1 LLRFLibsPy Documentation

This is a draft of the documentation of the python library LLRFLibsPy. Later we will convert this document into Latex.

In this document, we will cover the following topics:

- What is LLRFLibsPy and how does it fit a larger framework standardizing the LLRF software (e.g., high-level applications, test software and automation)?
- Internal architecture of the LLRFLibsPy
- How LLRFLibsPy can be obtained and installed
- Explain each function in the library, including the description, interfaces, algorithms (formula) and the scenarios of usage
- Also include the table that we have in the README mapping the function and the examples

1.1 Introduction

After years' development of digital LLRF systems, their architecture are now quite mature with similar patterns. Though pushing the RF field stability to extreme and applying advanced algorithms like modern control algorithms and machine learning are still guiding the R&D, standardization of the LLRF hardware, firmware and software has been started. DESY is in advance to standardize the LLRF hardware based on the MicroTCA platform. Some basic firmware/software libraries have been developed at DESY and PSI:

- PSI firmware/software library:

  PSI Common/PSI Fix: (firmware framework and algorithm libraries)

  https://github.com/paulscherrerinstitute/psi_fpga_all,
  https://github.com/paulscherrerinstitute/psi_common,
  https://github.com/paulscherrerinstitute/psi_fix

  LLRFLibs: (LLRF algorithm library implemented in C)

  https://git.psi.ch/GFA/RF/Libraries/llrflibs

  LLRFLibsPy: (LLRF algorithm library implemented in Python)

  https://git.psi.ch/GFA/RF/Libraries/llrflibspy

  ooEpics: (C++ framework for EPICS module development)

  https://git.psi.ch/GFA/RF/Libraries/ooepics

  ooPye: (Python framework for soft IOC development)

  https://git.psi.ch/GFA/RF/Libraries/ooPye

- DESY firmware / software library:

  FWK: https://gitlab.desy.de/fpgafw/fwk (firmware framework library, documentation can be found at https://fpgafw.pages.desy.de/docs-pub/fwk/index.html).

  ChimeraTK: https://github.com/ChimeraTK (software framework library)

These libraries are used either to build the architecture, framework and infrastructure of the firmware/software or to implement domain algorithms related to LLRF and accelerator controls. To understand the motivations to develop these libraries, let us first have a look at the roles and locations of these libraries within the LLRF system.

A general logical (functional) architecture of LLRF systems is depicted in Figure 1. Note that different implementations may map the functional blocks into different physical architecture. For example, the new SwissFEL X-band LLRF system combines the “LO & Clock Generator” and “RF Transceiver” into a single custom chassis, whereas DESY’s MicroTCA system has a separated box for “LO & Clock Generator” and implements the “RF Transceiver” into an RTM board.

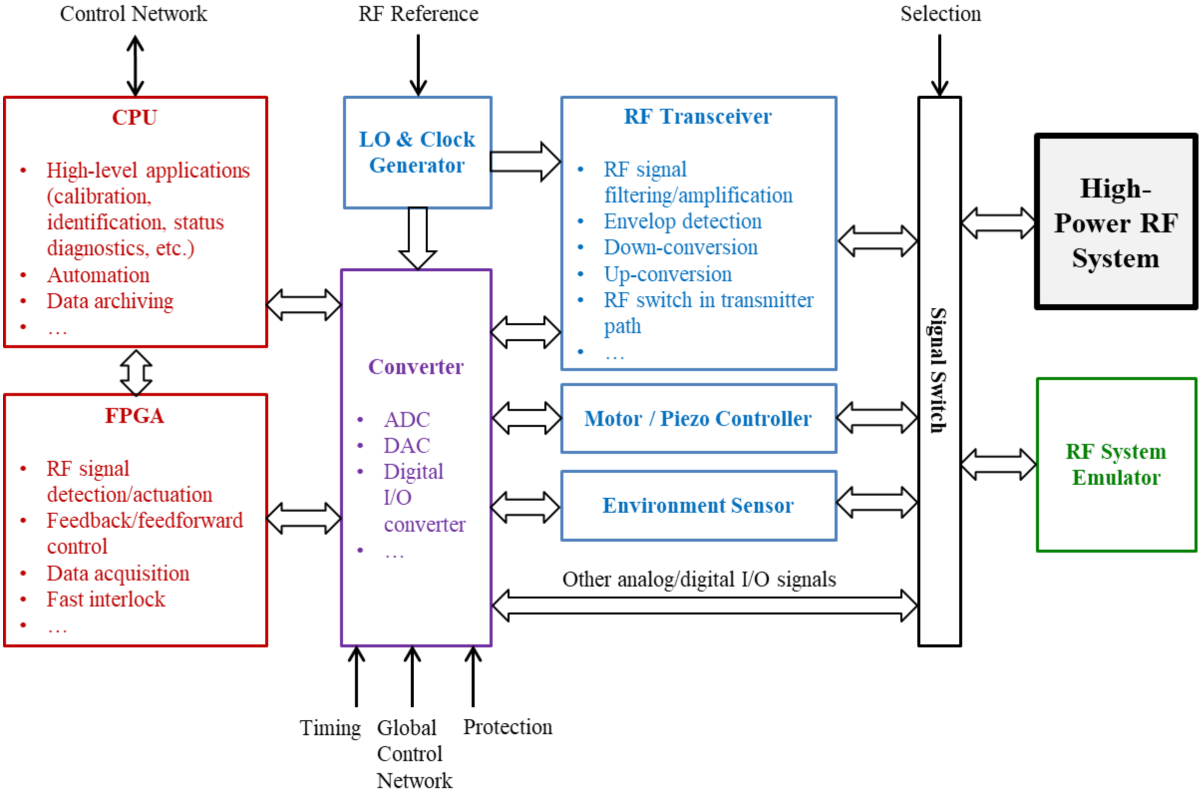

Figure 1: A general functional architecture of a LLRF system

In Figure 1, the FPGA and CPU (a LLRF station may have multiple FPGAs and CPUs) are the digital processing units hosting the firmware and software entities. The logistical (functional) architecture of the firmware and software are depicted in Figure 2, including allocations into the FPGA and CPU.

The firmware architecture are explained as follows:

- We separate the “Platform Support Firmware Modules” and the “Application Firmware Modules”. These two parts can be developed by different teams since different domain knowledge is required. To implement the platform support modules, the developers need to know the detailed protocols and electronic characteristics to access the on-board hardware components, such as the ADCs, DACs, memories and data buses (e.g., PCI express). On the other hand, the application firmware requires domain knowledge for RF signal detection, filtering, feedback control, and so on.

- The “Application Interface” of firmware should be designed as generic as possible to decouple the platform and application modules.

The following firmware libraries can be beneficial for LLRF firmware development:

- ***Firmware Framework Library***

  Such a library helps construct the firmware project structure, simulate the firmware code, and automate the firmware synthesize. Of course, platform support modules for accessing widely used hardware components (e.g., popular ADC chips, DAC chips, PCI express driver) can also be compiled into the library.

- ***Basic Firmware Library***

  This library collects firmware build-up components (e.g., FIFO, filters… as shown in the diagram) that can be used in any firmware implementation.

- ***RF Control Firmware Library***

  This library aims at implementing generic high-level configurable LLRF control modules. These modules can be assembled and configured to derive a working LLRF controller quickly.

These libraries enables fast prototyping of LLRF controllers containing most fundamental features to satisfy most of the RF control requirements. The LLRF developer can make further development based on these libraries to include specific features for solving the particular RF control problems that they are facing.

The software architecture are explained as follows:

- A layered architecture is constructed. The "OS & Drivers" layer is at the bottom that is platform specific. The "Platform Control Interface" is a wrapper to separate the platform and application software, which has the similar role as the "Application Interface" in the firmware.
- The "Low-Level Applications" are usually *soft real time* for fast processing. In most LLRF systems, this layer is thin since most real-time functions can be implemented in the FPGA firmware, leaving this layer small only implementing necessary software-based real-time functions like data streaming, fast (complex) diagnostics/protection, and pulse-to-pulse feedback (for pulsed machine).
- The "High-Level Applications" are non-real-time functions performing the automation of the operation of LLRF system. This part requires lots of LLRF domain knowledge (e.g., cavity model, control theory, signal processing theory, etc.) since the high-level applications need to interpret the raw results from FPGA to values with physical meanings.

The following Software libraries can be beneficial for LLRF software development:

- ***Software Framework Library***

  The library defines a common software architecture and implements the infrastructures for multithreading, hardware access and network communication.

- ***LLRF Algorithm Library (C/C++)***

  This library implements RF control domain algorithms. If written in C/C++ languages, the library can be used in "Low-Level Applications" for fast processing.

- ***LLRF Algorithm Library (Python)***

  This library is suitable to implement "High-Level Applications", including the RF test and conditioning automation software. Python is an excellent language for such applications. In addition to the same algorithms as in the C/C++ libraries, the Python library may also implement the features for noise analysis, feedback controller design, analysis and simulation. With such features, we can further reduce the reliance on Matlab for controller design.

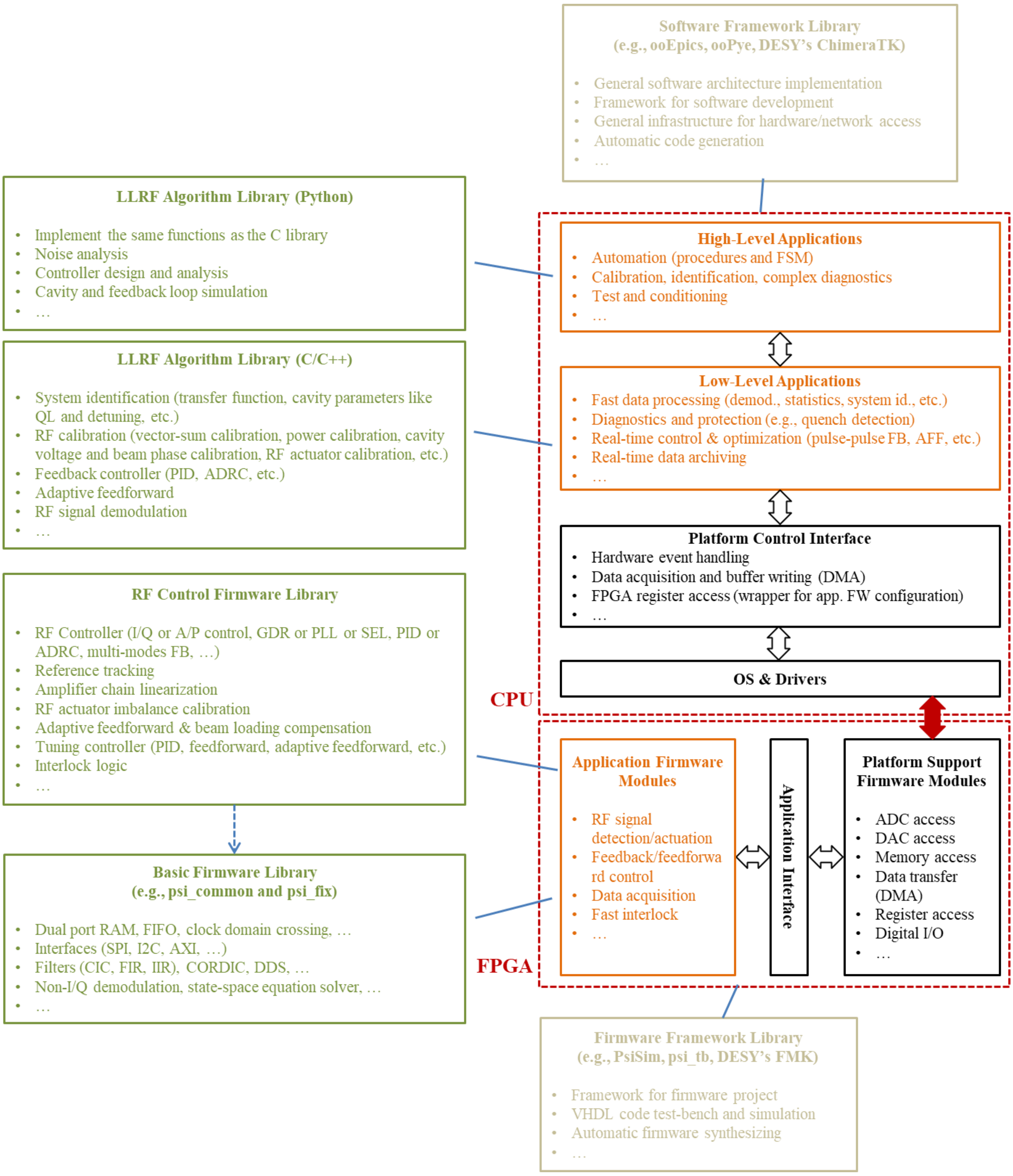

Figure 2: A general functional architecture of the LLRF firmware and software and the libraries

This document describes the architecture, algorithms, and manual for the ***LLRFLibsPy*** library implemented at PSI.

