Python Kontrol Library

Release 1.0.0-beta.1

TSANG Terrence Tak Lun

Mar 05, 2024

CONTENTS:

Python package for KAGRA suspension and seismic isolation control.

CHAPTER

ONE

KONTROL

The name "Kontrol" is a blend word combining KAGRA, the gravitational-wave detector in Japan, and control, as in control system. The package contains necessary features for commissioning a KAGRA suspension. These features include:

- **Sensor and actuation utilities** (`kontrol.sensact`): Calibration and Sensing/Actuation Matrices
- **System Modeling** (`kontrol.curvefit`): Fitting frequency response data using Transfer function model.
- **Basic suspension controller design** (`kontrol.regulator`): Damping and position controller with stabilizing post filters such as low-pass and notch filters.

To interface the results generated from the above functionalities to the KAGRA control system, Kontrol also provides:

- **Foton utilities** (`kontrol.foton`): converting transfer function to/from Foton strings.
- **Ezca wrapper** (`kontrol.ezca`): Fetch/put control matrices to/from the digital system.

The above features form a control design pipeline from calibration to controller design for commissioning KAGRA suspension with basic functionality.

Besides the basic functionalities, Kontrol also contains advanced features are being continuously developed in order to further enhance seismic isolation performance. Currently, Kontrol contains:

- **H-infinity optimal complementary filters** (`kontrol.ComplementaryFilter`): Solves complementary control problems, optimizing control filters for sensor fusion, sensor correction, and vibration isolation control problems[1][2].
- **Dynamic mode decomposition** (`kontrol.dmd`): Dynamic mode decomposition for time-series forecasting and modeling. For future model predictive control work.

While Kontrol was initially created for optimizing KAGRA control systems, the package is also suitable for other gravitational-wave detectors, including LIGO, Virgo, and future detectors, such as the Einstein Telescope due to their similarities.

To familiarize users with the package, step-by-step tutorials are provided. Upon finishing the tutorials, the users should be able to convert the scripts into usable ones interfacing real data that can be used for the physical systems.

- **Documentation:** https://kontrol.readthedocs.io/
- **Repository:** https://github.com/terrencetec/kontrol.git

[1] T. T. L. Tsang, T. G. F. Li, T. Dehaeze, C. Collette. Optimal Sensor Fusion Method for Active Vibration Isolation Systems in Ground-Based Gravitational-Wave Detectors. https://arxiv.org/pdf/2111.14355.pdf

[2] Terrence Tak Lun Tsang. Optimizing Active Vibration Isolation Systems in Ground-Based Interferometric Gravitational-Wave Detectors. https://gwdoc.icrr.u-tokyo.ac.jp/cgi-bin/DocDB/ShowDocument?docid=14296

1.1 Getting Started

1.1.1 Dependencies

Required

- control>=0.9
- numpy
- matplotlib
- scipy

Optional

- ezca (Needed for accessing EPICs records/real-time model process variables. Use conda to install it.)
- vishack or dttxml (For extracting data from diaggui xml files.)

If you would like to install Kontrol on your local machine with, then pip should install the required dependencies automatically for you. However, if you use Kontrol in a Conda environment, you should install the dependencies before installing Kontrol to avoid using pip. In Conda environment, simply type

```
conda install -c conda-forge numpy scipy matplotlib control ezca
```

Using **Conda** is strongly recommended because `control` depends on `slycot` which can be cumbersome to install without conda. Check this issue out if you wish to install `slycot` on a Linux machine.

1.1.2 Install from PyPI

```
pip install kontrol
```

1.1.3 Install from source

For local usage, type

```
$ git clone https://github.com/terrencetec/kontrol.git
$ cd kontrol
$ pip install .
```

For k1ctr workstations, make sure a virtual environment is enabled before installing any packages.

1.1.4 Concept and Tutorials

Check out *How to use Kontrol* and *Tutorials* to see how this package can be utilized for setting up and optimizing seismic isolation control systems.

1.2 How to use Kontrol

`Kontrol` was designed to help setting up a KAGRA vibration isolation system. But, because many control systems in gravitational-wave detectors are set up in a similar way, `Kontrol` can help setting up other control systems as well. You should have `Kontrol` installed. If you haven't, refer to *Getting Started*.

`Kontrol` provides the necessary modules and functionality for setting a control system.

1. Sensors and actuators calibration and alignment (`kontrol.sensact`).
    1. `kontrol.sensact.calibrate()` for calibrating linear sensors.
    2. `kontrol.curvefit.TransferFunctionFit` for calibrated inertial sensors (sensors with frequency responses.)
    3. `kontrol.sensact.SensingMatrix` for refining sensing matrices for aligning signals.
2. System modeling (`kontrol.curvefit`).
    1. `kontrol.curvefit.TransferFunctionFit` for modeling system processes.
    2. `kontrol.curvefit.spectrum_fit()` for modeling frequency spectrums with the magnitude response of a transfer function.
3. Controller design (`kontrol.regulator`).
    1. `kontrol.regulator.oscillator.pid()` for designing position and damping PID controllers for oscillatory systems (systems with complex poles) with coefficients determined by critical criteria.
    2. `kontrol.regulator.post_filter` for designing post lower-pass and notch filters with stability constrains.

`Kontrol` also provides advanced features for optimizing seismic isolation systems.

1. H-infinity optimization for solving complementary filter problems (`kontrol.ComplementaryFilter`).
    1. Optimizes complementary filters for sensor fusion.
    2. Optimizes sensor correction filters for sensor correction.
    3. Optimizes feedback controller with known disturbance and noise.

All aforementioned funtionalities are detailed in Chapter 6 and 8 in Ref.[1]. These methods solve the static optimal control problem for a seismic isolation system. And, you can find example usages of them in the *Tutorials*.

References

[1] Terrence Tak Lun Tsang. Optimizing Active Vibration Isolation Systems in Ground-Based Interferometric Gravitational-Wave Detectors. https://gwdoc.icrr.u-tokyo.ac.jp/cgi-bin/DocDB/ShowDocument?docid=14296

1.3 Tutorials

This tutorial is mainly divided into two parts, *Basic Suspension Commissioning* and *Advanced Control Methods*. There's an extra section called General Utilites and there you will find miscellaneous tools that are generally useful.

In the first part, the commissioning of the active control system for a stage of a virtual suspension with 3 degrees of freedom is demonstrated. This covers topics including: sensor calibration, sensing matrices, system modeling, and controller design. By the end of the first part, users should be able to commission suspensions to achieve basic damping and position control with appropriate filtering.

The second part of the tutorial covers specialized frontier control topics for seismic isolation that are not considered standard. This section is not compulsory for suspension commissioning. The content in this section is everchanging and being updated as new methods are being developed and coded into Kontrol.

Each subsection solves a small problem and contributes to a small step towards solving the bigger problem, e.g. suspension commissioning and control optimization. You will find a general description giving context for the small problem in each subsection. And you will find a link to the Jupyter notebook containing the solution for the problem.

Each Jupyter notebook is structured similarly as follows. All notebooks begins with preparing and visualizing the mock measurements that we will be processing. In reality, the data will be measured instead of generated. With the measurement data obtained, we will proceed to demonstrate the use of the `Kontrol` package to solve the underlying problems. At last, we will export the results in some way for further usages, e.g. for further processes or implementation. This way, the notebooks can be thought as a process in a data pipeline.

The style of this tutorial is inspired from the actual commissioning of a KAGRA suspension. As in, these are all the necessary steps that we need to go through when setting up the active isolation systems. Therefore, the notebooks can be easily modified for actual usage simply by replacing the data.

Content

2. *Sensor Correction Filter Synthesis*
3. *Sensor fusion and optimal controller (just words)*

3. *General Utilities*

Kontrol has more than what's covered in the tutorials. Be sure to check out the *Kontrol API* section for detailed documentation.

1.3.1 Basic Suspension Commissioning

We were asking to set up the active control system for a stage, particularly, the inverted pendulum stage, of the suspension in KAGRA. Ultimately, the goal is to obtain a controller that can be used for active damping and positon control. We'll go through the necessary steps to achieve so using the `Kontrol` package. Information about the suspension will be given along the way as necessary.

This section closely follows Chapter 6 in Ref.[1]. Check the reference out if you wish to understand what the functions/methods are actually doing in the background.

Sensors and actuators

The first step is to set up the sensors and actuators for the active isolation system. Without sensors and actuators, there's no control. The goal here is to set up the sensors and actuators such that we can use them to obtain a measurement of the frequency response of the system that we want to control. To achieve this, we need to

1. Calibrate the sensors so they meausre physical units.
2. Align the sensors (and actuators, using control matrices) so we get a readout in the control basis.

Linear sensor calibration

We were told to calibrate a relative displacement sensor for the suspension. We took a caliper and measured the actual displacement while recording the sensor output. We scanned the displacement from -10 to 10 mm with 1 mm interval and data below is what we got.

```
displacement = [-10., -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10]␣
↪ # millimeters
```

and

```
output = [-32765., -32760, -32741, -32680, -32504, -32060, -31068, -29109, -25691, -
↪20421, -13241, -4596, 4598, 13243, 20423, 25693, 29111, 31070, 32062, 32506, 32682]
```

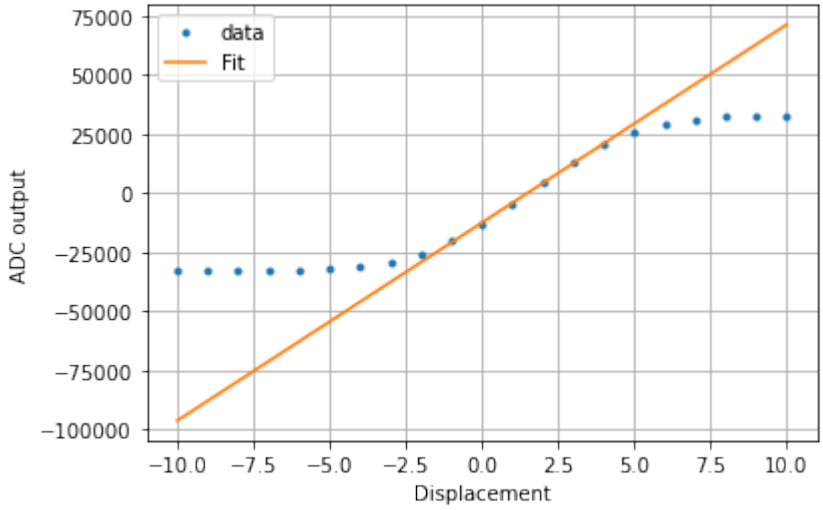
The goal is to obtain a value that converts the output to displacement. Click the link below to see how we can use the `kontrol.sensact.calibrate()` function to obtain the calibration factor.

[1] Terrence Tak Lun Tsang. Optimizing Active Vibration Isolation Systems in Ground-Based Interferometric Gravitational-Wave Detectors. https://gwdoc.icrr.u-tokyo.ac.jp/cgi-bin/DocDB/ShowDocument?docid=14296

Calibration of a Linear Sensor

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol


# # Commented below is how the data was generated
# x = np.linspace(-10, 10, 21)
# x0 = 1.5
# y0 = 1
# a = 32768
# m = 0.25
# y = a*scipy.special.erf(m*(x-x0)) + y0
# #

# There's the data we obtained by measurement.
displacement = [-10., -9, -8, -7, -6, -5, -4, -3,
                -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  # millimeters
output = [-32765., -32760, -32741, -32680, -32504, -32060,
          -31068, -29109, -25691, -20421, -13241, -4596, 4598,
          13243, 20423, 25693, 29111, 31070, 32062, 32506, 32682]

# This method firstly fits the 3 data cloest to the middle of the full range
# with a straight line
# and keeps adding data to the set within a certain non-linearity (5% by default)
# compared to the fitted line.
# The new data set is then fitted and the process terminates when
# no more data points can be added.
slope, intercept = kontrol.sensact.calibrate(displacement, output)

# This commented method fits the data to with an error function erf().
# Particularly Useful for optical displacement sensors.
# slope, intercept = kontrol.sensact.calibrate(displacement, output, method="erf")

fit = slope*np.array(displacement) + intercept

plt.plot(displacement, output, ".", label="data")
plt.plot(displacement, fit, label="Fit")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("ADC output")
plt.xlabel("Displacement")
```

[1]:
```
Text(0.5, 0, 'Displacement')
```



[2]:
```
# The slope has a unit of of ADC count per Displacement unit.
# The inverse of the slope converts the ADC output to displacement.
# Hence, the calibration factor is
print(f"Calibration factor: {1/slope*1000} microns per [ADC count]")
```

```
Calibration factor: 0.11950857794565542 microns per [ADC count]
```

The calibration factor turned out to be 0.1195 microns per ADC count.

Inertial sensor calibration

We were told to calibrate an inertial sensor: the geophone. The inertial sensor is different from a relative sensor because it has a frequency response. The transfer function (from velocity to output) of a geophone is given by

$$H(s) = G\frac{s^2}{s^2 + \frac{\omega_n}{Q}s + \omega_n^2},$$

where G, ω_n, Q, are the calibration values we need to obtain.

We happened to have a spare calibrated seismometer so we can measure the ground velocity. We placed the geophone next to the seismometer and took a measurement simultaneously. The ratio between the geophone output and the seismometer output happens to be the frequency response of $H(s)$, which we can use to obtain the calibration values.

The goal is to fit the frequency response with the transfer function model $H(s)$ and click the link below to see how we can use `kontrol.curvefit.TransferfunctionFit` class to obtain the fit.

Calibration of an Inertial Sensor

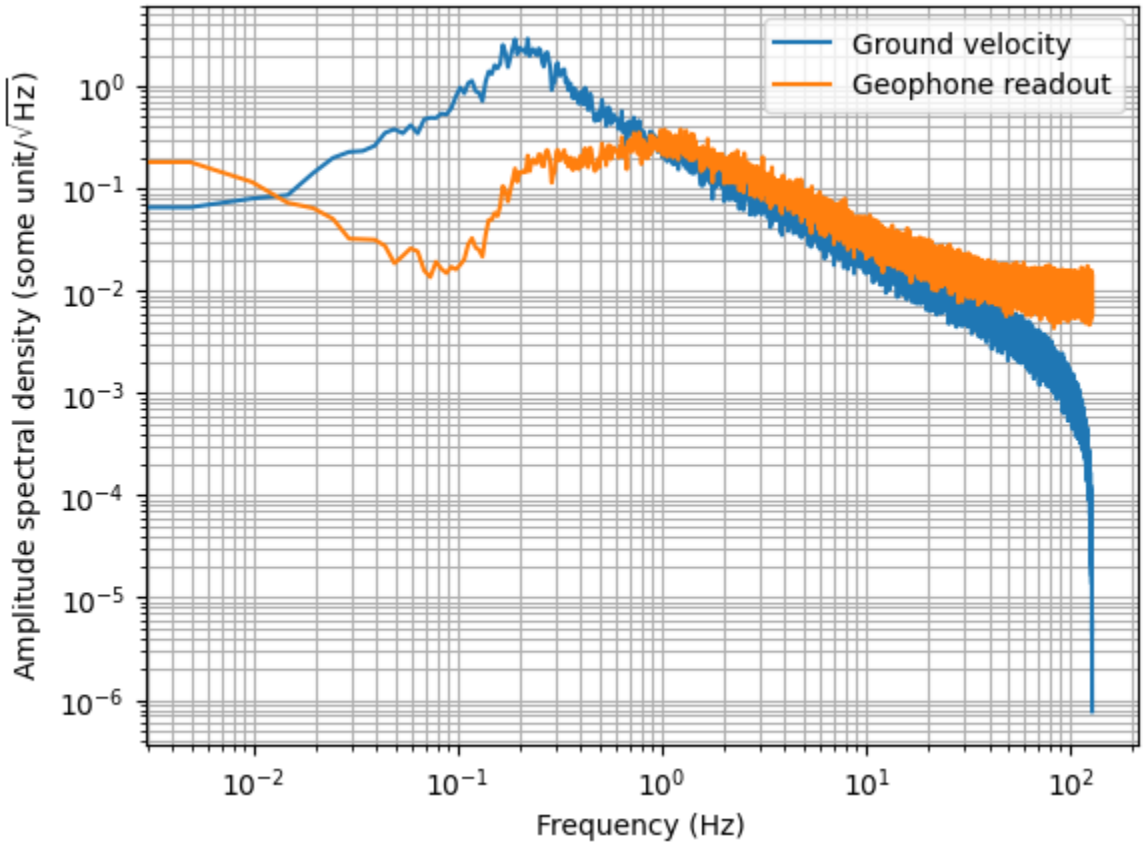
[1]:
```
import control
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol.curvefit
```

(continues on next page)

(continued from previous page)

```
np.random.seed(123)  # Fix the random seed so it's reproducible.
fs = 256  # Hz
t = np.linspace(0, 1024, 1024*fs)
s = control.tf("s")

# Simulate measurement data (replace this part with real measurements)
# Obtain some ground motion.
# Shape of the ground velocity spectrum
ground_tf = s*(0.2*2*np.pi)**2 / (s**2 + (0.2*2*np.pi)/(2)*s + (0.2*2*np.pi)**2)
random = np.random.normal(loc=0, scale=np.sqrt(fs/2), size=len(t))
_, ground_velocity = control.forced_response(ground_tf, U=random, T=t)
# ground_velocity is what the seismometer measure.
# For simplicity, let's just assume the seismometer is noiseless so it reads exactly␣
↪ground_velocity.

# The geophone is not noiseless so it reads slightly different fro the seismometer.
# Generate the analog-to-digital converter noise.
adc_noise_tf = 10**(-2)/s*(s+0.1*2*np.pi)
random2 = np.random.normal(loc=0, scale=np.sqrt(fs/2), size=len(t))
_, adc_noise = control.forced_response(adc_noise_tf, U=random2, T=t)

# Define the geophone transfer function (We don't know these values yet!)
G = 1.5
wn = 1*2*np.pi
Q = 1/np.sqrt(2)
geophone_tf = G * s**2 / (s**2 + wn/Q*s + wn**2)


# Generate the geophone readout
_, geophone_readout = control.forced_response(geophone_tf, U=ground_velocity, T=t)
geophone_readout += adc_noise  # Add ADC noise


# Here's what we measured
f, ground_psd = scipy.signal.welch(ground_velocity, fs=1/(t[1]-t[0]), nperseg=int(len(t)/
↪5))
_, geophone_psd = scipy.signal.welch(geophone_readout, fs=1/(t[1]-t[0]),␣
↪nperseg=int(len(t)/5))

plt.loglog(f, ground_psd**0.5, label="Ground velocity")
plt.loglog(f, geophone_psd**0.5, label="Geophone readout")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel(r"Amplitude spectral density (some unit/$\sqrt{\mathrm{Hz}}$)")
plt.xlabel("Frequency (Hz)")
```

[1]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



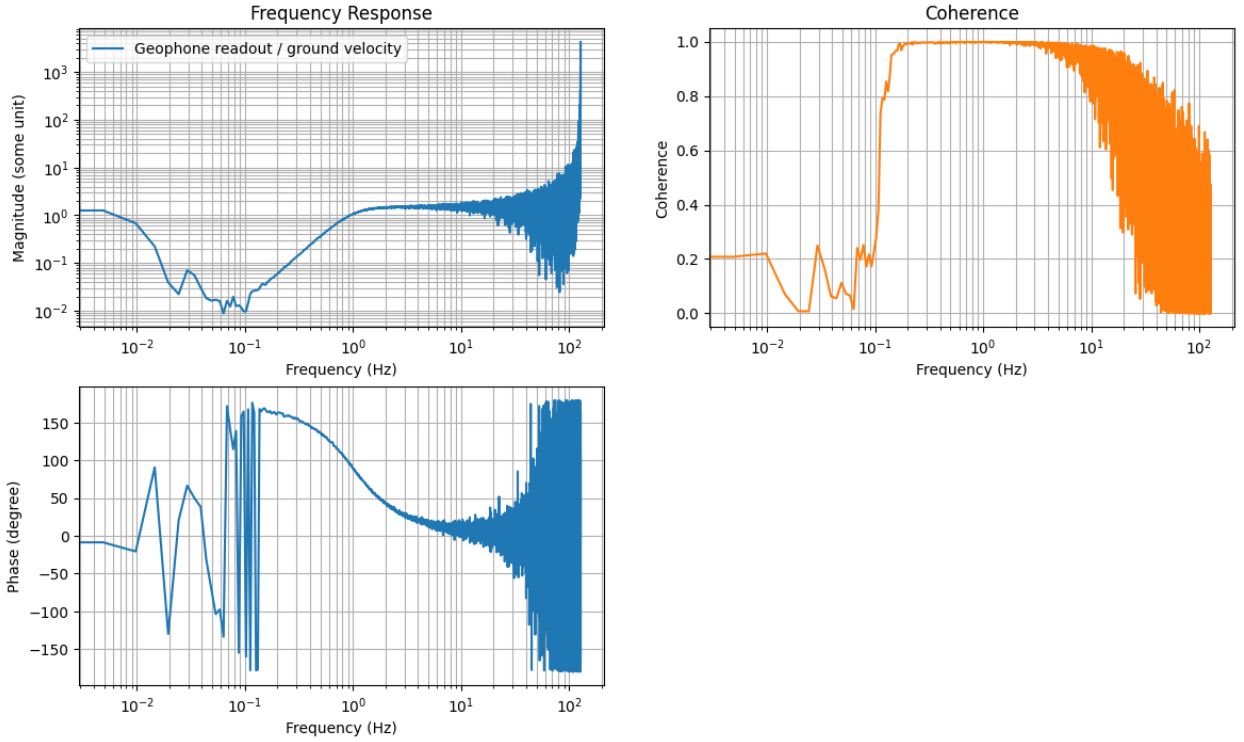
[2]:
```
# We can obtain the cross-spectral density between the two signals
# and obtain a frequency response of the geophone and the coherence between the two.
f, ground_geophone_csd = scipy.signal.csd(ground_velocity, geophone_readout, fs=1/(t[1]-
↪t[0]), nperseg=int(len(t)/5))
frequency_response = ground_geophone_csd/ground_psd
coherence = abs(ground_geophone_csd)**2 / (ground_psd*geophone_psd)
# ^The data can be obtained from the digital system directly.
# If you're working with real data, you should be able to skip everything above.

plt.figure(figsize=(14, 8))
plt.subplot(221)
plt.title("Frequency Response")
plt.loglog(f, abs(frequency_response), label="Geophone readout / ground velocity")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel(r"Magnitude (some unit)")
plt.xlabel("Frequency (Hz)")

plt.subplot(223)
plt.semilogx(f, 180/np.pi*np.angle(frequency_response), label="Frequency response")
plt.grid(which="both")
plt.ylabel(r"Phase (degree)")
plt.xlabel("Frequency (Hz)")
```

(continues on next page)

(continued from previous page)

```
plt.subplot(222)
plt.title("Coherence")
plt.semilogx(f, coherence, color="C1", label="Coherence")
plt.grid(which="both")
plt.ylabel("Coherence")
plt.xlabel("Frequency (Hz)")
```

[2]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[3]:
```
# As can be seen, the two signals are not perfectly coherence at all frequencies.
# We should pick out the part where they have good coherence,
# i.e. where the geophone is measuring the ground motion, but not noise.
# The coherence function is a good tool for this.
# Let's pick out the frequency response data where coherence is larger than 0.99
frequency_response_pick = frequency_response[coherence>0.99]
f_pick = f[coherence>0.99]
# This is the data that we want to fit.
```

[4]:
```
# Fitting with kontrol.curvefit.TransferFunctionFit.
# To use it, we need to specify a few attributes.
# We need to select the x_data, y_data, model, optimizer, and optimizer_kwargs.
curvefit = kontrol.curvefit.TransferFunctionFit()
curvefit.xdata = f_pick
curvefit.ydata = frequency_response_pick

# Kontrol.curvefit.model has a library of generic models for fitting.
# However, let's just define our own geophone model for the sake of understanding what's␣
↪happening.
```

(continues on next page)

(continued from previous page)

```
# Model has a signature func(x, args, **kwargs)->array
# x is the independent variable and args are the parameters that define the model.
def model(x, args):
    G, wn, Q = args
    s = control.tf("s")
    geophone_tf = G * s**2 / (s**2+wn/Q*s+wn**2)
    return geophone_tf(1j*2*np.pi*x)

curvefit.model = model

# Normally, we don't need to specify the optimizer.
# By default it's using scipy.optimizer.minimize, which is a local optimizer.
# It requires initial guess of the G, wn, and Q to start optimizing.
# It's easy enough to obtain the inital guess from specifications.
# But let's just assume we don't know any of that but we know a range for those␣
↪parameters.
# In this case, we can use a global optimization algorithm.
curvefit.optimizer = scipy.optimize.differential_evolution

# We know G is the high-frequency gain, from the plot, it falls between 0.1 and 10.
# We know wn is the corner frequency and it falls between 0.1 and 10 Hz or 0.1*2*np.pi␣
↪and 10*2*np.pi rad/s
# We know Q value is the height at the corner frequency and it falls between 0.5 and 10.
bounds = [(0.1, 10), (0.1*2*np.pi, 10*2*np.pi), (0.5, 10)]  # Try widening the bounds␣
↪except for 0.5 for the Q.
curvefit.optimizer_kwargs = {"bounds": bounds}
result = curvefit.fit()  # This returns a scipy.optimize.OptimizeResult object.
```

[5]:
```
# To obtain the model parameters, use result.x
G_fit, wn_fit, Q_fit = result.x

# Let's reconstruct the fitted geophone transfer function
geophone_tf_fit = G_fit * s**2 / (s**2+wn_fit/Q_fit*s+wn_fit**2)

# Compare the true transfer function and the fit
print("True geophone transfer function:", geophone_tf)
print("Fitted geophone transfer function:", geophone_tf_fit)
```

```
True geophone transfer function:
       1.5 s^2
---------------------
s^2 + 8.886 s + 39.48

Fitted geophone transfer function:
      1.495 s^2
---------------------
s^2 + 8.862 s + 39.41
```

[6]:
```
# Close enough, let's try to obtain a calibration filter so we can measure velocity with␣
↪the geophone.
# Recall that the geophone transfer function converts ground velocity
```

(continues on next page)

(continued from previous page)

```
# to geophone output, the inverse of it converts geophone output to ground velocity.
# The inverse is what we need to "calibrate" the geophone.
# And it can be used as a calibration filter in the digital system to get velocity from␣
↪geophone.
calibration_filter = 1 / geophone_tf_fit

# To install it to the KAGRA digital system, we need to convert the transfer function
# from analytic form to a Foton string.
# Kontrol.TransferFunction object offers this functionality.
calibration_filter = kontrol.TransferFunction(calibration_filter)
# ^Converting itself into a Kontrol.TransferFunction object

foton_string = calibration_filter.foton(root_location="n")
# ^the option is unnesscary but "n" is the one we typically use.
print("This is what we need to put into the digital system:\n", foton_string)
# This is the filter that will convert the geophone readout into velocity.
```

```
This is what we need to put into the digital system:
 zpk([0.705233+i*0.707833;0.705233+i*-0.707833],[-0;-0],0.667771,"n")
```

To install the calibration filter to the KAGRA digital system, we have obtained a Foton string of the calibration filter using the `kontrol.TransferFunction` class:

```
zpk([0.705852+i*0.707336;0.705852+i*-0.707336],[-0;-0],0.667203,"n")
```

With this filter, we can start measuring velocity with the geophone. If we want displacement instead, we can simply add an integrator.

Sensing matrices

With all the sensors calibrated, we can now read the motion of the suspension. However, the sensors are usually not placed in a way they aligned perfectly with the basis that we are interested in. As in, we get 3 readouts, $\vec{y} = (y_1, y_2, y_3)$, but we want to control in some basis $\vec{x} = (x_1, x_2, x_3)$, which is typically the Cartesian/Euler angle basis.

With some geometry and linear algebra, we were able to obtain a (geometric) sensing matrix

$$\mathbf{A} = \begin{pmatrix} -0.333 & -0.333 & 0.666 \\ 0.577 & -0.577 & 0. \\ 0.333 & 0.333 & 0.333 \end{pmatrix},$$

such that $\vec{x} \approx \mathbf{A}\vec{y}$. With high hopes we installed this matrix into the digital system, hoping to measure the crisp distinguishable 0.06, 0.1, 0.2 Hz resonances in the $\vec{x} = (x_1, x_2, x_3)$ degrees of freedom measurement. However, you inspect the readouts and discovered cross-coupling between the three degrees of freedom, i.e. sensing matrix is not perfect.

The goal is to refine the sensing matrix to reduce the observable cross-couplings so the sensors are aligned with the control basis. Click the link below to see how we can use `kontrol.sensact.SensingMatrix` to obtain a new sensing matrix that aligns the sensors.

Diagonalizing a Sensing Matrix

[1]:
```
import control
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol


# Setup and virtually obtain the measurements
# Ignore the first and second block if you're not interested in how the
# data was generated.

# Original sensing matrix.
# Usually obtained from first priniciples, i.e. geometry
# This sensing matrix in particular is a sensing matrix for the LVDT sensors
# at the inverted pendulum stage.
sensing_matrix_inv = [
    [-np.sin(30*np.pi/180), np.cos(30*np.pi/180), 1],
    [-np.sin(150*np.pi/180), np.cos(150*np.pi/180), 1],
    [-np.sin(270*np.pi/180), np.cos(270*np.pi/180), 1]
]
sensing_matrix = np.linalg.inv(np.array(sensing_matrix_inv))
#
# Define the dymanics of the system
# Let's say x1 resonant at 0.06 Hz, x2 resonant at 0.1 Hz, and x3 0.2 Hz.
# To make things more complicated, let's assume there're two x1-x2 coupled modes, at 0.5␣
↪and 1 Hz.
s = control.tf("s")
fs = 32
t = np.linspace(0, 1024, 1024*fs)
# 5 modes
w1 = 0.06*2*np.pi
w2 = 0.1*2*np.pi
w3 = 0.2*2*np.pi
w4 = 0.5*2*np.pi
w5 = 1*2*np.pi
q1 = 10
q2 = 10
q3 = 10
q4 = 20
q5 = 20
tf1 = w1**2 / (s**2+w1/q1*s+w1**2)
tf2 = w2**2 / (s**2+w2/q2*s+w2**2)
tf3 = w3**2 / (s**2+w3/q3*s+w3**2)
tf4 = 0.01*w4**2 / (s**2+w4/q4*s+w4**2)
tf5 = 0.01*w5**2 / (s**2+w5/q5*s+w5**2)

# We give the system a "kick" so it excites every mode.
# It's equivalent to doing a white noise injection
impulse_response1 = control.impulse_response(tf1, T=t)
```

(continues on next page)

(continued from previous page)

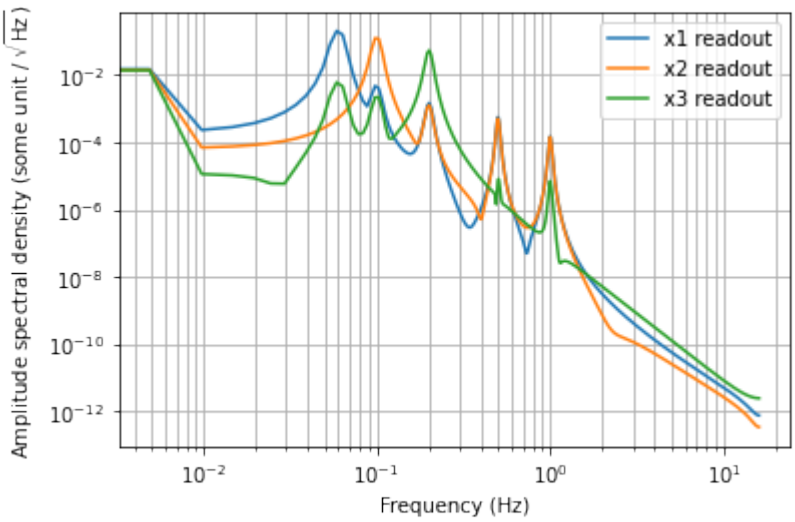
```
impulse_response2 = control.impulse_response(tf2, T=t)
impulse_response3 = control.impulse_response(tf3, T=t)
impulse_response4 = control.impulse_response(tf4, T=t)
impulse_response5 = control.impulse_response(tf5, T=t)

mode1 = impulse_response1.outputs
mode2 = impulse_response2.outputs
mode3 = impulse_response3.outputs
mode4 = impulse_response4.outputs
mode5 = impulse_response5.outputs

# This is the true motion of the table.
x1 = mode1+0.707*mode4+0.707*mode5
x2 = mode2+0.707*mode4-0.707*mode5
x3 = mode3
x = np.array([x1, x2, x3])


# Let's suppose we got the geometry wrong by a bit.
sensing_matrix_inv_real = [
    [-np.sin(33*np.pi/180), np.cos(33*np.pi/180), 1.01],
    [-np.sin(147*np.pi/180), np.cos(147*np.pi/180), 0.97],
    [-np.sin(273*np.pi/180), np.cos(273*np.pi/180), 0.95]
]
sensing_matrix_inv_real = np.array(sensing_matrix_inv_real)

# The raw sensor readouts.
y = sensing_matrix_inv_real @ x
y1 = y[0, :]
y2 = y[1, :]
y3 = y[2, :]

# We installed the sensing matrix and attempted to align the readouts
# into the control basis
x_read = sensing_matrix @ y

# We obtained 3 readouts
x1_read = x_read[0, :]
x2_read = x_read[1, :]
x3_read = x_read[2, :]

# Usually in frequency domain
f, x1_psd = scipy.signal.welch(x1_read, fs=fs, nperseg=int(len(t)/5))
f, x2_psd = scipy.signal.welch(x2_read, fs=fs, nperseg=int(len(t)/5))
f, x3_psd = scipy.signal.welch(x3_read, fs=fs, nperseg=int(len(t)/5))

plt.loglog(f, x1_psd**0.5, label="x1 readout")
plt.loglog(f, x2_psd**0.5, label="x2 readout")
plt.loglog(f, x3_psd**0.5, label="x3 readout")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit / $\sqrt{\mathrm{Hz}}$)")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Frequency (Hz)")
```

