Kontrol: KAGRA Control Python Library for VIS Control Systems

Tsang Terrence Tak Lun
The Chinese University of Hong Kong

June 20, 2020

1 Introduction

Kontrol is a python library for KAGRA control systems. The name "Kontrol" is still pronounced as "control" but started with a "K" which comes from the "KAGRA". The purpose of this library is to cover all control related setup in KAGRA, at least for the VIS subgroup, in hope to standardize KAGRA's control system routine. As we all know, VIS control work in KAGRA was done by various members in the VIS group and the styles of implementation have no unification in any sense because communication, protocols and consents were simply lacking. As a result, some parts of the actual system may appear cryptic to others, which might be very bad for long term. Therefore, we introduce this library as an open platform for us. As the library develops, we hope to standardize KAGRA's control system routine and enhance reproducibility and repeatability which has been lacking in some KAGRA control related systems. More importantly, we hope to improve communication and encourage discussion between colleagues through openness. Nevertheless, this library encourages automation so tedious work can be handled by the computer.

The library should cover a wide range of topics in VIS related controls, including sensor and actuator diagonalization, system identification, filter design, controller synthesis and so on, essentially, everthing related to setting up the control system in KAGRA. For now, there are mainly three functions, namely, actuator diagonalization, finding sensor correction gain, and complementary filter optimization. Everything I do to the control system in KAGRA in the future will be implemented as methods in this library as time goes on. I strongly encourage related experts to co-develop this library since it will be very important to share information and techniques as KAGRA proceeds to years long operations while people come and go.

This document is not an official documentation of this library. This is just an introduction to the library serving the purpose of announcing of the existence of this library. This library was initialized under the context that the signal recycling mirrors need upgrades so the functions available do exactly what needs to be done for part of the upgrades. This document has instructions on how to install the library and how to use it in KAGRA workstations. Main functions are briefly introduced along with the theoretical concept behind. Examples are provided at the end of each introduction. Similar versions of the examples can be found in /kontrol/test/.

- Documentation: https://kontrol.readthedocs.io
- GitHub repository: https://github.com/terrencetec/kontrol.git

2 How to Install Kontrol

2.1 The Repository

If you have git installed (git is available in KAGRA workstation.), you can clone the repository directly. Move to a directory where you want to clone the repository, then type

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

To update the content, in the /kontrol directory, type

```
$ git pull
```

2.2 Virtual Environment

If you are working on KAGRA workstations, make sure to activate python virtual environment before installing any Python packages. This will separate virtual environment packages with the global packages which is used by many KAGRA software such as guardian, see http://klog.icrr.u-tokyo.ac.jp/osl/?r=10756 for why we should do this.

Some workstations (k1ctr1, k1ctr2, k1ctr7) have Conda installed so we can use conda virtual environment conveniently. To create a virtual environment, type

```
$ conda create -n [virtual-environment-name] python=3.6
```

Replace [virtual-environment-name] with your virtual environment name. I was using Python 3.6 while I was developing the library, so I recommend to specify the python version as well. Though, there should be no problem using this library for Python 3.6+.

To show available virtual environments, type

```
$ conda info -e
```

Then, activate the virtual environment, type

```
$ conda activate [virtual-environment-name]
```

To, activate the virtual environment, type

```
$ conda deactivate
```

If you wish to remove the virtual environment, type

```
$ conda env remove -n [virtual-environment-name]
```

2.3 Library Dependencies

After that, install the library dependencies via conda install.

- numpy
- scipy
- matplotlib
- control
- ezca

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

This will install the Python packages along with their dependency requirements.

2.4 Installing Kontrol

Now, we will use pip to install the Kontrol library in the Conda virtual environment. I am not sure if this is the best practice, but this confirmed to work. But, first, make sure that the virtual environment is activated. There should be a bracket with the virtual environment name on the left of the command line prompt. Like

```
(my−virtual−environment) controls@k1ctr7$ _
```

After that, type "which pip" in the command line

```
(my−virtual−environment) controls@k1ctr7$ which pip
/home/controls/anaconda3/envs/my−virtual−environment/bin/pip
```

to confirm that we are using the local pip, but not the global one so not to mess up with global python libraries.[1] Fix it if the command returns /usr/local/bin/pip.