1.2 Installation

To be documented …

1.3 Architecture and Contents

The LLRFLibsPy library consists of several Python modules as shown in Figure 3. The associations between these modules are also shown in the diagram. Each module is a Python source file implementing routines that can be directly called in the user codes. The list of routines and the example codes to test the routines are summarized in Tables 1 to 9.

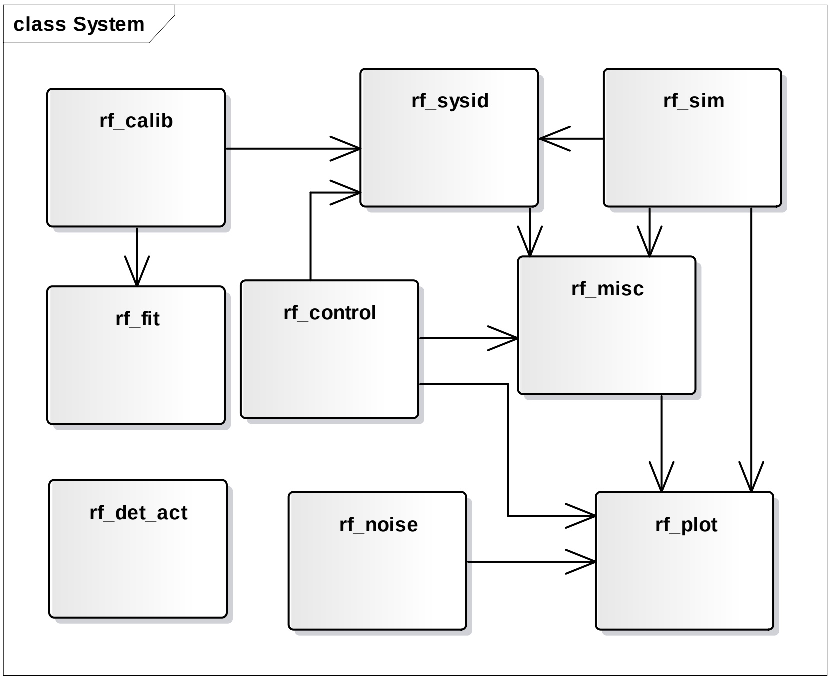

Figure 3: LLRFLibsPy library modules and their associations

Table 1: Module `rf_sysid.py`

Function	Example	Comments
`prbs`	`example_sysid_prbs_etfe.py`	Produce the PRBS signal for system identification
`etfe`	`example_sysid_prbs_etfe.py`	Empirical Transfer Function Estimation (ETFE)
`half_bw_decay`	`example_calib_for_ref_sc_cavity.py`	Calculate the cavity half-bandwidth from the RF pulse decay stage
`detuning_decay`	`example_calib_for_ref_sc_cavity.py`	Calculate the cavity detuning from the RF pulse decay stage
`cav_drv_est`	`example_calib_for_ref_nc_cavity.py`	Estimate the cavity drive and reflected signals from probe signal
`cav_par_pulse`	`example_calib_for_ref_sc_cavity.py` `example_sysid_caveq.py`	Calculate the cavity parameters (half-bandwidth and detuning) within the pulse by directly solving the cavity equation
`cav_par_pulse_obs`	`example_calib_for_ref_sc_cavity.py` `example_sysid_caveq.py`	Calculate the cavity parameters (half-bandwidth and detuning) within the pulse using the ADRC observer
`cav_beam_pulse_obs`	`example_sysid_caveq.py`	Calculate the beam drive voltage within the pulse using the ADRC observer
`cav_observer`		Construct the ADRC observer and estimate the denoised cavity voltage and the general disturbances
`iden_impulse`	`example_aff_ilc.py`	Identify the impulse response of a real/complex SISO system from data
`beta_powers`		Identify the cavity input coupling factor using steady-state forward & reflected powers

Table 2: Module `rf_calib.py`

Function	Example	Comments
calib_vprobe	example_calib_virtual_probe.py	Calibrate cavity virtual probe with the forward & reflected signals
calib_dac_offs	example_calib_dac_offs.py	Calibrate DAC offset for I/Q modulator used in direct upconversion
calib_iqmod	example_calib_iqm.py	Calibrate I/Q modulator imbalance for direct upconversion
calib_cav_for	example_calib_for_ref_nc_cavity.py	Calibrate cavity forward signal using forward and probe
calib_ncav_for_ref	example_calib_for_ref_nc_cavity.py	Calibrate cavity forward & reflected signals with constant QL and detuning
calib_scav_for_ref	example_calib_for_ref_sc_cavity.py	Calibrate cavity forward & reflected signals with time-varying QL and detuning
for_ref_volt2power	example_sim_cavity_basic.py	Calibrate forward & reflected power from forward & reflected voltage
phasing_energy	example_fit_funcs.py (demo fitting)	Calibrate energy gain and beam phase with phase scan and energy measurement
egain_cav_power	example_power_to_vacc.py	Estimate steady-state standing-wave cavity voltage from drive power
egain_cgstr_power	example_power_to_vacc.py	Estimate constant-gradient traveling-wave structure ACC voltage from drive power
calib_vsum_poor		Calibrate vector sum by rotating and scaling all signals referring to a selected channel
calib_sys_gain_pha	example_calib_sys_gain_phase.py	Calibrate system gain and system phase

Table 3: Module `rf_control.py`

Function	Example	Comments
ss_discrete	example_sim_cavity_basic.py	Discretize a continuous state-space system and compare the frequency responses
ss_cascade	example_feedback_analysis.py	Cascade two state-space systems (either continuous or discrete - C/D)
ss_freqresp	example_feedback_analysis.py	Calculate and plot the frequency response of a state-space system (C/D)
basic_rf_controller	example_feedback_basic.py	Derive a basic continuous RF I/Q controller: P + I + frequency notches
control_step	example_feedback_basic.py	Perform one time-step execution of the discretized controller
loop_analysis	example_feedback_analysis.py	Analyze the sensitivity/complementary sensitivity of an RF control loop (C/D)
cav_sp_ff	example_sp_ff.py	Derive the setpoint and feedforward waveforms for desired cavity voltage and beam loading
ADRC_controller	example_feedback_adrc.py	Derive a basic ADRC controller (the observer and gain)
ADRC_control_step	example_feedback_adrc.py	Perform one time-step execution of the discretized controller including the ADRC observer
AFF_timerev_lpf	example_aff_timerev_lpf.py	Time-reversed low pass filter-based adaptive feedforward
AFF_ilc_design	example_aff_ilc.py	Derive the ILC gain matrix from the impulse response and weighting matrices
AFF_ilc	example_aff_ilc.py	Apply the ILC algorithm to calculate

		the feedforward correction signal
resp_inv_svd	example_resp_matrix_inv.py	Response matrix inversion with SVD (with singular value filtering)
resp_inv_lsm	example_resp_matrix_inv.py	Response matrix inversion with lease-square method (with regularization)

Table 4: Module `rf_det_act.py`

Function	Example	Comments
noniq_demod	example_demod.py	Perform non-I/Q demodulation of a given raw sampling waveform
twop_demod	example_demod.py	Demodulate raw waveform with every two samples
asyn_demod	example_demod.py	Demodulate raw waveform sampled by asynchronous clock (a reference signal waveform is needed)
self_demod_ap	example_demod.py	Demodulate raw waveform with Hilbert transform, returning the amplitude and phase waveforms
iq2ap_wf	example_demod.py	Convert I/Q waveforms to amplitude/phase waveforms
ap2iq_wf		Convert amplitude/phase waveforms to I/Q waveforms
norm_phase		Normalize phase (scalar or waveform) to a specific range (default ±180°)
pulse_info		Derive the pulse info like pulse width, pulse offset, etc.

Table 5: Module `rf_fit.py`

Function	Example	Comments
fit_sincos	example_fit_funcs.py	Fit the sine or cosine function
fit_circle	example_fit_funcs.py	Fit the 2D points to a circle function
fit_ellipse	example_fit_funcs.py	Fit the 2D points to an ellipse function
fit_Gaussian	example_fit_funcs.py	Fit a 1D Gaussian function

Table 6: Module `rf_noise.py`

Function	Example	Comments
calc_psd_coherent	example_noise_psd.py	Calculate the power spectral density (PSD) of a coherent sampled data series
calc_psd	example_noise_psd.py	Calculate the PSD of a general data series
rand_unif		Generate uniform distributed random numbers
gen_noise_from_psd	example_noise_time_series.py	Generate noise series from a given PSD spectrum
calc_rms_from_psd	example_noise_time_series.py	Calculate the RMS jitter from a given PSD spectrum
notch_filt	example_calib_for_ref_sc_cavity.py	Apply notch filter to a data series
design_notch_filter	example_feedback_analysis.py	Design a notch filter in state-space format
filt_step	example_feedback_basic.py	Apply a single time step of state-space filter
moving_avg		Moving average without compensating for the group delay

Table 7: Module `rf_sim.py`

Function	Example	Comments
cav_ss	example_sim_cavity_basic.py	Derive a continuous state-space equation of a cavity

cav_ss_mech		Cavity state-space equation with mechanical modes (to be implemented)
cav_impulse	example_aff_ilc.py	Derive the cavity impulse response from the cavity parameters
sim_ncav_pulse	example_sim_cavity_basic.py	Simulate cavity (with constant QL and detuning) response to a pulsed input
sim_ncav_step	example_sim_cavity_basic.py	Simulate cavity (with constant QL and detuning) response for a time step
sim_ncav_step_simple	example_sim_cavity_basic.py	Simulate cavity (with constant QL and detuning) response for a time step (simplified cavity equation only with the fundamental passband mode)
rf_power_req	example_power_req.py example_power_req2.py	Calculate the required RF power for desired cavity voltage and beam current
opt_QL_detuning	example_power_req.py example_power_req2.py	Calculate the optimal QL and detuning for minimizing the reflection power

Table 8: Module `rf_misc.py`

Function	Example	Comments
save_mat		Save a dictionary into a Matlab .mat file
load_mat		Load a Matlab .mat file into a dictionary
get_curtime_str		Get a string of the current date/time
get_bit		Get a bit of an integer
add_tf		Adding two transfer functions in numerator/denominator
plot_ellipse	example_fit_funcs.py	Plot an ellipse using its characteristics
plot_Guassian	example_fit_funcs.py	Plot a 1D Gaussian distribution

Table 9: Module `rf_plot.py`

Function	Example	Comments
plot_ss_discrete		Plot for function `rf_control.ss_discrete`
plot_ss_freqresp		Plot for function `rf_control.ss_freqresp`
plot_basic_rf_controller		Plot for function `rf_control.basic_rf_controller`
plot_loop_analysis		Plot for function `rf_control.loop_analysis`
plot_calc_psd		Plot for function `rf_noise.calc_psd` and `rf_noise.calc_psd_coherent`
plot_cav_ss		Plot for function `rf_sim.cav_ss`
plot_rf_power_req		Plot for function `rf_sim.rf_power_req`
plot_plot_ellipse		Plot for function `rf_misc.plot_ellipse`
plot_plot_Guassian		Plot for function `rf_misc.plot_Guassian`

All examples are self-descriptive and can be run directly. One can change the settings in the examples and examine the different results. The examples can be used as templates demonstrating how the library routines can be used in a user's software.

