4ecf6239adc2f8e996689b0cef9ca8d9d
[74] https://github.com/PTB-M4D/PyDynamic/commit/bc42fd142f823feff3c15058ee252b0998541739
[75] https://github.com/PTB-M4D/PyDynamic/commit/1409856acf2b576e28f6e2993de58c459baa6243
[76] https://github.com/PTB-M4D/PyDynamic/commit/ab30b4bd355b8a176eae022da7cc4f4a826da924
[77] https://github.com/PTB-M4D/PyDynamic/commit/c51b98b576f1777f3915c995aa32f0c26fad0431
[78] https://github.com/PTB-M4D/PyDynamic/commit/fb7a99a707758862b696b9a018e5aaba21c08df1
[79] https://github.com/PTB-M4D/PyDynamic/commit/1fcfc439d97ad554842ed2b019af1d456c391e98
[80] https://github.com/PTB-M4D/PyDynamic/commit/475a75453a59990997c79bdf5930757345b9ffe0
[81] https://github.com/PTB-M4D/PyDynamic/commit/a007797cb2669b731c31bd0785eb0d817aa73bb3
[82] https://github.com/PTB-M4D/PyDynamic/commit/a59f98dec11131b19679beaa44366fea16629c9f
[83] https://github.com/PTB-M4D/PyDynamic/commit/e2bd085121a3747aa407edec66c9b7d819f05161
[84] https://github.com/PTB-M4D/PyDynamic/commit/f6dcd4efabbc58ad258616bced1ce0369863b751
[85] https://github.com/PTB-M4D/PyDynamic/commit/abc0fd98f32e00cfb9df0c786fce5f36b98f2798
[86] https://github.com/PTB-M4D/PyDynamic/commit/76d978eaa5bf8b02e0ac00595b50856f8cc5983d
[87] https://github.com/PTB-M4D/PyDynamic/commit/57f4c83b6b42e831d978c8eb21a6a27deca8fa24

- **examples:** Higher uncertainty, tight layout (58401c3[88])
- **examples:** Refining plot output (3d0e64c[89])
- **examples:** Calculate and highlight 10 biggest coeffs (7e754af[90])
- **examples:** Change order and appearance of plots (3138d51[91])
- **examples:** Realtime_dwt with multi-level-decomposition (6d48ba7[92])
- **examples:** Plot detail coefficients (53ca6f5[93])
- **examples:** Apply dwt to signal (93edef5[94])
- **examples:** Add script to examine realtime Wavelet (eaf13e7[95])
- **IIRuncFilter:** Fix wrong formula reference (0999569[96])
- **propagate_filter:** Adjust return values of IIRuncFilter (02a2350[97])
- **IIRuncFilter:** Describe non-use of b, a, Uab if state (0889475[98])
- **propagate_filter:** Enhance specification of "kind" (ee2062d[99])

See all commits in this version[100]

3.6 v1.11.1 (2021-10-20)

3.6.1 Fix

- **IIRuncFilter:** Introduce a missing import of scipy.signal.tf2ss (17fe115[101])

3.7 v1.11.0 (2021-10-15)

3.7.1 Feature

- **plot_vectors_and_covariances_comparison:** Introduce function to conveniently compare vectors (e2b3b0c[102])
- **normalize_vector_or_matrix:** Make normalize_vector_or_matrix() publicly available (52b1256[103])
- **is_2d_square_matrix:** Make is_2d_square_matrix() publicly available (e303e6b[104])

[88] https://github.com/PTB-M4D/PyDynamic/commit/58401c3c4e981ab002156c8f5fefea78546a9e36
[89] https://github.com/PTB-M4D/PyDynamic/commit/3d0e64ca5f34590583b3f1dfb5956be11b27e730
[90] https://github.com/PTB-M4D/PyDynamic/commit/7e754afc1f6853fa1b0287716a784b3cec7f9f74
[91] https://github.com/PTB-M4D/PyDynamic/commit/3138d517143d4855daa83e7f4e7b51ebf345b3a1
[92] https://github.com/PTB-M4D/PyDynamic/commit/6d48ba74d3cc0105dd8272a112b0dfdadf3dcea7
[93] https://github.com/PTB-M4D/PyDynamic/commit/53ca6f5402bfe01152ec954ff1304560978301d0
[94] https://github.com/PTB-M4D/PyDynamic/commit/93edef5f3eaaf31e48e249d5d7b90349b38c1359
[95] https://github.com/PTB-M4D/PyDynamic/commit/eaf13e78bcb28169c9ec95adf8707db9f7a59a02
[96] https://github.com/PTB-M4D/PyDynamic/commit/0999569d6bb023ddd34cba12686b21637e374b93
[97] https://github.com/PTB-M4D/PyDynamic/commit/02a235000c80b638a30dfb077b87d78492117a05
[98] https://github.com/PTB-M4D/PyDynamic/commit/088947508297618196 8c3d02434919bfce2ce10f
[99] https://github.com/PTB-M4D/PyDynamic/commit/ee2062dc4687208175ade5028727b6ec14344d75
[100] https://github.com/PTB-M4D/PyDynamic/compare/v1.11.1...v2.0.0
[101] https://github.com/PTB-M4D/PyDynamic/commit/17fe115301e048d68a9fd27087cf9739fd3b5bd1
[102] https://github.com/PTB-M4D/PyDynamic/commit/e2b3b0c530fe3970919beeec14c96587a86653af
[103] https://github.com/PTB-M4D/PyDynamic/commit/52b125679472b227612951e869958e1e695dbcfe
[104] https://github.com/PTB-M4D/PyDynamic/commit/e303e6b920c96010e417dec9013e3b6f639466c8

3.7.2 Fix

- **version:** Reintroduce **version** variable into PyDynamic/**init**.py (0349b09[105])

3.7.3 Documentation

- **CONTRIBUTING:** Mention necessity of installing PyDynamic itself for testing (1571585[106])

3.8 v1.10.0 (2021-09-28)

3.8.1 Feature

- **propagate_DFT:** Make some helpers to check for shapes of inputs publicly available (dc97b3f[107])

3.8.2 Fix

- **fit_som:** Apply minor changes to fit_som example to make it executable again (0157fc7[108])

3.9 v1.9.2 (2021-09-21)

3.9.1 Fix

- **PyPI package:** Include examples in packag and make sure all tests pass with delivered version (f8326d5[109])

3.10 v1.9.1 (2021-09-15)

3.10.1 Fix

- **DFT_deconv:** Replace the imaginary part of H by Y's imaginary part in one of the equations (a4252dd[110])

[105] https://github.com/PTB-M4D/PyDynamic/commit/0349b09eeff34a2144a8dc86a9bccc6aed0387cd
[106] https://github.com/PTB-M4D/PyDynamic/commit/157158536c1ed7f57f34ba33578ee1cb60716b21
[107] https://github.com/PTB-M4D/PyDynamic/commit/dc97b3faddcb854670a9a5f9dcd4ff38efb575d9
[108] https://github.com/PTB-M4D/PyDynamic/commit/0157fc7f775851359efc6d1a97ca87c9325108be
[109] https://github.com/PTB-M4D/PyDynamic/commit/f8326d52d9dc289b5069993658500670013 20a3e
[110] https://github.com/PTB-M4D/PyDynamic/commit/a4252ddd850908467933d7de41f1da99d57d3ea1

3.10.2 Documentation

- Introduce Python 3.9 to the docs and actually provide requirements*.txt files (19dcef2[111])

3.11 v1.9.0 (2021-05-11)

3.11.1 Feature

- **interp1d_unc:** Add cubic bspline interpolation-kind (f0c6d19[112])

3.11.2 Documentation

- **interp1d_unc:** Add example for kind = "cubic" (8c2ce38[113])

3.12 v1.8.0 (2021-04-28)

3.12.1 Feature

- **propagate_convolution:** Convolution with full covariance propagation (299165e[114])

3.12.2 Documentation

- **propagate_convolution:** Add module description (ffafa00[115])
- **DFT_deconv:** Improve wording of docstring for return value (e866aa4[116])

3.13 v1.7.0 (2021-02-16)

3.13.1 Feature

- **FIRuncFilter:** Add FIR filter with full covariance support (b937a8b[117])

[111] https://github.com/PTB-M4D/PyDynamic/commit/19dcef2f5b1d0516dc9ebf462d5115a5554c8cec
[112] https://github.com/PTB-PSt1/PyDynamic/commit/f0c6d19bad71816f5c6d95803f734e77567931ea
[113] https://github.com/PTB-PSt1/PyDynamic/commit/8c2ce38af4cb4a391da9e82c23c5754f494892ad
[114] https://github.com/PTB-PSt1/PyDynamic/commit/299165ef630ef54bcb877e8a3260038805609e4f
[115] https://github.com/PTB-PSt1/PyDynamic/commit/ffafa00ea8961bdd4e4414645102f6870063f3d7
[116] https://github.com/PTB-PSt1/PyDynamic/commit/e866aa41212767f9b24f966bfabf005c9f6cdf39
[117] https://github.com/PTB-PSt1/PyDynamic/commit/b937a8b979f947df3149aeda38401b90fc522ef4

3.13.2 Documentation

- **trimOrPad:** Enhance docstring (`4c0da58`[118])
- **_fir_filter:** Adjust docstring and ValueError (`090b8f8`[119])

3.14 v1.6.1 (2020-10-29)

3.14.1 Fix

- Flip theta in uncertainty formulas of FIRuncFilter (`dd04eea`[120])

[118] https://github.com/PTB-PSt1/PyDynamic/commit/4c0da58e2c9d2bcae212a757f7b80ef65f602cd1
[119] https://github.com/PTB-PSt1/PyDynamic/commit/090b8f832c54e4cb969de70d5825d5000410eacc
[120] https://github.com/PTB-PSt1/PyDynamic/commit/dd04eeace70ce4fe7a81fb432cc117f80af74d4f

CHAPTER

FOUR

CONTRIBUTOR COVENANT CODE OF CONDUCT

4.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

4.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

4.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

4.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

4.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at https://github.com/Met4FoF/agentMET4FOF/graphs/contributors. All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

4.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

4.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

4.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

4.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

4.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

4.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant[121], version 2.0, available at https://www.contributor-covenant.org/version/2/0/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder[122].

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

[121] https://www.contributor-covenant.org
[122] https://github.com/mozilla/diversity

CHAPTER

FIVE

ADVICES AND TIPS FOR CONTRIBUTORS

If you want to become active as developer, we provide all important information here to make the start as easy as possible. The code you produce should be seamlessly integrable into PyDynamic by aligning your work with the established workflows. This guide should work on all platforms and provide everything needed to start developing for PyDynamic. Please open an issue or ideally contribute to this guide as a start, if problems or questions arise.

5.1 Guiding principles

The PyDynamic development process is based on the following guiding principles:

- support all major Python versions supported upstream [123].
- actively maintain, ensuring security vulnerabilities or other issues are resolved in a timely manner
- employ state-of-the-art development practices and tools, specifically
  - follow semantic versioning[124]
  - use conventional commit messages[125]
  - consider the PEP8 style guide, wherever feasible

5.2 Get started developing

5.2.1 Get the code on GitHub and locally

For collaboration, we recommend forking the repository as described here[126]. Simply apply the changes to your fork and open a Pull Request on GitHub as described here[127]. For small changes it will be sufficient to just apply your changes on GitHub and send the PR right away. For more comprehensive work, you should clone your fork and read on carefully.

[123] https://devguide.python.org/#status-of-python-branches
[124] https://semver.org/
[125] https://www.conventionalcommits.org/en/v1.0.0/
[126] https://help.github.com/en/articles/fork-a-repo
[127] https://help.github.com/en/articles/creating-a-pull-request

5.2.2 Initial development setup

This guide assumes you already have a valid runtime environment for PyDynamic as described in the installation section of the docs[128]. To start developing, install the known to work dependencies' versions for your specific Python version.

Afterwards you should always install PyDynamic itself in "development mode"[129] into your virtual environment to be able to properly test against the shipped version:

```
(PyDynamic_venv) $ python -m pip install -e .
```

For the testing refer to *the according testing section in this guide*.

5.2.3 Advised toolset

We use *black*[130] to implement our coding style, *Sphinx*[131] for automated generation of our documentation on ReadTheDocs[132]. We use *pytest*[133] managed by *tox*[134] as testing framework backed by *hypothesis*[135] and *coverage*[136]. For automated releases we use *python-semantic-release*[137] in our pipeline on *CircleCI*[138] . All requirements for contributions are derived from this. If you followed the steps for the initial development setup[139] you have everything at your hands.

5.2.4 Coding style

As long as the readability of mathematical formulations is not impaired, our code should follow PEP8[140]. For automating this uniform formatting task we use the Python package *black*[141]. It is easy to handle and integrable into most common IDEs[142], such that it is automatically applied.

5.2.5 Commit messages

PyDynamic commit messages follow some conventions to be easily human and machine-readable.

[128] https://pydynamic.readthedocs.io/en/latest/INSTALL.html#install-known-to-work-dependencies-versions
[129] https://packaging.python.org/tutorials/installing-packages/#installing-from-a-local-src-tree
[130] https://pypi.org/project/black/
[131] https://pypi.org/project/Sphinx/
[132] https://pydynamic.readthedocs.io/en/latest/
[133] https://pypi.org/project/pytest/
[134] https://pypi.org/project/tox/
[135] https://pypi.org/project/hypothesis/
[136] https://pypi.org/project/coverage/
[137] https://github.com/relekang/python-semantic-release
[138] https://app.circleci.com/pipelines/github/PTB-M4D/PyDynamic
[139] https://pydynamic.readthedocs.io/en/latest/INSTALL.html#install-known-to-work-dependencies-versions
[140] https://www.python.org/dev/peps/pep-0008/
[141] https://pypi.org/project/black/
[142] https://github.com/psf/black#editor-integration

Commit message structure

Conventional commit messages[143] are required for the following:

- Releasing automatically according to semantic versioning[144]
- Generating a changelog automatically[145]

Parts of the commit messages and links appear in the changelogs of subsequent releases as a result. We use the following types:

- *feat*: for commits that introduce new features (this correlates with MINOR in semantic versioning)
- *docs*: for commits that contribute significantly to documentation
- *fix*: commits in which bugs are fixed (this correlates with PATCH in semantic versioning)
- *build*: Commits that affect packaging
- *ci*: Commits that affect the CI pipeline
- *test*: Commits that apply significant changes to tests
- *chore*: Commits that affect other non-PyDynamic components (e.g., ReadTheDocs, Git, ...)
- *revert*: commits, which undo previous commits using `git revert`[146]
- *refactor*: commits that merely reformulate, rename or similar
- *style*: commits, which essentially make changes to line breaks and whitespace
- *wip*: Commits which are not recognizable as one of the above-mentioned types until later, usually during a PR merge. The merge commit is then marked as the corresponding type.

Of the types mentioned above, the following appear in separate sections of the changelog:

- *Feature*: *feat*
- *Documentation*: *docs*
- *Fix*: *fix*
- *Test*: *test*

Commit message styling

Based on conventional commit messages, the so-called should complete the following sentence:

> If this commit is applied, it will...

The first line of a commit message should not exceed 100 characters.

[143] https://www.conventionalcommits.org/en/v1.0.0/#summary
[144] https://semver.org/
[145] https://pydynamic.readthedocs.io/en/latest/CHANGELOG.html
[146] https://git-scm.com/docs/git-revert

BREAKING CHANGEs

If a commit changes parts of PyDynamic's public interface so that previously written code may no longer be executable, it is considered a BREAKING CHANGE (correlating with MAJOR in semantic versioning). This has to be marked by putting *BREAKING CHANGE* in the footer of the message as described in the conventional commit messages [147]. The introduced change and especially how to convert breaking code to further work should follow without a blank line and will be included in the changelog in full length.

Examples

For examples please check out the Git Log[148].

5.2.6 Testing

We strive to increase our code coverage[149] with every change introduced. This requires that every new feature and every change to existing features is accompanied by appropriate *pytest* testing. We test the basic components for correctness and, if necessary, the integration into the big picture. It is usually sufficient to create appropriately named[150] methods in one of the existing modules in the subfolder test. If necessary, add a new module that is appropriately named.

To execute the whole of the PyDynamic test suite simply run:

```
(PyDynamic_venv) $ python -m pytest
```

Further details, e.g. to execute only one specific test, can be found in the official pytest docs[151].

5.3 Workflow for adding completely new functionality

In case you add a new feature you generally follow the pattern:

- read through and follow this contribution advices and tips, especially regarding the *advised tool* set and *coding style*
- open an according issue to submit a feature request and get in touch with other PyDynamic developers and users
- fork the repository or update the *master* branch of your fork and create an arbitrary named feature branch from *master*
- decide which package and module your feature should be integrated into
- if there is no suitable package or module, create a new one and a corresponding module in the *test* subdirectory with the same name prefixed by *test_*
- after adding your functionality add it to all higher-level `__all__` variables in the module itself and in the higher-level `__init__.py`s
- if new dependencies are introduced, add them to *setup.py* or *dev-requirements.in*
- during development write tests in alignment with existing test modules, for example *test_interpolate*[152] or *test_propagate_filter*[153]

[147] https://www.conventionalcommits.org/en/v1.0.0/#summary
[148] https://github.com/PTB-M4D/PyDynamic/commits/master
[149] https://codecov.io/gh/PTB-M4D/PyDynamic
[150] https://docs.pytest.org/en/latest/goodpractices.html#conventions-for-python-test-discovery
[151] https://docs.pytest.org/en/latest/how-to/usage.html
[152] https://github.com/PTB-M4D/PyDynamic/blob/master/test/test_interpolate.py
[153] https://github.com/PTB-M4D/PyDynamic/blob/master/test/test_propagate_filter.py

- write docstrings in the NumPy docstring format[154] for all public functions you add
- as early as possible create a draft pull request onto the upstream's *master* branch
- once you think your changes are ready to merge, request a review[155] from the *PTB-M4D/pydynamic-devs* (you will find them in the according drop-down) and mark your PR as *ready for review*[156]
- at the latest now you will have the opportunity to review the documentation automatically generated from the docstrings on ReadTheDocs after your reviewers will set up everything
- resolve the conversations and have your pull request merged

5.4 Documentation

The documentation of PyDynamic consists of three parts. Every adaptation of an existing feature and every new feature requires adjustments on all three levels:

- user documentation on ReadTheDocs
- examples in the form of Jupyter notebooks for extensive features and Python scripts for features which can be comprehensively described with few lines of commented code
- developer documentation in the form of comments in the code

5.4.1 User documentation

To locally generate a preview of what ReadTheDocs will generate from your docstrings, you can simply execute after activating your virtual environment:

```
(PyDynamic_venv) $ sphinx-build docs/ docs/_build
Sphinx v3.1.1 in Verwendung
making output directory...
[...]
build abgeschlossen.

The HTML pages are in docs/_build.
```

After that, you can open the file ./docs/_build/index.html relative to the project's root with your favourite browser. Simply re-execute the above command after each change to the docstrings to update your local version of the documentation.

[154] https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard
[155] https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/requesting-a-pull-request-review
[156] https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/changing-the-stage-of-a-pull-request#marking-a-pull-request-as-ready-for-review

5.4.2 Examples

We want to provide extensive sample material for all PyDynamic features in order to simplify the use or even make it possible in the first place. We collect the examples in a separate repository PyDynamic_tutorials [157] separate from PyDynamic to better distinguish the core functionality from additional material. Please refer to the corresponding README for more information about the setup and create a pull request accompanying the pull request in PyDynamic according to the same procedure we describe here.

5.4.3 Comments in the code

Regarding comments in the code we recommend investing 45 minutes for the PyCon DE 2019 Talk of Stefan Schwarzer, a 20+-years Python developer: Commenting code - beyond common wisdom[158].

5.5 Manage dependencies

As stated in the README and above we use *pip-tools*[159] for dependency management. The requirements' subdirectory contains a *requirements.txt* and a *dev-requirements.txt* for all supported Python versions, with a suffix naming the version, for example *requirements-py36.txt*[160] To keep them up to date semi-automatically we use the bash script *requirements/upgrade_dependencies.sh*[161]. It contains extensive comments on its use. *pip-tools*' command `pip-compile` finds the right versions from the dependencies listed in *setup.py* and the *dev-requirements.in*.

5.6 Licensing

All contributions are released under PyDynamic's GNU Lesser General Public License v3.0[162].

[157] https://github.com/PTB-M4D/PyDynamic_tutorials
[158] https://www.youtube.com/watch?v=tP5uWCruaBs&list=PLHd2BPBhxqRLEhEaOFMWHBGpzyyF-ChZU&index=22&t=0s
[159] https://pypi.org/project/pip-tools/
[160] https://github.com/PTB-M4D/PyDynamic/blob/master/requirements/requirements-py36.txt
[161] https://github.com/PTB-M4D/PyDynamic/blob/master/requirements/upgrade_dependencies.sh
[162] https://github.com/PTB-M4D/PyDynamic/blob/master/licence.txt

CHAPTER

SIX

EXAMPLES

On the project website in the *examples* subfolder and the separate PyDynamic_tutorials[163] repository you can find various examples illustrating the application of PyDynamic. Here we provide only a quick starter.

6.1 Quick Examples

Uncertainty propagation for the application of an FIR filter with coefficients *b* with which an uncertainty *ub* is associated. The filter input signal is *x* with known noise standard deviation *sigma*. The filter output signal is *y* with associated uncertainty *uy*.

```
from PyDynamic.uncertainty.propagate_filter import FIRuncFilter
y, uy = FIRuncFilter(x, sigma, b, ub)
```

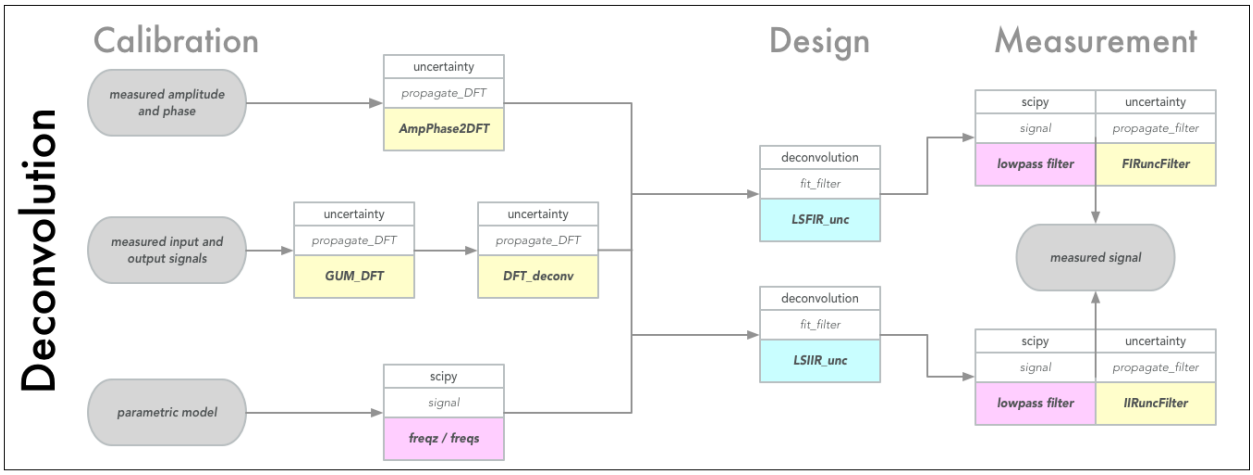
Uncertainty propagation through the application of the discrete Fourier transform (DFT). The time domain signal is *x* with associated squared uncertainty *ux*. The result of the DFT is the vector *X* of real and imaginary parts of the DFT applied to *x* and the associated uncertainty *UX*.

```
from PyDynamic.uncertainty.propagate_DFT import GUM_DFT
X, UX = GUM_DFT(x, ux)
```

Sequential application of the Monte Carlo method for uncertainty propagation for the case of filtering a time domain signal *x* with an IIR filter *b,a* with uncertainty associated with the filter coefficients *Uab* and signal noise standard deviation *sigma*. The filter output is the signal *y* and the Monte Carlo method calculates point-wise uncertainties *uy* and coverage intervals *Py* corresponding to the specified percentiles.

```
from PyDynamic.uncertainty.propagate_MonteCarlo import SMC
y, uy, Py = SMC(x, sigma, b, a, Uab, runs=1000, Perc=[0.025,0.975])
```

[163] https://github.com/PTB-M4D/PyDynamic_tutorials



6.2 Detailed examples

More comprehensive examples you can find in provided Jupyter notebooks, which require additional dependencies to be installed. This can be achieved by appending `[examples]` to PyDynamic in the install command, e.g.

```
pip install PyDynamic[examples]
```

Afterwards you can browser through the following list:

```
%pylab inline
import numpy as np
import scipy.signal as dsp
from palettable.colorbrewer.qualitative import Dark2_8

colors = Dark2_8.mpl_colors
rst = np.random.RandomState(1)
```

```
Populating the interactive namespace from numpy and matplotlib
```

6.2.1 Design of a digital deconvolution filter (FIR type)

```
from PyDynamic.model_estimation.fit_filter import LSFIR
from PyDynamic.misc.SecondOrderSystem import *
from PyDynamic.misc.testsignals import shocklikeGaussian
from PyDynamic.misc.filterstuff import kaiser_lowpass, db
from PyDynamic.uncertainty.propagate_filter import FIRuncFilter
from PyDynamic.misc.tools import make_semiposdef
```

```
# parameters of simulated measurement
Fs = 500e3
Ts = 1 / Fs

# sensor/measurement system
f0 = 36e3; uf0 = 0.01*f0
```

(continues on next page)

(continued from previous page)

```
S0 = 0.4; uS0= 0.001*S0
delta = 0.01; udelta = 0.1*delta

# transform continuous system to digital filter
bc, ac = sos_phys2filter(S0,delta,f0)
b, a = dsp.bilinear(bc, ac, Fs)

# Monte Carlo for calculation of unc. assoc. with [real(H),imag(H)]
f = np.linspace(0, 120e3, 200)
Hfc = sos_FreqResp(S0, delta, f0, f)
Hf = dsp.freqz(b,a,2*np.pi*f/Fs)[1]

runs = 10000
MCS0 = S0 + rst.randn(runs)*uS0
MCd  = delta+ rst.randn(runs)*udelta
MCf0 = f0 + rst.randn(runs)*uf0
HMC = np.zeros((runs, len(f)),dtype=complex)
for k in range(runs):
    bc_,ac_ = sos_phys2filter(MCS0[k], MCd[k], MCf0[k])
    b_,a_ = dsp.bilinear(bc_,ac_,Fs)
    HMC[k,:] = dsp.freqz(b_,a_,2*np.pi*f/Fs)[1]

H = np.r_[np.real(Hf), np.imag(Hf)]
uAbs = np.std(np.abs(HMC),axis=0)
uPhas= np.std(np.angle(HMC),axis=0)
UH= np.cov(np.hstack((np.real(HMC),np.imag(HMC))),rowvar=0)
UH= make_semiposdef(UH)
```

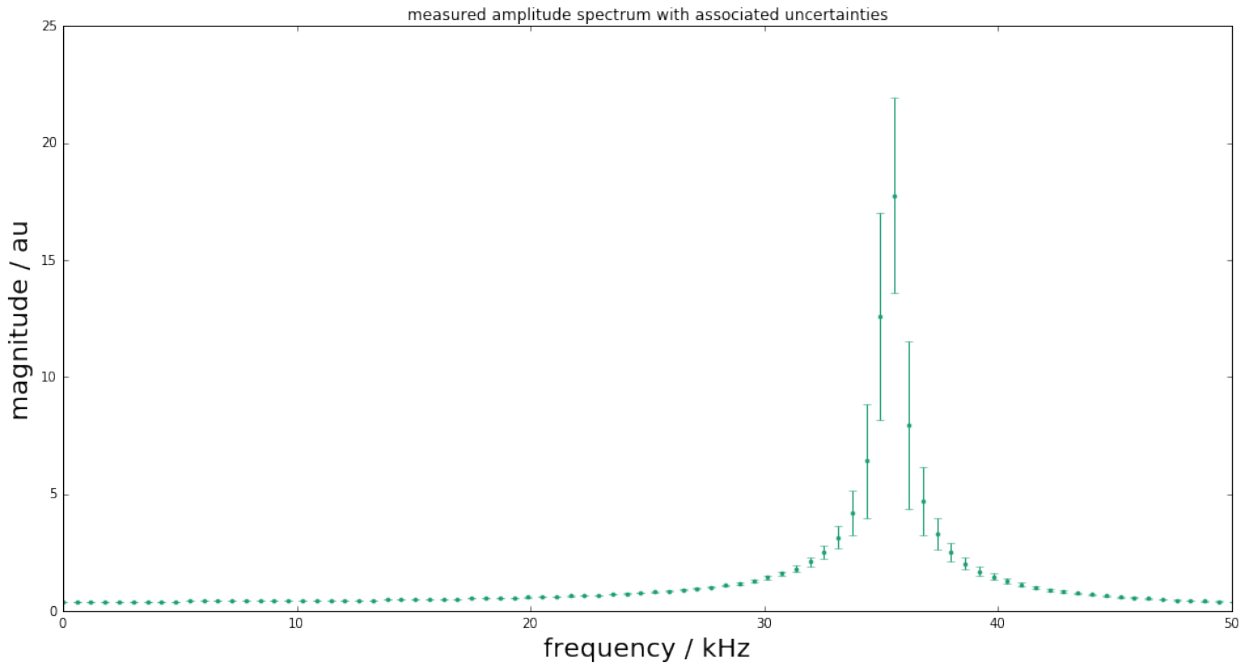
Problem description

Assume information about a linear time-invariant (LTI) measurement system to be available in terms of its frequency response values $H(j\omega)$ at a set of frequencies together with associated uncertainties:

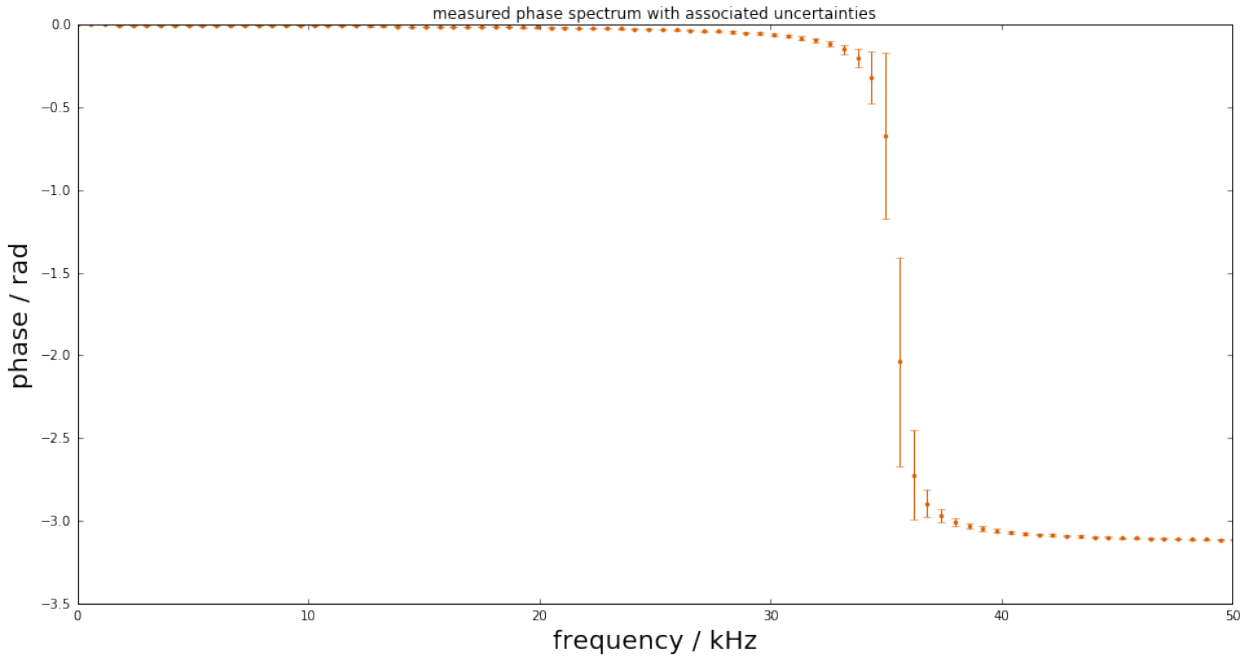
$$\mathbf{H} = (|H(j\omega_1)|, \ldots, |H(j\omega_N)|, \angle H(j\omega_1), \ldots, \angle H(j\omega_N)) \tag{6.1}$$

$$u(\mathbf{H}) = (u(|H(j\omega_1)|), \ldots, u(|H(j\omega_N)|), u(\angle H(j\omega_1)), \ldots, u(\angle H(j\omega_N)))$$

```
figure(figsize=(16,8))
errorbar(f*1e-3, np.abs(Hf), uAbs, fmt=".", color=colors[0])
title("measured amplitude spectrum with associated uncertainties")
xlim(0,50)
xlabel("frequency / kHz",fontsize=20)
ylabel("magnitude / au",fontsize=20);
```




```
figure(figsize=(16,8))
errorbar(f*1e-3, np.angle(Hf), uPhas, fmt=".", color=colors[1])
title("measured phase spectrum with associated uncertainties")
xlim(0,50)
xlabel("frequency / kHz",fontsize=20)
ylabel("phase / rad",fontsize=20);
```



Simulated measurement

Measurements with this system are then modeled as a convolution of the system's impulse response

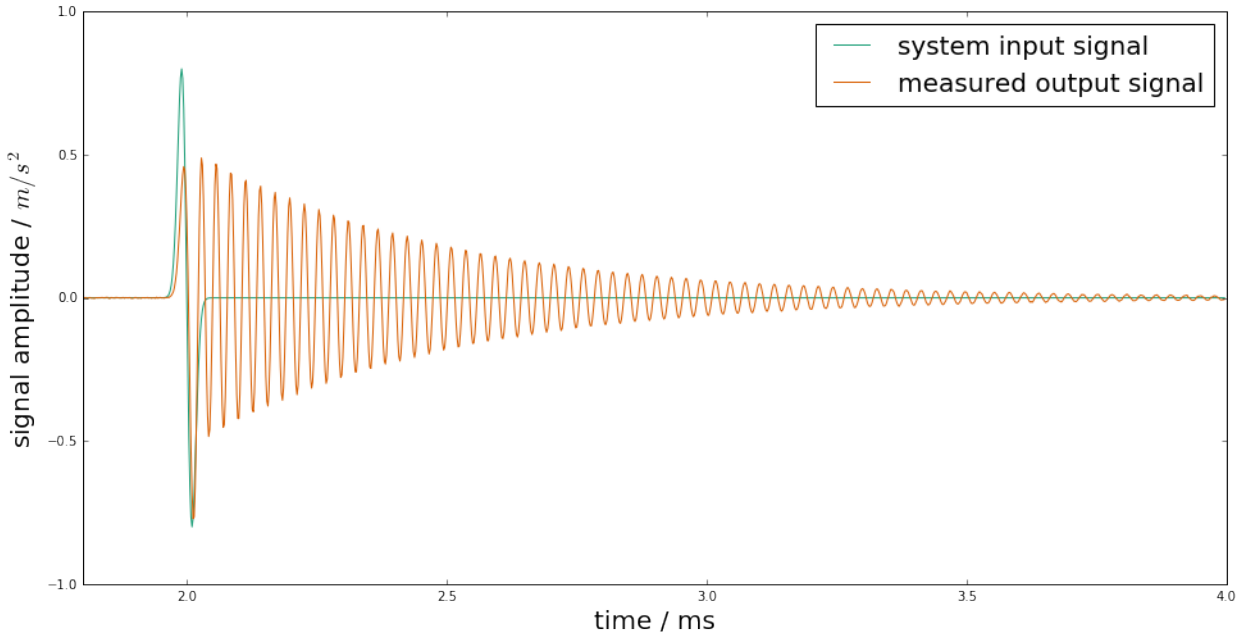
$$h(t) = \mathcal{F}^{-1}(H(j\omega))$$

with the input signal $x(t)$, after an analogue-to-digital conversion producing the measured signal

$$y[n] = (h * x)(t_n) \qquad n = 1, \ldots, M$$

```
# simulate input and output signals
time = np.arange(0, 4e-3 - Ts, Ts)
#x = shocklikeGaussian(time, t0 = 2e-3, sigma = 1e-5, m0=0.8)
m0 = 0.8; sigma = 1e-5; t0 = 2e-3
x = -m0*(time-t0)/sigma * np.exp(0.5)*np.exp(-(time-t0) ** 2 / (2 * sigma ** 2))
y = dsp.lfilter(b, a, x)
noise = 1e-3
yn = y + rst.randn(np.size(y)) * noise
```

```
figure(figsize=(16,8))
plot(time*1e3, x, label="system input signal", color=colors[0])
plot(time*1e3, yn,label="measured output signal", color=colors[1])
legend(fontsize=20)
xlim(1.8,4); ylim(-1,1)
xlabel("time / ms",fontsize=20)
ylabel(r"signal amplitude / $m/s^2$",fontsize=20);
```



Design of the deconvolution filter

The aim is to derive a digital filter with finite impulse response (FIR)

$$g(z) = \sum_{k=0}^{K} b_k z^{-k}$$

such that the filtered signal

$$\hat{x}[n] = (g * y)[n] \qquad n = 1, \ldots, M$$

is an estimate of the system's input signal at the discrete time points.

