GUM2DFT documentation

version 0.0.96

Sascha Eichstädt, Björn Ludwig

Februar 18, 2021

Contents

GUM2DFT

version 0.0.96 for Python 3.8

GUM2DFT is an open-source Python software to carry out uncertainty propagation for DFT, inverse DFT and for deconvolution in the frequency domain. All methods are part of the module `GUM2DFT.py`, whereas all other files present examples for the application of the module.

Installation

From a command line interface (e.g. Windows cmd) run: `python setup.py install`.

By default, this installs GUM2DFT in your Python site-packages path. However, you can specify any other path; see `python setup.py --help` for more instructions. After installation you can import methods of the module in Python by:

```
from GUM2DFT import GUM_DFT, GUM_iDFT
```

Test

To ensure all code runs as expected on your machine install the corresponding test dependencies by issuing:

`pip install install -r requirements.txt`.

Then again on the command line run: `pytest`. This should execute the 15 tests which ensure proper behaviour of the provided code and executability of the examples on your machine.

Documentation

You find the documentation in the file `GUM2DFT.pdf` . The documentation can be regenerated after installing all dependencies (see section 'Test') by issuing `sphinx-build -b pdf doc/ .` on your command line. After that you will find an automatically generated `GUM2DFT.pdf` in the root folder.

Changes

- **version 0.0.96:**
    - Fix some errors due to float division with integers
    - Introduce parameter mask into *GUM_DFT()*
    - Improve error messages in GUM2DFT.py
    - Insert type hints
    - Simplify installation by introducing dependencies into setup.py
    - Extend documentation
- **version 0.95:**
    - included method for multiplication in the frequency domain *GUM_multiply()*
    - changed license to LGPLv3 to match that of **PyDynamic**
- **version 0.93:**
    - correction in *GUM_iDFT* for the case of diagonal input covariance matrix

License

DISCLAIMER

Simulated example

This is a clean simulated deconvolution example. That is, using the measured frequency response an output signal is generated by means of convolution and thereafter the input signal estimated by deconvolution. The advantage of this approach is to verify the general applicability of the method.

```
%pylab inline
import os
from pathlib import Path

from GUM2DFT import GUM_DFT, GUM_iDFT, GUMdeconv, AmpPhase2DFT
from statsmodels.tsa.arima_process import ArmaProcess

# Make sure we load data from correct folder
file_directory = Path.cwd()
```

Measurement system

```
calib = np.loadtxt(os.path.join(file_directory, "data", "calib_PA055.dat"))
f = calib[:, 0]
Nf = 2 * len(f) - 1
FR = calib[:, 1] * np.exp(1j * calib[:, 3])
uAmp = calib[:, 2]
uPhas = calib[:, 4]
```