1.4 *rf_sysid Routines*

1.4.1 `prbs`

`prbs(n, lower_b = -1.0, upper_b = 1.0)`

```
Generate PRBS signal.
    PRBS monic polynomials:
        PRBS3    = x^3 + x^2 + 1
        PRBS4    = x^4 + x^3 + 1
        PRBS5    = x^5 + x^3 + 1
        PRBS6    = x^6 + x^5 + 1
        PRBS7    = x^7 + x^6 + 1
```

```
            PRBS8   = x^8 + x^6 + x^5 + x^4 + 1
            PRBS9   = x^9 + x^5 + 1
            PRBS10  = x^10 + x^7 + 1
            PRBS11  = x^11 + x^9 + 1
            PRBS12  = x^12 + x^11 + x^10 + x^4 + 1
            PRBS13  = x^13 + x^12 + x^11 + x^8 + 1
            PRBS14  = x^14 + x^13 + x^12 + x^2 + 1
            PRBS15  = x^15 + x^14 + 1
            PRBS16  = x^16 + x^14 + x^13 + x^11 + 1
            PRBS17  = x^17 + x^14 + 1
            PRBS18  = x^18 + x^11 + 1
            PRBS19  = x^19 + x^18 + x^17 + x^14 + 1
            PRBS20  = x^20 + x^3 + 1
            PRBS23  = x^23 + x^18 + 1
            PRBS31  = x^31 + x^28 + 1

    Input:
        n       - int, number of points
        lower_b - float, lower boundary
        upper_b - float, upper boundary
    Output:
        status  - boolean, True for success
        data    - numpy array, 1-D array of PRBS signal
```

The routine generates a Pseudorandom Binary Sequence (PRBS), which can be used as a stimuli signal for system identification. Note that the number of point *n* should satisfy

$$n = 2^k - 1, \tag{1}$$

with *k* an integer, to achieve a constant magnitude in the Power Spectral Density (PSD) of the resulting time series.

1.4.2 etfe

etfe(u, y, r = 1, exclude_transient = True, transient_batch_num = 1, fs = 1.0)

```
    Implement the Empirical Transfer Function Estimation (ETFE).

    Input:
        u                   - numpy array, system input waveform
        y                   - numpy array, system output waveform
        r                   - int, periods (number of batches) in the waveform
        exclude_transient   - boolean, True to exclude the transient (the first
                              "transient_batch_num" batches will be excluded
                              from calculation)
        transient_batch_num - int, number of transient batches
        fs                  - float, sampling frequency, Hz
    Output:
        status      - boolean, True for success
        freq        - numpy array, frequency vector
        G           - numpy array (complex), frequency response of the system
        u_fft       - numpy array (complex), spectrum of the input waveform
        y_fft       - numpy array (complex), spectrum of the output waveform
```

The ETFE algorithm estimates the frequency response of the system using the Discrete Fourier Transform (DFT) of the output (y) and input (u), given by:

$$\begin{aligned}
&G(f_k) = Y(f_k)/U(f_k), \\
&\text{where} \\
&U(f_k) = DFT\{u(t)\},\ Y(f_k) = DFT\{y(t)\}, \\
&\text{with } f_k = kf_s/N,\ k = 0,\ldots,N-1
\end{aligned} \tag{2}$$

Here U and Y are the DFT of u and y, respectively. The DFT is calculated with N samples sampled at frequency f_s. There are two methods implemented to improve the performance of ETFE:

a. Using repeated inputs (with r batches of data by repeating the same input series by r times) and making average to reduce the effects of measurement noise.

b. Remove the data in the transient of the system response (i.e., the first several batches of data).

1.4.3 half_bw_decay

half_bw_decay(amp_wf, decay_ids, decay_ide, Ts)

```
Calculate the half-bandwidth of a standing-wave cavity from RF pulse decay
(refer to LLRF Book [1] section 9.4.3).

Input:
    amp_wf    - numpy array, amplitude waveform
    decay_ids - int, starting index of calculation window
    decay_ide - int, ending index of calculation window
    Ts        - float, sampling time, s
Output:
    status    - boolean, True for successful calculation
    half_bw   - float, half bandwidth, rad/s
```

1.4.4 detuning_decay

detuning_decay(pha_wf_deg, decay_ids, decay_ide, Ts)

```
Calculate the detuning of a standing-wave cavity from RF pulse decay (refer
to LLRF Book section 9.4.3).

Input:
    pha_wf_deg - numpy array, phase waveform, deg
    decay_ids  - int, starting index of calculation window
    decay_ide  - int, ending index of calculation window
    Ts         - float, sampling time, s
Output:
    status     - boolean, True for successful calculation
    detuning   - float, detuning, rad/s
```

1.4.5 cav_drv_est

cav_drv_est(vc, half_bw, Ts, detuning = 0.0, beta = 1e4)

```
Calculate the theoretical drive and reflected waveform for the cavity probe
waveform.

Input:
    vc       - numpy array (complex), cavity probe waveform
    half_bw  - float, half bandwidth of the cavity, rad/s
    Ts       - float, sampling time, s
    detuning - float, detuning of the cavity, rad/s
    beta     - float, input coupling factor (needed for NC cavities; for SC
               cavities, can use the default value, or you can specify it if
               more accurate calibration is needed)
Output:
    status   - boolean, success (True) or fail (False)
    vf       - numpy array (complex), estimated cavity drive waveform
    vr       - numpy array (complex), estimated cavity reflected waveform

Note: 1. If the cavity probe signal contains pass-band modes, better notch
         filter them
      2. The estimate is useful to determine how to move the measured
         forward and reflected signals to align with the timing of the
         cavity probe signal
```

This function estimates the waveforms of the cavity drive signal ($\mathbf{v}_{for}$) and reflected signal ($\mathbf{v}_{ref}$) using the cavity probe signal ($\mathbf{v}_C$) as reference. The input coupling factor (β), half-bandwidth ($\omega_{1/2}$), and detuning ($\Delta\omega$) should be given. The algorithm uses the following cavity equation with voltage drives (Eq. (3.31) of LLRF Book [1]):

$$\dot{\mathbf{v}}_C + \left(\omega_{1/2} - j\Delta\omega\right)\mathbf{v}_C = 2\omega_{1/2}\left(\frac{\beta}{\beta+1}\mathbf{v}_{for} + \mathbf{v}_{b0}\right) \tag{3}$$

with the beam drive voltage $\mathbf{v}_{b0} = 0$. The forward and reflected voltage waveforms are calculated as

$$\begin{aligned} \mathbf{v}_{for} &= \frac{\beta+1}{\beta}\cdot\frac{\dot{\mathbf{v}}_C + \left(\omega_{1/2} - j\Delta\omega\right)\mathbf{v}_C}{2\omega_{1/2}}, \\ \mathbf{v}_{ref} &= \mathbf{v}_C - \mathbf{v}_{for}. \end{aligned} \tag{4}$$

The implementation has used the discretized cavity equation to avoid directly calculating the derivative of the cavity voltage.

1.4.6 `cav_par_pulse`

`cav_par_pulse(vc, vf, half_bw, Ts, beta = 1e4)`

```
Calculate the half-bandwidth and detuning of a standing-wave cavity within
an RF pulse with beam off (directly solve the cavity equation)(refer to LLRF
Book section 9.4.3).

Input:
    vc      - numpy array (complex), cavity probe waveform (reference plane)
    vf      - numpy array (complex), cavity forward waveform (calibrated to
               the same reference plan as the cavity probe signal)
    half_bw - float, half bandwidth of the cavity (derived from early part
               of decay), rad/s
    Ts      - float, sampling time, s
    beta    - float, input coupling factor (needed for NC cavities; for SC
               cavities, can use the default value, or you can specify it if
               more accurate calibration is needed)
Output:
    status  - boolean, success (True) or fail (False)
    wh_pul  - numpy array, half-bandwidth in the pulse, rad/s
    dw_pul  - numpy array, detuning in the pulse, rad/s
```

The implementation actually uses the polar differential equations of the cavity given by Eq. (4.24) in the LLRF Book. The derivative is calculated with the `gradient` function of `numpy`, which is sensitive to the measurement noise. In the example code, one can compare the result of this function with that of `cav_par_pulse_obs`, which avoids calculating the derivative and therefore, gives more smooth results.

1.4.7 `cav_par_pulse_obs`

`cav_par_pulse_obs(vc, vf, half_bw, Ts, vb = None, beta = 1e4, pole_scale = 50)`

```
Calculate the half-bandwidth and detuning of a standing-wave cavity within
an RF pulse with beam on/off (use ADRC observer) (refer to the paper: Geng Z
(2017a) Superconducting cavity control and model identification based on
active disturbance rejection control. IEEE Trans Nucl Sci 64(3):951-958).

Input:
    vc         - numpy array (complex), cavity probe waveform (reference
                  plane)
    vf         - numpy array (complex), cavity forward waveform (calibrated
                  to the same reference plan as the cavity probe signal)
```

```
    half_bw     - float, half bandwidth of the cavity (derived from early
                  part of decay), rad/s
    Ts          - float, sampling time, s
    vb          - numpy array (complex), beam drive waveform (calibrated to
                  the same reference plan as the cavity probe signal)
    beta        - float, input coupling factor (needed for NC cavities; for
                  SC cavities, can use the default value, or you can specify
                  it if more accurate calibration is needed)
    pole_scale  - float, scale of the cavity half-bandwidth for the observer
                  pole, it should be tens of times of the closed-loop
                  bandwidth of the cavity
Output:
    status      - boolean, success (True) or fail (False)
    wh_pul      - numpy array, half-bandwidth in the pulse, rad/s
    dw_pul      - numpy array, detuning in the pulse, rad/s
    vc_est      - numpy array (complex), cavity probe waveform (estimated by
                  observer)
    f_est       - numpy array (complex), general disturbance WF (estimated
                  by observer)
```

This function uses a disturbance observer to estimate the cavity voltage (smoothen the cavity probe signal) and a general disturbance term. From the general disturbance term, the cavity half-bandwidth and detuning at each time step can be estimated. The algorithm is described in the referred paper. The major benefit of the observer is that we can avoid calculating the derivatives and obtain less noisy results. Note that the beam drive term (calibrated to the same reference plane as the cavity probe signal) should be input to calculate the time-varying half-bandwidth and detuning if the beam is present.