Publication

- Elster and Link "Uncertainty evaluation for dynamic measurements modelled by a linear time-invariant system" Metrologia, 2008
- Vuerinckx R, Rolain Y, Schoukens J and Pintelon R "Design of stable IIR filters in the complex domain by automatic delay selection" IEEE Trans. Signal Process. 44 2339–44, 1996

Determine FIR filter coefficients such that

$$H(j\omega)g(e^{j\omega/F_s}) \approx e^{-j\omega n_0/F_s} \qquad \text{for} \qquad |\omega| \leq \omega_1$$

with a pre-defined time delay n_0 to improve the fit quality (typically half the filter order).

Consider as least-squares problem

$$(y - Xb)^T W^{-1} (y - Xb)$$

with - y real and imaginary parts of the *reciprocal* and phase shifted measured frequency response values - X the model matrix with entries $e^{-jk\omega/Fs}$ - b the sought FIR filter coefficients - W a weighting matrix (usually derived from the uncertainties associated with the frequency response measurements

Filter coefficients and associated uncertainties are thus obtained as

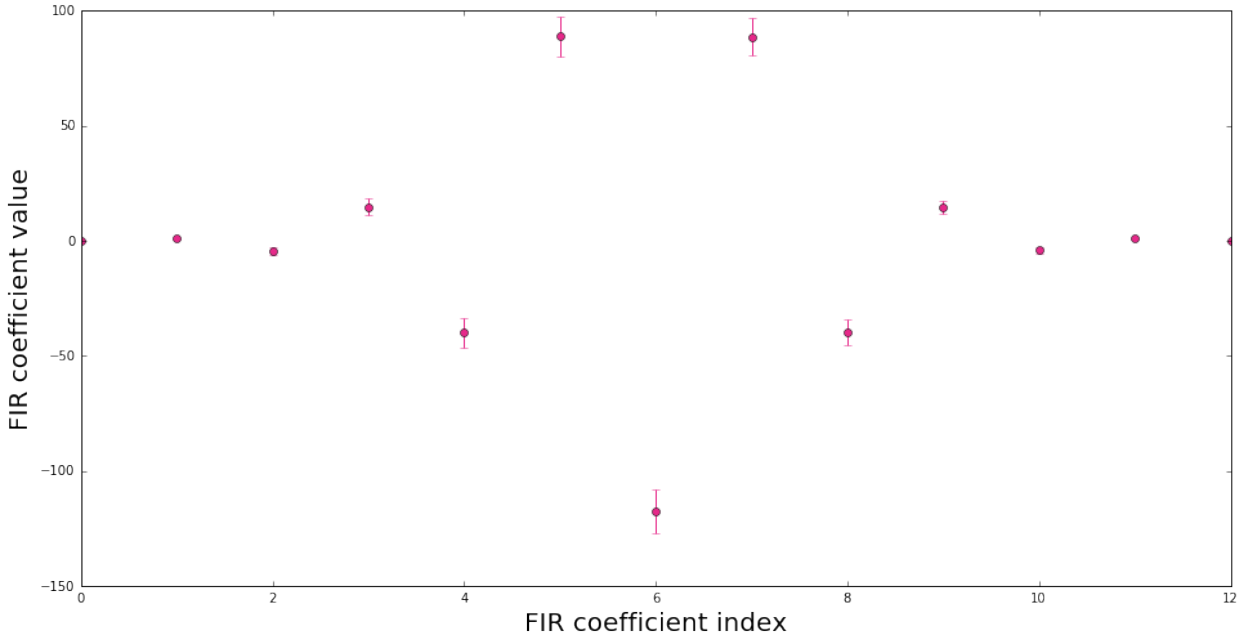
$$b = \left(X^T W^{-1} X\right)^{-1} X^T W^{-1} y$$

$$u_b = \left(X^T W^{-1} X\right)^{-1} X^T W^{-1} U_y W^{-1} X \left(X^T W^{-1} X\right)^{-1}$$

```
# Calculation of FIR deconvolution filter and its assoc. unc.
N = 12; tau = N//2
bF, UbF = LSFIR(H,N,tau,f,Fs,UH=UH)
```

```
Least-squares fit of an order 12 digital FIR filter to the
reciprocal of a frequency response given by 400 values
and propagation of associated uncertainties.
Final rms error = 1.545423e+01
```

```
figure(figsize=(16,8))
errorbar(range(N+1), bF, np.sqrt(np.diag(UbF)), fmt="o", color=colors[3])
xlabel("FIR coefficient index", fontsize=20)
ylabel("FIR coefficient value", fontsize=20);
```



In order to render the ill-posed estimation problem stable, the FIR inverse filter is accompanied with an FIR low-pass filter.

Application of the deconvolution filter for input estimation is then carried out as

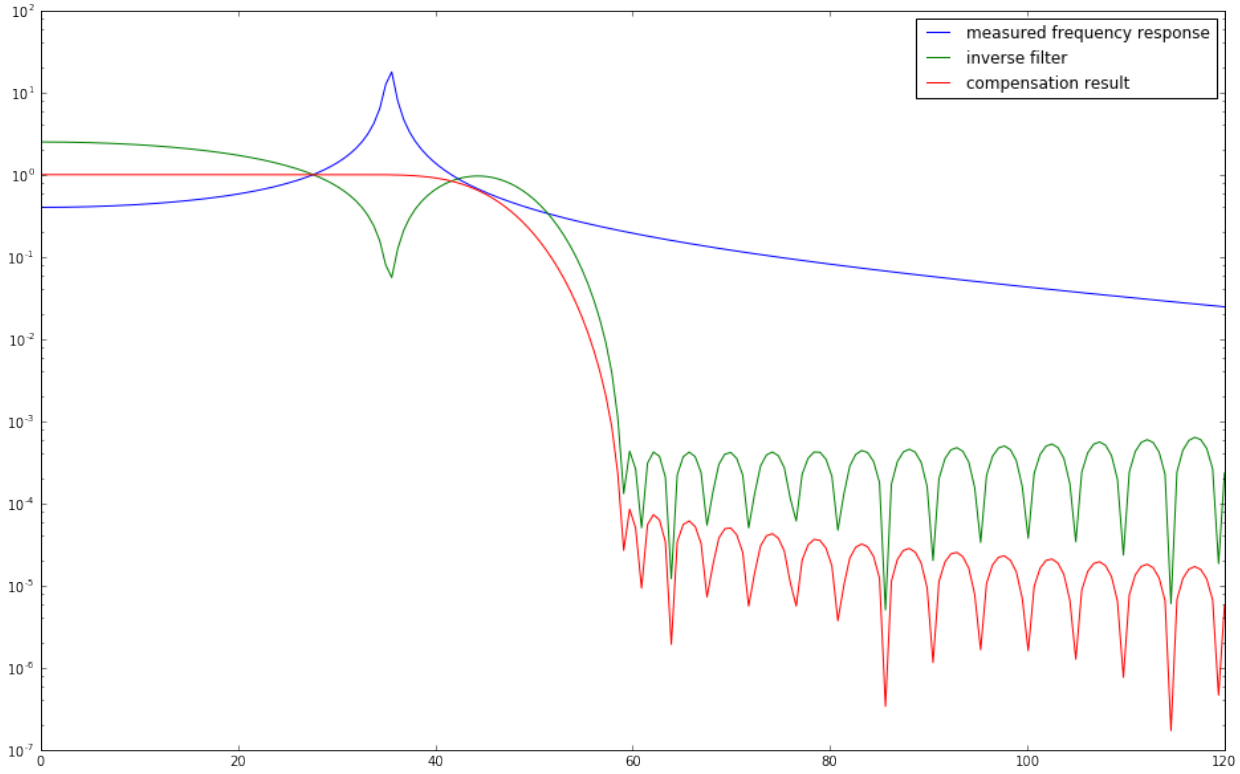
$$\hat{x}[n - n_0] = (g * (g_{low} * y)[n]$$

with point-wise associated uncertainties calculated as

$$u^2(\hat{x}[n - n_0] = b^T U_{x_{low}[n]} b + x_{low}^T[n] U_b x_{low}[n] + trace(U_{x_{low}[n]} U_b)$$

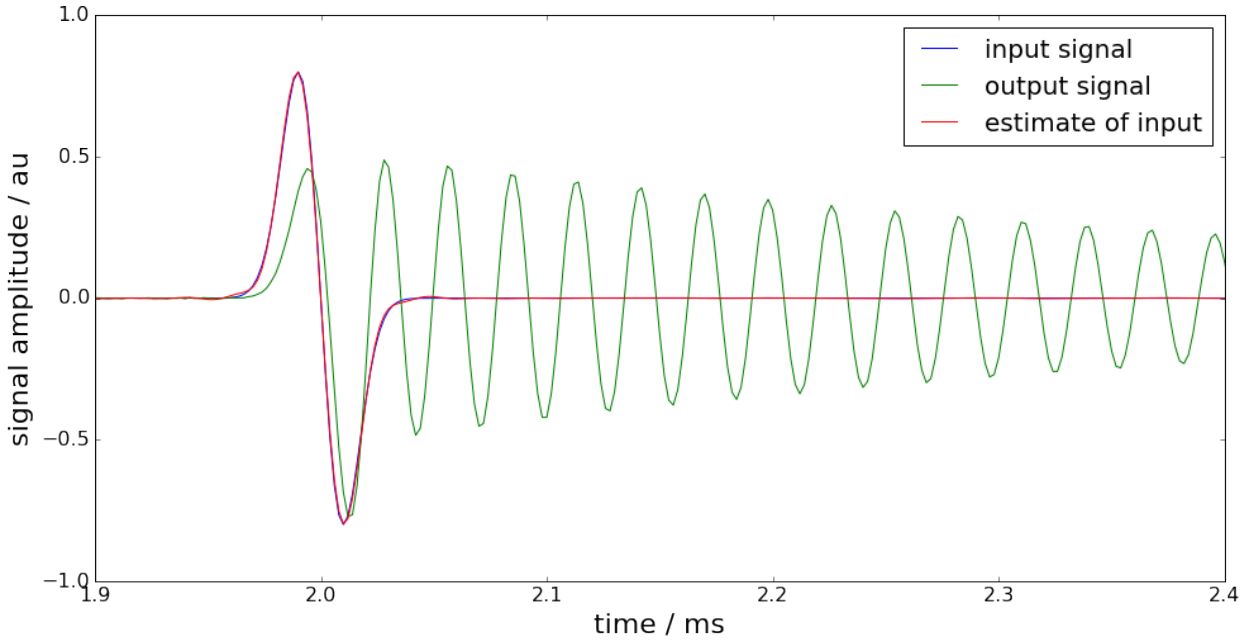
```
fcut = f0+10e3; low_order = 100
blow, lshift = kaiser_lowpass(low_order, fcut, Fs)
shift = -tau - lshift
```

```
figure(figsize=(16,10))
HbF = dsp.freqz(bF,1,2*np.pi*f/Fs)[1]*dsp.freqz(blow,1,2*np.pi*f/Fs)[1]
semilogy(f*1e-3, np.abs(Hf), label="measured frequency response")
semilogy(f*1e-3, np.abs(HbF),label="inverse filter")
semilogy(f*1e-3, np.abs(Hf*HbF), label="compensation result")
legend();
```

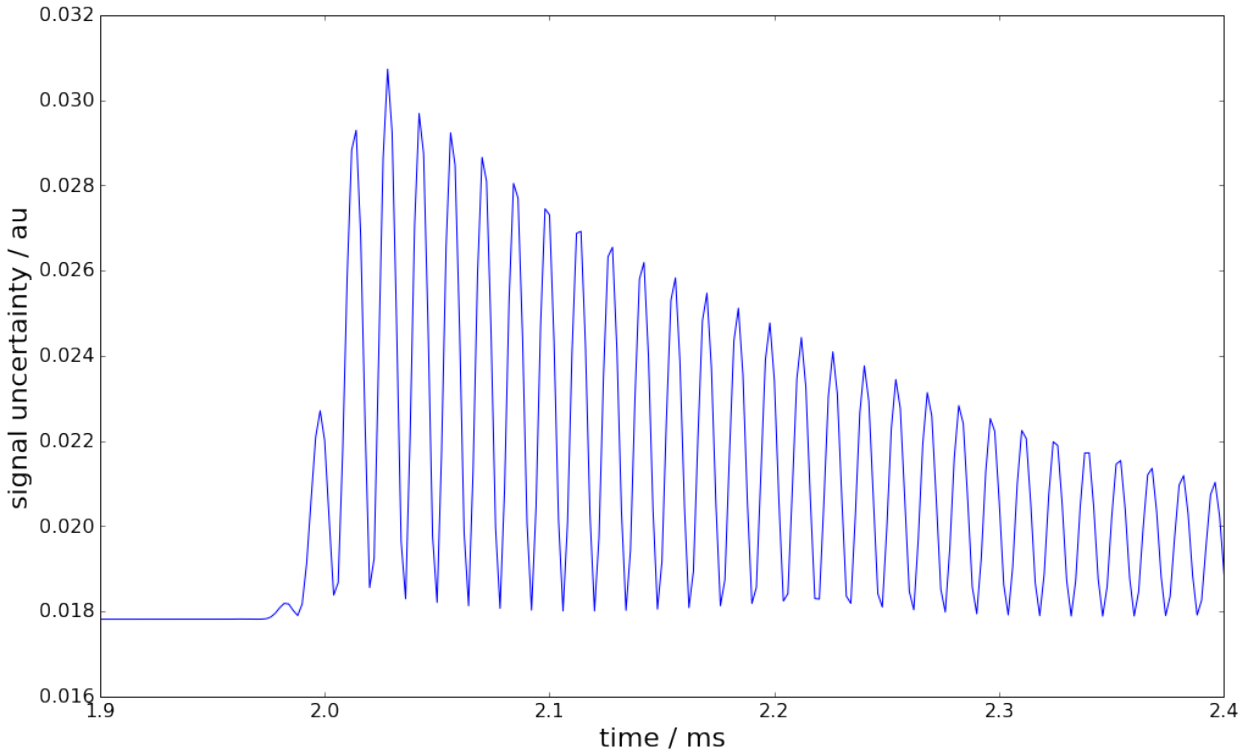



```
xhat,Uxhat = FIRuncFilter(yn,noise,bF,UbF,shift,blow)
```

```
figure(figsize=(16,8))
plot(time*1e3,x, label='input signal')
plot(time*1e3,yn,label='output signal')
plot(time*1e3,xhat,label='estimate of input')
legend(fontsize=20)
xlabel('time / ms',fontsize=22)
ylabel('signal amplitude / au',fontsize=22)
tick_params(which="both",labelsize=16)
xlim(1.9,2.4); ylim(-1,1);
```




```
figure(figsize=(16,10))
plot(time*1e3,Uxhat)
xlabel('time / ms',fontsize=22)
ylabel('signal uncertainty / au',fontsize=22)
subplots_adjust(left=0.15,right=0.95)
tick_params(which='both', labelsize=16)
xlim(1.9,2.4);
```



Basic workflow in PyDynamic

Fit an FIR filter to the reciprocal of the measured frequency response

```
from PyDynamic.model_estimation.fit_filter import LSFIR
bF, UbF = LSFIR(H,N,tau,f,Fs,verbose=False,UH=UH)
```

with

- `H` the measured frequency response values
- `UH` the covariance (i.e. uncertainty) associated with real and imaginary parts of `H`
- `N` the filter order
- `tau` the filter delay in samples
- `f` the vector of frequencies at which `H` is given
- `Fs` the sampling frequency for the digital FIR filter

Propagate the uncertainty associated with the measurement noise and the FIR filter through the deconvolution process

```
xhat,Uxhat = FIRuncFilter(yn,noise,bF,UbF,shift,blow)
```

with

- `yn` the noisy measurement
- `noise` the std of the noise
- `shift` the total delay of the FIR filter and the low-pass filter
- `blow` the coefficients of the FIR low-pass filter

```
%pylab inline
import scipy.signal as dsp
```

```
Populating the interactive namespace from numpy and matplotlib
```

6.2.2 Uncertainty propagation for IIR filters

```
from PyDynamic.misc.testsignals import rect
from PyDynamic.uncertainty.propagate_filter import IIRuncFilter
from PyDynamic.uncertainty.propagate_MonteCarlo import SMC
from PyDynamic.misc.tools import make_semiposdef
```

Digital filters with infinite impulse response (IIR) are a common tool in signal processing. Consider the measurand to be the output signal of an IIR filter with z-domain transfer function

$$G(z) = \frac{\sum_{n=0}^{N_b} b_n z^{-n}}{1 + \sum_{m=1}^{N_a} a_m z^{-m}}.$$

The measurement model is thus given by

$$y[k] = \sum_{n=0}^{N_b} b_n x[k-n] - \sum_{m=1}^{N_a} a_m y[k-m]$$

As input quantities to the model the input signal values $x[k]$ and the IIR filter coefficients $(b_0, \ldots, a_{N_a})$ are considered.

Linearisation-based uncertainty propagation

Scientific publication

```
A. Link and C. Elster,
“Uncertainty evaluation for IIR filtering using a
state-space approach,”
Meas. Sci. Technol., vol. 20, no. 5, 2009.
```

The linearisation method for the propagation of uncertainties through the IIR model is based on a state-space model representation of the IIR filter equation

$$z^\top[n+1] = \begin{pmatrix} -a_1 & \cdots & \cdots & -a_{N_a} \\ 0 & & & \\ \vdots & & I_{N_a-1} & \\ 0 & & & \end{pmatrix} z^\top[n] + \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} X[n], \tag{6.2}$$

$$y[n] = (d_1, \ldots, d_{N_a}) z^\top[n] + b_0 X[n] + \Delta[n], \tag{6.3}$$

$$y[n] = d^T z[n] + b_0 y[n] \tag{6.4}$$

$$u^2(y[n]) = \phi^T(n) U_\mu \phi(n) + d^T P_z[n] d + b_0^2, \tag{6.5}$$

where

$$\boldsymbol{\phi}(n) = \left(\frac{\partial x[n]}{\partial \mu_1}, \ldots, \frac{\partial x[n]}{\partial \mu_{N+N_a+N_b+1}} \right)^T$$

$$\mathbf{P}_\mathbf{z}[n] = \sum_{m<n} \left(\frac{\partial \mathbf{z}[n]}{\partial y[m]} \right) \left(\frac{\partial \mathbf{z}[n]}{\partial y[m]} \right)^T u^2(y[m]).$$

The linearization-based uncertainty propagation method for IIR filters provides

- propagation schemes for white noise and colored noise in the filter input signal
- incorporation of uncertainties in the IIR filter coefficients
- online evaluation of the point-wise uncertainties associated with the IIR filter output

Implementation in PyDynamic

`y,Uy = IIRuncFilter(x,noise,b,a,Uab)`

with

- `x` the filter input signal sequency
- `noise` the standard deviation of the measurement noise in x
- `b,a` the IIR filter coefficient
- `Uab` the covariance matrix associated with $(a_1, \ldots, b_{N_b})$

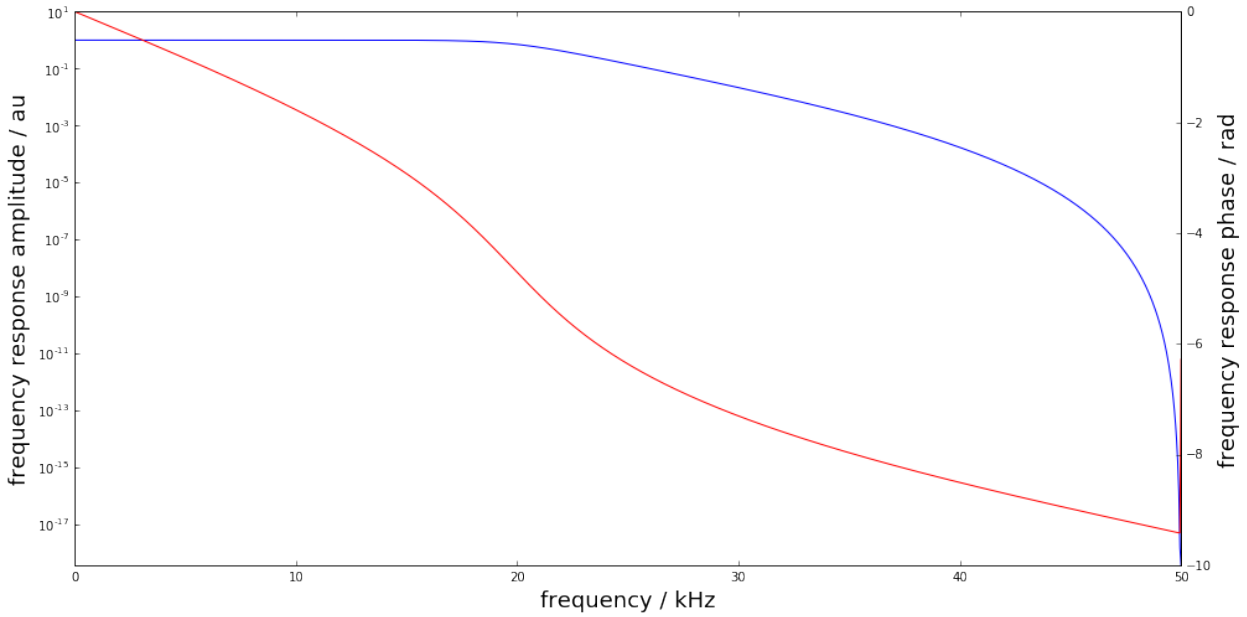
Remark

```
Implementation for more general noise processes than white noise is considered for one␣
↪of the next revisions.
```

Example

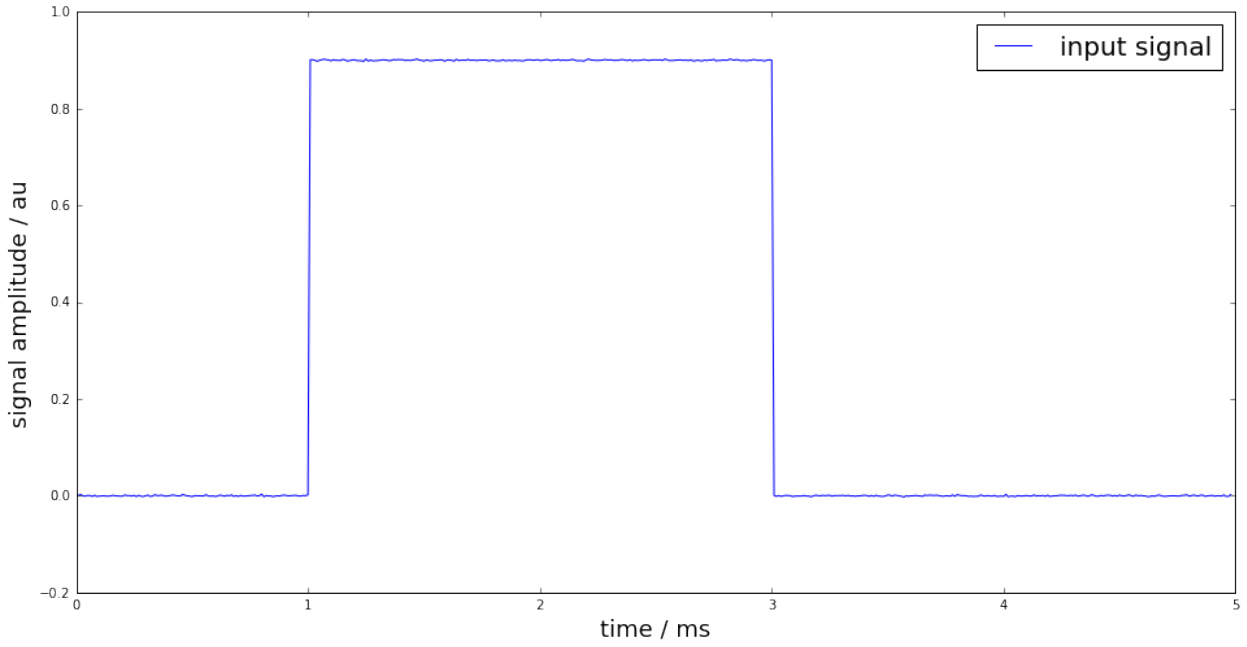
```
# parameters of simulated measurement
Fs = 100e3
Ts = 1.0/Fs

# nominal system parameter
fcut = 20e3
L = 6
b,a = dsp.butter(L,2*fcut/Fs,btype='lowpass')
```

```
f = linspace(0,Fs/2,1000)
figure(figsize=(16,8))
semilogy(f*1e-3, abs(dsp.freqz(b,a,2*np.pi*f/Fs)[1]))
ylim(0,10);
xlabel("frequency / kHz",fontsize=18); ylabel("frequency response amplitude / au",
↪fontsize=18)
ax2 = gca().twinx()
ax2.plot(f*1e-3, unwrap(angle(dsp.freqz(b,a,2*np.pi*f/Fs)[1])),color="r")
ax2.set_ylabel("frequency response phase / rad",fontsize=18);
```




```
time = np.arange(0,499*Ts,Ts)
t0 = 100*Ts; t1 = 300*Ts
height = 0.9
noise = 1e-3
x = rect(time,t0,t1,height,noise=noise)
```

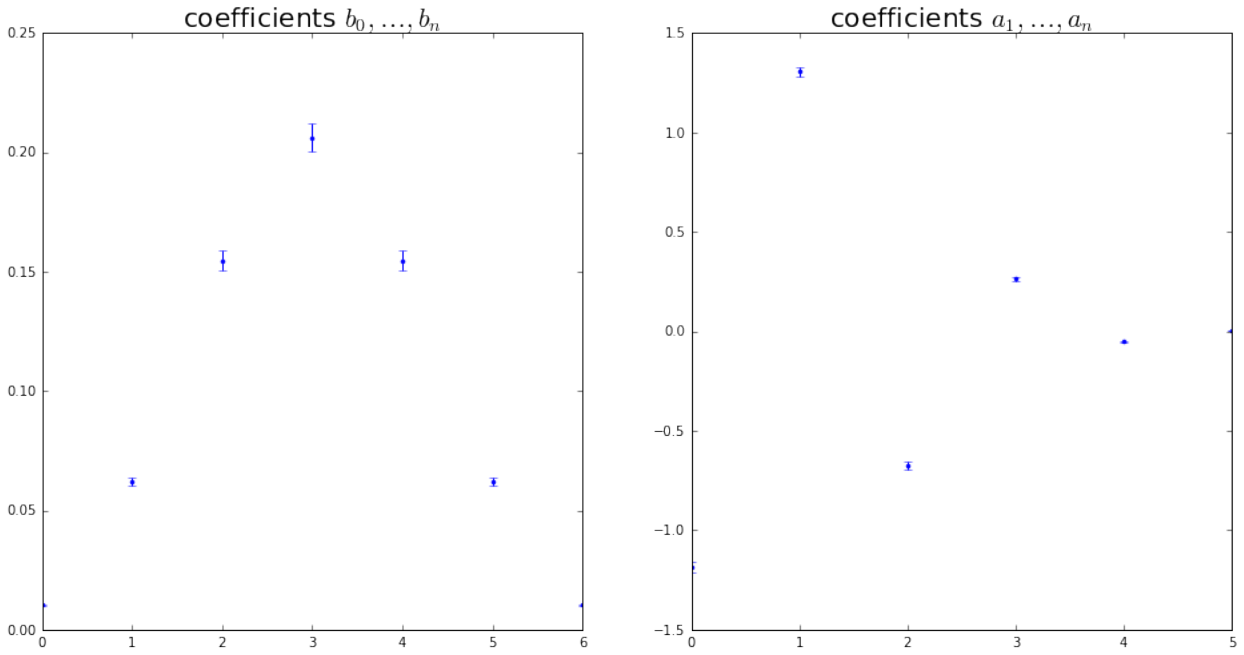
```
figure(figsize=(16,8))
plot(time*1e3, x, label="input signal")
legend(fontsize=20)
xlabel('time / ms',fontsize=18)
ylabel('signal amplitude / au',fontsize=18);
```




```
# uncertain knowledge: fcut between 19.8kHz and 20.2kHz
runs = 10000
FC = fcut + (2*np.random.rand(runs)-1)*0.2e3
AB = np.zeros((runs,len(b)+len(a)-1))

for k in range(runs):
    bb,aa = dsp.butter(L,2*FC[k]/Fs,btype='lowpass')
    AB[k,:] = np.hstack((aa[1:],bb))

Uab = make_semiposdef(np.cov(AB,rowvar=0))
```

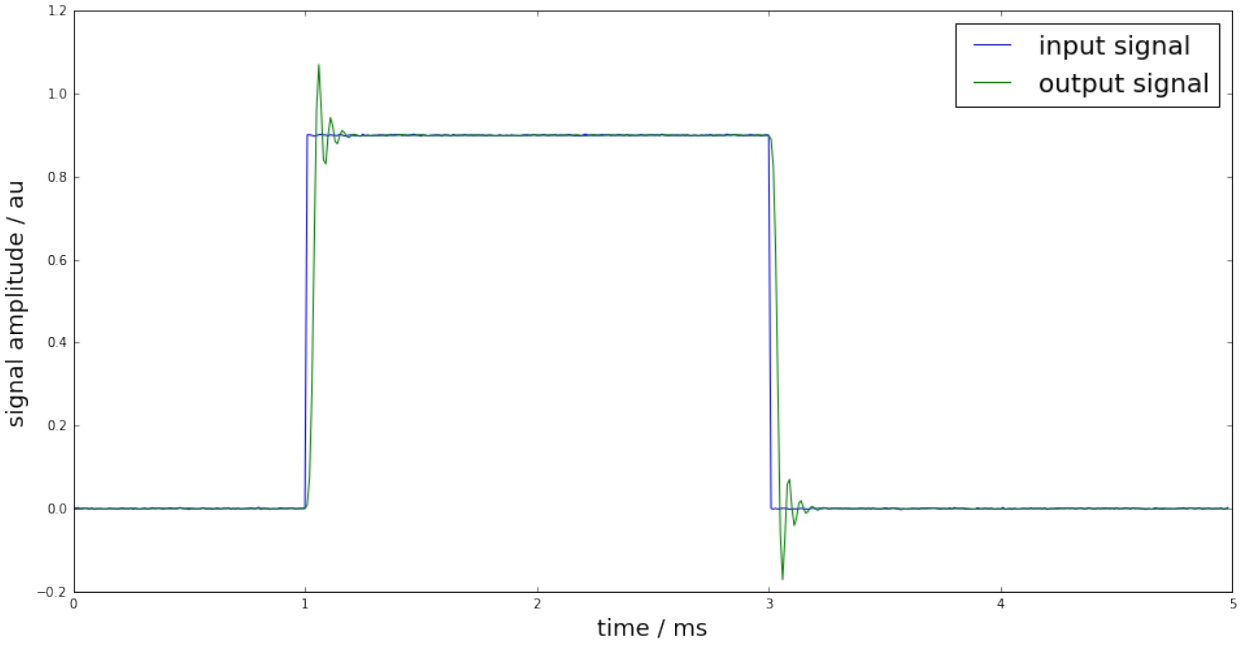
Uncertain knowledge: low-pass cut-off frequency is between 19.8 and 20.2 kHz

```
figure(figsize=(16,8))
subplot(121)
errorbar(range(len(b)), b, sqrt(diag(Uab[L:,L:])),fmt=".")
title(r"coefficients $b_0,\ldots,b_n$",fontsize=20)
subplot(122)
errorbar(range(len(a)-1), a[1:], sqrt(diag(Uab[:L, :L])),fmt=".");
title(r"coefficients $a_1,\ldots,a_n$",fontsize=20);
```

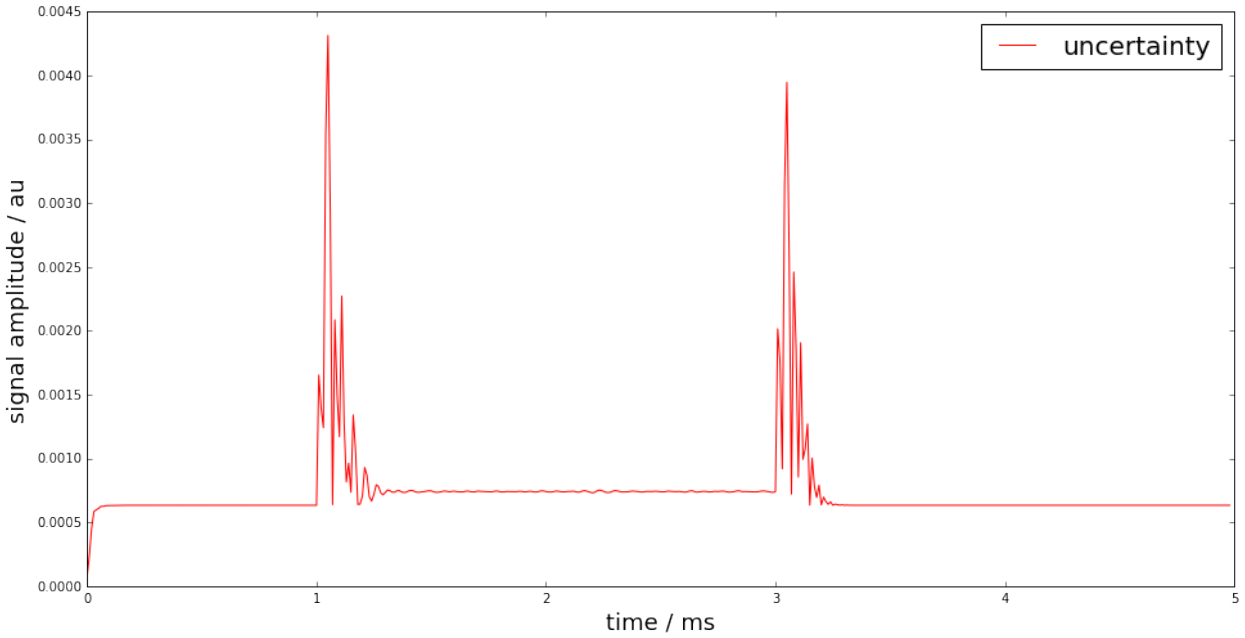


Estimate of the filter output signal and its associated uncertainty

```
y,Uy = IIRuncFilter(x,noise,b,a,Uab)

figure(figsize=(16,8))
plot(time*1e3, x, label="input signal")
plot(time*1e3, y, label="output signal")
legend(fontsize=20)
xlabel('time / ms',fontsize=18)
ylabel('signal amplitude / au',fontsize=18);
```



```
figure(figsize=(16,8))
plot(time*1e3, Uy, "r", label="uncertainty")
legend(fontsize=20)
xlabel('time / ms',fontsize=18)
ylabel('signal amplitude / au',fontsize=18);
```



Monte-Carlo method for uncertainty propagation

The linearisation-based uncertainty propagation can become unreliable due to the linearisation errors. Therefore, a Monte-Carlo method for digital filters with uncertain coefficients has been proposed in

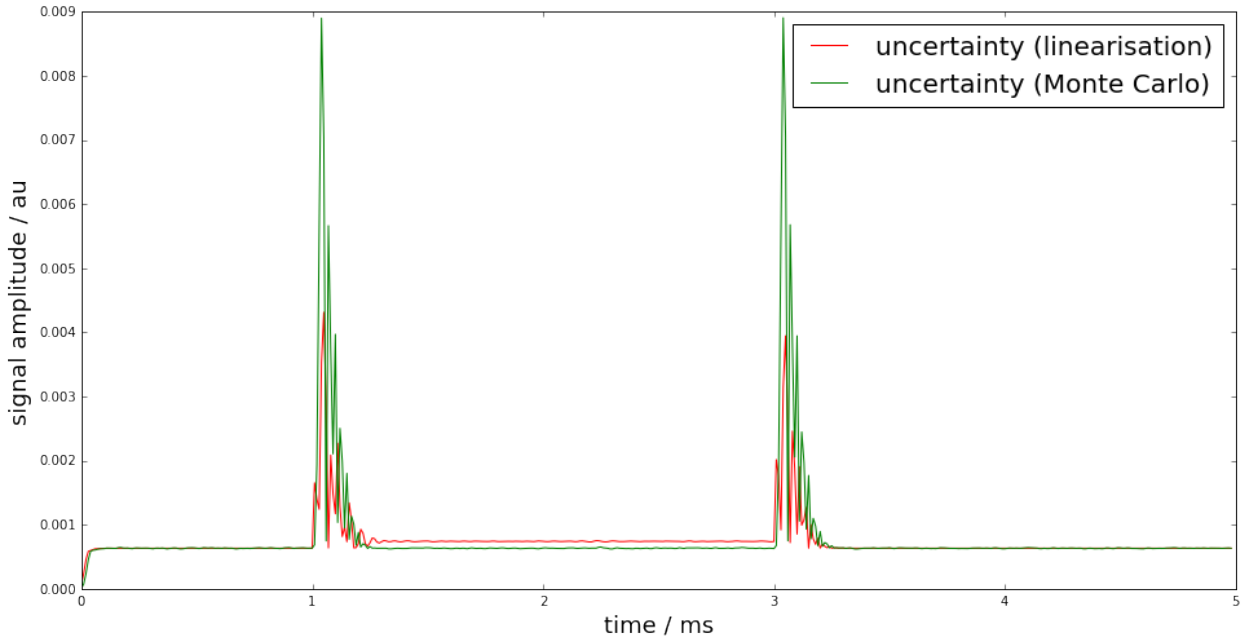
```
S. Eichstädt, A. Link, P. Harris, and C. Elster,
“Efficient implementation of a Monte Carlo method
for uncertainty evaluation in dynamic measurements,”
Metrologia, vol. 49, no. 3, 2012.
```

The proposed Monte-Carlo method provides - a memory-efficient implementation of the GUM Monte-Carlo method - online calculation of point-wise uncertainties, estimates and coverage intervals by taking advantage of the sequential character of the filter equation

$$y[k] = \sum_{n=0}^{N_b} b_n x[k-n] - \sum_{m=1}^{N_a} a_m y[k-m]$$

```
yMC,UyMC = SMC(x,noise,b,a,Uab,runs=10000)

figure(figsize=(16,8))
plot(time*1e3, Uy, "r", label="uncertainty (linearisation)")
plot(time*1e3, UyMC, "g", label="uncertainty (Monte Carlo)")
legend(fontsize=20)
xlabel('time / ms',fontsize=18)
ylabel('signal amplitude / au',fontsize=18);
```

```
SMC progress:  0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
```



Basic workflow in PyDynamic

Using GUM linearization

```
y,Uy = IIRuncFilter(x,noise,b,a,Uab)
```

Using sequential GUM Monte Carlo method

```
yMC,UyMC = SMC(x,noise,b,a,Uab,runs=10000)
```

```
SMC progress:  0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
```

```
%pylab inline
colors = [[0.1,0.6,0.5], [0.9,0.2,0.5], [0.9,0.5,0.1]]
```

```
Populating the interactive namespace from numpy and matplotlib
```

6.2.3 Deconvolution in the frequency domain (DFT)

```
from PyDynamic.uncertainty.propagate_DFT import GUM_DFT,GUM_iDFT
from PyDynamic.uncertainty.propagate_DFT import DFT_deconv, AmpPhase2DFT
from PyDynamic.uncertainty.propagate_DFT import DFT_multiply
```

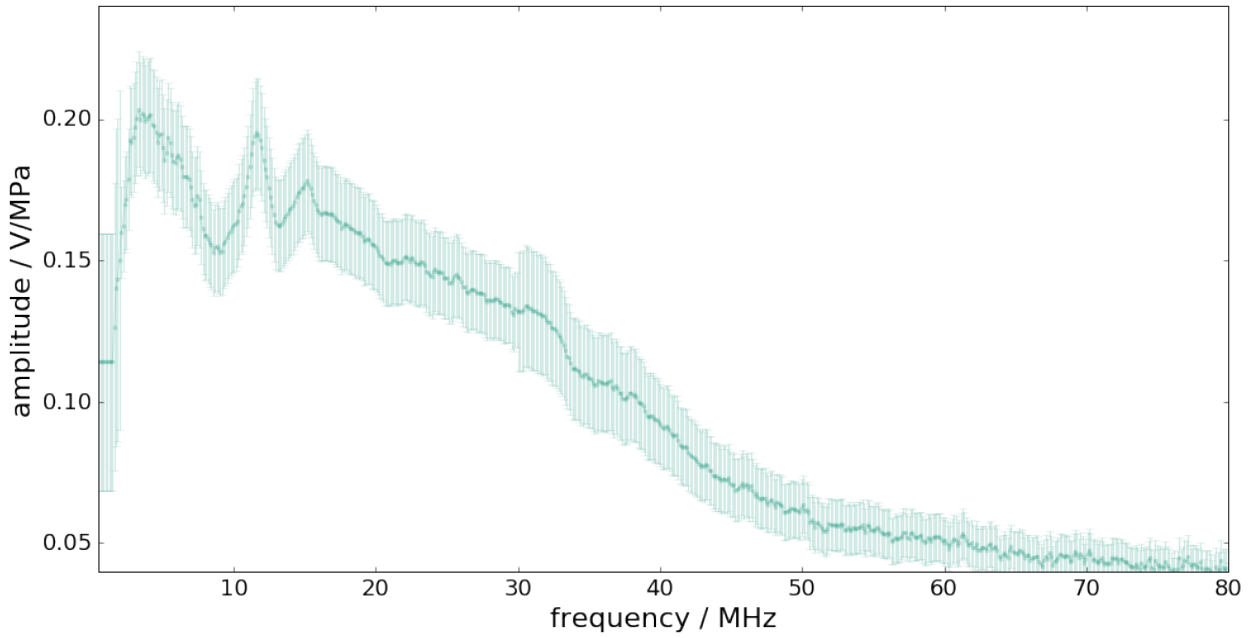
```
#%% reference data
ref_file = np.loadtxt("DFTdeconv reference_signal.dat")
time = ref_file[:,0]
ref_data = ref_file[:,1]
Ts = 2e-9
N = len(time)

#%% hydrophone calibration data
calib = np.loadtxt("DFTdeconv calibration.dat")
f = calib[:,0]
FR = calib[:,1]*np.exp(1j*calib[:,3])
Nf = 2*(len(f)-1)

uAmp = calib[:,2]
uPhas= calib[:,4]
UAP = np.r_[uAmp,uPhas*np.pi/180]**2

#%% measured hydrophone output signal
meas = np.loadtxt("DFTdeconv measured_signal.dat")
y = meas[:,1]
# assumed noise std
noise_std = 4e-4
Uy = noise_std**2
```

Consider knowledge about the measurement system is available in terms of its frequency response with uncertainties associated with amplitude and phase values.