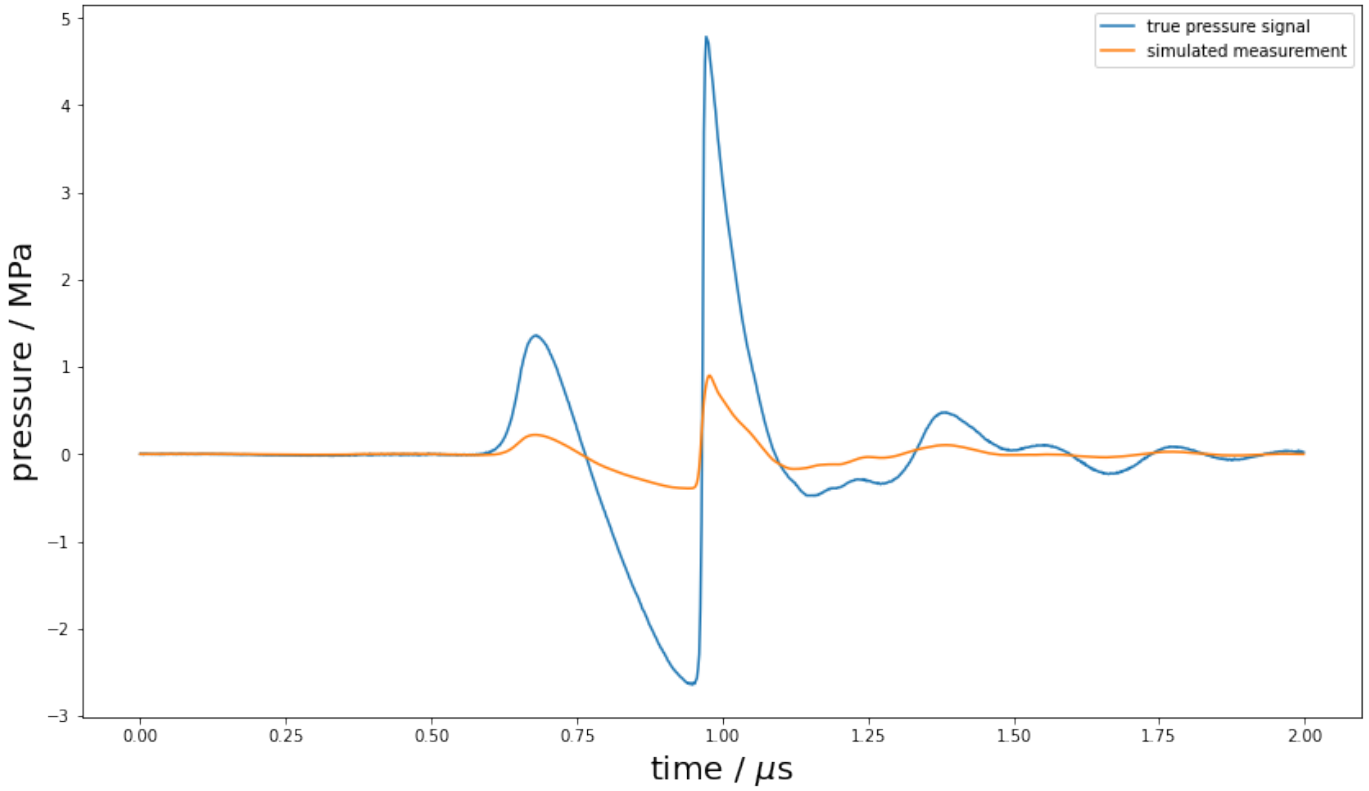
Input signal and simulated output signal

```
noise_std = 1e-4

time, pressure = np.loadtxt(
    os.path.join(file_directory, "data", "reference_signal.dat")
).T
Ts = 2e-9

P = np.fft.rfft(np.r_[pressure, np.zeros(2 * len(f) - len(pressure) - 1)])
x = np.fft.irfft(FR * P)[: len(time)]
```

```
# adding correlated noise (ARMA process)
N = len(time)
ar = np.array([0.75, -0.6])
ma = np.array([0.45, 0.1])
ARMA = ArmaProcess(ar, ma)
noise = ARMA.generate_sample(N) * 0.01

x += np.random.randn(len(time)) * noise_std
```

```
acov = ARMA.acovf(nobs=N) * noise_std ** 2

Ux = np.zeros((N, N))
for k in range(N):
    Ux[k, k:] = acov[: N - k]
Ux = Ux + Ux.T - np.diag(np.diag(Ux))
```

```
figure(figsize=(14, 8))
clf()
plot(time * 1e6, pressure, label="true pressure signal")
plot(time * 1e6, x, label="simulated measurement")
legend(loc="best")
xlabel(r"time / $\mu$s", fontsize=20)
ylabel("pressure / MPa", fontsize=20)
```