```
[1]: Text(0.5, 0, 'Frequency (Hz)')
```



Here, it's worth spending time to explain what we are seeing from our readouts.

First of all, we know there's a resonance in x1 at 0.06 Hz. However, we're observing more than that in the spectrum. From the blue curve, the 0.1 Hz and the 0.2 Hz are also observable. Since we know that the system only resonates at 0.1 Hz and 0.2 Hz in the x2 and x3 direction, respectively, the extra peaks we see from the x1 readout must be cross-couplings from the other degrees of freedom. To get rid of them, we need to measure these cross-couplings, in magnitude and phase (0 or 180 degrees), and construct a coupling matrix as shown below.

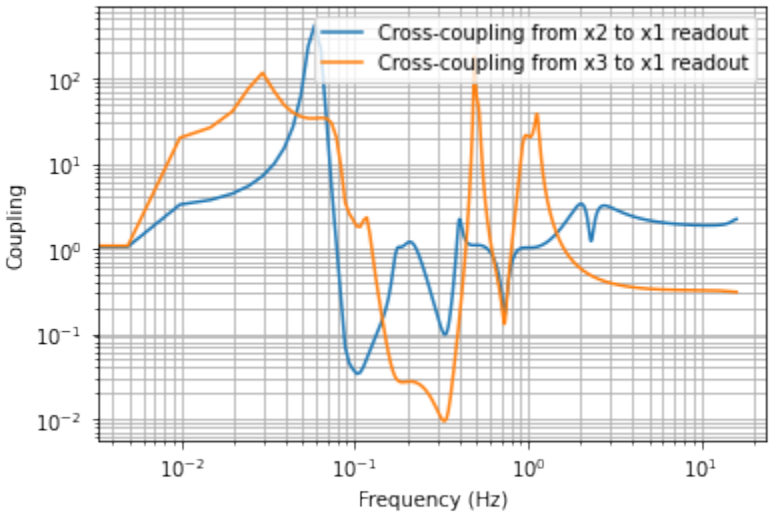
```
[2]: # Obtain the cross-couplings, or "transfer function", from x2 to x1, x3 to x1, ...
f, csd_21 = scipy.signal.csd(x2_read, x1_read, fs=fs, nperseg=int(len(t)/5))
f, csd_31 = scipy.signal.csd(x3_read, x1_read, fs=fs, nperseg=int(len(t)/5))
f, csd_12 = scipy.signal.csd(x1_read, x2_read, fs=fs, nperseg=int(len(t)/5))
f, csd_32 = scipy.signal.csd(x3_read, x2_read, fs=fs, nperseg=int(len(t)/5))
f, csd_13 = scipy.signal.csd(x1_read, x3_read, fs=fs, nperseg=int(len(t)/5))
f, csd_23 = scipy.signal.csd(x2_read, x3_read, fs=fs, nperseg=int(len(t)/5))

tf_21 = csd_21/x2_psd
tf_31 = csd_31/x3_psd
tf_12 = csd_12/x1_psd
tf_32 = csd_32/x3_psd
tf_13 = csd_13/x1_psd
tf_23 = csd_23/x2_psd

# Plotting two as an example.
plt.figure(121)
plt.loglog(f, abs(tf_21), label="Cross-coupling from x2 to x1 readout")
plt.loglog(f, abs(tf_31), label="Cross-coupling from x3 to x1 readout")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Coupling")
plt.xlabel("Frequency (Hz)")
```

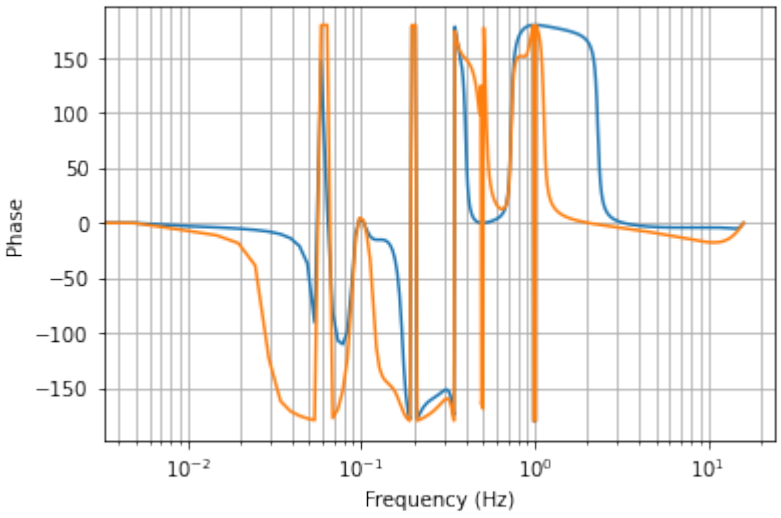
(continues on next page)

(continued from previous page)

```
plt.figure(122)
plt.semilogx(f, 180/np.pi*np.angle(tf_21), label="Cross-coupling from x2 to x1 readout")
plt.semilogx(f, 180/np.pi*np.angle(tf_31), label="Cross-coupling from x3 to x1 readout")
# plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Phase")
plt.xlabel("Frequency (Hz)")
```

[2]:
```
Text(0.5, 0, 'Frequency (Hz)')
```





Now, this cross-coupling plot is not useful at all frequencies. It's only useful where the cross-couplings are observed. That means, for the x1 readout, we're looking at 0.1 Hz for the x2 to x1 coupling and 0.2 Hz for the x3 to x1 coupling.

[3]:
```
coupling21 = tf_21[np.argmin(abs(f-0.1))]
coupling31 = tf_31[np.argmin(abs(f-0.2))]
coupling12 = tf_12[np.argmin(abs(f-0.06))]
coupling32 = tf_32[np.argmin(abs(f-0.2))]
coupling13 = tf_13[np.argmin(abs(f-0.06))]
```

(continues on next page)

(continued from previous page)

```
coupling23 = tf_23[np.argmin(abs(f-0.1))]

print("Cross-couplings")
print(f"From x2 to x1 readout: mag:{abs(coupling21):.3g}, phase:{180/np.pi*np.
↪angle(coupling21):.3g} degrees")
print(f"From x3 to x1 readout: mag:{abs(coupling31):.3g}, phase:{180/np.pi*np.
↪angle(coupling31):.3g} degrees")
print(f"From x1 to x2 readout: mag:{abs(coupling12):.3g}, phase:{180/np.pi*np.
↪angle(coupling12):.3g} degrees")
print(f"From x3 to x2 readout: mag:{abs(coupling32):.3g}, phase:{180/np.pi*np.
↪angle(coupling32):.3g} degrees")
print(f"From x1 to x3 readout: mag:{abs(coupling13):.3g}, phase:{180/np.pi*np.
↪angle(coupling13):.3g} degrees")
print(f"From x2 to x3 readout: mag:{abs(coupling23):.3g}, phase:{180/np.pi*np.
↪angle(coupling23):.3g} degrees")
```

```
Cross-couplings
From x2 to x1 readout: mag:0.0381, phase:3.79 degrees
From x3 to x1 readout: mag:0.0273, phase:180 degrees
From x1 to x2 readout: mag:0.00233, phase:-147 degrees
From x3 to x2 readout: mag:0.0235, phase:1.38 degrees
From x1 to x3 readout: mag:0.0295, phase:-180 degrees
From x2 to x3 readout: mag:0.0175, phase:-0.923 degrees
```

Here, it's important to only include cross-couplings that has a phase close to 0 to 180 degrees. If it's not 0 or 180 degrees, it's not cross-coupling resulting from sensor misalignment. In this case, x1 to x2 coupling has a phase of -147 degrees. From the spectrum, the x1 resonance at 0.06 Hz is not visible from the x2 readout.

[4]:
```
# Construct the coupling matrix.
# Diagonals should be ones, assuming that the sensors were mostly aligned.
coupling_matrix = [
    [1, abs(coupling21), -abs(coupling31)],  # Use negative sign for phase ~180 degrees
    [0, 1, abs(coupling32)],  # Recall coupling12 is not real.
    [-abs(coupling13), abs(coupling23), 1]
]
coupling_matrix = np.array(coupling_matrix)

# Using Kontrol
sensing_matrix = kontrol.sensact.SensingMatrix(matrix=sensing_matrix)
sensing_matrix.coupling_matrix = coupling_matrix
diagonalized_matrix = sensing_matrix.diagonalize()

print(f"New sensing matrix:\n {diagonalized_matrix}")
```

```
New sensing matrix:
 [[-0.34648554 -0.30189757  0.67664218]
 [ 0.56999299 -0.58521291 -0.00830399]
 [ 0.31314695  0.33465657  0.35344143]]
```

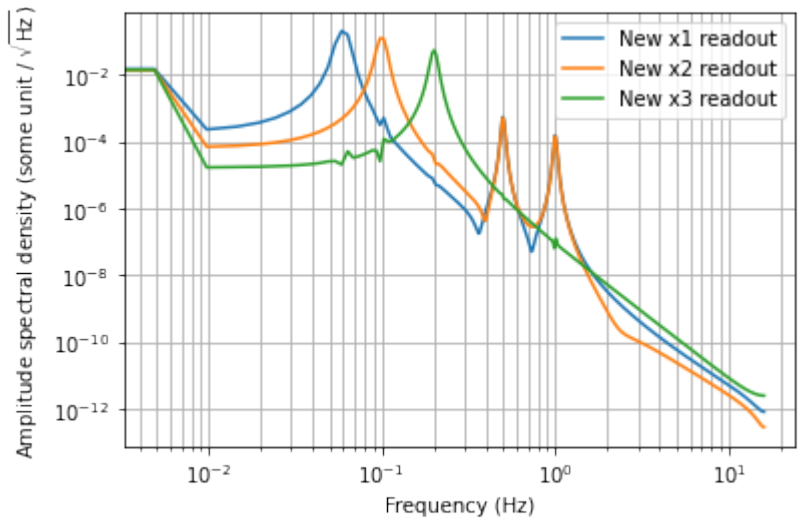
[5]:
```
# Install the new matrix and obtain new readouts.
x_new = diagonalized_matrix @ y

x1_new = x_new[0, :]
```

(continues on next page)

(continued from previous page)

```
x2_new = x_new[1, :]
x3_new = x_new[2, :]

f, x1_new_psd = scipy.signal.welch(x1_new, fs=fs, nperseg=int(len(t)/5))
f, x2_new_psd = scipy.signal.welch(x2_new, fs=fs, nperseg=int(len(t)/5))
f, x3_new_psd = scipy.signal.welch(x3_new, fs=fs, nperseg=int(len(t)/5))


plt.loglog(f, x1_new_psd**0.5, label="New x1 readout")
plt.loglog(f, x2_new_psd**0.5, label="New x2 readout")
plt.loglog(f, x3_new_psd**0.5, label="New x3 readout")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit / $\sqrt{\mathrm{Hz}}$)")
plt.xlabel("Frequency (Hz)")
# ^As can be seen, the cross-couplings have been drastically reduced.
```

[5]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[6]:
```
# Additional tips:
# To obtain/install the sensing matrix, use kontrol.ezca
# Commented below only works in a control workstation

# import kontrol.ezca
# ezca = kontrol.ezca.Ezca("K1:VIS-BS")
# sensing_matrix = ezca.get_matrix("IP_LVDT2EUL")  # Get matrix
# ezca.put_matrix(diagonalized_matrix, "IP_LVDT2EUL")  # Install matrix
```

[7]:
```
# Additional tips 2:
# Sometimes there's an additional matrix after the first sensing matrix for the␣
↪diaognalization purpose.
# To obtain that matrix instead, use the identity matrix when declaring the matrix␣
↪option:
```

(continues on next page)

(continued from previous page)

```
sensing_matrix1 = kontrol.sensact.SensingMatrix(matrix=np.eye(3), coupling_
↪matrix=coupling_matrix)
additional_matrix = sensing_matrix1.diagonalize()
additional_matrix
```

[7]:
```
SensingMatrix([[ 1.00083373e+00, -3.86143137e-02,  2.82590738e-02],
               [-6.94024575e-04,  1.00043766e+00, -2.35239074e-02],
               [ 2.95396759e-02, -1.86278778e-02,  1.00124495e+00]])
```

[8]:
```
# The equivalent matrix is the same after the matrix multiplication.
equivalent_matrix = additional_matrix @ sensing_matrix
# ^Note that the sensor readouts passes through the sensing matrix before the additional␣
↪matrix.
equivalent_matrix
```

[8]:
```
SensingMatrix([[-0.34648554, -0.30189757,  0.67664218],
               [ 0.56999299, -0.58521291, -0.00830399],
               [ 0.31314695,  0.33465657,  0.35344143]])
```

By identifying the cross-couplings between sensor channels, we were able to obtain a new sensing matrix,

$$\mathbf{A}_{\text{new}} = \begin{pmatrix} -0.346 & -0.301 & 0.676 \\ 0.569 & -0.585 & -0.00830 \\ 0.313 & 0.334 & 0.353 \end{pmatrix},$$

that aligns the sensors to the control basis.

To obtain or install the matrix to the digital system, we can define a `kontrol.ezca.Ezca` instance and use the `get_matrix()` or `put_matrix()` methods.

Caveats

- Sensor cross-coupling is only true when the phase is close to 0 or 180 degrees.

Actuation matrices

Like sensing matrices, the initial actuation matrices are obtained from first principles using geometry. However, the diagonalization of an actuation matrix is not as simple. Ideally, we want the actuation in one degree of freedom to move the system in that degree of freedom only. However, most of the time, this won't happen with a diagonalization of a scalar actuation matrix. This is because the actuation cross-coupling is frequency dependent, meaning that it would require a transfer matrix (a matrix of transfer functions) to fully decouple all degrees of freedom.

The proper way to approach this is to measure all non-diagonal frequency responses from actuation to output, put them into matrix form, invert it, and fit them using transfer functions. This can be extremely tedious and fortunately unneccessary. If all degrees of freedom are under the action of feedback control, then the actuation cross-coupling will be suppressed. If a diagonalization is required, refer to the transfer function modeling section below.

System modeling

Now with the sensors and actuators set up, we can start characterizing the system that we'd like to control. The goal is to obtain a model of the system so we can eventually design a controller for it.

Transfer function modeling

Using the actuation in the x_1 direction, we excite the system across all frequencies (using a white noise or sweep sine signal). The suspension responded by moving in that direction at all frequencies. We measured the magnitude and phase relative to the actuation signal. This gives us the frequency response of the system, which is simply the transfer function evaluated along the imaginary axis.

Modeling frequency with a transfer function can be challenging due to numerical instability and large dynamic range. Luckily, with a few assumptions, we can obtain a fit easily. Experienced user can even obtain a reasonable fit without the use of optimization. This can be used as an initial guess for a local optimization.

The goal is to obtain a transfer function, which has frequency response that matches the data that we obtained. We can use `kontrol.curvefit.TransferFunctionFit` class, like what we did in *Inertial sensor calibration*. Here, instead of defining the model, we can use the predefined `kontrol.curvefit.ComplexZPK` class as the model. We can fit the frequency response 2 ways, with or without an initial guess. Click the links below to see how the frequency response can be fitted.

Transfer Function Modeling (without initial guess)

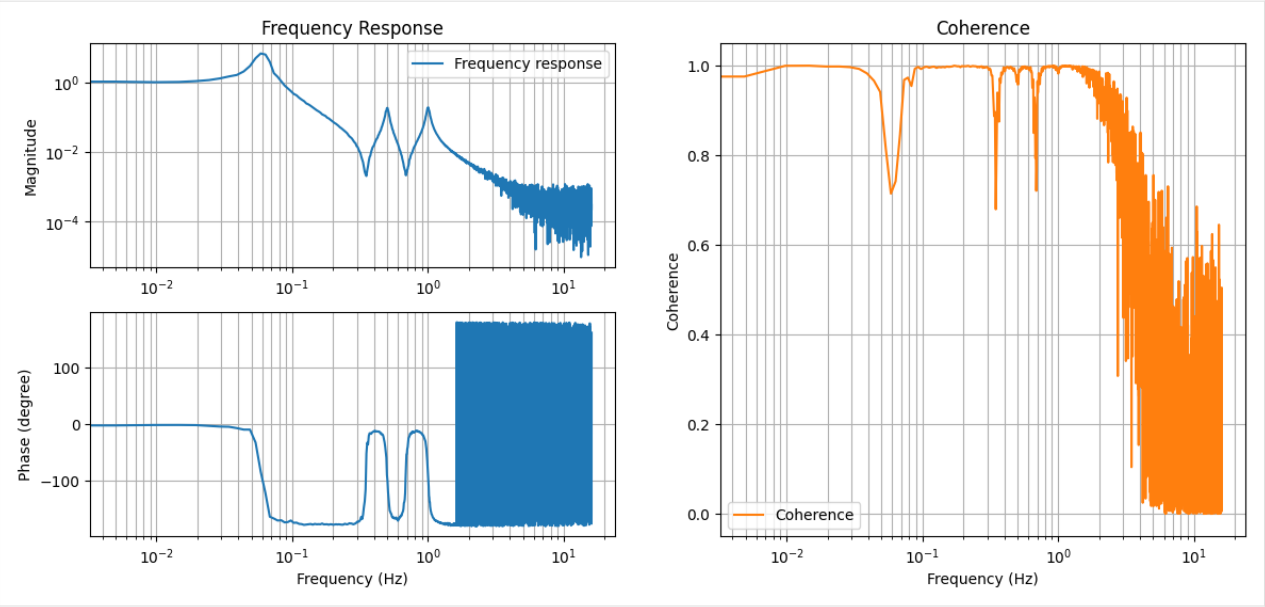
[1]:
```
import control
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol

# Define the transfer function (We don't know this yet!)
# This is the transfer function we defined for x1 in sensor matrix diagonalization␣
↪tutorial.
s = control.tf("s")
fs = 32
t = np.linspace(0, 1024, 1024*fs)

w1 = 0.06*2*np.pi
w4 = 0.5*2*np.pi
w5 = 1*2*np.pi
q1 = 10
q4 = 20
q5 = 20
tf1 = w1**2 / (s**2+w1/q1*s+w1**2)
tf4 = 0.01*w4**2 / (s**2+w4/q4*s+w4**2)
tf5 = 0.01*w5**2 / (s**2+w5/q5*s+w5**2)

tf = tf1 + tf4 + tf5

# For the sake of reproducibility
np.random.seed(123)
```

(continues on next page)

(continued from previous page)

```
# Unit white noise injection
u = np.random.normal(loc=0, scale=1/np.sqrt(fs), size=len(t))

# Measure the output
_, y, = control.forced_response(tf, U=u, T=t)

# Let's add some measurement noise
y += np.random.normal(loc=0, scale=1e-3/np.sqrt(fs), size=len(t))

# Obtain the frequency response
f, p_uy = scipy.signal.csd(u, y, fs=fs, nperseg=int(len(t)/5))
_, p_uu = scipy.signal.welch(u, fs=fs, nperseg=int(len(t)/5))
_, p_yy = scipy.signal.welch(y, fs=fs, nperseg=int(len(t)/5))

frequency_response = p_uy/p_uu
coherence = abs(p_uy)**2/p_uu/p_yy

plt.figure(figsize=(14, 6))
plt.subplot(221)
plt.title("Frequency Response")
plt.loglog(f, abs(frequency_response), label="Frequency response")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
# plt.xlabel("Frequency (Hz)")

plt.subplot(223)
plt.semilogx(f, 180/np.pi*np.angle(frequency_response))
# plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Phase (degree)")
plt.xlabel("Frequency (Hz)")

plt.subplot(122)
plt.title("Coherence")
plt.semilogx(f, coherence, color="C1", label="Coherence")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Coherence")
plt.xlabel("Frequency (Hz)")
# ^Here's what we measured
```

[1]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[2]:
```
# Data at frequency above ~2 Hz is plagued with noise, let's filter them out.
# Data at ~0.32 and ~0.65 Hz have low coherence. But that doesn't mean we should
# filter them out.
# The low coherence was due to the presences of the complex zeros in the transfer␣
↪function,
# which is a characteristic of the system we're trying to moddel.
# And we wanna get rid of the 0 frequency data point.
frequency_response_fit = frequency_response[(f>0)*(f<2)]
f_fit = f[(f>0)*(f<2)]
```

[3]:
```
# Seed the random seed
np.random.seed(2)  # For education purposes, try other values yourself.

# To use a curvefit instance, we need to define several attributes,
# xdata, ydata, model, cost, and optimizer (and optionally optimizer_kwargs, ...).
# The kontrol.curvefit.TransferFunctionFit class has predefined cost and optimizer.
# However, if we want to obtain a fit without an initial guess, we have to define a
# global optimizer (differential_evolution in this case).
curvefit = kontrol.curvefit.TransferFunctionFit()
curvefit.xdata = f_fit
curvefit.ydata = frequency_response_fit

# To use a ComplexZPK instance, we need to specify the number of complex zeros pairs and
# number of complex pole pairs.
# From the frequency response data, we see 2 notches and 3 peaks, which correspond to
# 2 complex zero pairs and 3 complex pole pairs.
# The roll-off at high frequency is 2 orders of magnitude per decade, which means
# there're 2 more poles than zeros. The analysis above agrees with this observation.
# Here, log_args=True means that we're scaling the parameters logrithmically,
# which is useful because we know the poles and zeros span across different magnitude in␣
↪frequency.
curvefit.model = kontrol.curvefit.model.ComplexZPK(nzero_pairs=2, npole_pairs=3, log_
↪args=True)
```

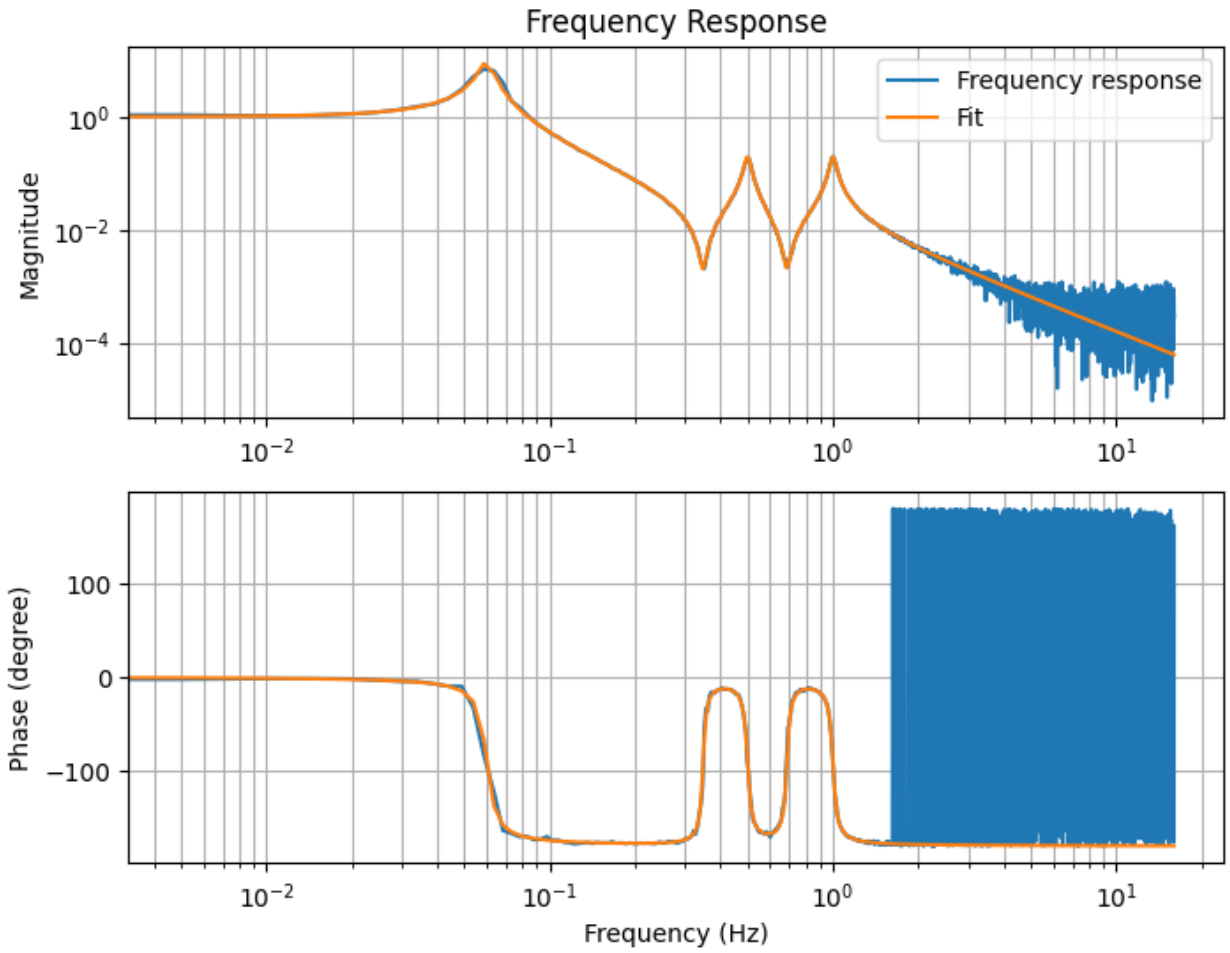
(continues on next page)

(continued from previous page)

```
# Optimizer
curvefit.optimizer = scipy.optimize.differential_evolution

# Boundary for the arguments.
# The model parameters is arranged in a list as [f1, q1, f2, q2, ... k],
# where f1, and q1 is the frequency and Q factor of the zeros/poles, and k is the static␣
↪gain.
# The bounds for the frequency is easy. We can simply define it as the min and max of the
# frequency data we obtained in the frequency response.
# The Q value denotes the height or depth of the peak or depth.
# For complex zero/pole pairs, it cannot be lower than 0.5.
# And, from the frequency response plot, the height doesn't seem to be higher than
# 2 orders of magnitude.
# Setting the upper bound to 1e3 seems reasonable.
# As for the static gain, the magnitude at low frequency is close to 1e0.
# Setting the bound between 1e-1 and 1e1 seems reasonable.

# For frequency and Q factor
# Remeber we're fitting in log scale
bounds = [(np.log10(min(f_fit)), np.log10(max(f_fit))), (np.log10(0.5), np.log10(1e3))]
bounds *= 2+3  # Number of complex zeros pairs and number of complex pole pairs

# For the gain
bound_gain = (np.log10(1e-1), np.log10(1e1))

# Finally,
bounds.append(bound_gain)

# optimizer kwargs
curvefit.optimizer_kwargs = {"bounds": bounds}

# Fit
results = curvefit.fit()
```

[4]:
```
plt.figure(figsize=(8, 6))
plt.subplot(211)
plt.title("Frequency Response")
plt.loglog(f, abs(frequency_response), label="Frequency response")
plt.loglog(f, abs(curvefit.model(f, args=results.x)), label="Fit")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
# plt.xlabel("Frequency (Hz)")

plt.subplot(212)
plt.semilogx(f, 180/np.pi*np.angle(frequency_response))
plt.semilogx(f, 180/np.pi*np.angle(curvefit.model(f, args=results.x)))
# plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Phase (degree)")
plt.xlabel("Frequency (Hz)")
```

[4]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[5]:
```
# Let's inspect the fitted transfer function
print("True transfer function:\n", tf)
print("Fitted transfer function:\n", curvefit.model.tf.minreal())
```

```
True transfer function:

       0.6356 s^4 + 0.1786 s^3 + 14.89 s^2 + 1.629 s + 56.48
-------------------------------------------------------------------
s^6 + 0.5089 s^5 + 49.56 s^4 + 11.23 s^3 + 397 s^2 + 16.01 s + 55.38

Fitted transfer function:

        0.636 s^4 + 0.1785 s^3 + 14.91 s^2 + 1.626 s + 56.65
-----------------------------------------------------------------------
s^6 + 0.5179 s^5 + 49.55 s^4 + 11.48 s^3 + 396.7 s^2 + 17.01 s + 55.2
```

[6]:
```
# ^Close enough!
# Let's export the transfer function for future purposes.
curvefit.model.tf.save("transfer_function_x1_without_guess.pkl")
```

[7]:
```
# Alternatively, export the Foton string and install it to the digital system
curvefit.model.tf.foton(root_location="n")
```

[7]:
```
'zpk([0.006973+i*0.347667;0.006973+i*-0.347667;0.015365+i*0.687354;0.015365+i*-0.687354],
↪[0.003204+i*0.059844;0.003204+i*-0.059844;0.012652+i*0.499616;0.012652+i*-0.499616;0.
↪025355+i*0.999705;0.025355+i*-0.999705],1.02636,"n")'
```

Transfer Function Modeling (with initial guess)

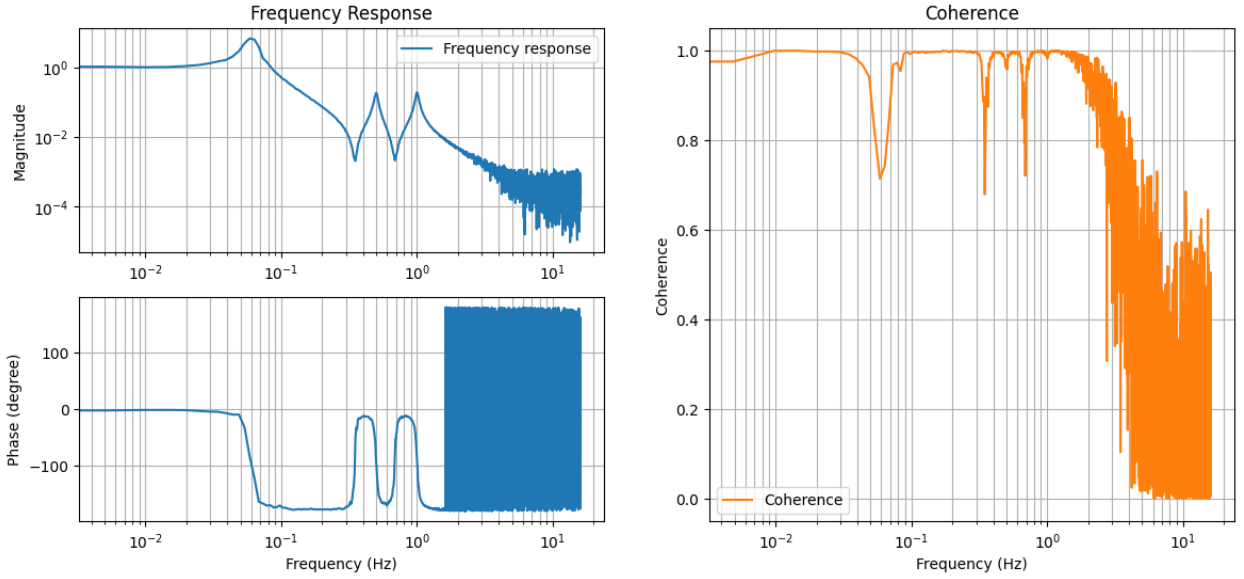
[1]:
```
import control
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol

# Define the transfer function (We don't know this yet!)
# This is the transfer function we defined for x1 in sensor matrix diagonalization␣
↪tutorial.
s = control.tf("s")
fs = 32
t = np.linspace(0, 1024, 1024*fs)

w1 = 0.06*2*np.pi
w4 = 0.5*2*np.pi
w5 = 1*2*np.pi
q1 = 10
q4 = 20
q5 = 20
tf1 = w1**2 / (s**2+w1/q1*s+w1**2)
tf4 = 0.01*w4**2 / (s**2+w4/q4*s+w4**2)
tf5 = 0.01*w5**2 / (s**2+w5/q5*s+w5**2)

tf = tf1 + tf4 + tf5

# For the sake of reproducibility
np.random.seed(123)

# Unit white noise injection
u = np.random.normal(loc=0, scale=1/np.sqrt(fs), size=len(t))

# Measure the output
_, y, = control.forced_response(tf, U=u, T=t)

# Let's add some measurement noise
y += np.random.normal(loc=0, scale=1e-3/np.sqrt(fs), size=len(t))

# Obtain the frequency response
f, p_uy = scipy.signal.csd(u, y, fs=fs, nperseg=int(len(t)/5))
_, p_uu = scipy.signal.welch(u, fs=fs, nperseg=int(len(t)/5))
_, p_yy = scipy.signal.welch(y, fs=fs, nperseg=int(len(t)/5))
```

(continues on next page)

(continued from previous page)

```
frequency_response = p_uy/p_uu
coherence = abs(p_uy)**2/p_uu/p_yy

plt.figure(figsize=(14, 6))
plt.subplot(221)
plt.title("Frequency Response")
plt.loglog(f, abs(frequency_response), label="Frequency response")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
# plt.xlabel("Frequency (Hz)")

plt.subplot(223)
plt.semilogx(f, 180/np.pi*np.angle(frequency_response))
# plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Phase (degree)")
plt.xlabel("Frequency (Hz)")

plt.subplot(122)
plt.title("Coherence")
plt.semilogx(f, coherence, color="C1", label="Coherence")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Coherence")
plt.xlabel("Frequency (Hz)")
# ^Here's what we measured
```

[1]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[2]:
```
# Data at frequency above ~2 Hz is plagued with noise, let's filter them out.
# Data at ~0.32 and ~0.65 Hz have low coherence. But that doesn't mean we should
```

(continues on next page)

(continued from previous page)

```
# filter them out.
# The low coherence was due to the presences of the complex zeros in the transfer␣
↪function,
# which is a characteristic of the system we're trying to moddel.
# And we wanna get rid of the 0 frequency data point.
frequency_response_fit = frequency_response[(f>0)*(f<2)]
f_fit = f[(f>0)*(f<2)]
```

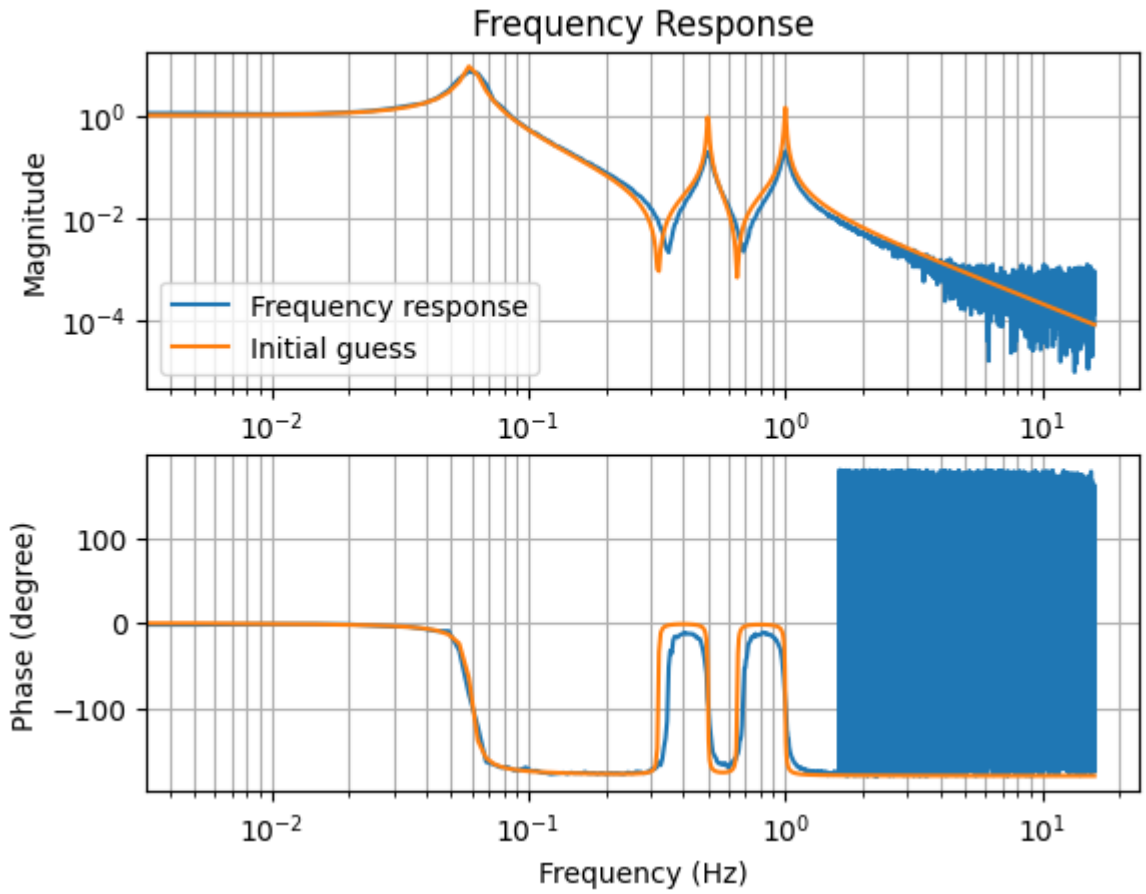
[3]:
```
# Seed the random seed
np.random.seed(2)  # For education purposes, try other values yourself.

