PyDynamic Documentation

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

Jun 12, 2020

Contents

1 Getting started 3

2 Evaluation of uncertainties 27

3 Model estimation 41

4 Design of deconvolution filters 47

5 Fitting filters and transfer functions models 49

6 Miscellaneous 51

7 Indices and tables 61

8 References 63

Bibliography 65

Python Module Index 67

Index 69

PyDynamic is a Python software package developed jointly by mathematicians from Physikalisch-Technische Bundesanstalt (Germany) and National Physical Laboratory (UK) as part of the joint European Research Project EMPIR 14SIP08 Dynamic[1].

For the PyDynamic homepage go to GitHub[2].

PyDynamic is written in Python 3 and strives to run with all Python versions with upstream support[3]. Currently it is tested to work with Python 3.5 to 3.8.

Contents:

[1] https://mathmet.org/projects/14SIP08
[2] https://github.com/PTB-PSt1/PyDynamic
[3] https://devguide.python.org/#status-of-python-branches

CHAPTER 1

Getting started

1.1 Installation

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

```
pip install PyDynamic
```

This will download the latest version from the Python package repository and copy it into your local folder of third-party libraries. 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
```

Updates of the software can be installed via

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

For collaboration we recommend using Github Desktop[4] or any other git-compatible version control software and cloning the repository[5]. In this way, any updates to the software will be highlighted in the version control software and can be applied very easily.

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[6].

[4] https://desktop.github.com
[5] https://github.com/PTB-PSt1/PyDynamic
[6] https://github.com/PTB-PSt1/PyDynamic/graphs/contributors

1.2 Quick Examples

On the project website you can find various examples illustrating the application of the software in the examples folder. Here is just a short list to get you started.

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])
```

1.3 Detailed examples

```
%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
```

1.3.1 Design of a digital deconvolution filter (FIR type)

```
from PyDynamic.deconvolution.fit_filter import LSFIR_unc
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
```

(continues on next page)

(continued from previous page)

```
# 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)
```

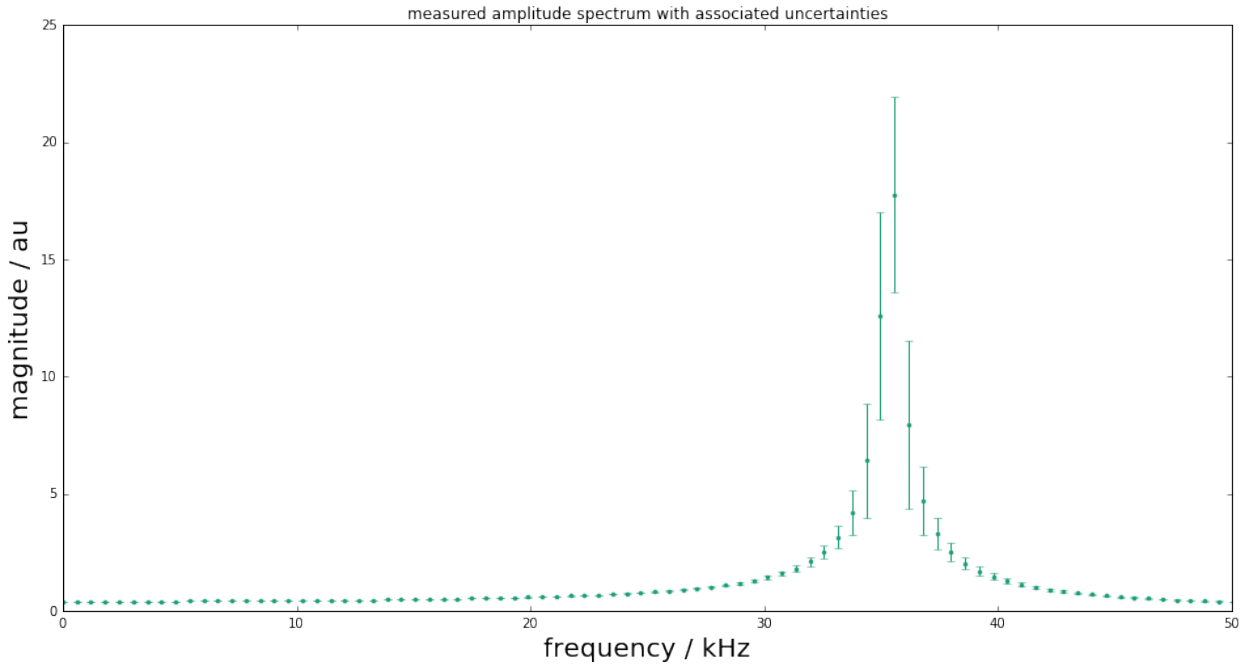
Problem description

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:

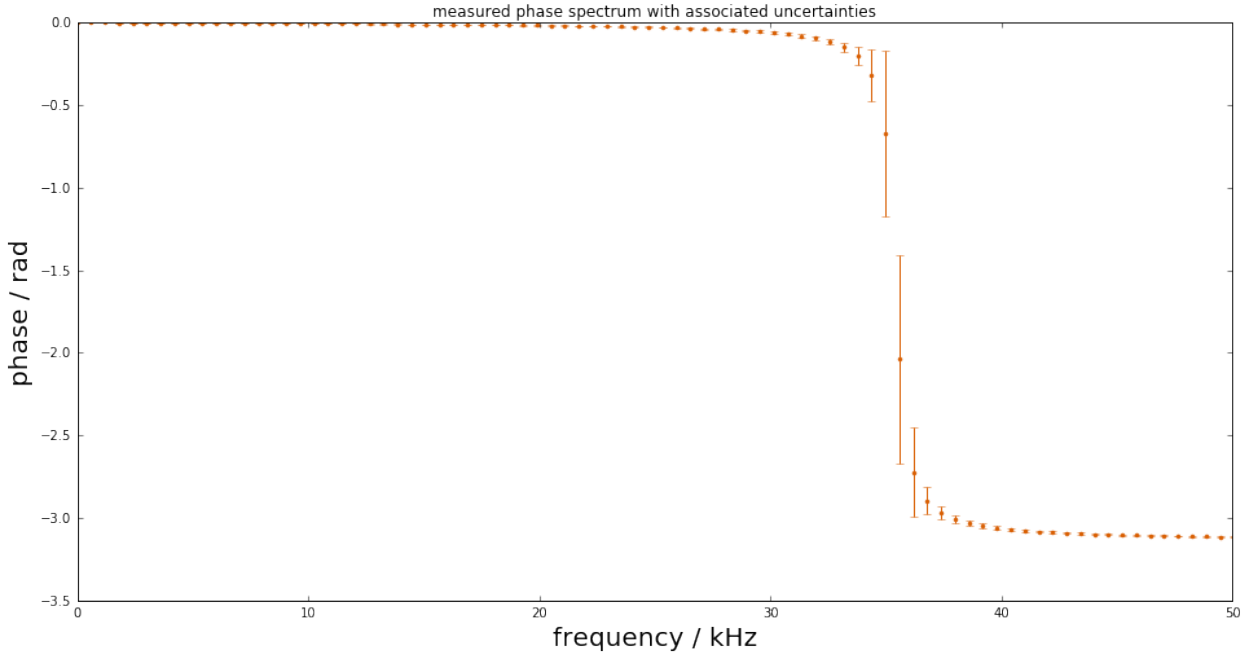
$$\mathbf{H} = (|H(j\omega_1)|, \ldots, |H(j\omega_N)|, \angle H(j\omega_1), \ldots, \angle H(j\omega_N)) \tag{1.1}$$

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

```
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

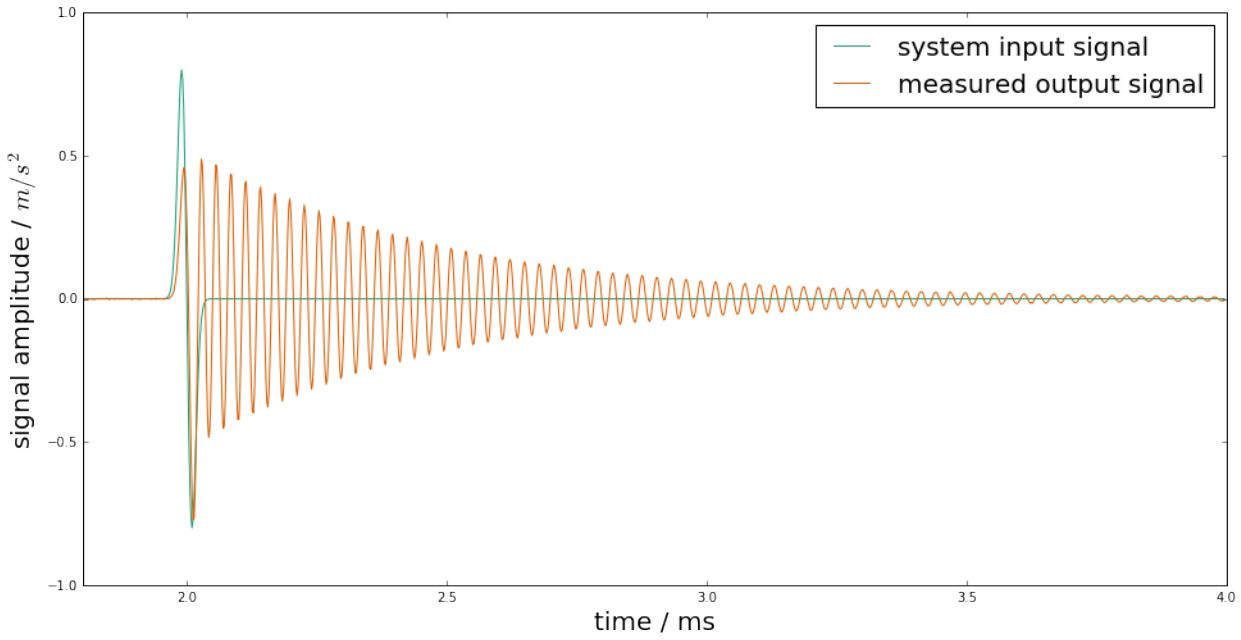
$$h(t) = \mathcal{F}^{-1}(H(j\omega))$$

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

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

<<<<<<< HEAD is an estimate of the system's input signal at the discrete time points ======= is an estimate of the system's input signal at the discrete time points. >>>>>>> devel1

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

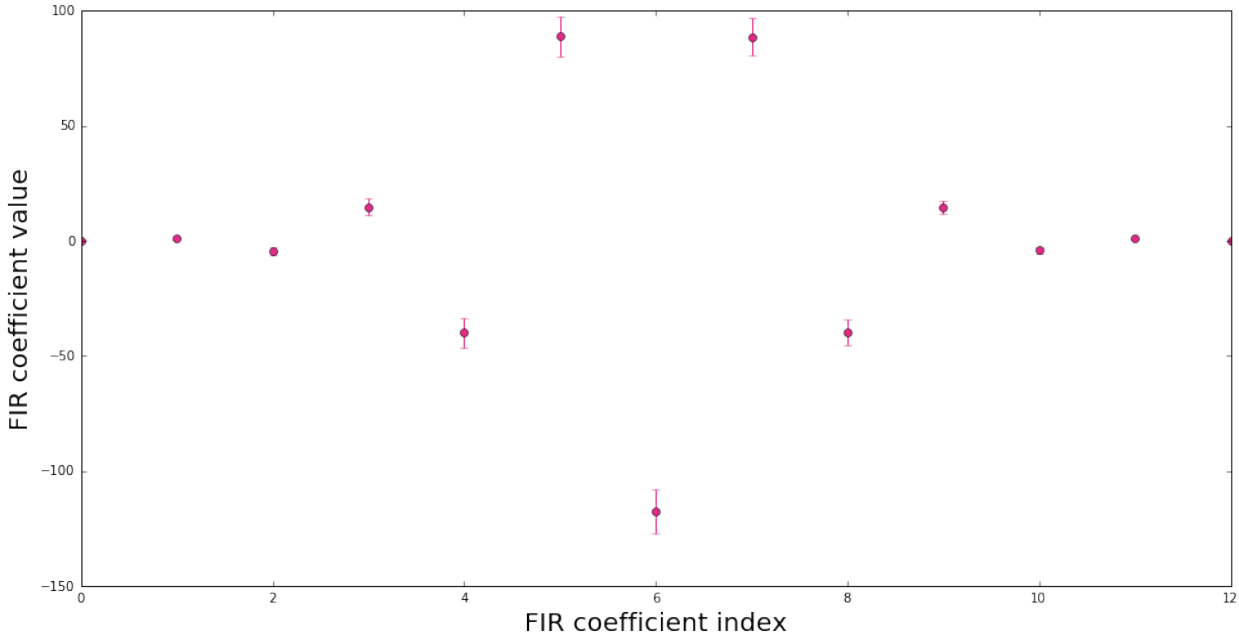
$$b = \left(X^T W^{-1} X\right)^{-1} X^T W^{-1} y$$

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

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