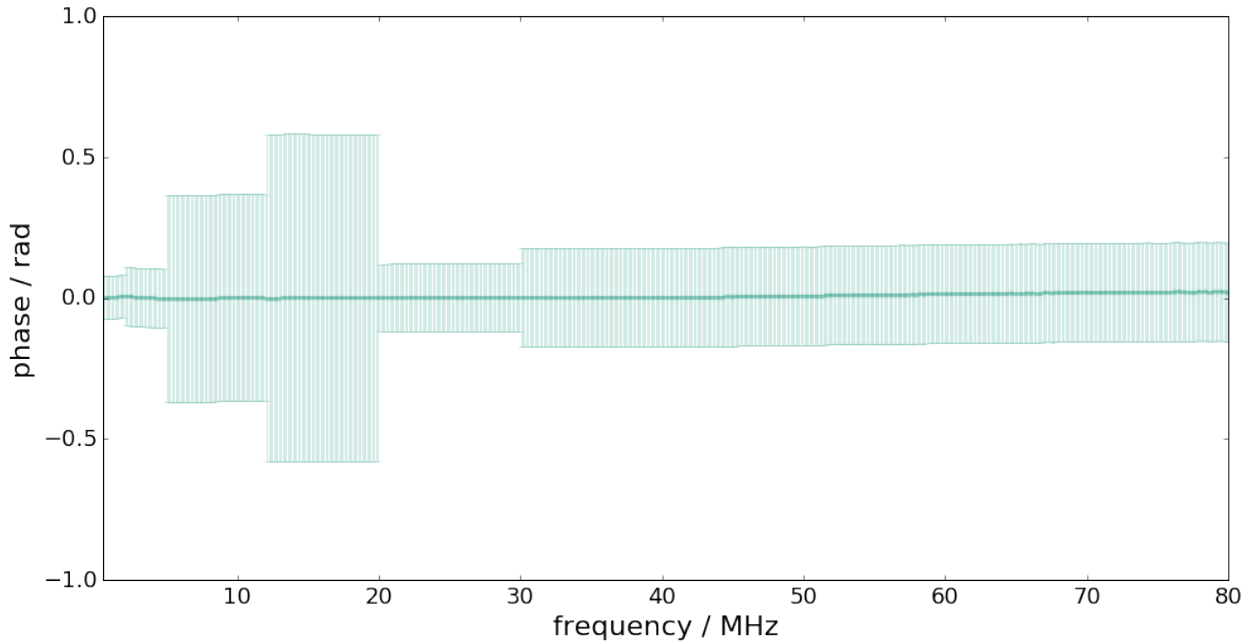
$$\mathbf{H} = (|H(f_1)|, \ldots, \angle H(f_N))$$

$$u_H = (u_{|H(f_1)|}, \ldots, u_{\angle H(f_N)})$$

```
figure(figsize=(16,8))
errorbar(f * 1e-6, abs(FR), 2 * sqrt(UAP[:len(UAP) // 2]), fmt= ".-", alpha=0.2,␣
↪color=colors[0])
xlim(0.5, 80)
ylim(0.04, 0.24)
xlabel("frequency / MHz", fontsize=22); tick_params(which= "both", labelsize=18)
ylabel("amplitude / V/MPa", fontsize=22);
```




```
figure(figsize=(16,8))
errorbar(f * 1e-6, unwrap(angle(FR)) * pi / 180, 2 * UAP[len(UAP) // 2:], fmt= ".-",␣
↪alpha=0.2, color=colors[0])
xlim(0.5, 80)
ylim(-0.2, 0.3)
xlabel("frequency / MHz", fontsize=22); tick_params(which= "both", labelsize=18)
ylim(-1,1)
ylabel("phase / rad", fontsize=22);
```



The measurand is the input signal $\mathbf{x} = (x_1, \ldots, x_M)$ to the measurement system with corresponding measurement

model given by

$$y[n] = (h * x)[n] + \varepsilon[n]$$

Input estimation is here to be considered in the Fourier domain.

The estimation model equation is thus given by

$$\hat{x} = \mathcal{F}^{-1}\left(\frac{Y(f)}{H(f)} H_L(f)\right)$$

with - $Y(f)$ the DFT of the measured system output signal - $H_L(f)$ the frequency response of a low-pass filter

Estimation steps

1) DFT of y and propagation of uncertainties to the frequency domain
2) Propagation of uncertainties associated with amplitude and phase of system to corr. real and imaginary parts
3) Division in the frequency domain and propagation of uncertainties
4) Multiplication with low-pass filter and propagation of uncertainties
5) Inverse DFT and propagation of uncertainties to the time domain

Propagation from time to frequency domain

With the DFT defined as

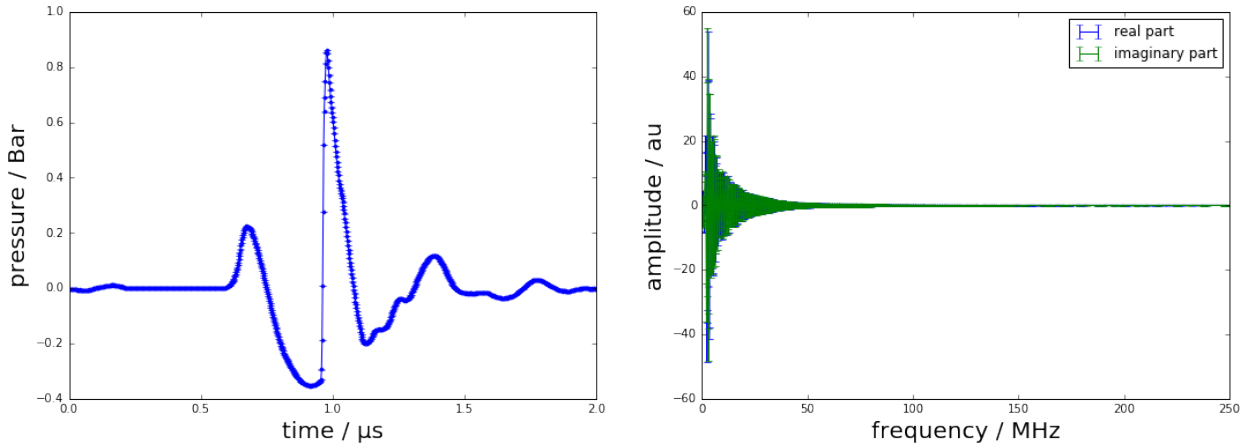
$$Y_k = \sum_{n=0}^{N-1} y_n \exp(-\mathrm{j}k\beta_n)$$

with $\beta_n = 2\pi n/N$, the uncertainty associated with the DFT outcome represented in terms of real and imaginary parts, is given by

$$U_Y = \begin{pmatrix} C_{\cos} U_y C_{\cos}^{\mathsf{T}} & C_{\cos} U_y C_{\sin}^{\mathsf{T}} \\ (C_{\cos} U_y C_{\sin}^{\mathsf{T}})^{\mathsf{T}} & C_{\sin} U_y C_{\sin}^{\mathsf{T}} \end{pmatrix}$$

```
Y,UY = GUM_DFT(y,Uy,N=Nf)
```

```
figure(figsize=(18,6))
subplot(121)
errorbar(time*1e6, y, sqrt(Uy)*ones_like(y),fmt=".-")
xlabel("time / µs",fontsize=20); ylabel("pressure / Bar",fontsize=20)
subplot(122)
errorbar(f*1e-6, Y[:len(f)],sqrt(UY[:len(f)]),label="real part")
errorbar(f*1e-6, Y[len(f):],sqrt(UY[len(f):]),label="imaginary part")
legend()
xlabel("frequency / MHz",fontsize=20); ylabel("amplitude / au",fontsize=20);
```



Uncertainties for measurement system w.r.t. real and imaginary parts

In practice, the frequency response of the measurement system is characterised in terms of its amplitude and phase values at a certain set of frequencies. GUM uncertainty evaluation, however, requires a representation by real and imaginary parts.

$$H_k = A_k \cos(P_k) + \mathrm{j} A_k \sin(P_k)$$

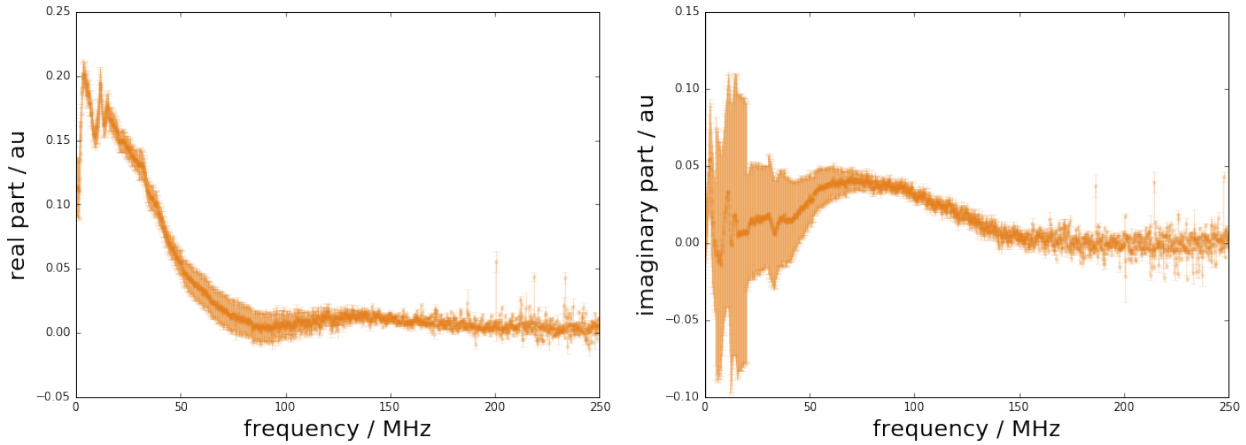
GUM uncertainty propagation

$$C_{RI} = \begin{pmatrix} R_A & R_P \\ I_A & I_P \end{pmatrix}.$$

$$U_H = C_{RI} \begin{pmatrix} U_{AA} & U_{AP} \\ U_{AP}^\mathsf{T} & U_{PP} \end{pmatrix} C_{RI}^\mathsf{T} = \begin{pmatrix} U_{11} & U_{12} \\ U_{21}^\mathsf{T} & U_{22} \end{pmatrix}.$$

```
H, UH = AmpPhase2DFT(np.abs(FR),np.angle(FR),UAP)
```

```
Nf = len(f)
figure(figsize=(18,6))
subplot(121)
errorbar(f*1e-6, H[:Nf], sqrt(diag(UH[:Nf,:Nf])),fmt=".-",color=colors[2],alpha=0.2)
xlabel("frequency / MHz",fontsize=20); ylabel("real part / au",fontsize=20)
subplot(122)
errorbar(f*1e-6, H[Nf:],sqrt(diag(UH[Nf:,Nf:])),fmt=".-",color=colors[2],alpha=0.2)
xlabel("frequency / MHz",fontsize=20); ylabel("imaginary part / au",fontsize=20);
```



Deconvolution in the frequency domain

The deconvolution problem can be decomposed into a division by the system's frequency response followed by a multiplication by a low-pass filter frequency response.

$$X(f) = \frac{Y(f)}{H(f)} H_L(f)$$

which in real and imaginary part becomes

$$X = \frac{(\Re_Y \Re_H + \Im_Y \Im_H) + \mathrm{j}(-\Re_Y \Im_H + \Im_Y \Re_H)}{\Re_H^2 + \Im_H^2} (\Re_{H_L} + \mathrm{j}\Im_{H_L})$$

Sensitivities for division part

$$R_{RY} = \frac{\partial \Re_X}{\partial \Re_Y} = \frac{\Re_H}{\Re_H^2 + \Im_H^2} \tag{6.6}$$

$$R_{IY} = \frac{\partial \Re_X}{\partial \Im_Y} = \frac{\Im_H}{\Re_H^2 + \Im_H^2} \tag{6.7}$$

$$R_{RH} = \frac{\partial \Re_X}{\partial \Re_H} = \frac{-\Re_Y \Re_H^2 + \Re_Y \Im_H^2 - 2\Im_Y \Im_H \Re_H}{(\Re_H^2 + \Im_H^2)^2} \tag{6.8}$$

$$R_{IH} = \frac{\partial \Re_X}{\partial \Im_H} = \frac{\Im_Y \Re_H^2 - \Im_Y \Im_H^2 - 2\Re_Y \Re_H \Im_H}{(\Re_H^2 + \Im_H^2)^2} \tag{6.9}$$

$$I_{RY} = \frac{\partial \Im_X}{\partial \Re_Y} = \frac{-\Im_H}{\Re_H^2 + \Im_H^2} \tag{6.10}$$

$$I_{IY} = \frac{\partial \Im_X}{\partial \Im_Y} = \frac{\Re_H}{\Re_H^2 + \Im_H^2} \tag{6.11}$$

$$I_{RH} = \frac{\partial \Im_X}{\partial \Re_H} = \frac{-\Im_Y \Re_H^2 + \Im_Y \Im_H^2 + 2\Re_Y \Im_H \Re_H}{(\Re_H^2 + \Im_H^2)^2} \tag{6.12}$$

$$I_{IH} = \frac{\partial \Im_X}{\partial \Im_H} = \frac{-\Re_Y \Re_H^2 + \Re_Y \Im_H^2 - 2\Im_Y \Re_H \Im_H}{(\Re_H^2 + \Im_H^2)^2} \tag{6.13}$$

Uncertainty blocks for multiplication part

$$U_{XRR} = \Re_{H_L} U_{ARR} \Re_{H_L} - \Im_{H_L} U_{ARI}^T \Re_{H_L} - \Re_{H_L} U_{ARI} \Im_{H_L} + \Im_{H_L} U_{AII} \Im_{H_L} \tag{6.14}$$

$$U_{XRI} = \Re_{H_L} U_{ARR} \Im_{H_L} - \Im_{H_L} U_{ARI}^T \Im_{H_L} + \Re_{H_L} U_{ARI} \Re_{H_L} - \Im_{H_L} U_{AII} \Re_{H_L} \tag{6.15}$$

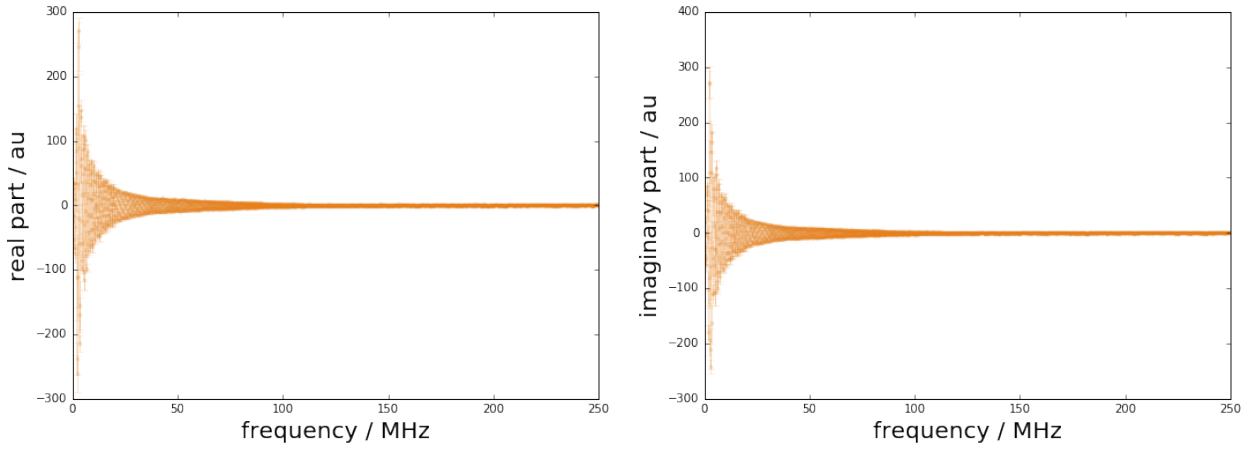
$$U_{XIR} = U_{YRI}^T \tag{6.16}$$

$$U_{XII} = \Im_{H_L} U_{ARR} \Im_{H_L} + \Re_{H_L} U_{ARI}^T \Im_{H_L} + \Im_{H_L} U_{ARI} \Re_{H_L} + \Re_{H_L} U_{AII} \Re_{H_L} \tag{6.17}$$

```
# low-pass filter for deconvolution
def lowpass(f,fcut=80e6):
    return 1/(1+1j*f/fcut)**2

HLc = lowpass(f)
HL = np.r_[np.real(HLc), np.imag(HLc)]
```

```
XH,UXH = DFT_deconv(H,Y,UH,UY)

XH, UXH = DFT_multiply(XH, UXH, HL)
```

```
figure(figsize=(18,6))
subplot(121)
errorbar(f*1e-6, XH[:Nf], sqrt(diag(UXH[:Nf,:Nf])),fmt=".-",color=colors[2],alpha=0.2)
xlabel("frequency / MHz",fontsize=20); ylabel("real part / au",fontsize=20)
subplot(122)
errorbar(f*1e-6, XH[Nf:],sqrt(diag(UXH[Nf:,Nf:])),fmt=".-",color=colors[2],alpha=0.2)
xlabel("frequency / MHz",fontsize=20); ylabel("imaginary part / au",fontsize=20);
```



Propagation from frequency to time domain

The inverse DFT equation is given by

$$X_n = \frac{1}{N} \sum_{k=0}^{N-1} (\Re_k \cos(k\beta_n) - \Im_k \sin(k\beta_n))$$

The sensitivities for the GUM propagation of uncertainties are then

$$\frac{\partial X_n}{\partial \Re_k} = \frac{1}{N} \qquad \text{for } k = 0 \tag{6.18}$$

$$\frac{\partial X_n}{\partial \Re_k} = \frac{2}{N}\cos(k\beta_n) \qquad \text{for } k = 1, \ldots, N/2-1 \tag{6.19}$$

$$\frac{\partial X_n}{\partial \Im_k} = 0 \qquad \text{for } k = 0 \tag{6.20}$$

$$\frac{\partial X_n}{\partial \Im_k} = -\frac{2}{N}\sin(k\beta_n) \qquad \text{for } k = 1, \ldots, N/2-1. \tag{6.21}$$

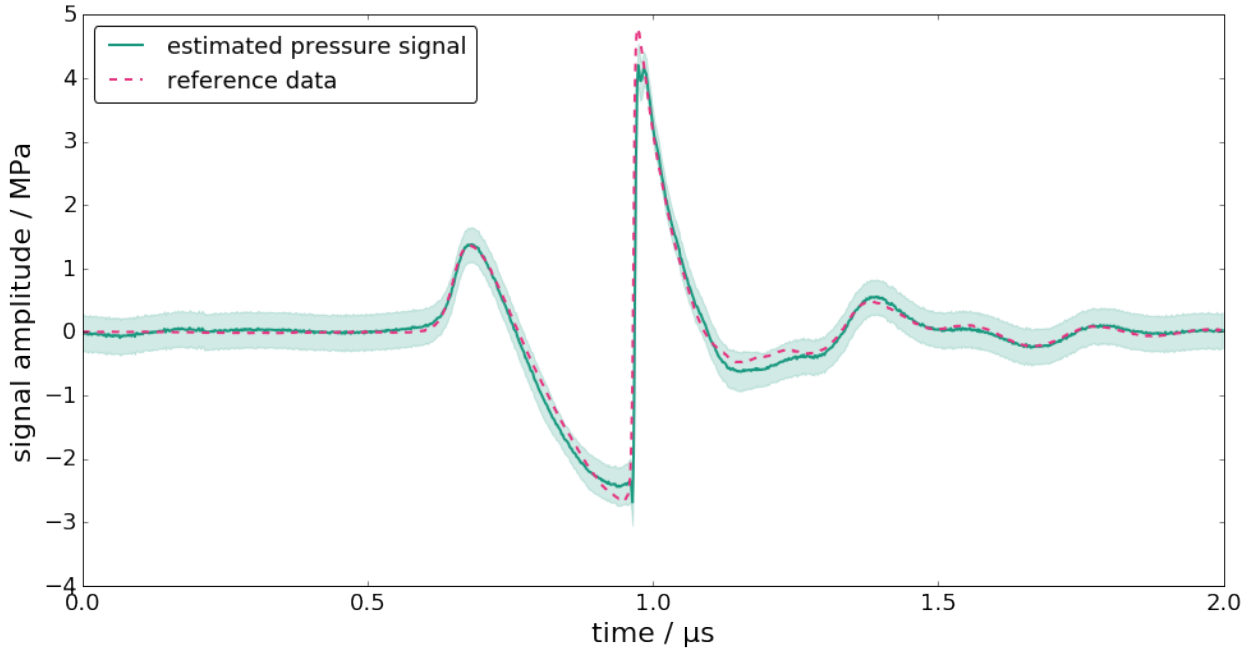
GUM uncertainty propagation for the inverse DFT

$$C_F U_F C_F^{\mathsf{T}} = \left(\widetilde{C}_{\cos}, \widetilde{C}_{\sin}\right) \begin{pmatrix} U_{RR} & U_{RI} \\ U_{IR} & U_{II} \end{pmatrix} \begin{pmatrix} \widetilde{C}_{\cos}^{\mathsf{T}} \\ \widetilde{C}_{\sin}^{\mathsf{T}} \end{pmatrix} \tag{6.22}$$

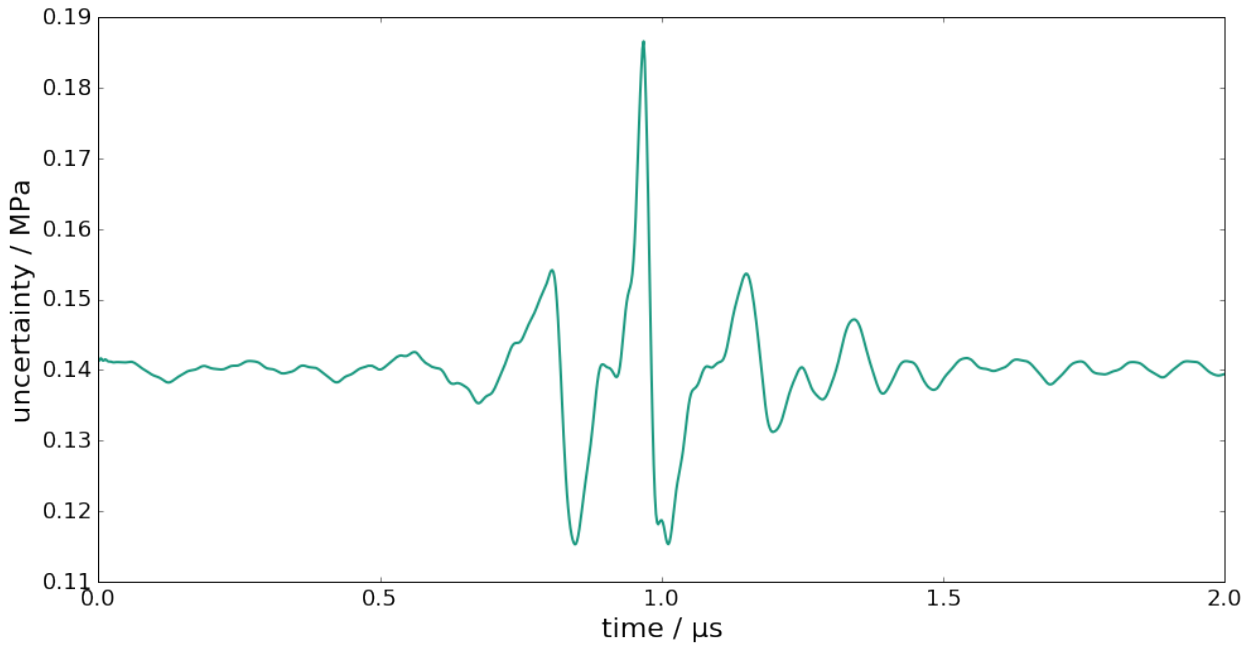
$$= \widetilde{C}_{\cos} U_{RR} \widetilde{C}_{\cos}^{\mathsf{T}} + 2\widetilde{C}_{\cos} U_{RI} \widetilde{C}_{\sin}^{\mathsf{T}} + \widetilde{C}_{\sin} U_{II} \widetilde{C}_{\sin}^{\mathsf{T}} \tag{6.23}$$

```
xh,Uxh = GUM_iDFT(XH,UXH,Nx=N)
```

```
ux = np.sqrt(np.diag(Uxh))

figure(figsize=(16,8))
plot(time*1e6,xh,label="estimated pressure signal",linewidth=2,color=colors[0])
plot(time * 1e6, ref_data, "--", label= "reference data", linewidth=2, color=colors[1])
fill_between(time * 1e6, xh + 2 * ux, xh - 2 * ux, alpha=0.2, color=colors[0])
xlabel("time / µs", fontsize=22)
ylabel("signal amplitude / MPa", fontsize=22)
tick_params(which= "major", labelsize=18)
legend(loc= "upper left", fontsize=18, fancybox=True)
xlim(0, 2);
```



```
figure(figsize=(16,8))
plot(time * 1e6, ux, label= "uncertainty", linewidth=2, color=colors[0])
xlabel("time / µs", fontsize=22)
ylabel("uncertainty / MPa", fontsize=22)
tick_params(which= "major", labelsize=18)
xlim(0,2);
```



Summary of PyDynamic workflow for deconvolution in DFT domain

```
Y,UY = GUM_DFT(y,Uy,N=Nf)

H, UH = AmpPhase2DFT(A, P, UAP)

XH,UXH = DFT_deconv(H,Y,UH,UY)

XH, UXH = DFT_multiply(XH, UXH, HL)
```

6.2.4 DFT and inverse DFT with PyDynamic - best practice guide

The discrete Fourier transform (DFT) and its inverse (iDFT) are common tools in dynamic metrology. For the corresponding propagation of uncertainties, *PyDynamic* implements the main tools required:

Uncertainty propagation for the discrete Fourier transform

```
GUM_DFT(x,Ux,N=None,window=None,CxCos=None,CxSin=None,returnC=False,mask=None)
```

Uncertainty propagation for the inverse discrete Fourier transform

```
GUM_iDFT(F,UF,Nx=None,Cc=None,Cs=None,returnC=False)
```

Uncertainty propagation for convolution in the frequency domain

```
DFT_multiply(Y, UY, F, UF=None)
```

Uncertainty propagation for deconvolution in the frequency domain

```
DFT_deconv(H, Y, UH, UY)
```

In the following we discuss common use cases for these methods and present guidance on how to utilize the optional arguments of the above methods.

Prerequisites

Get started with the notebook by importing some packages:

[1]:
```
# base imports
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.pyplot import plot
from numpy import (
    abs,
    arange,
    diag,
    empty,
    fft,
    full_like,
    mean,
    pi,
    random,
    round,
    sqrt,
    std,
    zeros_like,
)
from numpy.testing import assert_allclose
# convenience imports
from scipy.signal import butter, cheby2, freqs

from PyDynamic.misc.filterstuff import grpdelay
from PyDynamic.misc.testsignals import multi_sine
from PyDynamic.misc.tools import (
    complex_2_real_imag as c2ri,
    real_imag_2_complex as ri2c,
    shift_uncertainty,
)
from PyDynamic.uncertainty.propagate_DFT import (
    DFT_deconv,
    DFT_multiply,
    GUM_DFT,
    GUM_iDFT,
)

# set up matplotlib
%matplotlib inline
```

(continues on next page)

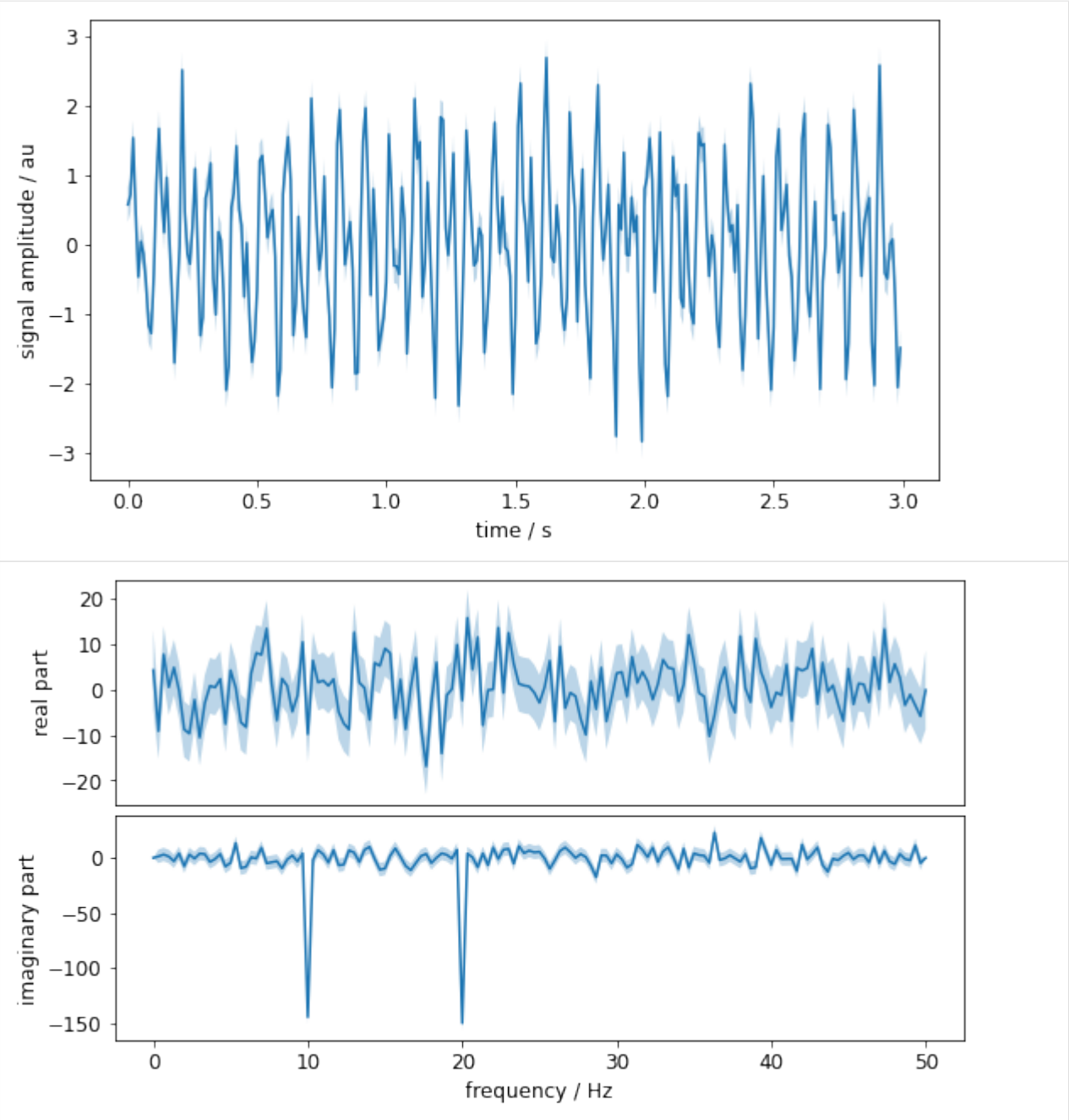
(continued from previous page)

```
# %matplotlib notebook
matplotlib.rc("font", size=12)
matplotlib.rc("figure", figsize=(9, 5))

# small helper functions for visualization
def get_amplitudes(V):
    return abs(ri2c(V))


def plot_unc(ax, time, values, values_unc, label="", **args):
    ax.plot(time, values, label=label, **args)
    ax.fill_between(time, values - values_unc, values + values_unc, alpha=0.3, **args)
    return ax
```

1) Discrete Fourier Transform (DFT)

The first and most basic scenario is the application of the discrete Fourier transform to analyse a time domain signal in the frequency domain.

[2]:
```
Fs = 100  # sampling frequency in Hz
Ts = 1 / Fs  # sampling interval in s
N = 300  # number of samples
time = arange(0, N * Ts, Ts)  # time instants
noise_std = 0.5  # signal noise standard deviation

# time domain signal
x = multi_sine(time, amps=[1.0, 1.0], freqs=[10, 20], noise=noise_std)
ux = full_like(x, noise_std ** 2)

X, UX = GUM_DFT(x, ux)  # application of DFT with propagation of uncertainties
f = fft.rfftfreq(N, Ts)  # frequency values

# plotting
_, ax = plt.subplots()
plot_unc(ax, time, x, ux)
ax.set_xlabel("time / s")
ax.set_ylabel("signal amplitude / au")

fig, ax = plt.subplots(2, 1)
plot_unc(ax[0], f, X[: len(f)], sqrt(diag(UX)[: len(f)]))
ax[0].set_ylabel("real part")
ax[0].set_xticks([])

plot_unc(ax[1], f, X[len(f) :], sqrt(diag(UX)[len(f) :]))
ax[1].set_ylabel("imaginary part")
ax[1].set_xlabel("frequency / Hz")

fig.subplots_adjust(hspace=0.05)
```



2) Inverse Discrete Fourier Transform (iDFT)

Let's transform our signal spectrum back into the time-domain. This yields an *exact* identity of `x` and `x_back_and_forth` and their assigned uncertainties. (Of course, this should be of no surprise, as we didn't modify anything and the DFT and iDFT form an identity pair.)