1.4.8 `cav_beam_pulse_obs`

cav_beam_pulse_obs(vc, vf, wh_pul, dw_pul, half_bw, Ts, beta = 1e4, pole_scale = 20)

```
Calculate the beam drive waveform of a standing-wave cavity given the intra-
pulse half-bandwidth and detuning (refer to the paper: Geng Z (2017a)
Superconducting cavity control and model identification based on active
disturbance rejection control. IEEE Trans Nucl Sci 64(3):951-958).

Input:
    vc          - numpy array (complex), cavity probe waveform (reference
                  plane)
    vf          - numpy array (complex), cavity forward waveform (calibrated
                  to the same reference plane as the cavity probe signal)
    wh_pul      - numpy array, half-bandwidth in the pulse, rad/s
    dw_pul      - numpy array, detuning in the pulse, rad/s
    half_bw     - float, half bandwidth of the cavity (derived from early
                  part of decay), rad/s
    Ts          - float, sampling time, s
    beta        - float, input coupling factor (needed for NC cavities; for
                  SC cavities, can use the default value, or you can specify
                  it if more accurate calibration is needed)
    pole_scale  - float, scale of the cavity half-bandwidth for the observer
                  pole, it should be tens of times of the closed-loop
                  bandwidth of the cavity
Output:
    status      - boolean, success (True) or fail (False)
    vb_est      - numpy array (complex), beam drive waveform (estimated by
                  observer)
    vc_est      - numpy array (complex), cavity probe waveform (estimated by
                  observer)
    f_est       - numpy array (complex), general disturbance WF (estimated
                  by observer)
```

Similar to the "`cav_par_pulse_obs`" function, this function calculates the waveform of the beam drive voltage given the cavity probe, forward signals and the intra-pulse half-bandwidth and detuning (e.g., calculated when the beam is off). Here we assume that the cavity voltage with the beam present is well controlled to be the same as the case without beam. Only in this case, the half-bandwidth and detuning of the cavity during the RF pulse can be assumed unchanged after injecting the beam.

1.4.9 `cav_observer`

`cav_observer(vc, vf, half_bw, Ts, beta = 1e4, pole_scale = 50)`

```
Estimate the cavity voltage (denoised) and the general disturbance with the
ADRC observer (refer to the paper: Geng Z (2017a) Superconducting cavity
control and model identification based on active disturbance rejection
control. IEEE Trans Nucl Sci 64(3):951-958).

Input:
    vc          - numpy array (complex), cavity probe waveform (reference
                  plane)
    vf          - numpy array (complex), cavity forward waveform (calibrated
                  to the same reference plane as the cavity probe signal)
    half_bw     - float, half bandwidth of the cavity (derived from early
                  part of decay), rad/s
    Ts          - float, sampling time, s
    beta        - float, input coupling factor (needed for NC cavities; for
                  SC cavities, can use the default value, or you can specify
                  it if more accurate calibration is needed)
    pole_scale  - float, scale of the cavity half-bandwidth for the observer
                  pole, it should be tens of times of the closed-loop
                  bandwidth of the cavity
Output:
    status      - boolean, success (True) or fail (False)
    vc_est      - numpy array (complex), cavity probe waveform (estimated by
                  observer)
    f_est       - numpy array (complex), general disturbance WF (estimated
                  by observer)
```

1.4.10 `iden_impulse`

`iden_impulse(U, Y, order = 20)`

```
Identify the impulse response using the input-output data.

Input:
    U      - numpy array (complex), input waveforms
    Y      - numpy array (complex), output waveforms
    order  - int, order of the impulse response
Output:
    status - boolean, success (True) or fail (False)
    h      - numpy array (complex), impulse response
```

This function estimates the impulse response of a system using its input and output signals. Complex signals are allowed for the RF system with its input and output as phasors representing the signals' complex envelopes. The relationship between the input u, output y, and the system's impulse response h is described by Eq. (4.41) of the LLRF Book. Here we model the system as a Finite Impulse Response (FIR) filter, therefore, the order should be large enough to cover most of the significant elements in the impulse response h.

1.4.11 `beta_powers`

`beta_powers(pf, pr, weak = False)`

```
Identify the cavity input coupling factor using the steady state forward and
reflected powers (refer to LLRF Book section 9.4.1).

Input:
    pf   - float, forward power in steady-state
    pr   - float, reflected power in steady-state (with the same unit as pf)
    weak - boolean, True for weak coupling (beta < 1), False for strong
           coupling (beta > 1)
```

```
Output:
    status - boolean, True for success
    beta   - float, cavity input coupling factor
```

1.5 rf_calib Routines

1.5.1 `calib_vprobe`

`calib_vprobe(vc, vf_m, vr_m)`

```
Calibrate the virtual probe signal of the cavity with
                vc = m * vf_m + n * vr_m
aiming at reconstructing the probe signal with the forward and reflected
signals. This is useful when the probe is temporally not available.

Input:
    vc   - numpy array (complex), cavity probe waveform (as reference plane)
    vf_m - numpy array (complex), cavity forward waveform (meas.)
    vr_m - numpy array (complex), cavity reflected waveform (meas.)
Output:
    status - boolean, success (True) or fail (False)
    m, n   - float (complex), calibration coefficients

Note: the waveforms should have been aligned in time, i.e., the relative
delays between them should have been removed
```

One example to apply the virtual probe calibration is for the SwissFEL Gun, whose probe if strongly affected by the mechanical vibrations caused by the cooling system. As shown in Figure 4, the forward and reflected signals picked before the Gun cavity input coupler are calibrated with two complex coefficients (*m* and *n*) to match one of the cavity probe signal (see the equation in the function description above).

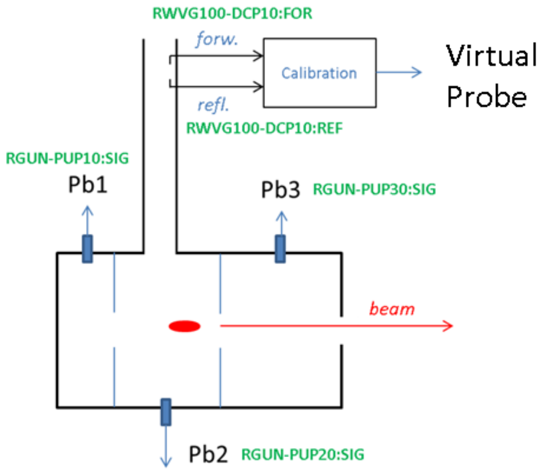

Figure 4: Virtual probe calibration for SwissFEL RF Gun

After calibration, the virtual probe signal was used as the input for RF feedback. Figure 5 compares the amplitudes and phases of the physical probe (Pb1) and virtual probe. It can be seen that the resonance peaks in the probe amplitude (caused by the mechanical vibrations applied to the probe antenna) is not visible in the virtual probe signal.

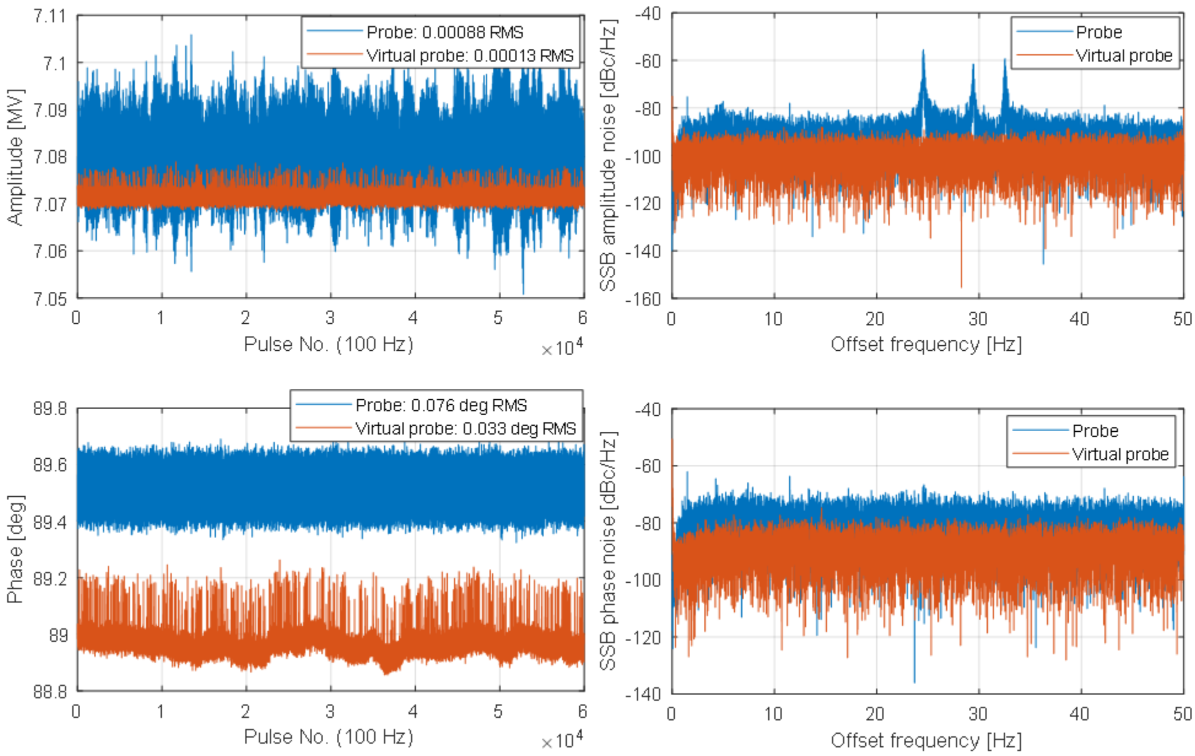

Figure 5: Comparison of cavity probe (affected by mechanical vibrations) and virtual probe signals

1.5.2 `calib_dac_offs`

`calib_dac_offs(vdac, viqm, sig_ids, sig_ide, leak_ids, leak_ide, delay = 0)`

```
Calibrate the DAC offset to remove carrier leakage for direct upconversion
(refer to LLRF Book section 9.2.2).

Input:
    vdac     - numpy array (complex), DAC output waveform
    viqm     - numpy array (complex), I/Q modulator output meas. waveform
    sig_ids  - int, starting index of signals
    sig_ide  - int, ending index of signals
    leak_ids - int, starting index of leakage
    leak_ide - int, ending index of leakage
    delay    - int, delay in number of points between I/Q out and DAC WFs,
                it is needed to be set reflecting the actual delay for better
                results
Output:
    status   - boolean, success (True) or fail (False)
    offset_I - float, I offset to be added to the actual DAC output
    offset_Q - float, Q offset to be added to the actual DAC output
```

1.5.3 `calib_iqmod`

`calib_iqmod(vdac, viqm)`

```
Calibrate the I/Q modulator imbalance, return inv(M) with
     [real(viqm) imag(viqm)]' = M * [real(vdac) imag(vdac)]' +
                                [iqm_offs_I iqm_offs_Q]'
This method is OK to calibrate the imbalance matrix M (use its inverse
to pre-distort the DAC output) but the offset is not accurate to determine
the DAC offset calibration. It is suggested to calibrate the DAC offset
first using the routine "calib_dac_offs" before executing this function
(refer to paper: Geng Z, Hong B (2016) Design and calibration of an RF
actuator for low-level RF systems. IEEE Trans Nucl Sci 63(1):281-287).
```

```
Input:
    vdac    - numpy array (complex), DAC actuation points
    viqm    - numpy array (complex), I/Q modulator meas. points
Output:
    status - boolean, success (True) or fail (False)
    invM    - numpy matrix, inversion of M, can be directly used to correct
              the I/Q modulator imbalance
    viqm_s - numpy array (complex), scaled I/Q modulator output
    viqm_c - numpy array (complex), re-constructed I/Q mod. output
```

1.5.4 calib_cav_for

calib_cav_for(vc, vf_m, pul_ids, pul_ide, half_bw, detuning, Ts, beta = 1e4)

```
Calibrate the cavity forward signal: vf = a * vf_m. The resulting "vf" is
the cavity drive signal at the same reference plane as the cavity probe "vc"
(refer to LLRF Book section 4.6.1.3).

Input:
    vc        - numpy array (complex), cavity probe waveform(reference plane)
    vf_m      - numpy array (complex), cavity forward waveform (meas.)
    pul_ids   - int, starting index for calculation (calculation window
                should cover the entire pulse for NC cavities; use the part
                close to pulse end for SC cavities)
    pul_ide   - int, ending index for calculation
    half_bw   - float, half bandwidth of the cavity, rad/s
    detuning - float, detuning of the cavity, rad/s
    Ts        - float, sampling time, s
    beta      - float, input coupling factor (needed for NC cavities; for SC
                cavities, can use the default value, or you can specify it if
                more accurate calibration is needed)
Output:
    status    - boolean, success (True) or fail (False)
    a         - float (complex), calibrate coefficient

Note: the waveforms should have been aligned in time, i.e., the relative
delays between them should have been removed
```