Then, navigate to /kontrol, where you see the file setup.py. Type

```
$ pip install   .
```

Then the Kontrol package should be visible from your virtual environment. We don't recommend using pip in Conda environment. For installing other packages, use Conda install instead.

2.5 Alternative Virtual Environment (Not recommended and probably not work on k1ctr workstations unless the DGS is updated)

In case Conda is not available. We can use Python's venv to create virtual environment.[2] Navigate to where you want to place the virtual environment directory, then type

```
$ python3 −m venv −−without−pip [virtual−environment−name]
```

Since Ezca is not cannot be installed via pip, we have to rely on the global Ezca package. To do so, we can make global packages visible from the virtual environment by setting

```
include−system−site−packages = True
```

in the pyvenv.cfg file in the virtual environment folder. To activate , type

```
$ source [virtual−environment−name]/bin/activate
```

Then, just follow section 2.4 to install Kontrol. To deactivate, just type

```
$ deactivate
```

[1] Thanks to Kouseki Miyo for pointing this out.
[2] Thanks to Takahiro Yamamoto for fixing the no pip3 problem in k1ctr.

3 Main Function Explanations

3.1 Actuator Diagonalization

3.1.1 Concept

This function applies DC offsets to different degrees of freedom in a stage separately and measures the displacement offsets to calculate the actuation coupling matrix **C** defined as follows.

$$\mathbf{X} = \mathbf{CA} \tag{1}$$

Where repeated index implies summation, **X** is the change in displacements, **C** is an invertible square matrix with the size being the number of DoFs, and **A** is the change in actuation signals. For example, in the case of an inverted pendulum, $\mathbf{X} = [\Delta L\ \Delta T\ \Delta Y]^T$, i.e. a vector defined by the change in longitudinal displacement, transverse displacement and yaw angular displacement, **A** is then a vector defined by the change in actuation signals in the three directions in the digital system, and **C** is a 3-by-3 matrix. For systems without acutation coupling, **C** is a diagonal matrix with the entries being the transfer function gains in a particular frequency, typically at 0 Hz. But in general, there will be non-diagonal elements and the goal of actuation diagonalization is to find **C** so we can replace **C** by $\mathbf{CC}^{-1}\mathbf{D}$, where **D** the desired diagonal matrix. We can replace **D** by $\mathrm{diag}(C_{11}, C_{22}, ..., C_{NN})$, where C_{ii} is the i^{th} diagonal element of **C**, if we started with a roughly diagonalized old system which requires re-diagonalization. This will preserve the "strength" of the actuation signal so further transfer functions measured will be similar to the old ones.

The function will apply DC offsets to DoFs of interest and measure the change in displacement to calculate individual elements of the coupling matrix. Then, it will return the "EUL2COIL" matrix which can be entered via the MEDM screen. The new EUL2COIL matrix is defined as follows.

$$\mathrm{EUL2COIL}_{\mathrm{new}} = \mathrm{EUL2COIL}_{\mathrm{old}}\mathbf{C}^{-1}\mathrm{diag}(C_{11}, C_{22}, ..., C_{NN}). \tag{2}$$

3.1.2 Example

The following example is going to attempt diagonalizing the actuators in the IP stage of the BS. It is using fakeezca as ezca so the script will not interact with the real-time system. One at a time, it will act, with a ramp time of 0.1 second, an _additional force of 1000, 2000 and 3000, to longitudinal, transverse and yaw direction "K1:VIS-BS_IP_TEST_[L,T,Y]_OFFSET" and return to the original force before acting another. After applying the force offset, it will measure the 1 second time-averaged displacements from "K1:VIS-BS_IP_DAMP_[L,T,Y]_INMON". These channels and numbers can be specified individually to method if necessary.

```
from kontrol import visutils
from kontrol import fakeezca as ezca
# import ezca # Alternatively  import fakeezca as ezca for testing.

BS = visutils.Vis('BS', ezca) # Define a visutils.Vis  object with optic\
    # name and the ezca module.

stage = 'IP'

dofs = ['L', 'T', 'Y'] # Make sure the order is the same as appeared\
    # in the real-time system.

force =[1000, 2000, 3000] # Actuate 1000 counts in longitudinal,\
    # 2000 counts in transverse and 3000 counts in yaw. Do specify\
    # this or else the program will default actuations to 1000 counts.

no_of_coils = 3 # Optional. Determined by the EUL2COIL matrix if\
    # not specified.

t_ramp = 0.1 # For testing pupose, we put a small number.\
    # In practice, this should be around 10 seconds or more.

t_avg = 1 # Again, this is for test only. Put a reasonable number
# when using this.

EUL2COIL_new = BS.actuator_diag(stage, dofs, act_block='TEST',
    act_suffix='OFFSET', sense_block='DAMP', sense_suffix='INMON',
    matrix='EUL2COIL', force=force, no_of_coils=no_of_coils,
    update_matrix=False, t_ramp=t_ramp, t_avg=t_avg, dt=1/8)

print(EUL2COIL_new)
```