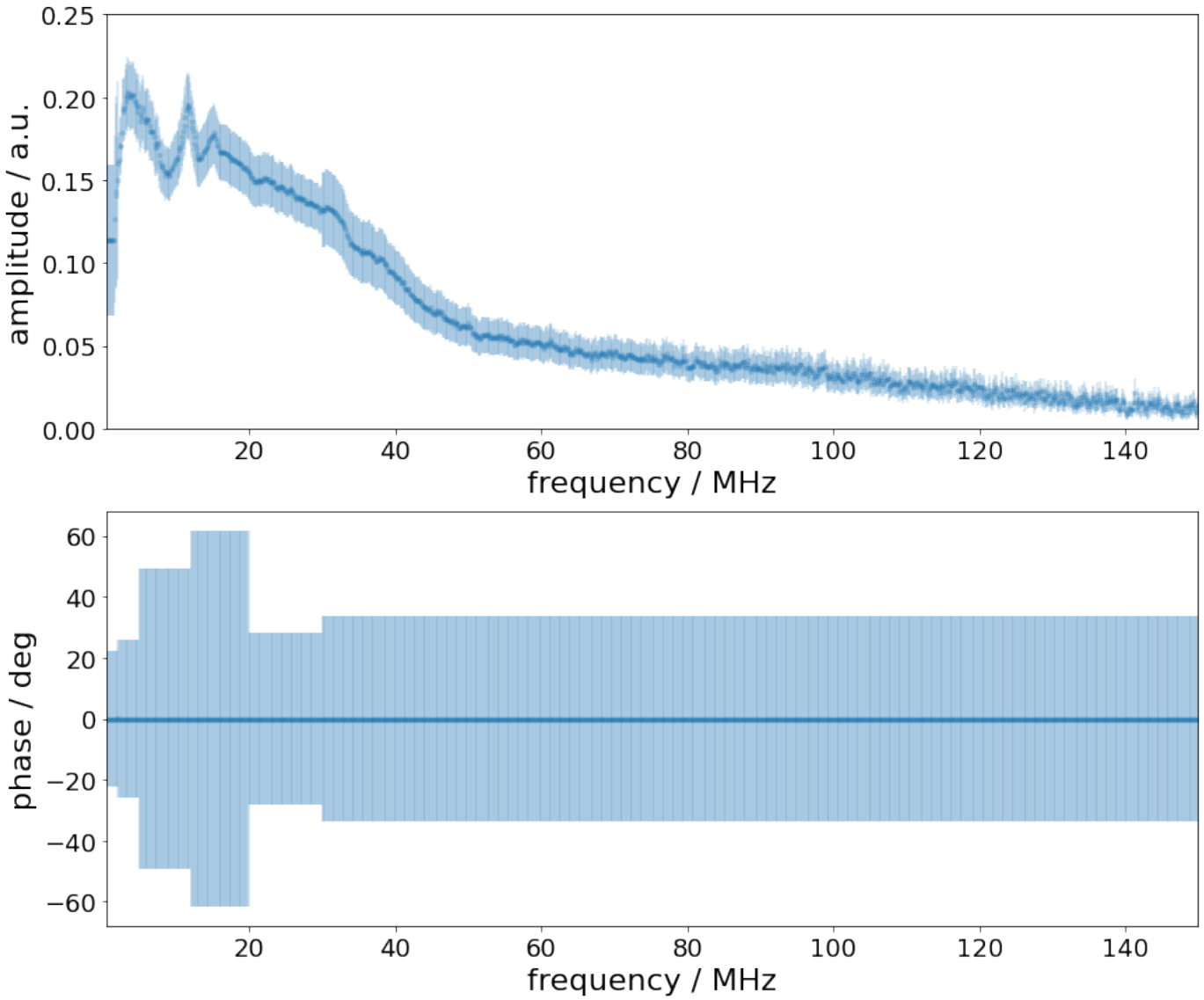


Regularized inverse system for deconvolution

```
# low-pass filter
fcut = 120e6
HL = 1 / (1 + 1j * f / fcut) ** 2
uAmp /= np.abs(HL)
# inverse system for input estimation
Hc = FR / HL
H = r_[real(Hc), imag(Hc)]
```

```
figure(figsize=(14, 12))
clf()
subplot(211)
errorbar(f * 1e-6, calib[:, 1], 2 * uAmp, fmt=".", alpha=0.2)
xlim(0.5, 150)
ylim(0, 0.25)
xlabel("frequency / MHz", fontsize=22)
```