1.5.5 calib_ncav_for_ref

calib_ncav_for_ref(vc, vf_m, vr_m, pul_ids, pul_ide, half_bw, detuning, Ts, beta = 1.0)

```
Calibrate the cavity forward and reflected signals for a normal conducting
cavity with constant loaded Q and detuning:
        vf = a * vf_m + b * vr_m
        vr = c * vf_m + d * vr_m
The resulting "vf" and "vr" are the cavity drive/reflected signals at the
same reference plane as the cavity probe "vc" (refer to LLRF Book section
9.3.5).

Input:
    vc         - numpy array (complex), cavity probe waveform (reference
                 plane)
    vf_m       - numpy array (complex), cavity forward waveform (meas.)
    vr_m       - numpy array (complex), cavity reflected waveform (meas.)
    pul_ids    - int, starting index for calculation (calculation window
                 should cover entire pulse)
    pul_ide    - int, ending index for calculation
    half_bw    - float, half bandwidth of the cavity, rad/s
    detuning   - float, detuning of the cavity, rad/s
    Ts         - float, sampling time, s
    beta       - float, input coupling factor (needed for NC cavities)
Output:
    status     - boolean, success (True) or fail (False)
```

```
    a,b,c,d   - float (complex), calibrate coefficients

Note: the waveforms should have been aligned in time, i.e., the relative
delays between them should have been removed
```

The implementation used a simplified algorithm compared to the one described in the Sect. 9.3.5 of the LLRF Book. This simplification benefits from the assumption that the loaded Q and detuning of the cavity are constant. First, the theoretical cavity forward and reflected signals are estimated with the `cav_drv_est` function. Then, we estimate the coefficients $m = a + b$ and $n = b + d$ using the `calib_vprobe` function. After that, we estimate a and b based on the estimated theoretical forward signal and the measured forward and reflected signals (also using `calib_vprobe` – use its feature of least-square fitting). Finally, all the four parameters a, b, c, and d are calculated.

1.5.6 calib_scav_for_ref

calib_scav_for_ref(vc, vf_m, vr_m, pul_ids, pul_ide, decay_ids, decay_ide, half_bw, detuning, Ts, beta = 1e4)

```
Calibrate the cavity forward and reflected signals for a superconducting
cavity with time-varying loaded Q or detuning:
        vf = a * vf_m + b * vr_m
        vr = c * vf_m + d * vr_m
The resulting "vf" and "vr" are the cavity drive/reflected signals at the
same reference plane as the cavity probe "vc" (refer to LLRF Book section
9.3.5).

Input:
    vc         - numpy array (complex), cavity probe waveform (reference
                 plane)
    vf_m       - numpy array (complex), cavity forward waveform (meas.)
    vr_m       - numpy array (complex), cavity reflected waveform (meas.)
    pul_ids    - int, starting index for calculation (calculation window
                 should take the pulse end part close to decay)
    pul_ide    - int, ending index for calculation
    decay_ids  - int, starting id of calculation window at decay stage
    decay_ide  - int, ending id of calculation window at decay stage
    half_bw    - float, half bandwidth of the cavity (derived from early part
                 of decay), rad/s
    detuning   - float, detuning of the cavity (derived from early part of
                 decay), rad/s
    Ts         - float, sampling time, s
    beta       - float, input coupling factor (for SC cavities, can use the
                 default value, or you can specify it if more accurate
                 calibration is needed)
Output:
    status     - boolean, success (True) or fail (False)
    a,b,c,d    - float (complex), calibrate coefficients

Note: the waveforms should have been aligned in time, i.e., the relative
delays between them should have been removed
```

The implementation is slightly different compare to the algorithm described in the Sect. 9.3.5 of the LLRF Book. The major difference is the method to calculate the parameter a. In this implementation, a is calculated using the estimated theoretical forward signal around the end of the RF pulse, where the cavity half-bandwidth and detuning is assumed to be the same as the parameter values input to the function.

1.5.7 for_ref_volt2power

for_ref_volt2power(roQ_or_RoQ, QL, vf_pcal = None, vr_pcal = None, beta = 1e4, machine = 'linac')

```
Convert the calibrated forward and reflected signals (in physical unit, V)
to forward and reflected power (in W) (refer to LLRF Book section 3.3.9).
```

```
Input:
    roQ_or_RoQ  - float, cavity r/Q of Linac or R/Q of circular accelerator,
                  Ohm (see the note below)
    QL          - float, loaded quality factor of the cavity
    vf_pcal     - numpy array (complex), forward waveform (calibrated to
                  physical unit)
    vr_pcal     - numpy array (complex), reflected waveform (calibrated to
                  physical unit)
    beta        - float, input coupling factor (needed for NC cavities; for
                  SC cavities, can use the default value, or you can specify
                  it if more accurate result is needed)
    machine     - string, 'linac' or 'circular', used to select r/Q or R/Q
Output:
    status      - boolean, success (True) or fail (False)
    for_power   - numpy array, waveform of forward power (if input is not
                  None), W
    ref_power   - numpy array, waveform of reflected power (if input is not
                  None), W
    C           - float (complex), calibration coefficient: power_W = C *
                  Volt_V^2

Note: Linacs define the "r/Q = Vacc**2 / (w0 * U)" while circular machines
define "R/Q = Vacc**2 / (2 * w0 * U)", where Vacc is the accelerating
voltage, "w0" is the angular cavity resonance frequency and U is the cavity
energy storage. Therefore, to use this function, one needs to specify the
"machine" to be "linac" or "circular". Generally, we have "R/Q = 1/2 * r/Q"
```

When using this function (and other functions requiring specification of `machine`), one must notice the difference of the definitions of the normalized effective shunt impedance in storage rings and Linacs. In a Linac, the *normalized effective shunt impedance* (*r/Q*) of a standing-wave cavity is defined as

$$r/Q = \frac{V_{acc}^2}{\omega_0 W},\ r = \frac{V_{acc}^2}{P_{cav}} \tag{5}$$

where V_{acc} is the accelerating voltage taking into account the transit time factor, P_{cav} is the RF power dissipated in the cavity wall, ω_0 is the cavity resonance frequency (angular frequency), and W is the stored electromagnetic field energy in the cavity. Here r is the *effective shunt impedance*. In a storage ring, we have the following definition:

$$R/Q = \frac{V_{acc}^2}{2\omega_0 W},\ R = \frac{V_{acc}^2}{2P_{cav}}. \tag{6}$$

Therefore, we have the following relation between the *R/Q* given in storage-ring cavity specifications and the *r/Q* given in Linac cavity specifications:

$$R/Q = \frac{1}{2} r/Q. \tag{7}$$

In the LLRF Book, the loaded resistance R_L (Sect. 3.3.1) is defined following the Linac convention. It can be rewritten as follows in either Linac or storage-ring conventions:

$$R_L = \frac{1}{2}(r/Q)Q_L = (R/Q)Q_L. \tag{8}$$

Now let's workout the method how to convert the forward/reflected voltages (already calibrated to physical units) into powers. The relation between the forward (reflected) current phasor and voltage phasor is

$$\mathbf{i}_{for} = \frac{\beta \mathbf{v}_{for}}{(\beta+1) R_L}, \ \mathbf{i}_{ref} = \frac{\beta \mathbf{v}_{ref}}{(\beta+1) R_L}, \tag{9}$$

where β is the input coupling factor of the cavity. Therefore, the forward (reflected) power can be calculated from the forward (reflected) voltage as

$$\begin{aligned} P_{for} &= \frac{\mathbf{v}_{for}\mathbf{i}^*_{for}}{2} = \frac{\beta}{\beta+1}\cdot\frac{|\mathbf{v}_{for}|^2}{2R_L}, \\ P_{ref} &= \frac{\mathbf{v}_{ref}\mathbf{i}^*_{ref}}{2} = \frac{\beta}{\beta+1}\cdot\frac{|\mathbf{v}_{ref}|^2}{2R_L}. \end{aligned} \tag{10}$$

Here * is the conjugate of the phasor. Note that for superconducting cavities, β is much larger than 1, then the terms with β can be neglected and we reach the Eq. (9.35) of the LLRF Book.

1.5.8 phasing_energy

phasing_energy(phi_vec, E_vec, machine = 'linac')

```
Calibrate the beam phase using the beam energy measured by scanning RF phase
(refer to LLRF Book section 9.3.2.3).

Input:
    phi_vec - numpy array, phase measurement, degree
    E_vec   - numpy array, beam energy measurement
    machine - string, 'linac' or 'circular', see the note below
Output:
    status  - boolean, success (True) or fail (False)
    Egain   - float, maximum energy gain of the RF station
    phi_off - float, phase offset that should be added to the phase
              measurement, deg

Note: circular accelerator use sine as convention of beam phase definition,
resulting in 90 degree for on-crest acceleration; while for Linacs, the
cosine convention is used with 0 degree for on-crest acceleration
```

1.5.9 egain_cav_power

egain_cav_power(pfor, roQ_or_RoQ, QL, beta = 1e4, machine = 'linac')

```
Standing-wave cavity energy gain estimate from RF drive power without beam
loading (refer to LLRF Book section 9.3.2.1, Eq. (9.13)).

Input:
    pfor        - float, cavity drive power, W
    roQ_or_RoQ  - float, cavity r/Q of Linac or R/Q of circular accelerator,
                  Ohm (see the note of section 1.5.7)
    QL          - float, loaded quality factor of the cavity
    beta        - float, input coupling factor (needed for NC cavities; for
                  SC cavities, can use the default value, or you can specify
                  it if more accurate result is needed)
    machine     - string, 'linac' or 'circular', used to select r/Q or R/Q
Output:
    status      - boolean, success (True) or fail (False)
    vc0         - float, desired cavity voltage for the given drive power
```

1.5.10 egain_cgstr_power

egain_cgstr_power(pfor, f0, rs, L, Q, Tf)

```
Traveling-wave constant gradient structure energy gain estimate from RF
drive power without beam loading (refer to LLRF Book section 9.3.2.1, Eq.
(9.14)).

Input:
    pfor - float, structure input RF power, W
    f0   - float, RF operating frequency, Hz
    rs   - float, shunt impedance per unit length, Ohm/m
    L    - float, length of the traveling-wave structure, m
    Q    - float, quality factor of the TW structure
    Tf   - float, filling time of the TW structure, s

Output:
    status - boolean, success (True) or fail (False)
    vc0    - float, desired accelerating voltage for the given drive power
```

1.5.11 calib_vsum_poor

calib_vsum_poor(amp_vec, pha_vec, ref_ch = 0)

```
Calibrate the vector sum (poor man's solution by rotating and scaling all
other channels referring to the first channel).

Input:
    amp_vec - numpy array, amplitude of all channels
    pha_vec - numpy array, phase of all channels, deg
    ref_ch  - int, reference channel, between 0 and (number of channels - 1)
Output:
    status  - boolean, success (True) or fail (False)
    scale   - numpy array, scale factor for all channels
    phase   - numpy array, phase offset adding to all channels, deg
```