```
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

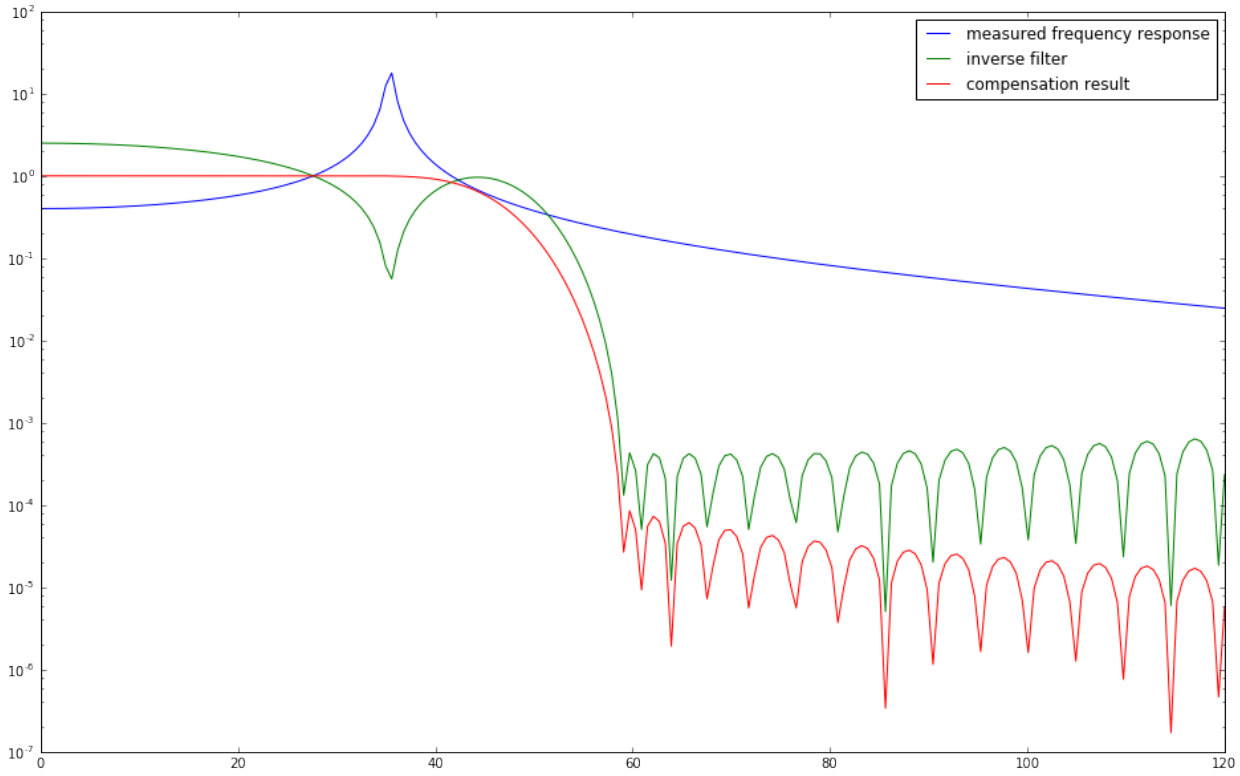
$$\hat{x}[n - n_0] = (g * (g_{low} * y)[n]$$

with point-wise associated uncertainties calculated as

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

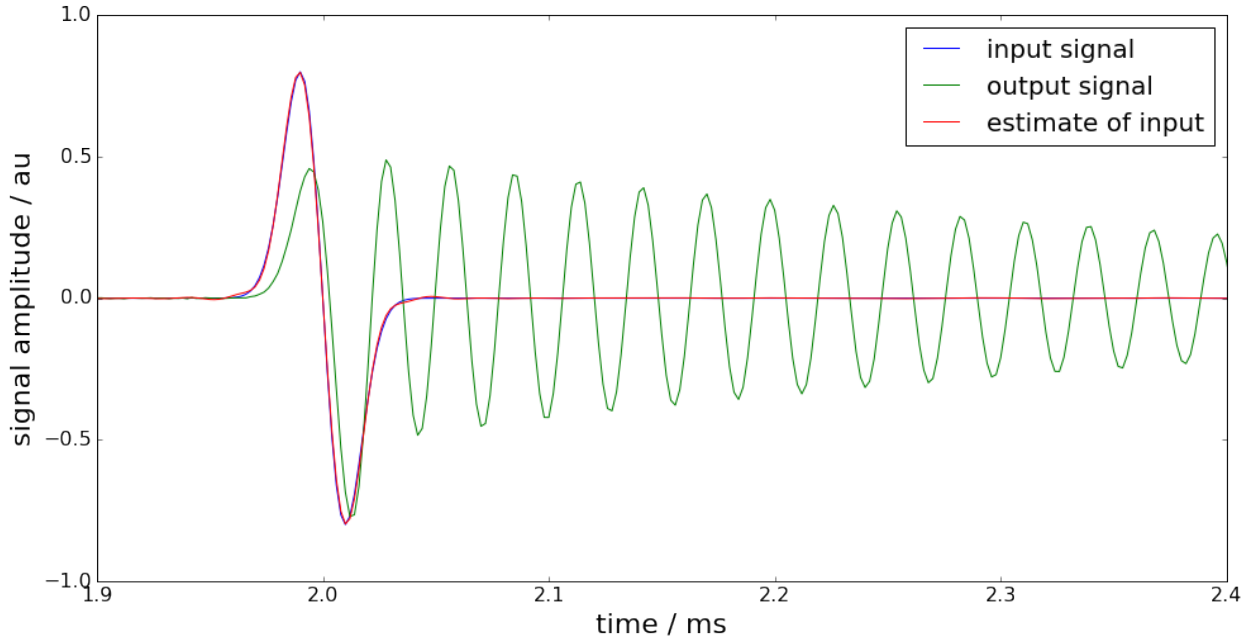
```
fcut = f0+10e3; low_order = 100
blow, lshift = kaiser_lowpass(low_order, fcut, Fs)
shift = -tau - lshift
```

```
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();
```

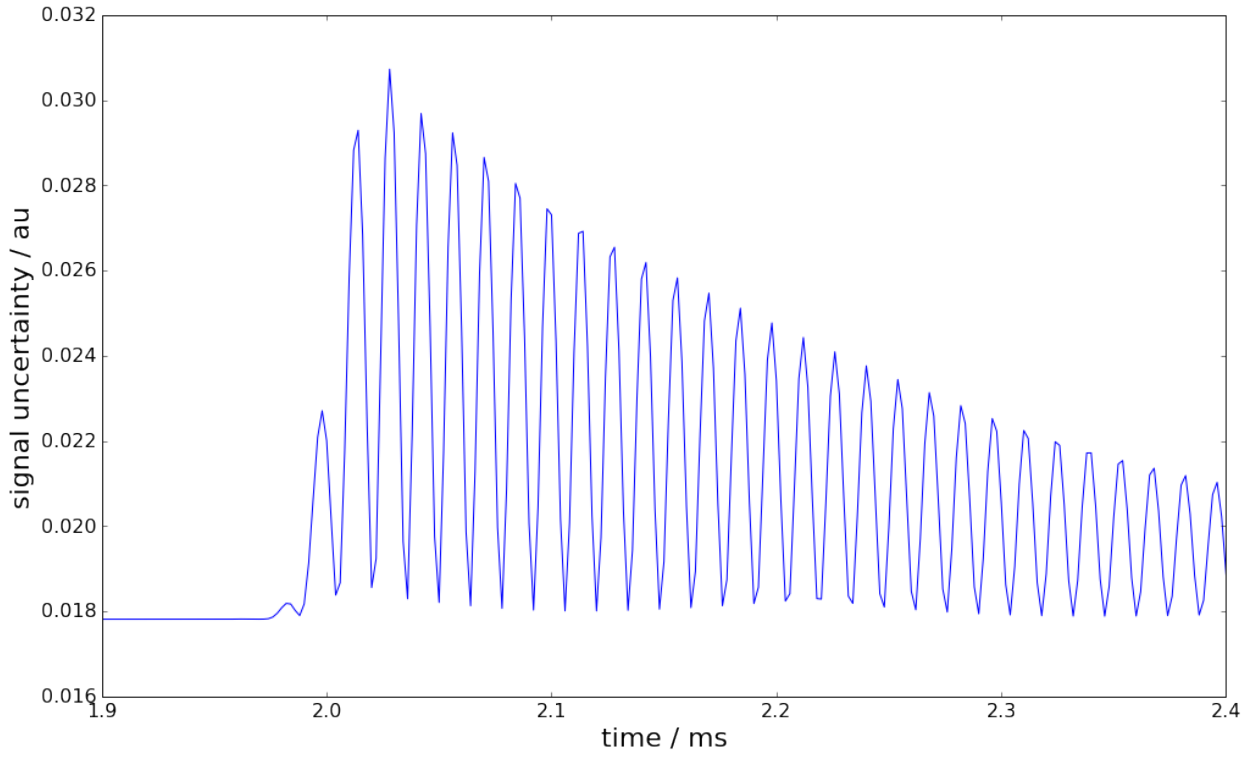



```
xhat,Uxhat = FIRuncFilter(yn,noise,bF,UbF,shift,blow)
```

```
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.deconvolution.fit_filter import LSFIR_unc
bF, UbF = LSFIR_unc(H,UH,N,tau,f,Fs, verbose=False)
```

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
```

1.3.2 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

$$z^\top[n+1] = \begin{pmatrix} -a_1 & \cdots & \cdots & -a_{N_a} \\ 0 & & & \\ \vdots & & I_{N_a-1} & \\ 0 & & & \end{pmatrix} z^\top[n] + \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} X[n], \tag{1.2}$$

$$y[n] = (d_1, \ldots, d_{N_a}) z^\top[n] + b_0 X[n] + \Delta[n], \tag{1.3}$$

$$y[n] = d^T z[n] + b_0 y[n] \tag{1.4}$$

$$u^2(y[n]) = \phi^T(n) U_\mu \phi(n) + d^T P_z[n] d + b_0^2, \tag{1.5}$$

where

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

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

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 $(a_1, \ldots, b_{N_b})$

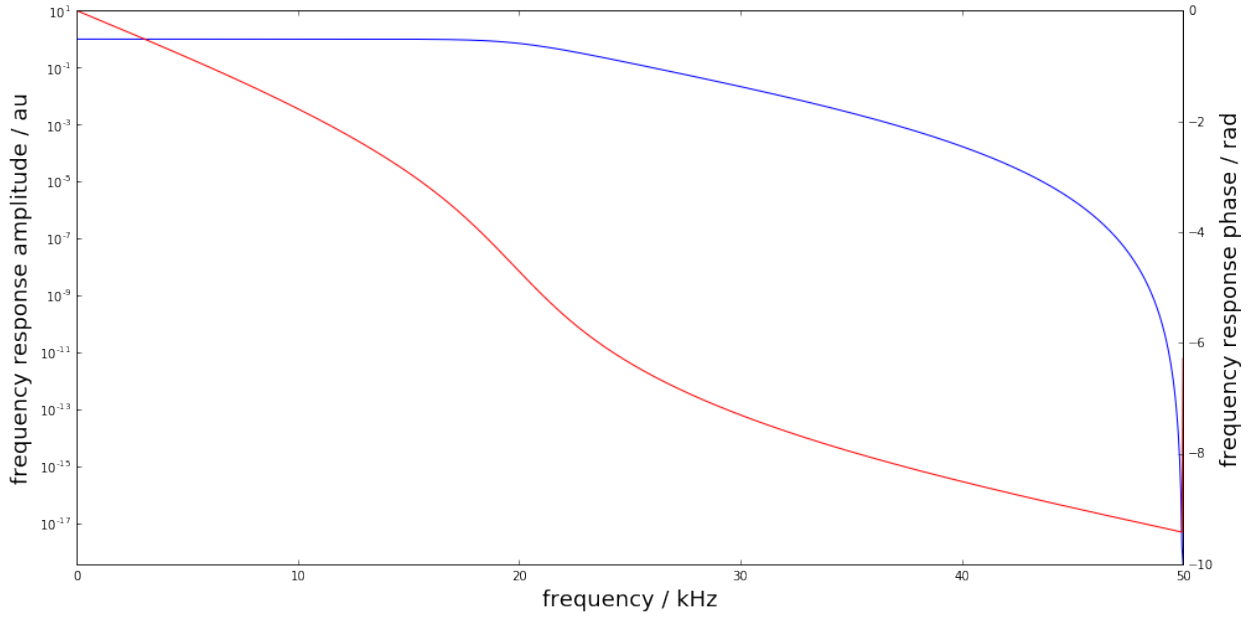
Remark