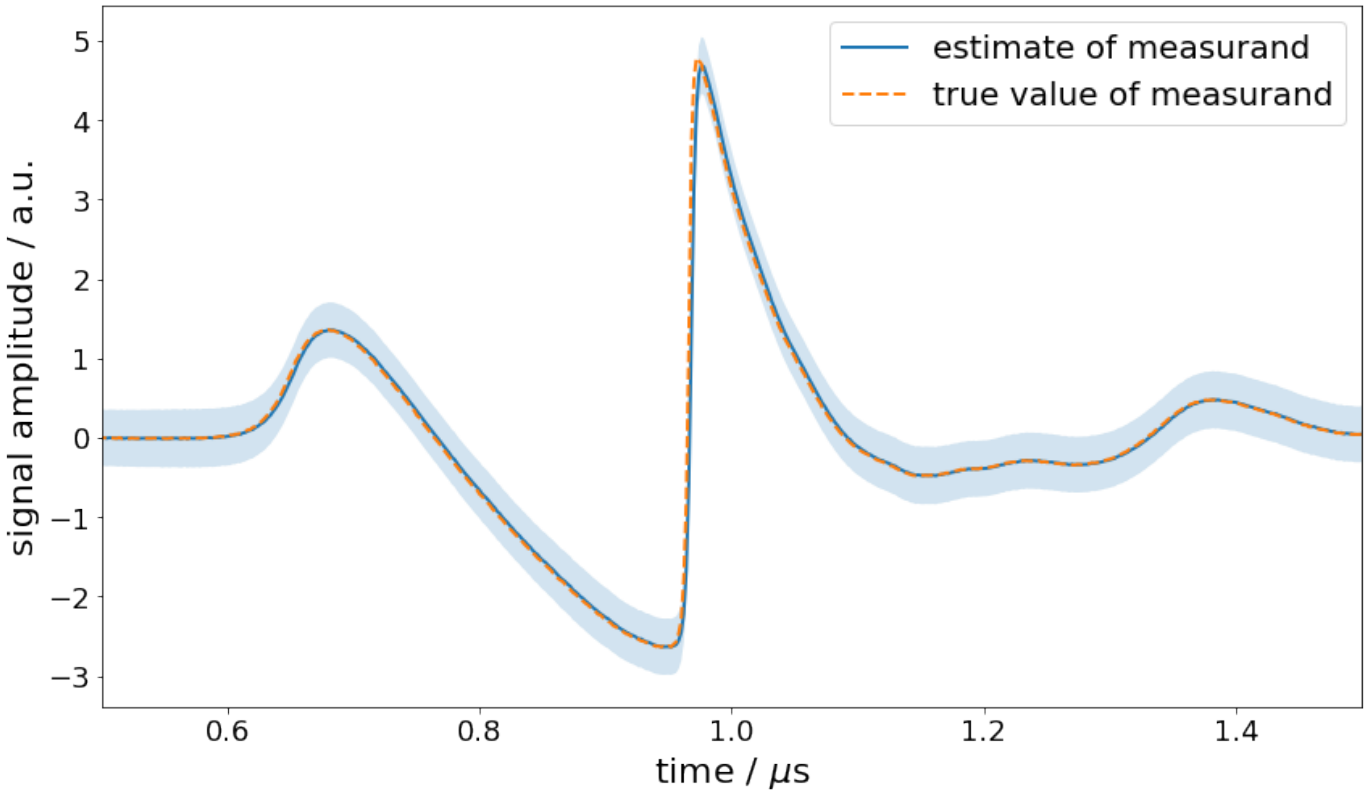
```
tick_params(which="both", labelsize=18)
ylabel("amplitude / a.u.", fontsize=22)
subplot(212)
errorbar(f * 1e-6, calib[:, 2], 2 * uPhas, fmt=".", alpha=0.2)
xlim(0.5, 150)
xlabel("frequency / MHz", fontsize=22)
tick_params(which="both", labelsize=18)
ylabel("phase / deg", fontsize=22)
```



Propagation of uncertainties

```
# propagation to real/imag domain
UH = AmpPhase2DFT(
    np.abs(FR), np.angle(FR) * np.pi / 180, np.r_[uAmp, uPhas * np.pi / 180]
)[1]

# propagation from time to frequency domain
Y, UY = GUM_DFT(x, Ux, N=Nf)

# propagation through deconvolution equation
XH, UXH = GUMdeconv(H, Y, UH, UY)

# propagation back to time domain
xh, Uxh = GUM_iDFT(XH, UXH, Nx=len(x))
```

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

figure(figsize=(14, 8))
clf()
plot(time * 1e6, xh, label="estimate of measurand", linewidth=2)
plot(time * 1e6, pressure, "--", label="true value of measurand", linewidth=2)
fill_between(time * 1e6, xh + ux, xh - ux, alpha=0.2)
xlabel(r"time / $\mu$s", fontsize=22)
ylabel("signal amplitude / a.u.", fontsize=22)
tick_params(which="major", labelsize=18)
legend(loc="best", fontsize=20, fancybox=True)
xlim(0.5, 1.5)
```



Example for the application of GUM2DFT