# To use a curvefit instance, we need to define several attributes,
# xdata, ydata, model, cost, and optimizer (and optionally optimizer_kwargs, ...).
# The kontrol.curvefit.TransferFunctionFit class has predefined cost and optimizer.
# In addition, we have to specify an intial guess x0.
curvefit = kontrol.curvefit.TransferFunctionFit()
curvefit.xdata = f_fit
curvefit.ydata = frequency_response_fit

# To use a ComplexZPK instance, we need to specify the number of complex zeros pairs and
# number of complex pole pairs.
# From the frequency response data, we see 2 notches and 3 peaks, which correspond to
# 2 complex zero pairs and 3 complex pole pairs.
# The roll-off at high frequency is 2 orders of magnitude per decade, which means
# there're 2 more poles than zeros. The analysis above agrees with this observation.
# Here, log_args=True means that we're scaling the parameters logrithmically,
# which is useful because we know the poles and zeros span across different magnitude in␣
↪frequency.
curvefit.model = kontrol.curvefit.model.ComplexZPK(nzero_pairs=2, npole_pairs=3, log_
↪args=True)

# Inital guess, peaks are at 0.06 (Q~10), 0.5 (Q~100), and 1 (Q~100) Hz.
# Notches at 0.32 Hz (Q~100), 0.65 Hz (Q~100)
# Gain at 1.

# The model parameters is arranged in a list as [f1, q1, f2, q2, ... k],
# where f1, and q1 is the frequency and Q factor of the zeros/poles, and k is the static␣
↪gain.
x0 = [0.32, 100, 0.65, 100, 0.06, 10, 0.5, 100, 1, 100, 1]
x0 = np.log10(x0)  # Remember we're fitting the logarithm of the parameters.

# Compare inital guess with the measurement.
# plt.figure(figsize=(8, 6))
plt.subplot(211)
plt.title("Frequency Response")
plt.loglog(f, abs(frequency_response), label="Frequency response")
plt.loglog(f, abs(curvefit.model(f, args=x0)), label="Initial guess")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
# plt.xlabel("Frequency (Hz)")
```

(continues on next page)

(continued from previous page)

```
plt.subplot(212)
plt.semilogx(f, 180/np.pi*np.angle(frequency_response))
plt.semilogx(f, 180/np.pi*np.angle(curvefit.model(f, args=x0)))
# plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Phase (degree)")
plt.xlabel("Frequency (Hz)")
```

[3]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[4]:
```
# ^Inital guess good enough, let's proceed.
curvefit.x0 = x0
curvefit.options = {"maxiter":len(x0)*1000}  # options given to the scipy.optimize.
↪minimize function.
# We increase the maximum number of iteration because it wasn't enough.
results = curvefit.fit()
```

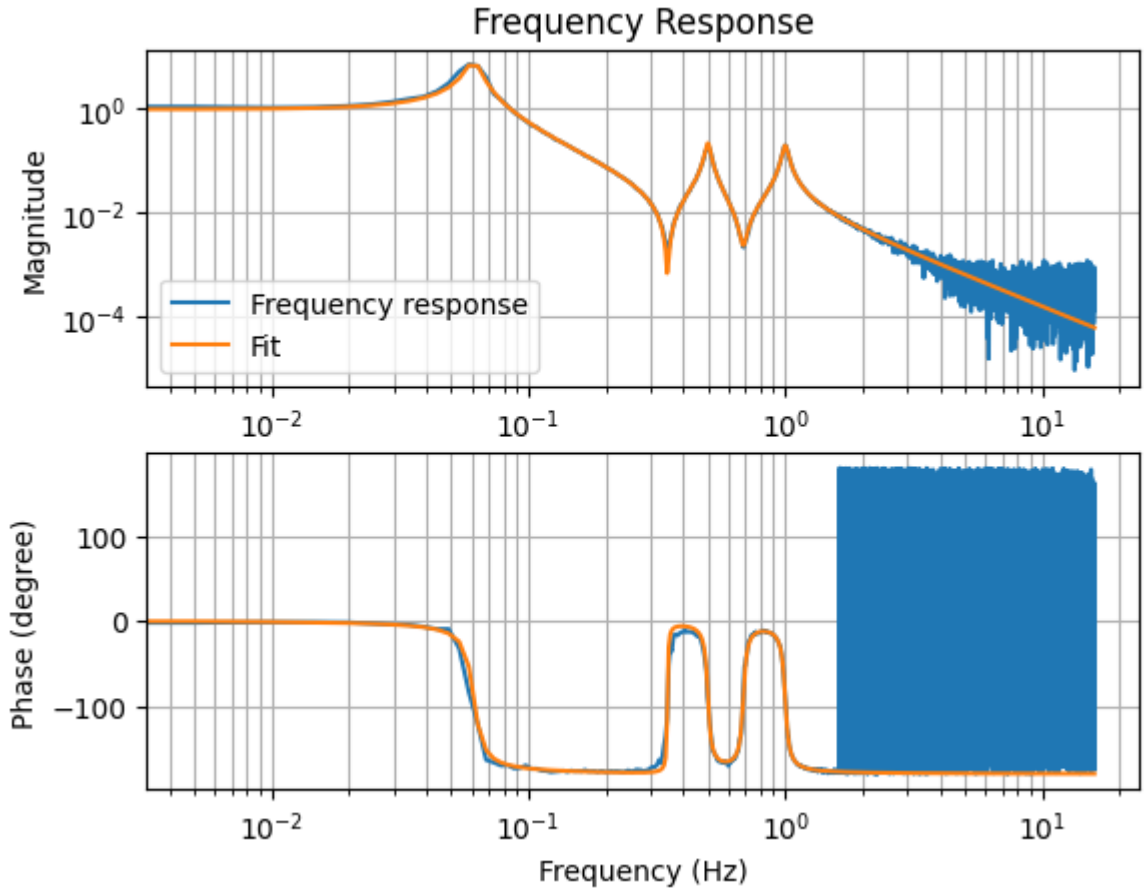
[5]:
```
# plt.figure(figsize=(8, 6))
plt.subplot(211)
plt.title("Frequency Response")
plt.loglog(f, abs(frequency_response), label="Frequency response")
plt.loglog(f, abs(curvefit.model(f, args=results.x)), label="Fit")
plt.legend(loc=0)
```

(continues on next page)

(continued from previous page)

```
plt.grid(which="both")
plt.ylabel("Magnitude")
# plt.xlabel("Frequency (Hz)")

plt.subplot(212)
plt.semilogx(f, 180/np.pi*np.angle(frequency_response))
plt.semilogx(f, 180/np.pi*np.angle(curvefit.model(f, args=results.x)))
# plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Phase (degree)")
plt.xlabel("Frequency (Hz)")
```

[5]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[6]:
```
# Let's inspect the fitted transfer function
print("True transfer function:\n", tf)
print("Fitted transfer function:\n", curvefit.model.tf.minreal())
```

```
True transfer function:

       0.6356 s^4 + 0.1786 s^3 + 14.89 s^2 + 1.629 s + 56.48
-------------------------------------------------------------------
s^6 + 0.5089 s^5 + 49.56 s^4 + 11.23 s^3 + 397 s^2 + 16.01 s + 55.38
```

(continues on next page)

(continued from previous page)

```
Fitted transfer function:

        0.6354 s^4 + 0.1516 s^3 + 14.86 s^2 + 0.9622 s + 56.46
-------------------------------------------------------------------------
s^6 + 0.5065 s^5 + 49.55 s^4 + 11.09 s^3 + 396.9 s^2 + 19.01 s + 57.55
```

[7]:
```
# ^Close enough!
# Let's export the transfer function for future purposes.
curvefit.model.tf.save("transfer_function_x1_with_guess.pkl")
```

[8]:
```
# Alternatively, export the Foton string and install it to the digital system
curvefit.model.tf.foton(root_location="n")
```

[8]:
```
'zpk([0.002157+i*0.347726;0.002157+i*-0.347726;0.016831+i*0.686452;0.016831+i*-0.686452],
↪[0.003622+i*0.061092;0.003622+i*-0.061092;0.011376+i*0.499619;0.011376+i*-0.499619;0.
↪025306+i*0.999682;0.025306+i*-0.999682],0.981029,"n")'
```

Using kontrol, we obtained a transfer function of the system:

$$\frac{0.635s^4 + 0.1785s^3 + 14.91s^2 + 1.626s + 56.65}{s^6 + 0.5719s^5 + 49.55s^4 + 11.48s^3 + 396.7s^2 + 17.01s + 55.2}$$

and we've used `kontrol.TransferFunction.save()` method to export the transfer function object for future purposes.

Tips:

- Without the initial guess, the parameter space must be bounded. There's no guarantee that the solution coverged to a global optimum. We've to use a differnt random number seed and many trials to obtain satisfactory results.
- With the initial guess, the fit refines the initial parameters without iterations. However, it could require experience to obtain an initial guess.

Controller design

With the transfer function of the system identified, we can start designing a controller for damping and position control purposes. In this section, we will use the plant identified in *Transfer function modeling* and create controllers around it.

Damping control

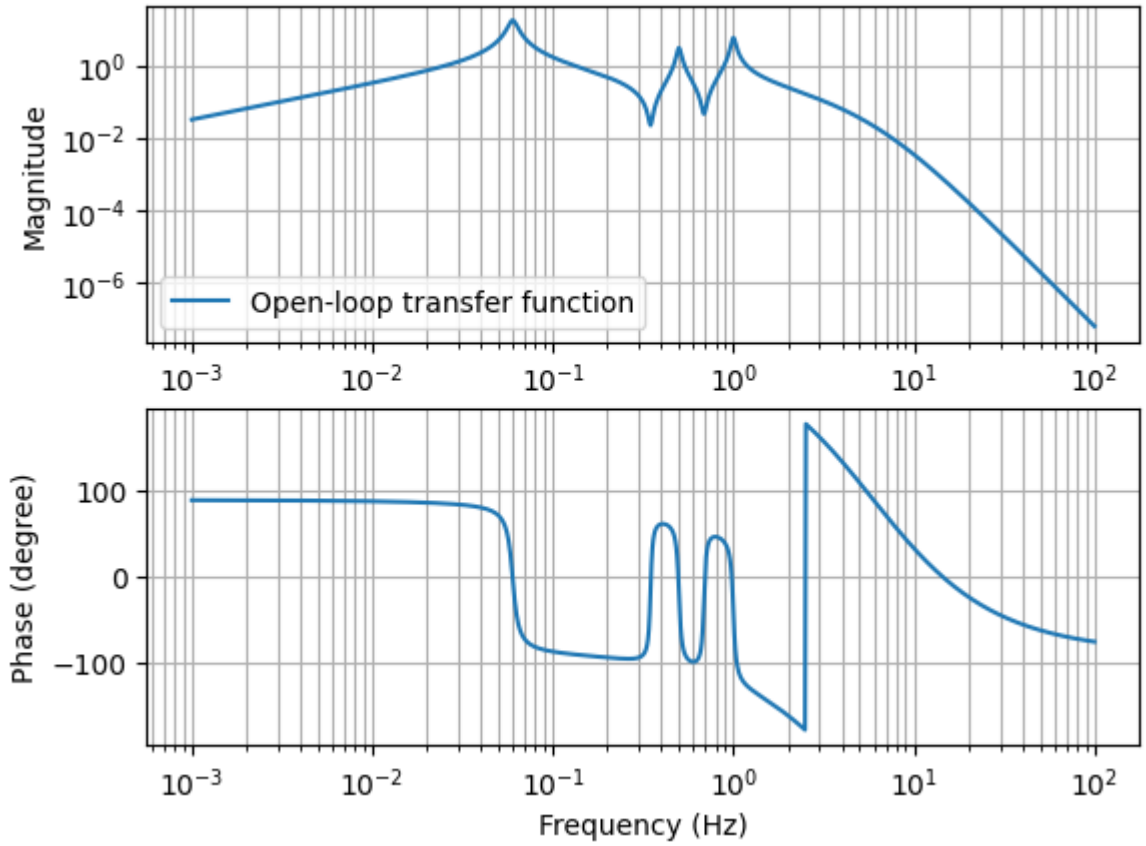
We were asked to design a damping controller for the suspension. The resonances of the system are annoying since they amplify ground motion. And, when responding to an impulse input, the oscilaltion takes a long time to be naturally damped.

The goal is to design a damping controller to suppress the suspension motion quickly after being hit by an impulse. We also need an additional 4th-order low-pass filter to reduce the noise being injected at high frequency and maybe notch filters maintain stability. Click the link below to see how we can use the `kontrol.regulator` module to design the appropriate controllers.

Damping control

[7]:
```
import control
import numpy as np
import matplotlib.pyplot as plt

import kontrol


# Load the transfer function
plant = kontrol.load_transfer_function("../system_modeling/transfer_function_x1_without_
↪guess.pkl")

# Get a controller. regulator_type="D" for derivative control, which is a natural choice␣
↪for damping.
# This function generates a critical damping controller.
controller = kontrol.regulator.oscillator.pid(plant, regulator_type="D")

# Get a low-pass filter.
# This lower the cut-off frequency of the low-pass filter until
# a specified phase margin is attained (defaults to 45 degrees.)
# Use the phase_margin option to override this.
low_pass = kontrol.regulator.post_filter.post_low_pass(plant, regulator=controller,␣
↪order=4)

# Final filter.
controller *= low_pass

# Inspect the open-loop transfer function
oltf = controller*plant

f = np.logspace(-3, 2, 1024)

plt.subplot(211)
plt.loglog(f, abs((oltf)(1j*2*np.pi*f)), label="Open-loop transfer function")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")

plt.subplot(212)
plt.semilogx(f, 180/np.pi*np.angle(oltf(1j*2*np.pi*f)))
# plt.legend(loc=0)
plt.grid(which="both")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Phase (degree)")
```

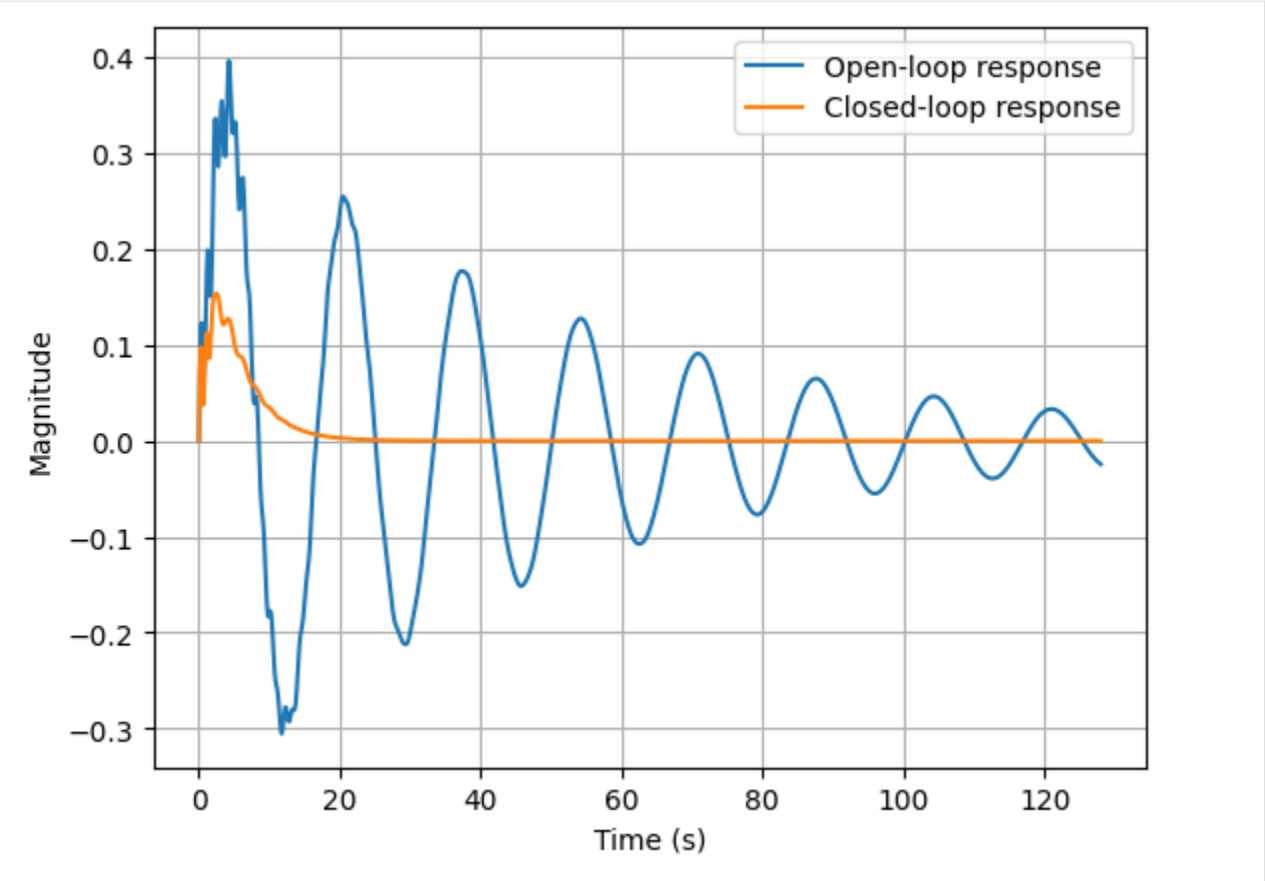
[7]:
```
Text(0, 0.5, 'Phase (degree)')
```



[2]:
```
# ^This system has a UGF of around 1.2 Hz with a phase margin of about 45 degrees.
# Let's look at the open-loop and closed-loop performance against an impulse
t = np.linspace(0, 128, 1024)
_, y_open = control.impulse_response(plant, T=t)
_, y_close = control.impulse_response(plant/(1+oltf), T=t)
```

[3]:
```
plt.plot(t, y_open, label="Open-loop response")
plt.plot(t, y_close, label="Closed-loop response")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
plt.xlabel("Time (s)")
# ^Damped.
```

[3]:
```
Text(0.5, 0, 'Time (s)')
```



[6]:
```
# Let's export the Foton string so we can install it into the digital system
controller = kontrol.TransferFunction(controller)
controller.foton(root_location="n")
```

[6]:
```
'zpk([-0],[5.960099+i*0.001106;5.960099+i*-0.001106;5.962312+i*0.001107;5.962312+i*-0.
↪001107],32.5136,"n")'
```

We've used `kontrol.load_transfer_function()` to import the transfer function we've saved before. Using `kontrol.regulator.oscillator.pid()` and `kontrol.regulator.post_filter.post_low_pass()` functions, we obtained a damping controller

$$K(s) = \frac{1.018 \times 10^7 s}{s^4 + 149.8 s^3 + 2.102 \times 10^5 s + 1.968 \times 10^6},$$

which critically damps the system.

The controller seems to be able to damp the system reasonable well and we decided to export the Foton string by converting the controller into a `kontrol.TransferFunction` object and use `kontrol.TransferFunction.foton()` method to export the Foton string.

```
zpk([-0],[5.960099+i*0.001106;5.960099+i*-0.001106;5.962312+i*0.001107;5.962312+i*-0.
↪001107],32.5136,"n")
```

Position control

While the suspension is being actively damped, this is not enough. For some degrees of freedom, we want be able to control the system to follow a setpoint. After all, the suspension hangs the mirror and provides coarse alignment control. We were ask to design another controller that provides both position and damping control. In addition, we received complaints from others saying that the damping control inject too much noise at high frequencies. We were also told that we don't need to damp the higher-frequency modes so the cut-off frequency of the low-pass can be lowered so we can reduce injected noise. Click the link below to see how can use the `kontrol.regulator` module to achieve all the above.

Position control

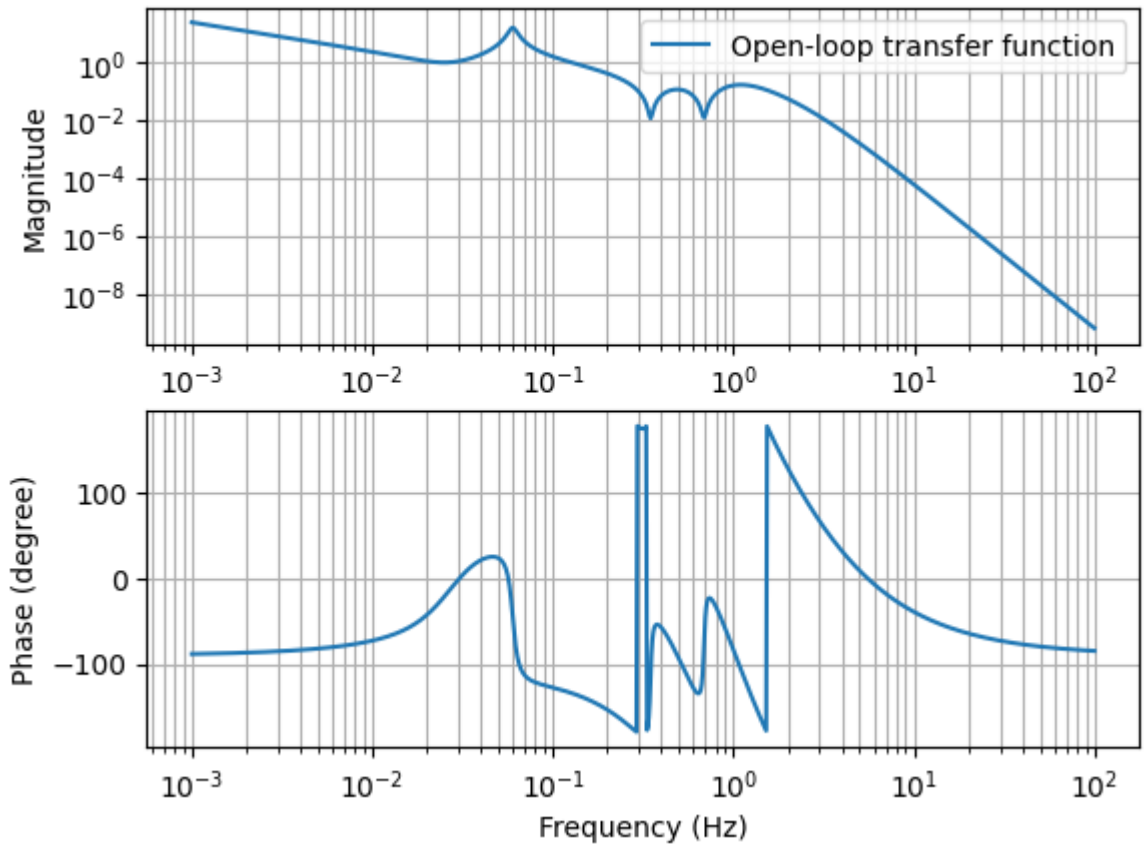
[1]:
```
import control
import numpy as np
import matplotlib.pyplot as plt

import kontrol


# Load the transfer function
plant = kontrol.load_transfer_function("../system_modeling/transfer_function_x1_without_
↪guess.pkl")

# Get a controller. regulator_type="PID" for PID controller.
# The I component gives the system the ability to trace a setpoint with 0 error.
# The P component speeds up the control.
controller = kontrol.regulator.oscillator.pid(plant, regulator_type="PID")

# We don't need to damp the high frequency modes.
# Get some notch filters
# the notch_peaks_above=0.1 means we'll notch all peaks above 0.1 Hz.
notches = kontrol.regulator.post_filter.post_notch(plant, regulator=controller, notch_
↪peaks_above=0.1)
# ^this returns a list of notch filters.
notches = np.prod(notches) # To get a transfer function, we take the product of the␣
↪notches.

# Get a low-pass filter.
# This lower the cut-off frequency of the low-pass filter until
# a specified phase margin is attained (defaults to 45 degrees.)
# Use the phase_margin option to override this.
low_pass = kontrol.regulator.post_filter.post_low_pass(plant, regulator=controller, post_
↪filter=notches, order=4)

# Final filter.
controller *= low_pass*notches

# Inspect the open-loop transfer function
oltf = controller*plant

f = np.logspace(-3, 2, 1024)
```

(continues on next page)

(continued from previous page)

```
plt.subplot(211)
plt.loglog(f, abs((oltf)(1j*2*np.pi*f)), label="Open-loop transfer function")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")

plt.subplot(212)
plt.semilogx(f, 180/np.pi*np.angle(oltf(1j*2*np.pi*f)))
# plt.legend(loc=0)
plt.grid(which="both")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Phase (degree)")
```

[1]:
```
Text(0, 0.5, 'Phase (degree)')
```



[2]:
```
# ^This system has a UGF of around 0.128 Hz with a phase margin of about 45 degrees.
# Let's look at the open-loop and closed-loop performance against a step response, i.e. setpoint=1 at t=0.
t = np.linspace(0, 128, 1024)
_, y_open = control.step_response(plant, T=t)
_, y_close = control.step_response(oltf/(1+oltf), T=t)
```

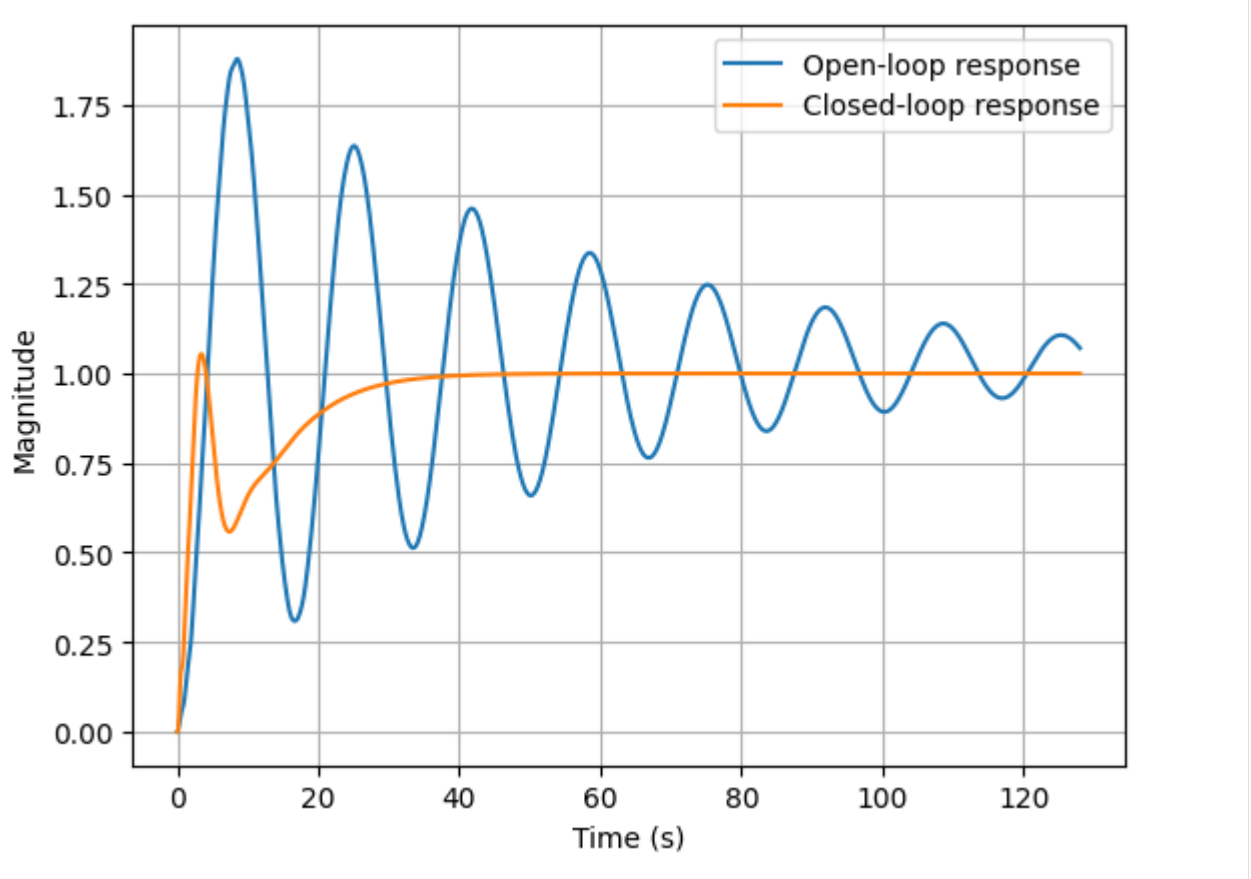
[3]:
```
plt.plot(t, y_open, label="Open-loop response")
plt.plot(t, y_close, label="Closed-loop response")
```

(continues on next page)

(continued from previous page)

```
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
plt.xlabel("Time (s)")
# Positon controlled
```

[3]:
```
Text(0.5, 0, 'Time (s)')
```



[4]:
```
# Let's export the Foton string so we can install it into the digital system
controller = kontrol.TransferFunction(controller)
controller.foton(root_location="n")
```

[4]:
```
'zpk([0.012454+i*0.024317;0.012454+i*-0.024317;0.012652+i*0.499616;0.012652+i*-0.499616;
↪0.025355+i*0.999705;0.025355+i*-0.999705],[-0;0.249888+i*0.432819;0.249888+i*-0.432819;
↪0.500013+i*0.866048;0.500013+i*-0.866048;1.887186+i*0.000255;1.887186+i*-0.000255;1.
↪887697+i*0.000256;1.887697+i*-0.000256],0.024269,"n")'
```

And we eventually obtained a controller. The open-loop transfer function has a unit gain frequency of 0.128 Hz and phase margin of 45 degrees. The gain at 10 Hz is 2 orders of magnitude lower than the one we obtained in *Damping control*. It follows that the noise injected is also that much lower. From simulation, the system is able to trace a unit step input without excess oscillation. We're happy with the results and exported the final controller.