3.2 Find Sensor Correction Gain

3.2.1 Concept

Consider the following block diagram of a simple "RDN"[3] control model. The output, written as the superposition

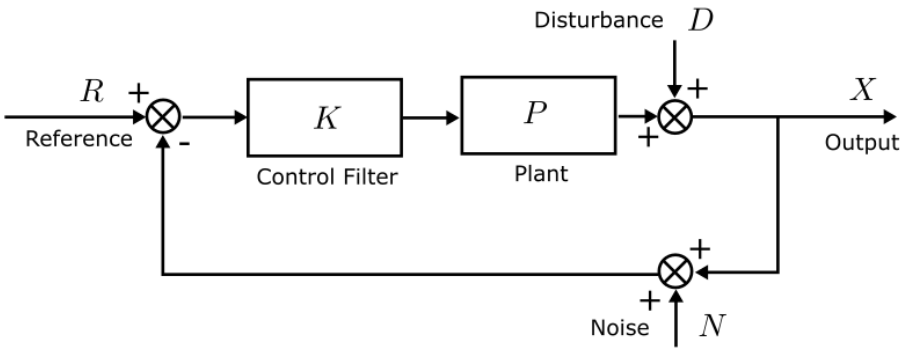

Figure 1: RND control model

of the reference, disturbance and noise, is given by

$$X = \frac{KP}{1+KP} R + \frac{1}{1+KP} D - \frac{KP}{1+KP} N. \tag{3}$$

The fluctuation of the output is then

$$\langle X^2 \rangle = \left| \frac{1}{1+KP} \right|^2 \langle D^2 \rangle + \left| \frac{KP}{1+KP} \right|^2 \langle N^2 \rangle . \tag{4}$$

At the limit $|KP| \gg 1$, the fluctuation of the output becomes $\langle X^2 \rangle \approx \langle N^2 \rangle$. Which means that the output becomes the noise. Coming back to the conversation of sensor correction at the preisolator stage, the control model of interest is given in Fig. 2.

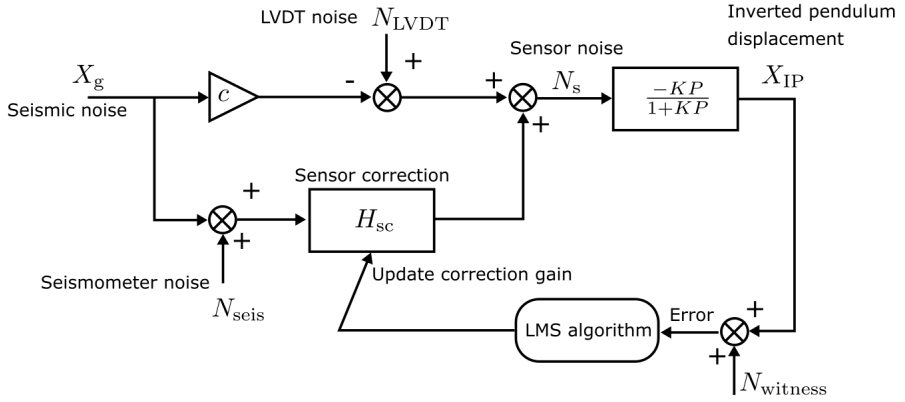

Figure 2: Sensor correction scheme in the preisolator stage. An LMS algorithm can be used to find the sensor correction gain to minimize the intercalibration mistmatch by minimizing the error signal defined by the inverted pendulum displacement.