```
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);
```



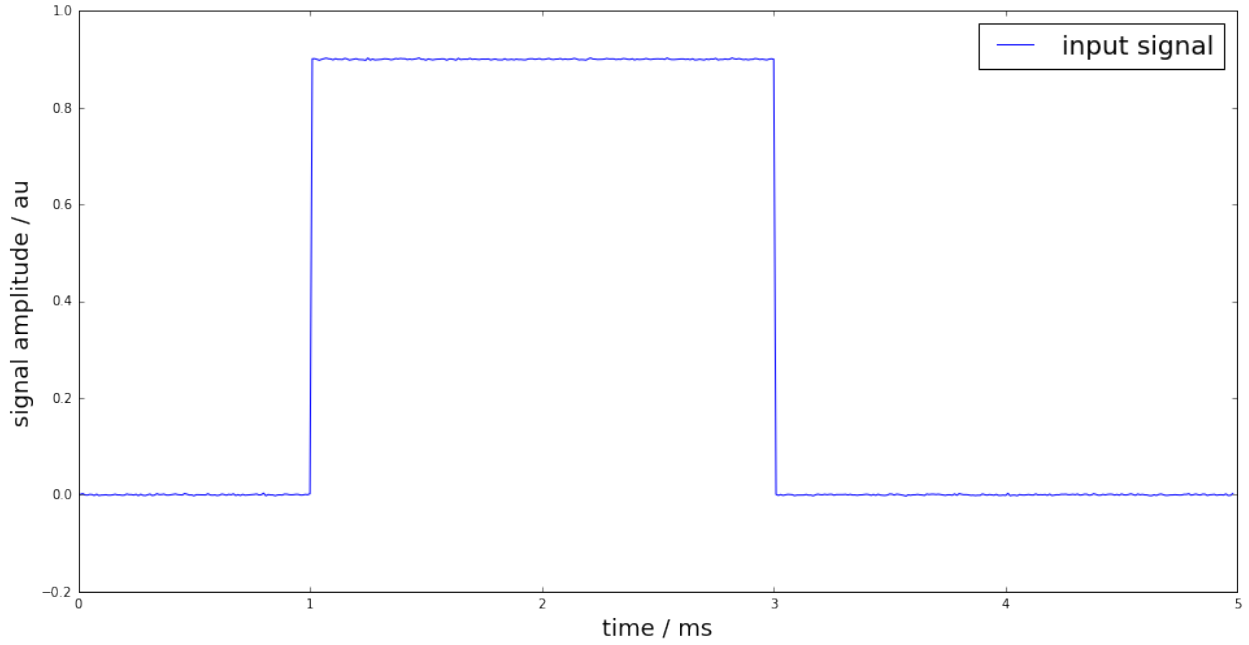

```
time = np.arange(0,499*Ts,Ts)
t0 = 100*Ts; t1 = 300*Ts
height = 0.9
noise = 1e-3
x = rect(time,t0,t1,height,noise=noise)
```

```
figure(figsize=(16,8))
plot(time*1e3, x, label="input signal")
```

(continues on next page)

(continued from previous page)

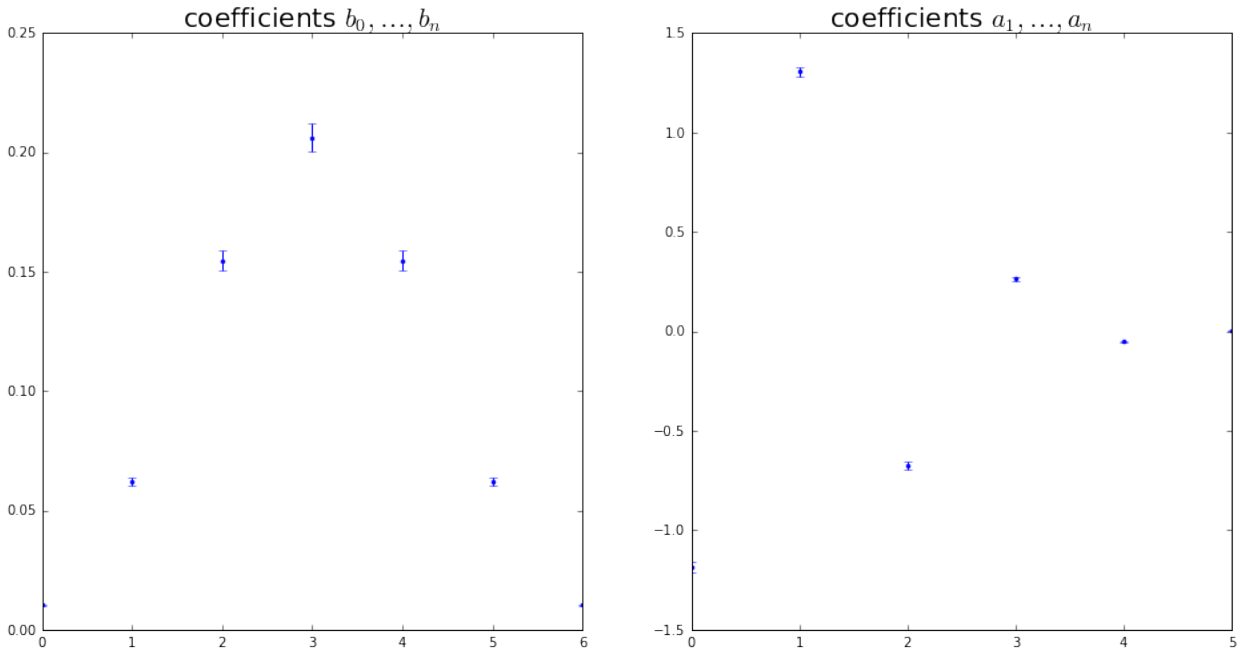
```
legend(fontsize=20)
xlabel('time / ms',fontsize=18)
ylabel('signal amplitude / au',fontsize=18);
```




```
# 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))
```

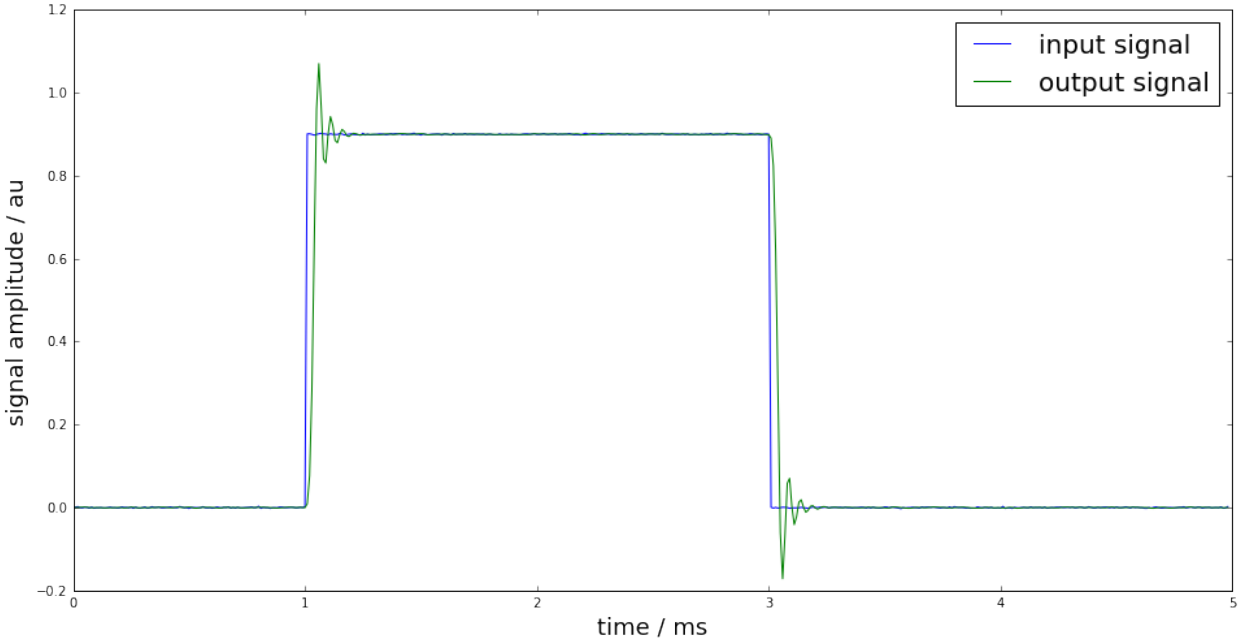
Uncertain knowledge: low-pass cut-off frequency is between 19.8 and 20.2 kHz

```
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);
```

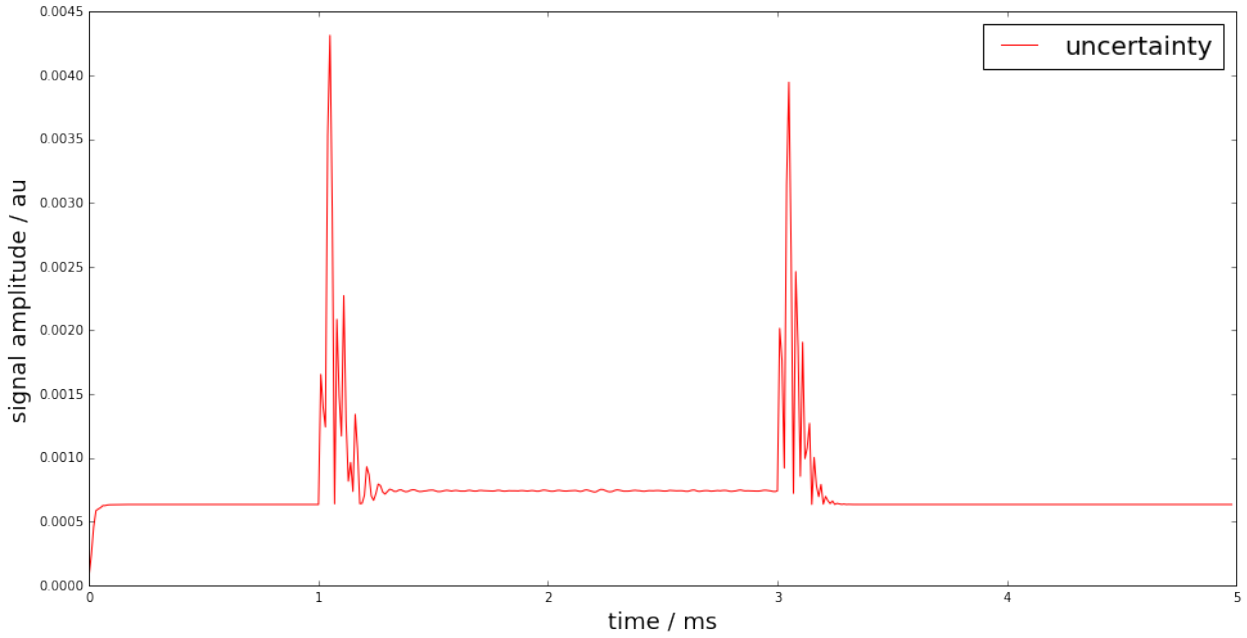


Estimate of the filter output signal and its associated uncertainty