```
%pylab inline
```

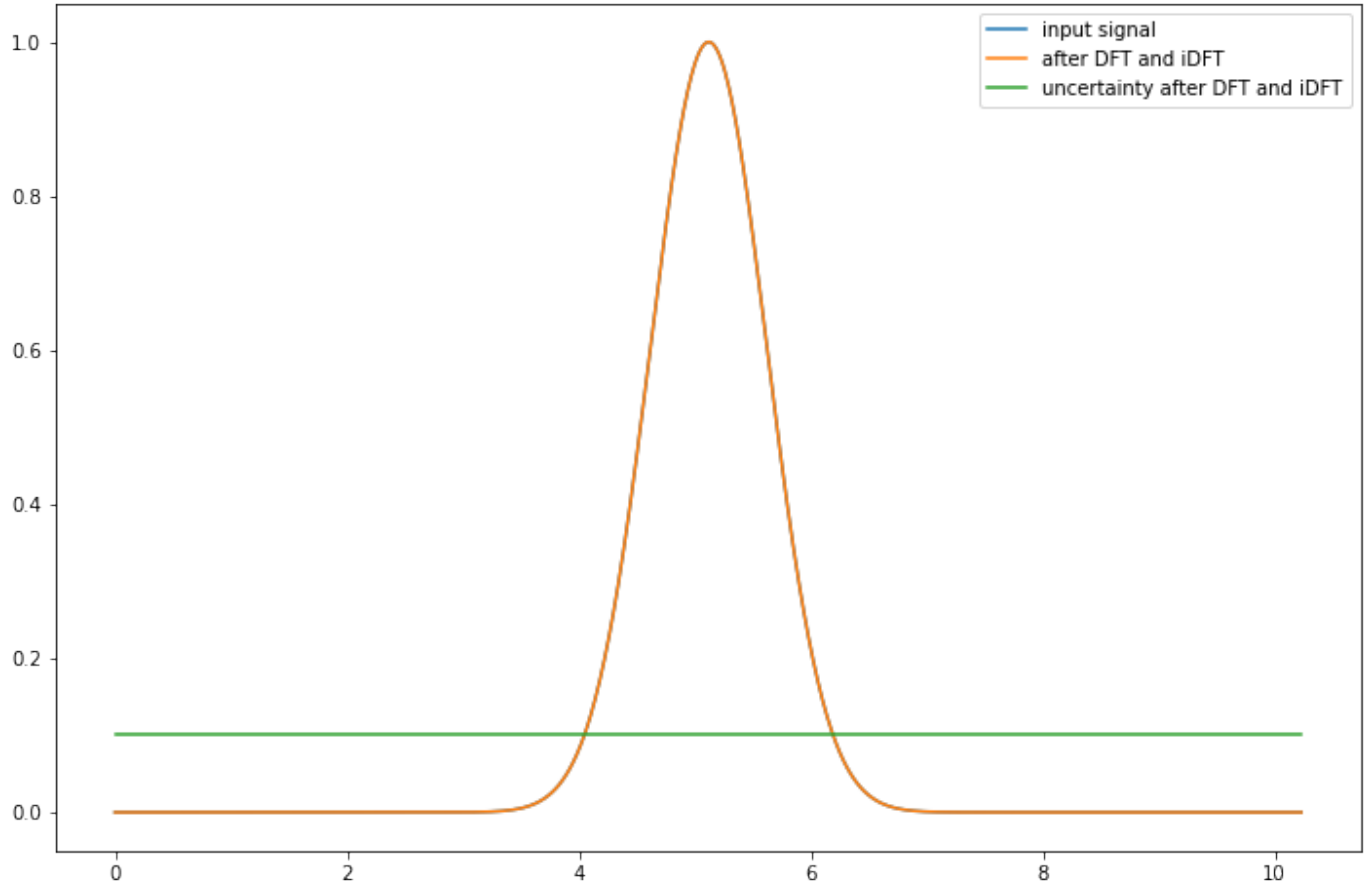
```
from numpy.fft import rfft, rfftfreq
numpy.random.seed(100)
from GUM2DFT import GUM_DFT, GUM_iDFT, AmpPhase2Time, DFT2AmpPhase, Time2AmpPhase
```

Verification of DFT and inverse DFT routines by cascading application of both.