Note: In the plot the original signal is plotted with an offset of 0.1 to better visualize despite the identity.

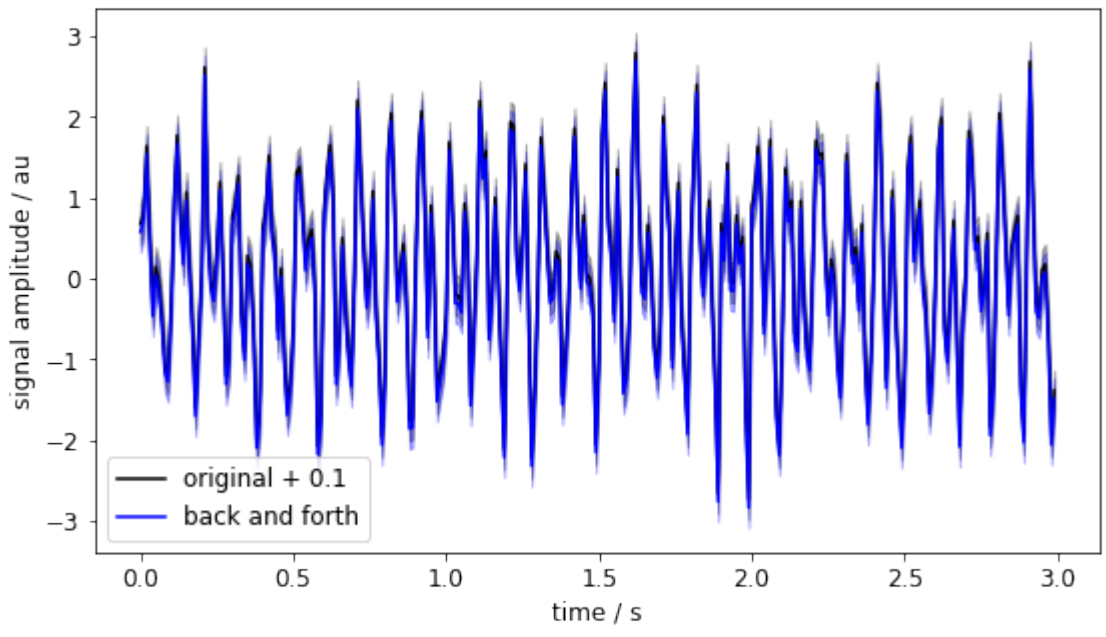
[3]:
```
# transform the frequency spectrum back to the time domain
x_back_and_forth, ux_back_and_forth = GUM_iDFT(X, UX)
```

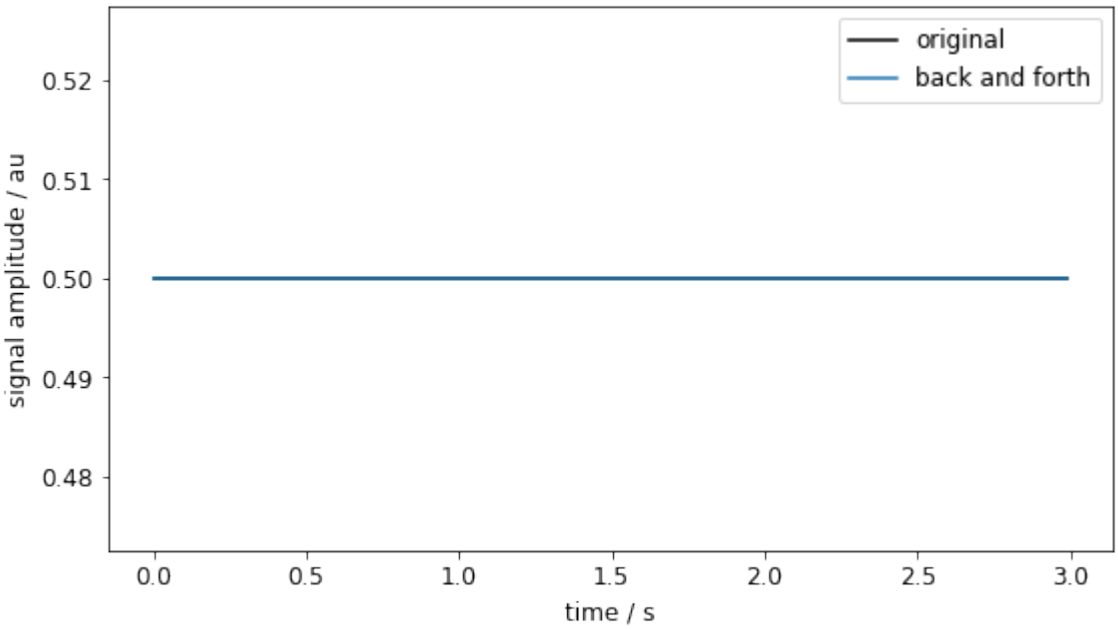
(continues on next page)

(continued from previous page)

```
# check numerical closeness
assert_allclose(x, x_back_and_forth)
assert_allclose(ux, diag(ux_back_and_forth))

# visualize the time-signal
_, ax = plt.subplots()
plot_unc(ax, time, x + 0.1, ux, label="original + 0.1", color="k")
plot_unc(
    ax,
    time,
    x_back_and_forth,
    diag(ux_back_and_forth),
    label="back and forth",
    color="b",
)
ax.set_xlabel("time / s")
ax.set_ylabel("signal amplitude / au")
ax.legend()

# visualize the uncertainties associated with x and x_back_and_forth
_, ax = plt.subplots()
ax.plot(time, sqrt(ux), "-k", label="original")
ax.plot(time, sqrt(diag(ux_back_and_forth)), label="back and forth")
ax.set_xlabel("time / s")
ax.set_ylabel("signal amplitude / au")
ax.legend()
```

[3]:
```
<matplotlib.legend.Legend at 0x7ffb54d6b550>
```





3) Multiply Spectra in the Frequency Domain

Multiplication in the frequency domain corresponds to a convolution of two signals in the time domain.

Let's consider again the signal `x` from above. We have already transformed it to the frequency domain in *section 1)*, which resulted in the spectrum `X` of the signal. We now want to apply a lowpass filter `H` that attenuates the highest of the both dominant frequencies. So we should design a filter such that it has a cutoff frequency around 15Hz.

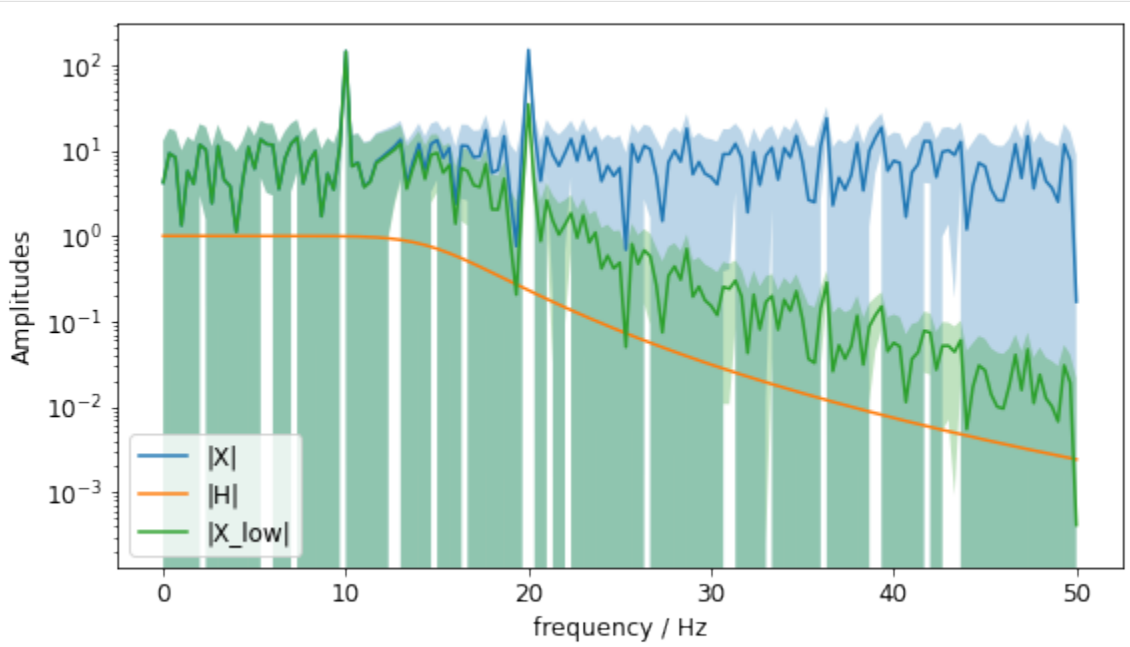
[4]:
```
cutoff_frequency = 15
b, a = butter(5, 2 * pi * cutoff_frequency, "low", analog=True)
_, H = freqs(
    b, a, worN=2 * pi * f
)  # get our filter at the same positions as X is already known

# bring H into the required shape for DFT_multiply
H = c2ri(H)

# apply the lowpass H to X
X_low, UX_low = DFT_multiply(X, H, UX)

# transform to time domain
x_low, ux_low = GUM_iDFT(X_low, UX_low)

# calculate group delay of filter to later adjust time signal for
d, _ = grpdelay(b, a, Fs=2 * pi * f[-1], nfft=1)
d = int(round(d))

# adjust x_low for group delay
x_low, u_low = shift_uncertainty(x_low, ux_low, d)

# visualize the multiplication by showing its effect on spectrum amplitudes
```

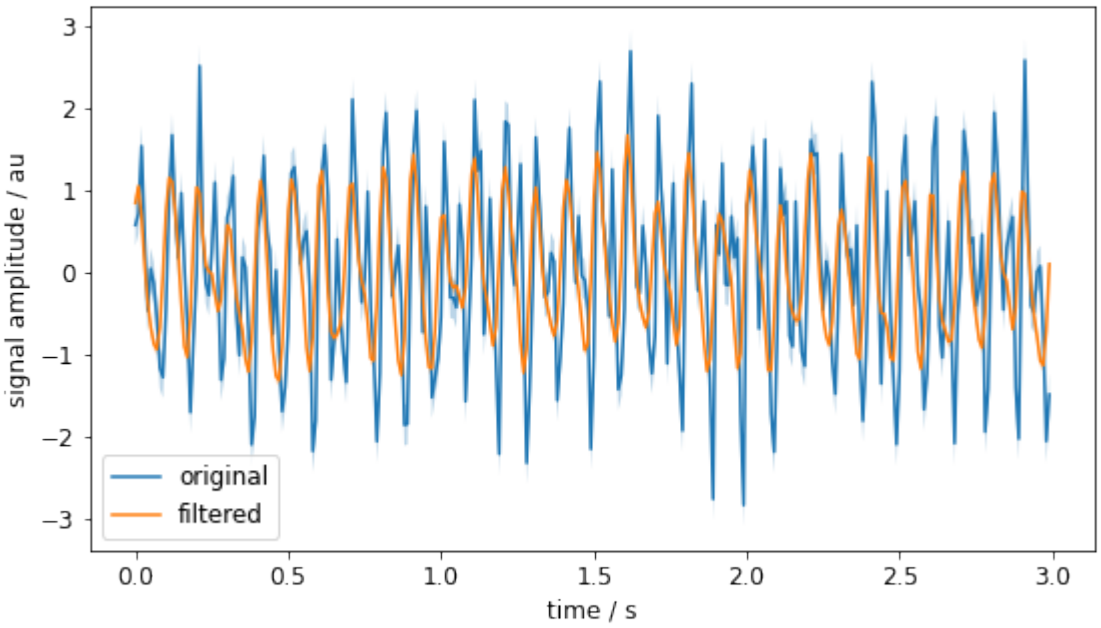
(continues on next page)

(continued from previous page)

```
_, ax = plt.subplots()
plot_unc(ax, f, get_amplitudes(X), get_amplitudes(sqrt(diag(UX))), label="|X|")
plot_unc(ax, f, get_amplitudes(H), zeros_like(f), label="|H|")
plot_unc(
    ax, f, get_amplitudes(X_low), get_amplitudes(sqrt(diag(UX_low))), label="|X_low|"
)
ax.set_xlabel("frequency / Hz")
ax.set_ylabel("Amplitudes")
ax.set_yscale("log")
ax.legend()

# visualize the time-domain by comparing the original `x` and lowpass-filtered `x_low`␣
↪signals
_, ax = plt.subplots()
plot_unc(ax, time, x, ux, label="original")
plot_unc(ax, time, x_low, diag(ux_low), label="filtered")
ax.set_xlabel("time / s")
ax.set_ylabel("signal amplitude / au")
ax.legend()
```

[4]: <matplotlib.legend.Legend at 0x7ffb54c88fa0>





4) Deconvolve Signals by Division of Spectra

It could be of interest to remove the effect of a system's transfer function on a signal. This can be achieved in the frequency domain by a simple division of the signal spectrum and the system's transfer function. This corresponds to a deconvolution in the time-domain (a.k.a. convolution with the inverse filter).

In our example, let's undo the lowpass operation from *section 3)*. But to make it a bit more interesting, let's assume, we don't know the exact cutoff-frequency but only with an uncertainty of 1Hz.

There are two steps:

1. Check the influence of the cutoff frequency on the actual frequency spectrum. This is done by a Monte Carlo method.
2. Division of both spectra: The reconstructed signal can then be transformed back to the time-domain, where the uncertainties of original and reconstruction are compared.

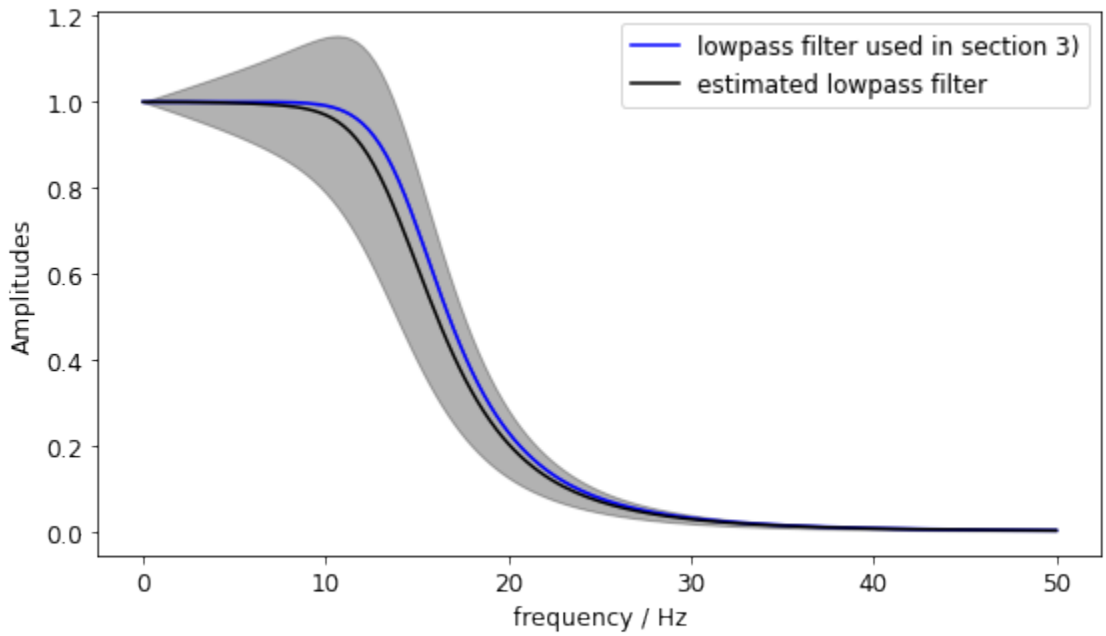
[5]:
```
cutoff_frequency_estimate = 14.5  # Hz
cutoff_frequency_unc = 1.0  # Hz

# step 1: Monte Carlo
mc_runs = 100
h_array = empty((mc_runs, len(f)), dtype="complex")
for i in range(mc_runs):
    cf = random.normal(cutoff_frequency_estimate, cutoff_frequency_unc)
    b, a = butter(5, 2 * pi * cf, "low", analog=True)
    _, H_tmp = freqs(b, a, worN=2 * pi * f)
    h_array[i, :] = H_tmp

H_estimated_lowpass = mean(c2ri(h_array), axis=0)
UH_estimated_lowpass = std(c2ri(h_array), axis=0)
```

(continues on next page)

(continued from previous page)

```
# visualize the uncertain lowpass filter
_, ax = plt.subplots()
plot_unc(
    ax,
    f,
    get_amplitudes(H),
    zeros_like(f),
    label="lowpass filter used in section 3)",
    color="b",
)
plot_unc(
    ax,
    f,
    get_amplitudes(H_estimated_lowpass),
    get_amplitudes(UH_estimated_lowpass),
    label="estimated lowpass filter",
    color="k",
)
ax.set_xlabel("frequency / Hz")
ax.set_ylabel("Amplitudes")
ax.set_yscale("linear")
ax.legend()
```

[5]:
```
<matplotlib.legend.Legend at 0x7ffb566651f0>
```



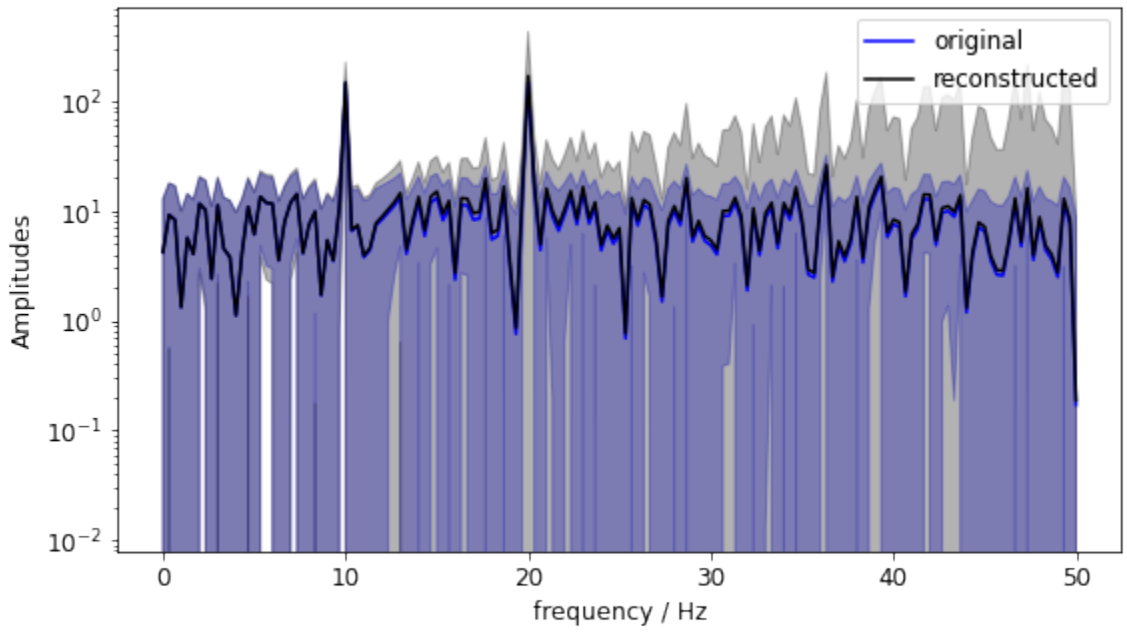
[6]:
```
# step 2
X_recon, UX_recon = DFT_deconv(
    H_estimated_lowpass, X_low, diag(UH_estimated_lowpass), UX_low
)

# visualize the uncertain reconstruction of the spectrum amplitudes
```

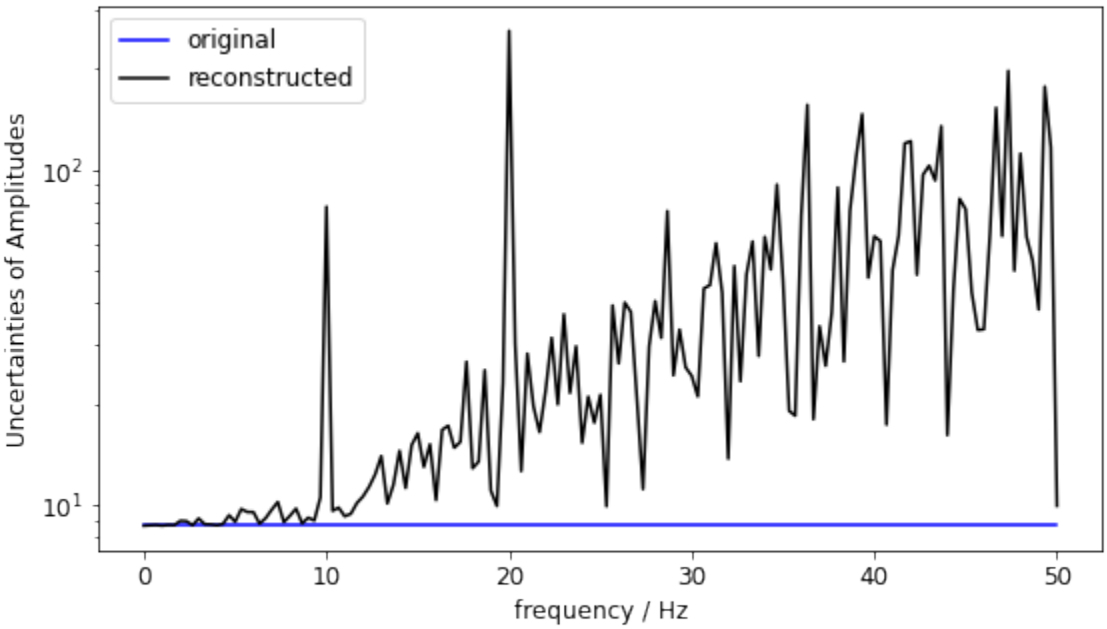
(continues on next page)

(continued from previous page)

```
_, ax = plt.subplots()
plot_unc(
    ax,
    f,
    get_amplitudes(X),
    get_amplitudes(sqrt(diag(UX))),
    label="original",
    color="b",
)
plot_unc(
    ax,
    f,
    get_amplitudes(X_recon),
    get_amplitudes(sqrt(diag(UX_recon))),
    label="reconstructed",
    color="k",
)
ax.set_xlabel("frequency / Hz")
ax.set_ylabel("Amplitudes")
ax.set_yscale("log")
ax.legend()

# visualize the uncertain reconstruction of the spectrum amplitude uncertainties
_, ax = plt.subplots()
plot(f, get_amplitudes(sqrt(diag(UX))), "b", label="original")
plot(f, get_amplitudes(sqrt(diag(UX_recon))), "k", label="reconstructed")
ax.set_xlabel("frequency / Hz")
ax.set_ylabel("Uncertainties of Amplitudes")
ax.set_yscale("log")
ax.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x7ffb56775100>
```





Note: We just divided by the lowpass' transfer function. To readers with some background in control theory it should be obvious that this amplifies high frequencies and might lead to instabilities if used inside a control loop. However, because we are taking the signal's and filter's uncertainty into account, the reconstructed spectrum shows us just that - *that the higher spectrum values are much more uncertain.*

5) Exemplary Regularization

To counter the effect seen in the last section, it is common to introduce some kind of regularization. E.g. in the reconstruction above, we may have the additional information about our input signal that all the interesting parts are below 25Hz and only noise is expected above. It is therefore safe to overlay the reconstructed signal with an additional (deterministic) low-pass with its cutoff-frequency at 35Hz.

For further information on this topic, please refer to e.g. Eichstädt & Wilkens (2017)[164]

[7]:
```
# define a filter to remove high frequencies of reconstructed signal
b_reg, a_reg = cheby2(5, 40, 2 * pi * 35, "low", analog=True)
_, H_reg = freqs(b_reg, a_reg, worN=2 * pi * f)
H_reg = c2ri(H_reg)

# apply regularization to reconstructed signal
X_reg, UX_reg = DFT_multiply(X_recon, H_reg, UX_recon)

# visualize and compare the regularization of the spectrum amplitudes
_, ax = plt.subplots()
plot_unc(
    ax,
    f,
    get_amplitudes(X),
    get_amplitudes(sqrt(diag(UX))),
    label="original",
```

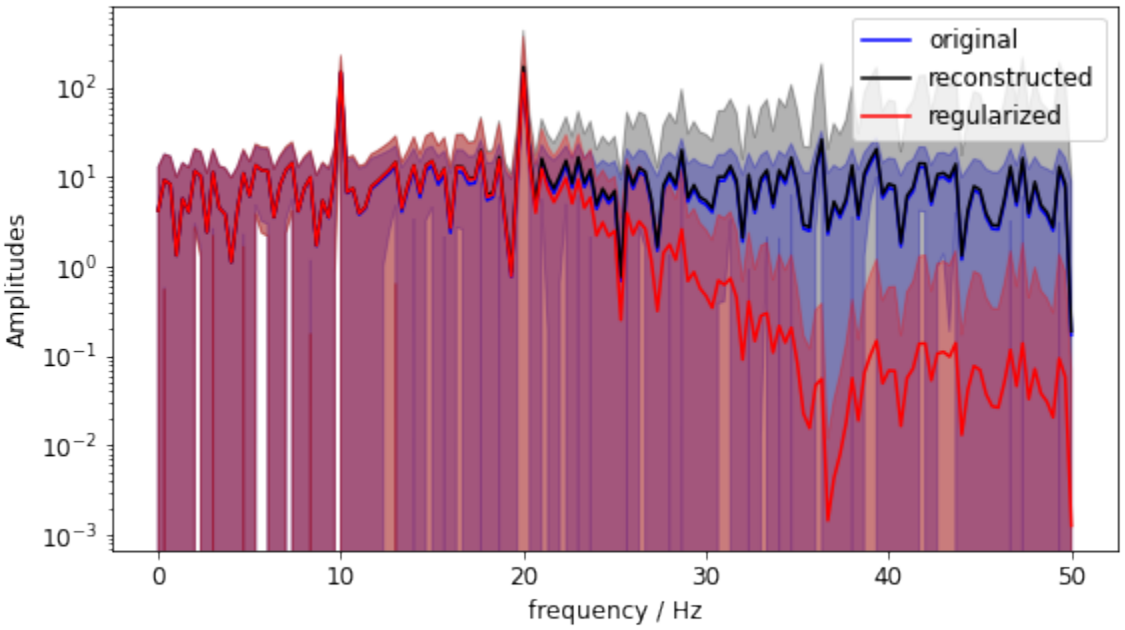
(continues on next page)

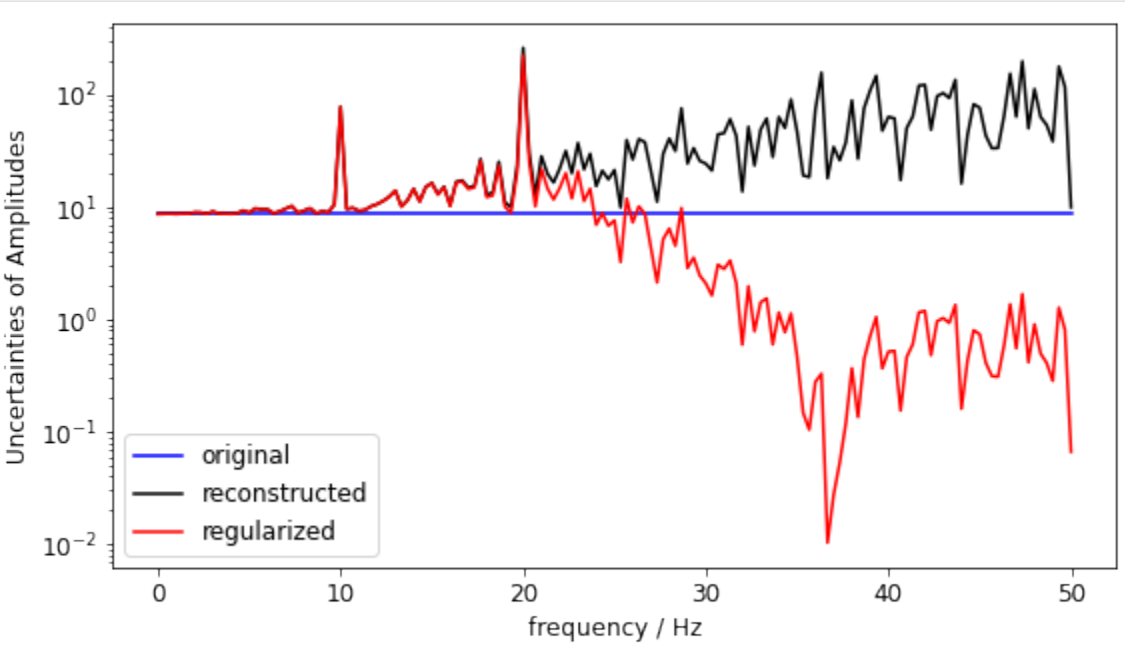
[164] https://doi.org/10.1121/1.4983827

(continued from previous page)

```
    color="b",
)
plot_unc(
    ax,
    f,
    get_amplitudes(X_recon),
    get_amplitudes(sqrt(diag(UX_recon))),
    label="reconstructed",
    color="k",
)
plot_unc(
    ax,
    f,
    get_amplitudes(X_reg),
    get_amplitudes(sqrt(diag(UX_reg))),
    label="regularized",
    color="r",
)
ax.set_xlabel("frequency / Hz")
ax.set_ylabel("Amplitudes")
ax.set_yscale("log")
ax.legend()


# visualize and compare the regularization of the spectrum amplitudes
_, ax = plt.subplots()
plot(f, get_amplitudes(sqrt(diag(UX))), "b", label="original")
plot(f, get_amplitudes(sqrt(diag(UX_recon))), "k", label="reconstructed")
plot(f, get_amplitudes(sqrt(diag(UX_reg))), "r", label="regularized")
ax.set_xlabel("frequency / Hz")
ax.set_ylabel("Uncertainties of Amplitudes")
ax.set_yscale("log")
ax.legend()
```

[7]:
```
<matplotlib.legend.Legend at 0x7ffb548503a0>
```





[1]:
```
%pylab inline
```

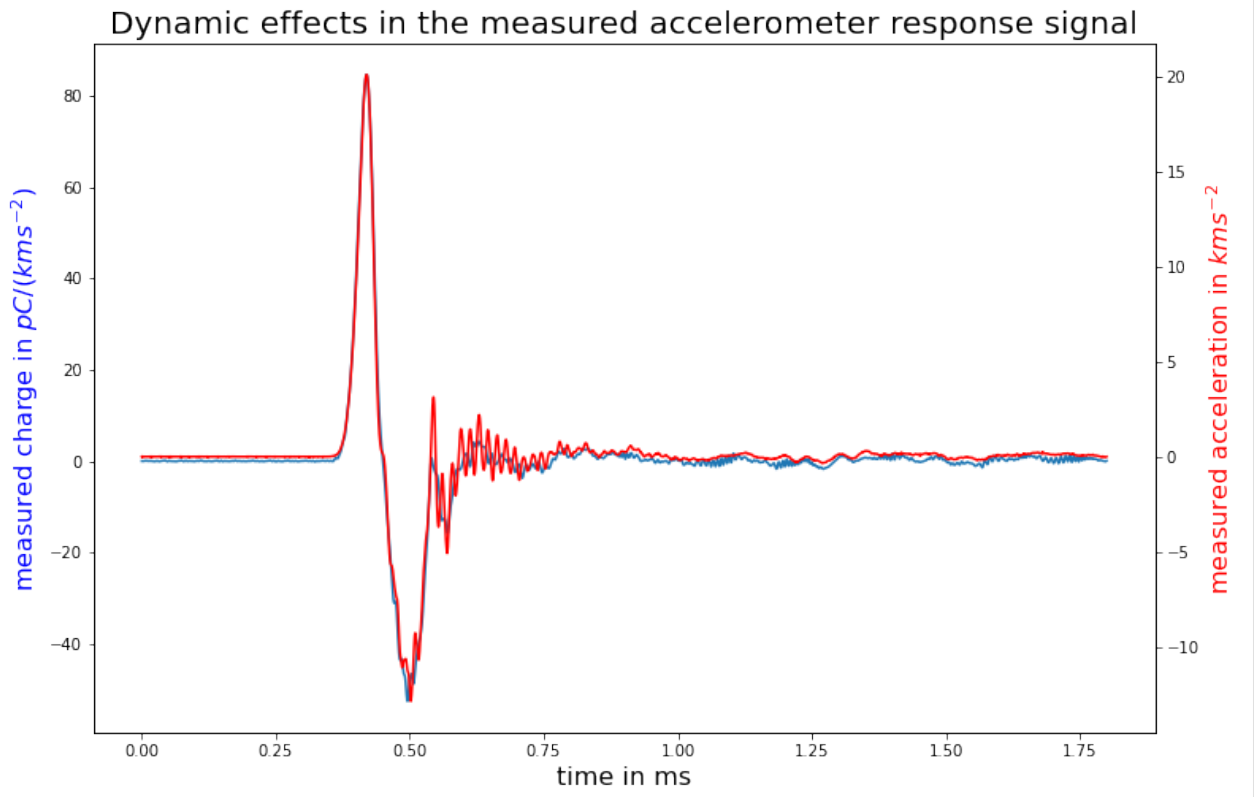
```
Populating the interactive namespace from numpy and matplotlib
```

6.2.5 Input estimation for shock acceleration

[2]:
```
Ts = 1e-7
Fs = 1 / Ts
```

[3]:
```
measured_input = loadtxt("measured_input_accel.txt") * 1e3
measured_output = loadtxt("measured_output_accel.txt") * 1e3
time = arange(0, len(measured_input - 1) * Ts, Ts)
```

[4]:
```
figure(figsize=(12, 8))
plot(time * 1e3, measured_input, label="measured input signal")
ylabel(r"measured charge in $pC/(kms^{-2})$", fontsize=16, color="b")
xlabel("time in ms", fontsize=16)
ax2 = gca().twinx()
ax2.plot(time * 1e3, measured_output, "r", label="measured output signal")
ax2.set_ylabel(r"measured acceleration in $kms^{-2}$", fontsize=16, color="r")
title("Dynamic effects in the measured accelerometer response signal", fontsize=20);
```



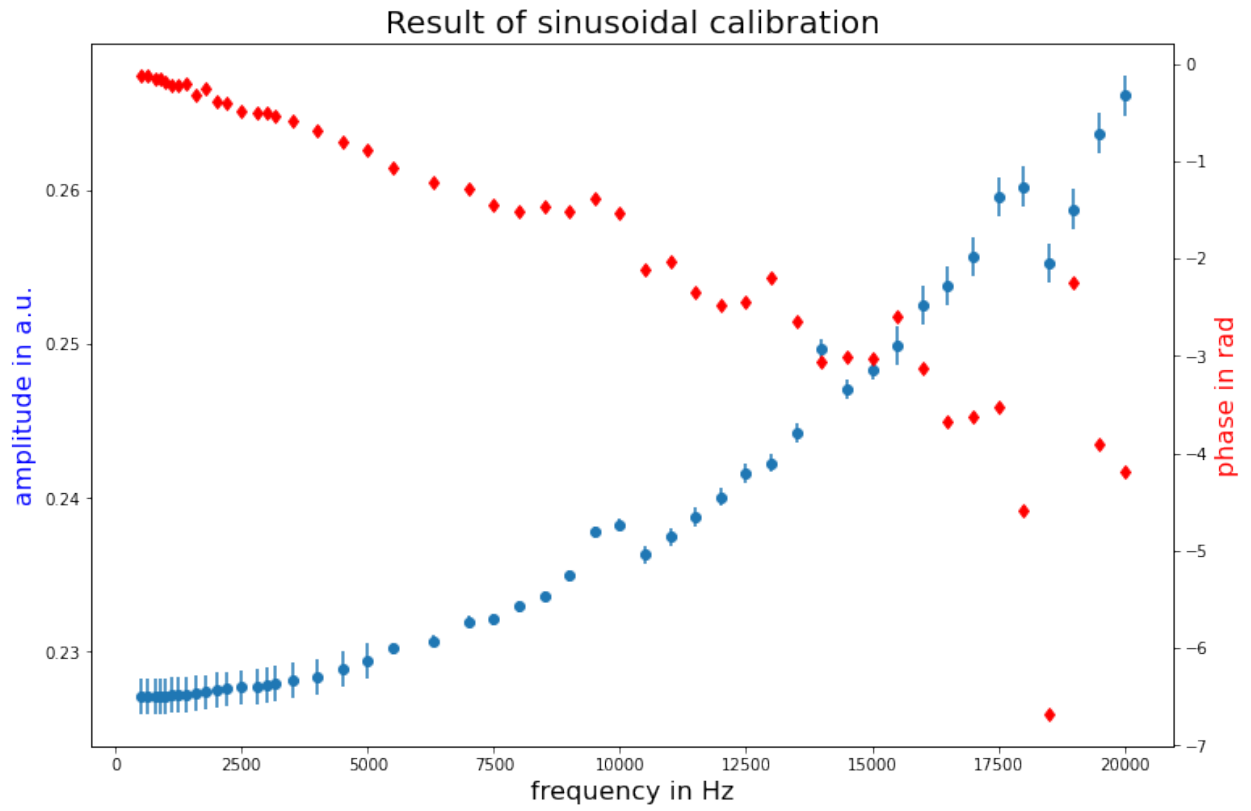
[5]:
```
sinusoidal_calib_results = loadtxt("sinusoidal_calibration_values.txt")
frequencies = sinusoidal_calib_results[:, 0]
abs_values = sinusoidal_calib_results[:, 1]
unc_abs = r_[
    abs_values[(frequencies <= 5000)] * 0.005,
    abs_values[(5001 <= frequencies) & (frequencies <= 10000)] * 0.0015,
    abs_values[(10001 <= frequencies) & (frequencies <= 15000)] * 0.0025,
    abs_values[(15001 <= frequencies) & (frequencies <= 20000)] * 0.005,
```