```
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

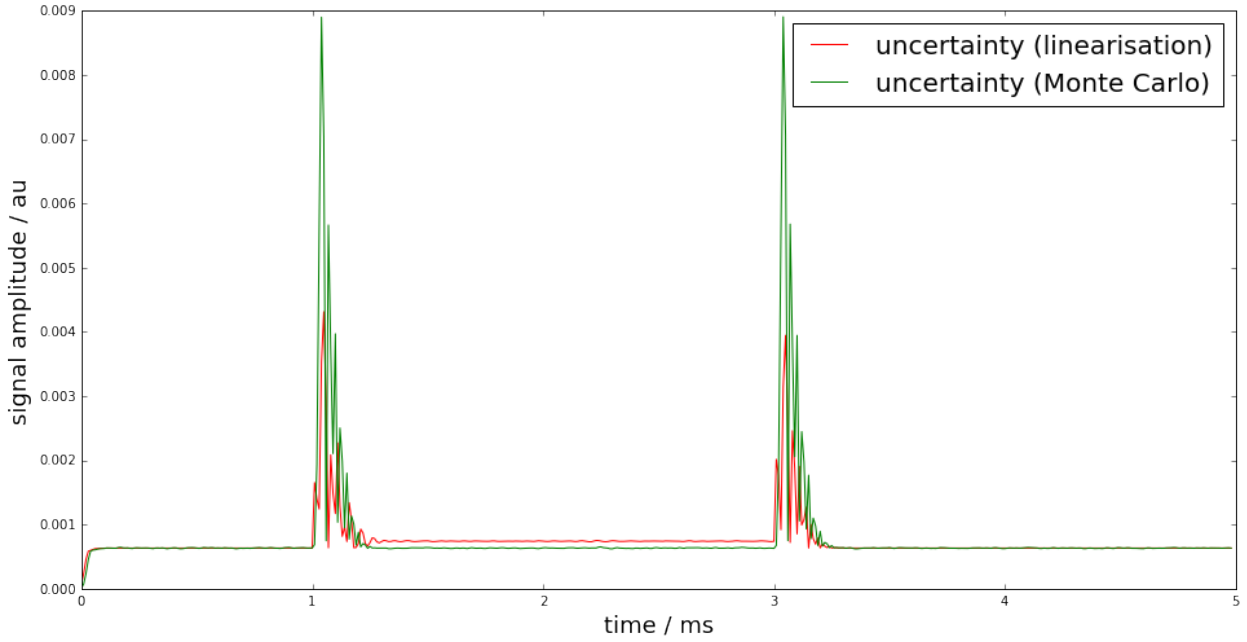
```
S. Eichstädt, A. Link, P. Harris, and C. Elster,
"Efficient implementation of a Monte Carlo method
for uncertainty evaluation in dynamic measurements,"
Metrologia, vol. 49, no. 3, 2012.
```

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
```

1.3.3 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
```

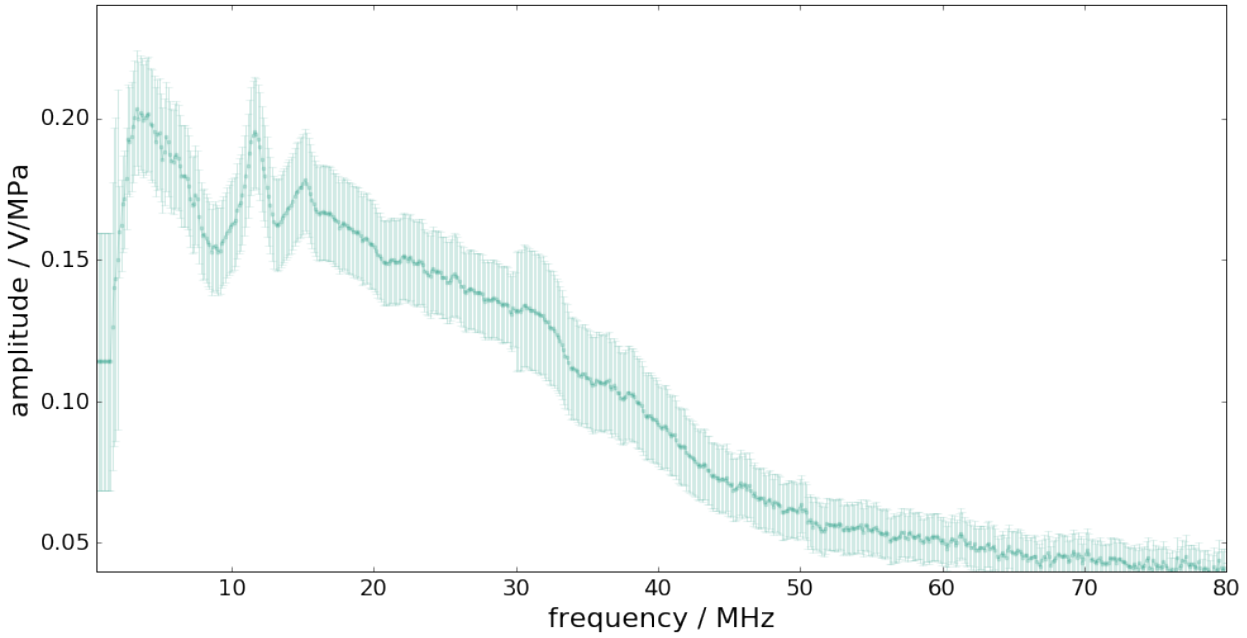
```
#%% reference data
ref_file = np.loadtxt("DFTdeconv reference_signal.dat")
time = ref_file[:,0]
ref_data = ref_file[:,1]
Ts = 2e-9
N = len(time)

#%% 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.

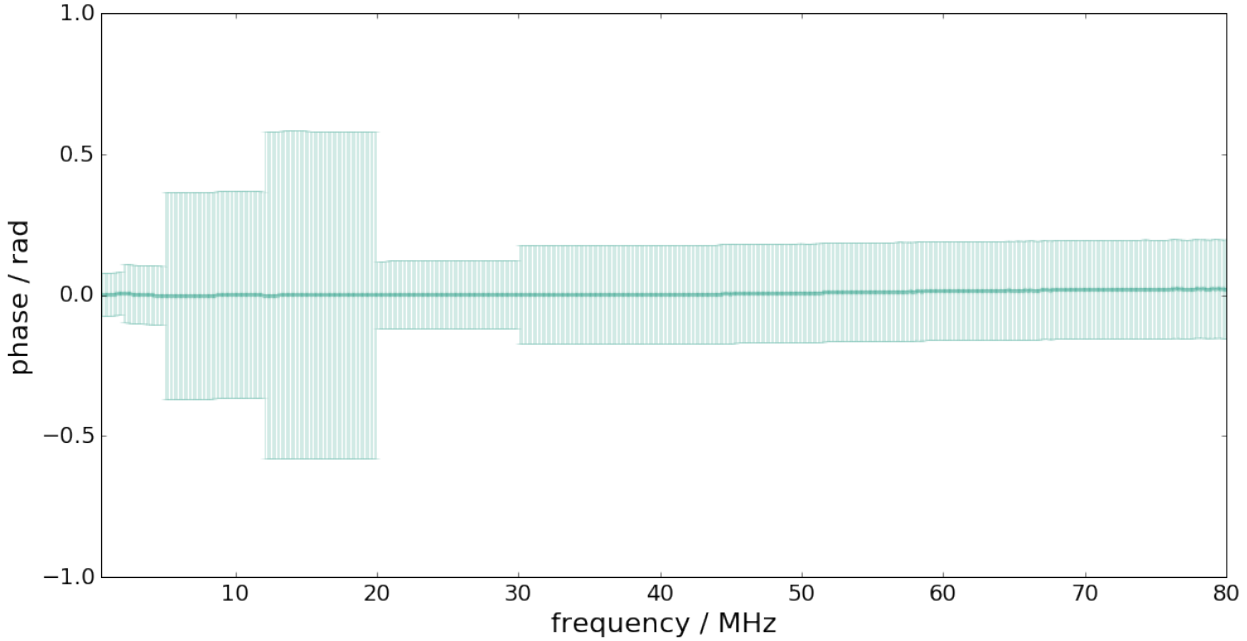
$$\mathbf{H} = (|H(f_1)|, \ldots, \angle H(f_N))$$

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

```
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

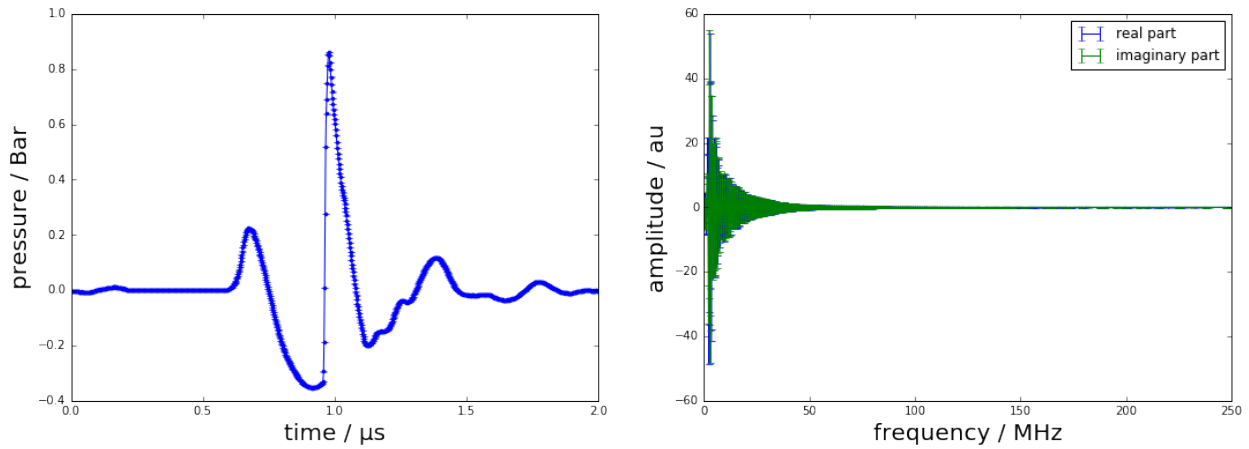
$$Y_k = \sum_{n=0}^{N-1} y_n \exp(-\mathrm{j}k\beta_n)$$

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

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

```
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.

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

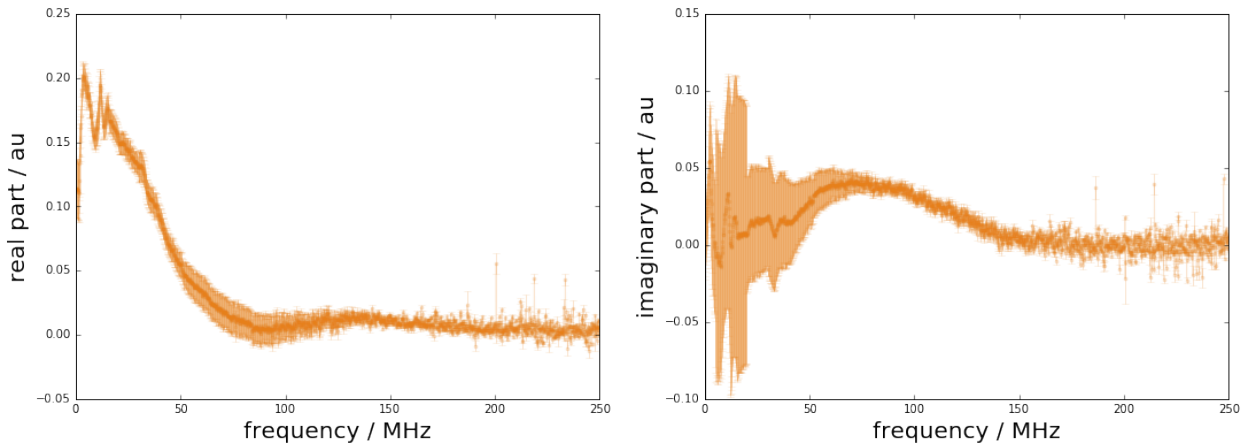
GUM uncertainty propagation

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

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

```
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

$$R_{RY} = \frac{\partial \Re_X}{\partial \Re_Y} = \frac{\Re_H}{\Re_H^2 + \Im_H^2} \tag{1.6}$$

$$R_{IY} = \frac{\partial \Re_X}{\partial \Im_Y} = \frac{\Im_H}{\Re_H^2 + \Im_H^2} \tag{1.7}$$

$$R_{RH} = \frac{\partial \Re_X}{\partial \Re_H} = \frac{-\Re_Y \Re_H^2 + \Re_Y \Im_H^2 - 2\Im_Y \Im_H \Re_H}{(\Re_H^2 + \Im_H^2)^2} \tag{1.8}$$

$$R_{IH} = \frac{\partial \Re_X}{\partial \Im_H} = \frac{\Im_Y \Re_H^2 - \Im_Y \Im_H^2 - 2\Re_Y \Re_H \Im_H}{(\Re_H^2 + \Im_H^2)^2} \tag{1.9}$$

$$I_{RY} = \frac{\partial \Im_X}{\partial \Re_Y} = \frac{-\Im_H}{\Re_H^2 + \Im_H^2} \tag{1.10}$$

$$I_{IY} = \frac{\partial \Im_X}{\partial \Im_Y} = \frac{\Re_H}{\Re_H^2 + \Im_H^2} \tag{1.11}$$

$$I_{RH} = \frac{\partial \Im_X}{\partial \Re_H} = \frac{-\Im_Y \Re_H^2 + \Im_Y \Im_H^2 + 2\Re_Y \Im_H \Re_H}{(\Re_H^2 + \Im_H^2)^2} \tag{1.12}$$

$$I_{IH} = \frac{\partial \Im_X}{\partial \Im_H} = \frac{-\Re_Y \Re_H^2 + \Re_Y \Im_H^2 - 2\Im_Y \Re_H \Im_H}{(\Re_H^2 + \Im_H^2)^2} \tag{1.13}$$