1.5.12 calib_sys_gain_pha

calib_sys_gain_pha(vact, vf, beta = 1e4)

```
Calibrate the system gain and system phase (refer to LLRF Book section
9.4.2).

Input:
    vact  - numpy array (complex), RF actuation waveform (usually DAC output)
    vf    - numpy array (complex), cavity forward signal (calibrated to the
            reference plane of the cavity voltage or vector-sum voltage)
    beta  - float, input coupling factor (needed for NC cavities; for SC
            cavities, can use the default value, or you can specify it if
            more accurate result is needed)
Output:
    status    - boolean, success (True) or fail (False)
    sys_gain  - numpy array, waveform of system gain
    sys_phase - numpy array, waveform of system phase, deg
```

1.6 rf_control Routines

1.6.1 ss_discrete

ss_discrete(Ac, Bc, Cc, Dc, Ts, method = 'zoh', alpha = 0.3, plot = False, plot_pno = 1000)

```
Derive the discrete state-space equation from a continuous one and compare
their frequency responses.

Input:
    Ac, Bc, Cc, Dc - numpy matrix (complex), continuous state-space model
    Ts             - float, sampling time, s
```

```
    method         - string, 'gbt, 'bilinear', 'euler', 'backward_diff',
                     'zoh', 'foh' or 'impulse' (see document of
                     signal.cont2discrete)
    alpha          - float, 0 to 1 (see document of signal.cont2discrete)
    plot           - boolean, enable the plot of frequency responses
    plot_pno       - int, number of point in the plot
Output:
    status         - boolean, success (True) or fail (False)
    Ad, Bd, Cd, Dd - numpy matrix (complex), discrete state-space model
```

This function converts the continuous state-space equation

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}_c\mathbf{x}(t) + \mathbf{B}_c\mathbf{u}(t), \\ \mathbf{y}(t) &= \mathbf{C}_c\mathbf{x}(t) + \mathbf{D}_c\mathbf{u}(t) \end{aligned} \tag{11}$$

into a discrete state-space equation as

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{A}_d\mathbf{x}(k) + \mathbf{B}_d\mathbf{u}(k), \\ \mathbf{y}(k) &= \mathbf{C}_d\mathbf{x}(k) + \mathbf{D}_d\mathbf{u}(k) \end{aligned} \tag{12}$$

where k is the index of the samples. If enabled, the frequency response of the continuous and discrete state-space equations can be plot for comparison. The parameters should be tuned to match the discrete system frequency response to the continuous system frequency response as much as possible.

1.6.2 `ss_cascade`

`ss_cascade(A1, B1, C1, D1, A2, B2, C2, D2, Ts = None)`

```
Cascade two state-space systems with the first system applied to the input
first. This function works for both continuous system (Ts is None) and
discrete systems (Ts has a nonzero floating value).

Input:
    A1, B1, C1, D1 - numpy matrix (complex), state-space model of system 1
    A2, B2, C2, D2 - numpy matrix (complex), state-space model of system 2
    Ts             - float, sampling time, s
Output:
    status         - boolean, success (True) or fail (False)
    A, B, C, D     - numpy matrix (complex), state-space model of cascaded
                     system
```

1.6.3 `ss_freqresp`

`ss_freqresp(A, B, C, D, Ts = None, plot = False, plot_pno = 1000, plot_maxf = 0.0, title = 'Frequency Response')`

```
Plot the frequency response of a state-space system. This function works for
both continuous system (Ts is None) and discrete systems (Ts has a nonzero
floating value).

Input:
    A, B, C, D - numpy matrix (complex), state-space model of system
    Ts         - float, sampling time, s
    plot       - boolean, enable the plot of frequency response
    plot_pno   - int, number of point in the plot
    plot_maxf  - float, frequency range (+-) to be plotted, Hz
    title      - string, title showed on the plot
Output:
    status     - boolean, success (True) or fail (False)
    f_wf       - numpy array, frequency waveform, Hz
    A_wf_dB    - numpy array, amplitude response waveform, dB
    P_wf_deg   - numpy array, phase response waveform, deg
    h          - numpy array (complex), complex response
```

1.6.4 `basic_rf_controller`

basic_rf_controller(Kp, Ki, notch_conf = None, plot = False, plot_pno = 1000, plot_maxf = 0.0)

```
Generate a basic RF controller in the format of a continuous state-space
equation with
 - I/Q control strategy (input/output are all complex signals)
 - Configurable for PI control + frequency notches

Input:
    Kp          - float, proportional (P) gain
    Ki          - float, integral (I) gain
    notch_conf  - dict, with the following items:
                  'freq_offs'    : list, offset frequency to be notched, Hz
                  'gain'         : list, gain of the notch (inverse of
                                    suppress ratio)
                  'half_bw'      : list, half bandwidth of the notch filter,
                                    rad/s
    plot        - boolean, enable the plot of frequency response of the
                  controller
    plot_pno    - int, number of point in the plot
    plot_maxf   - float, frequency range (+-) to be plotted, Hz
Output:
    status             - boolean, success (True) or fail (False)
    Akc, Bkc, Ckc, Dkc - numpy matrix (complex), continuous state-space
                         controller
```

The phasor transfer function of the basic RF controller is given by

$$K(\hat{s}) = K_P + \frac{K_I}{\hat{s}} + \sum_{i=1}^{n} \frac{g_i \omega_{1/2,i}}{\hat{s} + \omega_{1/2,i} - j\omega_{notch,i}} + \sum_{i=1}^{n} \frac{g_i \omega_{1/2,i}}{\hat{s} + \omega_{1/2,i} + j\omega_{notch,i}} . \tag{13}$$

This is a Proportional-Integral (PI) controller with frequency notches at $\omega_{notch,i}$, $i = 1,\ldots,n$. This controller is used in an I/Q control loop. It accepts a complex input ($I + jQ$) and produces a complex output actuating on the I and Q channels via its real and imaginary components, respectively. The parameters are described as follows:

- K_P: proportional gain
- K_I: integral gain
- g_i: max gain of the *i*th notch filter
- $\omega_{1/2,i}$: half bandwidth of the *i*th notch filter
- $\omega_{notch,i}$: frequency to be suppressed by the *i*th notch filter

Note that we apply notches at both sidebands with frequency offsets of $\pm\omega_{notch,i}$. If it is clear that the disturbance to be suppressed appears only at one sideband, the notch filter at the other sideband can be discarded. The returned controller is in the format of state-space equation. Figure 6 shows the open-loop bode plot of an RF control loop consisting of a cavity and a basic RF controller. The cavity parameters are: resonance frequency = 500 MHz, half-bandwidth = 281.7 kHz, detuning = 2 * 281.7 kHz. The controller is a P controller with K_P = 10 and with notches at 1×, 2×, and 3×1.04 MHz.

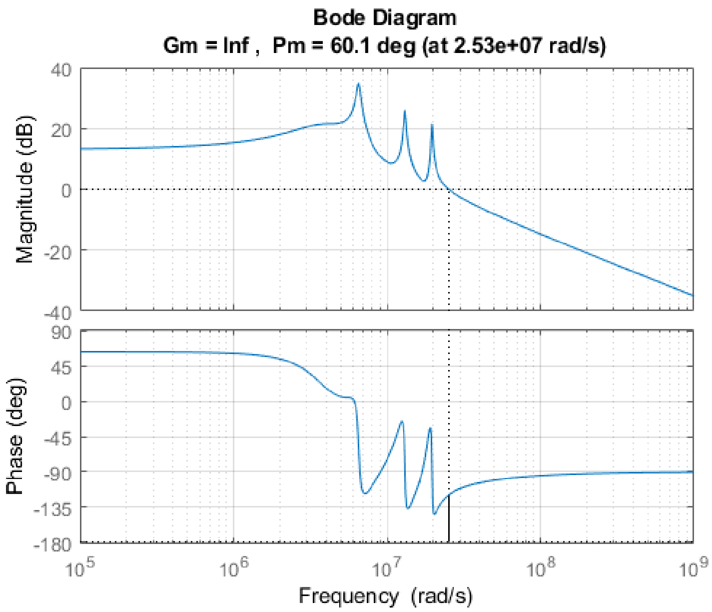

Figure 6: Bode plot of an RF control loop open-loop transfer function

1.6.5 control_step

control_step(Akd, Bkd, Ckd, Dkd, err_step, state_k0, ff_step = 0.0)

```
Controller execute for one step based on the discrete state-space equation.

Input:
    Akd, Bkd, Ckd, Dkd - numpy matrix (complex), discrete state-space
                         controller
    err_step  - complex, system output error (input to controller) of this
                step
    state_k0  - numpy matrix (complex), state of the last step
    ff_step   - complex, feedforward of this step
Output:
    status    - boolean, success (True) or fail (False)
    ctrl_step - complex, overall output of the controller of this step
    ctrl_out  - complex, feedback output (exclude feedforward wrt ctrl_step)
    state_k   - numpy matrix (complex), state of this step (should be input
                to the next execution of this function)

Question: in the second equation, shall we use "state_k0" or "state_k"?
```

1.6.6 loop_analysis

loop_analysis(AG, BG, CG, DG, AK, BK, CK, DK, Ts = None, delay_s = 0, plot = True, plot_pno = 100000, plot_maxf = 0.0, label = '')

```
Control loop analysis, including
 - derive the open loop transfer function
 - calculate the sensitivity and complementary sensitivity functions
This function works for both continuous system (Ts is None) and discrete
systems (Ts has a nonzero floating value).

Input:
    AG, BG, CG, DG - numpy matrix (complex), plant model
    AK, BK, CK, DK - numpy matrix (complex), controller
    Ts             - float, sampling time, s
    delay_s        - float, loop delay, s
    plot           - boolean, enable the plot of bode and Nyquist plots
    plot_pno       - int, number of point in the plot
    plot_maxf      - float, frequency range (+-) to be plotted, Hz
```

```
Output:
    status          - boolean, success (True) or fail (False)
    S_max           - float, maximum sensitivity, dB
    T_max           - float, maximum complementary sensitivity, dB
```

The open-loop transfer function is constructed by cascading the system state-space equation (`AG`, `BG`, `CG`, `DG`) and the controller state-space equation (`AK`, `BK`, `CK`, `DK`). The frequency response of the open loop can be used to construct the bode-plot and Nyquist plot, with which the closed-loop stability and the phase/gain margins can be examined.