(continues on next page)

(continued from previous page)

```
]
phase_values = sinusoidal_calib_results[:, 2]
unc_phase = (
    r_[
        ones(len(frequencies[frequencies <= 5000])) * 0.25,
        ones(len(frequencies[frequencies > 5000])) * 0.5,
    ]
    * pi
    / 180
)
```

[6]:
```
figure(figsize=(12, 8))
errorbar(frequencies, abs_values, unc_abs, fmt="o")
xlabel("frequency in Hz", fontsize=16)
ylabel("amplitude in a.u.", fontsize=16, color="b")
title("Result of sinusoidal calibration", fontsize=20)
ax2 = gca().twinx()
ax2.errorbar(frequencies, phase_values, unc_phase, fmt="d", color="r")
ax2.set_ylabel("phase in rad", fontsize=16, color="r");
```



[3]:
```
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal as dsp

rst = np.random.RandomState(1)
```

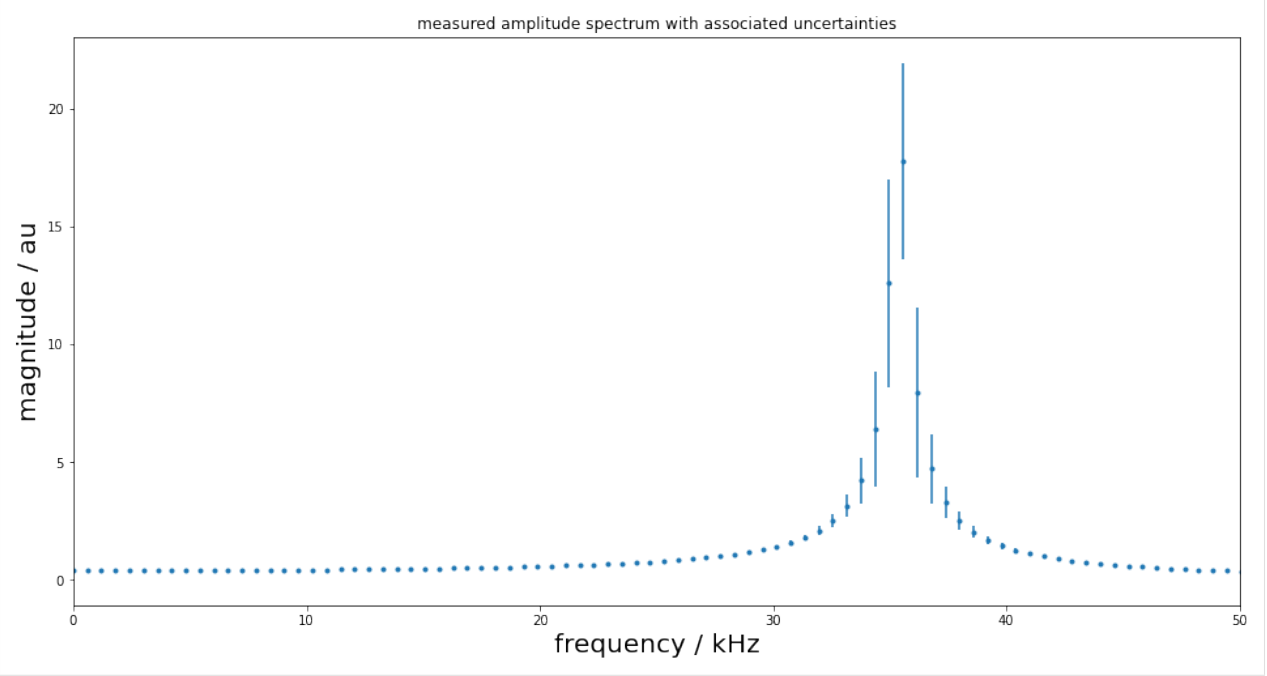
[4]:
```
from PyDynamic.model_estimation.fit_filter import LSFIR
from PyDynamic.uncertainty.propagate_filter import FIRuncFilter
from PyDynamic.misc.SecondOrderSystem import *
from PyDynamic.misc.filterstuff import kaiser_lowpass
from PyDynamic.misc.tools import make_semiposdef
```

6.2.6 Design of a digital deconvolution filter (FIR type)

[5]:
```
# parameters of simulated measurement
Fs = 500e3
Ts = 1 / Fs

# sensor/measurement system
f0 = 36e3
uf0 = 0.01 * f0
S0 = 0.4
uS0 = 0.001 * S0
delta = 0.01
udelta = 0.1 * delta

# transform continuous system to digital filter
bc, ac = sos_phys2filter(S0, delta, f0)
b, a = dsp.bilinear(bc, ac, Fs)

# Monte Carlo for calculation of unc. assoc. with [real(H),imag(H)]
f = np.linspace(0, 120e3, 200)
Hfc = sos_FreqResp(S0, delta, f0, f)
Hf = dsp.freqz(b, a, 2 * np.pi * f / Fs)[1]

runs = 10000
MCS0 = S0 + rst.randn(runs) * uS0
MCd = delta + rst.randn(runs) * udelta
MCf0 = f0 + rst.randn(runs) * uf0
HMC = np.zeros((runs, len(f)), dtype=complex)
for k in range(runs):
    bc_, ac_ = sos_phys2filter(MCS0[k], MCd[k], MCf0[k])
    b_, a_ = dsp.bilinear(bc_, ac_, Fs)
    HMC[k, :] = dsp.freqz(b_, a_, 2 * np.pi * f / Fs)[1]

H = np.r_[np.real(Hf), np.imag(Hf)]
uAbs = np.std(np.abs(HMC), axis=0)
uPhas = np.std(np.angle(HMC), axis=0)
UH = np.cov(np.hstack((np.real(HMC), np.imag(HMC))), rowvar=0)
UH = make_semiposdef(UH)
```

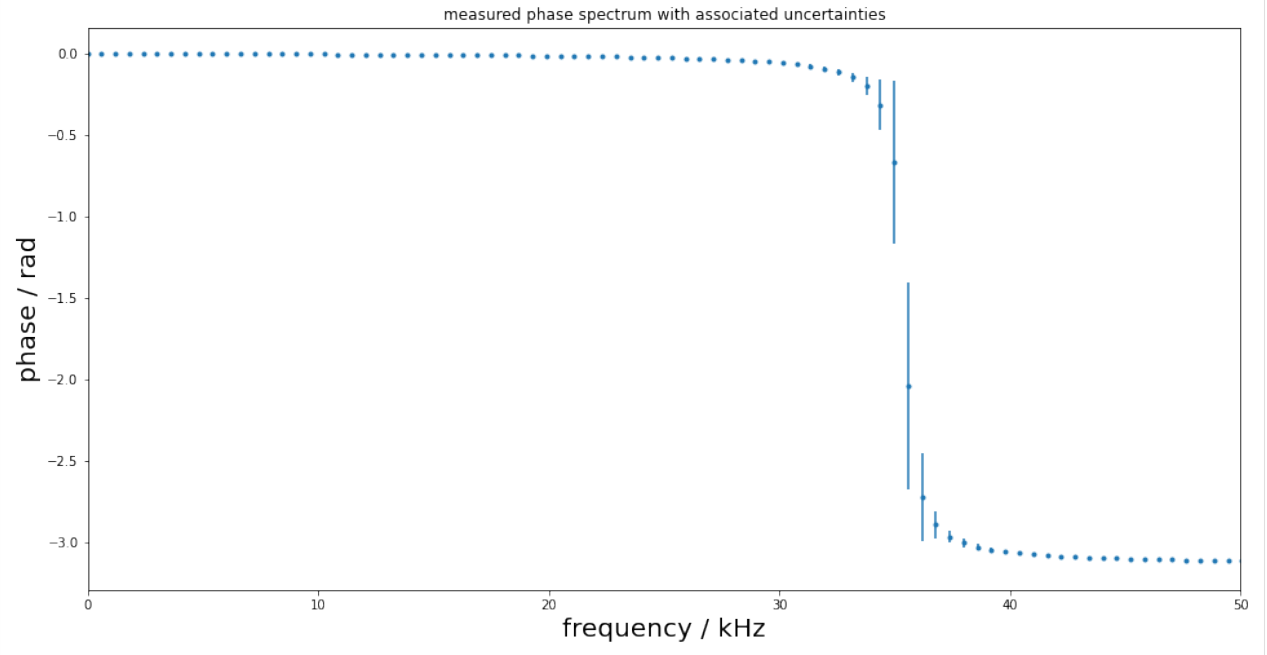
[6]:
```
plt.figure(figsize=(16, 8))
plt.errorbar(f * 1e-3, np.abs(Hf), uAbs, fmt=".")
plt.title("measured amplitude spectrum with associated uncertainties")
plt.xlim(0, 50)
plt.xlabel("frequency / kHz", fontsize=20)
plt.ylabel("magnitude / au", fontsize=20)
```

[6]:
```
Text(0, 0.5, 'magnitude / au')
```



[7]:
```
plt.figure(figsize=(16, 8))
plt.errorbar(f * 1e-3, np.angle(Hf), uPhas, fmt=".")
plt.title("measured phase spectrum with associated uncertainties")
plt.xlim(0, 50)
plt.xlabel("frequency / kHz", fontsize=20)
plt.ylabel("phase / rad", fontsize=20)
```

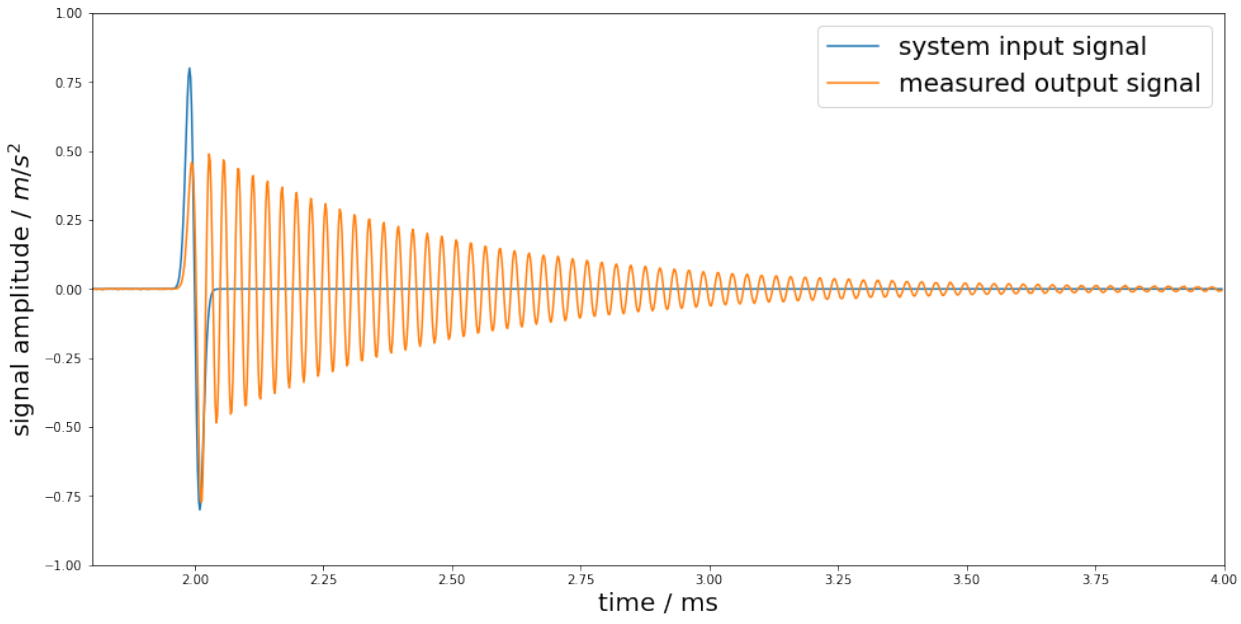
[7]:
```
Text(0, 0.5, 'phase / rad')
```



[8]:
```
# simulate input and output signals
time = np.arange(0, 4e-3 - Ts, Ts)
# x = shocklikeGaussian(time, t0 = 2e-3, sigma = 1e-5, m0=0.8)
m0 = 0.8
sigma = 1e-5
t0 = 2e-3
x = (
    -m0
    * (time - t0)
    / sigma
    * np.exp(0.5)
    * np.exp(-((time - t0) ** 2) / (2 * sigma ** 2))
)
y = dsp.lfilter(b, a, x)
noise = 1e-3
yn = y + rst.randn(np.size(y)) * noise
```

[9]:
```
plt.figure(figsize=(16, 8))
plt.plot(time * 1e3, x, label="system input signal")
plt.plot(time * 1e3, yn, label="measured output signal")
plt.legend(fontsize=20)
plt.xlim(1.8, 4)
plt.ylim(-1, 1)
plt.xlabel("time / ms", fontsize=20)
plt.ylabel(r"signal amplitude / $m/s^2$", fontsize=20)
```

[9]:
```
Text(0, 0.5, 'signal amplitude / $m/s^2$')
```

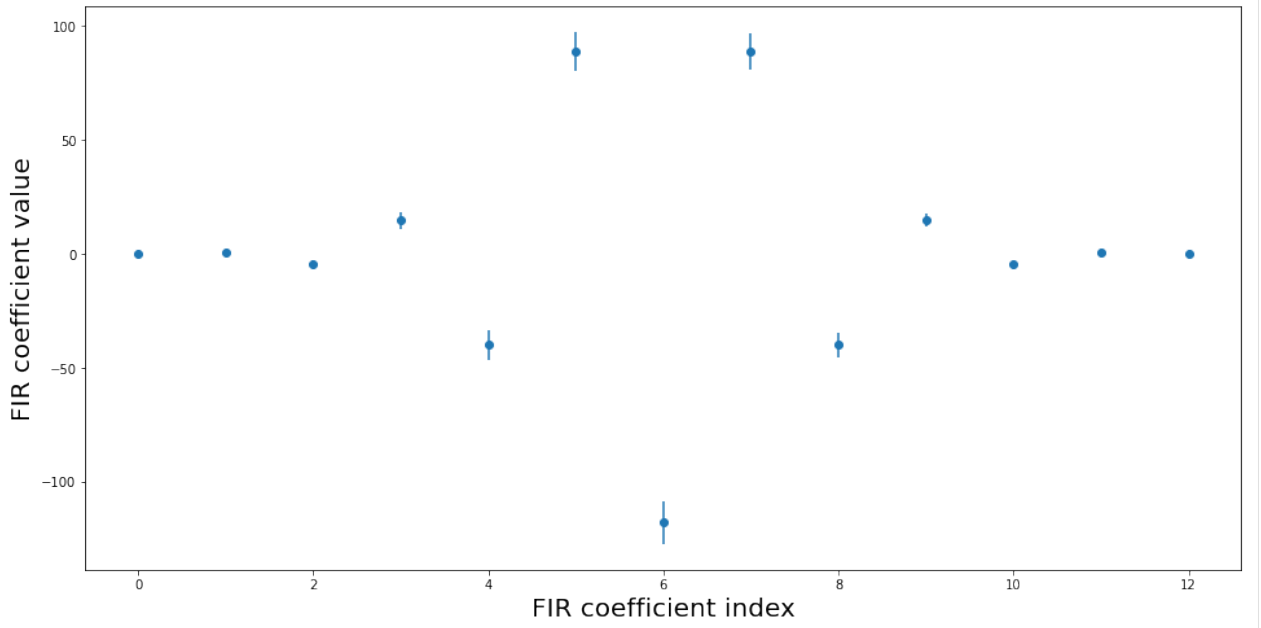


[10]:
```
# Calculation of FIR deconvolution filter and its assoc. unc.
N = 12
tau = N // 2
bF, UbF = LSFIR(H, N, f, Fs, tau, inv=True, UH=UH)
```

```
LSFIR: Least-squares fit of an order 12 digital FIR filter to the reciprocal of a␣
↪frequency response given by 400 values. The frequency response's associated␣
↪uncertainties are propagated via a truncated singular-value decomposition and linear␣
↪matrix propagation with None as lower bound for the singular values to be considered␣
↪for the pseudo-inverse.
LSFIR: Calculation of filter coefficients finished. Final rms error = 0.
↪0001934613524619455
```

[11]:
```
plt.figure(figsize=(16, 8))
plt.errorbar(range(N + 1), bF, np.sqrt(np.diag(UbF)), fmt="o")
plt.xlabel("FIR coefficient index", fontsize=20)
plt.ylabel("FIR coefficient value", fontsize=20)
```

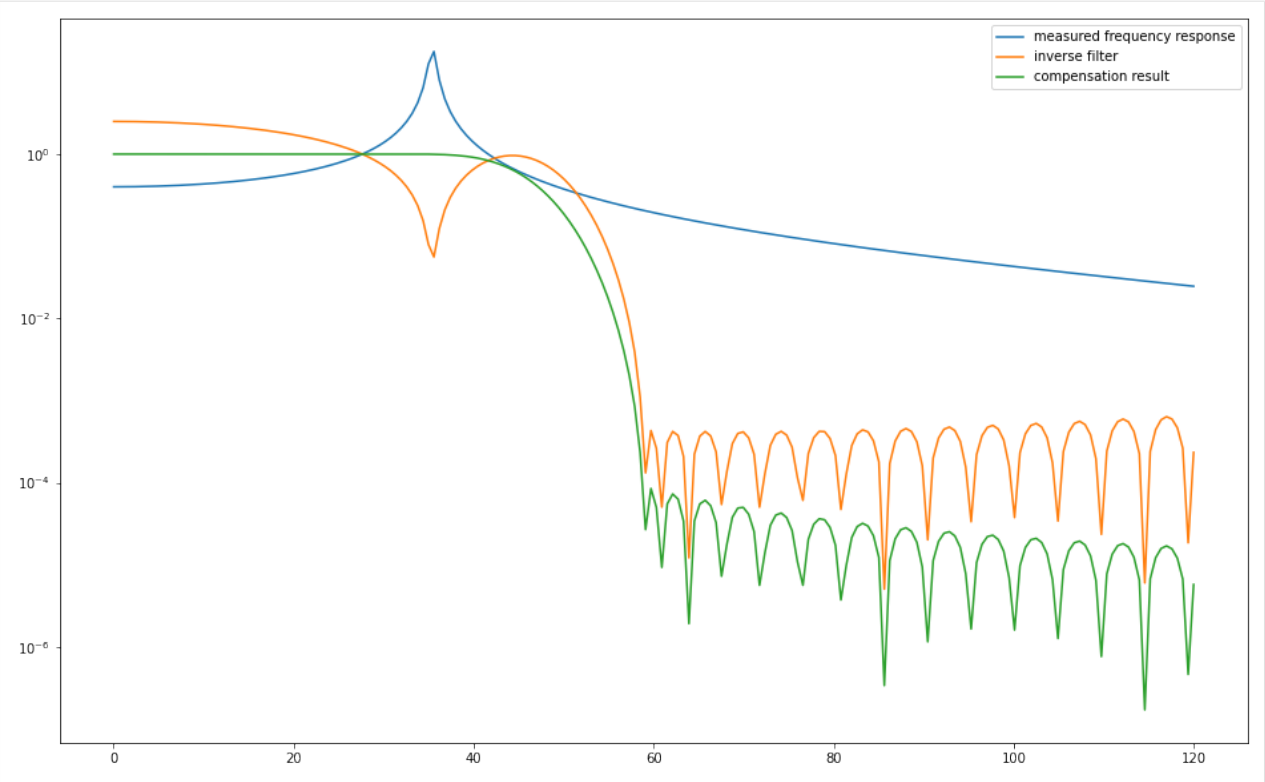
[11]:
```
Text(0, 0.5, 'FIR coefficient value')
```



[12]:
```
fcut = f0 + 10e3
low_order = 100
blow, lshift = kaiser_lowpass(low_order, fcut, Fs)
shift = tau + lshift
```

[13]:
```
plt.figure(figsize=(16, 10))
HbF = (
    dsp.freqz(bF, 1, 2 * np.pi * f / Fs)[1] * dsp.freqz(blow, 1, 2 * np.pi * f / Fs)[1]
)
plt.semilogy(f * 1e-3, np.abs(Hf), label="measured frequency response")
plt.semilogy(f * 1e-3, np.abs(HbF), label="inverse filter")
plt.semilogy(f * 1e-3, np.abs(Hf * HbF), label="compensation result")
plt.legend()
```

[13]:
```
<matplotlib.legend.Legend at 0x7f98bbcbe8b0>
```

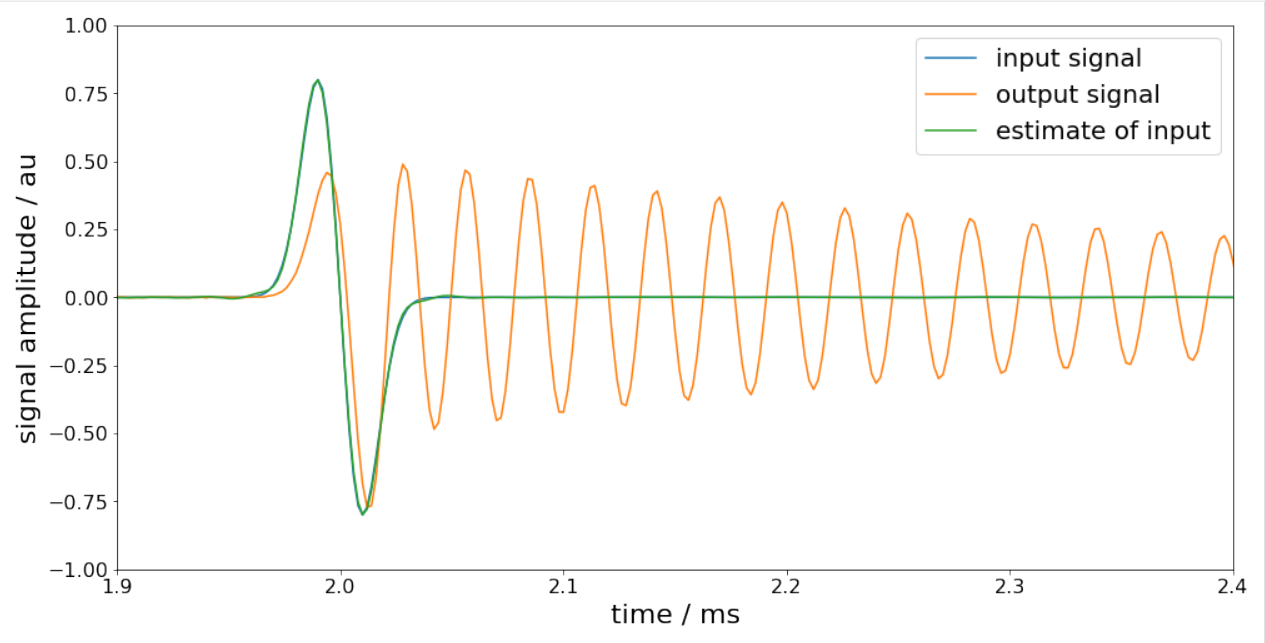


[14]:
```
xhat, Uxhat = FIRuncFilter(yn, noise, bF, UbF, shift, blow)
```

```
FIRuncFilter: Output uncertainty will be given as 1D-array of point-wise standard␣
↪uncertainties. Although this requires significantly lesser computations, it ignores␣
↪correlation information. Every FIR-filtered signal will have off-diagonal entries in␣
↪its covariance matrix (assuming the filter is longer than 1). To get the full output␣
↪covariance matrix, use 'return_full_covariance=True'.
```

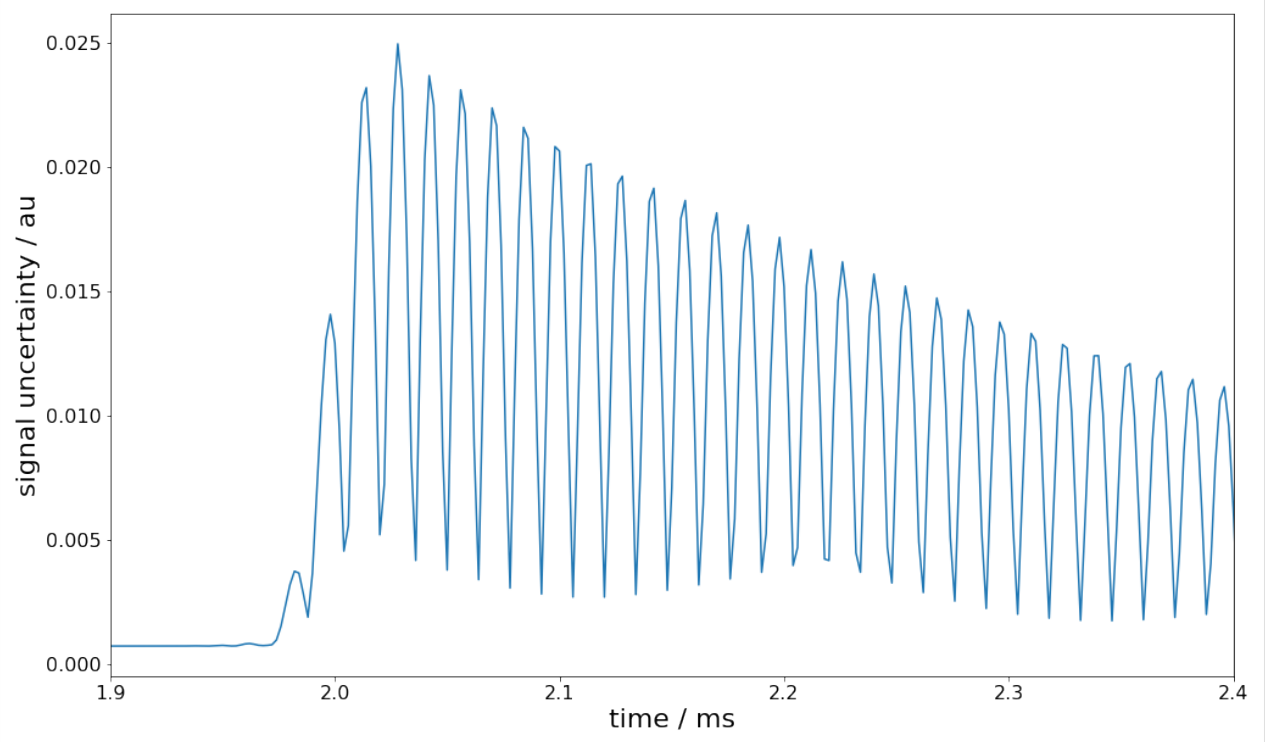
[15]:
```
plt.figure(figsize=(16, 8))
plt.plot(time * 1e3, x, label="input signal")
plt.plot(time * 1e3, yn, label="output signal")
plt.plot(time * 1e3, xhat, label="estimate of input")
plt.legend(fontsize=20)
plt.xlabel("time / ms", fontsize=22)
plt.ylabel("signal amplitude / au", fontsize=22)
plt.tick_params(which="both", labelsize=16)
plt.xlim(1.9, 2.4)
plt.ylim(-1, 1)
```

[15]:
```
(-1.0, 1.0)
```



[16]:
```
plt.figure(figsize=(16, 10))
plt.plot(time * 1e3, Uxhat)
plt.xlabel("time / ms", fontsize=22)
plt.ylabel("signal uncertainty / au", fontsize=22)
plt.subplots_adjust(left=0.15, right=0.95)
plt.tick_params(which="both", labelsize=16)
plt.xlim(1.9, 2.4)
```

[16]:
```
(1.9, 2.4)
```



CHAPTER

SEVEN

GET ASSISTANCE IN USING PYDYNAMIC

We prepared a collection of tutorials and examples to document, explain and illustrate the possibilities offered by *PyDynamic*. We will add more and more examples over time, especially those that are currently included in PyDynamic's codebase subfolder examples[165].

The following links take you to the webpages containing the tutorials' documentation.

7.1 Getting started with the tutorials

- Start a notebook server remotely or locally[166]

7.2 Deconvolution

1. Basic measurement data pre-processing[167]
2. Preparation of calibration data[168]
3. Interpolation and extrapolation of calibration data[169]
4. Calculation of impulse response of hydrophone[170]
5. Deconvolution in the frequency domain[171]
6. Regularized deconvolution[172]

[165] https://github.com/PTB-M4D/PyDynamic/tree/master/src/PyDynamic/examples
[166] https://pydynamic-tutorials.readthedocs.io/en/latest/README.html
[167] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/deconvolution/01%20Basic%20measurement%20data%20pre-processing.html
[168] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/deconvolution/02%20Preparation%20of%20calibration%20data.html
[169] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/deconvolution/03%20Interpolation%20and%20extrapolation%20of%20calibration%20data.html
[170] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/deconvolution/04%20Calculation%20of%20impulse%20response%20of%20hydrophone.html
[171] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/deconvolution/05%20Deconvolution%20in%20the%20frequency%20domain.html
[172] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/deconvolution/06%20Regularized%20deconvolution.html

7.3 Uncertainty

These tutorials introduce PyDynamic's capabilities of processing time-series with the propagation of associated measurement uncertainties.

1. Basic measurement data pre-processing[173]
2. Basic interpolation[174]
3. Basic extrapolation[175]

[173] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/uncertainty/01%20Basic%20measurement%20data%20pre-processing.html
[174] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/uncertainty/02%20Basic%20interpolation.html
[175] https://pydynamic-tutorials.readthedocs.io/en/latest/PyDynamic_tutorials/uncertainty/03%20Basic%20extrapolation.html

CHAPTER

EIGHT

EVALUATION OF UNCERTAINTIES

The evaluation of uncertainties is a fundamental part of the measurement analysis in metrology. The analysis of dynamic measurements typically involves methods from signal processing, such as digital filtering, the discrete Fourier transform (DFT), or simple tasks like interpolation. For most of these tasks, methods are readily available, for instance, as part of `scipy.signal`[176]. This module of PyDynamic provides the corresponding methods for the evaluation of uncertainties.

The package consists of the following modules:

- *PyDynamic.uncertainty.propagate_DFT*: Uncertainty evaluation for the DFT
- *PyDynamic.uncertainty.propagate_DWT*: Uncertainty evaluation for the DWT
- *PyDynamic.uncertainty.propagate_convolution*: Uncertainty evaluation for convolutions
- *PyDynamic.uncertainty.propagate_filter*: Uncertainty evaluation for digital filtering
- *PyDynamic.uncertainty.propagate_MonteCarlo*: Monte Carlo methods for digital filtering
- *PyDynamic.uncertainty.interpolate*: Uncertainty evaluation for interpolation

8.1 Uncertainty evaluation for convolutions

This module assists in uncertainty propagation for the convolution operation

The convolution operation is a common operation in signal and data processing. Convolving signals is mathematically similar to a filter application.

This module contains the following function:

- *convolve_unc()*: Convolution with uncertainty propagation based on FIR-filter

PyDynamic.uncertainty.propagate_convolution.**convolve_unc**(*x1*, *U1*, *x2*, *U2*, *mode='full'*)

Discrete convolution of two signals with uncertainty propagation

This function supports the convolution modes of `numpy.convolve()`[177] and `scipy.ndimage.convolve1d()`[178].

Parameters

- **x1** (*np.ndarray, (N,)*) – first input signal
- **U1** (*np.ndarray, (N, N)*) – full 2D-covariance matrix associated with x1. If the signal is fully certain, use U1 = None to make use of more efficient calculations.
- **x2** (*np.ndarray, (M,)*) – second input signal

[176] https://scipy.github.io/devdocs/reference/signal.html#module-scipy.signal

- **U2** (*np.ndarray, (M, M)*) – full 2D-covariance matrix associated with x2. If the signal is fully certain, use U2 = None to make use of more efficient calculations.
- **mode** (*str*[179], *optional*) – numpy.convolve()[180]-modes:
  - full: len(y) == N+M-1 (default)
  - valid: len(y) == max(M, N) - min(M, N) + 1
  - same: len(y) == max(M, N) (value+covariance are padded with zeros)

  scipy.ndimage.convolve1d()[181]-modes:
  - nearest: len(y) == N (value+covariance are padded with by stationary assumption)
  - reflect: len(y) == N
  - mirror: len(y) == N

Returns

- **conv** (*np.ndarray*) – convoluted output signal
- **Uconv** (*np.ndarray*) – full 2D-covariance matrix of y

References

See also:

numpy.convolve()[182] scipy.ndimage.convolve1d()[183]

8.2 Uncertainty evaluation for the DFT

Functions for the propagation of uncertainties in the application of the DFT

The *PyDynamic.uncertainty.propagate_DFT* module implements functions for the propagation of uncertainties in the application of the DFT, inverse DFT, deconvolution and multiplication in the frequency domain, transformation from amplitude and phase to real and imaginary parts and vice versa.

The corresponding scientific publications is S. Eichstädt und V. Wilkens GUM2DFT — a software tool for uncertainty evaluation of transient signals in the frequency domain. *Measurement Science and Technology*, 27(5), 055001, 2016. [DOI: 10.1088/0957-0233/27/5/055001[184]]

This module contains the following functions:

- *GUM_DFT()*: Calculation of the DFT of the time domain signal x and propagation of the squared uncertainty Ux associated with the time domain sequence x to the real and imaginary parts of the DFT of x
- *GUM_iDFT()*: GUM propagation of the squared uncertainty UF associated with the DFT values F through the inverse DFT
- *GUM_DFTfreq()*: Return the Discrete Fourier Transform sample frequencies

[177] https://numpy.org/doc/stable/reference/generated/numpy.convolve.html#numpy.convolve
[178] https://scipy.github.io/devdocs/reference/generated/scipy.ndimage.convolve1d.html#scipy.ndimage.convolve1d
[179] https://docs.python.org/3/library/stdtypes.html#str
[180] https://numpy.org/doc/stable/reference/generated/numpy.convolve.html#numpy.convolve
[181] https://scipy.github.io/devdocs/reference/generated/scipy.ndimage.convolve1d.html#scipy.ndimage.convolve1d
[182] https://numpy.org/doc/stable/reference/generated/numpy.convolve.html#numpy.convolve
[183] https://scipy.github.io/devdocs/reference/generated/scipy.ndimage.convolve1d.html#scipy.ndimage.convolve1d
[184] https://dx.doi.org/10.1088/0957-0233/27/5/055001

- *DFT_transferfunction()*: Calculation of the transfer function H = Y/X in the frequency domain with X being the Fourier transform of the system's input signal and Y that of the output signal
- *DFT_deconv()*: Deconvolution in the frequency domain
- *DFT_multiply()*: Multiplication in the frequency domain
- *AmpPhase2DFT()*: Transformation from magnitude and phase to real and imaginary parts
- *DFT2AmpPhase()*: Transformation from real and imaginary parts to magnitude and phase
- *AmpPhase2Time()*: Transformation from amplitude and phase to time domain
- *Time2AmpPhase()*: Transformation from time domain to amplitude and phase

PyDynamic.uncertainty.propagate_DFT.**AmpPhase2DFT**(*A: numpy.ndarray*[185], *P: numpy.ndarray*[186], *UAP: numpy.ndarray*[187], *keep_sparse: Optional*[188]*[bool*[189]*] = False*) → Tuple[190][numpy.ndarray[191], numpy.ndarray[192]]

Transformation from magnitude and phase to real and imaginary parts

Calculate the vector F=[real,imag] and propagate the covariance matrix UAP associated with [A, P]

Parameters

- **A** (*np.ndarray of shape (N,)*) – vector of magnitude values
- **P** (*np.ndarray of shape (N,)*) – vector of phase values (in radians)
- **UAP** (*np.ndarray of shape (2N,2N) or of shape (2N,)*) – covariance matrix associated with (A,P) or vector of squared standard uncertainties [u^2(A),u^2(P)]
- **keep_sparse** (*bool*[193], *optional*) – whether to transform sparse matrix to numpy array or not

Returns

- **F** (*np.ndarray of shape (2N,)*) – vector of real and imaginary parts of DFT result
- **UF** (*np.ndarray of shape (2N,2N)*) – covariance matrix associated with F

Raises *ValueError*[194] – If dimensions of A, P and UAP do not match.

PyDynamic.uncertainty.propagate_DFT.**AmpPhase2Time**(*A: numpy.ndarray*[195], *P: numpy.ndarray*[196], *UAP: numpy.ndarray*[197]) → Tuple[198][numpy.ndarray[199], numpy.ndarray[200]]

Transformation from amplitude and phase to time domain

GUM propagation of covariance matrix UAP associated with DFT amplitude A and phase P to the result of the inverse DFT. Uncertainty UAP is assumed to be given for amplitude and phase with blocks: UAP = [[u(A,A), u(A,P)],[u(P,A),u(P,P)]]

Parameters

- **A** (*np.ndarray of shape (N,)*) – vector of amplitude values

[185] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[186] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[187] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[188] https://docs.python.org/3/library/typing.html#typing.Optional
[189] https://docs.python.org/3/library/functions.html#bool
[190] https://docs.python.org/3/library/typing.html#typing.Tuple
[191] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[192] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[193] https://docs.python.org/3/library/functions.html#bool
[194] https://docs.python.org/3/library/exceptions.html#ValueError

- **P** (*np.ndarray of shape (N,)*) – vector of phase values (in rad)
- **UAP** (*np.ndarray of shape (2N, 2N)*) – covariance matrix associated with [A,P]

Returns

- **x** (*np.ndarray of shape (N,)*) – vector of time domain values
- **Ux** (*np.ndarray of shape (2N, 2N)*) – covariance matrix associated with x

Raises ValueError[201] – If dimension of UAP is not even.