```
zpk([0.012454+i*0.024317;0.012454+i*-0.024317;0.012652+i*0.499616;0.012652+i*-0.499616;0.
↪025355+i*0.999705;0.025355+i*-0.999705],[-0;0.249888+i*0.432819;0.249888+i*-0.432819;0.
↪500013+i*0.866048;0.500013+i*-0.866048;1.887186+i*0.000255;1.887186+i*-0.000255;1.
↪887697+i*0.000256;1.887697+i*-0.000256],0.024269,"n")
```

And, this concludes the basic suspension commissioning section: We're able to setup the sensors and actuators to measure the frequency response of the system. We also modeled the frequency response of the system and contructed 2 types of controllers for it.

But, we're not satisfied. In the *Advanced Control Methods* section, we will continue working on advanced techniques to further improve the active control performance.

1.3.2 Advanced Control Methods

H-infinity method

We were asked to improve the seismic isolation performance. The system that we set up using the methods above don't help mitigating seismic noise since we were only using relative sensors. We cannot achieve active seismic isolation without utilizing inertial sensors.

There are several things that we can do to achieve active isolation and in some sense optimize it. Chapter 7 and 8 in Ref.[?] describe several concepts in seismic isolation: sensor fusion, sensor correction, and feedback control, and propose an H-infinity method to optimize these subsystems in a seismic isolation system.

H-infinity sensor fusion

There're two types of sensors that are used to measure the motion of the inverted pendulum, relative sensor and inertial sensor. The two sensors have different noise characteristics, one has better noise performance at some frequencies and the other one is better at other frequencies. We were asked to design a set of complementary filters which can be used to combine the two sensors into a "super sensor" that has the advantages of both sensors.

`Kontrol` provides an H-infinity approach to optimize complementary filters. To use this, we'll need 2 things.

1. Spectrums of the sensor noises.
2. Transfer functions that have magnitude responses matching the spectrums of the noises.

In this section, we'll demonstrate how we can use Kontrol to optimize complementary filters using the H-infinity method. The first two subsections show how we can obtain the necessary materials mentioned above and the last subsection shows how we can obtain the optimal complementary filters given those materials.

We also assume that the relative sensor and the inertial sensor are aligned and inter-calibrated so they read the same signal. **This is extremely important**.

Caveats

This section solves the sensor fusion problem using the H-infinity method. The case presented is a hypothetical scenario as we only take the intrinsic sensor noises of the relative and inertial sensors into account, whereas there're more noises involved in reality. Doing so allows the demonstration to be simplified without much complications. So, please keep in mind that this section only serves the purpose of laying down the necessary procedures to use the method. Therefore, if you wish to use this method, do take into account all necessary noise sources in the sensors.

In some systems, the relative sensors are "corrected" via a control scheme called "sensor correction" and it is the corrected relative sensor that is combined with the inertial sensor. The corrected sensor has a noise profile depending on the sensor correction filter that we will be optimizing in the next section *H-infinity sensor correction*. But because the sensor correction and other problems can be treated as a sensor fusion problem, it is beneficial to show how a sensor fusion problem is solved before going into other problems. This is why we begin with a hypotheical sensor fusion scenario.

The correct procedure is:

1. Firstly optimize a sensor correction filter.
2. Evalute the effective noise presence in the corrected relative readout.

3. Model and use that to optimize the complementary filters instead.

Estimating inertial sensor noise

Sensor noise of an inertial sensor cannot be measured individually because there's always some signal that is present in the readout. However, if we have multiple sensors measuring a common signal, we can use the 3-channel correlation method to estimate all individual sensor noises of the 3 sensors.

We placed 3 inertial sensors on the ground and aligned them so they measure the ground motion in the same direction. Click the link below to see how we can use the `kontrol.spectral.three_channel_correlation()` function to estimate the sensor noises from the spectrums.

Three Channel Correlation Method

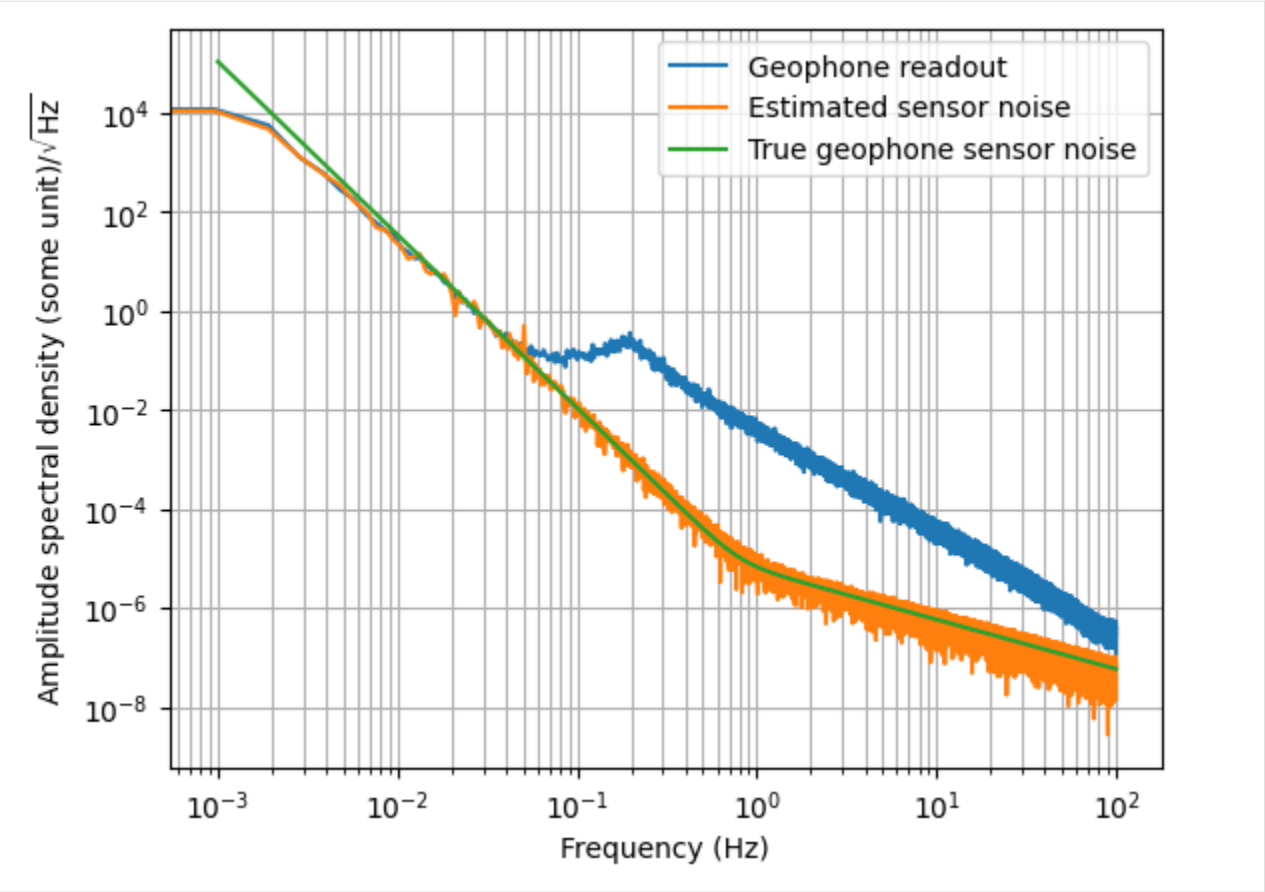
[1]:
```
import control
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol


# Generate measurements
# Geophone noise spectrum (micron/sqrt(Hz))
f = np.linspace(1e-3, 1e2, 1024*512)
def noise_model(f, na, nb, a, b):
    return np.sqrt((na/f**a)**2 + (nb/f**b)**2)
n_geophone = noise_model(f, na=1*10**-5.46, nb=1*10**-5.23, a=3.5, b=1)
n_relative = noise_model(f, na=1*10**-2.07, nb=1*10**-2.3, a=0.5, b=0)

# Ground motion
s = control.tf("s")
wn = 0.2*2*np.pi  # Secondary microseism
q = 2.5  # 10 times height
ground_motion_tf = 0.1 * wn**2 / (s**2+wn/q*s+wn**2)

# Generate time series measurements
np.random.seed(1)
t, geophone_noise1 = kontrol.spectral.asd2ts(n_geophone, f=f)
_, geophone_noise2 = kontrol.spectral.asd2ts(n_geophone, f=f)
_, geophone_noise3 = kontrol.spectral.asd2ts(n_geophone, f=f)
_, relative_noise = kontrol.spectral.asd2ts(n_relative, f=f)

# Ground motion
fs = 1/(t[1]-t[0])
u = np.random.normal(loc=0, scale=np.sqrt(fs/2), size=len(t))
_, ground_motion = control.forced_response(ground_motion_tf, U=u, T=t)

# 3 geophone measurements
readout1 = ground_motion + geophone_noise1
readout2 = ground_motion + geophone_noise2
```

(continues on next page)

(continued from previous page)

```
readout3 = ground_motion + geophone_noise3
readout_relative = relative_noise

# Obtain the measured spectral densities.
fs = 1/(t[1]-t[0])
f_, p11 = scipy.signal.welch(readout1, fs=fs, nperseg=int(len(t)/5))
f_, p22 = scipy.signal.welch(readout2, fs=fs, nperseg=int(len(t)/5))
f_, p33 = scipy.signal.welch(readout3, fs=fs, nperseg=int(len(t)/5))
f_, noise_relative = scipy.signal.welch(readout_relative, fs=fs, nperseg=int(len(t)/5))

_, p12 = scipy.signal.csd(readout1, readout2, fs=fs, nperseg=int(len(t)/5))
_, p13 = scipy.signal.csd(readout1, readout3, fs=fs, nperseg=int(len(t)/5))
_, p21 = scipy.signal.csd(readout2, readout1, fs=fs, nperseg=int(len(t)/5))
_, p23 = scipy.signal.csd(readout2, readout3, fs=fs, nperseg=int(len(t)/5))
_, p31 = scipy.signal.csd(readout3, readout1, fs=fs, nperseg=int(len(t)/5))
_, p32 = scipy.signal.csd(readout3, readout2, fs=fs, nperseg=int(len(t)/5))
# ^Measurement data generated.

# Estimate the sensor nosie using three channel correlation method
noise1, noise2, noise3 = kontrol.spectral.three_channel_correlation(
    p11, psd2=p22, psd3=p33, csd12=p12, csd13=p13, csd21=p21, csd23=p23, csd31=p31,␣
↪csd32=p32)

plt.loglog(f_, p11**0.5, label="Geophone readout")
plt.loglog(f_, noise1**0.5, label="Estimated sensor noise")
# plt.loglog(f_, noise_relative**0.5, label="Relative sensor noise")
plt.loglog(f, n_geophone, label="True geophone sensor noise")
# plt.loglog(f, n_relative, label="Relative")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[1]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[2]:
```
# Export the noises
import pickle

with open("noise_spectrum_inertial.pkl", "wb") as fh:
    pickle.dump(noise1**0.5, fh)
with open("noise_spectrum_relative.pkl", "wb") as fh:
    pickle.dump(noise_relative**0.5, fh)
with open("noise_spectrum_frequency.pkl", "wb") as fh:
    pickle.dump(f_, fh)
```

And, we were able to generate an estimation of the inertial sensor noise spectrum. And we've exported the noise spectrums for future usages.

Sensor noise modeling

Now that the sensor noises are identified, we can construct transfer function models for them. The `kontrol.curvefit.TransferFunctionFit` class that we used earlier for modeling frequency response data can be used for this purpose. But, there's a wrapper function that is better fitted for our purpose as it's not nearly as tedious. See the notebook below to see how we can use the `kontrol.curvefit.spectrum_fit()` function to model spectrums as the magnitude responses of transfer functions.

Sensor Noise Modeling

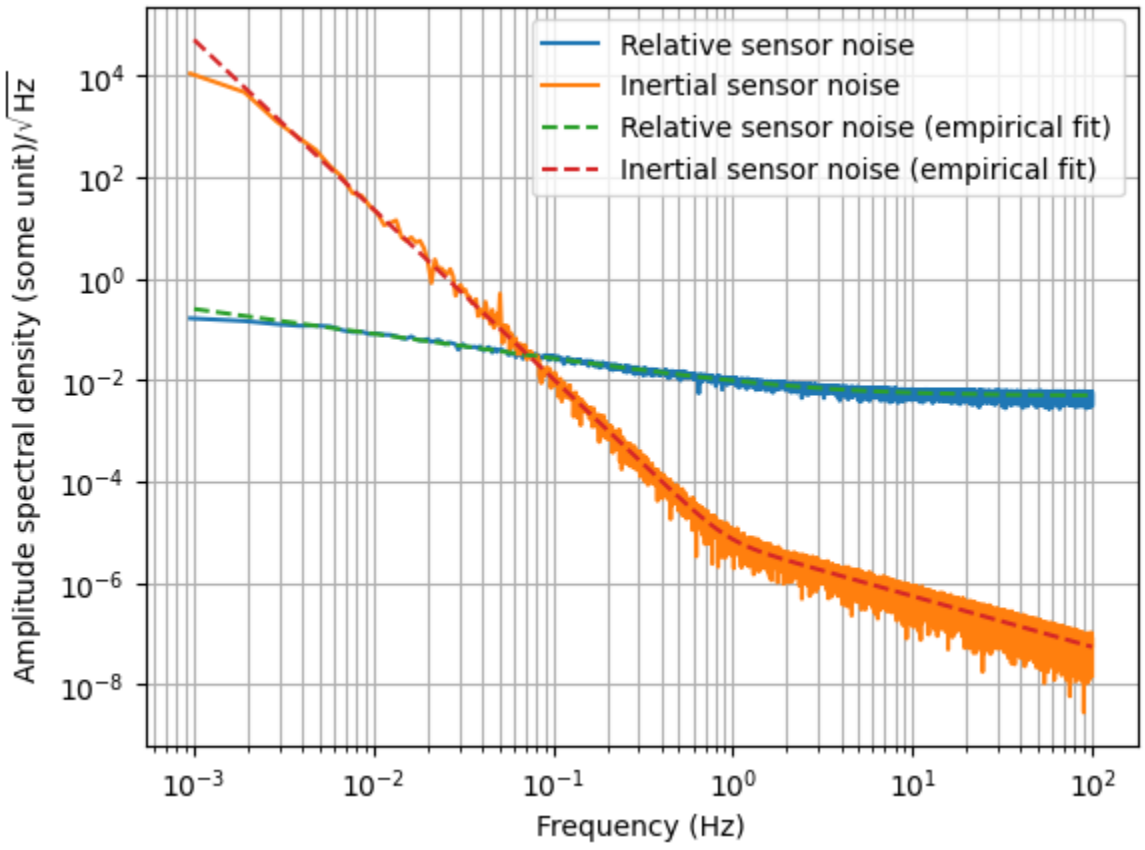
[1]:
```
import pickle

import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol


# Load sensor noise data generated earlier.
with open("noise_spectrum_frequency.pkl", "rb") as fh:
    f = pickle.load(fh)
with open("noise_spectrum_relative.pkl", "rb") as fh:
    noise_relative = pickle.load(fh)
with open("noise_spectrum_inertial.pkl", "rb") as fh:
    noise_inertial = pickle.load(fh)

# Get rid of the DC value
noise_relative = noise_relative[f>0]
noise_inertial = noise_inertial[f>0]
f = f[f>0]

# Fitting an empirical model is helpful but not necessary.
# Here we assume we know the empirical form of the noise and
# simply use a generic minimization algorithm to fit the empirical models.
def noise_model(f, na, nb, a, b):
    na = 10**na  # Parameter rescaling.
    nb = 10**nb
    return np.sqrt((na/f**a)**2 + (nb/f**b)**2)


def cost(args, xdata, ydata):
    na, nb, a, b = args
    ymodel = noise_model(xdata, na, nb, a, b)
    error = kontrol.curvefit.error_func.noise_error(ydata, ymodel)  # mean square␣
↪logarithmic error.
    return error


res_relative = scipy.optimize.minimize(cost, args=(f, noise_relative), x0=[-2.07, -2.3,␣
↪0.5, 0])
res_inertial = scipy.optimize.minimize(cost, args=(f, noise_inertial), x0=[-5.46, -5.23,␣
↪3.5, 1])
# ^If we cannot guess the x0 values, we can try to use a global optimization alogrithm
# such as differential evolution instead.

# With the empirical model found, we can define a logspace frequency axis,
# which helps a lot with the next steps.
f_log = np.logspace(-3, 2, 1024)  # And with less data points so fit faster.
noise_relative_empirical = noise_model(f_log, *res_relative.x)
```

(continues on next page)

(continued from previous page)

```
noise_inertial_empirical = noise_model(f_log, *res_inertial.x)

# Visualize
plt.loglog(f, noise_relative, label="Relative sensor noise")
plt.loglog(f, noise_inertial, label="Inertial sensor noise")
plt.loglog(f_log, noise_relative_empirical, "--", label="Relative sensor noise␣
↪(empirical fit)")
plt.loglog(f_log, noise_inertial_empirical, "--", label="Inertial sensor noise␣
↪(empirical fit)")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[1]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



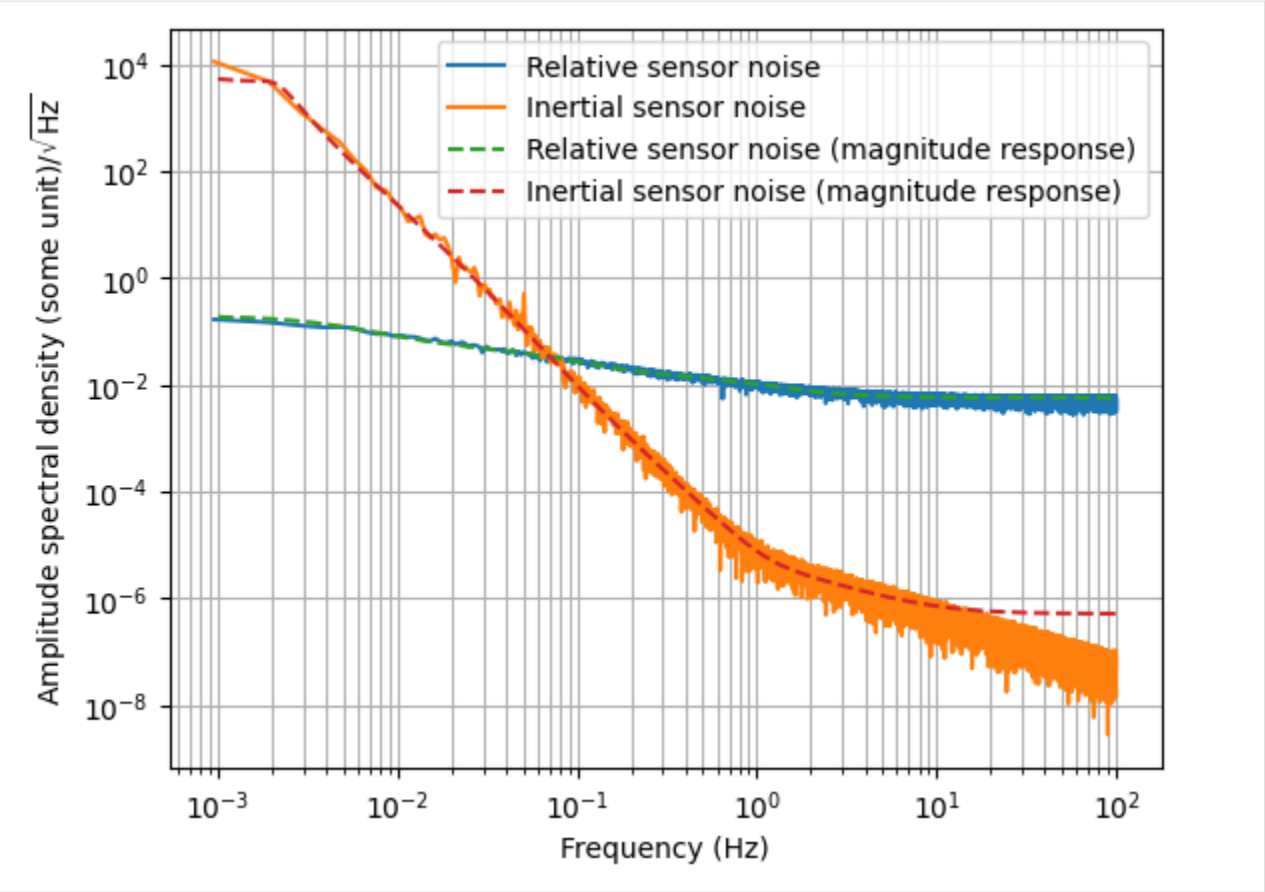
[2]:
```
# Now we see the empirical model that we obtained fits the noise spectrum well
# we can model the empirical model itself instead of the measured noise data.
# H-infinity method requires transfer functions to be flat at both ends
# so we need to flatten the data.
# Select the frequency band of interests.
f_lower = 2e-3
f_upper = 1e1
noise_relative_flat = noise_relative_empirical.copy()
```

(continues on next page)

(continued from previous page)

```
noise_inertial_flat = noise_inertial_empirical.copy()
noise_relative_flat[f_log<f_lower] = noise_relative_empirical[f_log>f_lower][0]
noise_relative_flat[f_log>f_upper] = noise_relative_empirical[f_log<f_upper][-1]
noise_inertial_flat[f_log<f_lower] = noise_inertial_empirical[f_log>f_lower][0]
noise_inertial_flat[f_log>f_upper] = noise_inertial_empirical[f_log<f_upper][-1]

# Model the flattened spectrums with magnitude responses
order_relative = 3  # Order of the transfer functions. Guess.
order_inertial = 4
tf_relative = kontrol.curvefit.spectrum_fit(f_log, noise_relative_flat,
                                            nzero=order_relative, npole=order_relative)
tf_inertial = kontrol.curvefit.spectrum_fit(f_log, noise_inertial_flat,
                                            nzero=order_inertial, npole=order_inertial)

# Optionally inspect.
plt.loglog(f, noise_relative, label="Relative sensor noise")
plt.loglog(f, noise_inertial, label="Inertial sensor noise")
# plt.loglog(f_log, noise_relative_empirical, "--", label="Relative sensor noise␣
↪(empirical fit)")
# plt.loglog(f_log, noise_inertial_empirical, "--", label="Inertial sensor noise␣
↪(empirical fit)")
# plt.loglog(f_log, noise_relative_flat, "-.", label="Relative sensor noise (flat ends)")
# plt.loglog(f_log, noise_inertial_flat, "-.", label="Inertial sensor noise (flat ends)")
plt.loglog(f_log, abs(tf_relative(1j*2*np.pi*f_log)), "--", label="Relative sensor noise␣
↪(magnitude response)")
plt.loglog(f_log, abs(tf_inertial(1j*2*np.pi*f_log)), "--", label="Inertial sensor noise␣
↪(magnitude response)")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[2]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[3]:
```
# Export the transfer functions
tf_relative.save("tf_relative.pkl")
tf_inertial.save("tf_inertial.pkl")
```

In the tutorial, we have fitted empirical models to the noise data as an optional intermediate step. With the empirical models obtained, we can rescale the frequency axis to logspace with fewer data points, which helped speeding up the final modeling. It also allows us to flatten the spectrums at both ends, which is necessary for the H-infinity method. In the end, we've obtained 2 `kontrol.TransferFunction` objects which have magnitude responses matching the relative and inertial sensor noise spectrums. We've also exported those transfer function models for future usages.

Tips:

- `kontrol.curvefit.spectrum_fit()` requires user to specify the number of zeros and poles, or the order of the transfer function. To obtain a reasonable number, a general rule of thumb is to run it multiple times with increasing order until there's pole-zero cancellation or when excess poles and zeros leak out of the frequency band of interest.

Complementary filter synthesis

With the transfer function models of the noise spectrums obtained, we can finally create complementary filters using the `kontrol.ComplementaryFilter` class. The transfer function models we obtained are specified as `kontrol.ComplementaryFilter.noise1` and `kontrol.ComplementaryFilter.noise2` attributes. And, to make the optimization meaningful, we need to specify the inverse of the target attenuation of the noises as `kontrol.ComplementaryFilter.weight1` and `kontrol.ComplementaryFilter.weight2` attributes. Conveniently, we don't need to do extra work because the noise models themselves can be used. Click the link below to see what it means exactly.

Complementary Filter Synthesis

[1]:
```
import pickle

import control
import numpy as np
import matplotlib.pyplot as plt

import kontrol

# Loading the transfer function models of the sensor noises
tf_relative = kontrol.load_transfer_function("tf_relative.pkl")
tf_inertial = kontrol.load_transfer_function("tf_inertial.pkl")

# Create a ComplementaryFilter instance so we can use the synthesis methods.
comp = kontrol.ComplementaryFilter()

# Specifying the noise models.
comp.noise2=tf_relative
comp.noise1=tf_inertial

# Weights are the inverse of the target attenuation.
# In this case, we want to suppress the sensor noise to another one, which ever is lower.
comp.weight2=1/tf_inertial
comp.weight1=1/tf_relative

# Synthesis.
# Note. Sometimes this produces filters that make no sense, or kind of suboptimal.
# If that happens, try interchanging the number 1 and 2 from the above noise and weight␣
↪specifications.
# (This is one of the cases so we happened to start with 2, instead of 1.)
h2, h1 = comp.hinfsynthesis()
```

[2]:
```
# Sometimes the filters may contain meaningless coefficients that corresponds to
# zeros or poles at extremely high frequencies.
# This will become a problem during implementation so it's best to get rid of them now.
# In this case, we have one such coefficient in h1. The first numerator coefficient is␣
↪an outlier.
# print(h1)  # < uncomment this line to see it.

# To get rid of it, we define a new filter using the numerator and denominator of h1,␣
```

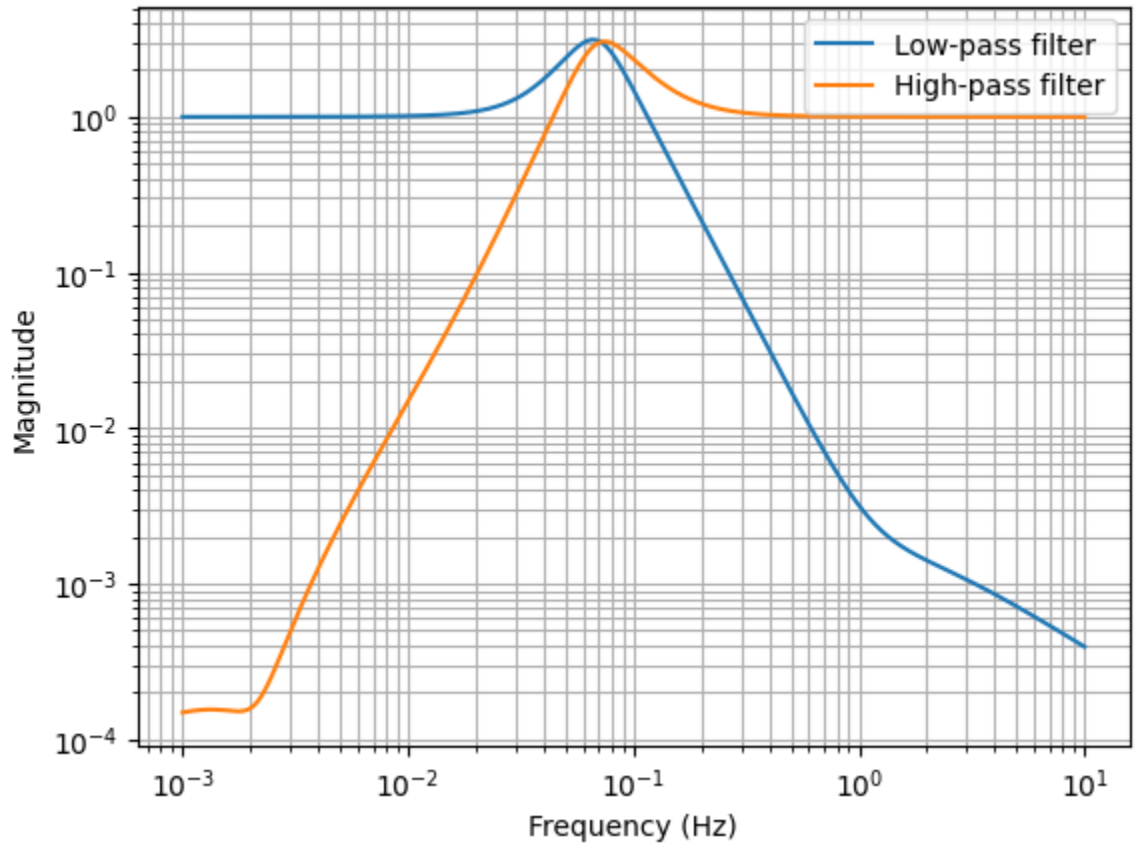
(continues on next page)

(continued from previous page)

```
↪without that coefficient.
h1 = control.tf(h1.num[0][0][1:], h1.den[0][0])
h2 = 1 - h1
# ^Doing the above should not change the result or performance.
# (Just make sure this block is ran only once.)
# Optionally, we can plot the new and old filters to see if they match.
```

[5]:
```
# Inspect the filters
f = np.logspace(-3, 1, 1024)
plt.loglog(f, abs(h1(1j*2*np.pi*f)), label="Low-pass filter")
plt.loglog(f, abs(h2(1j*2*np.pi*f)), label="High-pass filter")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
plt.xlabel("Frequency (Hz)")
```

[5]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



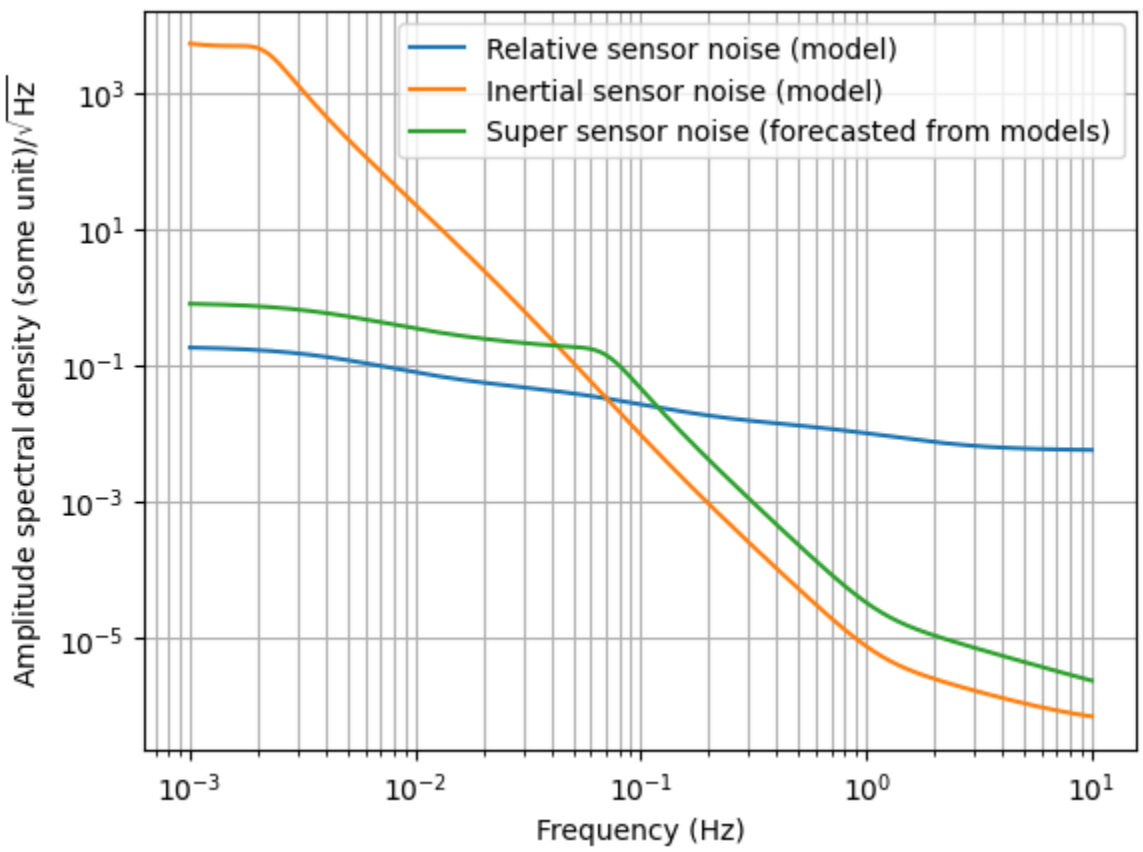
[6]:
```
# Now let's forecast the noise of the super sensor.
plt.loglog(f, abs(tf_relative(1j*2*np.pi*f)), label="Relative sensor noise (model)")
plt.loglog(f, abs(tf_inertial(1j*2*np.pi*f)), label="Inertial sensor noise (model)")
plt.loglog(f, comp.noise_super(f), label="Super sensor noise (forecasted from models)")
plt.legend(loc=0)
plt.grid(which="both")
```

(continues on next page)

(continued from previous page)

```
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[6]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



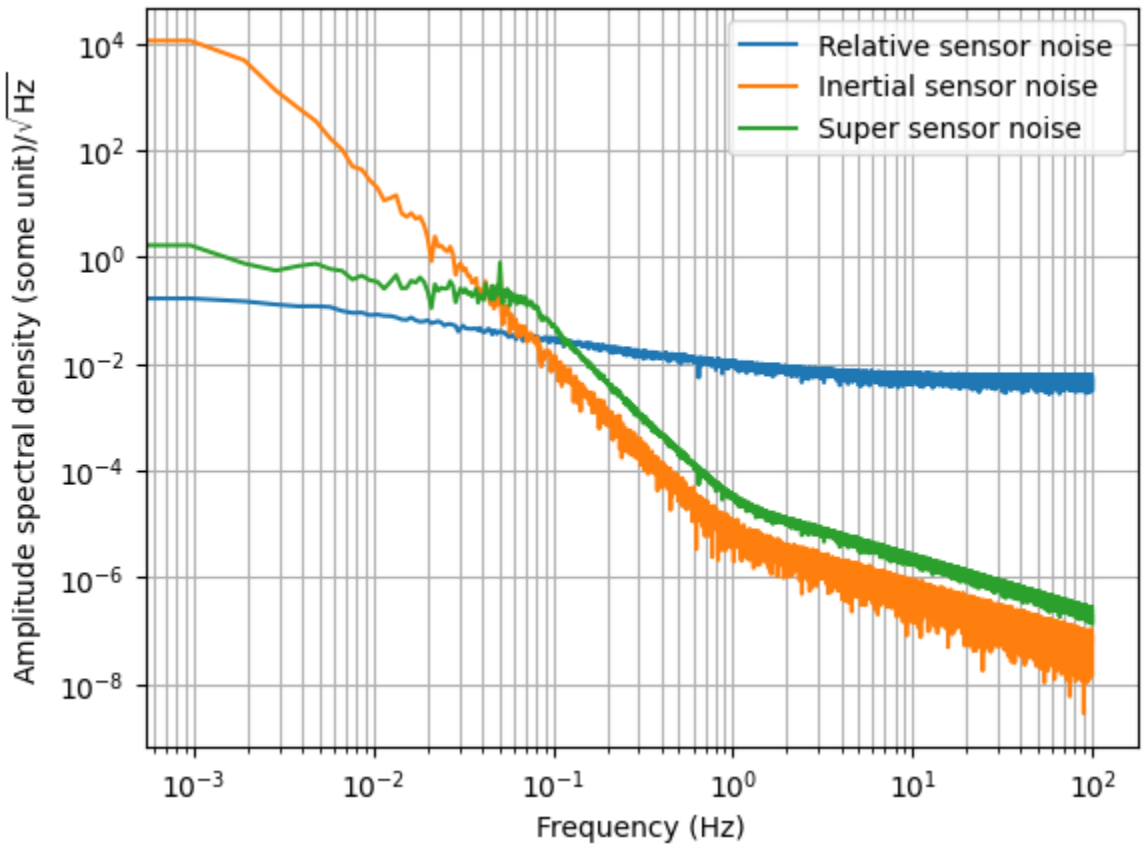
[7]:
```
# As you can see, with these weight specifications, the complementary filters
# are perfectly shaped in such a way the super sensor noise is suppressed everywhere
# (except at the blending frequency).
# The super sensor noise has a multiplificative factor from the lower boundary
# and it turned out to be the H-infinity norm, which is what the H-infinity synthesis␣
↪minimizes.
# Because of this, the super sensor noise looks equidistant from the lower boundary
# at all frequencies in a loglog plot.
# If this is not what we want, we can always specify custom specifications using the␣
↪weights.
# And in that case, the H-infinity norm represents the deviation from the specifications.

# Now, we must note that the filters are only optimal with respect to the cost function,
# which is derived from the flattened noise spectrum.
# Let's import the realistic sensor noises that we measured
# and see how the complementary filters realistically perform.