1.6.7 cav_sp_ff

cav_sp_ff(half_bw, filling_len, flattop_len, Ts, pno, vc0 = 1.0, detuning = 0.0, beta = 1e4, const_fpow = True, ib0 = None, phib_deg = 0.0, beam_ids = 0, beam_ide = 0, roQ_or_RoQ = 0.0, QL = 3e6, machine = 'linac')

```
Generate the basic cavity setpoint and feedforward tables used to configure
the LLRF controller (later may apply smooth to the edges) (refer to LLRF
Book section 9.2.1).

Input:
    half_bw     - float, half bandwidth of the cavity, rad/s
    filling_len - int, length of cavity filling time, number of samples
    flattop_len - int, length of the flattop for beam acc., number of
                  samples
    Ts          - float, sampling time, s
    pno         - int, number of samples in the returned waveforms
    vc0         - float, desired cavity voltage at the flattop, V
    detuning    - float, detuning of the cavity, rad/s
    beta        - float, input coupling factor (needed for NC cavities; for
                  SC cavities, can use the default value, or you can specify
                  it if more accurate result is needed)
    const_fpow  - boolean, True for forcing constant filling drive
                  power/phase
    ib0         - float, average beam current, A
    phib_deg    - float, beam-accelerating phase (0 for on-crest), deg
    beam_ids    - int, beam starting sample index
    beam_ide    - int, beam ending sample index
    roQ_or_RoQ  - r/Q of Linac or R/Q of circular accelerator, Ohm (see the
                  note in section 1.4.7)
    QL          - float, loaded quality factor of the cavity
    machine     - string, 'linac' or 'circular', used to select r/Q or R/Q
Output:
    status      - boolean, success (True) or fail (False)
    sp          - numpy array (complex), set point waveform (for controller)
    vf_ff       - numpy array (complex), feedforward waveform (for
                  controller)
    vb          - numpy array (complex), beam drive voltage waveform
    Tpul        - numpy array, time array for the waveforms
```

1.6.8 ADRC_controller

ADRC_controller(half_bw, pole_scale = 50.0)

```
Generate the continuous ADRC controller. We assume that the system gain has
been normalized to 1 and the system phase corrected to 0. Therefore, the
referred cavity equation is
        dvc/dt + (half_bw - 1j * detuning)*vc = half_bw * vd
where vc is the vector of cavity voltage and vd is the vector of cavity
drive (in principle vd = 2 * beta * vf / (beta + 1)). (Refer to LLRF Book
section 4.2.3).

Input:
    half_bw    - float, half bandwidth of the cavity, rad/s
    pole_scale - float, define the pole location of the observer
```

```
Output:
    status                 - boolean, success (True) or fail (False)
    Aobc, Bobc, Cobc, Dobc - numpy matrix (complex), continuous ADRC
                             observer
    b0                     - float, ADRC gain
```

1.6.9 ADRC_control_step

ADRC_control_step(Akd, Bkd, Ckd, Dkd, Aobd, Bobd, b0, sp_step, vc_step, vd_step, state_k0, state_ob0, ff_step = 0.0, apply_to_err = False)

```
Controller execute for one step based on the controller's discrete state-
space equation including the ADRC observer (Refer to LLRF Book Fig. 4.6).

Input:
    Akd, Bkd, Ckd, Dkd - numpy matrix (complex), discrete RF controller
    Aobd, Bobd         - numpy matrix (complex), discrete ADRC observer (C,D
                         not needed)
    b0                 - float, ADRC gain
    sp_step            - complex, cavity voltage setpoint of this time step
    vc_step            - complex, cavity voltage meas. of this time step
    vd_step            - complex, cavity drive of the last time step
    state_k0           - numpy matrix (complex), controller state of the
                         last step
    state_ob0          - numpy matrix (complex), ADRC observer state of the
                         last step
    ff_step            - complex, feedforward of this step
    apply_to_err       - boolean, True to apply ADRC to error, or apply to
                         whole cavity voltage
Output:
    status             - boolean, success (True) or fail (False)
    ctrl_step          - complex, overall output of the controller of this
                         step
    ctrl_out           - complex, feedback output (exclude feedforward wrt
                         ctrl_step)
    state_k            - numpy matrix (complex), controller state of this
                         step (should input to the next execution of this
                         function)
    state_ob           - numpy matrix (complex), ADRC observer state of this
                         step (should input to the next execution of this
                         function)
    f                  - complex, estimated general disturbance by the ADRC
                         observer

Question: in the second equation, shall we use "state_k0" or "state_k"?

Note: 1. Looks like ADRC can mitigate the instability caused by the cavity's
         passband modes, and then no notch filter is required to filter the
         cavity voltage measurement
      2. Basic ADRC is perfect with a proportional controller - resulting in
         zero steady-state error. Looks like with other controller (e.g., PI
         control), ADRC needs to be revised for good performance
      3. Adding feedforward to the basic ADRC also causes some problem, such
         as the steady state error becomes nonzero
      4. The solution to handle the problems of point 2 and 3 above is to
         apply ADRC to the errors of the cavity drive and voltage. See the
         paper:
         S. Zhao et al, Tracking and disturbance rejection in non-minimum
         phase systems, Proceedings of the 33rd Chinese Control Conference,
         pp. 3834-3839, July 28-30, 2014, Nanjing, China
```

1.6.10 AFF_timerev_lpf

AFF_timerev_lpf(vfb, fcut, fs, vff_cor = None)

```
Time-reversed low-pass filter, we only apply the first order IIR low-pass,
which gives up to 90 degrees phase lead (refer to LLRF Book section 4.5.1).

Input:
    vfb     - numpy array (complex), feedback control waveform
    fcut    - float, cut-off frequency of the low-pass filter, Hz
    fs      - float, sampling frequency, Hz
    vff_cor - numpy array (complex), buffer storing the filtered waveform
Output:
    status  - boolean, success (True) or fail (False)
    vff_cor - numpy array (complex), feedforward correction waveform
```

1.6.11 AFF_ilc_design

AFF_ilc_design(h, pulw, P = None, Q = None)

```
Adaptive feedforward with optimal iterative learning control (ILC) (refer to
LLRF Book section 4.5.2).

Input:
    h      - numpy array (complex), impulse response
    pulw   - int, pulse width as number of points
    P, Q   - numpy matrix, positive-definite weight matrices
Output:
    status - boolean, success (True) or fail (False)
    L      - numpy matrix (complex), gain matrix of ILC
```

1.6.12 AFF_ilc

AFF_ilc(vc_err, L)

```
Apply the ILC (refer to LLRF Book section 4.5.2).
Input:
    vc_err  - numpy array (complex), error of the cavity voltage waveform
    L       - numpy matrix (complex), gain matrix of ILC
Output:
    vff_cor - numpy array (complex), feedforward correction waveform
```

1.6.13 resp_inv_svd

resp_inv_svd(R, singular_val_filt = 0.0)

```
Response matrix inversion with SVD (refer to Beam Control Book [2] section
2.4.2).
Input:
    R                 - numpy matrix, response matrix
    singular_val_filt - float, threshold of singular values, the ones
                        smaller or equal to it will be discarded
Output:
    Rinv - numpy matrix, inversion of the response matrix
```

1.6.14 resp_inv_lsm

resp_inv_lsm(R, regu = 0.0)

```
Response matrix inversion with least-square method (refer to Beam Control
Book section 2.4.3).
Input:
    R    - numpy matrix, response matrix
    regu - float, regularization factor (should > 0)
Output:
    Rinv - numpy matrix, inversion of the response matrix
```

1.7 rf_det_act Routines

1.7.1 noniq_demod

noniq_demod(raw_wf, n, m = 1)

```
Non-I/Q demodulation (refer to LLRF Book section 5.2.2).
Input:
    raw_wf - numpy array, 1-D array of the raw waveform
    m, n   - integer, non-I/Q parameters (n samples cover m IF cycles)
Output:
    status - boolean, success (True) or fail (False)
    I, Q   - numpy array, I/Q waveforms
```

This function demodulates the input waveform into I and Q components. The algorithm follows Eq. (5.22) of the LLRF Book. However, the implementation applies some tricks originally aiming at implementing this algorithm in FPGA. See Figure 7. The “non-IQ Coefficients” are

$$C_I = \frac{2}{n}\sin(l\Delta\varphi),\ C_Q = \frac{2}{n}\cos(l\Delta\varphi),\ l = 0,...,n-1 \quad \text{where } \Delta\varphi = 2\pi\ f_{IF} / f_{Clock} = 2\pi m/n. \tag{14}$$

Note that in Figure 7, all the components are synchronized by the same clock as ADC.

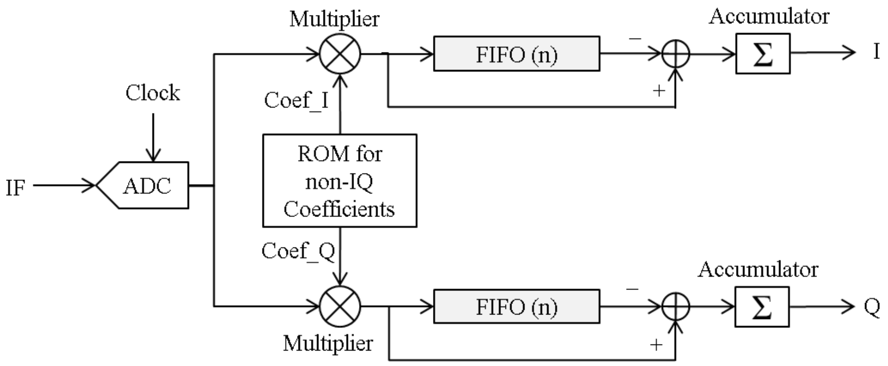

Figure 7: Implementation of non-I/Q demodulation algorithm (with minimum usage of multipliers)

1.7.2 twop_demod

twop_demod(raw_wf, f_if, fs)

```
Demodulation with two points (refer to LLRF Book section 5.2.2).
Input:
    raw_wf - numpy array, raw waveform to be demodulated
    f_if   - float, IF frequency, Hz
    fs     - float, sampling frequency, Hz
Output:
    status - boolean, success (True) or fail (False)
    I, Q   - numpy array, I/Q waveforms
```

1.7.3 asyn_demod

asyn_demod(raw_wf, ref_wf)

```
Asynchronous demodulation (refer to LLRF Book section 5.2.5).
Input:
    raw_wf      - numpy array, signal waveform to be demodulated
    ref_wf      - numpy array, samples of the RF reference signal
Output:
    status      - boolean, success (True) or fail (False)
    I, Q        - numpy array, I/Q waveforms of final demodulation
    Isig, Qsig  - numpy array, I/Q waveforms of raw_wf (with inaccurate
                  phase)
```

```
        Iref, Qsig  - numpy array, I/Q waveforms of ref_wf (with inaccurate
                      phase)
```

This function demodulates the input raw waveform using the reference tracking method. Note that the implementation here only works for buffered waveforms. You need to input the whole waveform of the signal to be demodulated and the waveform of the reference signal sampled at the same time. Modifications are needed if this algorithm is applied to streamed samples (e.g., in FPGA). Here summarize the key points in the implementation:

- The carrier (IF) frequency of the signal is estimated via FFT.
- The function `twop_demod` is used to demodulate both waveforms.
- The reference phase is used to detect the phase slope (representing the error of the estimated carrier frequency), which is used to correct the phase of the signal to be demodulated.

1.7.4 **self_demod_ap**

self_demod_ap(raw_wf, n = 1)

```
    Self-demodulation with Hilbert transform (later may implement padding to
    mitigate the edge effects).

    Input:
        raw_wf - numpy array, signal waveform to be demodulated
        n      - int, number of points covering full cycles (for coherent
                 sampling)
    Output:
        status - boolean, success (True) or fail (False)
        A, P   - numpy array, amplitude and phase waveforms, P in degree
```

For a real time-domain signal $y(t)$, its Hilbert transform is defined as

$$H\left[y(t)\right] := y^H(t) = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{y(\tau)}{t-\tau}d\tau . \tag{15}$$

Then for signal $y(t)$, a complex signal can be constructed as

$$\mathbf{y}(t) = y(t) + jy^H(t). \tag{16}$$

From it the amplitude and phase of the signal $y(t)$ can be calculated as

$$A_y(t) = \sqrt{\left[y(t)\right]^2 + \left[y^H(t)\right]^2},\ \varphi_y(t) = \tan^{-1}\left[\frac{y^H(t)}{y(t)}\right]. \tag{17}$$

This function can be used to estimate the amplitude and phase of a waveform without knowing the sampling parameters such as the carrier frequency or the sampling frequency. With the estimated amplitude and phase, the amplitude and phase jitter of the waveform can be examined.