PyDynamic.uncertainty.propagate_DFT.**DFT2AmpPhase**(*F: numpy.ndarray*[202], *UF: numpy.ndarray*[203], *keep_sparse: Optional*[204]*[bool*[205]*] = False*, *tol: Optional*[206]*[float*[207]*] = 1.0*, *return_type: Optional*[208]*[str*[209]*] = 'separate'*) → Union[210][Tuple[211][numpy.ndarray[212], numpy.ndarray[213]], Tuple[214][numpy.ndarray[215], numpy.ndarray[216], numpy.ndarray[217]]]

Transformation from real and imaginary parts to magnitude and phase

Calculate the matrix U_AP = [[U1,U2],[U2^T,U3]] associated with magnitude and phase of the vector F=[real,imag] with associated covariance matrix U_F=[[URR,URI],[URI^T,UII]]

Parameters

- **F** (*np.ndarray of shape (2M,)*) – vector of real and imaginary parts of a DFT result
- **UF** (*np.ndarray of shape (2M,2M)*) – covariance matrix associated with F
- **keep_sparse** (*bool*[218]*, optional*) – if true then UAP will be sparse if UF is one-dimensional
- **tol** (*float*[219]*, optional*) – lower bound for A/uF below which a warning will be issued concerning unreliable results
- **return_type** (*str*[220]*, optional*) – If "separate" then magnitude and phase are returned as separate arrays A and P. Otherwise the list [A, P] is returned
- **separate** (*If return_type ==*) –

Returns

- **A** (*np.ndarray*) – vector of magnitude values
- **P** (*np.ndarray*) – vector of phase values in radians, in the range [-pi, pi], but only present if `return_type = 'separate'`
- **UAP** (*np.ndarray*) – covariance matrix associated with (A,P)
- *Otherwise*

Returns

- **AP** (*np.ndarray*) – vector of magnitude and phase values
- **UAP** (*np.ndarray*) – covariance matrix associated with AP

[195] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[196] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[197] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[198] https://docs.python.org/3/library/typing.html#typing.Tuple
[199] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[200] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[201] https://docs.python.org/3/library/exceptions.html#ValueError

PyDynamic.uncertainty.propagate_DFT.**DFT_deconv**(*H:* *numpy.ndarray*[221], *Y:* *numpy.ndarray*[222], *UH:* *numpy.ndarray*[223], *UY:* *numpy.ndarray*[224]) → Tuple[225][numpy.ndarray[226], Union[227][Tuple[228][numpy.ndarray[229], numpy.ndarray[230], numpy.ndarray[231]], numpy.ndarray[232]]]

Deconvolution in the frequency domain

GUM propagation of uncertainties for the deconvolution X = Y/H with Y and H being the Fourier transform of the measured signal and of the system's impulse response, respectively.

This function returns the covariance matrix as a tuple of blocks if too large for complete storage in memory.

Parameters

- **H** (*np.ndarray of shape (2M,)*) – real and imaginary parts of frequency response values (M an even integer)
- **Y** (*np.ndarray of shape (2M,)*) – real and imaginary parts of DFT values
- **UH** (*np.ndarray of shape (2M,2M) or (2M,)*) – full covariance or diagonal of the covariance matrix associated with H
- **UY** (*np.ndarray of shape (2M,2M) or (2M,)*) – full covariance or diagonal of the covariance matrix associated with Y

Returns

- **X** (*np.ndarray of shape (2M,)*) – real and imaginary parts of DFT values of deconv result
- **UX** (*np.ndarray of shape (2M,2M) or 3-tuple of np.ndarray of shape (M,M)*) – Covariance matrix associated with real and imaginary part of X. If the matrix fully assembled does not fit the memory, we return the auto-covariance for the real parts `URRX` and the imaginary parts `UIIX` and the covariance between the real and imaginary parts `URIX` as separate np.ndarrays[233] arranged as follows: `(URRX, URIX, UIIX)`

[202] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[203] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[204] https://docs.python.org/3/library/typing.html#typing.Optional
[205] https://docs.python.org/3/library/functions.html#bool
[206] https://docs.python.org/3/library/typing.html#typing.Optional
[207] https://docs.python.org/3/library/functions.html#float
[208] https://docs.python.org/3/library/typing.html#typing.Optional
[209] https://docs.python.org/3/library/stdtypes.html#str
[210] https://docs.python.org/3/library/typing.html#typing.Union
[211] https://docs.python.org/3/library/typing.html#typing.Tuple
[212] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[213] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[214] https://docs.python.org/3/library/typing.html#typing.Tuple
[215] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[216] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[217] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[218] https://docs.python.org/3/library/functions.html#bool
[219] https://docs.python.org/3/library/functions.html#float
[220] https://docs.python.org/3/library/stdtypes.html#str

References

- Eichstädt and Wilkens [Eichst2016]

Raises ValueError[234] – If dimensions of H, Y, UY and UH do not match accordingly.

PyDynamic.uncertainty.propagate_DFT.**DFT_multiply**(*Y: numpy.ndarray*[235], *F: numpy.ndarray*[236], *UY: numpy.ndarray*[237], *UF: Optional*[238]*[numpy.ndarray*[239]*] = None*) → Tuple[240][numpy.ndarray[241], numpy.ndarray[242]]

Multiplication in the frequency domain

GUM uncertainty propagation for multiplication in the frequency domain, where the second factor F may have an associated uncertainty. This method can be used, for instance, for the application of a low-pass filter in the frequency domain or the application of deconvolution as a multiplication with an inverse of known uncertainty.

Parameters

- **Y** (*np.ndarray of shape (2M,)*) – real and imaginary parts of the first factor
- **F** (*np.ndarray of shape (2M,)*) – real and imaginary parts of the second factor
- **UY** (*np.ndarray either of shape (2M,) or of shape (2M,2M)*) – covariance matrix or squared uncertainty associated with Y
- **UF** (*np.ndarray of shape (2M,2M), optional*) – covariance matrix associated with F

Returns

- **YF** (*np.ndarray of shape (2M,)*) – the product of Y and F
- **UYF** (*np.ndarray of shape (2M,2M)*) – the uncertainty associated with YF

Raises ValueError[243] – If dimensions of Y and F do not match.

PyDynamic.uncertainty.propagate_DFT.**DFT_transferfunction**(*X*, *Y*, *UX*, *UY*)

Calculation of the transfer function H = Y/X in the frequency domain

[221] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[222] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[223] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[224] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[225] https://docs.python.org/3/library/typing.html#typing.Tuple
[226] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[227] https://docs.python.org/3/library/typing.html#typing.Union
[228] https://docs.python.org/3/library/typing.html#typing.Tuple
[229] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[230] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[231] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[232] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[233] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[234] https://docs.python.org/3/library/exceptions.html#ValueError
[235] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[236] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[237] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[238] https://docs.python.org/3/library/typing.html#typing.Optional
[239] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[240] https://docs.python.org/3/library/typing.html#typing.Tuple
[241] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[242] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[243] https://docs.python.org/3/library/exceptions.html#ValueError

Calculate the transfer function with X being the Fourier transform of the system's input signal and Y that of the output signal.

Parameters

- **X** (*np.ndarray*) – real and imaginary parts of the system's input signal
- **Y** (*np.ndarray*) – real and imaginary parts of the system's output signal
- **UX** (*np.ndarray*) – covariance matrix associated with X
- **UY** (*np.ndarray*) – covariance matrix associated with Y

Returns

- **H** (*np.ndarray*) – real and imaginary parts of the system's frequency response
- **UH** (*np.ndarray*) – covariance matrix associated with H
- This function only calls *DFT_deconv*.

PyDynamic.uncertainty.propagate_DFT.**GUM_DFT**(*x: numpy.ndarray*[244], *Ux: Union*[245]*[numpy.ndarray*[246], *float*[247]*], N: Optional*[248]*[int*[249]*] = None*, *window: Optional*[250]*[numpy.ndarray*[251]*] = None*, *CxCos: Optional*[252]*[numpy.ndarray*[253]*] = None*, *CxSin: Optional*[254]*[numpy.ndarray*[255]*] = None*, *returnC: Optional*[256]*[bool*[257]*] = False*, *mask: Optional*[258]*[numpy.ndarray*[259]*] = None*) → Union[260][Tuple[261][numpy.ndarray[262], Union[263][Tuple[264][numpy.ndarray[265], numpy.ndarray[266], numpy.ndarray[267]], numpy.ndarray[268]]], Tuple[269][numpy.ndarray[270], Union[271][Tuple[272][numpy.ndarray[273], numpy.ndarray[274], numpy.ndarray[275]], numpy.ndarray[276]], Dict[277][str[278], numpy.ndarray[279]]]]

Calculation of the DFT with propagation of uncertainty

Calculation of the DFT of the time domain signal x and propagation of the squared uncertainty Ux associated with the time domain sequence x to the real and imaginary parts of the DFT of x.

Parameters

- **x** (*np.ndarray of shape (M,)*) – vector of time domain signal values
- **Ux** (*np.ndarray of shape (M,) or of shape (M,M) or float*[280]) – covariance matrix associated with x, or vector of squared standard uncertainties, or noise variance as float
- **N** (*int*[281]*, optional*) – length of time domain signal for DFT; N>=len(x)
- **window** (*np.ndarray of shape (M,), optional*) – vector of the time domain window values
- **CxCos** (*np.ndarray, optional*) – cosine part of sensitivity matrix
- **CxSin** (*np.ndarray, optional*) – sine part of sensitivity matrix
- **returnC** (*bool*[282]*, optional*) – if True, return sensitivity matrix blocks, if False (default) do not return them
- **mask** (*ndarray of dtype bool, optional*) – calculate DFT values and uncertainties only at those frequencies where mask is True

Returns

- **F** (*np.ndarray*) – vector of complex valued DFT values or of its real and imaginary parts
- **UF** (*np.ndarray*) – covariance matrix associated with real and imaginary part of F
- **CxCos and CxSin** (*Dict*) – Keys are "CxCos", "CxSin" and values the respective sensitivity matrix entries

References

- Eichstädt and Wilkens [Eichst2016]

Raises `ValueError`[283] – If N < len(x)

PyDynamic.uncertainty.propagate_DFT.**GUM_DFTfreq**(*N*, *dt=1*)

Return the Discrete Fourier Transform sample frequencies

Parameters

- **N** (*int*[284]) – window length
- **dt** (*float*[285]) – sample spacing (inverse of sampling rate)

Returns **f** – Array of length `n//2 + 1` containing the sample frequencies

[244] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[245] https://docs.python.org/3/library/typing.html#typing.Union
[246] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[247] https://docs.python.org/3/library/functions.html#float
[248] https://docs.python.org/3/library/typing.html#typing.Optional
[249] https://docs.python.org/3/library/functions.html#int
[250] https://docs.python.org/3/library/typing.html#typing.Optional
[251] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[252] https://docs.python.org/3/library/typing.html#typing.Optional
[253] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[254] https://docs.python.org/3/library/typing.html#typing.Optional
[255] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[256] https://docs.python.org/3/library/typing.html#typing.Optional
[257] https://docs.python.org/3/library/functions.html#bool
[258] https://docs.python.org/3/library/typing.html#typing.Optional
[259] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[260] https://docs.python.org/3/library/typing.html#typing.Union
[261] https://docs.python.org/3/library/typing.html#typing.Tuple
[262] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[263] https://docs.python.org/3/library/typing.html#typing.Union
[264] https://docs.python.org/3/library/typing.html#typing.Tuple
[265] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[266] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[267] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[268] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[269] https://docs.python.org/3/library/typing.html#typing.Tuple
[270] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[271] https://docs.python.org/3/library/typing.html#typing.Union
[272] https://docs.python.org/3/library/typing.html#typing.Tuple
[273] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[274] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[275] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[276] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[277] https://docs.python.org/3/library/typing.html#typing.Dict
[278] https://docs.python.org/3/library/stdtypes.html#str
[279] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[280] https://docs.python.org/3/library/functions.html#float
[281] https://docs.python.org/3/library/functions.html#int
[282] https://docs.python.org/3/library/functions.html#bool
[283] https://docs.python.org/3/library/exceptions.html#ValueError

Return type ndarray

See also:

None[286] :numpy.fft.rfftfreq

PyDynamic.uncertainty.propagate_DFT.**GUM_iDFT**(*F: numpy.ndarray*[287], *UF: numpy.ndarray*[288], *Nx: Optional*[289]*[int*[290]*] = None*, *Cc: Optional*[291]*[numpy.ndarray*[292]*] = None*, *Cs: Optional*[293]*[numpy.ndarray*[294]*] = None*, *returnC: Optional*[295]*[bool*[296]*] = False*) → Union[297][Tuple[298][numpy.ndarray[299], numpy.ndarray[300]], Tuple[301][numpy.ndarray[302], numpy.ndarray[303], Dict[304][str[305], numpy.ndarray[306]]]]

Propagation of squared uncertainties UF associated with the DFT values F

GUM propagation of the squared uncertainties UF associated with the DFT values F through the inverse DFT.

The matrix UF is assumed to be for real and imaginary part with blocks: UF = [[u(R,R), u(R,I)],[u(I,R),u(I,I)]] and real and imaginary part obtained from calling rfft (DFT for real-valued signal)

Parameters

- **F** (*np.ndarray of shape (2M,)*) – vector of real and imaginary parts of a DFT result
- **UF** (*np.ndarray of shape (2M,2M)*) – covariance matrix associated with real and imaginary parts of F
- **Nx** (*int*[307]*, optional*) – number of samples of iDFT result
- **Cc** (*np.ndarray, optional*) – cosine part of sensitivities (without scaling factor 1/N)
- **Cs** (*np.ndarray, optional*) – sine part of sensitivities (without scaling factor 1/N)
- **returnC** (*bool*[308]*, optional*) – If True, return sensitivity matrix blocks (without scaling factor 1/N), if False do not return them

Returns

- **x** (*np.ndarray*) – vector of time domain signal values
- **Ux** (*np.ndarray*) – covariance matrix associated with x
- **Cc and Cs** (*Dict*) – Keys are “Cc”, “Cs” and values the respective sensitivity matrix entries

References

- Eichstädt and Wilkens [Eichst2016]

Raises ValueError[309] – If Nx is not smaller than dimension of UF - 2

[284] https://docs.python.org/3/library/functions.html#int
[285] https://docs.python.org/3/library/functions.html#float
[286] https://docs.python.org/3/library/constants.html#None

PyDynamic.uncertainty.propagate_DFT.**Time2AmpPhase**(*x: numpy.ndarray*[310], *Ux: Union*[311]*[numpy.ndarray*[312], *float*[313]*]*) → Tuple[314][numpy.ndarray[315], numpy.ndarray[316], numpy.ndarray[317]]

Transformation from time domain to amplitude and phase via DFT

Parameters

- **x** (*np.ndarray of shape (N,)*) – time domain signal
- **Ux** (*np.ndarray of shape (N,) or of shape (N,N) or float*[318]) – covariance matrix associated with x, or vector of squared standard uncertainties, or noise variance as float

Returns

- **A** (*np.ndarray*) – amplitude values
- **P** (*np.ndarray*) – phase values
- **UAP** (*np.ndarray*) – covariance matrix associated with [A,P]

PyDynamic.uncertainty.propagate_DFT.**Time2AmpPhase_multi**(*x*, *Ux*, *selector=None*)

Transformation from time domain to amplitude and phase

Perform transformation for a set of M signals of the same type.

Parameters

- **x** (*np.ndarray of shape (M, nx)*) – M time domain signals of length nx
- **Ux** (*np.ndarray of shape (M,)*) – squared standard deviations representing noise variances of the signals x

[287] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[288] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[289] https://docs.python.org/3/library/typing.html#typing.Optional
[290] https://docs.python.org/3/library/functions.html#int
[291] https://docs.python.org/3/library/typing.html#typing.Optional
[292] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[293] https://docs.python.org/3/library/typing.html#typing.Optional
[294] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[295] https://docs.python.org/3/library/typing.html#typing.Optional
[296] https://docs.python.org/3/library/functions.html#bool
[297] https://docs.python.org/3/library/typing.html#typing.Union
[298] https://docs.python.org/3/library/typing.html#typing.Tuple
[299] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[300] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[301] https://docs.python.org/3/library/typing.html#typing.Tuple
[302] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[303] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[304] https://docs.python.org/3/library/typing.html#typing.Dict
[305] https://docs.python.org/3/library/stdtypes.html#str
[306] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[307] https://docs.python.org/3/library/functions.html#int
[308] https://docs.python.org/3/library/functions.html#bool
[309] https://docs.python.org/3/library/exceptions.html#ValueError
[310] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[311] https://docs.python.org/3/library/typing.html#typing.Union
[312] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[313] https://docs.python.org/3/library/functions.html#float
[314] https://docs.python.org/3/library/typing.html#typing.Tuple
[315] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[316] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[317] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[318] https://docs.python.org/3/library/functions.html#float

- **selector** (*np.ndarray of shape (L,), optional*) – indices of amplitude and phase values that should be returned; default is 0:N-1

Returns

- **A** (*np.ndarray of shape (M,N)*) – amplitude values
- **P** (*np.ndarray of shape (M,N)*) – phase values
- **UAP** (*np.ndarray of shape (M, 3N)*) – diagonals of the covariance matrices: [diag(UAA), diag(UAP), diag(UPP)]

8.3 Uncertainty evaluation for the DWT

This module assists in uncertainty propagation for the discrete wavelet transform

The *PyDynamic.uncertainty.propagate_DWT* module implements methods for the propagation of uncertainties in the application of the discrete wavelet transform (DWT).

This modules contains the following functions:

- *dwt()*: single level DWT
- *wave_dec()*: wavelet decomposition / multi level DWT
- *wave_dec_realtime()*: multi level DWT
- *inv_dwt()*: single level inverse DWT
- *wave_rec()*: wavelet reconstruction / multi level inverse DWT
- *filter_design()*: provide common wavelet filters (via pywt.Wavelet[319])
- *dwt_max_level()*: return the maximum achievable DWT level

PyDynamic.uncertainty.propagate_DWT.**dwt**(*x*, *Ux*, *lowpass*, *highpass*, *states=None*, *realtime=False*, *subsample_start=1*)

Apply low-pass `lowpass` and high-pass `highpass` to time-series data `x`

The uncertainty is propagated through the transformation by using *PyDynamic.uncertainty.propagate_filter.IIRuncFilter()*.

Return the subsampled results.

Parameters

- **x** (*np.ndarray*) – filter input signal
- **Ux** (*float*[320] *or np.ndarray*) – float: standard deviation of white noise in x 1D-array: point-wise standard uncertainties of non-stationary white noise
- **lowpass** (*np.ndarray*) – FIR filter coefficients representing a low-pass for decomposition
- **highpass** (*np.ndarray*) – FIR filter coefficients representing a high-pass for decomposition
- **states** (*dictionary of internal high/lowpass-filter states, optional (default: None)*) – allows to continue at the last used internal state from previous call
- **realtime** (*Boolean, optional (default: False)*) – for realtime applications, no signal padding has to be done before decomposition

[319] https://pywavelets.readthedocs.io/en/latest/ref/wavelets.html#pywt.Wavelet

- **subsample_start** (*int*[321], *optional (default: 1)*) – At which position the subsampling should start, typically 1 (default) or 0. You should be happy with the default. We only need this to realize *wave_dec_realtime()*.

Returns

- **c_approx** (*np.ndarray*) – subsampled low-pass output signal
- **U_approx** (*np.ndarray*) – subsampled low-pass output uncertainty
- **c_detail** (*np.ndarray*) – subsampled high-pass output signal
- **U_detail** (*np.ndarray*) – subsampled high-pass output uncertainty
- **states** (*dictionary of internal high/lowpass-filter states*) – allows to continue at the last used internal state in next call

PyDynamic.uncertainty.propagate_DWT.**dwt_max_level**(*data_length*, *filter_length*)

Return the highest achievable DWT level, given the provided data/filter lengths

Parameters

- **data_length** (*int*[322]) – length of the data *x*, on which the DWT will be performed
- **filter_length** (*int*[323]) – length of the lowpass which will be used to perform the DWT

Returns n_max

Return type int[324]

PyDynamic.uncertainty.propagate_DWT.**filter_design**(*kind*)

Provide low- and highpass filters suitable for discrete wavelet transformation

This wraps pywt.Wavelet[325].

Parameters **kind** (*string*) – filter name, i.e. db4, coif6, gaus9, rbio3.3, …

- supported families: pywt.families()[326]
- supported wavelets: pywt.wavelist()[327]

Returns

- **ld** (*np.ndarray*) – low-pass filter for decomposition
- **hd** (*np.ndarray*) – high-pass filter for decomposition
- **lr** (*np.ndarray*) – low-pass filter for reconstruction
- **hr** (*np.ndarray*) – high-pass filter for reconstruction

PyDynamic.uncertainty.propagate_DWT.**inv_dwt**(*c_approx*, *U_approx*, *c_detail*, *U_detail*, *lowpass*, *highpass*, *states=None*, *realtime=False*)

Single step of inverse discrete wavelet transform

Parameters

- **c_approx** (*np.ndarray*) – low-pass output signal

[320] https://docs.python.org/3/library/functions.html#float
[321] https://docs.python.org/3/library/functions.html#int
[322] https://docs.python.org/3/library/functions.html#int
[323] https://docs.python.org/3/library/functions.html#int
[324] https://docs.python.org/3/library/functions.html#int
[325] https://pywavelets.readthedocs.io/en/latest/ref/wavelets.html#pywt.Wavelet
[326] https://pywavelets.readthedocs.io/en/latest/ref/wavelets.html#pywt.families
[327] https://pywavelets.readthedocs.io/en/latest/ref/wavelets.html#pywt.wavelist

- **U_approx** (*np.ndarray*) – low-pass output uncertainty
- **c_detail** (*np.ndarray*) – high-pass output signal
- **U_detail** (*np.ndarray*) – high-pass output uncertainty
- **lowpass** (*np.ndarray*) – FIR filter coefficients representing a low-pass for reconstruction
- **highpass** (*np.ndarray*) – FIR filter coefficients representing a high-pass for reconstruction
- **states** (*dict*[328], *optional (default: None)*) – internal high/lowpass-filter states, allows to continue at the last used internal state from previous call
- **realtime** (*Boolean, optional (default: False)*) – for realtime applications, no signal padding has to be undone after reconstruction

Returns

- **x** (*np.ndarray*) – upsampled reconstructed signal
- **Ux** (*np.ndarray*) – upsampled uncertainty of reconstructed signal
- **states** (*dictionary of internal high/lowpass-filter states*) – allows to continue at the last used internal state in next call

PyDynamic.uncertainty.propagate_DWT.**wave_dec**(*x*, *Ux*, *lowpass*, *highpass*, *n=- 1*)

Multilevel discrete wavelet transformation of time-series x with uncertainty Ux

Parameters

- **x** (*np.ndarray*) – input signal
- **Ux** (*float*[329] *or np.ndarray*) – float: standard deviation of white noise in x 1D-array: point-wise standard uncertainties of non-stationary white noise
- **lowpass** (*np.ndarray*) – decomposition low-pass for wavelet_block
- **highpass** (*np.ndarray*) – decomposition high-pass for wavelet_block
- **n** (*int*[330], *optional (default: -1)*) – consecutive repetitions of wavelet_block user is warned, if it is not possible to reach the specified depth use highest possible level if set to -1 (default)

Returns

- **coeffs** (*list of arrays*) – order of arrays within list is: [cAn, cDn, cDn-1, …, cD2, cD1]
- **Ucoeffs** (*list of arrays*) – uncertainty of coeffs, same order as coeffs
- **original_length** (*int*) – equals to len(x) necessary to restore correct length

PyDynamic.uncertainty.propagate_DWT.**wave_dec_realtime**(*x*, *Ux*, *lowpass*, *highpass*, *n=1*, *level_states=None*)

Multilevel discrete wavelet transformation of time-series x with uncertainty Ux

Similar to *wave_dec()*, but allows to start from the internal_state of a previous call.

Parameters

- **x** (*np.ndarray*) – input signal
- **Ux** (*float*[331] *or np.ndarray*) – float: standard deviation of white noise in x 1D-array: point-wise standard uncertainties of non-stationary white noise

[328] https://docs.python.org/3/library/stdtypes.html#dict
[329] https://docs.python.org/3/library/functions.html#float
[330] https://docs.python.org/3/library/functions.html#int

- **lowpass** (*np.ndarray*) – decomposition low-pass for wavelet_block
- **highpass** (*np.ndarray*) – decomposition high-pass for wavelet_block
- **n** (*int*[332], *optional (default: 1)*) – consecutive repetitions of wavelet_block There is no maximum level in continuous wavelet transform, so the default is n=1
- **level_states** (*dict*[333], *optional (default: None)*) – internal state from previous call

Returns

- **coeffs** (*list of arrays*) – order of arrays within list is: [cAn, cDn, cDn-1, …, cD2, cD1]
- **Ucoeffs** (*list of arrays*) – uncertainty of coeffs, same order as coeffs
- **original_length** (*int*) – equals to len(x) necessary to restore correct length
- **level_states** (*dict*) – last internal state

PyDynamic.uncertainty.propagate_DWT.**wave_rec**(*coeffs*, *Ucoeffs*, *lowpass*, *highpass*, *original_length=None*)

Multilevel discrete wavelet reconstruction from levels back into time-series

Parameters

- **coeffs** (*list of arrays*) – order of arrays within list is: [cAn, cDn, cDn-1, …, cD2, cD1] where:
  - cAi: approximation coefficients array from i-th level
  - cDi: detail coefficients array from i-th level
- **Ucoeffs** (*list of arrays*) – uncertainty of coeffs, same order as coeffs
- **lowpass** (*np.ndarray*) – reconstruction low-pass for wavelet_block
- **highpass** (*np.ndarray*) – reconstruction high-pass for wavelet_block
- **original_length** (*int*[334], *optional (default: None)*) – necessary to restore correct length of original time-series

Returns

- **x** (*np.ndarray*) – reconstructed signal
- **Ux** (*np.ndarray*) – uncertainty of reconstructed signal

8.4 Uncertainty evaluation for digital filtering

This module contains functions for the propagation of uncertainties through the application of a digital filter using the GUM approach.

This modules contains the following functions:

- *FIRuncFilter()*: Uncertainty propagation for signal y and uncertain FIR filter theta
- *IIRuncFilter()*: Uncertainty propagation for the signal x and the uncertain IIR filter (b,a)

[331] https://docs.python.org/3/library/functions.html#float
[332] https://docs.python.org/3/library/functions.html#int
[333] https://docs.python.org/3/library/stdtypes.html#dict
[334] https://docs.python.org/3/library/functions.html#int

- *IIR_get_initial_state()*: Get a valid internal state for *IIRuncFilter()* that assumes a stationary signal before the first value.

Note: The Elster-Link paper for FIR filters assumes that the autocovariance is known and that noise is stationary!

PyDynamic.uncertainty.propagate_filter.**FIRuncFilter**(*y*, *sigma_noise*, *theta*, *Utheta=None*, *shift=0*, *blow=None*, *kind='corr'*, *return_full_covariance=False*)

Uncertainty propagation for signal y and uncertain FIR filter theta

A preceding FIR low-pass filter with coefficients *blow* can be provided optionally.

This method keeps the signature of *PyDynamic.uncertainty.FIRuncFilter*, but internally works differently and can return a full covariance matrix. Also, sigma_noise can be a full covariance matrix.

Parameters

- **y** (*np.ndarray*) – filter input signal
- **sigma_noise** (*float*[335] *or np.ndarray*) –
  - float: standard deviation of white noise in y
  - 1D-array: interpretation depends on kind
  - 2D-array: full covariance of input
- **theta** (*np.ndarray*) – FIR filter coefficients
- **Utheta** (*np.ndarray, optional*) –
  - 1D-array: coefficient-wise standard uncertainties of filter
  - 2D-array: covariance matrix associated with theta

  if the filter is fully certain, use *Utheta = None* (default) to make use of more efficient calculations. see also the comparison given in <examplesDigital filteringFIRuncFilter_runtime_comparison.py>
- **shift** (*int*[336]*, optional*) – time delay of filter output signal (in samples) (defaults to 0)
- **blow** (*np.ndarray, optional*) – optional FIR low-pass filter
- **kind** (*string, optional*) – only meaningful in combination with sigma_noise a 1D numpy array
  - "diag": point-wise standard uncertainties of non-stationary white noise
  - "corr": single sided autocovariance of stationary colored noise (default)
- **return_full_covariance** (*bool*[337]*, optional*) – whether or not to return a full covariance of the output, defaults to False

Returns

- **x** (*np.ndarray*) – FIR filter output signal
- **Ux** (*np.ndarray*) –
  - return_full_covariance == False : point-wise standard uncertainties associated with x (default)
  - return_full_covariance == True : covariance matrix containing uncertainties associated with x

References

- Elster and Link 2008 [Elster2008]

See also:

PyDynamic.model_estimation.fit_filter

PyDynamic.uncertainty.propagate_filter.**IIR_get_initial_state**(*b*, *a*, *Uab=None*, *x0=1.0*, *U0=1.0*, *Ux=None*)

Calculate the internal state for the IIRuncFilter-function corresponding to stationary non-zero input signal.

Parameters

- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray*) – filter denominator coefficients
- **Uab** (*np.ndarray, optional (default: None)*) – covariance matrix for (a[1:],b)
- **x0** (*float*[338]*, optional (default: 1.0)*) – stationary input value
- **U0** (*float*[339]*, optional (default: 1.0)*) – stationary input uncertainty
- **Ux** (*np.ndarray, optional (default: None)*) – single sided autocovariance of stationary (colored/correlated) noise (needed in the *kind="corr"* case of *IIRuncFilter()*)

Returns **internal_state** – dictionary of state

Return type dict[340]

PyDynamic.uncertainty.propagate_filter.**IIRuncFilter**(*x*, *Ux*, *b*, *a*, *Uab=None*, *state=None*, *kind='corr'*)

Uncertainty propagation for the signal x and the uncertain IIR filter (b,a)

Parameters

- **x** (*np.ndarray*) – filter input signal
- **Ux** (*float*[341] *or np.ndarray*) – float: standard deviation of white noise in x (requires *kind="diag"*) 1D-array: interpretation depends on kind
- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray*) – filter denominator coefficients
- **Uab** (*np.ndarray, optional (default: None)*) – covariance matrix for (a[1:],b)
- **state** (*dict*[342]*, optional (default: None)*) – An internal state (z, dz, P, cache) to start from - e.g. from a previous run of IIRuncFilter.
  - If not given, (z, dz, P) are calculated such that the signal was constant before the given range
  - If given, the input parameters (b, a, Uab) are ignored to avoid repetitive rebuild of the internal system description (instead, the cache is used). However a valid new state (i.e. with new b, a, Uab) can always be generated by using *IIR_get_initial_state()*.

[335] https://docs.python.org/3/library/functions.html#float
[336] https://docs.python.org/3/library/functions.html#int
[337] https://docs.python.org/3/library/functions.html#bool
[338] https://docs.python.org/3/library/functions.html#float
[339] https://docs.python.org/3/library/functions.html#float
[340] https://docs.python.org/3/library/stdtypes.html#dict

- **kind** (*string, optional (default: "corr")*) – defines the interpretation of Ux, if Ux is a 1D-array "diag": point-wise standard uncertainties of non-stationary white noise "corr": single sided autocovariance of stationary (colored/correlated) noise (default)

Returns

- **y** (*np.ndarray*) – filter output signal
- **Uy** (*np.ndarray*) – uncertainty associated with y
- **state** (*dict*) – dictionary of updated internal state

Note: In case of *a == [1.0]* (FIR filter), the results of *IIRuncFilter()* and *FIRuncFilter()* might differ! This is because IIRuncFilter propagates uncertainty according to the (first-order Taylor series of the) GUM, whereas FIRuncFilter takes full variance information into account (which leads to an additional term). This is documented in the description of formula (33) of [Elster2008] . The difference can be visualized by running *PyDynamic/examples/digital_filtering/validate_FIR_IIR_MC.py*

References

- Link and Elster [Link2009]

8.5 Monte Carlo methods for digital filtering

"Monte Carlo methods for the propagation of uncertainties for digital filtering

The propagation of uncertainties via the FIR and IIR formulae alone does not enable the derivation of credible intervals, because the underlying distribution remains unknown. The GUM-S2 Monte Carlo method provides a reference method for the calculation of uncertainties for such cases.

This module contains the following functions:

- *MC()*: Standard Monte Carlo method for application of digital filter
- *SMC()*: Sequential Monte Carlo method with reduced computer memory requirements
- *UMC()*: Update Monte Carlo method for application of digital filters with reduced computer memory requirements
- *UMC_generic()*: Update Monte Carlo method with reduced computer memory requirements

PyDynamic.uncertainty.propagate_MonteCarlo.**MC**(*x*, *Ux*, *b*, *a*, *Uab*, *runs=1000*, *blow=None*, *alow=None*, *return_samples=False*, *shift=0*, *verbose=True*)

Standard Monte Carlo method

Monte Carlo based propagation of uncertainties for a digital filter (b,a) with uncertainty matrix U_θ for $\theta = (a_1, \ldots, a_{N_a}, b_0, \ldots, b_{N_b})^T$

Parameters

- **x** (*np.ndarray*) – filter input signal
- **Ux** (*float*[343] *or np.ndarray*) – standard deviation of signal noise (float), point-wise standard uncertainties or covariance matrix associated with x

[341] https://docs.python.org/3/library/functions.html#float
[342] https://docs.python.org/3/library/stdtypes.html#dict

- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray*) – filter denominator coefficients
- **Uab** (*np.ndarray*) – uncertainty matrix U_θ
- **runs** (*int*[344]*, optional*) – number of Monte Carlo runs
- **return_samples** (*bool*[345]*, optional*) – whether samples or mean and std are returned

Returns

- **y, Uy** (*np.ndarray*) – filtered output signal and associated uncertainties, only returned if return_samples is `False`
- **Y** (*np.ndarray*) – array of Monte Carlo results, only returned if return_samples is `True`

References

- Eichstädt, Link, Harris and Elster [Eichst2012]

PyDynamic.uncertainty.propagate_MonteCarlo.**SMC**(*x*, *noise_std*, *b*, *a*, *Uab=None*, *runs=1000*, *Perc=None*, *blow=None*, *alow=None*, *shift=0*, *return_samples=False*, *phi=None*, *theta=None*, *Delta=0.0*)

Sequential Monte Carlo method

Sequential Monte Carlo propagation for a digital filter (b,a) with uncertainty matrix U_θ for $\theta = (a_1, \ldots, a_{N_a}, b_0, \ldots, b_{N_b})^T$

Parameters

- **x** (*np.ndarray*) – filter input signal
- **noise_std** (*float*[346]) – standard deviation of signal noise
- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray*) – filter denominator coefficients
- **Uab** (*np.ndarray*) – uncertainty matrix U_θ
- **runs** (*int*[347]*, optional*) – number of Monte Carlo runs
- **Perc** (*list*[348]*, optional*) – list of percentiles for quantile calculation
- **blow** (*np.ndarray*) – optional low-pass filter numerator coefficients
- **alow** (*np.ndarray*) – optional low-pass filter denominator coefficients
- **shift** (*int*[349]) – integer for time delay of output signals
- **return_samples** (*bool*[350]*, otpional*) – whether to return y and Uy or the matrix Y of MC results
- **phi** (*np.ndarray, optional*) – parameters for AR(MA) noise model $\epsilon(n) = \sum_k \phi_k \epsilon(n-k) + \sum_k \theta_k w(n-k) + w(n)$ with $w(n) \sim N(0, noise_std^2)$
- **theta** (*np.ndarray, optional*) – parameters for AR(MA) noise model $\epsilon(n) = \sum_k \phi_k \epsilon(n-k) + \sum_k \theta_k w(n-k) + w(n)$ with $w(n) \sim N(0, noise_std^2)$

[343] https://docs.python.org/3/library/functions.html#float
[344] https://docs.python.org/3/library/functions.html#int
[345] https://docs.python.org/3/library/functions.html#bool

- **Delta** (*float*[351]*, optional*) – upper bound on systematic error of the filter

If `return_samples` is `False`, the method returns:

Returns

- **y** (*np.ndarray*) – filter output signal (Monte Carlo mean)
- **Uy** (*np.ndarray*) – uncertainties associated with y (Monte Carlo point-wise std)
- **Quant** (*np.ndarray*) – quantiles corresponding to percentiles `Perc` (if not `None`)

Otherwise the method returns:

Returns **Y** – array of all Monte Carlo results

Return type np.ndarray

References

- Eichstädt, Link, Harris, Elster [Eichst2012]

PyDynamic.uncertainty.propagate_MonteCarlo.**UMC**(*x*, *b*, *a*, *Uab*, *runs=1000*, *blocksize=8*, *blow=1.0*, *alow=1.0*, *phi=0.0*, *theta=0.0*, *sigma=1*, *Delta=0.0*, *runs_init=100*, *nbins=1000*, *credible_interval=0.95*)

Batch Monte Carlo for filtering using update formulae for mean, variance and (approximated) histogram. This is a wrapper for the UMC_generic function, specialised on filters

Parameters

- **x** (*np.ndarray, shape (nx,)*) – filter input signal
- **b** (*np.ndarray, shape (nbb,)*) – filter numerator coefficients
- **a** (*np.ndarray, shape (naa,)*) – filter denominator coefficients, normalization (a[0] == 1.0) is assumed
- **Uab** (*np.ndarray, shape (naa + nbb - 1,)*) – uncertainty matrix U_θ
- **runs** (*int*[352]*, optional*) – number of Monte Carlo runs
- **blocksize** (*int*[353]*, optional*) – how many samples should be evaluated for at a time
- **blow** (*float*[354] *or np.ndarray, optional*) – filter coefficients of optional low pass filter
- **alow** (*float*[355] *or np.ndarray, optional*) – filter coefficients of optional low pass filter
- **phi** (*np.ndarray, optional,*) – see misc.noise.ARMA noise model
- **theta** (*np.ndarray, optional*) – see misc.noise.ARMA noise model
- **sigma** (*float*[356]*, optional*) – see misc.noise.ARMA noise model
- **Delta** (*float*[357]*, optional*) – upper bound of systematic correction due to regularisation (assume uniform distribution)

[346] https://docs.python.org/3/library/functions.html#float
[347] https://docs.python.org/3/library/functions.html#int
[348] https://docs.python.org/3/library/stdtypes.html#list
[349] https://docs.python.org/3/library/functions.html#int
[350] https://docs.python.org/3/library/functions.html#bool
[351] https://docs.python.org/3/library/functions.html#float

- **runs_init** (*int*[358], *optional*) – how many samples to evaluate to form initial guess about limits
- **nbins** (*int*[359], *list of int, optional*) – number of bins for histogram
- **credible_interval** (*float*[360], *optional*) – must be in [0,1] central credible interval size

By default, phi, theta, sigma are chosen such, that N(0,1)-noise is added to the input signal.