Uncertainty blocks for multiplication part

$$U_{XRR} = \Re_{H_L} U_{ARR} \Re_{H_L} - \Im_{H_L} U_{ARI}^T \Re_{H_L} - \Re_{H_L} U_{ARI} \Im_{H_L} + \Im_{H_L} U_{AII} \Im_{H_L} \tag{1.14}$$

$$U_{XRI} = \Re_{H_L} U_{ARR} \Im_{H_L} - \Im_{H_L} U_{ARI}^T \Im_{H_L} + \Re_{H_L} U_{ARI} \Re_{H_L} - \Im_{H_L} U_{AII} \Re_{H_L} \tag{1.15}$$

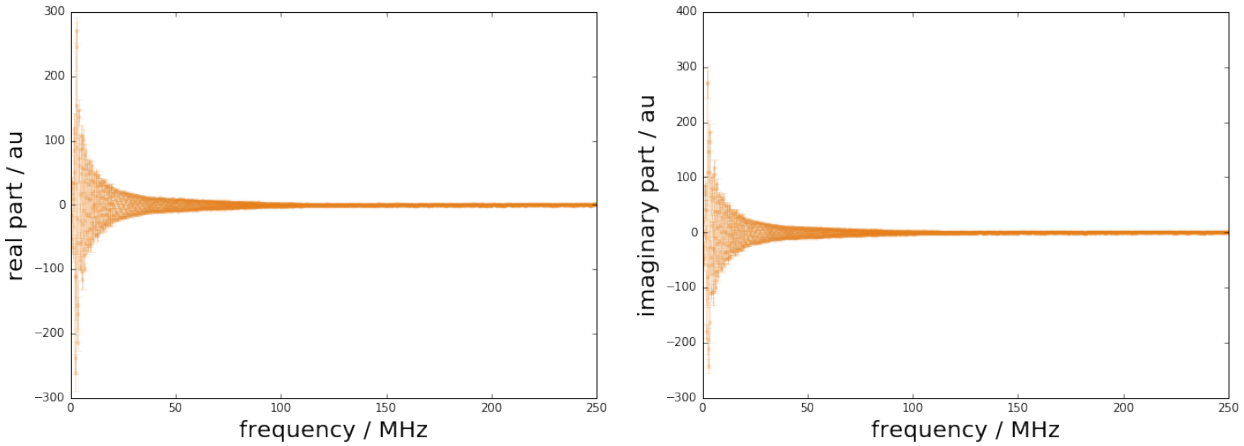
$$U_{XIR} = U_{YRI}^T \tag{1.16}$$

$$U_{XII} = \Im_{H_L} U_{ARR} \Im_{H_L} + \Re_{H_L} U_{ARI}^T \Im_{H_L} + \Im_{H_L} U_{ARI} \Re_{H_L} + \Re_{H_L} U_{AII} \Re_{H_L} \tag{1.17}$$

```
# 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

$$\frac{\partial X_n}{\partial \Re_k} = \frac{1}{N} \quad \text{for } k = 0 \tag{1.18}$$

$$\frac{\partial X_n}{\partial \Re_k} = \frac{2}{N}\cos(k\beta_n) \quad \text{for } k = 1,\ldots,N/2-1 \tag{1.19}$$

$$\frac{\partial X_n}{\partial \Im_k} = 0 \quad \text{for } k = 0 \tag{1.20}$$

$$\frac{\partial X_n}{\partial \Im_k} = -\frac{2}{N}\sin(k\beta_n) \quad \text{for } k = 1,\ldots,N/2-1. \tag{1.21}$$

GUM uncertainty propagation for the inverse DFT

$$C_F U_F C_F^{\mathsf{T}} = \left(\widetilde{C}_{\cos}, \widetilde{C}_{\sin}\right)\begin{pmatrix} U_{RR} & U_{RI} \\ U_{IR} & U_{II} \end{pmatrix}\begin{pmatrix} \widetilde{C}_{\cos}^{\mathsf{T}} \\ \widetilde{C}_{\sin}^{\mathsf{T}} \end{pmatrix} \tag{1.22}$$

$$= \widetilde{C}_{\cos} U_{RR} \widetilde{C}_{\cos}^{\mathsf{T}} + 2\widetilde{C}_{\cos} U_{RI} \widetilde{C}_{\sin}^{\mathsf{T}} + \widetilde{C}_{\sin} U_{II} \widetilde{C}_{\sin}^{\mathsf{T}} \tag{1.23}$$

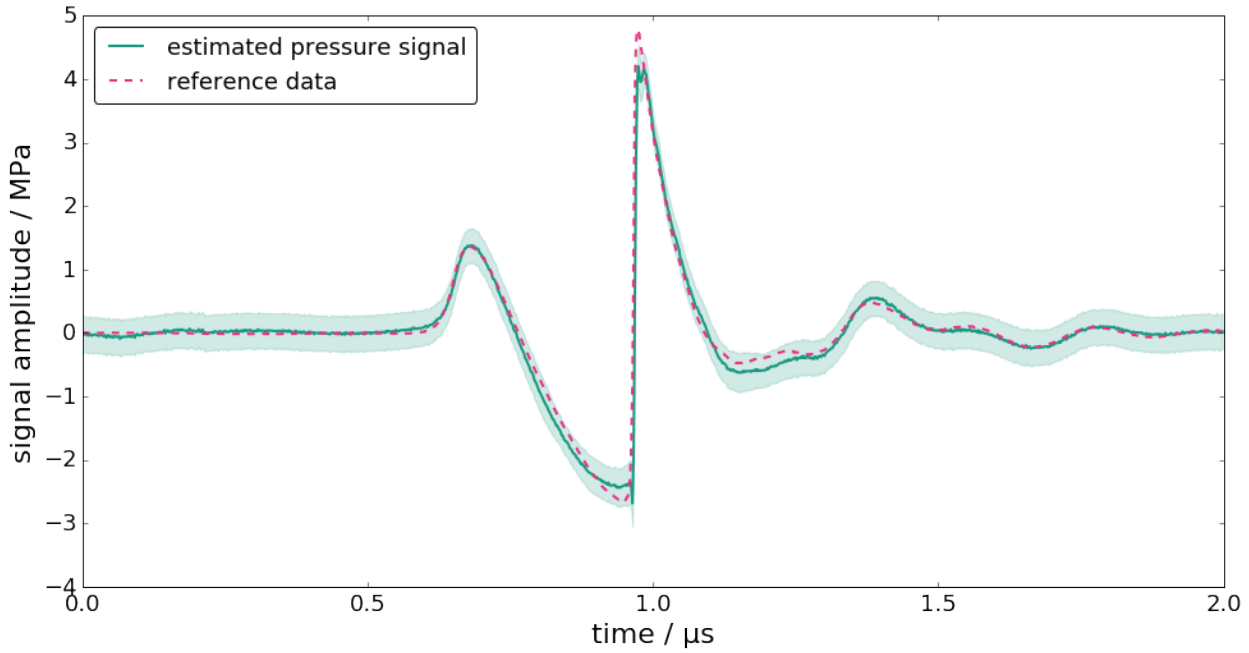
```
xh,Uxh = GUM_iDFT(XH,UXH,Nx=N)
```

```
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])
```

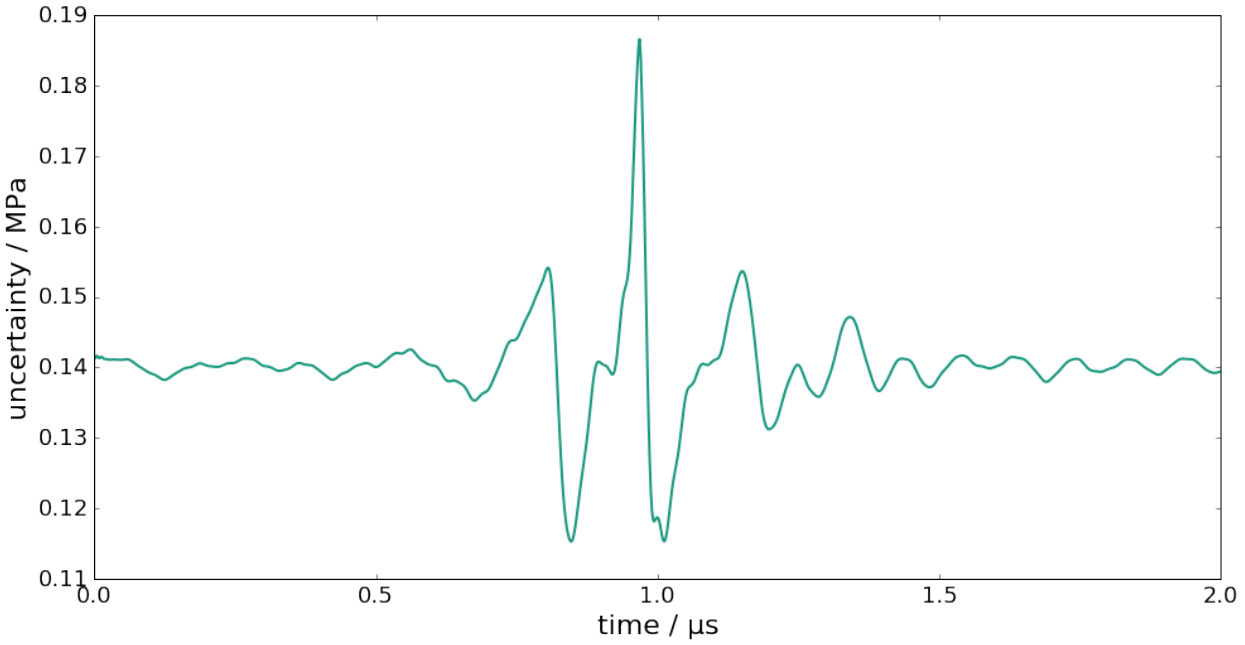
(continues on next page)

(continued from previous page)