# Import sensor noise data
with open("noise_spectrum_frequency.pkl", "rb") as fh:
```

(continues on next page)

(continued from previous page)

```
    f_ = pickle.load(fh)
with open("noise_spectrum_relative.pkl", "rb") as fh:
    noise_relative = pickle.load(fh)
with open("noise_spectrum_inertial.pkl", "rb") as fh:
    noise_inertial = pickle.load(fh)


# Inspect. Actually looks fine, with one caveat.
plt.loglog(f_, noise_relative, label="Relative sensor noise")
plt.loglog(f_, noise_inertial, label="Inertial sensor noise")
plt.loglog(f_, comp.noise_super(f_, noise1=noise_inertial, noise2=noise_relative), label=
↪"Super sensor noise")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[7]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



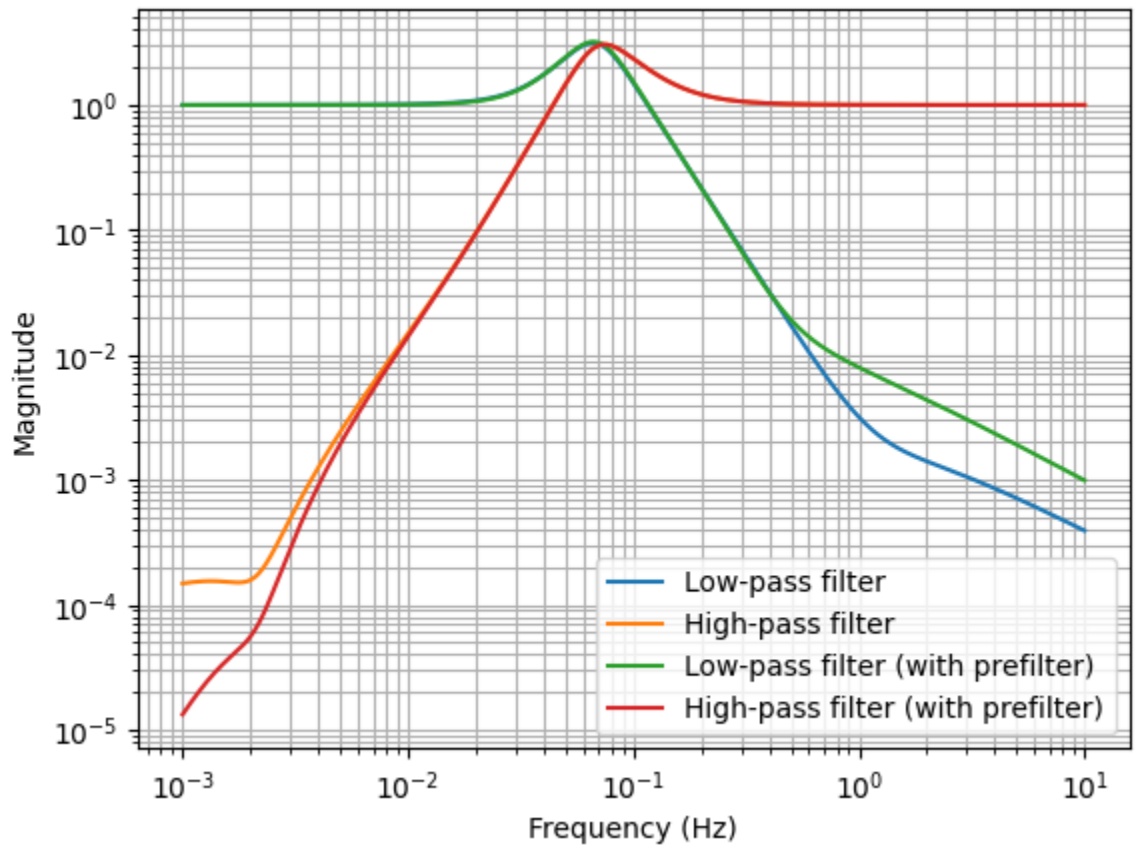
[8]:
```
# Note how the how-pass filter becomes flattened below 2e-3 Hz.
# That's because we had to flatten the noise spectrum at both ends in order to use the H-
↪infinity method.
# In practice, we want to keep the roll-off until 0 Hz to avoid integration error.
# There are many ways to achieve this.
```

(continues on next page)

(continued from previous page)

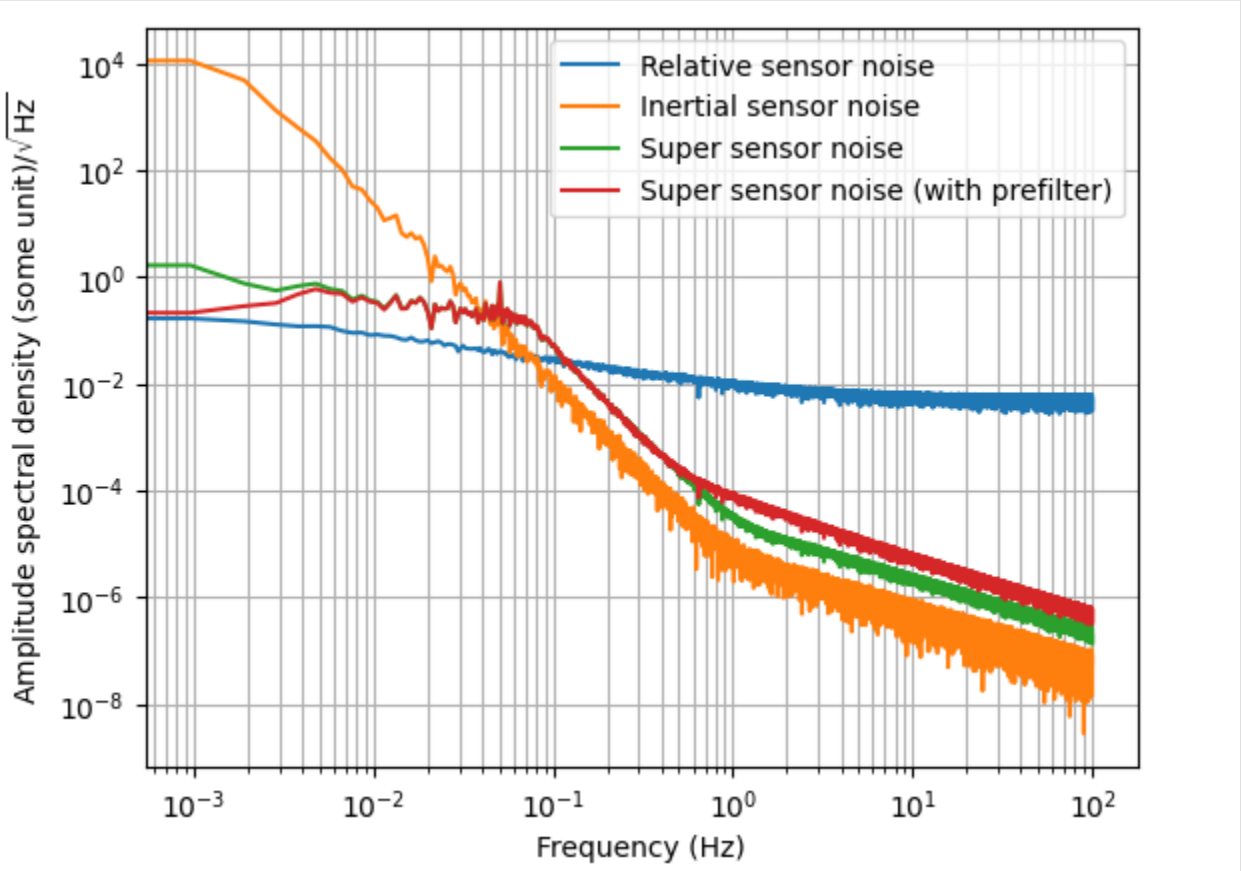
```
# Here we'll use an extremely simple method.

# Add a 3rd-order high-pass prefilter to the high-pass filter
# Note. We're matching the order of the roll-off to the complementary high-pass.
# It could be 3/4.
s = control.tf("s")
wc = 2*np.pi*2e-3  # cut-off frequency
prefilter = (s / (s+wc))**3

# Redefine the complementary high-pass filter and  low-pass filter.
h2_prefilt = h2 * prefilter
h1_prefilt = 1 - h2_prefilt  # Complementary condition.

# Inspect
plt.loglog(f, abs(h1(1j*2*np.pi*f)), label="Low-pass filter")
plt.loglog(f, abs(h2(1j*2*np.pi*f)), label="High-pass filter")
plt.loglog(f, abs(h1_prefilt(1j*2*np.pi*f)), label="Low-pass filter (with prefilter)")
plt.loglog(f, abs(h2_prefilt(1j*2*np.pi*f)), label="High-pass filter (with prefilter)")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
plt.xlabel("Frequency (Hz)")
```

[8]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[9]:
```
# New forecast.
plt.loglog(f_, noise_relative, label="Relative sensor noise")
plt.loglog(f_, noise_inertial, label="Inertial sensor noise")
plt.loglog(f_, comp.noise_super(f_, noise1=noise_inertial, noise2=noise_relative), label=
↪"Super sensor noise")
plt.loglog(f_, comp.noise_super(f_, noise1=noise_inertial, noise2=noise_relative,␣
↪filter1=h2_prefilt, filter2=h1_prefilt), label="Super sensor noise (with prefilter)")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[9]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[10]:
```
# As can be seen, the addition of the high-pass prefilter provides further suppression␣
↪at low-frequency
# and this should prevent integration error from accumulating.
# However, it also rises the sensor noise at above 1 Hz and this is the trade-off that␣
↪we have to make.
# We can lower that by tweaking the order and cut-off frequency of the prefilter.
# But we're not geting into that and let's assume that the super sensor noise level is␣
↪acceptable.

# Again, to export the filters to Foton string so we can install them, we can use␣
↪kontrol.TransferFunction.
```

(continues on next page)

(continued from previous page)

```
print("Low-pass with prefilt\n", kontrol.TransferFunction(h1_prefilt).foton(root_
↪location="n"), "\n")
print("High-pass\n", kontrol.TransferFunction(h2_prefilt).foton(root_location="n"), "\n")
print("High-pass Prefilter\n", kontrol.TransferFunction(prefilter).foton(root_location="n
↪"))
# ^Printing the prefilter separately because we might have this at a different filter␣
↪bank,
# such as the input filters for the inertial sensors.
```

```
Low-pass with prefilt
 zpk([0.00185588;0.002061+i*0.000170;0.002061+i*-0.000170;0.00376473;0.00381207;0.
↪023713+i*0.025604;0.023713+i*-0.025604;0.038824;0.0561513;0.117467;0.195753;0.
↪304844+i*0.410168;0.304844+i*-0.410168;1.70477;2.06538;10.6556],[0.00199992;0.
↪002000+i*0.000000;0.002000+i*-0.000000;0.00376472;0.00376474;0.0511919;0.055021+i*-0.
↪011625;0.055021+i*0.011625;0.051606+i*-0.040288;0.051606+i*0.040288;0.022296+i*-0.
↪065921;0.022296+i*0.065921;0.195169;0.196346;2.06538;2.06539;11.3038],1,"n")

High-pass
 zpk([-0;-0;-0;0.000592232;0.000587+i*-0.002051;0.000587+i*0.002051;0.00376473;0.014638;
↪0.0332445;0.055905;0.122886+i*-0.075068;0.122886+i*0.075068;0.154938;0.195764;2.06394;
↪2.06538;11.3032],[0.00199988;0.002000+i*0.000000;0.002000+i*-0.000000;0.00376472;0.
↪00376474;0.0511919;0.055021+i*-0.011625;0.055021+i*0.011625;0.051606+i*-0.040288;0.
↪051606+i*0.040288;0.022296+i*-0.065921;0.022296+i*0.065921;0.195169;0.196346;2.06538;2.
↪06539;11.3038],11883.2,"n")

High-pass Prefilter
 zpk([-0;-0;-0],[0.00199997;0.002000+i*0.000000;0.002000+i*-0.000000],1.25e+08,"n")
```

And, we've obtained 2 complementary filters which solve the sensor fusion problem in such a way the super sensor noise are minimum at all frequencies with respect to the lower boundary. We've also exported the Foton string using the `kontrol.TransferFunction.foton()` method so we can implement the filters into the digital control system. We've also encountered some practical issues that might occur and they are listed below as tips.

Tips

- Complementary filters generated by the `kontrol.ComplementaryFilter.hinfsynthesis()` method may contain meaningless coefficients. It's important to get rid of them as they will show as astronomically high-frequecy zeros/poles, that cannot be implemented in a digital system.
- Because of the way H-infinty method (or rather, most methods) works, the noise models we specified have flat ends. This means the filters do no have the necessary features to properly roll-off the noises beyond the frequency band of interest. We can add prefilters to fix the problem but this may require some tweaking.
- Sometimes the synthesis method can produce filters that make no sense. Interchanging the relative and inertial sensor noises (and weights) may solve the issue.

H-infinity sensor correction

Relative sensors are seismic noise-coupled so they inject seismic noise to the isolation platform under feedback control. We have a seismometer on the ground that measures the ground motion close to the isolation platform and we can use that to reduce the seismic noise coupling in the relative sensors. This is simply done by subtracting the seismometer readout from the relative readout, thereby "correcting" the relative sensors. However, doing so injects seismometer noise to the relative readout. Therefore, we have to design a "sensor correction filter" to filter excessive noise while retaining seismic signal for sensor correction.

In *H-infinity sensor fusion*, we have used the `kontrol.ComplementaryFilter` class to optimize complementary filters and we've successfully obtain a pair of complementary filters that can be implemented. It turns out that the sensor correction problem can be solved using the very same class and methods. Instead of complementary low-pass and high-pass filters, we'd be obtaining a seismic transmissivity and a sensor correction filter, which are also complementary.

The required procedure to optimize a sensor correction filter is very similar to that already presented in section *H-infinity sensor fusion* so we will not repeat what's already demonstrated. Instead, we assume that we have already obtained

1. The transfer function model with magnitude response matching the noise spectrum of the relative sensor.
2. The same for the seismometer.

We will also need the seismic noise model but it turned out to be more involved so we will demonstrate what needs to be done. There's another notible difference between the sensor correction and the sensor fusion problem. The noise performance of the super sensor in a sensor fusion problem is completely dependent on the filtered sensor noises. In sensor correction, the corrected relative sensor is dependent on the filtered seismometer noise and filtered the seismic noise but is also dependent on the unfiltered intrinsic relative sensor noise. The intrinsic relative sensor noise acts as an ambient noise and we shall see how this can be utilized in the optimization.

Seismic noise spectrum modification and modeling

We read Chapter 8.3.3 in Ref.[?] and realized we need to modify the seismic noise spectrum before modeling it. Otherwise the sensor correction filter might actually amplify the seismometer noise and the seismic noise, instead of attenuating them. Click the link below to see how we can obtain a proper seismic noise model for the optimization purpose.

Seismic Noise Spectrum Modification and Modeling

[1]:
```
import pickle

import control
import numpy as np
import matplotlib.pyplot as plt
import scipy

import kontrol


f = np.linspace(1e-3, 1e2, 1024*512)
def noise_model(f, na, nb, a, b):
    return np.sqrt((na/f**a)**2 + (nb/f**b)**2)
n_geophone = noise_model(f, na=1*10**-5.46, nb=1*10**-5.23, a=3.5, b=1)
```

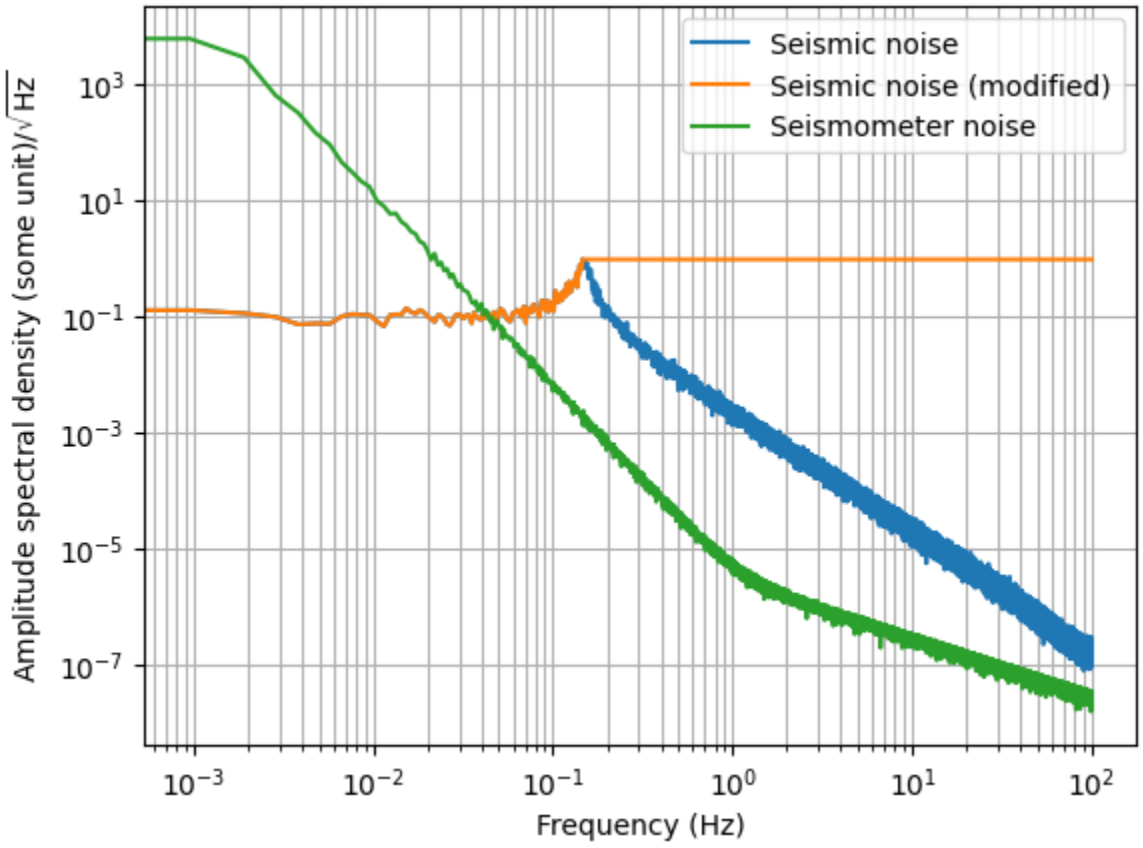
(continues on next page)

(continued from previous page)

```
# Let's assume seismometer is 2 times less noisy than geophone.
n_seismometer = n_geophone / 2

# Generate time series measurements
np.random.seed(1)
t, seismometer_noise = kontrol.spectral.asd2ts(n_seismometer, f=f)

# Ground motion
s = control.tf("s")
wn = 0.15*2*np.pi  # Secondary microseism
q = 10
ground_motion_tf = 0.1 * wn**2 / (s**2+wn/q*s+wn**2)

# Generate time series measurements
fs = 1/(t[1]-t[0])
u = np.random.normal(loc=0, scale=np.sqrt(fs/2), size=len(t))
_, ground_motion = control.forced_response(ground_motion_tf, U=u, T=t)

# Obtain the measured spectral densities.
f_, p_seismic = scipy.signal.welch(ground_motion, fs=fs, nperseg=int(len(t)/5))
_, p_seismometer = scipy.signal.welch(seismometer_noise, fs=fs, nperseg=int(len(t)/5))
```

[2]:
```
seismic_modified = kontrol.spectral.pad_above_maxima(p_seismic**0.5, order=7500)
# order is passed to scipy.signal.argrelmax() which is used to determine the local␣
↪maxima. Keep increasing until it works.

plt.loglog(f_, p_seismic**0.5, label="Seismic noise")
plt.loglog(f_, seismic_modified, label="Seismic noise (modified)")
plt.loglog(f_, p_seismometer**0.5, label="Seismometer noise")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

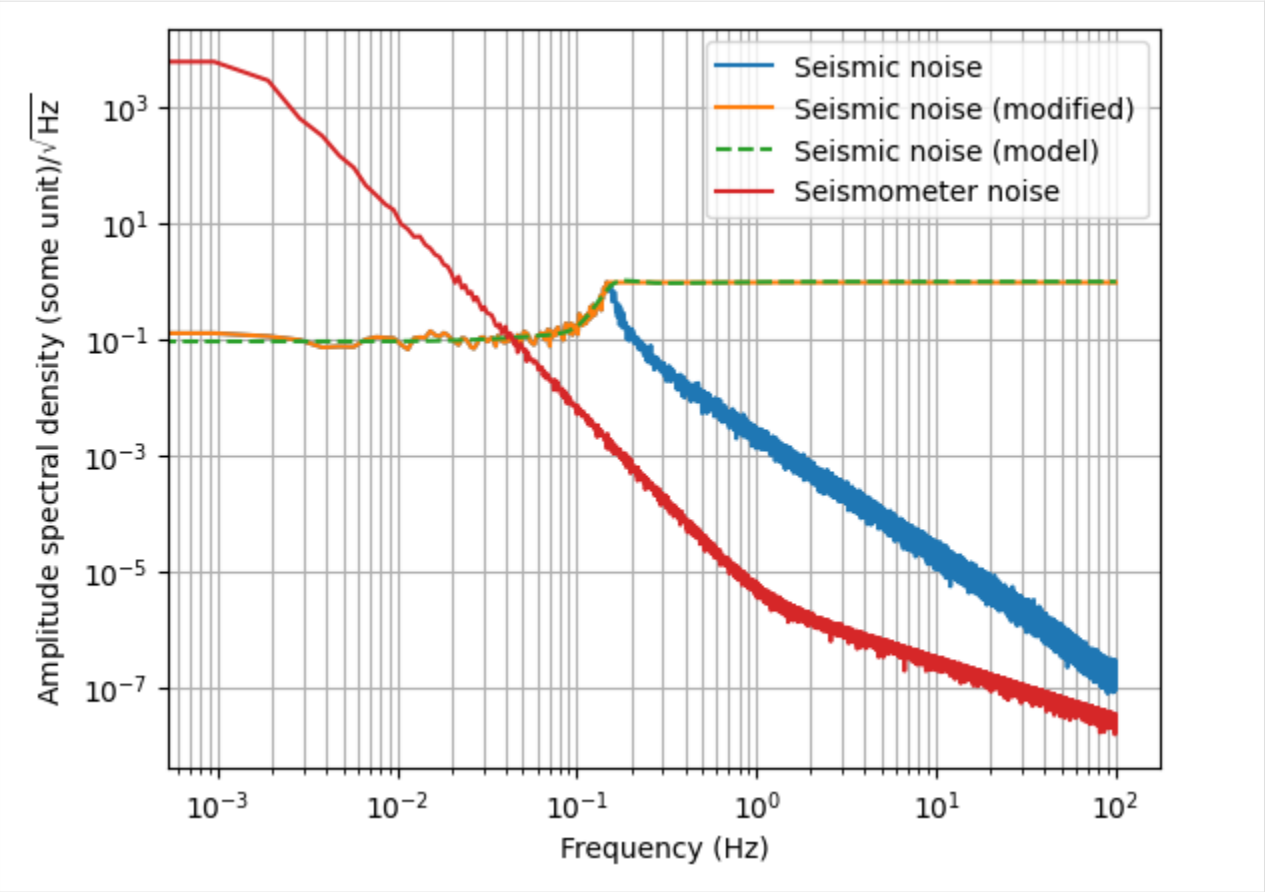
[2]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[3]:
```
# We don't need to fit the whole spectrum because the modified
# spectrum is flat all the way above the microseism frequency.
# Let's get rid of the data above 1 Hz.
# Data to be modelled:
ydata = seismic_modified[(f_>0) & (f_<1)]
xdata = f_[(f_>0) & (f_<1)]
order = 3
tf_seismic = kontrol.curvefit.spectrum_fit(xdata, ydata, nzero=order, npole=order)
```

[4]:
```
plt.loglog(f_, p_seismic**0.5, label="Seismic noise")
plt.loglog(f_, seismic_modified, label="Seismic noise (modified)")
plt.loglog(f_, abs(tf_seismic(1j*2*np.pi*f_)), "--", label="Seismic noise (model)")
plt.loglog(f_, p_seismometer**0.5, label="Seismometer noise")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[4]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[5]:
```
# Export the transfer function model and noise spectrum for future usages.
tf_seismic.save("tf_seismic.pkl")

with open("noise_spectrum_seismic.pkl", "wb") as fh:
    pickle.dump(p_seismic**0.5, fh)
with open("noise_spectrum_seismometer.pkl", "wb") as fh:
    pickle.dump(p_seismometer**0.5, fh)
with open("noise_spectrum_frequency.pkl", "wb") as fh:
    pickle.dump(f_, fh)
```

And we've obtained a seismic noise model that has a flat spectrum above the secondary microseism. The reasons behind this are rather involved and the explanation can be found in the references. But in simple words, the main reasons we had to do this are because

1. Make the sensor correction filter a high-pass filter.
2. So the sensor correction filter won't amplify any seismometer and seismic noise below the relative sensor noise level.

Again, this model is not a representation for the seismic noise. It is created for the sole purpose of H-infinity optimization.

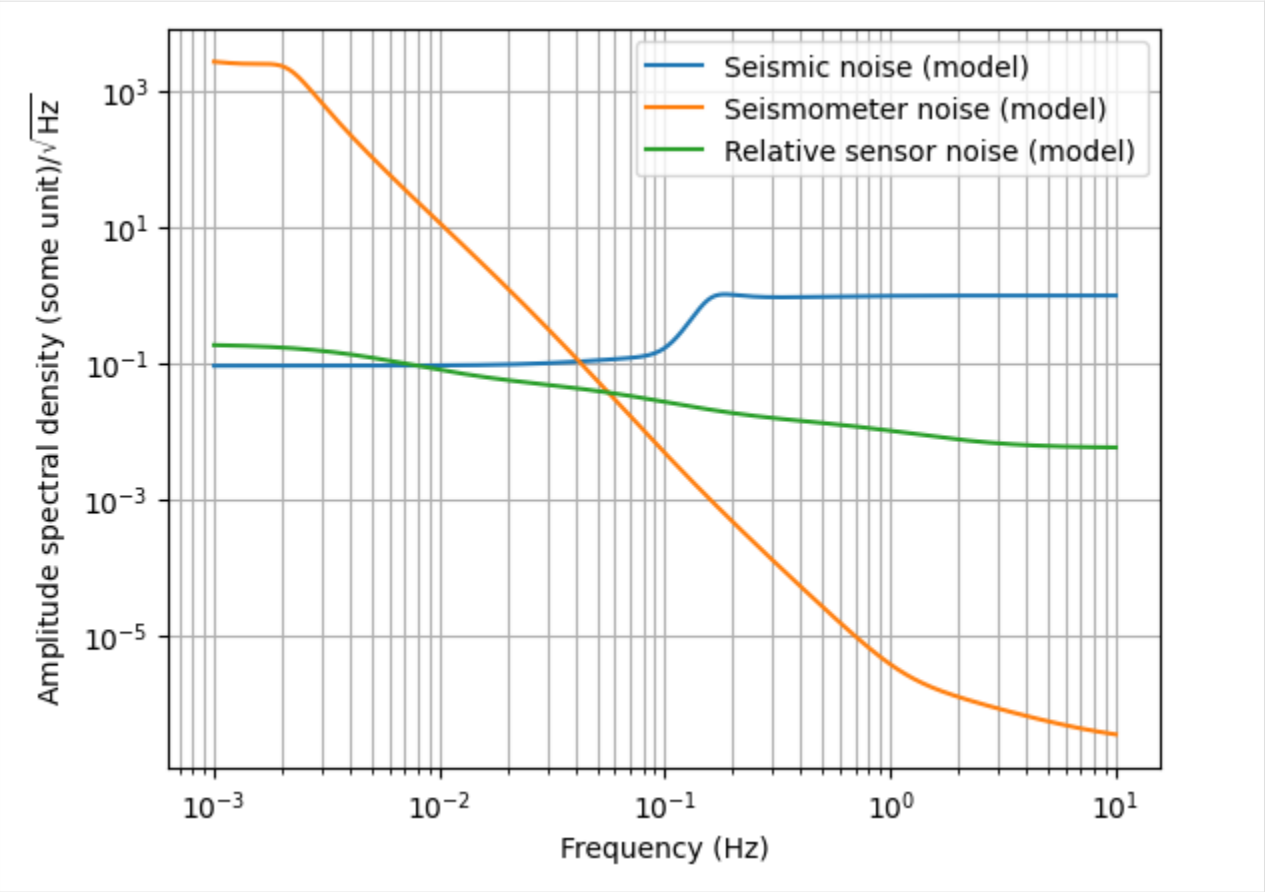
Sensor correction filter synthesis

With all the necessary components obtained, we can use `kontrol.ComplementaryFilter` again to optimize, not complementary filters, but the sensor correction filter. Instead of specifying two sensor noise models to the `ComplementaryFilter` instance, we specify the seismic noise model and the seismometer noise model. The prescense of the relative sensor noise changes how we specify the target attenuation (weights) as the lower boundary of the corrected sensor noise depends on it. Click the link below to see how we cope with that and optimize a sensor correction filter.

Sensor Correction Filter Synthesis

[1]:
```
import pickle

import control
import numpy as np
import matplotlib.pyplot as plt

import kontrol


# Load the transfer function models
tf_seismic = kontrol.load_transfer_function("tf_seismic.pkl")
tf_relative = kontrol.load_transfer_function("tf_relative.pkl")
tf_inertial = kontrol.load_transfer_function("tf_inertial.pkl")

# We've assumed the seismometer noise to be 2 times lower than that of the inertial␣
↪sensors.
tf_seismometer = tf_inertial / 2

# Let's visualize before moving on
f = np.logspace(-3, 1, 1024)

plt.loglog(f, abs(tf_seismic(1j*2*np.pi*f)), label="Seismic noise (model)")
plt.loglog(f, abs(tf_seismometer(1j*2*np.pi*f)), label="Seismometer noise (model)")
plt.loglog(f, abs(tf_relative(1j*2*np.pi*f)), label="Relative sensor noise (model)")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[1]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



Now, let's take a moment to think about how the sensor correction is going to affect the noise of the relative sensor.

Without sensor correction, the effective noise of the relative sensor is approximately the upper bound of the relative sensor (green curve) and the seismic noise (blue curve). The blue curve domainates at high frequencies and that's what we want to get rid of by means of sensor correction.

With sensor correction, we have the ability to suppress the blue curve. However, this introduces seismometer noise (orange curve) to the relative sensor and we want to suppress this as well.

The simultaneous suppression of seismic noise and seismometer noise constitute a sensor fusion problem and we already have the tools to solve it as discussed in last section of the advanced control methods tutorials. However, if we do it that way, the seismic noise (blue) is going to be suppressed to a level close to the seismometer noise (orange) at high frequencies. While the traditional wisdom says we want to suppress seismic as much as possible, this is not necessarily optimal. This is because the relative sensor noise (green) is still in play and that is not being suppressed by anything. Any noise suppressed below that level is not useful as the corrected relative sensor noise is going to be dominated by the relative sensor noise itself anyway. Instead, we should indentify what is true lower boundary of the corrected relative sensor noise and use that as the target attenuation.

In this case, the lower boundary is roughly the relative sensor noise except between ~10^{-2} Hz and ~5×10^{-2} Hz, where the seismic noise and seismometer noise dominates. To do this propoerly, we should model that lower bound using a transfer function like how we modelled other noise spectrums. However, let's just assume that there's not much difference so we can use the relative sensor noise as the lower boundary for simplicity.