At the high gain limit, the inverted pendulum displacement reads

$$X_{\mathrm{IP}} = N_{\mathrm{s}} = (H_{\mathrm{sc}} - c)\, X_{\mathrm{g}} + N_{\mathrm{LVDT}} + H_{\mathrm{sc}} N_{\mathrm{seis}} \tag{5}$$

[3] I claim that I coined the term "RDN" , which stands for reference input, disturbance, and noise. This model has been used allover the control theory community but no one has ever conveniently named it.

Here, c is a coupling constant to the LVDT noise since the LVDT measures differential displacement between the preisolator plateform and the ground. c is an unknown, but if there is no inter-calibration mismatch between the LVDT and the seismometer, then c is one. However, there will be some calibration difference in general, so the goal of this function is to find the correct sensor correction gain to match the seismic noise coupling c.

When the ground motion dominates, then the closed-loop displacement reads $X_{\mathrm{IP}} = (H_{\mathrm{sc}} - c)X_{\mathrm{g}}$. It follows that minimizing X_{IP} automatically minimizes the difference between the sensor correction filter H_{sc} and the coupling constant c, which is what we desire in a sensor correction scheme. Therefore, we can use an adaptive sensor correction filter and an LMS algorithm to update the gain of the filter. The LMS algorithm takes an error signal and an input, and will minimize the mean square of the error signal (fluctuation). In this case, the input signal is the ground noise. We can use the seismometer readout as the input. However, we cannot use the corrected LVDT as the witness sensor measuring the error signal. This is because the error will become zero anyway. Note the minus sign in Eqn. 3, the error signal will then becomes $e = X_{\mathrm{IP}} + N_{\mathrm{witness}} = -N_{\mathrm{s}} + N_{\mathrm{s}} = 0$. A perfect candidate of the witness sensor would be the inertial sensors on the preisolator. In this case, the error signal reads $e = -N_{\mathrm{s}} + N_{\mathrm{witness}} = (H_{\mathrm{sc}} - c)X_{\mathrm{g}} + N_{\mathrm{inertial}}$. So the mean square is

$$\left\langle e^2 \right\rangle = |H_{\mathrm{sc}} - c|^2 \left\langle X_{\mathrm{g}}^2 \right\rangle + \left\langle N_{\mathrm{inertial}}^2 \right\rangle \tag{6}$$

and it follows that minimizing the mean square error means matching the sensor correction gain to the seismic noise coupling. However, it is important to note that the input and error must have a zero mean value. This means the signals must be high-passed before using. Also, a basic assumption of this approach is that the seismic noise is high. So, the performance of this method is better when the seismic noise is high.

3.3 Example

The following script will automatically correction the sensor correction gain in the channel "K1:VIS-BS_IP_SENSCORR_L_GAIN" using the "K1:VIS-BS_IP_SENSCORR_L_INMON" as the LMS filter input and "K1:VIS-BS_IP_BLEND_ACCL_OUT16" as the error signal. This particular example uses fake Ezca instead of the real Ezca, so nothing will happen to the real system. One important note here is that we ought not to set the integration time to be too high. This is to avoid the low frequency sensor noise induced from the sensor correction filter from affecting the optimization.

```
from kontrol import visutils
from kontrol import fakeezca as ezca
import kontrol
import matplotlib.pyplot as plt

BS = visutils.Vis('BS', ezca)

rms_threshold = 0.01 # The adaptive loop terminates when the RMS of the
    # adaptive gain is less than 0.01. This corresponds to 1% of
    # inter calibration mismatch.

t_int = 10 # The integration time for calculating the RMS.

update_law = kontrol.unsorted.nlms_update # Normalized LMS algorithm
    # Normal LMS algorithm is also avaiable in kontrol.unsorted, but is less
    # robust.

reducing_lms_step = True # If True, then the step size of the LMS will
    # be reduced by a factor of reduction ratio when the mean square error
    # is higher or equal to previous iterations. This leads to better
    # convergence of the sensor correction gain.

timeout = 20 # The loop will terminate regardless of the convergence when
    # the algorithm has been running for 20 seconds. Set to, say 300 when
    # using it in the real system.

ts, gains, inputs, errors = BS.find_sensor_correction_gain(
    gain_channel='IP_SENSCORR_L_GAIN',
    input_channel='IP_SENSCORR_L_INMON',
    error_channel='IP_BLEND_ACCL_OUT16',
    rms_threshold=rms_threshold, t_int=t_int, dt=1/8, update_law=update_law,
    step_size=0.5, step_size_limits=(1e-3, 1),
    reducing_lms_step=reducing_lms_step,
    reduction_ratio=0.99, timeout=timeout)

plt.subplot(211)
plt.plot(ts, gains, label='Gain')
plt.legend(loc=0)
plt.subplot(212)
plt.plot(ts, errors, label='Error')
plt.legend(loc=0)
plt.show()
```