```
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)
```

CHAPTER 2

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`[7]. 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_filter*: Uncertainty evaluation for digital filtering
- *PyDynamic.uncertainty.propagate_MonteCarlo*: Monte Carlo methods for digital filtering
- *PyDynamic.uncertainty.interpolation*: Uncertainty evaluation for interpolation

2.1 Uncertainty evaluation for the DFT

The *PyDynamic.uncertainty.propagate_DFT* module implements methods 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[8]]

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

[7] https://docs.scipy.org/doc/scipy/reference/signal.html#module-scipy.signal
[8] http://dx.doi.org/10.1088/0957-0233/27/5/055001

- *GUM_DFTfreq()*: Return the Discrete Fourier Transform sample frequencies
- *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.**GUM_DFT**(*x*, *Ux*, *N=None*, *window=None*, *CxCos=None*, *CxSin=None*, *returnC=False*, *mask=None*)

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** (*numpy.ndarray of shape (M,)*) – vector of time domain signal values
- **Ux** (*numpy.ndarray*) – covariance matrix associated with x, shape (M,M) or vector of squared standard uncertainties, shape (M,) or noise variance as float
- **N** (*int, optional*) – length of time domain signal for DFT; N>=len(x)
- **window** (*numpy.ndarray, optional of shape (M,)*) – vector of the time domain window values
- **CxCos** (*numpy.ndarray, optional*) – cosine part of sensitivity matrix
- **CxSin** (*numpy.ndarray, optional*) – sine part of sensitivity matrix
- **returnC** (*bool, optional*) – if true, return sensitivity matrix blocks for later use
- **mask** (*ndarray of dtype bool*) – calculate DFT values and uncertainties only at those frequencies where mask is *True*

Returns

- **F** (*numpy.ndarray*) – vector of complex valued DFT values or of its real and imaginary parts
- **UF** (*numpy.ndarray*) – covariance matrix associated with real and imaginary part of F

References

- Eichstädt and Wilkens [Eichst2016]

PyDynamic.uncertainty.propagate_DFT.**GUM_iDFT**(*F*, *UF*, *Nx=None*, *Cc=None*, *Cs=None*, *returnC=False*)

GUM propagation of the squared uncertainty 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, 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** (*if true, return sensitivity matrix blocks (without scaling factor 1/N)*) –

Returns

- **x** (*np.ndarray*) – vector of time domain signal values
- **Ux** (*np.ndarray*) – covariance matrix associated with x

References

- Eichstädt and Wilkens [Eichst2016]

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

Return the Discrete Fourier Transform sample frequencies

Parameters

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

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

Return type ndarray

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

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

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.**DFT_deconv**(*H*, *Y*, *UH*, *UY*)

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 (N an even integer)
- **Y** (*np.ndarray of shape (2M,)*) – real and imaginary parts of DFT values
- **UH** (*np.ndarray of shape (2M,2M)*) – covariance matrix associated with H
- **UY** (*np.ndarray of shape (2M,2M)*) – 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)*) – covariance matrix associated with real and imaginary part of X

References

- Eichstädt and Wilkens [Eichst2016]

PyDynamic.uncertainty.propagate_DFT.**DFT_multiply**(*Y*, *F*, *UY*, *UF=None*)

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 shape (2M,) or shape (2M,2M)*) – covariance matrix or squared uncertainty associated with Y
- **UF** (*np.ndarray of shape (2M,2M)*) – covariance matrix associated with F (optional), default is None

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

PyDynamic.uncertainty.propagate_DFT.**AmpPhase2DFT**(*A*, *P*, *UAP*, *keep_sparse=False*)

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)*) – covariance matrix associated with (A,P) or vector of squared standard uncertainties [u^2(A),u^2(P)]

- **keep_sparse** (*bool, optional*) – whether to transform sparse matrix to numpy array or not

Returns

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

PyDynamic.uncertainty.propagate_DFT.**DFT2AmpPhase**(*F*, *UF*, *keep_sparse=False*, *tol=1.0*, *return_type='separate'*)

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, optional*) – if true then UAP will be sparse if UF is one-dimensional
- **tol** (*float, optional*) – lower bound for A/uF below which a warning will be issued concerning unreliable results
- **return_type** (*str, optional*) – If "separate" then magnitude and phase are returned as separate arrays. Otherwise the array [A, P] is returned

If *return_type* is *separate*:

Returns

- **A** (*np.ndarray*) – vector of magnitude values
- **P** (*np.ndarray*) – vector of phase values in radians, in the range [-pi, pi]
- **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

PyDynamic.uncertainty.propagate_DFT.**AmpPhase2Time**(*A*, *P*, *UAP*)

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
- **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*) – vector of time domain values

- **Ux** (*np.ndarray*) – covariance matrix associated with x

PyDynamic.uncertainty.propagate_DFT.**Time2AmpPhase**(*x*, *Ux*)
Transformation from time domain to amplitude and phase

Parameters

- **x** (*np.ndarray of shape (N,)*) – time domain signal
- **Ux** (*np.ndarray of shape (N,N)*) – squared uncertainty associated with x

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,N)*) – M time domain signals of length N
- **Ux** (*np.ndarray of shape (M,)*) – squared standard deviations representing noise variances of the signals x
- **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(UPP), diag(UAA), diag(UPA)]

2.2 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)

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'*)
Uncertainty propagation for signal y and uncertain FIR filter theta

Parameters

- **y** (*np.ndarray*) – filter input signal
- **sigma_noise** (*float or np.ndarray*) – float: standard deviation of white noise in y 1D-array: interpretation depends on kind
- **theta** (*np.ndarray*) – FIR filter coefficients
- **Utheta** (*np.ndarray*) – covariance matrix associated with theta
- **shift** (*int*) – time delay of filter output signal (in samples)
- **blow** (*np.ndarray*) – optional FIR low-pass filter
- **kind** (*string*) – only meaningfull in combination with isinstance(sigma_noise, numpy.ndarray) "diag": point-wise standard uncertainties of non-stationary white noise "corr": single sided autocovariance of stationary (colored/corrlated) noise (default)

Returns

- **x** (*np.ndarray*) – FIR filter output signal
- **ux** (*np.ndarray*) – point-wise uncertainties associated with x

References

- Elster and Link 2008 [Elster2008]

See also:

`PyDynamic.deconvolution.fit_filter`

`PyDynamic.uncertainty.propagate_filter.`**`IIRuncFilter`**(*x*, *noise*, *b*, *a*, *Uab*)

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

Parameters

- **x** (*np.ndarray*) – filter input signal
- **noise** (*float*) – signal noise standard deviation
- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray*) – filter denominator coefficients
- **Uab** (*np.ndarray*) – covariance matrix for (a[1:],b)

Returns

- **y** (*np.ndarray*) – filter output signal
- **Uy** (*np.ndarray*) – uncertainty associated with y

References

- Link and Elster [Link2009]

2.3 Monte Carlo methods 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 implements Monte Carlo methods for the propagation of uncertainties for digital filtering.

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 or np.ndarray*) – standard deviation of signal noise (float), point-wise standard uncertainties or covariance matrix associated with x
- **b** (*np.ndarray*) – filter numerator coefficients
- **a** (*np.ndarray*) – filter denominator coefficients
- **Uab** (*np.ndarray*) – uncertainty matrix U_θ
- **runs** (*int, optional*) – number of Monte Carlo runs
- **return_samples** (*bool, optional*) – whether samples or mean and std are returned

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

Returns

- **y** (*np.ndarray*) – filter output signal
- **Uy** (*np.ndarray*) – uncertainty associated with

Otherwise the method returns:

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

Return type np.ndarray

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*) – 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, optional*) – number of Monte Carlo runs
- **Perc** (*list, 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*) – integer for time delay of output signals
- **return_samples** (*bool, otpional*) – whether to return y and Uy or the matrix Y of MC results
- **theta** (*phi,*) – 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 $w(n) \sim N(0, noise_std^2)$
- **Delta** (*float,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, optional*) – number of Monte Carlo runs
- **blocksize** (*int, optional*) – how many samples should be evaluated for at a time
- **blow** (*float or np.ndarray, optional*) – filter coefficients of optional low pass filter
- **alow** (*float 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, optional*) – see misc.noise.ARMA noise model
- **Delta** (*float, optional*) – upper bound of systematic correction due to regularisation (assume uniform distribution)
- **runs_init** (*int, optional*) – how many samples to evaluate to form initial guess about limits
- **nbins** (*int, list of int, optional*) – number of bins for histogram
- **credible_interval** (*float, 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, optional*) – number of Monte Carlo runs
- **blocksize** (*int, optional*) – how many samples should be evaluated for at a time
- **runs_init** (*int, optional*) – how many samples to evaluate to form initial guess about limits
- **nbins** (*int, list of int, optional*) – number of bins for histogram
- **return_samples** (*bool, optional*) – see return-value of documentation
- **n_cpu** (*int, 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

References

- Eichstädt, Link, Harris, Elster [Eichst2012]

2.4 Uncertainty evaluation for interpolation

The *PyDynamic.uncertainty.interpolation* module implements methods for the propagation of uncertainties in the application of standard interpolation methods as provided by scipy.interpolate.interp1d[9].

This module contains the following function:

- *interp1d_unc()*: Interpolate arbitrary time series considering the associated uncertainties

PyDynamic.uncertainty.interpolation.**interp1d_unc**(*t_new: numpy.ndarray, t: numpy.ndarray, y: numpy.ndarray, uy: numpy.ndarray, kind: Optional[str] = 'linear', copy=True, bounds_error: Optional[bool] = None, fill_value: Union[float, Tuple[float, float], str, None] = nan, fill_unc: Union[float, Tuple[float, float], str, None] = nan, assume_sorted: Optional[bool] = True, returnC: Optional[bool] = False*) → Union[Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray], Tuple[numpy.ndarray, numpy.ndarray, numpy.ndarray, numpy.ndarray]]

Interpolate a 1-D function considering the associated uncertainties

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

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

An equal number of each of the original timestamps (or frequencies), values and associated uncertainties is required.

Parameters

- **t_new** (*(M,) array_like*) – A 1-D array of real values representing the timestamps (or frequencies) at which to evaluate the interpolated values. t_new can be sorted in any order.
- **t** (*(N,) array_like*) – A 1-D array of real values representing timestamps (or frequencies) in ascending order.
- **y** (*(N,) array_like*) – A 1-D array of real values. The length of y must be equal to the length of t.
- **uy** (*(N,) array_like*) – A 1-D array of real values representing the standard uncertainties associated with y.
- **kind** (*str, optional*) – Specifies the kind of interpolation for y as a string ('previous', 'next', 'nearest' or 'linear'). Default is 'linear'.
- **copy** (*bool, optional*) – If True, the method makes internal copies of t and y. If False, references to t and y are used. The default is to copy.
- **bounds_error** (*bool, 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"*.

[9] https://docs.scipy.org/doc/scipy/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.
  - If a two-element tuple, then the first element is used as a fill value for *t_new < t[0]* and the second element is used for *t_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.

  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, optional*) – If False, values of t can be in any order and they are sorted first. If True, t has to be an array of monotonically increasing values.
- **returnC** (*bool, 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.

If *returnC* is False, which is the default behaviour, the method returns:

Returns

- **t_new** (*(M,) array_like*) – interpolation timestamps (or frequencies)
- **y_new** (*(M,) array_like*) – interpolated values
- **uy_new** (*(M,) array_like*) – interpolated associated standard uncertainties

Otherwise the method returns:

Returns

- **t_new** (*(M,) array_like*) – interpolation timestamps (or frequencies)
- **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 $U_{y_{new}} = C \cdot \mathrm{diag}(u_y^2) \cdot C^T$

References

- White [White2017]

CHAPTER 3

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

3.1 Fitting filters to frequency response or reciprocal

The module `PyDynamic.model_estimation.fit_filter` contains several functions to carry out a least-squares fit 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 associated uncertainties.

This module contains the following functions:

- `LSIIR()`: Least-squares IIR filter fit to a given frequency response
- `LSFIR()`: Least-squares fit of a digital FIR filter to a given frequency response
- `invLSFIR()`: Least-squares fit of a digital FIR filter to the reciprocal of a given frequency response.
- `invLSFIR_unc()`: Design of FIR filter as fit to reciprocal of frequency response values with uncertainty
- `invLSFIR_uncMC()`: Design of FIR filter as fit to reciprocal of frequency response values with uncertainty via Monte Carlo