[2]:
```
# Synthesis
comp = kontrol.ComplementaryFilter()
comp.noise1 = tf_seismic
```

(continues on next page)

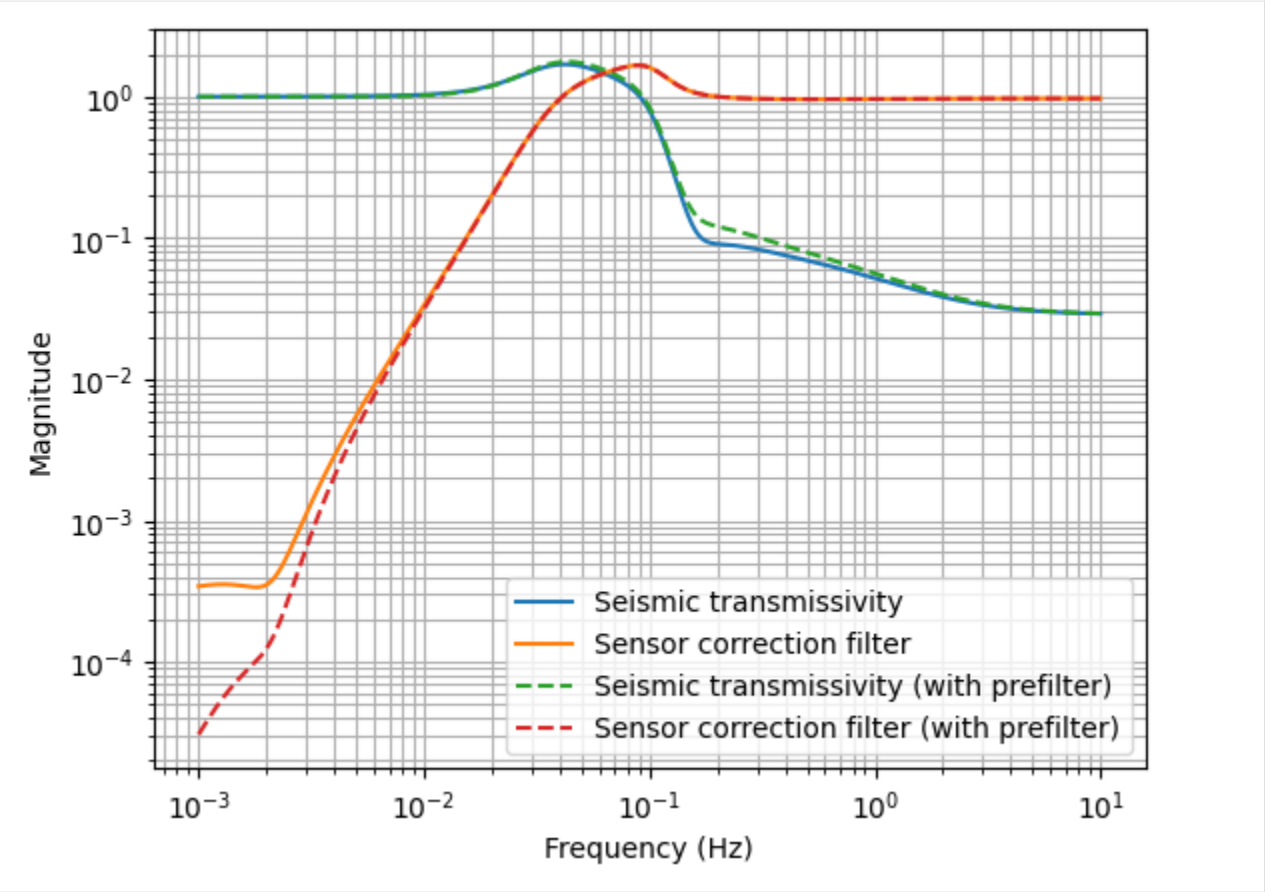
(continued from previous page)

```
comp.noise2 = tf_seismometer
comp.weight1 = 1 / tf_relative  # Using the relative sensor noise as the lower boundary.
comp.weight2 = 1 / tf_relative
h1, h2 = comp.hinfsynthesis()
```

[3]:
```
# Let's not forget about the prefilter.
s = control.tf("s")
wc = 2*np.pi*2e-3  # cut-off frequency
prefilter = (s / (s+wc))**3

# Redefine the sensor correction filter and seismic transmissivity
h2_prefilt = h2 * prefilter
h1_prefilt = 1 - h2_prefilt  # Complementary condition.
```

[4]:
```
# Inspect
plt.loglog(f, abs(h1(1j*2*np.pi*f)), label="Seismic transmissivity")
plt.loglog(f, abs(h2(1j*2*np.pi*f)), label="Sensor correction filter")
plt.loglog(f, abs(h1_prefilt(1j*2*np.pi*f)), "--", label="Seismic transmissivity (with␣
↪prefilter)")
plt.loglog(f, abs(h2_prefilt(1j*2*np.pi*f)), "--", label="Sensor correction filter (with␣
↪prefilter)")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Magnitude")
plt.xlabel("Frequency (Hz)")
```

[4]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[5]:
```
# Let's forecast the noise of the corrected relative sensor.

# Load noise data
with open("noise_spectrum_relative.pkl", "rb") as fh:
    noise_relative = pickle.load(fh)
with open("noise_spectrum_seismometer.pkl", "rb") as fh:
    noise_seismometer = pickle.load(fh)
with open("noise_spectrum_seismic.pkl", "rb") as fh:
    noise_seismic = pickle.load(fh)
with open("noise_spectrum_frequency.pkl", "rb") as fh:
    f_ = pickle.load(fh)
```

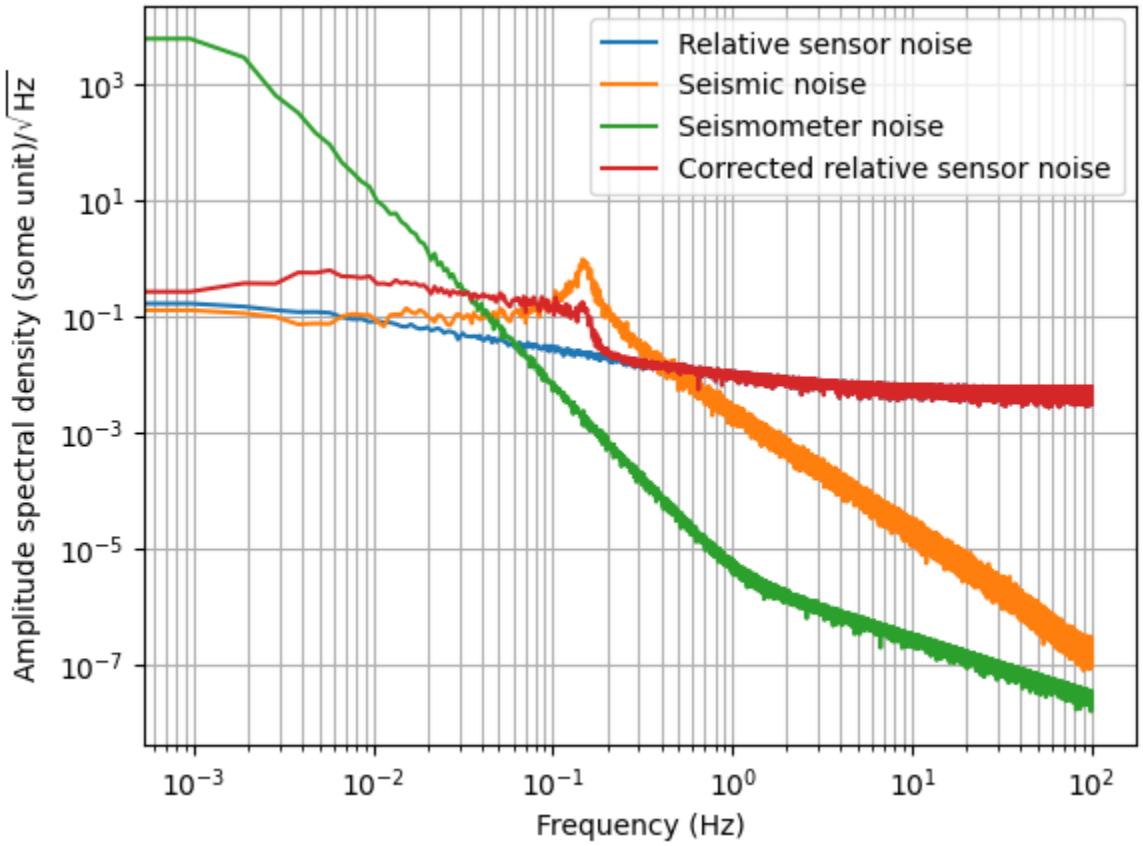
[6]:
```
# The corrected relative sensor noise is composed of two components,
# the intrinsic relative sensor noise and the part related to sensor correction

# The part related to sensor correction:
noise_sensor_correction = comp.noise_super(f_, noise1=noise_seismic, noise2=noise_
↪seismometer, filter1=h1_prefilt, filter2=h2_prefilt)

# The corrected relative sensor noise:
noise_corrected = kontrol.core.math.quad_sum(noise_relative, noise_sensor_correction)
# ^Remember, uncorrelated noises sum in quadrature (power).

# Visusalize
```

(continues on next page)

(continued from previous page)

```
plt.loglog(f_, noise_relative, label="Relative sensor noise")
plt.loglog(f_, noise_seismic, label="Seismic noise")
plt.loglog(f_, noise_seismometer, label="Seismometer noise")
plt.loglog(f_, noise_corrected, label="Corrected relative sensor noise")
plt.legend(loc=0)
plt.grid(which="both")
plt.ylabel("Amplitude spectral density (some unit)/$\sqrt{\mathrm{Hz}}$")
plt.xlabel("Frequency (Hz)")
```

[6]:
```
Text(0.5, 0, 'Frequency (Hz)')
```



[7]:
```
# Now we're seeing some reduction especially around the secondary microseim.
# However, we're also injecting noise at lower frequency.
# To justify this tradeoff, let's compare the RMS of the seismic noise and the corrected␣
↪relative sensor
print("Seismic noise RMS:", kontrol.spectral.asd2rms(noise_seismic, f=f_, return_
↪series=False))
print("Corrected relative sensor RMS:", kontrol.spectral.asd2rms(noise_corrected, f=f_,␣
↪return_series=False))
```

```
Seismic noise RMS: 0.1434520357166474
Corrected relative sensor RMS: 0.10451631749875732
```

[8]:
```
# As you can see, with the sensor correction scheme, the noise RMS is lowered,
```

(continues on next page)

(continued from previous page)

```
# given that the RMS of an uncorrected relative sensor is mostly contributed from the␣
↪seismic noise.
# (we aren't even minimizing the RMS with H2 optimization)
# Of course, this is not always true in reality and this is only an example with mock␣
↪data.

# And, the corrected relative sensor noise is the one that we need to model for sensor␣
↪fusion!

# Export the sensor correction filter.
print("Sensor correction filter:\n", kontrol.TransferFunction(h2_prefilt).foton(root_
↪location="n"))
```

```
Sensor correction filter:
 zpk([-0;-0;-0;0.000634057;0.000567+i*0.002048;0.000567+i*-0.002048;0.0155221;0.0332424;
↪0.102979;0.048242+i*0.106069;0.048242+i*-0.106069;0.814654;2.06538],[0.00199992;0.
↪002000+i*0.000000;0.002000+i*-0.000000;0.00376468;0.022393+i*0.030733;0.022393+i*-0.
↪030733;0.050882+i*0.005582;0.050882+i*-0.005582;0.0563568;0.033088+i*0.096811;0.
↪033088+i*-0.096811;0.837395;2.06538],29011.4,"n")
```

Using the `kontrol.ComplementaryFilter.hinfsynthesis()` method, we were able to obtain a sensor correction filter. We have also computed the expected noise spectrum of the corrected relative sensor. It has suppressed seismic coupling, particularly around the secondary microseism. However, seismometer noise is injected at lower frequencies, which is expected. To justify the tradeoff, we compared the noise RMS of the seismic noise (uncorrected relative sensor) and the corrected relative sensor. And, we found that we've indeed reduced the noise RMS of the relative sensor with the sensor correction scheme and the H-infinity optimal filter.

Sensor fusion and optimal controller (just words)

For here, to finally complete the job, the next step would be to model the noise of the corrected relative sensor and use what we've learnt in *H-infinity sensor fusion* to optimize the sensor fusion of the corrected relative sensor and the inertial sensor. We'll just leave it here so we don't repeat.

The controller that we designed in the *Controller design* section utilizes concepts like critical damping and phase margins to design a controller to achieve position and damping control. However, this is not optimal for active seismic noise isolation. In principle, we want a controller that has higher gain where the seismic disturbnce is high, and lower gain where control noise becomes dominated. This turns out to be also a sensor fusion problem as disturbance and noise couplings are also complementary. The controller can be easily optimized for active isolation using a `kontrol.ComplementaryFilter` instance. The optimized result would be the sensitivity function and the complementary sensitivity function, which can be used to derive the optimal controller. Chapter 8.2.3 and 8.3.6 in Ref.[7] provides examples of how feedback controllers can be optimized for seismic isolation so check it out if you're interested.

1.3.3 General Utilities

Coming soon.

References

1.4 Kontrol API

1.4.1 Subpackages

kontrol.complementary_filter

Primary modules

kontrol.complementary_filter.complementary_filter

Complementary filter class for sythesis

class kontrol.complementary_filter.complementary_filter.**ComplementaryFilter**(*noise1=None*, *noise2=None*, *weight1=None*, *weight2=None*, *filter1=None*, *filter2=None*, *f=None*)

Bases: `object`

Complementary filter synthesis class.

Parameters

- **noise1** (TransferFunction) – Sensor noise 1 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 1.
- **noise2** (TransferFunction) – Sensor noise 2 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 2.
- **weight1** (TransferFunction, *optional*) – Weighting function 1. Frequency dependent specification for noise 1. Defaults None.
- **weight2** (TransferFunction, *optional*) – Weighting function 2. Frequency dependent specification for noise 2. Defaults None.
- **filter1** (TransferFunction, *optional*) – The complementary filter for noise 1. Defaults None.
- **filter2** (TransferFunction, *optional*) – The complementary filter for noise 2. Defaults None.
- **f** (*array, optional*) – The frequency axis in Hz for evaluating the super sensor noise. Defaults None.

h2synthesis()

Complementary filter synthesis that minimizes the $\mathcal{H}_2$-norm of the super sensor noise. The noise must be specified before using this method.

hinfsynthesis()

Complementary filter synthesis that minimizes the $\mathcal{H}_\infty$-norm of the super sensor noise. The noise must be specified before using this method.

Notes

This is a utility class for complementary filter synthesis, sensor noise estimation and analysis. To use synthesis methods, specify `noise1` and `noise2` as the transfer function models for the sensor noises, and specify `weight1` and `weight2` as the frequency-dependent specifications for the two sensor noises.

References

property f

The frequency axis in Hz.

property filter1

First complementary filter.

property filter2

Second complementary filter.

h2synthesis(*clean_filter=True*, ***kwargs*)

Synthesize complementary filters using H2 synthesis.

Returns

- **filter1** (*TransferFunction*) – The complementary filter filtering noise 1.
- **filter2** (*TransferFunction*) – The complementary filter filtering noise 2.
- **clean_filter** (*boolean, optional*) – Remove small outlier coefficients from filters. Defaults True.
- ***kwargs* – Keyword arguments passed to kontrol.TransferFunction.clean()

hinfsynthesis(*clean_filter=True*, ***kwargs*)

Synthesize complementary filters using H-inifinity synthesis.

Returns

- **filter1** (*TransferFunction*) – The complementary filter filtering noise 1.
- **filter2** (*TransferFunction*) – The complementary filter filtering noise 2.
- **clean_filter** (*boolean, optional*) – Remove small outlier coefficients from filters. Defaults True.
- ***kwargs* – Keyword arguments passed to kontrol.TransferFunction.clean()

property noise1

Transfer function model of sensor noise 1.

property noise2

Transfer function model of sensor noise 2.

noise_super(*f=None*, *noise1=None*, *noise2=None*, *filter1=None*, *filter2=None*)

Compute and return predicted the ASD of the super sensor noise

Parameter

f
[array, optional] The frequency axis in Hz. Use self.f if None. Defaults None.

noise1
[array, optional] The amplitude spectral density of noise 1. Use self.noise1 and self.f to estimate if None. Defaults None.

noise2
[array, optional] The amplitude spectral density of noise 2. Use self.noise1 and self.f to estimate if None. Defaults None.

filter1
[TransferFunction, optional] The complementary filter for filtering noise1 Use self.filter1 if not specified. Defaults None

filter2
[TransferFunction, optional] The complementary filter for filtering noise1 Use self.filter2 if not specified. Defaults None

returns
The amplitude spectral density of the super sensor noise.

rtype
array

Secondary modules

kontrol.complementary_filter.predefined

Some predefined complementary filters

kontrol.complementary_filter.predefined.**generalized_sekiguchi**(*fb*, *order_low_pass*, *order_high_pass*)

Generalized Sekiguchi Filter

Parameters

- **fb** (*float*) – Blending frequency in Hz.
- **order_low_pass** (*int*) – Order of roll-off of the low-pass filter
- **order_high_pass** (*int*) – Order of roll-off of the high-pass filter

Returns

- **lpf** (*kontrol.TransferFunction*) – The low-pass filter
- **hpf** (*kontrol.TransferFunction*) – The high-pass filter

kontrol.complementary_filter.predefined.**lucia**(*coefs*)

Lucia Trozzo's complementary filter

Parameters
coefs (*list of float or numpy.ndarray of floats*) – Takes 7 parameters, in specification order: p_1, p_2, z_1, w_1, q_1, w_2, q_2 . See notes for the meaning of the parameters.

Returns

- **lpf** (*control.xferfcn.TransferFunction*) – Complementary low-pass filter
- **hpf** (*control.xferfcn.Transferfunction*) – Complementary high-pass filter

Notes

The modified Sekiguchi complemetary filter is structured as:

$$H = K\frac{s^3(s+z_1)(s^2+w_1/q_1\,s+w_1^2)(s^2+w_2/q_2\,s+w_2^2)}{(s+p_1)^5(s+p_2)^3}$$

where K is a constant normalizing the filter at high frequency.

kontrol.complementary_filter.predefined.**modified_sekiguchi**(*coefs*)

Modified Sekiguchi Filter with guaranteed 4th-order high-pass.

Parameters

coefs (*list of (int or float) or numpy.ndarray*) – 4 coefficients defining the modified Sekiguchi filter.

Returns

- **lpf** (*control.xferfcn.TransferFunction*) – Complementary low-pass filter
- **hpf** (*control.xferfcn.Transferfunction*) – Complementary high-pass filter

Notes

The modified Sekiguchi complemetary filter is structured as:

$$H = \frac{s^7 + 7a_1s^6 + 21a_2^2s^5 + 35a_3^3s^4}{(s+a_4)^7}$$

where a_1, a_2, a_3, a_4 are some parameters that defines the filter.

kontrol.complementary_filter.predefined.**sekiguchi**(*coefs*)

4th-order complementary filters specified by the blending frequencies.

Parameters

coefs (*float*) – Blending frequency of the filter in [rad/s]

Returns

- **lpf** (*control.xferfcn.TransferFunction*) – Complementary low pass-filter
- **hpf** (*control.xferfcn.Transferfunction*) – Complementary high pass-filter

Notes

4th-order complemetary filter used in Sekiguchi's thesis [1]_ whose high-pass is structured as:

$$H = \frac{s^7 + 7w_bs^6 + 21w_b^2s^5 + 35w_b^3s^4}{(s+w_b)^7}$$

where w_b is the blending frequency in [rad/s].

References

kontrol.complementary_filter.synthesis

Filter synthesis functions.

kontrol.complementary_filter.synthesis.**generalized_plant**(*noise1*, *noise2*, *weight1*, *weight2*)

Return the generalized plant of a 2 complementary filter system

Parameters

- **noise1** (TransferFunction) – Sensor noise 1 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 1.
- **noise2** (TransferFunction) – Sensor noise 2 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 2.
- **weight1** (TransferFunction) – Weighting function 1. Frequency dependent specification for noise 1.
- **weight2** (TransferFunction) – Weighting function 2. Frequency dependent specification for noise 2.

Returns

The plant.

Return type

control.xferfcn.TransferFunction

kontrol.complementary_filter.synthesis.**h2complementary**(*noise1*, *noise2*, *weight1=None*, *weight2=None*)

H2 optimal complementary filter synthesis

Parameters

- **noise1** (TransferFunction) – Sensor noise 1 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 1.
- **noise2** (TransferFunction) – Sensor noise 2 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 2.
- **weight1** (TransferFunction, *optional*) – Weighting function 1. Frequency dependent specification for noise 1. Defaults None.
- **weight2** (TransferFunction, *optional*) – Weighting function 2. Frequency dependent specification for noise 2.

Returns

- **filter1** (*TransferFunction*) – The complementary filter filtering noise1.
- **filter2** (*TransferFunction*) – The complementary filter filtering noise2.

Notes

This function ultilizes control.robust.h2syn() which depends on the slycot module. If you are using under a conda virtual environment, the slycot module can be installed easily from conda-forge. Using pip to install slycot is a bit more involved (I have yet to suceed installing slycot in my Windows machine). Please refer to the python-control package for further instructions.

It is possible that h2syn yields no solution for some tricky noise profiles . Try adjusting the noise profiles at some irrelevant frequencies.

Thomas Dehaeze [1]_ had the idea first so credits goes to him. (Properly cite when the paper is published.)

References

kontrol.complementary_filter.synthesis.**hinfcomplementary**(*noise1*, *noise2*, *weight1=None*, *weight2=None*)

H-infinity optimal complementary filter synthesis

Parameters

- **noise1** (TransferFunction) – Sensor noise 1 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 1.
- **noise2** (TransferFunction) – Sensor noise 2 transfer function model. A transfer function that has magnitude response matching the amplitude spectral density of noise 2.
- **weight1** (TransferFunction, *optional*) – Weighting function 1. Frequency dependent specification for noise 1. Defaults None.
- **weight2** (TransferFunction, *optional*) – Weighting function 2. Frequency dependent specification for noise 2.

Returns

- **filter1** (*TransferFunction*) – The complementary filter filtering noise1.
- **filter2** (*TransferFunction*) – The complementary filter filtering noise2.

Notes

This function ultilizes control.robust.hinfsyn() which depends on the slycot module. If you are using under a conda virtual environment, the slycot module can be installed easily from conda-forge. Using pip to install slycot is a bit more involved (I have yet to suceed installing slycot in my Windows machine). Please refer to the python-control package for further instructions.

Thomas Dehaeze [1]_ had the idea first so credits goes to him. (Properly cite when the paper is published.)

References

kontrol.core

kontrol.core.controlutils module

Utility function related to python-control library.

kontrol.core.controlutils.**check_tf_equal**(*tf1*, *tf2*, *allclose_kwargs={}*, *minreal=True*)

Check if two transfer functions are approximatedly equal by np.allclose.

Parameters

- **tf1** (*control.xferfcn.TransferFunction*) – Transfer function 1.
- **tf2** (*control.xferfcn.TrasnferFunction*) – Transfer function 2.
- **allclose_kwargs** (*dict, optional*) – Keyword arguments passed to np.allclose(), which is used to compare the list of poles and zeros and dcgain. Defaults {}.
- **minreal** (*boolean*) – Use control.minreal to remove canceling zeros and poles before comparison.

Return type
boolean

kontrol.core.controlutils.**clean_tf**(*tf*, *tol_order=5*, *small_number=1e-25*)

Remove numerator/denominator coefficients that are small outliers

Parameters

- **tf** (TransferFunction) – The transfer function to be cleaned
- **tol_order** (*float, optional*) – If the coefficient is tol_order order smaller than the rest of the coefficients, then this coefficient is an outlier. Defaults 5.
- **small_number** (*float, optional*) – A small number to be added to the log10 in case 0 is encountered.

Returns
tf_cleaned – The cleaned transfer function.

Return type
TransferFunction

kontrol.core.controlutils.**clean_tf2**(*tf*, *tol_order=5*, *small_number=1e-25*)

Remove zeros/poles that are outliers.

Parameters

- **tf** (TransferFunction) – The transfer function to be cleaned.
- **tol_order** (*float, optional*) – If the frequency of the zero/pole is ``tol_order" order away from the mean order, then this zero/pole is an outlier. Defaults 5.
- **small_number** (*float, optional*) – A small number to be added to the log10() in case 0 is encountered

Returns
tf_cleaned – The cleaned transfer function.

Return type
TransferFunction

kontrol.core.controlutils.**clean_tf3**(*tf*, *tol_order=5*, *small_number=1e-25*)

Remove first coefficient if it is much smaller than the second

Parameters

- **tf** (TransferFunction) – Transfer function to be cleaned

- **tol_order** (*float, optional*) – First coefficient is removed if it is ``tol_order'' order smaller than the second coefficient. Remove only if there exists such coefficient in both numerator and denominator.
- **small_number** (*float, optional*) – A small number to be added to the log10() in case 0 is encountered.

Returns
tf_cleaned – The cleaned TransferFunction

Return type
TransferFunction

kontrol.core.controlutils.**complex2wq**(*complex_frequency*)

Convert a complex frequency to frequency and Q-factor.

Parameters
complex_frequency (*complex*) – Complex frequency in rad/s.

Returns
- **wn** (*float*) – The frequency in rad/s
- **q** (*float*) – The Q factor.

kontrol.core.controlutils.**convert_unstable_tf**(*control_tf*)

Convert transfer function with unstable zeros and poles to tf without.

Parameters
control_tf (*control.xferfcn.TransferFunction*) – The transfer function to be converted.

Returns
tf_new – control_tf with unstable zeros and poles flipped.

Return type
control.xferfcn.TransferFunction

Note: Warning can occur when the order gets high, e.g. 100. This happens due to numerical accuracy, i.e. coefficients get astronomically high when order gets high.

---

kontrol.core.controlutils.**generic_tf**(*zeros=None*, *poles=None*, *zeros_wn=None*, *zeros_q=None*, *poles_wn=None*, *poles_q=None*, *dcgain=1.0*, *unit='f'*)

Construct a generic transfer function object.

Parameters
- **zeros** (*array, optional*) – List of zeros in frequencies. Defaults [].
- **poles** (*array, optional*) – List of poles. Defaults [].
- **zeros_wn** (*array, optional*) – List of natural frequencies of numerator second-order sections.
- **zeros_q** (*array, optional.*) – List of Q-value of numerator second-order sections.
- **poles_wn** (*array, optional*) – List of natural frequencies of denominator second-order sections.
- **poles_q** (*array, optional.*) – List of Q-value of denominator second-order sections.
- **dcgain** (*float, optional*) – The DC gain of the transfer function. Defaults 1.

- **unit** (*str, optional*) – The unit of the zeros, poles and the natural frequencies. Choose from ["f", "s", "Hz", "omega"]. Defaults "f".

Returns
tf – The transfer function.

Return type
control.xferfcn.TransferFunction

Note: No negative sign is needed for negative zeros and poles. For instance, *generic_tf(poles=[1], unit="s")* means $\frac{1}{s+1}$.

---

kontrol.core.controlutils.**outlier_exists**(*tf*, *f*, *unit='f'*)

Checks for zeros and poles outside the frequency range.

Parameters

- **tf** (*control.xferfcn.TransferFunction*) – The transfer function.
- **f** (*array*) – The frequency axis of interest.
- **unit** (*str, optional*) – The unit of the zeros, poles and the natural frequencies. Choose from ["f", "s", "Hz", "omega"]. Defaults "f".

Returns
If there's any zero or pole outside the frequency range.

Return type
boolean

kontrol.core.controlutils.**outliers**(*tf*, *f*, *unit='f'*)

Returns a list of zeros and poles outside the frequency range.

Parameters

- **tf** (*control.xferfcn.TransferFunction*) – The transfer function.
- **f** (*array*) – The frequency axis of interest.
- **unit** (*str, optional*) – The unit of the zeros, poles and the natural frequencies. Choose from ["f", "s", "Hz", "omega"]. Defaults "f".

Returns

- **outlier_zeros** (*array*) – Zeros outside the frequency range.
- **outlier_poles** (*array*) – Poles outside the frequency range.

Note: We use the python-control package convention for the returned zeros and poles, so to preserve the type and format as much as possible for further processes.

---

kontrol.core.controlutils.**sos**(*natural_frequencies=None*, *quality_factors=None*, *gain=1.0*, *unit='f'*)

Second-order sections transfer fucntion.

Parameters

- **natural_freuqnecies** (*array, optional*) – List of natural frequencies. Defaults [].
- **quality_factors** (*list, optional*) – List of quality factors. Defaults [].
- **gain** (*float, optional*) – The gain of the transfer function. Defaults 1.

- **unit** (*str, optional*) – The unit of the zeros, poles and the natural frequencies. Choose from ["f", "s", "Hz", "omega"]. Defaults "f".

Returns
sos – The product of all second-order sections.

Return type
control.xferfcn.TransferFunction

Note: $K \prod \left(s^2 + \omega_n/Q\, s + \omega_n^2\right)$.

---

kontrol.core.controlutils.**tf_order_split**(*tf*, *max_order=20*)

Split TransferFunction objects into multiple ones with fewer order.

Parameters

- **tf** (TransferFunction) – The transfer function
- **max_order** (*int, optional*) – The maxmium order of the split transfer functions. Defaults to 20 (the foton limit).

Returns
The list of splitted transfer functions.

Return type
list of TransferFunction.

kontrol.core.controlutils.**tfmatrix2tf**(*sys*)

Convert a matrix of transfer functions to a MIMO transfer function.

Parameters
sys (*list of (list of control.xferfcn.TransferFunction)*) – The transfer function matrix representing a MIMO system. sys[i][j] is the transfer function from the i+1 input port to the j+1 output port.

Returns
The transfer function of the MIMO system.

Return type
control.xferfcn.TransferFunction

kontrol.core.controlutils.**zpk**(*zeros*, *poles*, *gain*, *unit='f'*, *negate=True*)

Zero-pole-gain definition of transfer function.

Parameters

- **zeros** (*list of floats*) – A list of the location of the zeros
- **poles** (*list of floats*) – A list of the location of the poles
- **gain** (*float*) – The static gain of the transfer function
- **unit** (*str, optional*) – The unit of the zeros, poles and the natural frequencies. Choose from ["f", "s", "Hz", "omega"]. Defaults "f".
- **negate** (*boolean, optional*) – Negate zeros and poles in specification so negative sign is not needed for stable zeros and poles. Default to be True.

Returns
zpk_tf – The zpk defined transfer function

Return type
control.xferfcn.TransferFunction

Notes

Refrain from specifying imaginary zeros and poles. Use kontrol.utils.sos() for second-order sections instead. The zero and poles are negated by default.

kontrol.core.math module

Common math library

kontrol.core.math.**log_mse**(*x1*, *x2*, *weight=None*, *small_multiplier=1e-06*)

Logarithmic mean square error/Mean square log error

Parameters

- **x1** (*array*) – Array
- **x2** (*array*) – Array
- **weight** (*array or None, optional*) – weighting function. If None, defaults to *np.ones_like(x1)*
- **small_multiplier** (*float, optional*) – *x1* will be modified by *x1 `+ `small_multiplier `* `np.min(x2)* if 0 exists in *x1*. Same goes to *x2*. If 0 both exists in *x1* and *x2*, raise.

Returns
Logarithmic mean square error between arrays x1 and x2.

Return type
float

kontrol.core.math.**mse**(*x1*, *x2*, *weight=None*)

Mean square error

Parameters

- **x1** (*array*) – Array
- **x2** (*array*) – Array
- **weight** (*array or None, optional*) – weighting function. If None, defaults to np.ones_like(x1)

Returns
Mean square error between arrays x1 and x2.

Return type
float

kontrol.core.math.**polyval**(*p*, *x*)

kontrol.core.math.**quad_sum**(**spectra*)

Takes any number of same length spectrum and returns the quadrature sum.

Parameters
***spectra** – Variable length argument list of the spectra.

Returns
qs – The quadrature sum of the spectra.

Return type
np.ndarray

kontrol.core.spectral module

Spectral analysis related function library.

kontrol.core.spectral.**asd2rms**(*asd*, *f=None*, *df=1.0*, *return_series=True*)

Calculate root-mean-squared value from amplitude spectral density

Parameters

- **asd** (*array*) – The amplitude spectral density
- **f** (*array, optional*) – The frequency axis. Defaults None.
- **df** (*float, optional*) – The frequency spacing. Defaults 1.
- **return_series** (*bool, optional*) – Returns the RMS as a frequency series, where each value is the integrated RMS from highest frequency. If False, returns a single RMS value for the whole spectrum. Defaults True.

Returns

- *array* – The integrated RMS series, if return_series==True.
- *float* – The integrated RMS value, if return_series==False.

Notes

The integrated RMS series is defined as

$$x_{\mathrm{RMS}}(f) = \int_{\infty}^{f} x(f')^2 \, df' \,,$$

where $x(f)$ is the amplitude spectral density.

When return_series is False, only $x_{\mathrm{RMS}}(0)$ is returned.

kontrol.core.spectral.**asd2ts**(*asd*, *f=None*, *fs=None*, *t=None*, *window=None*, *zero_mean=True*)

Simulate time series from amplitude spectral density

Parameters

- **asd** (*array*) – The amplitude spectral density.
- **f** (*array, optional*) – The frequency axis of the amplitude spectral density. This is used to calculate the sampling frequency. If fs is not specified, f must be specified. Defaults None.
- **fs** (*float, optional*) – The sampling frequency in Hz. If f is specified, the last element of f will be treated as the Nyquist frequency so the double of it would be the sampling frequency. Defaults None.
- **t** (*array, optional*) – The desired time axis for the time series. If t is specified, then the time series will be interpolated using numpy.interp() assuming that the time series is periodic. Default None.

- **window** (*array, optional*) – The FFT window used to compute the amplitude spectral density. Defaults None.
- **zero_mean** (*boolean, optional*) – Returns a time series with zero mean. Defaults True.

Returns

- **t** (*array*) – Time axis.
- **time_series** (*array*) – The time series

Notes

Using a custom time axis is not recommended as interpolation leads to distortion of the signal. To extend the signal in time, it's recommended to repeat the original time series instead.

kontrol.core.spectral.**pad_above_maxima**(*series*, *pad_index=-1*, ***kwargs*)

Pad series above local maxima

Parameters

- **series** (*array*) – The series to be padded.
- **pad_index** (*int, optional*) – The index of the local maxima. Defaults to -1.
- ****kwargs** – Keyword arguments passed to scipy.signal.argrelmin()

Returns
series_padded – The padded series

Return type
array

kontrol.core.spectral.**pad_above_minima**(*series*, *pad_index=-1*, ***kwargs*)

Pad series above local minima

Parameters

- **series** (*array*) – The series to be padded.
- **pad_index** (*int, optional*) – The index of the local minima. Defaults to -1.
- ****kwargs** – Keyword arguments passed to scipy.signal.argrelmin()

Returns
series_padded – The padded series

Return type
array

kontrol.core.spectral.**pad_below_maxima**(*series*, *pad_index=0*, ***kwargs*)

Pad series below local maxima

Parameters

- **series** (*array*) – The series to be padded.
- **pad_index** (*int, optional*) – The index of the local maxima. Defaults to 0.
- ****kwargs** – Keyword arguments passed to scipy.signal.argrelmin()

Returns
series_padded – The padded series

Return type
array

kontrol.core.spectral.**pad_below_minima**(*series*, *pad_index=0*, ***kwargs*)

Pad series below local minima

Parameters

- **series** (*array*) – The series to be padded.
- **pad_index** (*int, optional*) – The index of the local minima. Defaults to 0.
- ****kwargs** – Keyword arguments passed to scipy.signal.argrelmin()

Returns
series_padded – The padded series

Return type
array

kontrol.core.spectral.**three_channel_correlation**(*psd1*, *psd2=None*, *psd3=None*, *csd12=None*, *csd13=None*, *csd21=None*, *csd23=None*, *csd31=None*, *csd32=None*, *returnall=True*)

Noise estimation from three sensors' readouts.

Parameters

- **psd1** (*array*) – The power spectral density of the readout of the first sensor.
- **psd2** (*array, optional*) – The power spectral density of the readout of the second sensor. Defaults `None`.
- **psd3** (*array, optional*) – The power spectral density of the readout of the third sensor. Defaults `None`.
- **csd12** (*array, optional*) – Cross power spectral density between readout 1 and 2. If not specified, this will be estimated as `psd1*psd2/csd21`. Default `None`.
- **csd13** (*array, optional*) – Cross power spectral density between readout 1 and 3. If not specified, this will be estimated as `psd1*psd3/csd31`. Default `None`.
- **csd21** (*array, optional*) – Cross power spectral density between readout 2 and 1. If not specified, this will be estimated as `psd2*psd1/csd12`. Default `None`.
- **csd23** (*array, optional*) – Cross power spectral density between readout 2 and 3. If not specified, this will be estimated as `psd2*psd3/csd32`. Default `None`.
- **csd31** (*array, optional*) – Cross power spectral density between readout 3 and 1. If not specified, this will be estimated as `psd1*psd3/csd13`. Default `None`.
- **csd32** (*array, optional*) – Cross power spectral density between readout 3 and 1. If not specified, this will be estimated as `psd3*psd2/csd23`. Default `None`.
- **returnall** (*boolean, optional*) – If `True`, return all three noise estimations. If `False`, return noise estimation of first sensor only. Defaults True.

Returns

- **noise1** (*array*) – Power spectral density of the estimated noise in *psd1*.
- **noise2** (*array, optional*) – Power spectral density of the estimated noise in *psd2*. Returns only if `returnall==True`
- **noise3** (*array, optional*) – Power spectral density of the estimated noise in *psd3*. Returns only if `returnall==True`

Notes

The PSD of the noise in *psd1* is then computed as

$$P_{n_1 n_1}(f) = \left| P_{x_1 x_1}(f) - \frac{P_{x_1 x_3}(f)}{P_{x_2 x_3}(f)} P_{x_2 x_1} \right| ,$$

If `returnall` is `True`, at least `psd1`, `psd2`, `psd3`, (`csd13` or `csd31`), (`csd23` or `csd32`), and (`csd12` and `csd21`) must be provided.

References

`kontrol.core.spectral.`**`two_channel_correlation`**(*psd*, *coh*)

Noise estimation from two identical sensors' readouts.

Parameters

- **psd** (*array*) – The power spectral density of the readout of the sensor
- **coh** (*array*) – The coherence between readout of the sensor and another identical sensor.