3.4 Complementary Filter Optimnization

3.4.1 Concept

Let a complementary filter be a function of coefficients a_i. The low-pass and high-pass filter then reads

$$L = L(a_1, a_2, ..., a_M), \tag{7}$$

$$H = H(a_1, a_2, ..., a_M) = 1 - L, \tag{8}$$

where M is the number of parameters that is required to define the filter.

If the noise content is filtered by the low-pass filter is N_L and the noise content that is filtered high-pass filter is N_H, then the over noise reads $N = N(\mathbf{a}) = L\,N_L + H\,N_H$. But default, this function finds the parameters a_i such that the 2-norm (expected RMS/integrated RMS) of the overall noise is minimized, i.e. finding

$$\mathbf{a}_{\text{opt}} = \underset{\mathbf{a}\in\mathbb{R}^M}{\arg\min} \left[\int_0^\infty \left\langle N(\mathbf{a})^2 \right\rangle \, df \right], \tag{9}$$

where $\mathbf{a}$ is the vector with the parameters as entries and $\mathbf{a}_{\text{opt}}$ is the vector with optimized parameters.

The function supports complementary filter with more than 2 filters as well, so long as the specified filters are well defined and normalized.

The user has an option to select how the filter is optimized, locally and globally, depending on the situation. For local optimization, an initial guess of the parameters should be specified. As for global optimization, the bounds for the parameters should be specified. This can be tricky for some cases. So, meaningful parameters, such as blending frequency, are strongly encouraged when designing the filter. Also, global optimization can sometimes lead to failed results, e.g. one of the parameters reaches the bound. So, specifying the bounds carefully is also very important.

3.4.2 Example

The following example will optimize the complementary filter proposed by Sekiguchi to minimize the overall noise a simulated seismic noise with a peak at 2 Hz, a simulated LVDT white noise with and a simulated geophone noise.

```
from kontrol.utils import quad_sum
from kontrol.filter import complemetary_sekiguchi
from kontrol.optimize import optimize_complementary_filter
import numpy as np
import matplotlib.pyplot as plt
from control import *

f = np.linspace(1e-1,1e2,1000) # Frequency axis
noise_low_pass1 = np.ones_like(f)  # LVDT-like noise
noise_low_pass2_tf  = tf([(2*np.pi*2)**2],[1,2*np.pi*2/10,(2*np.pi*2)**2])
noise_low_pass2 = abs(noise_low_pass2_tf.horner(2*np.pi*1j*f)[0][0])    # Seismic-like noise
noise_low_pass = quad_sum(noise_low_pass1, noise_low_pass2)
noise_high_pass = 1/f**3.5 # Geophone-like noise
plt.figure()
plt.loglog(f,   noise_low_pass,  label='Noise_to_be_low-passed')
plt.loglog(f,   noise_high_pass,  label='Noise_to_be_high-passed')
plt.legend(loc=0)
plt.xlabel("Frequency")

# Using sekiguchi's filter
complementary_filter = complementary_sekiguchi
result  = optimize_complementary_filter(filter_=complementary_filter,
    bounds=[(2*np.pi*min(f), 2*np.pi*max(f))],
    spectra=[noise_low_pass, noise_high_pass],
    f=f,
    )

plt.figure()

lpf_opt,  hpf_opt = complementary_filter(result.x)  # In\
    # scipy.optimize.OptimizeResult, x is the attribute of be optimized parameters.
plt.subplot(231)
plt.loglog(f,   abs(lpf_opt.horner(2*np.pi*1j*f)[0][0]),    label='Complementary_low-pass')
plt.loglog(f,   abs(hpf_opt.horner(2*np.pi*1j*f)[0][0]),    label='Complementary_high-pass')
plt.legend(loc=0)
plt.xlabel("Frequency")
plt.title("1:_Optimized_Sekiguchi_filter")
plt.grid(True)
```