- `invLSIIR()`: Design of a stable IIR filter as fit to reciprocal of frequency response values

- *invLSIIR_unc()*: Design of a stable IIR filter as fit to reciprocal of frequency response values with uncertainty

PyDynamic.model_estimation.fit_filter.**LSIIR**(*Hvals*, *Nb*, *Na*, *f*, *Fs*, *tau=0*, *justFit=False*)
Least-squares IIR filter fit to a given frequency response.

This method uses Gauss-Newton non-linear optimization and pole mapping for filter stabilization

Parameters

- **Hvals** (*numpy array of (complex) frequency response values of shape (M,)*) –
- **Nb** (*integer numerator polynomial order*) –
- **Na** (*integer denominator polynomial order*) –
- **f** (*numpy array of frequencies at which Hvals is given of shape*) –
- **(M,)** –
- **Fs** (*sampling frequency*) –
- **tau** (*integer initial estimate of time delay*) –
- **justFit** (*boolean, when true then no stabilization is carried out*) –

Returns

- **b,a** (*IIR filter coefficients as numpy arrays*)
- **tau** (*filter time delay in samples*)

References

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

PyDynamic.model_estimation.fit_filter.**LSFIR**(*H*, *N*, *tau*, *f*, *Fs*, *Wt=None*)
Least-squares fit of a digital FIR filter to a given frequency response.

Parameters

- **H** (*(complex) frequency response values of shape (M,)*) –
- **N** (*FIR filter order*) –
- **tau** (*delay of filter*) –
- **f** (*frequencies of shape (M,)*) –
- **Fs** (*sampling frequency of digital filter*) –
- **Wt** (*(optional) vector of weights of shape (M,) or shape (M, M)*) –

Returns

Return type filter coefficients bFIR (ndarray) of shape (N+1,)

PyDynamic.model_estimation.fit_filter.**invLSFIR**(*H*, *N*, *tau*, *f*, *Fs*, *Wt=None*)
Least-squares fit of a digital FIR filter to the reciprocal of a given frequency response.

Parameters

- **H** (*np.ndarray of shape (M,) and dtype complex*) – frequency response values
- **N** (*int*) – FIR filter order
- **tau** (*float*) – delay of filter
- **f** (*np.ndarray of shape (M,)*) – frequencies
- **Fs** (*float*) – sampling frequency of digital filter
- **Wt** (*np.ndarray of shape (M,) – optional*) – vector of weights

Returns **bFIR** – filter coefficients

Return type np.ndarray of shape (N,)

References

- Elster and Link [Elster2008]

PyDynamic.model_estimation.fit_filter.**invLSFIR_unc**(*H*, *UH*, *N*, *tau*, *f*, *Fs*, *wt=None*, *verbose=True*, *trunc_svd_tol=None*)

Design of FIR filter as fit to reciprocal of frequency response values with uncertainty

Least-squares fit of a digital FIR filter to the reciprocal of a frequency response for which associated uncertainties are given for its real and imaginary part. Uncertainties are propagated using a truncated svd and linear matrix propagation.

Parameters

- **H** (*np.ndarray of shape (M,)*) – frequency response values
- **UH** (*np.ndarray of shape (2M,2M)*) – uncertainties associated with the real and imaginary part
- **N** (*int*) – FIR filter order
- **tau** (*float*) – delay of filter
- **f** (*np.ndarray of shape (M,)*) – frequencies
- **Fs** (*float*) – sampling frequency of digital filter
- **wt** (*np.ndarray of shape (2M,) – optional*) – array of weights for a weighted least-squares method
- **verbose** (*bool, optional*) – whether to print statements to the command line
- **trunc_svd_tol** (*float*) – lower bound for singular values to be considered for pseudo-inverse

Returns

- **b** (*np.ndarray of shape (N+1,)*) – filter coefficients of shape
- **Ub** (*np.ndarray of shape (N+1,N+1)*) – uncertainties associated with b

References

- Elster and Link [Elster2008]

PyDynamic.model_estimation.fit_filter.**invLSIIR**(*Hvals*, *Nb*, *Na*, *f*, *Fs*, *tau*, *justFit=False*, *verbose=True*)

Design of a stable IIR filter as fit to reciprocal of frequency response values

Least-squares fit of a digital IIR filter to the reciprocal of a given set of frequency response values using the equation-error method and stabilization by pole mapping and introduction of a time delay.

Parameters

- **Hvals** (*np.ndarray of shape (M,) and dtype complex*) – frequency response values.
- **Nb** (*int*) – order of IIR numerator polynomial.
- **Na** (*int*) – order of IIR denominator polynomial.
- **f** (*np.ndarray of shape (M,)*) – frequencies corresponding to Hvals
- **Fs** (*float*) – sampling frequency for digital IIR filter.
- **tau** (*float*) – initial estimate of time delay for filter stabilization.
- **justFit** (*bool*) – if True then no stabilization is carried out.
- **verbose** (*bool*) – If True print some more detail about input parameters.

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*) – time delay (in samples)

References

- Eichstädt, Elster, Esward, Hessling [Eichst2010]

PyDynamic.model_estimation.fit_filter.**invLSIIR_unc**(*H*, *UH*, *Nb*, *Na*, *f*, *Fs*, *tau=0*)

Design of stabel IIR filter as fit to reciprocal of given frequency response with uncertainty

Least-squares fit of a digital IIR filter to the reciprocal of a given set of frequency response values with given associated uncertainty. Propagation of uncertainties is carried out using the Monte Carlo method.

Parameters

- **H** (*np.ndarray of shape (M,) and dtype complex*) – frequency response values.
- **UH** (*np.ndarray of shape (2M,2M)*) – uncertainties associated with real and imaginary part of H
- **Nb** (*int*) – order of IIR numerator polynomial.
- **Na** (*int*) – order of IIR denominator polynomial.
- **f** (*np.ndarray of shape (M,)*) – frequencies corresponding to H
- **Fs** (*float*) – sampling frequency for digital IIR filter.
- **tau** (*float*) – initial estimate of time delay for filter stabilization.

Returns

- **b,a** (*np.ndarray*) – IIR filter coefficients
- **tau** (*int*) – time delay (in samples)
- **Uba** (*np.ndarray of shape (Nb+Na+1, Nb+Na+1)*) – uncertainties associated with [a[1:],b]

References

- Eichstädt, Elster, Esward and Hessling [Eichst2010]

See also:

PyDynamic.uncertainty.propagate_filter.IIRuncFilter *PyDynamic.model_estimation.fit_filter.invLSIIR*

PyDynamic.model_estimation.fit_filter.**invLSFIR_uncMC**(*H*, *UH*, *N*, *tau*, *f*, *Fs*, *verbose=True*)

Design of FIR filter as fit to reciprocal of frequency response values with uncertainty

Least-squares fit of a FIR filter to the reciprocal of a frequency response for which associated uncertainties are given for its real and imaginary parts. Uncertainties are propagated using a Monte Carlo method. This method may help in cases where the weighting matrix or the Jacobian are ill-conditioned, resulting in false uncertainties associated with the filter coefficients.

Parameters

- **H** (*np.ndarray of shape (M,) and dtype complex*) – frequency response values
- **UH** (*np.ndarray of shape (2M,2M)*) – uncertainties associated with the real and imaginary part of H
- **N** (*int*) – FIR filter order
- **tau** (*int*) – time delay of filter in samples
- **f** (*np.ndarray of shape (M,)*) – frequencies corresponding to H
- **Fs** (*float*) – sampling frequency of digital filter
- **verbose** (*bool, optional*) – whether to print statements to the command line

Returns

- **b** (*np.ndarray of shape (N+1,)*) – filter coefficients of shape
- **Ub** (*np.ndarray of shape (N+1, N+1)*) – uncertainties associated with b

References

- Elster and Link [Elster2008]

3.2 Identification of transfer function models

The module *PyDynamic.model_estimation.fit_transfer* contains a function for the identification of transfer function models.

This module contains the following function:

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

PyDynamic.model_estimation.fit_transfer.**fit_som**(*f*, *H*, *UH=None*, *weighting=None*, *MCruns=None*, *scaling=0.001*)

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** (*np.ndarray of shape (M,)*) – vector of frequencies
- **H** (*np.ndarray of shape (2M,)*) – real and imaginary parts of measured frequency response values at frequencies f
- **UH** (*np.ndarray of shape (2M,) or (2M,2M)*) – 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.
- **weighting** (*str or array*) – Type of weighting (None, 'diag', 'cov') or array of weights (length two times of f)
- **MCruns** (*int, optional*) – Number of Monte Carlo trials for propagation of uncertainties. When MCruns is 'None', matrix multiplication is used for the propagation of uncertainties. However, in some cases this can cause trouble.
- **scaling** (*float*) – scaling of least-squares design matrix for improved fit quality

Returns

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

CHAPTER 4

Design of deconvolution filters

Deprecated since version 1.2.71: The module *deconvolution* will be combined with the module *identification* and renamed to *model_estimation* in the next major release 2.0.0. From then on you should only use the new module *Model estimation* instead. The functions `LSFIR()`, `LSFIR_unc()`, `LSIIR()`, `LSIIR_unc()`, `LSFIR_uncMC()` are then prefixed with an "inv" for "inverse", indicating the treatment of the reciprocal of frequency response values. Please use the new function names (e.g. *`PyDynamic.model_estimation.fit_filter.invLSIIR_unc()`*) starting from version 1.4.1. The old function names without preceding "inv" will only be preserved until the release prior to version 2.0.0.

The `PyDynamic.deconvolution.fit_filter` module implements methods for the design of digital deconvolution filters by least-squares fitting to the reciprocal of a given frequency response with associated uncertainties.

This module for now still contains the following functions:

- `LSFIR()`: Least-squares fit of a digital FIR filter to the reciprocal of a given frequency response.
- `LSFIR_unc()`: Design of FIR filter as fit to reciprocal of frequency response values with uncertainty
- `LSFIR_uncMC()`: Design of FIR filter as fit to reciprocal of frequency response values with uncertainty via Monte Carlo
- `LSIIR()`: Design of a stable IIR filter as fit to reciprocal of frequency response values
- `LSIIR_unc()`: Design of a stable IIR filter as fit to reciprocal of frequency response values with uncertainty

CHAPTER 5

Fitting filters and transfer functions models

Deprecated since version 1.2.71: The package *identification* will be combined with the package *deconvolution* and renamed to *model_estimation* in the next major release 2.0.0. From version 1.4.1 on you should only use the new package *Model estimation* instead.

The package for now still contains the following modules:

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

5.1 Fitting filters to frequency response

Deprecated since version 1.2.71: The package *identification* will be combined with the package *deconvolution* and renamed to *model_estimation* in the next major release 2.0.0. From version 1.4.1 on you should only use the new package *Model estimation* instead.

This module contains several functions to carry out a least-squares fit to a given complex frequency response.

This module for now still contains the following functions:

- `LSIIR()`: Least-squares IIR filter fit to a given frequency response
- `LSFIR()`: Least-squares fit of a digital FIR filter to a given frequency response

5.2 Identification of transfer function models

Deprecated since version 1.2.71: The package *identification* will be combined with the package *deconvolution* and renamed to *model_estimation* in the next major release 2.0.0. From version 1.4.1 on you should only use the new package *Model estimation* instead.

The module `PyDynamic.identification.fit_transfer` contains several functions for the identification of transfer function models.

This module for now still contains the following function:

- `fit_sos()`: Fit second-order model to complex-valued frequency response

CHAPTER 6

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

6.1 Tools for 2nd order systems

The `PyDynamic.misc.SecondOrderSystem` module is a collection of methods that are used throughout the whole package, specialized for second order dynamic systems, such as the ones used for high-class accelerometers.

This module contains the following functions:

- `sos_FreqResp()`: Calculation of the system frequency response
- `sos_phys2filter()`: Calculation of continuous filter coefficients from physical parameters
- `sos_absphase()`: Propagation of uncertainty from physical parameters to real and imaginary part of system's transfer function using GUM S2 Monte Carlo
- `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 or ndarray shape (K,)*) – static gain
- **d** (*float or ndarray shape (K,)*) – damping parameter
- **f0** (*float 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_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*) – static gain
- **d** (*float*) – damping parameter
- **f0** (*float*) – resonance frequency

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

Return type ndarray

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

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

Parameters

- **S** (*float*) – static gain
- **d** (*float*) – damping
- **f0** (*float*) – resonance frequency
- **uS** (*float*) – uncertainty associated with static gain
- **ud** (*float*) – uncertainty associated with damping
- **uf0** (*float*) – uncertainty associated with resonance frequency
- **f** (*ndarray, shape (N,)*) – frequency values at which to calculate amplitue and phase

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_realimag**(*S*, *d*, *f0*, *uS*, *ud*, *uf0*, *f*, *runs=10000*)

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

Parameters

- **S** (*float*) – static gain
- **d** (*float*) – damping
- **f0** (*float*) – resonance frequency
- **uS** (*float*) – uncertainty associated with static gain
- **ud** (*float*) – uncertainty associated with damping
- **uf0** (*float*) – uncertainty associated with resonance frequency
- **f** (*ndarray, shape (N,)*) – frequency values at which to calculate real and imaginary part

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)]]

6.2 Tools for digital filters

The *PyDynamic.misc.filterstuff* module is a collection of methods 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 a given IIR filter is stable
- *savitzky_golay()*: Smooth (and optionally differentiate) data with a Savitzky-Golay filter