Returns

noise – Power spectral density of the estimated noise.

Return type

array

Notes

The PSD of the noise is computed as

$$P_{nn}(f) = P_{x_1 x_1}(f) \left(1 - C_{x_1 x_2}(f)^{\frac{1}{2}} \right) ,$$

where $P_{x_1 x_1}(f)$ is the power spectral density of the readout and $C_{x_1 x_2}(f)$ is the coherence between the two sensor readouts.

References

kontrol.core.foton module

KAGRA/LIGO Foton related utilities.

`kontrol.core.foton.`**`foton2tf`**(*foton_string*)

Convert a Foton string to TransferFunction

Parameters

foton_string (*str*) – The Foton string, e.g. zpk([0], [1; 1], 1, "n").

Returns

tf – The transfer function.

Return type

TransferFunction

kontrol.core.foton.**get_zpk2tf**(*get_zpk*, *plane='s'*)

Converts Foton's get_zpk() output to TransferFunction

Parameters

- **get_zpk** (*tuple(array, array, float)*) – The output from Foton's get_zpk() method of a filter instance.
- **plane** (*str, optional*) – "s", "f", or "n". Default "s".

Returns
tf – The converted transfer function.

Return type
TransferFunction

kontrol.core.foton.**notch**(*frequency*, *q*, *depth*, *significant_figures=6*)

Returns the foton expression of a notch filter.

Parameters

- **frequency** (*float*) – The notch frequency (Hz).
- **q** (*float*) – The quality factor.
- **depth** (*float*) – The depth of the notch filter (magnitude).
- **significant_figures** (*int, optional*) – Number of significant figures to print out. Defaults to 6.

Returns
The foton representation of this notch filter.

Return type
str

kontrol.core.foton.**rpoly2tf**(*foton_string*)

Converts rpoly Foton strings to TransferFunction

Parameters
foton_string (*str*) – The rpoly Foton string. E.g. rpoly([1; 2; 3], [2; 3; 4], 5)

kontrol.core.foton.**tf2foton**(*tf*, *expression='zpk'*, *root_location='s'*, *significant_figures=6*, *itol=1e-25*, *epsilon=1e-25*)

Convert a single transfer function to foton expression.

Parameters

- **tf** (TransferFunction) – The transfer function object.
- **expression** (*str, optional*) – Format of the foton expression. Choose from ["zpk", "rpoly"]. Defaults to "zpk".
- **root_location** (*str, optional*) – Root location of the zeros and poles for expression=="zpk". Choose from ["s", "f", "n"]. "s": roots in s-plane, i.e. zpk([...], [...], ..., "s"). "f": roots in frequency plane, i.e. zpk([...], [,,,], ..., "f"). "n": roots in frequency plane but negated and gains are normalized, i.e. real parts are positive zpk([...], [...], ..., "n"). Defaults to "s".
- **significant_figures** (*int, optional*) – Number of significant figures to print out. Defaults to 6.
- **itol** (*float, optional*) – Treating complex roots as real roots if the ratio of the imaginary part and the real part is smaller than this tolerance Defaults to 1e-25.

- **epsilon** (*float, optional*) – Small number to add to denominator to prevent division error. Defaults to 1e-25.

Returns
foton_expression – The foton expression in selected format.

Return type
str

kontrol.core.foton.**tf2rpoly**(*tf*, *significant_figures=6*)

Convert a transfer function to foton rpoly expression.

Parameters

- **tf** (TransferFunction) – The transfer function object
- **significant_figures** (*int, optional*) – Number of significant figures to print out. Defaults to 6.

Returns
Foton express in foton rpoly expression.

Return type
str

Notes

Only works for transfer functions with less than 20 orders.

kontrol.core.foton.**tf2zpk**(*tf*, *root_location='s'*, *significant_figures=6*, *itol=1e-25*, *epsilon=1e-25*)

Convert a single transfer function to foton zpk expression.

Parameters

- **tf** (TransferFunction) – The transfer function object.
- **root_location** (*str, optional*) – Root location of the zeros and poles. Choose from ["s", "f", "n"]. "s": roots in s-plane, i.e. zpk([...], [...], ..., "s"). "f": roots in frequency plane, i.e. zpk([...], [„„], ..., "f"). "n": roots in frequency plane but negated and gains are normalized, i.e. real parts are positive zpk([...], [...], ..., "n"). Defaults to "s".
- **significant_figures** (*int, optional*) – Number of significant figures to print out. Defaults to 6.
- **itol** (*float, optional*) – Treating complex roots as real roots if the ratio of the imaginary part and the real part is smaller than this tolerance Defaults to 1e-25.
- **epsilon** (*float, optional*) – Small number to add to denominator to prevent division error. Defaults to 1e-25.

Returns
The foton zpk expression in selected format.

Return type
str

Notes

Only works for transfer functions with less than 20 orders.

kontrol.core.foton.**zpk2tf**(*foton_string*)

Convert a Foton ZPK string to TransferFunction

Paramters

foton_string
[str] The Foton ZPK string, e.g. zpk([0], [1; 1], 1, "n").

returns
tf – The transfer function.

rtype
TransferFunction

kontrol.curvefit

Primary modules

kontrol.curvefit.curvefit

Base class for curve fitting

class kontrol.curvefit.curvefit.**CurveFit**(*xdata=None*, *ydata=None*, *model=None*, *model_kwargs=None*, *cost=None*, *optimizer=None*, *optimizer_kwargs=None*)

Bases: object

Base class for curve fitting

Parameters

- **xdata** (*array or None, optional*) – The independent variable data / data on the x axis. Defaults to None.
- **ydata** (*array or None, optional*) – The dependent variable data / data on the y axis. Defaults to None.
- **model** (*callable or None, optional*) – The model used to fit the data. The callable has a signature of func(x: array, args: array, **kwargs) -> array. args in model is an array of parameters that define the model. Defaults to None
- **model_kwargs** (*dict or None, optional*) – Keyword arguments passed to the model. Defaults to None.
- **cost** (*kontrol.curvefit.Cost or callable*) – Cost function. The callable has a signature of func(args, model, xdata, ydata) -> array. First argument is a list of parameters that will be passed to the model. This must be pickleable if multiprocessing is to be used. Defaults to None.
- **optimizer** (func(func, **kwargs) -> OptimizeResult, or None, optional) – The optimization algorithm use for minimizing the cost function.
- **optimizer_kwargs** (*dict or None, optional*) – Keyword arguments passed to the optimizer function. Defaults to None.

property cost

The cost function to be used to fit the data.

fit(*model_kwargs=None*, *cost_kwargs=None*, *optimizer_kwargs=None*)

Fit the data

Sets self.optimized_args and self.optimize_result.

Parameters

- **model_kwargs** (*dict or None, optional*) – Overriding keyword arguments in self.model_kwargs. Defaults to None.
- **cost_kwargs** (*dict or None, optional*) – Overriding keyword arguments in self.cost_kwargs. Defaults to None.
- **optimizer_kwargs** (*dict or None, optional*) – Overriding keyword argumetns in self.optimizer_kwargs. Defaults to None.

Return type
scipy.optimizer.OptimizeResult

property model

The model used to fit the data.

property model_kwargs

Keyword arguments passed to the model.

property optimize_result

Optimization Result

property optimized_args

The optimized arguments

property optimizer

The optimizer function to be used to fit the data.

property optimizer_kwargs

Keyword arguments passed to the optimizer function.

prefit()

Something to do before fitting.

property xdata

The independent variable data.

property ydata

The dependent variable data

property yfit

The fitted y values

kontrol.curvefit.transfer_function_fit

Transfer function fitting class

class kontrol.curvefit.transfer_function_fit.**TransferFunctionFit**(*xdata=None*, *ydata=None*, *model=None*, *model_kwargs=None*, *cost=None*, *weight=None*, *error_func_kwargs=None*, *optimizer=None*, *optimizer_kwargs=None*, *options=None*, *x0=None*)

Bases: *CurveFit*

Transfer function fitting class

This class is basically a `CurveFit` class with default cost functions and optimizer that is designed for fitting a transfer function. By default, the error function is `kontrol.curvefit.error_func.tf_error()`. The default optimizer is `scipy.optimize.minimize( ...,method="Nelder-Mead",...)`, with `options = { "adaptive": True, "maxiter": N*1000}`, where `N` is the number of variables. All of these can be overridden if specified.

Parameters

- **xdata** (`array` or `None`, optional) – Independent variable data. Defaults to `None`.
- **ydata** (`array` or `None`, optional) – Transfer function frequency response in complex numbers. Defaults to `None`.
- **model** (`func(x: ``array`, args: `array`, **kwargs)`` -> `array`, or `None`, optional) – The model used to fit the data. `args` in model is an `array` of parameters that define the model. Defaults to `None`
- **model_kwargs** (`dict` or `None`, optional) – Keyword arguments passed to the model. Defaults to `None`.
- **cost** (*kontrol.curvefit.Cost or callable*) – Cost function. The callable has a signature of func(args, model, xdata, ydata) -> array. First argument is a list of parameters that will be passed to the model. This must be pickleable if multiprocessing is to be used. Defaults to None.
- **weight** (`array` or `None`, optional) – Weighting function. Defaults `None`.
- **error_func_kwargs** (`dict` or `None`, optional) – Keyword arguments the will be passed to `error_func`, which is passed to the construct the cost function. Defaults to `None`.
- **optimizer** (`func(func, **kwargs)` -> `OptimizeResult`, or `None`, optional) – The optimization algorithm use for minimizing the cost function.
- **optimizer_kwargs** (`dict` or `None`, optional) – Keyword arguments passed to the optimizer function. Defaults to `None`.
- **options** (`dict` or `None`, optional) – The option arguments passed to the `optimizer`
- **x0** (`array`, optional) – Inital guess. Defaults to None.

property options

Option arguments passed to optimizer

property weight

Weighting function

property x0

Initial guess

Secondary modules

kontrol.curvefit.cost

Cost function base class

class kontrol.curvefit.cost.**Cost**(*error_func*, *error_func_kwargs=None*)

Bases: object

Cost function base class.

property error_func

The error function.

property error_func_kwargs

Keyword arguments passed to error_func

kontrol.curvefit.error_func

Error functions for curve fitting

kontrol.curvefit.error_func.**log_mse**(*x1*, *x2*, *weight=None*, *small_multiplier=1e-06*)

Logarithmic mean square error/Mean square log error

Parameters

- **x1** (*array*) – Array
- **x2** (*array*) – Array
- **weight** (*array or None, optional*) – weighting function. If None, defaults to *np.ones_like(x1)*
- **small_multiplier** (*float, optional*) – *x1* will be modified by *x1 `+ `small_multiplier `* `np.min(x2)* if 0 exists in *x1*. Same goes to *x2*. If 0 both exists in *x1* and *x2*, raise.

Returns

Logarithmic mean square error between arrays x1 and x2.

Return type

float

kontrol.curvefit.error_func.**mse**(*x1*, *x2*, *weight=None*)

Mean square error

Parameters

- **x1** (*array*) – Array
- **x2** (*array*) – Array
- **weight** (*array or None, optional*) – weighting function. If None, defaults to np.ones_like(x1)

Returns
Mean square error between arrays x1 and x2.

Return type
float

kontrol.curvefit.error_func.**noise_error**(*noise1*, *noise2*, *weight=None*, *small_number=1e-06*, *norm=2*)

Mean log error between two noise spectrums.

Parameters

- **noise1** (*array*) – Noise spectrum 1.
- **noise2** (*array*) – Noise spectrum 2.
- **weight** (*array or None, optional*) – Weighting function. Defaults None.
- **small_number** (*float, optional*) – A small number to be added in before evaluating np.log10 to prevent error. Defaults to 1e-6.
- **norm** (*int, optional*) – The type of norm used to estimate the error. Choose 1 for $mathcal{L}_1$ norm and 2 for $mathcal{L}_2$ norm. Other norms are are possible but not encouraged unless there's a specifiy reason. Defaults 2.

Returns
The mean log error.

Return type
float

Notes

$mathcal{L}_1$ norm is more robust while $mathcal{L}_2$ norm is more susceptible to outliers, but fits data with large dynamic range well.

kontrol.curvefit.error_func.**spectrum_error**(*spectrum1*, *spectrum2*, *weight=None*, *small_number=1e-06*, *norm=2*)

Mean log error between two spectrums. Alias of noise_error().

Parameters

- **spectrum1** (*array*) – Spectrum 1.
- **spectrum2** (*array*) – Spectrum 2.
- **weight** (*array or None, optional*) – Weighting function. Defaults None.
- **small_number** (*float, optional*) – A small number to be added in before evaluating np.log10 to prevent error. Defaults to 1e-6.
- **norm** (*int, optional*) – The type of norm used to estimate the error. Choose 1 for $mathcal{L}_1$ norm and 2 for $mathcal{L}_2$ norm. Other norms are are possible but not encouraged unless there's a specifiy reason. Defaults 2.

Returns
The mean log error.

Return type
float

Notes

$mathcal{L}_1$ norm is more robust while $mathcal{L}_2$ norm is more susceptible to outliers, but fits data with large dynamic range well.

kontrol.curvefit.error_func.**tf_error**(*tf1*, *tf2*, *weight=None*, *small_number=1e-06*, *norm=2*)

Mean log error between two transfer functions frequency response

Parameters

- **tf1** (*array*) – Frequency response of transfer function 1, in complex values.
- **tf2** (*array*) – Frequency response of transfer function 2, in complex values.
- **weight** (*array or None, optional*) – Weighting function. Defaults None.
- **small_number** (*float, optional*) – A small number to be added in before evaluating np.log10 to prevent error. Defaults to 1e-6.
- **norm** (*int, optional*) – The type of norm used to estimate the error. Choose 1 for $mathcal{L}_1$ norm and 2 for $mathcal{L}_2$ norm. Other norms are are possible but not encouraged unless there's a specifiy reason. Defaults 2.

Returns
Mean log error between two transfer function.

Return type
float

Notes

$mathcal{L}_1$ norm is more robust while $mathcal{L}_2$ norm is more susceptible to outliers, but fits data with large dynamic range well.

Subpackages

kontrol.curve.model

Model base class for curve fitting.

class kontrol.curvefit.model.model.**Model**(*args=None*, *nargs=None*, *log_args=False*)

Bases: object

Model base class for curve fitting

property **args**
Model parameters

property **nargs**
Number of model parameters

Transfer function models for transfer function fitting.

class kontrol.curvefit.model.transfer_function_model.**ComplexZPK**(*nzero_pairs*, *npole_pairs*, *args=None*, *log_args=False*)

Bases: *Model*

ZPK model with complex poles and zeros.

Parameters

- **nzero_pairs** (*int*) – Number of complex zero pairs.
- **npole_pairs** (*int*) – Number of complex pole pairs.
- **args** (*array, optional*) – The model parameters defined as `args = [f1, q1, f2, q2,..., fi, qi,..., fn, qn, k]`, where `f` are resonance frequencies of the complex zero/pole pairs, `q` are the quality factors, and `k` is the static gain, `i` is the number of complex zero pairs, and `n-i` is the number of of complex pole pairs.
- **log_args** (*boolean, optional*) – If true, model parameters passed to the model are assumed to be passed through a log10() function. So, when the real parameters will be assumed to be 10**args instead. Defaults to False.

fn_zero

List of resonance frequencies of the complex zeros.

Type
array

fn_pole

List of resonance frequencies of the complex poles.

Type
array

q_zero

List of Q-factors of the complex zeros.

Type
array

q_pole

List of Q-factors of the complex poles.

Type
array

Notes

The complex ZPK model is defined by:

$$G(s; f_i, q_i, k) = k\frac{\prod_i(\frac{s^2}{(2\pi f_i)^2} + \frac{1}{2\pi f_i q_i}s + 1)}{\prod_j(\frac{s^2}{(2\pi f_j)^2} + \frac{1}{2\pi f_j q_j}s + 1)}$$

property args

Model parameters in ZPK [f1, q1, f2, q2,..., fn, qn, k] format

property fn_pole

List of resonance frequencies of complex poles.

property fn_zero

List of resonance frequencies of complex zeros.

property gain

Static gain.

property npole_pairs

Number of complex pole pairs

property nzero_pairs

Number of complex zero pairs

property q_pole

List of quality factors of the complex poles

property q_zero

List of quality factors of the complex zeros

property tf

Returns a TransferFunction object of this ZPK model

class kontrol.curvefit.model.transfer_function_model.**CoupledOscillator**

Bases: *TransferFunctionModel*

class kontrol.curvefit.model.transfer_function_model.**DampedOscillator**(*args=None*)

Bases: *TransferFunctionModel*

Transfer function model for a damped oscillator.

Parameters

args (*array or None, optional.*) – The model parameters with three numbers. Structured as follows: `args = [k, fn, q]`, where `k` is the DC gain of the transfer function, `fn` is the resonance frequency in Hz, and `q` is the Q-factor. Defaults to None.

Notes

The model is definded as

$$G(s; k, \omega_n, q) = k \frac{\omega_n^2}{s^2 + \frac{\omega_n}{q} s + \omega_n^2}$$

where k is the DC gain of the transfer function, ω_n is the resonance frequency of the oscillator, and q is the Q-factor if the damped oscillator.

property args

Model parameters

property damped_oscillator_args

The model parameters with three numbers [k, fn, q]

property fn

Resonance frequency

property gain

DC gain

property q

Q factor

class kontrol.curvefit.model.transfer_function_model.**SimpleZPK**(*nzero*, *npole*, *args=None*, *log_args=False*)

Bases: *Model*

ZPK model with simple poles and zeros.

Parameters

- **nzero** (*int*) – Number of simple zeros.
- **npole** (*int*) – Number of simple poles.
- **args** (*array, optional*) – The model parameters defined as `args = [z1, z2,..., p1, p2,..., k]` where `z` are the locations of the zeros in Hz, `p` are the locations of the pole in Hz, and `k` is the static gain,
- **log_args** (*boolean, optional*) – If true, model parameters passed to the model are assumed to be passed through a log10() function. So, when the real parameters will be assumed to be 10**args instead. Defaults to False.

zero

List of zeros.

Type
array

pole

List of poles.

Type
array

gain

Static gain.

Type
float

tf

The transfer function representation of this ZPK model

Type
kontrol.TranserFunction

Notes

The simple ZPK model is defined as

$$G(s; z_1, z_2, ..., p_1, p_2, ..., k) = k \frac{\prod_i (\frac{s}{2\pi z_i} + 1)}{\prod_j (\frac{s}{2\pi p_j} + 1)}$$

property args

Model parameters in ZPK [z1, z2,..., p1, p2,..., k] format

property gain

Static gain

property npole

Number of complex pole pairs

property nzero

Number of simple zeros

property pole

List of poles

property tf

Returns a TransferFunction object of this ZPK model

property zero

List of zeros

class kontrol.curvefit.model.transfer_function_model.**TransferFunctionModel**(*nzero*, *npole*, *args=None*, *log_args=False*)

Bases: *Model*

Transfer function model class defined by numerator and denominator

Parameters

- **nzero** (*int*) – Number of zeros.
- **npole** (*int*) – Number of poles.
- **args** (*array or None, optional.*) – The model parameters. Structured as follows: [b_n, b_n-1,..., b_1, b_0, a_m, a_m-1,..., a_1, a_0], where b and a are the coefficients of the numerator and denominator respectively, ordered from higher-order to lower-order. Defaults to None.

tf

The last evaluted transfer function.

Type
kontrol.TransferFunction

Notes

The transfer function model is defined as

$$G(s, b_n, b_{n-1}, ..., b_1, b_0, a_m, a_{m-1}, ..., a_1, a_0) = \frac{\sum_{i=0}^{n} b_i s^i}{\sum_{j=0}^{m} a_j s^j}$$

property den

Denominator array

property npole

Number of zeros

property num

Numerator array

property nzero

Number of zeros

property tf

The Transfer Function object.

Basic mathematical models

class kontrol.curvefit.model.math_model.**Erf**(*args=None*)

Bases: *Model*

The error function erf(x)

Parameters
args (*array, optional*) – The model parameters structured as [amplitude, slope, x0, y0]. Defaults to None.

Notes

The function is defined as

$$f(x; a, m, x_0, y_0) = a\,\mathrm{erf}(m(x - x_0)) + y_0\,,$$

where $\mathrm{erf}(x)$ is the error function[1], a is the peak to peak amplitude, m is the slope at the inflection point, x_0 is the x offset , and y_0 is the y offset.

References

property amplitude

Peak to Peak amplitude of the error function.

property slope

Slope at the inflection point.

property x_offset

x offset

property y_offset

y offset

class kontrol.curvefit.model.math_model.**StraightLine**(*args=None*)

Bases: *Model*

Simply a straight line.

Parameters
args (*array, optional*) – The model parameters structued as: [slope, y-intercept]. Defaults to None.

slope

The slope of the line.

Type
float

intercept

The y-intercept of the line.

Type
float

[1] https://en.wikipedia.org/wiki/Error_function

Notes

The straight line is defined as

$$y(x; m, c) = mx + c\,,$$

where m is the slope and c is the y-intercept.

property intercept

The y-intercept of the line

property slope

The slope of the line

kontrol.dmd

Primary modules

kontrol.dmd.dmd

Dynamic mode decomposition class

class kontrol.dmd.dmd.**DMD**(*snapshot_1*, *snapshot_2=None*, *truncation_value=None*, *dt=1*, *run=False*)

Bases: object

Dynamic mode decomposition class

Parameters

- **snapshot_1** (*array*) – The 2-D snapshot array.
- **snapshot_2** (*array, optional*) – The 2-D snapshot array at next time step. If not provided, snapshot_1[:, :-1] becomes snapshot_1 and snapshot_2[:, 1:] becomes snapshot_2. Defaults None.
- **truncation_value** (*float, optional*) – The truncation value (order/rank) of the system. Defaults None.
- **dt** (*float, optional*) – The time difference between two snapshots. Defaults 1.
- **run** (*bool, optional*) – Run dynamic mode decomposition upon construction. Computes DMD modes, Reduced-order model, etc. truncation_value must be specified for this to work. Defaults False.

snapshot_1

The 2-D snapshot matrix

Type
array

snapshot_2

The 2-D snapshot matrix at next time step.

Type
array

`truncation_value`

The truncation value (order/rank) of the system.

Type
int

`dt`

The time difference between two snapshots.

Type
float

`u`

The left singular vector of the snapshot_1 matrix.

Type
array

`vh`

The right singular vector (conjugate-transposed) of the snapshot_1 matrix.

Type
array

`sigma`

The singular values of the snapshot_1 matrix

Type
array

`u_truncated`

The truncated left singular vector of the snapshot_1 matrix.

Type
array

`vh_truncated`

The truncated right singular vector (conjugate-transposed) of the snapshot_1 matrix.

Type
array

`sigma_truncated`

The truncated singular values of the snapshot_1 matrix

Type
array

`A_reduced`

The reduced-order model of the dynamics.

Type
array

`w_reduced`

The eigenvalues of the reduced-order model.

Type
array

`v_reduced`

The eigenvectors of the reduced-order model.

Type
array

`dmd_modes`

The DMD modes.

Type
array

`complex_frequencies`

The complex frequencies of the modes.

Type
array

`v_constant`

The constant vector of the time dynamics, determined by initial conditions.

Type
array

`time_dynamics`

The time dynamics of the ODE solution.

Type
array

`prediction`

The predicted time series.

Type
array

`property A_reduced`

Reduced-order model

`property complex_frequencies`

Complex frequencies

`compute_complex_frequencies`(*dt=None*)

Compute the complex frequencies

Parameters
dt (*float, optional*) – The time spacing between snapshots. Specified as self.dt or in constructor option. Defaults 1.

Returns
complex_frequencies – Array of complex frequencies

Return type
array

`compute_dmd_modes`()

Compute the DMD modes

Returns
dmd_modes – The DMD modes.

Return type
array

`compute_reduced_model()`

Compute the reduced-order model

Returns
A_reduced – Matrix representing the reduced-order model.

Return type
array

`property dt`

Time difference between the snapshots

`eig_reduced_model()`

Eigen decomposition of the reduced-order model

Returns

- **w_reduced** (*array*) – The list of eigenvalues.
- **v_reduced** (*array*) – Eigenvalues as columns.

`low_rank_approximation(truncation_value=None)`

Truncate the svd.

Paramters

truncation_value
[int, optional] The truncation value (order/rank) of the system. Specify as self.truncation_value or via the constuctor option. Defaults None.

returns

- **u_truncated** (*array*) – Truncated unitary matrix having left singular vectors as columns.
- **sigma_truncated** (*array*) – Truncated singular values.
- **vh_truncated** (*array*) – Truncated unitary matrix having right singular vectors as rows.

`predict(t, t_constant=None, i_constant=0)`

Predict the future states given a time array

Parameters

- **t** (*array*) – The time array
- **t_constant** (*float, optional*) – Time at which the constant vector is calculated. Defaults to be `t[0]` if not given.
- **i_constant** (*int, optional*) – Index of the `snapshot_1` at which the constant vector is calculated. Defaults 0.

Returns
prediction – Predicted states.

Return type
array

Notes

The constant vector is the vector that evolves and is evaluated using the initial/boundary values. Set `t_constant` and `i_constant` to where the prediction of the time series begins.

property prediction

The ODE solution

run()

Run the DMD algorithm, compute dmd modes and complex frequencies

property sigma

Singular values

property sigma_truncated

Singular values truncated

property snapshot_1

Snapshot 1

property snapshot_2

Snapshot 1

svd()

Decompose the snapshot 1

Returns

- **u** (*array*) – Unitary matrix having left singular vectors as columns.
- **sigma** (*array*) – The singular values.
- **vh** (*array*) – Unitary matrix having right singular vectors as rows.

property time_dynamics

Time dynamics of the ODE solution

property truncation_value

Truncation value (order/rank) of the system

property u

Left singular vectors

property u_truncated

Left singular vectors (truncated)

property v_constant

Constant vector of the ODE solution

property vh

Right singular vectors

property vh_truncated

Right singular vectors (truncated)

kontrol.dmd.forecast

kontrol.dmd.forecast.**dmd_forecast**(*past*, *t_past*, *t_future*, *order=None*, *truncation_value=None*, *truncate_threshold=0.99*, *t_constant=None*, *i_constant=-1*, *return_dmd=False*)

Produce a short term forecast from past time series data using DMD.

Parameters

- **past** (*array*) – Past time series
- **t_past** (*array*) – Past time axis.
- **t_future** (*array*) – The time stamps of the future prediction. Preferably some time after t_past.
- **order** (*int, optional*) – The order of hankelization. Defaults to `int(len(past)/2)`.
- **truncation_value** (*int, optional*) – The number of modes to be taken. If not provided, `kontrol.dmd.auto_truncate()` will be used with the parameter `truncate_threshold` to obtain the value.
- **truncate_threshold** (*float, optional*) – Truncate modes that has singular values less than this fraction of the total. This should within [0, 1], with 0 being all modes accepted and 1 being all modes truncated. Defaults to 0.99.
- **t_constant** (*int, optional*) – Time at which the constant vector is defined. Only make sense if it's a value in `t_past`. Defaults t_past[-1].
- **i_constant** (*int, optional*) – The index of the column vector in the data matrix (snapshot) at `t_constant`. Defaults -1.
- **return_dmd** (*boolean, optional*) – Return the DMD instance if True. Defaults to False.

Returns

- **forecast** (*array*) – The forecast
- **dmd** (*kontrol.dmd.DMD, optional*) – The DMD instance used to produce the forecast.

Examples

```
t_past = np.linspace(0, 10, 1024) # Some time axis.
t_future = np.linspace(10, 15, 512) # Forecast between t=[10, 15].
time_series = ... # Some time series
forecast = dmd_forecast(time_series, t_past, t_future)
```

Secondary modules

kontrol.dmd.utils

Utility functions for Dymanic Mode Decomposition.

kontrol.dmd.utils.**auto_truncate**(*sigma*, *threshold=0.99*)

Automatically get truncation value

Parameters

- **sigma** (*array*) – Singular values. Arranged from large to small values.
- **threshold** (*float, optional*) – Only include singular values so their sum is `threshold` of the original sum. Defaults 0.99.

Returns
truncation_value – Amount of singular values that are required to reach the threshold sum.

Return type
int

`kontrol.dmd.utils.`**hankel**(*array*, *order*)

Hankelize an array

Parameters

- **array** (*array*) – Array with rows as channels and columns as samples
- **order** (*int*) – The number of rows of the hankelized matrix.

Returns
hankel_array – The hankelized array

Return type
array

Examples

kontrol.regulator

Primary modules

kontrol.regulator.feedback

Algorithmic designs for feedback control regulators.

`kontrol.regulator.feedback.`**add_integral_control**(*plant*, *regulator=None*, *integrator_ugf=None*, *integrator_time_constant=None*, ***kwargs*)

Match and returns an integral gain.

This function finds an integral gain such that the UGF of the integral control matches that of the specified regulator. If `integrator_ugf` or `integrator_time_constant` is specified instead, these will be matched instead.

Parameters

- **plant** (`TransferFunction`) – The transfer function representation of the system to be feedback controlled.
- **regulator** (`TransferFunction`, *optional*) – The pre-regulator Use `kontrol.regulator.feedback.proportional_derivative()` or `kontrol.regulator.feedback.critical_damping()` to make one for oscillator-like systems.
- **integrator_ugf** (*float, optional*) – The unity gain frequency (Hz) of the integral control. This is the inverse of the integration time constant. If `integrator_time_constant is not None`, then this value will be ignored. If set to None, it'll be set to match the first UGF of the derivative control. Defaults to None.
- **integrator_time_constant** (*float, optional,*) – The integration time constant (s) for integral control. Setting this will override the `integrator_ugf` argument. Defaults to None.

Returns
ki – The integral control gain.

Return type
float

kontrol.regulator.feedback.**add_proportional_control**(*plant*, *regulator=None*, *dcgain=None*, ***kwargs*)

Match and returns proportional gain.

This function finds a proportional gain such that the proportional control UGF matches the first UGF of the specified regulator (typically a derivative control regulator). If `dcgain` is specified, the DC gain is matched instead.

Parameters

- **plant** (TransferFunction) – The transfer function representation of the system to be feedback controlled. The plant must contain at least one pair of complex poles.
- **regulator** (TransferFunction, *optional*) – The pre-regulator. If not specified, then `dcgain` must be specified. Defaults to None.
- **dcgain** (*float, optional*) – The desired DC gain of the open-loop transfer function. If not specified, the portional gain is tuned such that the portional control's first UGF matches that of the derivative control. Defaults to None.

Returns
kp – The proportional control gain.

Return type
float

kontrol.regulator.feedback.**critical_damp_calculated**(*plant*, *nmode=1*, ***kwargs*)

Find dominant mode and returns the approximate critical regulator.

Parameters

- **plant** (TransferFunction) – The transfer function representation of the system to be feedback controlled. The plant must contain at least one pair of complex poles.
- **nmode** (*int, optional*) – The ``nmode``th dominant mode to be damped. This number must be less than the number of modes in the plant. Defaults 1.

Returns
kd – The derivative gain for critically damping the dominant mode.

Return type
float

Notes

The plant must contain at least one complex pole pair. The plant must have a non-zero DC gain.

The critical regulator is approximated by

$$K(s) \approx \frac{2}{\omega_n K_{\mathrm{DC}}},$$

where ω_n is the resonance frequency of the dominant mode and K_{DC} is the sum of DC gains of the mode and that of the other higher-frequency modes..

kontrol.regulator.feedback.**critical_damp_optimize**(*plant*, *gain_step=1.1*, *ktol=1e-06*, ***kwargs*)

Optimize derivative damping gain and returns the critical regulator

Parameters

- **plant** (TransferFunction) – The transfer function representation of the system to be feedback controlled. The plant must contain at least one pair of complex poles.
- **gain_step** (*float, optional*) – The multiplicative factor of the gain for finding the gain upper bound. It must be greater than 1. Defaults to 1.1.
- **ktol** (*float, optional*) – The tolerance for the convergence condition. The convergence condition is (kd_max-kd_min)/kd_min > ktol. It must be greater than 0. Defaults to 1e-6.

Returns

kd – The derivative gain for critically damping the dominant mode.

Return type

float

Notes

Update on 2021-12-04: Use carefully. It only critically damps the mode that has the highest peak when multiplied by an differentiator. Note for myself: only 2 complex poles can become simple poles for plants with second-order rolloff.

Only works with plants that contain at least one complex pole pair. Works best with plants that only contain complex zeros/poles. If it returns unreasonably high gain, try lowering `gain_step`.

The algorithm goes as follows.

1. Find the minimum damping gain `k_min` such that the open-loop transfer function `k_min*s*plant` has maxmimum gain at unity gain frequency.

2. Iterate `i`: `k_i=k_min*i*gain_step` for i=1,2,3…

3. Terminate when `1/(1+k_i*s*plant)` has less complex pole pairs than the `plant` itself, i.e. one mode has been overdamped. Then, define `k_max=k_i`.

4. Iterate: Define `k_mid` as the logarithmic mean of `k_max` and `k_min`. If `1/(1+k_mid*s*plant)` has less complex pole pairs than `plant`, i.e. overdamps, then set `k_max=k_mid`. Otherwise, set `k_min=k_mid`.

5. Terminate when `(k_max - k_min)/k_min < ktol`, i.e. converges.

kontrol.regulator.feedback.**critical_damping**(*plant*, *method='calculated'*, ***kwargs*)

Returns the critical damping derivative control gain.

This functions calls `kontrol.regulator.feedback.critical_damp_calculated()` or `kontrol.regulator.feedback.cricical_damp_optimized()` and returns the derivative control gain.

Parameters

- **plant** (TransferFunction) – The transfer function representation of the system to be feedback controlled. The plant must contain at least one pair of complex poles.
- **method** (*str, optional*) – The method to be used for setting the gain. Choose from ["optimized", "calculated"].

  "optimized": the gain is optimized until the dominant complex pole pairs become two simple poles.

"calculated": the gain is set to $2/\omega_n/K_{DC}$, where ω_n is the resonance frequency in rad/s of the mode to be damped, and K_{DC} is the sum of DC gains of the mode and that of the other high-frequency modes.

Both method assumes that the plant has at least one pair of complex poles.

Defaults to "calculated".

- ****kwargs** (*dict*) – Method specific keyword arguments.

  See:

  - "optimized": kontrol.regulator.feedback.critical_damp_optimized
  - "calculated": kontrol.regulator.feedback.critical_damp_calculated

Returns
kd – The derivative gain for critically damping the dominant mode.

Return type
float

kontrol.regulator.feedback.**mode_composition**(*wn*, *q*, *k*)

Create a plant composed of many modes.