Returns

- **y** (*np.ndarray*) – filter output signal
- **Uy** (*np.ndarray*) – uncertainty associated with
- **y_cred_low** (*np.ndarray*) – lower boundary of credible interval
- **y_cred_high** (*np.ndarray*) – upper boundary of credible interval
- **happr** (*dict*) – dictionary keys: given nbin dictionary values: bin-edges val["bin-edges"], bin-counts val["bin-counts"]

References

- Eichstädt, Link, Harris, Elster [Eichst2012]
- ported to python in 2019-08 from matlab-version of Sascha Eichstaedt (PTB) from 2011-10-12
- copyright on updating formulae parts is by Peter Harris (NPL)

PyDynamic.uncertainty.propagate_MonteCarlo.**UMC_generic**(*draw_samples*, *evaluate*, *runs=100*, *blocksize=8*, *runs_init=10*, *nbins=100*, *return_samples=False*, *n_cpu=2*)

Generic Batch Monte Carlo using update formulae for mean, variance and (approximated) histogram. Assumes that the input and output of evaluate are numeric vectors (but not necessarily of same dimension). If the output of evaluate is multi-dimensional, it will be flattened into 1D.

Parameters

- **draw_samples** (*function(int nDraws)*) – function that draws nDraws from a given distribution / population needs to return a list of (multi dimensional) numpy.ndarrays
- **evaluate** (*function(sample)*) – function that evaluates a sample and returns the result needs to return a (multi dimensional) numpy.ndarray
- **runs** (*int*[361], *optional*) – number of Monte Carlo runs
- **blocksize** (*int*[362], *optional*) – how many samples should be evaluated for at a time
- **runs_init** (*int*[363], *optional*) – how many samples to evaluate to form initial guess about limits
- **nbins** (*int*[364], *list of int, optional*) – number of bins for histogram

[352] https://docs.python.org/3/library/functions.html#int
[353] https://docs.python.org/3/library/functions.html#int
[354] https://docs.python.org/3/library/functions.html#float
[355] https://docs.python.org/3/library/functions.html#float
[356] https://docs.python.org/3/library/functions.html#float
[357] https://docs.python.org/3/library/functions.html#float
[358] https://docs.python.org/3/library/functions.html#int
[359] https://docs.python.org/3/library/functions.html#int
[360] https://docs.python.org/3/library/functions.html#float

- **return_samples** (*bool*[365], *optional*) – see return-value of documentation
- **n_cpu** (*int*[366], *optional*) – number of CPUs to use for multiprocessing, defaults to all available CPUs

Example

draw samples from multivariate normal distribution: `draw_samples = lambda size: np.random.multivariate_normal(x, Ux, size)`

build a function, that only accepts one argument by masking additional kwargs: `evaluate = functools.partial(_UMCevaluate, nbb=b.size, x=x, Delta=Delta, phi=phi, theta=theta, sigma=sigma, blow=blow, alow=alow)` `evaluate = functools.partial(bigFunction, **dict_of_kwargs)`

By default the method

Returns

- **y** (*np.ndarray*) – mean of flattened/raveled simulation output i.e.: y = np.ravel(evaluate(sample))
- **Uy** (*np.ndarray*) – covariance associated with y
- **happr** (*dict*) – dictionary of bin-edges and bin-counts
- **output_shape** (*tuple*) – shape of the unraveled simulation output can be used to reshape y and np.diag(Uy) into original shape

If `return_samples` is `True`, the method additionally returns all evaluated samples. This should only be done for testing and debugging reasons, as this removes all memory-improvements of the UMC-method.

Returns **sims** – dict of samples and corresponding results of every evaluated simulation samples and results are saved in their original shape

Return type dict[367]

References

- Eichstädt, Link, Harris, Elster [Eichst2012]

[361] https://docs.python.org/3/library/functions.html#int
[362] https://docs.python.org/3/library/functions.html#int
[363] https://docs.python.org/3/library/functions.html#int
[364] https://docs.python.org/3/library/functions.html#int
[365] https://docs.python.org/3/library/functions.html#bool
[366] https://docs.python.org/3/library/functions.html#int
[367] https://docs.python.org/3/library/stdtypes.html#dict

8.6 Uncertainty evaluation for interpolation

This module assists in uncertainty propagation for 1-dimensional interpolation

The *PyDynamic.uncertainty.interpolate* module implements methods for the propagation of uncertainties in the application of standard interpolation methods as provided by `scipy.interpolate.interp1d`[368].

This module contains the following functions:

- *interp1d_unc()*: Interpolate arbitrary time series considering the associated uncertainties
- *make_equidistant()*: Interpolate a 1-D function equidistantly considering associated uncertainties

PyDynamic.uncertainty.interpolate.**interp1d_unc**(*x_new: numpy.ndarray*[369], *x: numpy.ndarray*[370], *y: numpy.ndarray*[371], *uy: numpy.ndarray*[372], *kind: Optional*[373]*[str*[374]*] = 'linear'*, *copy=True*, *bounds_error: Optional*[375]*[bool*[376]*] = None*, *fill_value: Optional*[377]*[Union*[378]*[float*[379]*, Tuple*[380]*[float*[381]*, float*[382]*], str*[383]*]] = nan*, *fill_unc: Optional*[384]*[Union*[385]*[float*[386]*, Tuple*[387]*[float*[388]*, float*[389]*], str*[390]*]] = nan*, *assume_sorted: Optional*[391]*[bool*[392]*] = True*, *returnC: Optional*[393]*[bool*[394]*] = False*) → Union[395][Tuple[396][numpy.ndarray[397], numpy.ndarray[398], numpy.ndarray[399]], Tuple[400][numpy.ndarray[401], numpy.ndarray[402], numpy.ndarray[403], numpy.ndarray[404]]]

Interpolate a 1-D function considering the associated uncertainties

x and y are arrays of values used to approximate some function $f: y = f(x)$.

Note that calling *interp1d_unc()* with NaNs present in input values results in undefined behaviour.

An equal number of each of the original x and y values and associated uncertainties is required.

Parameters

- **x_new** (*(M,) array_like*) – A 1-D array of real values to evaluate the interpolant at. x_new can be sorted in any order.
- **x** (*(N,) array_like*) – A 1-D array of real values.
- **y** (*(N,) array_like*) – A 1-D array of real values. The length of y must be equal to the length of x.
- **uy** (*(N,) array_like*) – A 1-D array of real values representing the standard uncertainties associated with y.
- **kind** (*str*[405]*, optional*) – Specifies the kind of interpolation for y as a string (‘previous’, ‘next’, ‘nearest’, ‘linear’ or ‘cubic’). Default is ‘linear’.
- **copy** (*bool*[406]*, optional*) – If True, the method makes internal copies of x and y. If False, references to x and y are used. The default is to copy.
- **bounds_error** (*bool*[407]*, optional*) – If True, a ValueError is raised any time interpolation is attempted on a value outside of the range of x (where extrapolation is necessary). If False, out of bounds values are assigned fill_value. By default, an error is raised unless `fill_value="extrapolate"`.

[368] https://scipy.github.io/devdocs/reference/generated/scipy.interpolate.interp1d.html#scipy.interpolate.interp1d

- **fill_value** (*array-like or (array-like, array_like) or "extrapolate", optional*) –
  - if a ndarray (or float), this value will be used to fill in for requested points outside of the data range. If not provided, then the default is NaN. The array-like must broadcast properly to the dimensions of the non-interpolation axes.
  - If a two-element tuple, then the first element is used as a fill value for `x_new < t[0]` and the second element is used for `x_new > t[-1]`. Anything that is not a 2-element tuple (e.g., list or ndarray, regardless of shape) is taken to be a single array-like argument meant to be used for both bounds as `below, above = fill_value, fill_value`.
  - If "extrapolate", then points outside the data range will be set to the first or last element of the values.
  - If cubic-interpolation, C2-continuity at the transition to the extrapolation-range is not guaranteed. This behavior might change in future implementations, see issue #210 for details.

  Both parameters fill_value and fill_unc should be provided to ensure desired behaviour in the extrapolation range.
- **fill_unc** (*array-like or (array-like, array_like) or "extrapolate", optional*) – Usage and behaviour as described in fill_value but for the uncertainties. Both parameters fill_value and fill_unc should be provided to ensure desired behaviour in the extrapolation range.
- **assume_sorted** (*bool*[408]*, optional*) – If False, values of x can be in any order and they are sorted first. If True, x has to be an array of monotonically increasing values.
- **returnC** (*bool*[409]*, optional*) – If True, return sensitivity coefficients for later use. This is only available for interpolation kind 'linear' and for fill_unc="extrapolate" at the moment. If False sensitivity coefficients are not returned and internal computation is slightly more efficient.

Returns

- **x_new** (*(M,) array_like*) – values at which the interpolant is evaluated
- **y_new** (*(M,) array_like*) – interpolated values
- **uy_new** (*(M,) array_like*) – interpolated associated standard uncertainties
- **C** (*(M,N) array_like*) – sensitivity matrix C, which is used to compute the uncertainties $U_{y_{new}} = C \cdot \mathrm{diag}(u_y^2) \cdot C^T$, only returned if returnC is False, which is the default behaviour.

References

- White [White2017]

PyDynamic.uncertainty.interpolate.**make_equidistant**(*x: numpy.ndarray*[410], *y: numpy.ndarray*[411], *uy: numpy.ndarray*[412], *dx: Optional*[413]*[float*[414]*] = 0.05*, *kind: Optional*[415]*[str*[416]*] = 'linear'*) → Tuple[417][numpy.ndarray[418], numpy.ndarray[419], numpy.ndarray[420]]

Interpolate a 1-D function equidistantly considering associated uncertainties

Interpolate function values equidistantly and propagate uncertainties accordingly.

x and y are arrays of values used to approximate some function $f: y = f(x)$.

Note that calling *interp1d_unc()* with NaNs present in input values results in undefined behaviour.

An equal number of each of the original x and y values and associated uncertainties is required.

Parameters

- **x** (*(N,) array_like*) – A 1-D array of real values.
- **y** (*(N,) array_like*) – A 1-D array of real values. The length of y must be equal to the length of x.
- **uy** (*(N,) array_like*) – A 1-D array of real values representing the standard uncertainties associated with y.

[369] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[370] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[371] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[372] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[373] https://docs.python.org/3/library/typing.html#typing.Optional
[374] https://docs.python.org/3/library/stdtypes.html#str
[375] https://docs.python.org/3/library/typing.html#typing.Optional
[376] https://docs.python.org/3/library/functions.html#bool
[377] https://docs.python.org/3/library/typing.html#typing.Optional
[378] https://docs.python.org/3/library/typing.html#typing.Union
[379] https://docs.python.org/3/library/functions.html#float
[380] https://docs.python.org/3/library/typing.html#typing.Tuple
[381] https://docs.python.org/3/library/functions.html#float
[382] https://docs.python.org/3/library/functions.html#float
[383] https://docs.python.org/3/library/stdtypes.html#str
[384] https://docs.python.org/3/library/typing.html#typing.Optional
[385] https://docs.python.org/3/library/typing.html#typing.Union
[386] https://docs.python.org/3/library/functions.html#float
[387] https://docs.python.org/3/library/typing.html#typing.Tuple
[388] https://docs.python.org/3/library/functions.html#float
[389] https://docs.python.org/3/library/functions.html#float
[390] https://docs.python.org/3/library/stdtypes.html#str
[391] https://docs.python.org/3/library/typing.html#typing.Optional
[392] https://docs.python.org/3/library/functions.html#bool
[393] https://docs.python.org/3/library/typing.html#typing.Optional
[394] https://docs.python.org/3/library/functions.html#bool
[395] https://docs.python.org/3/library/typing.html#typing.Union
[396] https://docs.python.org/3/library/typing.html#typing.Tuple
[397] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[398] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[399] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[400] https://docs.python.org/3/library/typing.html#typing.Tuple
[401] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[402] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[403] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[404] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[405] https://docs.python.org/3/library/stdtypes.html#str
[406] https://docs.python.org/3/library/functions.html#bool
[407] https://docs.python.org/3/library/functions.html#bool
[408] https://docs.python.org/3/library/functions.html#bool
[409] https://docs.python.org/3/library/functions.html#bool

- **dx** (*float*[421], *optional*) – desired interval length (defaults to 5e-2)
- **kind** (*str*[422], *optional*) – Specifies the kind of interpolation for y as a string ('previous', 'next', 'nearest', 'linear' or 'cubic'). Default is 'linear'.

Returns

- **x_new** (*(M,) array_like*) – values at which the interpolant is evaluated
- **y_new** (*(M,) array_like*) – interpolated values
- **uy_new** (*(M,) array_like*) – interpolated associated standard uncertainties

References

- White [White2017]

[410] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[411] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[412] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[413] https://docs.python.org/3/library/typing.html#typing.Optional
[414] https://docs.python.org/3/library/functions.html#float
[415] https://docs.python.org/3/library/typing.html#typing.Optional
[416] https://docs.python.org/3/library/stdtypes.html#str
[417] https://docs.python.org/3/library/typing.html#typing.Tuple
[418] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[419] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[420] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[421] https://docs.python.org/3/library/functions.html#float
[422] https://docs.python.org/3/library/stdtypes.html#str

CHAPTER

NINE

MODEL ESTIMATION

The estimation of the measurand in the analysis of dynamic measurements typically corresponds to a deconvolution problem. Therefore, a digital filter can be designed whose input is the measured system output signal and whose output is an estimate of the measurand. The package *Model estimation* implements methods for the design of such filters given an array of frequency response values or the reciprocal of frequency response values with associated uncertainties for the measurement system.

The package *Model estimation* also contains a function for the identification of transfer function models.

The package consists of the following modules:

- `PyDynamic.model_estimation.fit_filter`: least-squares fit to a given complex frequency response or its reciprocal
- `PyDynamic.model_estimation.fit_transfer`: identification of transfer function models

9.1 Fitting filters to frequency response or reciprocal

This module assists in carrying out least-squares IIR and FIR filter fits

It is possible to carry out a least-squares fit of digital, time-discrete IIR and FIR filters to a given complex frequency response and the design of digital deconvolution filters by least-squares fitting to the reciprocal of a given frequency response each with propagation of associated uncertainties.

This module contains the following functions:

- `LSIIR()`: Least-squares (time-discrete) IIR filter fit to a given frequency response or its reciprocal optionally propagating uncertainties.
- `LSFIR()`: Least-squares (time-discrete) FIR filter fit to a given frequency response or its reciprocal optionally propagating uncertainties either via Monte Carlo or via a singular-value decomposition and linear matrix propagation.

PyDynamic.model_estimation.fit_filter.**LSFIR**(*H: numpy.ndarray*[423], *N: int*[424], *f: numpy.ndarray*[425], *Fs: float*[426], *tau: int*[427], *weights: Optional*[428]*[numpy.ndarray*[429]*] = None*, *verbose: Optional*[430]*[bool*[431]*] = True*, *inv: Optional*[432]*[bool*[433]*] = False*, *UH: Optional*[434]*[numpy.ndarray*[435]*] = None*, *mc_runs: Optional*[436]*[int*[437]*] = None*, *trunc_svd_tol: Optional*[438]*[float*[439]*] = None*) → Tuple[440][numpy.ndarray[441], numpy.ndarray[442]]

Design of FIR filter as fit to freq. resp. or its reciprocal with uncertainties

Least-squares fit of a (time-discrete) digital FIR filter to the reciprocal of the frequency response values or actual frequency response values for which associated uncertainties are given for its real and imaginary part. Uncertainties are propagated either using a Monte Carlo method if mc_runs is provided as integer greater than one or otherwise using a truncated singular-value decomposition and linear matrix propagation. The Monte Carlo approach may help in cases where the weighting matrix or the Jacobian are ill-conditioned, resulting in false uncertainties associated with the filter coefficients.

Note: Uncertainty propagation via singular-value decomposition is not yet implemented, when fitting to the actual frequency response and not its reciprocal. Alternatively specify the number mc_runs of runs to propagate the uncertainties via the Monte Carlo method.

Parameters

- **H** (*array_like of shape (M,) or (2M,)*) – (Complex) frequency response values in dtype complex or as a vector containing the real parts in the first half followed by the imaginary parts
- **N** (*int*[443]) – FIR filter order
- **f** (*array_like of shape (M,)*) – frequencies at which H is given
- **Fs** (*float*[444]) – sampling frequency of digital FIR filter
- **tau** (*int*[445]) – time delay in samples for improved fitting
- **weights** (*array_like of shape (2M,), optional*) – vector of weights for a weighted least-squares method (default results in no weighting)
- **verbose** (*bool*[446]*, optional*) – If True (default) verbose output is printed to the command line
- **inv** (*bool*[447]*, optional*) – If False (default) apply the fit to the frequency response values directly, otherwise fit to the reciprocal of the frequency response values
- **UH** (*array_like of shape (2M,2M), optional*) – uncertainties associated with the real and imaginary part of H
- **mc_runs** (*int*[448]*, optional*) – Number of Monte Carlo runs greater than one. Only used, if uncertainties associated with the real and imaginary part of H are provided. Only one of mc_runs and trunc_svd_tol can be provided.
- **trunc_svd_tol** (*float*[449]*, optional*) – Lower bound for singular values to be considered for pseudo-inverse. Values smaller than this threshold are considered zero. Defaults to zero. Only one of mc_runs and trunc_svd_tol can be provided.

Returns

- **b** (*array_like of shape (N+1,)*) – The FIR filter coefficient vector in a 1-D sequence
- **Ub** (*array_like of shape (N+1,N+1)*) – Uncertainties associated with b. Will be None if UH is not provided or is None.

Raises NotImplementedError[450] – The least-squares fitting of a digital FIR filter to a frequency response H with propagation of associated uncertainties using a truncated singular-value decomposition and linear matrix propagation is not yet implemented. Alternatively specify the number mc_runs of runs to propagate the uncertainties via the Monte Carlo method.

References

- Elster and Link [Elster2008]

See also:

PyDynamic.uncertainty.propagate_filter.FIRuncFilter()

PyDynamic.model_estimation.fit_filter.**LSIIR**(*H: numpy.ndarray*[451], *Nb: int*[452], *Na: int*[453], *f: numpy.ndarray*[454], *Fs: float*[455], *tau: Optional*[456]*[int*[457]*] = 0*, *verbose: Optional*[458]*[bool*[459]*] = True*, *max_stab_iter: Optional*[460]*[int*[461]*] = 50*, *inv: Optional*[462]*[bool*[463]*] = False*, *UH: Optional*[464]*[numpy.ndarray*[465]*] = None*, *mc_runs: Optional*[466]*[int*[467]*] = 1000*) → Tuple[468][numpy.ndarray[469], numpy.ndarray[470], int[471], Optional[472][numpy.ndarray[473]]]

Least-squares (time-discrete) IIR filter fit to frequency response or reciprocal

For fitting an IIR filter model to the reciprocal of the frequency response values or directly to the frequency response values provided by the user, this method uses a least-squares fit to determine an estimate of the filter coefficients. The filter then optionally is stabilized by pole mapping and introduction of a time delay. Associated uncertainties are optionally propagated when provided using the GUM S2 Monte Carlo method.

Parameters

- **H** (*array_like of shape (M,)*) – (Complex) frequency response values.
- **Nb** (*int*[474]) – Order of IIR numerator polynomial.
- **Na** (*int*[475]) – Order of IIR denominator polynomial.
- **f** (*array_like of shape (M,)*) – Frequencies at which H is given.
- **Fs** (*float*[476]) – Sampling frequency for digital IIR filter.

[423] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[424] https://docs.python.org/3/library/functions.html#int
[425] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[426] https://docs.python.org/3/library/functions.html#float
[427] https://docs.python.org/3/library/functions.html#int
[428] https://docs.python.org/3/library/typing.html#typing.Optional
[429] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[430] https://docs.python.org/3/library/typing.html#typing.Optional
[431] https://docs.python.org/3/library/functions.html#bool
[432] https://docs.python.org/3/library/typing.html#typing.Optional
[433] https://docs.python.org/3/library/functions.html#bool
[434] https://docs.python.org/3/library/typing.html#typing.Optional
[435] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[436] https://docs.python.org/3/library/typing.html#typing.Optional
[437] https://docs.python.org/3/library/functions.html#int
[438] https://docs.python.org/3/library/typing.html#typing.Optional
[439] https://docs.python.org/3/library/functions.html#float
[440] https://docs.python.org/3/library/typing.html#typing.Tuple
[441] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[442] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[443] https://docs.python.org/3/library/functions.html#int
[444] https://docs.python.org/3/library/functions.html#float
[445] https://docs.python.org/3/library/functions.html#int
[446] https://docs.python.org/3/library/functions.html#bool
[447] https://docs.python.org/3/library/functions.html#bool
[448] https://docs.python.org/3/library/functions.html#int
[449] https://docs.python.org/3/library/functions.html#float
[450] https://docs.python.org/3/library/exceptions.html#NotImplementedError

- **tau** (*int*[477], *optional*) – Initial estimate of time delay for obtaining a stable filter (default = 0).
- **verbose** (*bool*[478], *optional*) – If True (default) be more talkative on stdout. Otherwise no output is written anywhere.
- **max_stab_iter** (*int*[479], *optional*) – Maximum count of iterations for stabilizing the resulting filter. If no stabilization should be carried out, this parameter can be set to 0 (default = 50). This parameter replaced the previous *justFit* which was dropped in PyDynamic 2.0.0.
- **inv** (*bool*[480], *optional*) – If False (default) apply the fit to the frequency response values directly, otherwise fit to the reciprocal of the frequency response values.
- **UH** (*array_like of shape (2M, 2M), optional*) – Uncertainties associated with real and imaginary part of H.
- **mc_runs** (*int*[481], *optional*) – Number of Monte Carlo runs (default = 1000). Only used if uncertainties UH are provided.

Returns

- **b** (*np.ndarray*) – The IIR filter numerator coefficient vector in a 1-D sequence.
- **a** (*np.ndarray*) – The IIR filter denominator coefficient vector in a 1-D sequence.
- **tau** (*int*) – Filter time delay (in samples).
- **Uab** (*np.ndarray of shape (Nb+Na+1, Nb+Na+1)*) – Uncertainties associated with *[a[1:],b]*. Will be None if UH is not provided or is None.

References

- Eichstädt et al. 2010 [Eichst2010]
- Vuerinckx et al. 1996 [Vuer1996]

See also:

PyDynamic.uncertainty.propagate_filter.IIRuncFilter()

9.2 Identification of transfer function models

This module contains a function for the identification of transfer function models:

- *fit_som()*: Fit second-order model to complex-valued frequency response

PyDynamic.model_estimation.fit_transfer.**fit_som**(*f: numpy.ndarray*[482], *H: numpy.ndarray*[483], *UH: Optional*[484]*[numpy.ndarray*[485]*] = None*, *weighting: Optional*[486]*[numpy.ndarray*[487]*] = None*, *MCruns: Optional*[488]*[int*[489]*] = 10000*, *scaling: Optional*[490]*[float*[491]*] = 0.001*, *verbose: Optional*[492]*[bool*[493]*] = False*)

Fit second-order model to complex-valued frequency response

Fit second-order model (spring-damper model) with parameters $S_0, delta$ and f_0 to complex-valued frequency response with uncertainty associated with real and imaginary parts.

For a transformation of an uncertainty associated with amplitude and phase to one associated with real and imaginary parts, see *PyDynamic.uncertainty.propagate_DFT.AmpPhase2DFT*.

Parameters

- **f** (*(M,) np.ndarray*) – vector of frequencies
- **H** (*(2M,) np.ndarray*) – real and imaginary parts of measured frequency response values at frequencies f
- **UH** (*(2M,) or (2M,2M) np.ndarray, optional*) – uncertainties associated with real and imaginary parts When UH is one-dimensional, it is assumed to contain standard uncertainties; otherwise it is taken as covariance matrix. When UH is not specified no uncertainties assoc. with the fit are calculated, which is the default behaviour.

[451] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[452] https://docs.python.org/3/library/functions.html#int
[453] https://docs.python.org/3/library/functions.html#int
[454] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[455] https://docs.python.org/3/library/functions.html#float
[456] https://docs.python.org/3/library/typing.html#typing.Optional
[457] https://docs.python.org/3/library/functions.html#int
[458] https://docs.python.org/3/library/typing.html#typing.Optional
[459] https://docs.python.org/3/library/functions.html#bool
[460] https://docs.python.org/3/library/typing.html#typing.Optional
[461] https://docs.python.org/3/library/functions.html#int
[462] https://docs.python.org/3/library/typing.html#typing.Optional
[463] https://docs.python.org/3/library/functions.html#bool
[464] https://docs.python.org/3/library/typing.html#typing.Optional
[465] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[466] https://docs.python.org/3/library/typing.html#typing.Optional
[467] https://docs.python.org/3/library/functions.html#int
[468] https://docs.python.org/3/library/typing.html#typing.Tuple
[469] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[470] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[471] https://docs.python.org/3/library/functions.html#int
[472] https://docs.python.org/3/library/typing.html#typing.Optional
[473] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[474] https://docs.python.org/3/library/functions.html#int
[475] https://docs.python.org/3/library/functions.html#int
[476] https://docs.python.org/3/library/functions.html#float
[477] https://docs.python.org/3/library/functions.html#int
[478] https://docs.python.org/3/library/functions.html#bool
[479] https://docs.python.org/3/library/functions.html#int
[480] https://docs.python.org/3/library/functions.html#bool
[481] https://docs.python.org/3/library/functions.html#int

- **weighting** (*str*[494] *or (2M,) np.ndarray, optional*) – Type of weighting associated with frequency responses, can be ('diag', 'cov') if UH is given, or Numpy array of weights, defaults to None, which means all values are considered equally important
- **MCruns** (*int*[495]*, optional*) – Number of Monte Carlo trials for propagation of uncertainties, defaults to 10000. When MCruns is set to 'None', matrix multiplication is used for the propagation of uncertainties. However, in some cases this can cause trouble.
- **scaling** (*float*[496]*, optional*) – scaling of least-squares design matrix for improved fit quality, defaults to 1e-3
- **verbose** (*bool*[497]*, optional*) – if True a progressbar will be printed to console during the Monte Carlo simulations, if False nothing will be printed out, defaults to False

Returns

- **p** (*np.ndarray*) – vector of estimated model parameters [S0, delta, f0]
- **Up** (*np.ndarray*) – covariance associated with parameter estimate

[482] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[483] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[484] https://docs.python.org/3/library/typing.html#typing.Optional
[485] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[486] https://docs.python.org/3/library/typing.html#typing.Optional
[487] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[488] https://docs.python.org/3/library/typing.html#typing.Optional
[489] https://docs.python.org/3/library/functions.html#int
[490] https://docs.python.org/3/library/typing.html#typing.Optional
[491] https://docs.python.org/3/library/functions.html#float
[492] https://docs.python.org/3/library/typing.html#typing.Optional
[493] https://docs.python.org/3/library/functions.html#bool
[494] https://docs.python.org/3/library/stdtypes.html#str
[495] https://docs.python.org/3/library/functions.html#int
[496] https://docs.python.org/3/library/functions.html#float
[497] https://docs.python.org/3/library/functions.html#bool

CHAPTER

TEN

MISCELLANEOUS

The *Miscellaneous* package provides various functions and methods which are used in the examples and in some of the other implemented routines.

The package contains the following modules:

- *PyDynamic.misc.SecondOrderSystem*: tools for 2nd order systems
- *PyDynamic.misc.filterstuff*: tools for digital filters
- *PyDynamic.misc.testsignals*: test signals
- *PyDynamic.misc.noise*: noise related functions
- *PyDynamic.misc.tools*: miscellaneous useful helper functions

10.1 Tools for 2nd order systems

A collection of functions to deal with second order dynamic systems

This module is used throughout PyDynamic and is specialized for second order dynamic systems, such as the ones used for high-class accelerometers.

This module contains the following functions:

- *sos_absphase()*: Propagation of uncertainty from physical parameters to real and imaginary part of system's transfer function using GUM S2 Monte Carlo
- *sos_FreqResp()*: Calculation of the system frequency response
- *sos_phys2filter()*: Calculation of continuous filter coefficients from physical parameters
- *sos_realimag()*: Propagation of uncertainty from physical parameters to real and imaginary part of system's transfer function using GUM S2 Monte Carlo

PyDynamic.misc.SecondOrderSystem.**sos_FreqResp**(*S*, *d*, *f0*, *freqs*)

Calculation of the system frequency response

The frequency response is calculated from the continuous physical model of a second order system given by

$H(f) = \frac{4S\pi^2 f_0^2}{(2\pi f_0)^2 + 2jd(2\pi f_0)f - f^2}$

If the provided system parameters are vectors then $H(f)$ is calculated for each set of parameters. This is helpful for Monte Carlo simulations by using draws from the model parameters

Parameters

- **S** (*float*[498] *or ndarray shape (K,)*) – static gain

- **d** (*float*[499] *or ndarray shape (K,)*) – damping parameter
- **f0** (*float*[500] *or ndarray shape (K,)*) – resonance frequency
- **freqs** (*ndarray shape (N,)*) – frequencies at which to calculate the freq response

Returns **H** – complex frequency response values(

Return type ndarray shape (N,) or ndarray shape (N,K)

PyDynamic.misc.SecondOrderSystem.**sos_absphase**(*S*, *d*, *f0*, *uS*, *ud*, *uf0*, *f*, *runs=10000*)

Propagation of uncertainty from physical parameters to amplitude and phase

Propagation of uncertainties from physical parameters to amplitude and phase of system's transfer function is performed using GUM S2 Monte Carlo.

Parameters

- **S** (*float*[501]) – static gain
- **d** (*float*[502]) – damping
- **f0** (*float*[503]) – resonance frequency
- **uS** (*float*[504]) – uncertainty associated with static gain
- **ud** (*float*[505]) – uncertainty associated with damping
- **uf0** (*float*[506]) – uncertainty associated with resonance frequency
- **f** (*ndarray, shape (N,)*) – frequency values at which to calculate amplitude and phase
- **runs** (*int*[507]*, optional*) – number of Monte Carlo runs

Returns

- **Hmean** (*ndarray, shape (N,)*) – best estimate of complex frequency response values
- **Hcov** (*ndarray, shape (2N,2N)*) – covariance matrix [[u(abs,abs), u(abs,phase)], [u(phase,abs), u(phase,phase)]]

PyDynamic.misc.SecondOrderSystem.**sos_phys2filter**(*S*, *d*, *f0*)

Calculation of continuous filter coefficients from physical parameters.

If the provided system parameters are vectors then the filter coefficients are calculated for each set of parameters. This is helpful for Monte Carlo simulations by using draws from the model parameters

Parameters

- **S** (*float*[508]) – static gain
- **d** (*float*[509]) – damping parameter
- **f0** (*float*[510]) – resonance frequency

Returns **b, a** – analogue filter coefficients

Return type ndarray

[498] https://docs.python.org/3/library/functions.html#float
[499] https://docs.python.org/3/library/functions.html#float
[500] https://docs.python.org/3/library/functions.html#float
[501] https://docs.python.org/3/library/functions.html#float
[502] https://docs.python.org/3/library/functions.html#float
[503] https://docs.python.org/3/library/functions.html#float
[504] https://docs.python.org/3/library/functions.html#float
[505] https://docs.python.org/3/library/functions.html#float
[506] https://docs.python.org/3/library/functions.html#float
[507] https://docs.python.org/3/library/functions.html#int

PyDynamic.misc.SecondOrderSystem.**sos_realimag**(*S*, *d*, *f0*, *uS*, *ud*, *uf0*, *f*, *runs=10000*)

Propagation of uncertainty from physical parameters to real and imaginary part

Propagation of uncertainties from physical parameters to real and imaginary part of system's transfer function is performed using GUM S2 Monte Carlo.

Parameters

- **S** (*float*[511]) – static gain
- **d** (*float*[512]) – damping
- **f0** (*float*[513]) – resonance frequency
- **uS** (*float*[514]) – uncertainty associated with static gain
- **ud** (*float*[515]) – uncertainty associated with damping
- **uf0** (*float*[516]) – uncertainty associated with resonance frequency
- **f** (*ndarray, shape (N,)*) – frequency values at which to calculate real and imaginary part
- **runs** (*int*[517]*, optional*) – number of Monte Carlo runs

Returns

- **Hmean** (*ndarray, shape (N,)*) – best estimate of complex frequency response values
- **Hcov** (*ndarray, shape (2N,2N)*) – covariance matrix [[u(real,real), u(real,imag)], [u(imag,real), u(imag,imag)]]

10.2 Tools for digital filters

This module is a collection of functions which are related to filter design

This module contains the following functions:

- *db()*: Calculation of decibel values $20\log_{10}(x)$ for a vector of values
- *ua()*: Shortcut for calculation of unwrapped angle of complex values
- *grpdelay()*: Calculation of the group delay of a digital filter
- *mapinside()*: Maps the roots of polynomial with coefficients a to the unit circle
- *kaiser_lowpass()*: Design of a FIR lowpass filter using the window technique with a Kaiser window.
- *isstable()*: Determine whether an IIR filter with certain coefficients is stable
- *savitzky_golay()*: Smooth (and optionally differentiate) data with a Savitzky-Golay filter

[508] https://docs.python.org/3/library/functions.html#float
[509] https://docs.python.org/3/library/functions.html#float
[510] https://docs.python.org/3/library/functions.html#float
[511] https://docs.python.org/3/library/functions.html#float
[512] https://docs.python.org/3/library/functions.html#float
[513] https://docs.python.org/3/library/functions.html#float
[514] https://docs.python.org/3/library/functions.html#float
[515] https://docs.python.org/3/library/functions.html#float
[516] https://docs.python.org/3/library/functions.html#float
[517] https://docs.python.org/3/library/functions.html#int

PyDynamic.misc.filterstuff.**db**(*vals*)

Calculation of decibel values $20\log_{10}(x)$ for a vector of values

PyDynamic.misc.filterstuff.**grpdelay**(*b*, *a*, *Fs*, *nfft=512*)

Calculation of the group delay of a digital filter

Parameters

- **b** (*ndarray*) – IIR filter numerator coefficients
- **a** (*ndarray*) – IIR filter denominator coefficients
- **Fs** (*float*[518]) – sampling frequency of the filter
- **nfft** (*int*[519]) – number of FFT bins

Returns

- **group_delay** (*np.ndarray*) – group delay values
- **frequencies** (*ndarray*) – frequencies at which the group delay is calculated

References

- Smith, online book [Smith]

PyDynamic.misc.filterstuff.**isstable**(*b*, *a*, *ftype='digital'*)

Determine whether an IIR filter with certain coefficients is stable

Determine whether IIR filter with coefficients *b* and *a* is stable by checking the roots of the polynomial *a*.

Parameters

- **b** (*ndarray*) – Filter numerator coefficients. These are only part of the input parameters for compatibility reasons (especially with MATLAB code). During the computation they are actually not used.
- **a** (*ndarray*) – Filter denominator coefficients.
- **ftype** (*string, optional*) – Filter type. 'digital' if in discrete-time (default) and 'analog' if in continuous-time.

Returns **stable** – Whether filter is stable or not.

Return type bool[520]

PyDynamic.misc.filterstuff.**kaiser_lowpass**(*L*, *fcut*, *Fs*, *beta=8.0*)

Design of a FIR lowpass filter using the window technique with a Kaiser window.

This method uses a Kaiser window. Filters of that type are often used as FIR low-pass filters due to their linear phase.

Parameters

- **L** (*int*[521]) – filter order (window length)
- **fcut** (*float*[522]) – desired cut-off frequency
- **Fs** (*float*[523]) – sampling frequency
- **beta** (*float*[524]) – scaling parameter for the Kaiser window

[518] https://docs.python.org/3/library/functions.html#float
[519] https://docs.python.org/3/library/functions.html#int
[520] https://docs.python.org/3/library/functions.html#bool

Returns

- **blow** (*ndarray*) – FIR filter coefficients
- **shift** (*int*) – delay of the filter (in samples)

PyDynamic.misc.filterstuff.**mapinside**(*a*)

Maps the roots of polynomial to the unit circle.