```
N = 1024     # number of time samples
Ts= 0.01     # sample distance
noise = 0.1  # standard deviation of noise

# time domain signal
time = arange(0,N*Ts,Ts)
t0 = time[int(N/2)]
sigma = 50*Ts
x = exp(-(time-t0)**2/(2*sigma**2))
#ux= eye(N)*noise**2
ux = ones(N)*noise**2
```

```
# DFT
F, UF = GUM_DFT(x,ux)

# inverse DFT
xh, Ux = GUM_iDFT(F,UF)
```

```
figure(figsize=(12,8));clf()
plot(time,x,label="input signal")
plot(time,xh,label="after DFT and iDFT")
plot(time, sqrt(diag(Ux)), label="uncertainty after DFT and iDFT")
legend(loc="best")
```

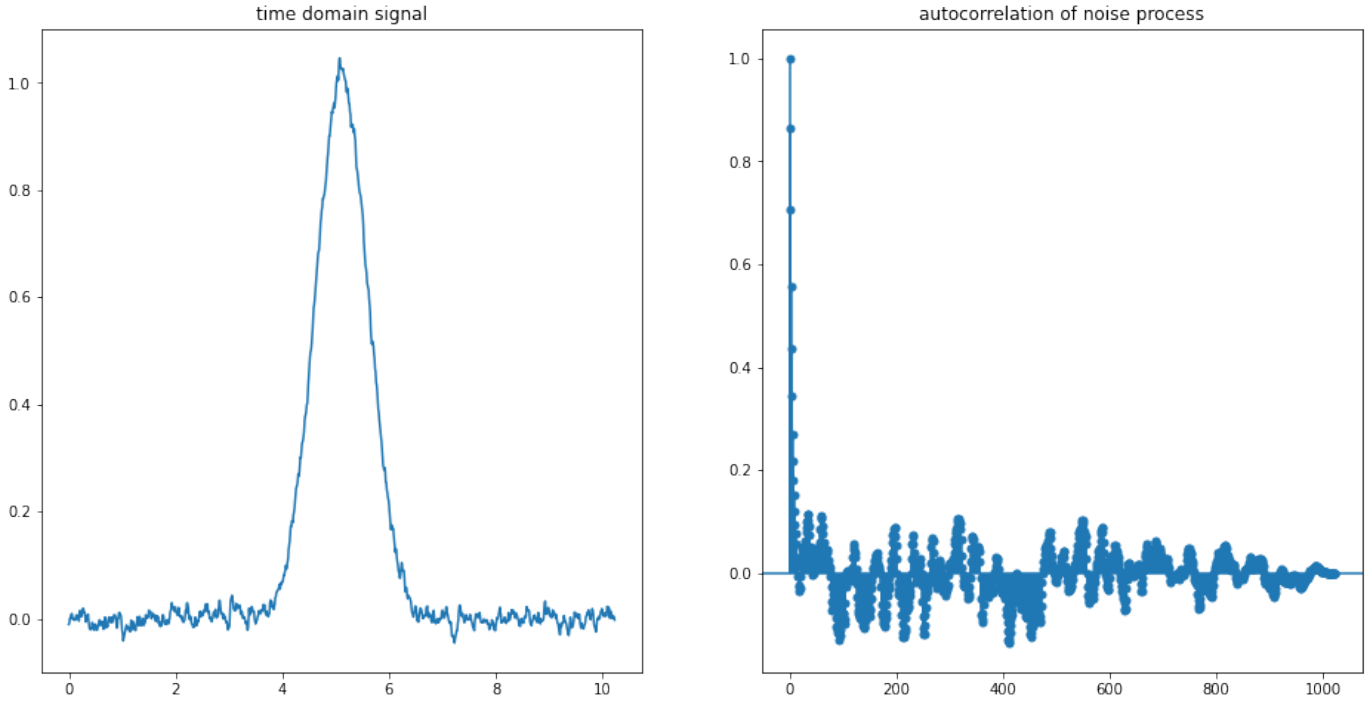


Correlated signal noise

```
import statsmodels.api as sm
from statsmodels.tsa.arima_process import ArmaProcess
```

```
N = 1024  # number of time samples
Ts = 0.01  # sample distance

ar = array([0.75, -0.6])
ma = array([0.45, 0.1])
noise = ArmaProcess(ar, ma).generate_sample(N) * 0.01