Parameters

- **wn** (*array*) –
- **(rad/s).** (*Frequencies*) –
- **q** (*array*) – Q factors.
- **k** (*array*) – Dcgains of the modes.

Returns
The composed plant.

Return type
TransferFunction

kontrol.regulator.feedback.**mode_decomposition**(*plant*)

Returns a list of single mode transfer functions

Parameters
plant (TransferFunction) – The transfer function with at list one pair of complex poles.

Returns

- **wn** (*array*) – Frequencies (rad/s).
- **q** (*array*) – Q factors.
- **k** (*array*) – Dcgains of the modes.

kontrol.regulator.oscillator

Control regulators designs for oscillator-like systems

kontrol.regulator.oscillator.**pid**(*plant*, *regulator_type='PID'*, *dcgain=None*, *integrator_ugf=None*, *integrator_time_constant=None*, *return_gain=False*, ***kwargs*)

PID-like controller design for oscillator-like systems

Parameters

- **plant** (TransferFunction) – The transfer function of the system that needs to be controlled.
- **regulator_type** (*str, optional*) – The type of the contorl regulator. Choose from {"PID", "PD", "PI", "I", "D"} for proportional-integral-derivative, proportional-derivative, proportional-integral, or derivative (velocity) control respectively. Defaults to "PID".
- **dcgain** (*float, optional*) – The DC gain of the OLTF of the proportional control. If set to None, it will be set automatically depending on the type of the controller. If regulator_type=="PI", then dcgain will be set such that the proportional control UGF matches that of the integral control. If regulator_type=="PD" or "PID", then the dcgain will be set such that it matches the UGF of the derivative control. Defaults to None.
- **integrator_ugf** (*float, optional*) – The unity gain frequency (Hz) of the integral control. This is the inverse of the integration time constant. If integrator_time_constant is not None, then this value will be ignored. If set to None, it'll be set to match the UGF of the derivative control. For regulator_type=="I", this must be specified. Defaults to None.
- **integrator_time_constant** (*float, optional,*) – The integration time constant (s) for integral control. Setting this will override the integrator_ugf argument. Defaults to None.
- **return_gain** (*boolean, optional*) – Return the PID gain instead.

Returns

- **regulator** (*TransferFunction, optional*) – The regulator. Return only if return_gain is False.
- **kp** (*float, optional*) – Proportional gain. Return only if return_gain is ``True".
- **ki** (*float, optional*) – Integral gain. Return only if return_gain is True.
- **kd** (*float, optional*) – Derivative gain. Return only if return_gain is True.

kontrol.regulator.post_filter

Functions for designing post-regulator filters

kontrol.regulator.post_filter.**post_low_pass**(*plant*, *regulator*, *post_filter=None*, *ignore_ugf_above=None*, *decades_after_ugf=1*, *phase_margin=45*, *f_start=None*, *f_step=1.1*, *low_pass=None*, *mtol=1e-06*, *small_number=1e-06*, *oscillatory=True*, ***kwargs*)

Add low-pass filter after regulator.

This function lowers/increase the the cutoff frequency of a low-pass filter until the phase margin at a dedicated ugf crosses the specified phase margin. Then, runs a bisection algorithm to poolish the cutoff frequency until the phase margin converges relative to the specified tolerance.

Parameters

- **plant** (TransferFunction) – The transfer function of the system that needs to be controlled.
- **regulator** (TransferFunction) – The regulator.
- **post_filter** (TransferFunction, *optional*) – Any post filters that will be applied on top of the regulator. Defaults None.
- **ignore_ugf_above** (*float, optional*) – Ignore unity gain frequencies higher than ignore_ugf_above (Hz). If not specified, defaults to 1 decade higher than the last UGF. This value can be overrided by the argument decades_after_ugf Note that there's no guarantee that the UGF will be lower than this. The priority is to match the target phase margin. Defaults to None.
- **decades_after_ugf** (*float, optional*) – Set ignore_ugf_above some decades higher than the UGF of the OLTF ignore_ugf_above is None. Defaults to 1.
- **phase_margin** (*float, optional,*) – The target phase margin (Degrees). Defaults to 45.
- **f_start** (*float, optional,*) – The cutoff frequency to start iterating with. If not specified, defaults to some decades higher than the highest UGF. "Some decade" is set by decades_after_ugf. Defaults None.
- **f_step** (*float, optional,*) – The gain that is used to multiply (or divide) the cutoff frequency during a coarse search. Defaults 1.1
- **low_pass** (*func(cutoff, order) -> TransferFunction, optional*) – The low-pass filter. If not specified, kontrol.regulator.predefined.lowpass() with order 2 will be used. Defaults to None.
- **mtol** (*float, optional*) – Tolerance for convergence of phase margin. Defaults to 1e-6.
- **small_number** (*float, optional*) – A small number as a delta f to detect whether the gain is a rising or lowering edge at the unity gain frequency. Defaults to 1e-6.
- **oscillatory** (*boolean, optional*) – Use the first mode of the oscillatory system to evaluate the phase margins to avoid having UGFs at steep phase slopes. The benefit of using this is to have a conservative phase margin estimate. The phase response of the first mode is the lower bound of the phase response. If False, use the plant itself to calculate phase margins. If the plant does not contain any complex poles, this option will be overridden to False. Defaults True.
- ****kwargs** – Keyword arguments passed to low_pass.

Returns

The low-pass filter.

Return type

TransferFunction

kontrol.regulator.post_filter.**post_notch**(*plant*, *regulator=None*, *post_filter=None*, *target_gain=None*, *notch_peaks_above=None*, *phase_margin=45*, *notch=None*, ***kwargs*)

Returns a list of notch filters that suppress resonance peaks.

This functions finds the resonances peak of the plant/OLTF above certain frequencies and returns a list of notch filters that suppress these peaks to the target gains.

Parameters

- **plant** (TransferFunction) – The transfer function of the system that needs to be controlled.
- **regulator** (TransferFunction, *optional*) – The regulator. Defaults to None.
- **post_filter** (TransferFunction, *optional*) – Any post filters that will be applied on top of the regulator. Defaults None.
- **target_gain** (*float, optional*) – The target open-loop gain for the suppressed peak. To ensure a stable system, a value of less than 1 is recommended. If not specified, the notch will fully suppress the peak. Default None.
- **phase_margin** (*float, optional*) – The target phase margin. Defaults to 45.
- **notch_peaks_above** (*float, optional*) – Notch modes that has freqeuncies above notch_peaks_above. If not specified, defaults to the highest unity gain frequency that is above phase_margin Defaults to None.
- **notch** (*func(frequency, q, depth) -> TransferFunction, optional*) – The notch filter. If not specified, kontrol.Notch() will be used. Defaults to None.
- ****kwargs** – Keyword arguments passed to notch().

Returns
A list of notch filters.

Return type
list of *TransferFunction*

Notes

This operation does not guarantee stability. It only finds resonances peaking out of the unity gain above some unity gain frequency and make notch filters to suppress them to a target gain level lower than the unity gain. The stability of the notched OLTF is not checked whatsoever.

Secondary modules

kontrol.regulator.predefined

Predefined regulator library.

kontrol.regulator.predefined.**low_pass**(*cutoff*, *order=1*, ***kwargs*)

Simple low-pass filter

Parameters

- **cutoff** (*float*) – Cutoff frequency (Hz)
- **order** (*int, optional*) – The order of the filter. Defaults to be 1.
- ****kwargs** – Keyword arguments holder. Not passed to anywhere.

Returns
The low-pass filter.

Return type
TransferFunction

Notes

The low-pass filter is defined as

$$L(s) = \left(\frac{2\pi f_c}{s + 2\pi f_c}\right)^n ,$$

where f_c is the cutoff frequency (Hz), n is the order of the filter.

kontrol.regulator.predefined.**notch**(*frequency*, *q*, *depth=None*, *depth_db=None*, ***kwargs*)

Notch filter defined in Foton.

Parameters

- **frequency** (*float*) – The notch frequency (Hz).
- **q** (*float*) – The quality factor.
- **depth** (*float, optional*) – The depth of the notch filter (magnitude). If not specified, depth_db will be used. Defaults None.
- **depth_db** (*float, optional*) – The depth of the notch filter (decibel). If not specified, depth will be used instead. Defaults None.

Returns
The notch filter

Return type
TransferFunction

Notes

The notch filter is defined by Foton, as

$$N(s) = \frac{s^2 + (2\pi f_n)/(dQ/2)s + (2\pi f_n)^2}{s^2 + (2\pi f_n)/(Q/2)s + (2\pi f_n)^2} ,$$

where f_n is the notch frequency, q is the quality factor , and d is the depth.

kontrol.regulator.predefined.**pid**(*kp=0*, *ki=0*, *kd=0*)

Alias of proportional_integral_derivative()

kontrol.regulator.predefined.**proportional_integral_derivative**(*kp=0*, *ki=0*, *kd=0*)

PID control build on proportional_derivative().

Parameters

- **kp** (*float, optional*) – The proportional control gain. Defaults to 0.
- **ki** (*float, optional*) – The integral control gain. Defaults to 0.
- **kd** (*float, optional*) – Defaults to 0.

Returns
The PID controller.

Return type
TransferFunction

Notes

The PID controller is defined as

$$K_{\mathrm{PID}}(s) = K_p + K_i/s + K_d s\,.$$

kontrol.sensact

Primary modules

kontrol.sensact.matrix

Base class for matrices, sensing and actuation matices.

class kontrol.sensact.matrix.**Matrix**(*matrix=None*, **args*, ***kwargs*)

Bases: ndarray

Base class for matrices

class kontrol.sensact.matrix.**SensingMatrix**(*matrix=None*, **args*, ***kwargs*)

Bases: *Matrix*

Base class for sensing matrices

diagonalize(*coupling_matrix=None*)

Diagonalize the sensing matrix, given a coupling matrix.

Parameters

coupling_matrix (*array, optional.*) – The coupling matrix. If None, self.coupling_matrix will be used. Default None.

Returns

The new sensing matrix.

Return type

array

Notes

The coupling matrix has (i, j) elements as coupling ratios x_i/x_j. For example, consider the 2-sensor configuration: I have a coupled sensing readout $x_{1,\mathrm{coupled}}$ that reads $x_{1,\mathrm{coupled}} = x_1 + 0.1x_2$. and, I have another coupled sensing readout $x_{2,\mathrm{coupled}}$ that reads $x_{2,\mathrm{coupled}} = -0.2x_1 + x_2$. Then, the coupling matrix is

$$\begin{bmatrix} 1 & 0.1 \\ -0.2 & 1 \end{bmatrix}.$$

kontrol.sensact.optical_lever

Optical lever sensing matrix

class kontrol.sensact.optical_lever.**HorizontalOpticalLeverSensingMatrix**(*r*, *alpha_h*, *r_lens=0*, *f=inf*, *alpha_v=0*, *phi_tilt=0*, *phi_len=0*, *delta_x=0*, *delta_y=0*, *delta_d=0*, *format='OPLEV2EUL'*, *coupling_matrix=None*, **args*, ***kwargs*)

Bases: *OpticalLeverSensingMatrix*

Horizontal optical lever sensing matrix.

Parameters

- **r** (*float*) – Lever arm.
- **alpha_h** (*float*) – Angle of incidence on the horizontal plane.
- **r_lens** (*float, optional*) – Lever arm from the optics to the convex lens. Default 0.
- **f** (*float, optional*) – Focal length of the convex lens. Default np.inf.
- **alpha_v** (*float, optional*) – Angle of incidence on the vertical plane. Default 0.
- **phi_tilt** (*float, optional*) – Angle from the tilt-sensing QPD frame to the yaw-pitch frame. Default 0.
- **phi_len** (*float, optional*) – Angle from the length-sensing QPD frame to the yaw-pitch frame. Default 0.
- **delta_x** (*float, optional*) – Horizontal miscentering of the beam spot at the optics plane. Default 0.
- **delta_y** (*float, optional*) – Vertical miscentering of the beam spot at the optics plane. Default 0.
- **delta_d** (*float, optional*) – Misplacement of the length-sensing QPD. Default 0.
- **format** (*str, optional*) – Format of the sensing matrix. Choose from

  "OPLEV2EUL": Default sensing matrix from KAGRA MEDM screen with input (TILT_PIT, TILT_YAW, LEN_PIT, LEN_YAW), and output (longitudinal, pitch and yaw).

  "xy": Matrix as shown in [1]_.
- **coupling_matrix** (*array, optional*) – The coupling matrix. Default None.
- ***args** – Variable length arguments passed to OpticalLeverSensingMatrix.
- ****kwargs** – Keyword arguments passed to OpticalLeverSensingMatrix.

property delta_d

Misplacement of the length-sensing QPD.

property f

Focal length of the convex lens.

property r

Lever arm.

property r_lens

Lever arm from the optics to the convex lens.

class kontrol.sensact.optical_lever.**OpticalLeverSensingMatrix**(*r_h*, *r_v*, *alpha_h*, *alpha_v*, *r_lens_h=0*, *r_lens_v=0*, *d_h=0*, *d_v=0*, *delta_x=0*, *delta_y=0*, *phi_tilt=0*, *phi_len=0*, *f_h=inf*, *f_v=inf*, *format='OPLEV2EUL'*, *coupling_matrix=None*, **args*, ***kwargs*)

Bases: *SensingMatrix*

Optical lever sensing matrix base class.

This is a sensing matrix that maps tilt-sensing QPD and length-sensing QPD (placed behind a lens) readouts to the suspended optics' longitudinal, pitch, and yaw displacements.

Parameters

- **r_h** (*float*) – Lever arm from the optics to the tilt-sensing QPD plane on the horizontal plane (amplifying yaw).
- **r_v** (*float*) – Lever arm from the optics to the tilt-sensing QPD plane on the vertical plane (amplifying pitch).
- **alpha_h** (*float*) – Angle of incidence on the horizontal plane.
- **alpha_v** (*float*) – Angle of incidence on the vertical plane.
- **r_lens_h** (*float, optional*) – Lever arm from optics to the lens on the horizontal plane. Defaults None. Specify if it exists.
- **r_lens_v** (*float, optional*) – Lever arm from optics to the lens on the vertical plane. Defaults None. Specify if it exists.
- **d_h** (*float, optional*) – Horizontal distance from the lens to the length-sensing QPD. Defaults None.
- **d_v** (*float, optional*) – Vertical distance from the lens to the length-sensing QPD. Defaults None.
- **delta_x** (*float, optional*) – Horizontal miscentering of the beam spot at the optics plane.
- **delta_y** (*float, optional*) – Vertical miscentering of the beam spot at the optics plane.
- **phi_tilt** (*float, optional*) – Angle from the tilt-sensing QPD frame to the yaw-pitch frame. Defaults None.
- **phi_len** (*float, optional*) – Angle from the length-sensing QPD frame to the yaw-pitch frame. Defaults None.
- **f_h** (*float, optional,*) – Focal length of the lens projected to the horizontal plane. Defaults `np.inf` (no lens).
- **f_v** (*float, optional,*) – Focal length of the lens projected to the vertical plane. Defaults `np.inf` (no lens).
- **format** (*str, optional*) – Format of the sensing matrix. Choose from

"OPLEV2EUL": Default sensing matrix from KAGRA MEDM screen
with input (TILT_PIT, TILT_YAW, LEN_PIT, LEN_YAW), and output (longitudinal, pitch and yaw).

"xy": Matrix as shown in [1]_.

- **coupling_matrix** (*array, optional*) – The coupling matrix. Default None.

Notes

We're using equation (29) from [1]_.

$$\begin{pmatrix} x_L \\ \theta_P \\ \theta_Y \end{pmatrix} = \mathbf{C}_{\text{miscenter}} \mathbf{C}_{\text{align}} \mathbf{C}_{\text{rotation}} \begin{pmatrix} x_{\text{tilt}} \\ y_{\text{tilt}} \\ x_{\text{len}} \\ y_{\text{len}} \end{pmatrix},$$

where x_L is the longitudinal displacement of the optics, θ_P is the pitch angular displacement of the optics, θ_Y is the yaw angular displacement of the optics, x_{tilt} is the horizontal displacement of the beam spot at the tilt-sensing QPD plane, y_{tilt} is the vertical displacement of the beam spot at the tilt-sensing QPD plane, x_{len} is the horizontal displacement of the beam spot at the length-sensing QPD plane, y_{len} is the vertical displacement of the beam spot at the length-sensing QPD plane,

$$\mathbf{C}_{\text{rotation}} = \begin{bmatrix} \cos\phi_{\text{tilt}} & \sin\phi_{\text{tilt}} & 0 & 0 \\ -\sin\phi_{\text{tilt}} & \cos\phi_{\text{tilt}} & 0 & 0 \\ 0 & 0 & \cos\phi_{\text{len}} & \sin\phi_{\text{len}} \\ 0 & 0 & -\sin\phi_{\text{len}} & \cos\phi_{\text{len}} \end{bmatrix},$$

$$\mathbf{C}_{\text{align}} = \begin{bmatrix} 2\sin\alpha_h & 0 & 2r_h \\ 2\sin\alpha_v & 2r_v & 0 \\ 2\sin\alpha_h\left(1-\frac{d_h}{f_h}\right) & 0 & 2\left[\left(1-\frac{d_h}{f_h}\right)r_{\text{lens},h}+d_h\right] \\ 2\sin\alpha_v\left(1-\frac{d_v}{f_v}\right) & 2\left[\left(1-\frac{d_v}{f_v}\right)r_{\text{lens},v}+d_v\right] & 0 \end{bmatrix}^{+},$$

$$\mathbf{C}_{\text{miscenter}} = \begin{bmatrix} 1 & \delta_y & \delta_x \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1},$$

ϕ_{tilt} is the angle between the tilt-sensing QPD and the yaw-pitch frame, ϕ_{len} is the angle between the length-sensing QPD and the yaw-pitch frame, r_h is the lever arm on the horzontal plane, r_v is the lever arm on the vertical plane, α_h is the angle of incidence on the horizontal plane, α_v is the angle of incidence on the vertical plane, $r_{\text{lens},h}$ is the lever arm between the optics and the lens on the horizontal plane, $r_{\text{lens},v}$ is the lever arm between the optics and the lens on the vertical plane, d_h is the distance between the lens and the length-sensing QPD on the horizontal plane, d_v is the distance between lens and the length-sensing QPD on the vertical plane, and f is the focal length of the convex lens.

References

property alpha_h

Angle of incidence on the horizontal plane.

property alpha_v

Angle of incidence on the vertical plane.

property d_h
Horizontal distance from the lens to the length-sensing QPD.

property d_v
Vertical distance from the lens to the length-sensing QPD.

property delta_x
Horizontal miscentering of the beam spot at the optics plane.

property delta_y
Horizontal miscentering of the beam spot at the optics plane.

property f_h
Focal length of the convex lens projected on the horizontal plane.

property f_v
Focal length of the convex lens projected on the vertical plane.

property format
Format of the sensing matrix.

Choose from

"OPLEV2EUL": Default sensing matrix from KAGRA MEDM screen
with input (TILT_PIT, TILT_YAW, LEN_PIT, LEN_YAW), and output (longitudinal, pitch and yaw).

"xy": Matrix as shown in [1]_.

References

property phi_len
Angle from the length-sensing QPD frame to the yaw-pitch frame.

property phi_tilt
Angle from the tilt-sensing QPD frame to the yaw-pitch frame.

property r_h
Lever arm from optics to tilt-sensing QPD on the horizontal plane.

property r_lens_h
Lever arm from optics to the lens on the horizontal plane.

property r_lens_v
Lever arm from optics to the lens on the vertical plane.

property r_v
Lever arm from optics to tilt-sensing QPD on the vertical plane.

update_matrices_decorator()
Update matrices and self upon setting new parameters.

class kontrol.sensact.optical_lever.**VerticalOpticalLeverSensingMatrix**(*r*, *alpha_v*, *r_lens=0*, *f=inf*, *alpha_h=0*, *phi_tilt=0*, *phi_len=0*, *delta_x=0*, *delta_y=0*, *delta_d=0*, *format='OPLEV2EUL'*, *coupling_matrix=None*, **args*, ***kwargs*)

Bases: *OpticalLeverSensingMatrix*

Vertical optical lever sensing matrix.

Parameters

- **r** (*float*) – Lever arm.
- **alpha_v** (*float*) – Angle of incidence on the vertical plane.
- **r_lens** (*float, optional*) – Lever arm from the optics to the convex lens. Default 0.
- **f** (*float, optional*) – Focal length of the convex lens. Default np.inf.
- **alpha_h** (*float, optional*) – Angle of incidence on the horizontal plane. Default 0.
- **phi_tilt** (*float, optional*) – Angle from the tilt-sensing QPD frame to the yaw-pitch frame. Default 0.
- **phi_len** (*float, optional*) – Angle from the length-sensing QPD frame to the yaw-pitch frame. Default 0.
- **delta_x** (*float, optional*) – Horizontal miscentering of the beam spot at the optics plane. Default 0.
- **delta_y** (*float, optional*) – Vertical miscentering of the beam spot at the optics plane. Default 0.
- **delta_d** (*float, optional*) – Misplacement of the length-sensing QPD. Default 0.
- **format** (*str, optional*) – Format of the sensing matrix. Choose from

  "OPLEV2EUL": Default sensing matrix from KAGRA MEDM screen
  with input (TILT_PIT, TILT_YAW, LEN_PIT, LEN_YAW), and output (longitudinal, pitch and yaw).

  "xy": Matrix as shown in [1]_.
- **coupling_matrix** (*array, optional*) – The coupling matrix. Default None.
- ***args** – Variable length arguments passed to OpticalLeverSensingMatrix.
- ****kwargs** – Keyword arguments passed to OpticalLeverSensingMatrix.

property delta_d

Misplacement of the length-sensing QPD.

property f

Focal length of the convex lens.

property r

Lever arm.

property r_lens

Lever arm from the optics to the convex lens.

kontrol.sensact.optical_lever.**c_align**(*r_h*, *r_v*, *alpha_h*, *alpha_v*, *r_lens_h=0*, *r_lens_v=0*, *d_h=0*, *d_v=0*, *f_h=inf*, *f_v=inf*, *roundoff=6*)

Return optical lever sensing matrix for a perfectly aligned case.

Parameters

- **r_h** (*float*) – Lever arm from the optics to the tilt-sensing QPD plane on the horizontal plane (amplifying yaw).

- **r_v** (*float*) – Lever arm from the optics to the tilt-sensing QPD plane on the vertical plane (amplifying pitch).
- **alpha_h** (*float*) – Angle of incidence on the horizontal plane.
- **alpha_v** (*float*) – Angle of incidence on the vertical plane.
- **r_lens_h** (*float, optional*) – Lever arm from optics to the lens on the horizontal plane. Defaults None. Specify if it exists.
- **r_lens_v** (*float, optional*) – Lever arm from optics to the lens on the vertical plane. Defaults None. Specify if it exists.
- **d_h** (*float, optional*) – Horizontal distance from the lens to the length-sensing QPD. Defaults None.
- **d_v** (*float, optional*) – Vertical distance from the lens to the length-sensing QPD. Defaults None.
- **f_h** (*float, optional,*) – Focal length of the lens projected to the horizontal plane. Defaults `np.inf` (no lens).
- **f_v** (*float, optional,*) – Focal length of the lens projected to the vertical plane. Defaults `np.inf` (no lens).
- **roundoff** (*int, optional*) – How many decimal places to keep.

Returns
The aligned optical lever sensing matrix

Return type
array

Notes

See $\mathbf{C}_{\text{align}}$ from [1]_.

If *f_h* or *f_v* or (*d_h* and *d_v*) or (*r_lens_h* and *r_lens_v*) are not specified, then we assume no length-sensing QPD so first column, third row, and forth row will be 0. elif *f_h* and *f_v* is specified, (*d_h* or *r_lens_h*) are not specified, then third row will be 0. elif *f_h* and *f_v* is specified, (*d_v* or *r_lens_v*) are not specified, then forth row will be 0.

References

kontrol.sensact.optical_lever.**c_miscenter**(*delta_x*, *delta_y*)

Returns the matrix for correcting miscentered optical lever beam.

Parameters

- **delta_x** (*float*) – Horizontal miscentering of the beam spot at the optics plane.
- **delta_y** (*float*) – Vertical miscentering of the beam spot at the optics plane.

Returns
The miscentering correction matrix.

Return type
array

kontrol.sensact.optical_lever.**c_rotation**(*phi_tilt*, *phi_len*)

Returns the rotation transformation matrix for tilt and length QPDs.

Parameters

- **phi_tilt** (*float*) – Angle from the tilt-sensing QPD frame to the yaw-pitch frame.
- **phi_len** (*float*) – Angle from the length-sensing QPD frame to the yaw-pitch frame.

Returns

Matrix that transform the QPD frames to the yaw-pitch frame.

Return type

array

Notes

See $\mathbf{C}_{\text{rotation}}$ from [1]_

References

kontrol.sensact.calibration

Calibration library for calibrating sensors from sensors measurements

kontrol.sensact.calibration.**calibrate**(*xdata*, *ydata*, *method='linear'*, ***kwargs*)

Fit the measurement data and returns the slope of the fit.

Parameters

- **xdata** (*array*) – The independent variable, e.g. the displacement of the sensor.
- **ydata** (*array*) – The dependent variable, e.g. the sensor readout.
- **method** (*str, optional*) –

  The method of the fit.
  Choose from ["linear", "erf"].

  - "linear": use `kontrol.sensact.calibration.calibrate_linear()`
  - "erf": use `kontrol.sensact.calibration.calibrate_erf()`

- ****kwargs** – Keyword arguments passed to the calibration methods.

Returns

- **slope** (*float*) – The slope of the fitted straight line.
- **intercept** (*float*) – The y-intercept of the fitted straight line.
- **linear_range** (*float, optional*) – The range of y where the sensor considered linear.

kontrol.sensact.calibration.**calibrate_erf**(*xdata*, *ydata*, *nonlinearity=5*, *return_linear_range=False*, *return_model=False*)

Fit an error function and return the 1st-order slope and intercept.

Parameters

- **xdata** (*array*) – The independent variable, e.g. the displacement of the sensor.

- **ydata** (*array*) – The dependent variable, e.g. the sensor readout.
- **nonlinearity** (*float, optional*) – The specification of non-linearity (%). Defined by the maximum deviation of the full_range. Default 5.
- **return_linear_range** (*boolean, optional*) – If True, return the linear range of ydata.
- **return_model** (*boolean, optional*) – Return a kontrol.curvefit.model.Model object with the fitted parameters.

Returns

- **slope** (*float*) – The slope of the fitted straight line.
- **intercept** (*float*) – The y-intercept of the fitted straight line.
- **linear_range** (*float, optional*) – The range of x where the sensor considered linear.

Notes

Credits to Kouseki Miyo, the inventor of this method.

We fit the following function

$$f(x; a, m, x_0, y_0) = a\,\mathrm{erf}(m(x - x_0)) + y_0$$

where a, m, x_0, y_0 are some parameters to be found. The $\mathrm{erf}(x)$ function is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x^2}\, dx\,.$$

If we taylor expand the exponential function and take the first-order approximation of the $\mathrm{erf}(x)$ function, we get

$$\mathrm{erf}(x) \approx \frac{2am}{\sqrt{\pi}}(x - x_0) + y_0\,.$$

So the (inverse) calibration factor is $\frac{2am}{\sqrt{\pi}}$, and the y-intercept is $\frac{2am}{\sqrt{\pi}}x_0 + y_0$.

kontrol.sensact.calibration.**calibrate_linear**(*xdata*, *ydata*, *nonlinearity=5*, *full_range=None*, *start_index=None*, *return_linear_range=False*, *return_model=False*)

Fit a straight line to the data and returns the slope and intercept

This functions recursively fits a straight line to chosen data points and terminates when no linear data point remains. It starts from 3 points closest to the middle of the full-range. After fitting a straight line to the 3 points, other data points that are within the linearity specification will be included to the dataset and this dataset will be use for the next fit. The process repeats until no data points can be added anymore.

Parameters

- **xdata** (*array*) – The independent variable, e.g. the displacement of the sensor.
- **ydata** (*array*) – The dependent variable, e.g. the sensor readout.
- **nonlinearity** (*float, optional*) – The specification of non-linearity (%). Defined by the maximum deviation of the full_range. Default 5.
- **full_range** (*float, optional*) – The full output range of the sensor. If not specified, it will be chosen to be `max(ydata) - min(ydata)`
- **start_index** (*int, optional*) – The index of the data point to start with. If not specified, it will be chosen to be index of the `ydata` closest to `(max(ydata) + min(ydata))/2`.

- **return_linear_range** (*boolean, optional*) – If True, return the linear range of ydata.
- **return_model** (*boolean, optional*) – Return a kontrol.curvefit.model.Model object with the fitted parameters.

Returns

- **slope** (*float*) – The slope of the fitted straight line.
- **intercept** (*float*) – The y-intercept of the fitted straight line.
- **linear_range** (*float, optional*) – The range of x where the sensor considered linear.
- **model** (*kontrol.curvefit.model.Model*) – The fitted model.

kontrol.transfer_function

Primary modules

kontrol.transfer_function.transfer_function

Transfer function class. Wrapper around control.TransferFunction to provide custom functionality related to KAGRA.

class kontrol.transfer_function.transfer_function.**TransferFunction**(**args*)

Bases: TransferFunction

Transfer function class

Parameters

***args** – Arguments passed to control.TransferFunction class.

clean(*tol_order=5*)

Remove numerator/denominator coefficients that are small outliers

Parameters

tol_order (*int, optional*) – If the coefficient is tol_order order smaller than the rest of the coefficients, then this coefficient is an outlier. Defaults 5.

foton(*expression='zpk'*, *root_location='s'*, *significant_figures=6*, *itol=1e-25*, *epsilon=1e-25*)

Foton expression of this transfer function

Calls kontrol.core.foton.tf2foton and returns a foton expression of this transfer function

Parameters

- **expression** (*str, optional*) – Format of the foton expression. Choose from ["zpk", "rpoly"]. Defaults to "zpk".
- **root_location** (*str, optional*) – Root location of the zeros and poles for expression=="zpk". Choose from ["s", "f", "n"]. "s": roots in s-plane, i.e. zpk([...], [...], ..., "s"). "f": roots in frequency plane, i.e. zpk([...], [„„], ..., "f"). "n": roots in frequency plane but negated and gains are normalized, i.e. real parts are positive zpk([...], [...], ..., "n"). Defaults to "s".
- **significant_figures** (*int, optional*) – Number of significant figures to print out. Defaults to 6.
- **itol** (*float, optional*) – Treating complex roots as real roots if the ratio of the imaginary part and the real part is smaller than this tolerance Defaults to 1e-25.

- **epsilon** (*float, optional*) – Small number to add to denominator to prevent division error. Defaults to 1e-25.

Returns
foton_expression – The foton expression in selected format.

Return type
str

lstrip(*element*, *fc=None*)

Remove zero or pole from the left.

Parameters

- **element** (*str*) – Element to be removed. Choose from ["any", "zero", "pole", "pair", "all"].
- **fc** (*float or None, optional*) – Cutoff frequency. If element is in ["any", "all"], remove any or all elements on the left of the cutoff. Defaults None.

rstrip(*element*, *fc=None*)

Remove zero or pole from the right.

Parameters

- **element** (*str*) – Element to be removed. Choose from ["any", "zero", "pole", "pair", "all"].
- **fc** (*float or None, optional*) – Cutoff frequency. If element is in ["any", "all"], remove any or all elements on the left of the cutoff. Defaults None.

save(*path*, *overwrite=True*)

Save the transfer function to a specified path.

This functions extracts the numerator and denominator coefficients and puts it into a pandas DataFrame with keys {"num" and "den"}. Then it outputs to the specified path with a pickle format.

Parameters

- **path** (*str*) – The string of the path.
- **overwrite** (*boolean, optional*) – Overwrite if the file exists.

stabilize()

Convert unstable zeros and poles to stable ones.

kontrol.transfer_function.transfer_function.**load_transfer_function**(*path*)

Load a kontrol TransferFunction object from path.

Parameters
path (*str*) – The path of the saved transfer function

Returns
The loaded TransferFunction object.

Return type
TransferFunction

kontrol.transfer_function.notch

Foton defined Notch filter.

class kontrol.transfer_function.notch.**Notch**(*frequency*, *q*, *depth=None*, *depth_db=None*)

Bases: *TransferFunction*

Notch Filter Object

Parameters

- **frequency** (*float*) – The notch frequency (Hz).
- **q** (*float*) – The quality factor.
- **depth** (*float*) – The depth of the notch filter (magnitude). If not specified, depth_db will be used. Defaults None.
- **depth_db** (*float, optional*) – The depth of the notch filter (decibel). If not specified, depth will be used instead. Defaults None.
- **Attritubes** –
- **----------** –
- **frequency** – The notch frequency (Hz).
- **q** – The quality factor.
- **depth** – The depth of the notch filter (magnitude).

Notes

The notch filter is defined by Foton, as

$$N(s) = \frac{s^2 + (2\pi f_n)/(dQ/2)s + (2\pi f_n)^2}{s^2 + (2\pi f_n)/(Q/2)s + (2\pi f_n)^2},$$

where f_n is the notch frequency, q is the quality factor , and d is the depth.

property depth

The depth of the notch filter (magnitude).

foton()

Foton expression of this notch filter.

property frequency

The notch frequency (Hz).

property q

The quality factor

1.4.2 Submodules

kontrol.ezca

1.5 Contact

Author
Terrence Tak Lun Tsang

E-mail addresses
tsangtt@cardiff.ac.uk (University contact) (Preferred)

terrence.tsang@ligo.org (LIGO contact)

ttltsang@link.cuhk.edu.hk (KAGRA contact)

astrotec@connect.hku.hk

terrencetec@gmail.com

1.6 For Developers

1.6.1 Standards and Tools

Please comply with the following standards/guides as much as possible.

Coding style

- **PEP 8**: https://www.python.org/dev/peps/pep-0008/

CHANGELOG

- **Keep a Changelog**: https://keepachangelog.com/en/1.0.0/

Versioning

- **Semantic Versioning**: https://semver.org/spec/v2.0.0.html

Packaging

- **PyPA**: https://www.pypa.io
- **python-packaging**: https://python-packaging.readthedocs.io

Documentation

- **NumPy docstrings**: https://numpydoc.readthedocs.io/en/latest/format.html
- **Sphinx**: https://www.sphinx-doc.org/
- **Read The Docs**: https://readthedocs.org/
- **Documenting Python Code: A Complete Guide**: https://realpython.com/documenting-python-code/

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z