Maps the roots of polynomial with coefficients a to the unit circle.

Parameters **a** (*ndarray*) – polynomial coefficients

Returns **a** – polynomial coefficients with all roots inside or on the unit circle

Return type ndarray

PyDynamic.misc.filterstuff.**savitzky_golay**(*y*, *window_size*, *order*, *deriv=0*, *delta=1.0*)

Smooth (and optionally differentiate) data with a Savitzky-Golay filter

The Savitzky-Golay filter removes high frequency noise from data. It has the advantage of preserving the original shape and features of the signal better than other types of filtering approaches, such as moving averages techniques.

Source obtained from scipy cookbook (online), downloaded 2013-09-13

Parameters

- **y** (*ndarray, shape (N,)*) – the values of the time history of the signal
- **window_size** (*int*[525]) – the length of the window. Must be an odd integer number
- **order** (*int*[526]) – the order of the polynomial used in the filtering. Must be less then *window_size* - 1.
- **deriv** (*int*[527], *optional*) – The order of the derivative to compute. This must be a non-negative integer. The default is 0, which means to filter the data without differentiating.
- **delta** (*float*[528], *optional*) – The spacing of the samples to which the filter will be applied. This is only used if deriv > 0. This includes a factor $n!/h^n$, where n is represented by *deriv* and $1/h$ by *delta*.

Returns **ys** – the smoothed signal (or it's n-th derivative).

Return type ndarray, shape (N,)

Notes

The Savitzky-Golay is a type of low-pass filter, particularly suited for smoothing noisy data. The main idea behind this approach is to make for each point a least-square fit with a polynomial of high order over a odd-sized window centered at the point.

[521] https://docs.python.org/3/library/functions.html#int
[522] https://docs.python.org/3/library/functions.html#float
[523] https://docs.python.org/3/library/functions.html#float
[524] https://docs.python.org/3/library/functions.html#float

References

- Savitzky et al. [Savitzky]
- Numerical Recipes [NumRec]

PyDynamic.misc.filterstuff.**ua**(*vals*)

Shortcut for calculation of unwrapped angle of complex values

10.3 Test signals

A collection of test signals which can be used to simulate dynamic measurements

This module contains the following functions:

- *GaussianPulse()*: Generate a Gaussian pulse at t0 with height m0 and std sigma
- *multi_sine()*: Generate a multi-sine signal as summation of single sine signals
- *rect()*: Rectangular signal of given height and width $t_1 - t_0$
- *shocklikeGaussian()*: Generate a signal that resembles a shock excitation as a Gaussian
- *sine()*: Generate a sine signal
- *squarepulse()*: Generates a series of rect functions to represent a square pulse signal

PyDynamic.misc.testsignals.**GaussianPulse**(*time*, *t0*, *m0*, *sigma*, *noise=0.0*)

Generate a Gaussian pulse at t0 with height m0 and std sigma

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants (equidistant)
- **t0** (*float*[529]) – time instant of signal maximum
- **m0** (*float*[530]) – signal maximum
- **sigma** (*float*[531]) – std of pulse
- **noise** (*float*[532]*, optional*) – std of simulated signal noise

Returns **x** – signal amplitudes at time instants

Return type np.ndarray of shape (N,)

class PyDynamic.misc.testsignals.**corr_noise**(*w*, *sigma*, *seed=None*)

Base class for generation of a correlated noise process

PyDynamic.misc.testsignals.**multi_sine**(*time*, *amps*, *freqs*, *noise=0.0*)

Generate a batch of a summation of sine signals with normally distributed noise

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants

[525] https://docs.python.org/3/library/functions.html#int
[526] https://docs.python.org/3/library/functions.html#int
[527] https://docs.python.org/3/library/functions.html#int
[528] https://docs.python.org/3/library/functions.html#float
[529] https://docs.python.org/3/library/functions.html#float
[530] https://docs.python.org/3/library/functions.html#float
[531] https://docs.python.org/3/library/functions.html#float
[532] https://docs.python.org/3/library/functions.html#float

- **amps** (*list*[533] *or np.ndarray of shape (M,) of floating point values*) – amplitudes of the sine signals
- **freqs** (*list*[534] *or np.ndarray of shape (M,) of floating point values*) – frequencies of the sine signals in Hz
- **noise** (*float*[535]*, optional*) – std of simulated signal noise (default = 0.0)

Returns **x** – signal amplitude at time instants

Return type np.ndarray of shape (N,)

PyDynamic.misc.testsignals.**rect**(*time*, *t0*, *t1*, *height=1*, *noise=0.0*)

Rectangular signal of given height and width t1-t0

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants (equidistant)
- **t0** (*float*[536]) – time instant of rect lhs
- **t1** (*float*[537]) – time instant of rect rhs
- **height** (*float*[538]) – signal maximum
- **noise** (*float*[539] *or numpy.ndarray of shape (N,), optional*) – float: standard deviation of additive white gaussian noise ndarray: user-defined additive noise

Returns **x** – signal amplitudes at time instants

Return type np.ndarray of shape (N,)

PyDynamic.misc.testsignals.**shocklikeGaussian**(*time*, *t0*, *m0*, *sigma*, *noise=0.0*)

Generate a signal that resembles a shock excitation as a Gaussian

The main shock is followed by a smaller Gaussian of opposite sign.

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants (equidistant)
- **t0** (*float*[540]) – time instant of signal maximum
- **m0** (*float*[541]) – signal maximum
- **sigma** (*float*[542]) – std of main pulse
- **noise** (*float*[543]*, optional*) – std of simulated signal noise

Returns **x** – signal amplitudes at time instants

Return type np.ndarray of shape (N,)

[533] https://docs.python.org/3/library/stdtypes.html#list
[534] https://docs.python.org/3/library/stdtypes.html#list
[535] https://docs.python.org/3/library/functions.html#float
[536] https://docs.python.org/3/library/functions.html#float
[537] https://docs.python.org/3/library/functions.html#float
[538] https://docs.python.org/3/library/functions.html#float
[539] https://docs.python.org/3/library/functions.html#float
[540] https://docs.python.org/3/library/functions.html#float
[541] https://docs.python.org/3/library/functions.html#float
[542] https://docs.python.org/3/library/functions.html#float
[543] https://docs.python.org/3/library/functions.html#float

PyDynamic.misc.testsignals.**sine**(*time*, *amp=1.0*, *freq=1.0*, *noise=0.0*)

Generate a batch of a sine signal with normally distributed noise

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants
- **amp** (*float*[544], *optional*) – amplitude of the sine (default = 1.0)
- **freq** (*float*[545], *optional*) – frequency of the sine in Hz (default = 1.0)
- **noise** (*float*[546], *optional*) – std of simulated signal noise (default = 0.0)

Returns **x** – signal amplitude at time instants

Return type np.ndarray of shape (N,)

PyDynamic.misc.testsignals.**squarepulse**(*time*, *height*, *numpulse=4*, *noise=0.0*)

Generates a series of rect functions to represent a square pulse signal

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants
- **height** (*float*[547]) – height of the rectangular pulses
- **numpulse** (*int*[548]) – number of pulses
- **noise** (*float*[549], *optional*) – std of simulated signal noise

Returns **x** – signal amplitude at time instants

Return type np.ndarray of shape (N,)

10.4 Noise related functions

Collection of noise-signals

This module contains the following functions:

- *ARMA()*: autoregressive moving average noise process
- *get_alpha()*: normal distributed signal amplitudes with equal power spectral density
- *power_law_acf()*: The theoretic right-sided autocorrelation (Rww) of different colors of noise
- *power_law_noise()*: normal distributed signal amplitudes with power spectrum f^{α}
- *power_law_acf()*: (theoretical) autocorrelation function of power law noise
- *white_gaussian()*: Draw random samples from a normal (Gaussian) distribution

PyDynamic.misc.noise.**ARMA**(*length*, *phi=0.0*, *theta=0.0*, *std=1.0*)

Generate time-series of a predefined ARMA-process

The process is generated based on this equation: $\sum_{j=1}^{\min(p,n-1)} \phi_j \epsilon[n-j] + \sum_{j=1}^{\min(q,n-1)} \theta_j w[n-j]$ where w is white gaussian noise. Equation and algorithm taken from [Eichst2012] .

[544] https://docs.python.org/3/library/functions.html#float
[545] https://docs.python.org/3/library/functions.html#float
[546] https://docs.python.org/3/library/functions.html#float
[547] https://docs.python.org/3/library/functions.html#float
[548] https://docs.python.org/3/library/functions.html#int
[549] https://docs.python.org/3/library/functions.html#float

Parameters

- **length** (*int*[550]) – how long the drawn sample will be
- **phi** (*float*[551], *list*[552] *or* *numpy.ndarray*[553], *shape (p,)*) – AR-coefficients
- **theta** (*float*[554], *list*[555] *or* *numpy.ndarray*[556]) – MA-coefficients
- **std** (*float*[557]) – std of the gaussian white noise that is fed into the ARMA-model

Returns **e** – time-series of the predefined ARMA-process

Return type np.ndarray of shape (length,)

References

- Eichstädt, Link, Harris and Elster [Eichst2012]

PyDynamic.misc.noise.**get_alpha**(*color_value=0*)

Return the matching alpha for the provided color value

Translate a color (given as string) into an exponent alpha or directly hand through a given numeric value of alpha.

Parameters **color_value** (*str*[558], *int*[559] *or* *float*[560]) –

- if string -> check against known color names -> return alpha
- if numeric -> directly return value

Returns **alpha** – exponent alpha or directly numeric value

Return type float[561]

PyDynamic.misc.noise.**power_law_acf**(*N*, *color_value='white'*, *std=1.0*)

The theoretic right-sided autocorrelation (Rww) of different colors of noise

Colors of noise are defined to have a power spectral density (Sww) proportional to math:*f^alpha*. Sww and Rww form a Fourier-pair. Therefore Rww = ifft(Sww).

PyDynamic.misc.noise.**power_law_noise**(*N=None*, *w=None*, *color_value='white'*, *mean=0.0*, *std=1.0*)

Generate colored noise

Generate colored noise by:

- generate white gaussian noise
- multiplying its Fourier-transform with $f^{\alpha/2}$
- inverse Fourier-transform to yield the colored/correlated noise
- further adjustments to fit to specified mean/std

based on [Zhivomirov2018] .

[550] https://docs.python.org/3/library/functions.html#int
[551] https://docs.python.org/3/library/functions.html#float
[552] https://docs.python.org/3/library/stdtypes.html#list
[553] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[554] https://docs.python.org/3/library/functions.html#float
[555] https://docs.python.org/3/library/stdtypes.html#list
[556] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[557] https://docs.python.org/3/library/functions.html#float
[558] https://docs.python.org/3/library/stdtypes.html#str
[559] https://docs.python.org/3/library/functions.html#int
[560] https://docs.python.org/3/library/functions.html#float
[561] https://docs.python.org/3/library/functions.html#float

Parameters

- **N** (*int*[562]) – length of noise to be generated
- **w** (*numpy.ndarray*[563]) – user-defined white noise, if provided, N is ignored!
- **color_value** (*str*[564], *int*[565] *or float*[566]) – if string -> check against known color names if numeric -> used as alpha to shape PSD
- **mean** (*float*[567]) – mean of the output signal
- **std** (*float*[568]) – standard deviation of the output signal

Returns **w_filt** – filtered noise signal

Return type np.ndarray

PyDynamic.misc.noise.**white_gaussian**(*N*, *mean=0*, *std=1*)

Draw random samples from a normal (Gaussian) distribution

Parameters

- **N** (*int*[569] *or tuple of ints*) – Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.
- **mean** (*float*[570] *or array_like of floats*) – Mean ("centre") of the distribution.
- **std** (*float*[571] *or array_like of floats*) – Standard deviation (spread or "width") of the distribution. Must be non-negative.

Returns Drawn samples from the parameterized normal distribution.

Return type np.ndarray

10.5 Miscellaneous useful helper functions

A collection of miscellaneous helper functions

This module contains the following functions:

- *FreqResp2RealImag()*: Calculate real and imaginary parts from frequency response
- *is_2d_matrix()*: Check if a np.ndarray is a matrix
- *is_2d_square_matrix()*: Check if a np.ndarray is a two-dimensional square matrix
- *is_vector()*: Check if a np.ndarray is a vector
- *make_semiposdef()*: Make quadratic matrix positive semi-definite
- *normalize_vector_or_matrix()*: Scale an array of numbers to the interval between zero and one
- *number_of_rows_equals_vector_dim()*: Check if a matrix and a vector match in size

[562] https://docs.python.org/3/library/functions.html#int
[563] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[564] https://docs.python.org/3/library/stdtypes.html#str
[565] https://docs.python.org/3/library/functions.html#int
[566] https://docs.python.org/3/library/functions.html#float
[567] https://docs.python.org/3/library/functions.html#float
[568] https://docs.python.org/3/library/functions.html#float
[569] https://docs.python.org/3/library/functions.html#int
[570] https://docs.python.org/3/library/functions.html#float
[571] https://docs.python.org/3/library/functions.html#float

- *plot_vectors_and_covariances_comparison()*: Plot two vectors and their covariances side-by-side for visual comparison
- *print_mat()*: Print matrix (2D array) to the console or return as formatted string
- *print_vec()*: Print vector (1D array) to the console or return as formatted string
- *progress_bar()*: A simple and reusable progress-bar
- *shift_uncertainty()*: Shift the elements in the vector x and associated uncertainties ux
- *trimOrPad()*: trim or pad (with zeros) a vector to desired length
- *complex_2_real_imag()*: Take a np.ndarray with dtype complex and return real and imaginary parts
- *real_imag_2_complex()*: Take a np.ndarray with real and imaginary parts and return dtype complex ndarray
- *separate_real_imag_of_mc_samples()*: Split a np.ndarray containing MonteCarlo samples' real and imaginary parts
- *separate_real_imag_of_vector()*: Split a np.ndarray containing real and imaginary parts into half

PyDynamic.misc.tools.**FreqResp2RealImag**(*Abs: numpy.ndarray*[572], *Phase: numpy.ndarray*[573], *Unc: numpy.ndarray*[574], *MCruns: Optional*[575]*[int*[576]*] = 1000*)

Calculate real and imaginary parts from frequency response

Calculate real and imaginary parts from amplitude and phase with associated uncertainties.

Parameters

- **Abs** (*(N,) array_like*) – amplitude values
- **Phase** (*(N,) array_like*) – phase values in rad
- **Unc** (*(2N, 2N) or (2N,) array_like*) – uncertainties either as full covariance matrix or as its main diagonal
- **MCruns** (*int*[577]*, optional*) – number of iterations for Monte Carlo simulation, defaults to 1000

Returns

- **Re, Im** (*(N,) array_like*) – best estimate of real and imaginary parts
- **URI** (*(2N, 2N) array_like*) – uncertainties assoc. with Re and Im

PyDynamic.misc.tools.**complex_2_real_imag**(*array: numpy.ndarray*[578]) → numpy.ndarray[579]

Take an array of any non-flexible scalar dtype to return real and imaginary part

The input array $x \in \mathbb{R}^m$ is reassembled to the form of the expected input of some of the functions in the modules *propagate_DFT* and *fit_filter*: $y = (\mathrm{Re}(x), \mathrm{Im}(x))$.

Parameters **array** (*np.ndarray of shape (M,)*) – the array to assemble the version with real and imaginary parts from

Returns the array of real and imaginary parts

Return type np.ndarray of shape (2M,)

[572] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[573] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[574] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[575] https://docs.python.org/3/library/typing.html#typing.Optional
[576] https://docs.python.org/3/library/functions.html#int
[577] https://docs.python.org/3/library/functions.html#int
[578] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[579] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

PyDynamic.misc.tools.**is_2d_matrix**(*ndarray: numpy.ndarray*[580]) → bool[581]

Check if a np.ndarray is a matrix, i.e. is of shape (n,m)

Parameters **ndarray** (*np.ndarray*) – the array to check

Returns True, if the array expands over exactly two dimensions, False otherwise

Return type bool[582]

PyDynamic.misc.tools.**is_2d_square_matrix**(*ndarray: numpy.ndarray*[583]) → bool[584]

Check if a np.ndarray is a two-dimensional square matrix, i.e. is of shape (n,n)

Parameters **ndarray** (*np.ndarray*) – the array to check

Returns True, if the array expands over exactly two dimensions of similar size, False otherwise

Return type bool[585]

PyDynamic.misc.tools.**is_vector**(*ndarray: numpy.ndarray*[586]) → bool[587]

Check if a np.ndarray is a vector, i.e. is of shape (n,)

Parameters **ndarray** (*np.ndarray*) – the array to check

Returns True, if the array expands over one dimension only, False otherwise

Return type bool[588]

PyDynamic.misc.tools.**make_equidistant**(**args*, ***kwargs*)

Deprecated since version 2.0.0: Please use *PyDynamic.uncertainty.interpolate.make_equidistant()*

PyDynamic.misc.tools.**make_semiposdef**(*matrix: numpy.ndarray*[589], *maxiter: Optional*[590]*[int*[591]*] = 10*, *tol: Optional*[592]*[float*[593]*] = 1e-12*, *verbose: Optional*[594]*[bool*[595]*] = False*) → numpy.ndarray[596]

Make quadratic matrix positive semi-definite by increasing its eigenvalues

Parameters

- **matrix** (*array_like of shape (N,N)*) – the matrix to process
- **maxiter** (*int*[597]*, optional*) – the maximum number of iterations for increasing the eigenvalues, defaults to 10
- **tol** (*float*[598]*, optional*) – tolerance for deciding if pos. semi-def., defaults to 1e-12
- **verbose** (*bool*[599]*, optional*) – If True print smallest eigenvalue of the resulting matrix, if False (default) be quiet

Returns quadratic positive semi-definite matrix

Return type (N,N) array_like

Raises **ValueError**[600] – If matrix is not square.

[580] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[581] https://docs.python.org/3/library/functions.html#bool
[582] https://docs.python.org/3/library/functions.html#bool
[583] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[584] https://docs.python.org/3/library/functions.html#bool
[585] https://docs.python.org/3/library/functions.html#bool
[586] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[587] https://docs.python.org/3/library/functions.html#bool
[588] https://docs.python.org/3/library/functions.html#bool

PyDynamic.misc.tools.**normalize_vector_or_matrix**(*numbers: numpy.ndarray*[601]) → numpy.ndarray[602]

Scale an array of numbers to the interval between zero and one

If all values in the array are the same, the output array will be constant zero.

Parameters **numbers** (*np.ndarray*) – the numpy.ndarray[603] to normalize

Returns the normalized array

Return type np.ndarray

PyDynamic.misc.tools.**number_of_rows_equals_vector_dim**(*matrix: numpy.ndarray*[604], *vector: numpy.ndarray*[605]) → bool[606]

Check if a matrix has the same number of rows as a vector

Parameters

- **matrix** (*np.ndarray*) – the matrix, that is supposed to have the same number of rows
- **vector** (*np.ndarray*) – the vector, that is supposed to have the same number of elements

Returns True, if the number of rows coincide, False otherwise

Return type bool[607]

PyDynamic.misc.tools.**plot_vectors_and_covariances_comparison**(*vector_1: numpy.ndarray*[608], *vector_2: numpy.ndarray*[609], *covariance_1: numpy.ndarray*[610], *covariance_2: numpy.ndarray*[611], *title: Optional*[612]*[str*[613]*] = 'Comparison between two vectors and corresponding uncertainties'*, *label_1: Optional*[614]*[str*[615]*] = 'vector_1'*, *label_2: Optional*[616]*[str*[617]*] = 'vector_2'*)

Plot two vectors and their covariances side-by-side for visual comparison

Parameters

- **vector_1** (*np.ndarray*) – the first vector to compare
- **vector_2** (*np.ndarray*) – the second vector to compare
- **covariance_1** (*np.ndarray*) – the first covariance matrix to compare

[589] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[590] https://docs.python.org/3/library/typing.html#typing.Optional
[591] https://docs.python.org/3/library/functions.html#int
[592] https://docs.python.org/3/library/typing.html#typing.Optional
[593] https://docs.python.org/3/library/functions.html#float
[594] https://docs.python.org/3/library/typing.html#typing.Optional
[595] https://docs.python.org/3/library/functions.html#bool
[596] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[597] https://docs.python.org/3/library/functions.html#int
[598] https://docs.python.org/3/library/functions.html#float
[599] https://docs.python.org/3/library/functions.html#bool
[600] https://docs.python.org/3/library/exceptions.html#ValueError
[601] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[602] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[603] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[604] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[605] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[606] https://docs.python.org/3/library/functions.html#bool
[607] https://docs.python.org/3/library/functions.html#bool

- **covariance_2** (*np.ndarray*) – the second covariance matrix to compare
- **title** (*str*[618], *optional*) – the title for the comparison plot, defaults to *"Comparison between two vectors and corresponding uncertainties"*
- **label_1** (*str*[619], *optional*) – the label for the first vector in the legend and title for the first covariance plot, defaults to "vector_1"
- **label_2** (*str*[620], *optional*) – the label for the second vector in the legend and title for the second covariance plot, defaults to "vector_2"

PyDynamic.misc.tools.**print_mat**(*matrix*, *prec=5*, *vertical=False*, *retS=False*)

Print matrix (2D array) to the console or return as formatted string

Parameters

- **matrix** (*(M,N) array_like*) –
- **prec** (*int*[621]) – the precision of the output
- **vertical** (*bool*[622]) – print out vertical or not
- **retS** (*bool*[623]) – print or return string

Returns **s** – if retS is True

Return type str[624]

PyDynamic.misc.tools.**print_vec**(*vector*, *prec=5*, *retS=False*, *vertical=False*)

Print vector (1D array) to the console or return as formatted string

Parameters

- **vector** (*(M,) array_like*) –
- **prec** (*int*[625]) – the precision of the output
- **vertical** (*bool*[626]) – print out vertical or not
- **retS** (*bool*[627]) – print or return string

Returns **s** – if retS is True

Return type str[628]

[608] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[609] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[610] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[611] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[612] https://docs.python.org/3/library/typing.html#typing.Optional
[613] https://docs.python.org/3/library/stdtypes.html#str
[614] https://docs.python.org/3/library/typing.html#typing.Optional
[615] https://docs.python.org/3/library/stdtypes.html#str
[616] https://docs.python.org/3/library/typing.html#typing.Optional
[617] https://docs.python.org/3/library/stdtypes.html#str
[618] https://docs.python.org/3/library/stdtypes.html#str
[619] https://docs.python.org/3/library/stdtypes.html#str
[620] https://docs.python.org/3/library/stdtypes.html#str
[621] https://docs.python.org/3/library/functions.html#int
[622] https://docs.python.org/3/library/functions.html#bool
[623] https://docs.python.org/3/library/functions.html#bool
[624] https://docs.python.org/3/library/stdtypes.html#str
[625] https://docs.python.org/3/library/functions.html#int
[626] https://docs.python.org/3/library/functions.html#bool
[627] https://docs.python.org/3/library/functions.html#bool
[628] https://docs.python.org/3/library/stdtypes.html#str

PyDynamic.misc.tools.**progress_bar**(*count*, *count_max*, *width: Optional[629][int[630]] = 30*, *prefix: Optional[631][str[632]] = ''*, *done_indicator: Optional[633][str[634]] = '#'*, *todo_indicator: Optional[635][str[636]] = '.'*, *fout: Optional[637] = None*)

A simple and reusable progress-bar

Parameters

- **count** (*int*[638]) – current status of iterations, assumed to be zero-based
- **count_max** (*int*[639]) – total number of iterations
- **width** (*int*[640], *optional*) – width of the actual progressbar (actual printed line will be wider), default to 30
- **prefix** (*str*[641], *optional*) – some text that will be printed in front of the bar (i.e. "Progress of ABC:"), if not given only progressbar itself will be printed
- **done_indicator** (*str*[642], *optional*) – what character is used as "already-done"-indicator, defaults to "#"
- **todo_indicator** (*str*[643], *optional*) – what character is used as "not-done-yet"-indicator, defaults to "."
- **fout** (*file-object*, *optional*) – where the progress-bar should be written/printed to, defaults to direct print to stdout

PyDynamic.misc.tools.**real_imag_2_complex**(*array: numpy.ndarray*[644]) → numpy.ndarray[645]

Take a np.ndarray with real and imaginary parts and return dtype complex ndarray

The input array $x \in \mathbb{R}^{2m}$ representing a complex vector $y \in \mathbb{C}^m$ has the form of the expected input of some of the functions in the modules *propagate_DFT* and *fit_filter*: $x = (\mathrm{Re}(y), \mathrm{Im}(y))$ or a np.ndarray containing several of these.

Parameters **array** (*np.ndarray of shape (N,2M) or of shape (2M,)*) – the array of any integer or floating dtype to assemble the complex version of

Returns the complex array

Return type np.ndarray of shape (N,M) or of shape (M,)

PyDynamic.misc.tools.**separate_real_imag_of_mc_samples**(*array: numpy.ndarray*[646]) → List[647][numpy.ndarray[648]]

Split a np.ndarray containing MonteCarlo samples real and imaginary parts

629 https://docs.python.org/3/library/typing.html#typing.Optional
630 https://docs.python.org/3/library/functions.html#int
631 https://docs.python.org/3/library/typing.html#typing.Optional
632 https://docs.python.org/3/library/stdtypes.html#str
633 https://docs.python.org/3/library/typing.html#typing.Optional
634 https://docs.python.org/3/library/stdtypes.html#str
635 https://docs.python.org/3/library/typing.html#typing.Optional
636 https://docs.python.org/3/library/stdtypes.html#str
637 https://docs.python.org/3/library/typing.html#typing.Optional
638 https://docs.python.org/3/library/functions.html#int
639 https://docs.python.org/3/library/functions.html#int
640 https://docs.python.org/3/library/functions.html#int
641 https://docs.python.org/3/library/stdtypes.html#str
642 https://docs.python.org/3/library/stdtypes.html#str
643 https://docs.python.org/3/library/stdtypes.html#str
644 https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
645 https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

The input array $x \in \mathbb{R}^{n \times 2m}$ representing an n-elemental array of complex vectors $y_i \in \mathbb{C}^m$ has the form of the expected input of some of the functions in the modules *propagate_DFT* and *fit_filter*: $x = (\mathrm{Re}(y_i), \mathrm{Im}(y_i))_{i=1,\ldots,n}$.

Parameters **array** (*np.ndarray of shape (N,2M)*) – the array of any integer or floating dtype to assemble the complex version of

Returns two-element list of the two arrays containing the real and imaginary parts

Return type list of two np.ndarrays of shape (N,M)

PyDynamic.misc.tools.**separate_real_imag_of_vector**(*vector: numpy.ndarray*[649]) → List[650][numpy.ndarray[651]]

Split a np.ndarray containing real and imaginary parts into half

The input array $x \in \mathbb{R}^{2m}$ representing a complex vector $y \in \mathbb{C}^m$ has the form of the expected input of some of the functions in the modules *propagate_DFT* and *fit_filter*: $x = (\mathrm{Re}(y), \mathrm{Im}(y))$.

Parameters **vector** (*np.ndarray of shape (2M,)*) – the array of any integer or floating dtype to assemble the complex version of

Returns two-element list of the two arrays containing the real and imaginary parts

Return type list of two np.ndarrays of shape (M,)

PyDynamic.misc.tools.**shift_uncertainty**(*x: numpy.ndarray*[652], *ux: numpy.ndarray*[653], *shift: int*[654])

Shift the elements in the vector x and associated uncertainties ux

This function uses `numpy.roll()`[655] to shift the elements in x and ux. See the linked official documentation for details.

Parameters

- **x** (*np.ndarray of shape (N,)*) – vector of estimates
- **ux** (*float*[656], *np.ndarray of shape (N,) or of shape (N,N)*) – uncertainty associated with the vector of estimates
- **shift** (*int*[657]) – amount of shift

Returns

- **shifted_x** (*(N,) np.ndarray*) – shifted vector of estimates
- **shifted_ux** (*float, np.ndarray of shape (N,) or of shape (N,N)*) – uncertainty associated with the shifted vector of estimates

Raises **ValueError**[658] – If shift, x or ux are of unexpected type, dimensions of x and ux do not fit or ux is of unexpected shape

[646] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[647] https://docs.python.org/3/library/typing.html#typing.List
[648] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[649] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[650] https://docs.python.org/3/library/typing.html#typing.List
[651] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[652] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[653] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[654] https://docs.python.org/3/library/functions.html#int
[655] https://numpy.org/doc/stable/reference/generated/numpy.roll.html#numpy.roll
[656] https://docs.python.org/3/library/functions.html#float
[657] https://docs.python.org/3/library/functions.html#int
[658] https://docs.python.org/3/library/exceptions.html#ValueError

PyDynamic.misc.tools.**trimOrPad**(*array: Union*[659]*[List*[660]*, numpy.ndarray*[661]*], length: int*[662]*, mode: Optional*[663]*[str*[664]*] = 'constant'*)

Trim or pad (with zeros) a vector to the desired length

Either trim or zero-pad an array to achieve the required *length*. Both actions are applied to the end of the array.

Parameters

- **array** (*list*[665]*, 1D np.ndarray*) – original data
- **length** (*int*[666]) – length of output
- **mode** (*str*[667]*, optional*) – handed over to np.pad, default "constant"

Returns **array_modified** – An either trimmed or zero-padded array

Return type np.ndarray of shape (length,)

[659] https://docs.python.org/3/library/typing.html#typing.Union
[660] https://docs.python.org/3/library/typing.html#typing.List
[661] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[662] https://docs.python.org/3/library/functions.html#int
[663] https://docs.python.org/3/library/typing.html#typing.Optional
[664] https://docs.python.org/3/library/stdtypes.html#str
[665] https://docs.python.org/3/library/stdtypes.html#list
[666] https://docs.python.org/3/library/functions.html#int
[667] https://docs.python.org/3/library/stdtypes.html#str

CHAPTER

ELEVEN

SIGNAL

This module implements the signals class and its derivatives

Signals are dynamic quantities with associated uncertainties, quantity and time units. A signal has to be defined together with a time axis.

Note: This module is work in progress!

class PyDynamic.signals.**Signal**(*time: numpy.ndarray*[668], *values: numpy.ndarray*[669], *Ts: Optional*[670]*[float*[671]*] = None*, *Fs: Optional*[672]*[float*[673]*] = None*, *uncertainty: Optional*[674]*[Union*[675]*[float*[676]*, numpy.ndarray*[677]*]] = None*)

Signal class which represents a common signal in digital signal processing

Parameters

- **time** (*np.ndarray*) – the time axis as np.ndarray[678] of floats, number of elements must coincide with number of values
- **values** (*np.ndarray*) – signal values' magnitudes, number of elements must coincide with number of elements in time
- **Ts** (*float*[679]*, optional*) – the sampling interval length, i.e. the difference between each two time stamps, defaults to the reciprocal of the sampling frequency if provided and the mean of all unique interval lengths otherwise
- **Fs** (*float*[680]*, optional*) – the sampling frequency, defaults to the reciprocal of the sampling interval length
- **uncertainty** (*float*[681] *or np.ndarray, optional*) – the uncertainties associated with the signal values, depending on the type and shape the following should be provided:
  - float: constant standard uncertainty for all values
  - 1D-array: element-wise standard uncertainties
  - 2D-array: covariance matrix

property Fs: float[682]

Sampling frequency, i.e. the sampling interval *Ts*' reciprocal

property Ts: float[683]

Sampling interval, i.e. (averaged) difference between each two time stamps

apply_filter(*b: numpy.ndarray*[684], *a: Optional*[685]*[numpy.ndarray*[686]*] = array([1.])*, *filter_uncertainty: Optional*[687]*[numpy.ndarray*[688]*] = None*, *MonteCarloRuns: Optional*[689]*[int*[690]*] = 10000*)

Apply digital filter (b, a) to the signal values

Apply digital filter (b, a) to the signal values and propagate the uncertainty associated with the signal. Time vector is assumed to be equidistant, as well as corresponding values should represent evenly spaced signal magnitudes.

Parameters

- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray, optional*) – filter denominator coefficients, defaults to $a = (1)$ for FIR-type filter
- **filter_uncertainty** (*np.ndarray, optional*) – For IIR-type filter provide covariance matrix U_θ associated with filter coefficient vector $\theta = (a_1, \ldots, a_{N_a}, b_0, \ldots, b_{N_b})^T$. For FIR-type filter provide one of the following:
  - 1D-array: coefficient-wise standard uncertainties of filter
  - 2D-array: covariance matrix associated with theta

  if the filter is fully certain, use *filter_uncertainty* = *None* (default) to make use of more efficient calculations.
- **MonteCarloRuns** (*int*[691], *optional*) – number of Monte Carlo runs, defaults to 10.000, only considered for IIR-type filters. Otherwise *FIRuncFilter* is applied directly

property name: str[692]

Signal name

property standard_uncertainties: numpy.ndarray[693]

Element-wise standard uncertainties associated to *values*

property time: numpy.ndarray[694]

Signal's time axis

property uncertainty: numpy.ndarray[695]

Uncertainties associated with the signal *values*

Depending on the uncertainties provided during initialization, one of following will be provided:

- 1D-array: element-wise standard uncertainties
- 2D-array: covariance matrix

property unit_time: str[696]

Unit of the *time* vector

property unit_values: str[697]

Unit of the *values* vector

property values: numpy.ndarray[698]

Signal values' magnitudes

[668] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[669] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[670] https://docs.python.org/3/library/typing.html#typing.Optional
[671] https://docs.python.org/3/library/functions.html#float
[672] https://docs.python.org/3/library/typing.html#typing.Optional
[673] https://docs.python.org/3/library/functions.html#float
[674] https://docs.python.org/3/library/typing.html#typing.Optional
[675] https://docs.python.org/3/library/typing.html#typing.Union
[676] https://docs.python.org/3/library/functions.html#float
[677] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[678] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[679] https://docs.python.org/3/library/functions.html#float
[680] https://docs.python.org/3/library/functions.html#float
[681] https://docs.python.org/3/library/functions.html#float
[682] https://docs.python.org/3/library/functions.html#float
[683] https://docs.python.org/3/library/functions.html#float
[684] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[685] https://docs.python.org/3/library/typing.html#typing.Optional
[686] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[687] https://docs.python.org/3/library/typing.html#typing.Optional
[688] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[689] https://docs.python.org/3/library/typing.html#typing.Optional
[690] https://docs.python.org/3/library/functions.html#int
[691] https://docs.python.org/3/library/functions.html#int
[692] https://docs.python.org/3/library/stdtypes.html#str
[693] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[694] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[695] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray
[696] https://docs.python.org/3/library/stdtypes.html#str
[697] https://docs.python.org/3/library/stdtypes.html#str
[698] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html#numpy.ndarray

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THIRTEEN

REFERENCES

BIBLIOGRAPHY

[Eichst2016] S. Eichstädt und V. Wilkens GUM2DFT — a software tool for uncertainty evaluation of transient signals in the frequency domain. Meas. Sci. Technol., 27(5), 055001, 2016. https://dx.doi.org/10.1088/0957-0233/27/5/055001

[Eichst2012] S. Eichstädt, A. Link, P. M. Harris and C. Elster Efficient implementation of a Monte Carlo method for uncertainty evaluation in dynamic measurements Metrologia, vol 49(3), 401 https://dx.doi.org/10.1088/0026-1394/49/3/401

[Eichst2010] S. Eichstädt, C. Elster, T. J. Esward and J. P. Hessling Deconvolution filters for the analysis of dynamic measurement processes: a tutorial Metrologia, vol. 47, nr. 5 https://stacks.iop.org/0026-1394/47/i=5/a=003?key=crossref.310be1c501bb6b6c2056bc9d22ec93d4

[Elster2008] C. Elster and A. Link Uncertainty evaluation for dynamic measurements modelled by a linear time-invariant system Metrologia, vol 45 464-473, 2008 https://dx.doi.org/10.1088/0026-1394/45/4/013

[Link2009] A. Link and C. Elster Uncertainty evaluation for IIR filtering using a state-space approach Meas. Sci. Technol. vol. 20, 2009 https://dx.doi.org/10.1088/0957-0233/20/5/055104

[Vuer1996] R. Vuerinckx, Y. Rolain, J. Schoukens and R. Pintelon Design of stable IIR filters in the complex domain by automatic delay selection IEEE Trans. Signal Proc., 44, 2339-44, 1996 https://dx.doi.org/10.1109/78.536690

[Smith] Smith, J.O. Introduction to Digital Filters with Audio Applications, https://ccrma.stanford.edu/~jos/filters/, online book

[Savitzky] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of Data by Simplified Least Squares Procedures. Analytical Chemistry, 1964, 36 (8), pp 1627-1639.

[NumRec] Numerical Recipes 3rd Edition: The Art of Scientific Computing W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery Cambridge University Press ISBN-13: 9780521880688

[White2017] White, D.R. Int J Thermophys (2017) 38: 39. https://doi.org/10.1007/s10765-016-2174-6

[Zhivomirov2018] Zhivomirov, H. 2018. A Method for Colored Noise Generation. Romanian Journal of Acoustics and Vibration. 15, 1 (Aug. 2018), 14-19: http://rjav.sra.ro/index.php/rjav/article/view/40

PYTHON MODULE INDEX

p

INDEX

N

P

R

S

T

U

V

W