# time domain signal
time = arange(0, N * Ts, Ts)
t0 = time[int(N / 2)]
sigma = 50 * Ts
x = exp(-((time - t0) ** 2) / (2 * sigma ** 2))
ux = diag(noise ** 2)
```

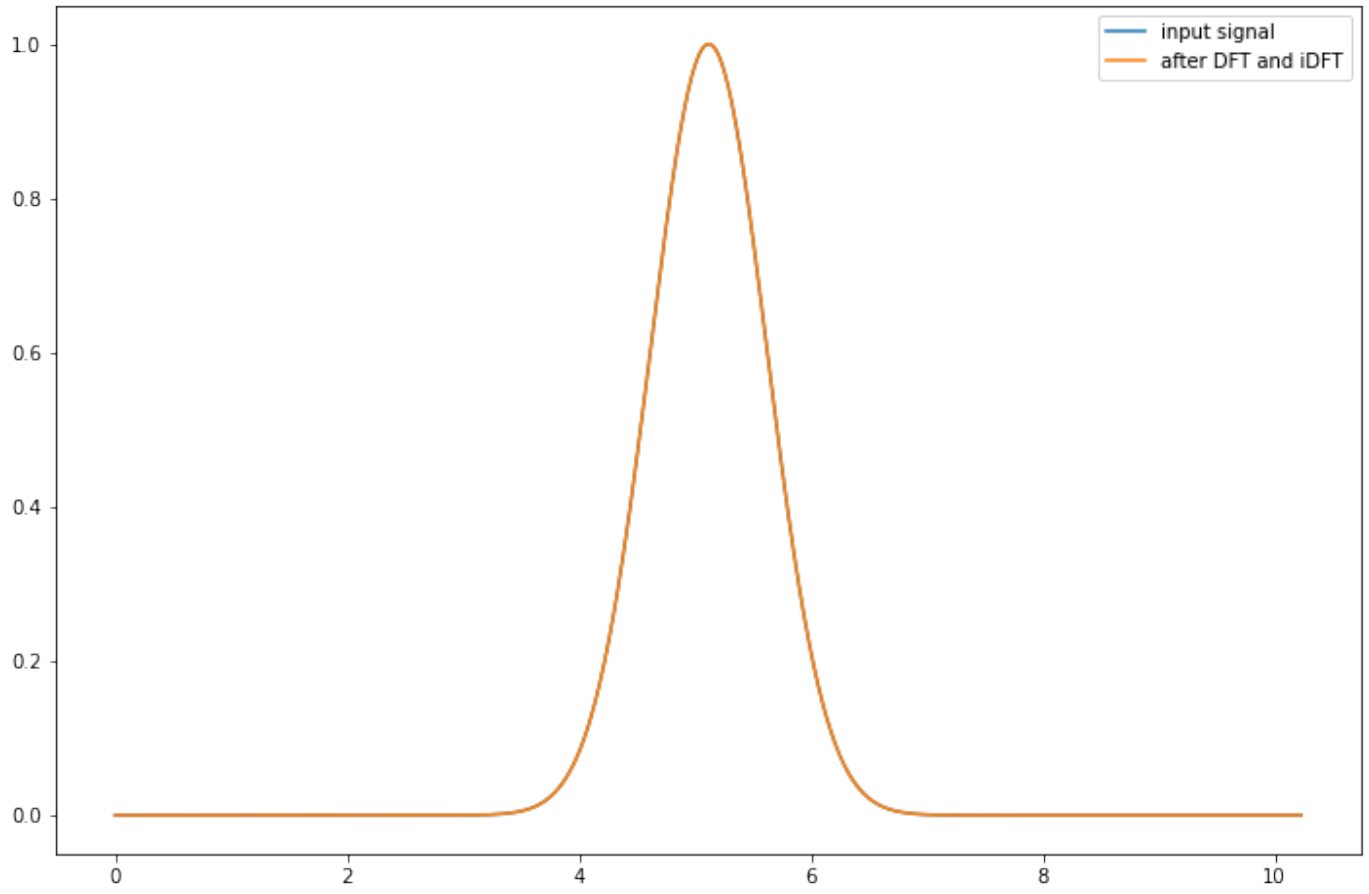
```
figure(figsize=(16, 8))
clf()
subplot(121)
plot(time, noise + x)
title("time domain signal")
ax = subplot(122)
sm.graphics.tsa.plot_acf(noise, ax=ax, alpha=1.0, lags=len(noise) - 1)
title("autocorrelation of noise process")
```

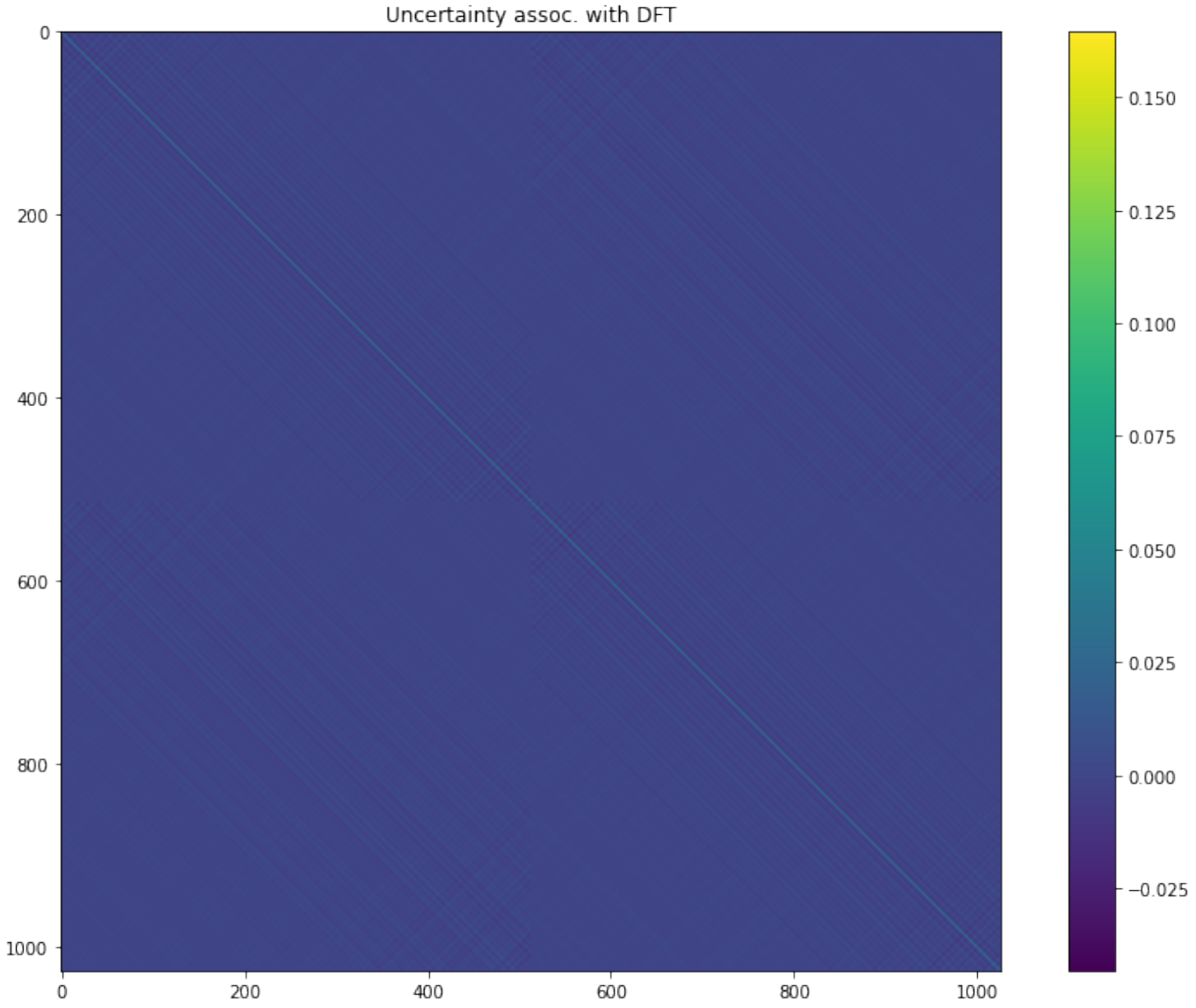



```
# DFT
F, UF = GUM_DFT(x, ux)

# inverse DFT
xh, Ux = GUM_iDFT(F, UF)
```

```
figure(figsize=(12, 8))
clf()
plot(time, x, label="input signal")
```

```
plot(time, xh, label="after DFT and iDFT")
legend(loc="best")

figure(figsize=(14, 10))
clf()
imshow(UF)
colorbar()
title(u"Uncertainty assoc. with DFT")
```




Uncertainty assoc. with DFT
0
200
400
600
800
1000
0
200
400
600
800
1000
0.150
0.125
0.100
0.075
0.050
0.025
0.000
−0.025