PyDynamic.misc.filterstuff.**db**(*vals*)
Calculation of decibel values $20 \log_{10}(x)$ for a vector of values

PyDynamic.misc.filterstuff.**ua**(*vals*)
Shortcut for calculation of unwrapped angle of complex 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*) – sampling frequency of the filter
- **nfft** (*int*) – 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.**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.**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*) – filter order (window length)
- **fcut** (*float*) – desired cut-off frequency
- **Fs** (*float*) – sampling frequency
- **beta** (*float*) – scaling parameter for the Kaiser window

Returns

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

PyDynamic.misc.filterstuff.**isstable**(*b*, *a*, *ftype='digital'*)
Determine whether *IIR filter (b,a)* is stable

Determine whether *IIR filter (b,a)* is stable by checking roots of the polynomial ´a´.

Parameters

- **b** (*ndarray*) – filter numerator coefficients
- **a** (*ndarray*) – filter denominator coefficients
- **ftype** (*string*) – type of filter (*digital* or *analog*)

Returns **stable** – whether filter is stable or not

Return type bool

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*) – the length of the window. Must be an odd integer number
- **order** (*int*) – the order of the polynomial used in the filtering. Must be less then *window_size* - 1.
- **deriv** (*int, optional*) – The order of the derivative to compute. This must be a nonnegative integer. The default is 0, which means to filter the data without differentiating.
- **delta** (*float, 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.

References

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

6.3 Test signals

The *PyDynamic.misc.testsignals* module is a collection of test signals which can be used to simulate dynamic measurements and test methods.

This module contains the following functions:

- *shocklikeGaussian()*: signal that resembles a shock excitation as a Gaussian followed by a smaller Gaussian of opposite sign
- *GaussianPulse()*: Generates a Gaussian pulse at t_0 with height m_0 and std $sigma$
- *rect()*: Rectangular signal of given height and width $t_1 - t_0$
- *squarepulse()*: Generates a series of rect functions to represent a square pulse signal
- *sine()*: Generate a sine signal

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

Generates a signal that resembles a shock excitation as a Gaussian followed by a smaller Gaussian of opposite sign.

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants (equidistant)
- **t0** (*float*) – time instant of signal maximum
- **m0** (*float*) – signal maximum

- **sigma** (*float*) – std of main pulse
- **noise** (*float, optional*) – std of simulated signal noise

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

Return type np.ndarray of shape (N,)

PyDynamic.misc.testsignals.**GaussianPulse**(*time*, *t0*, *m0*, *sigma*, *noise=0.0*)
Generates a Gaussian pulse at t0 with height m0 and std sigma

Parameters

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

Returns **x** – signal amplitudes 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*) – time instant of rect lhs
- **t1** (*float*) – time instant of rect rhs
- **height** (*float*) – signal maximum
- **noise** (*float 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.**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*) – height of the rectangular pulses
- **numpulse** (*int*) – number of pulses
- **noise** (*float, optional*) – std of simulated signal noise

Returns **x** – signal amplitude 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.**sine**(*time*, *amp=1.0*, *freq=6.283185307179586*, *noise=0.0*)
Generate a sine signal

Parameters

- **time** (*np.ndarray of shape (N,)*) – time instants
- **amp** (*float, optional*) – amplitude of the sine (default = 1.0)
- **freq** (*float, optional*) – frequency of the sine in Hz (default = $2 * \pi$)
- **noise** (*float, optional*) – std of simulated signal noise (default = 0.0)

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

Return type np.ndarray of shape (N,)

6.4 Noise related functions

Collection of noise-signals

This module contains the following functions:

- *get_alpha()*: normal distributed signal amplitudes with equal power spectral density
- *power_law_noise()*: normal distributed signal amplitudes with power spectrum *:math:f^lpha*
- *power_law_acf()*: (theoretical) autocorrelation function of power law noise
- *ARMA()*: autoregressive moving average noise process

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

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

Parameters **color_value** (*str, int or float*) – if string -> check against known color-names -> return alpha if numeric -> directly return value

Returns **alpha**

Return type float

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

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](A Method for Colored Noise Generation)

Parameters

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

Returns **w_filt**

Return type filtered noise signal

PyDynamic.misc.noise.**power_law_acf**(*N*, *color_value='white'*, *std=1.0*)
Return 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 *f^lpha*. Sww and Rww form a Fourier-pair. Therefore Rww = ifft(Sww).

PyDynamic.misc.noise.**ARMA**(*length*, *phi=0.0*, *theta=0.0*, *std=1.0*)
Generate time-series of a predefined ARMA-process 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] .

Parameters

- **length** (*int*) – how long the drawn sample will be
- **phi** (*float, list or numpy.ndarray, shape (p,)*) – AR-coefficients
- **theta** (*float, list or numpy.ndarray*) – MA-coefficients
- **std** (*float*) – std of the gaussian white noise that is feeded into the ARMA-model

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

Return type numpy.ndarray, shape (length,)

References

- Eichstädt, Link, Harris and Elster [Eichst2012]

6.5 Miscellaneous useful helper functions

The *PyDynamic.misc.tools* module is a collection of miscellaneous helper functions.

This module contains the following functions:

- *print_vec()*: Print vector (1D array) to the console or return as formatted string
- *print_mat()*: Print matrix (2D array) to the console or return as formatted string
- *make_semiposdef()*: Make quadratic matrix positive semi-definite
- *FreqResp2RealImag()*: Calculate real and imaginary parts from frequency response
- *make_equidistant()*: Interpolate non-equidistant time series to equidistant
- *trimOrPad()*: trim or pad (with zeros) a vector to desired length
- *progress_bar()*: A simple and reusable progress-bar

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*) – the precision of the output
- **vertical** (*bool*) – print out vertical or not
- **retS** (*bool*) – print or return string

Returns **s** – if retS is True

Return type str

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*) – the precision of the output
- **vertical** (*bool*) – print out vertical or not
- **retS** (*bool*) – print or return string

Returns **s** – if retS is True

Return type str

PyDynamic.misc.tools.**make_semiposdef**(*matrix*, *maxiter=10*, *tol=1e-12*, *verbose=False*)
Make quadratic matrix positive semi-definite by increasing its eigenvalues

Parameters

- **matrix** (*(N,N) array_like*) – the matrix to process
- **maxiter** (*int*) – the maximum number of iterations for increasing the eigenvalues
- **tol** (*float*) – tolerance for deciding if pos. semi-def.
- **verbose** (*bool*) – If *True* print smallest eigenvalue of the resulting matrix

Returns quadratic positive semi-definite matrix

Return type (N,N) array_like

PyDynamic.misc.tools.**FreqResp2RealImag**(*Abs*, *Phase*, *Unc*, *MCruns=10000.0*)
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
- **MCruns** (*bool*) – Iterations for Monte Carlo simulation

Returns

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

PyDynamic.misc.tools.**make_equidistant**(*t*, *y*, *uy*, *dt=0.05*, *kind='linear'*)
Interpolate non-equidistant time series to equidistant

Interpolate measurement values and propagate uncertainties accordingly.

Parameters

- **t** (*(N,) array_like*) – timestamps (or frequencies)
- **y** (*(N,) array_like*) – corresponding measurement values
- **uy** (*(N,) array_like*) – corresponding measurement values' standard uncertainties

- **dt** (*float, optional*) – desired interval length
- **kind** (*str, optional*) – Specifies the kind of interpolation for the measurement values as a string ('previous', 'next', 'nearest' or 'linear').

Returns

- **t_new** (*(M,) array_like*) – interpolation timestamps (or frequencies)
- **y_new** (*(M,) array_like*) – interpolated measurement values
- **uy_new** (*(M,) array_like*) – interpolated measurement values' standard uncertainties

References

- White [White2017]

PyDynamic.misc.tools.**trimOrPad**(*array*, *length*, *mode='constant'*)
Trim or pad (with zeros) a vector to desired length

PyDynamic.misc.tools.**progress_bar**(*count*, *count_max*, *width=30*, *prefix=''*, *done_indicator='#'*, *todo_indicator='.'*, *fout=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)
A simple and reusable progress-bar

Parameters

- **count** (*int*) – current status of iterations, assumed to be zero-based
- **count_max** (*int*) – total number of iterations
- **width** (*int, optional*) – width of the actual progressbar (actual printed line will be wider)
- **prefix** (*str, optional*) – some text that will be printed in front of the bar (i.e. "Progress of ABC:")
- **done_indicator** (*str, optional*) – what character is used as "already-done"-indicator
- **todo_indicator** (*str, optional*) – what character is used as "not-done-yet"-indicator
- **fout** (*file-object, optional*) – where the progress-bar should be written/printed to

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

CHAPTER 8

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

Python Module Index

p

PyDynamic.misc.filterstuff, 53
PyDynamic.misc.noise, 57
PyDynamic.misc.SecondOrderSystem, 51
PyDynamic.misc.testsignals, 55
PyDynamic.misc.tools, 58
PyDynamic.model_estimation.fit_filter, 41
PyDynamic.model_estimation.fit_transfer, 45
PyDynamic.uncertainty.interpolation, 38
PyDynamic.uncertainty.propagate_DFT, 27
PyDynamic.uncertainty.propagate_filter, 32
PyDynamic.uncertainty.propagate_MonteCarlo, 33

Index

A

AmpPhase2DFT() (*in module PyDynamic.uncertainty.propagate_DFT*), 30
AmpPhase2Time() (*in module PyDynamic.uncertainty.propagate_DFT*), 31
ARMA() (*in module PyDynamic.misc.noise*), 58

C

corr_noise (*class in PyDynamic.misc.testsignals*), 56

D

db() (*in module PyDynamic.misc.filterstuff*), 53
DFT2AmpPhase() (*in module PyDynamic.uncertainty.propagate_DFT*), 31
DFT_deconv() (*in module PyDynamic.uncertainty.propagate_DFT*), 29
DFT_multiply() (*in module PyDynamic.uncertainty.propagate_DFT*), 30
DFT_transferfunction() (*in module PyDynamic.uncertainty.propagate_DFT*), 29

F

FIRuncFilter() (*in module PyDynamic.uncertainty.propagate_filter*), 32
fit_som() (*in module PyDynamic.model_estimation.fit_transfer*), 46
FreqResp2RealImag() (*in module PyDynamic.misc.tools*), 59

G

GaussianPulse() (*in module PyDynamic.misc.testsignals*), 56
get_alpha() (*in module PyDynamic.misc.noise*), 57
grpdelay() (*in module PyDynamic.misc.filterstuff*), 53
GUM_DFT() (*in module PyDynamic.uncertainty.propagate_DFT*), 28
GUM_DFTfreq() (*in module PyDynamic.uncertainty.propagate_DFT*), 29
GUM_iDFT() (*in module PyDynamic.uncertainty.propagate_DFT*), 28

I

IIRuncFilter() (*in module PyDynamic.uncertainty.propagate_filter*), 33
interp1d_unc() (*in module PyDynamic.uncertainty.interpolation*), 38
invLSFIR() (*in module PyDynamic.model_estimation.fit_filter*), 42
invLSFIR_unc() (*in module PyDynamic.model_estimation.fit_filter*), 43
invLSFIR_uncMC() (*in module PyDynamic.model_estimation.fit_filter*), 45
invLSIIR() (*in module PyDynamic.model_estimation.fit_filter*), 44
invLSIIR_unc() (*in module PyDynamic.model_estimation.fit_filter*), 44
isstable() (*in module PyDynamic.misc.filterstuff*), 54

K

kaiser_lowpass() (*in module PyDynamic.misc.filterstuff*), 54

L

LSFIR() (*in module PyDynamic.model_estimation.fit_filter*), 42
LSIIR() (*in module PyDynamic.model_estimation.fit_filter*), 42

M

make_equidistant() (*in module PyDynamic.misc.tools*), 59
make_semiposdef() (*in module PyDynamic.misc.tools*), 59
mapinside() (*in module PyDynamic.misc.filterstuff*), 54

MC() (*in module PyDynamic.uncertainty.propagate_MonteCarlo*), 34

P

power_law_acf() (*in module PyDynamic.misc.noise*), 57
power_law_noise() (*in module PyDynamic.misc.noise*), 57
print_mat() (*in module PyDynamic.misc.tools*), 58
print_vec() (*in module PyDynamic.misc.tools*), 59
progress_bar() (*in module PyDynamic.misc.tools*), 60
PyDynamic.misc.filterstuff (*module*), 53
PyDynamic.misc.noise (*module*), 57
PyDynamic.misc.SecondOrderSystem (*module*), 51
PyDynamic.misc.testsignals (*module*), 55
PyDynamic.misc.tools (*module*), 58
PyDynamic.model_estimation.fit_filter (*module*), 41
PyDynamic.model_estimation.fit_transfer (*module*), 45
PyDynamic.uncertainty.interpolation (*module*), 38
PyDynamic.uncertainty.propagate_DFT (*module*), 27
PyDynamic.uncertainty.propagate_filter (*module*), 32
PyDynamic.uncertainty.propagate_MonteCarlo (*module*), 33

R

rect() (*in module PyDynamic.misc.testsignals*), 56

S

savitzky_golay() (*in module PyDynamic.misc.filterstuff*), 54
shocklikeGaussian() (*in module PyDynamic.misc.testsignals*), 55
sine() (*in module PyDynamic.misc.testsignals*), 56
SMC() (*in module PyDynamic.uncertainty.propagate_MonteCarlo*), 34
sos_absphase() (*in module PyDynamic.misc.SecondOrderSystem*), 52
sos_FreqResp() (*in module PyDynamic.misc.SecondOrderSystem*), 51
sos_phys2filter() (*in module PyDynamic.misc.SecondOrderSystem*), 52
sos_realimag() (*in module PyDynamic.misc.SecondOrderSystem*), 52
squarepulse() (*in module PyDynamic.misc.testsignals*), 56

T

Time2AmpPhase() (*in module PyDynamic.uncertainty.propagate_DFT*), 32
Time2AmpPhase_multi() (*in module PyDynamic.uncertainty.propagate_DFT*), 32
trimOrPad() (*in module PyDynamic.misc.tools*), 60

U

ua() (*in module PyDynamic.misc.filterstuff*), 53
UMC() (*in module PyDynamic.uncertainty.propagate_MonteCarlo*), 35
UMC_generic() (*in module PyDynamic.uncertainty.propagate_MonteCarlo*), 36