1.7.5 **iq2ap_wf**

iq2ap_wf(I, Q)

```
    I/Q to A/P waveforms.
    Input:
        I, Q - numpy array, I/Q waveforms
    Output:
        A, P - numpy array, amplitude/phase waveforms, P in degree
```

1.7.6 **ap2iq_wf**

ap2iq_wf(A, P)

```
    A/P to I/Q waveforms.
```

```
Input:
    A, P - numpy array, amplitude/phase waveforms, P in degree
Output:
    I, Q - numpy array, I/Q waveforms
```

1.7.7 norm_phase

norm_phase(P, cmd = '+-180')

```
Normalize phase to +-180 degree or between 0 and 360 degree.
Input:
    P      - float, phase in degree
    cmd    - string '+-180' or '0to360'
Output:
    Pnorm - float, normalized phase, degree
```

1.7.8 pulse_info

pulse_info(pulse, threshold = 0.1)

```
Get information of a pulse.

Input:
    pulse     - numpy array, the pulse data
    threshold - float, threshold for edge detection
Output:
    offs      - int, offset id of the pulse
    pulw      - int, pulse width as number of samples
    peak      - float, peak magnitude of the pulse

To be done:
    a. detect the rise time and falling time
    b. detect the flattop region and average
    c. calculate the energy in the pulse
```

1.8 rf_fit Routines

1.8.1 fit_sincos

fit_sincos(X_rad, Y, target = 'cos')

```
Fit the sine or cosine function
  y = A*cos(x + phi) + c = (A*cos(phi)) * cos(x) - (A*sin(phi)) * sin(x) + c
  y = A*sin(x + phi) + c = (A*cos(phi)) * sin(x) + (A*sin(phi)) * cos(x) + c

Input:
    X_rad   - numpy array, phase array in radian
    Y       - numpy array, value of the sine or cosine function
    target  - 'sin' or 'cos', determine which function to fit to
Output:
    status  - boolean, success (True) or fail (False)
    A       - float, amplitude of the function
    phi_rad - float, phase of the function, rad
    c       - float, offset of the function
```

1.8.2 fit_circle

fit_circle(X, Y)

```
Fit a circle. Given x, y data points and find x0, y0 and r
    (x - x0)^2 + (y - y0)^2 = r^2

Input:
    X      - numpy array, x-coordinate of the points
    Y      - numpy array, y-coordinate of the points
Output:
```

```
status - boolean, success (True) or fail (False)
x0, y0 - float, center coordinate of the circle
r      - float, radius of the circle
```

1.8.3 fit_ellipse

fit_ellipse(X, Y)

```
Fit an ellipse. Get the general ellipse function and its characteristics,
including semi-major axis "a", semi-minor axis "b", center coordinates "(x0,
y0)" and rotation angle "sita" (the angle from the positive horizontal axis
to the ellipse's major axis).
The general ellipse equation is:
        A*X^2 + B*X*Y + C*Y^2 + D*X + E*Y + F = 0
When making the fitting, we divide the equation by C to normalize the
coefficient of Y^2 to 1 and move it to the right side of the fitting
equation. The ellipse is derived with the following steps:
 1. Define a canonical ellipse:
        X1 = a * cos(phi)
        Y1 = b * sin(phi)
    where phi is a phase vector covering from 0 to 2*pi
 2. Rotate the canonical ellipse:
        X2 = X1 * cos(sita) - Y1 * sin(sita)
        Y2 = X1 * sin(sita) + Y1 * cos(sita)
 3. Add offset to the rotated ellipse:
        X = X2 + x0
        Y = Y2 + y0

See the webpage: https://en.wikipedia.org/wiki/Ellipse

Input:
    X      - numpy array, x-coordinate of the points
    Y      - numpy array, y-coordinate of the points
Output:
    status - boolean, success (True) or fail (False)
    Coef   - list, coefficients derived from the least-square fitting
    a      - float, semi-major
    b      - float, semi-minor
    x0, y0 - float, center of the ellipse
    sita   - float, angle of the ellipse (see above), rad
```

1.8.4 fit_Gaussian

fit_Gaussian(X, Y)

```
Fit a Gaussian distribution function.
See the webpage: https://pythonguides.com/scipy-normal-distribution/

Input:
    X      - numpy array, x-coordinate of the points
    Y      - numpy array, y-coordinate of the points
Output:
    status - boolean, success (True) or fail (False)
    a      - float, magnitude scale of the un-normalized distribution
    mu     - float, mean value
    sigma  - float, standard deviation
```

1.9 rf_noise Routines

1.9.1 calc_psd_coherent

calc_psd_coherent(data, fs, bit = 0, n_noniq = 1, plot = False)

```
Calculate the power spectral density of the input waveform (coherent
sampling) (refer to LLRF Book section 6.1.2).
```

```
Input:
    data         - numpy array, 1-D array of the raw waveform
    fs           - float, sampling frequency, Hz
    bit          - int, number of bit of data. If bit = 0, do not scale data
    n_noniq      - int, non-I/Q parameters (n samples cover a full cycle)
    plot         - boolean, True for plot the spectrum
Output:
    freq         - numpy array, frequency waveform, Hz
    amp_resp     - numpy array, amplitude response, dBFS/Hz
    pha_resp     - numpy array, phase response, degree
    signal_freq - float, signal frequency, Hz
    signal_mag   - float, signal level, dBFS
    noise_flr    - float, noise floor, dBFS/Hz
    snr          - float, signal-to-noise ratio, dB
    bin_db       - float, offset for an FFT bin freq space, dB
    status       - boolean, success (True) or fail (False)

Note: The unit dBFS is for ADC raw data, if for phase in radian, it should
be replaced by dBrad^2; if for relative amplitude, should be replaced by dB
```

1.9.2 calc_psd

calc_psd(data, fs, bit = 0, plot = False)

```
Calculate the power spectral density of the input waveform (nominal sampling)
(refer to LLRF Book section 6.1.2).

Input:
    data         - numpy array, 1-D array of the raw waveform
    fs           - float, sampling frequency, Hz
    bit          - int, number of bit of data. If bit = 0, do not scale data
    plot         - boolean, True for plot the spectrum
Output:
    freq         - numpy array, frequency waveform, Hz
    amp_resp     - numpy array, amplitude response, dBFS/Hz
    pha_resp     - numpy array, phase response, degree
    signal_freq - float, signal frequency, Hz
    signal_mag   - float, signal level, dBFS
    noise_flr    - float, noise floor, dBFS/Hz
    snr          - float, signal-to-noise ratio, dB
    bin_db       - float, offset for an FFT bin freq space, dB
    enbw_db      - float, offset for an FFT bin freq space (correct
                   windowing), dB
    status       - boolean, success (True) or fail (False)

Note: The unit dBFS is for ADC raw data, if for phase in radian, it should
be replaced by dBrad^2; if for relative amplitude, should be replaced by dB
```

1.9.3 rand_unif

rand_unif(low = 0.0, high = 1.0, n = 1)

```
Produce random number within a certain range.
Input:
    n    - int, number of output
    low  - float, low limit of the data
    high - float, high limit of the data
Output:
    val  - if n = 1, it is a float number, if n > 1, it is a np array
```

1.9.4 gen_noise_from_psd

gen_noise_from_psd(freq_vector, pn_vector, fs, N)

```
Generate noise series from DSB PSD.
Input:
    freq_vector - numpy array, offset frequency from carrier, Hz
```

```
        pn_vector   - numpy array, DSB noise PSD, dBrad^2/Hz for phase noise
        fs          - float, sampling frequency, Hz
        N           - float, number of samples
    Output:
        status      - boolean, success (True) or fail (False)
        pn_series   - numpy array, time series of phase/amplitude noise
        freq_p      - numpy array, interpreted freq_vector input
        pn_p        - numpy array, interpreted pn_vector input
```

This function generates a time series with its PSD spectrum the same as that given by input parameters. The implementation follows the algorithm below:

a. Derive the PSD at the N frequency points (corresponding to N data points, covering from f_s/N to f_s) by linearly interpreting the given PSD spectrum with the frequency and PSD both in the logarithm scale.
b. From the interpreted PSD, calculate the DFT magnitude of the positive frequencies by inversing the formula of the periodogram method. See Eq. (6.15) of the LLRF Book. Then derive the complex DFT at positive frequencies by assigning a random phase to each point.
c. Construct the full DFT from 0 to f_s (the DFT from $f_s/2$ to f_s is the conjugate of the image frequencies from $f_s/2$ down to 0), and calculate the time series with the inversed DFT (IDFT).

1.9.5 `calc_rms_from_psd`

`calc_rms_from_psd(freq_vector, pn_vector, freq_start, freq_end, fs, N)`

```
    Calculate the rms value from PSDs.

    Input:
        freq_vector - numpy array, offset frequency from carrier, Hz
        pn_vector   - numpy array, DSB noise PSD, dBrad^2/Hz for phase noise
        freq_start  - float, starting frequency for integration, Hz
        freq_end    - float, ending frequency for integration, Hz
        fs          - float, sampling frequency, Hz
        N           - int, number of points in the frequency integration range
    Output:
        status      - boolean, success (True) or fail (False)
        freq_p      - numpy array, interpreted freq_vector input
        pn_p        - numpy array, interpreted pn_vector input
        jitter_p    - numpy array, integrated jitter starting from freq_start to
                      different freq_p element values
```

This function interprets the given PSD spectrum (similar to `gen_noise_from_psd`) and calculates the RMS jitter by integrating the noise power from `freq_start` to different ending frequencies up to `freq_end`.

1.9.6 `notch_filt`

`notch_filt(wf, fnotch, Q, fs, b = None, a = None)`

```
    Apply notch filter to the signal.

    Input:
        wf      - numpy array, the waveform to be notch filtered
        fnotch  - float, frequency to be notched, Hz
        Q       - float, quality factor of the notch filter
        fs      - float, sampling frequency, Hz
        b, a    - numpy arrays filter coefficients. Note that the user can
                  either input "b, a", or "fnotch, Q, fs"
    Output:
        status  - boolean, success (True) or fail (False)
        wf_f    - numpy array, filtered waveform
        b, a    - numpy array, filter coefficients. If the same filter is used
                  to filter multiple waveforms, we can use it to avoid repeat
                  the filter synthesis
```

1.9.7 design_notch_filter

design_notch_filter(fnotch, Q, fs)

```
Design the notch filter (return a discrete filter).

Input:
    fnotch  - float, frequency to be notched, Hz
    Q       - float, quality factor of the notch filter
    fs      - float, sampling frequency, Hz
Output:
    status         - boolean, success (True) or fail (False)
    Ad, Bd, Cd, Dd - numpy matrix, discrete notch filter
```

1.9.8 filt_step

filt_step(Afd, Bfd, Cfd, Dfd, in_step, state_f0)

```
Apply a step of the filter.

Input:
    Afd, Bfd, Cfd, Dfd - numpy matrix, discrete filter model
    in_step            - float or complex, input of the time step
    state_f0           - numpy matrix, state of the filter of last step
Output:
    status   - boolean, success (True) or fail (False)
    out_step - float or complex, filter output of this time step
    state_f  - numpy matrix, state of the filter of this step, should input
               to the next execution
```

1.9.9 moving_avg

moving_avg(wf_in, n)

```
Moving average without compensating for the group delay.

Input:
    wf_in   - numpy array, input waveform
    n       - int, point number of moving average
Output:
    status  - boolean, success (True) or fail (False)
    wf_out  - numpy array, output waveform
```

1.10 rf_sim Routines

1.10.1 cav_ss

cav_ss(half_bw, detuning = 0.0, beta = 1e4, passband_modes = None, plot = False, plot_pno = 1000)

```
Derive the continuous state-space equation of the cavity:
 - include pass-band modes
 - we assume the half-bandwidth and detuning of the fundamental mode are
   constant
 - we output two models: one for RF drive voltage and the other for beam
   drive voltage (beam only interacts with the fundamental mode of the
   cavity)
(Refer to LLRF Book section 3.3.7 and 3.4.3).

Input:
    half_bw        - float, half bandwidth of the cavity (constant), rad/s
    detuning       - float, detuning of the cavity (constant), rad/s
    beta           - float, input coupling factor (needed for NC cavities;
                     for SC cavities, can use the default value, or you can
                     specify it if more accurate result is needed)
    passband_modes - dict, with the following items:
```

```
        'freq_offs' : list, offset frequencies of the modes, Hz
        'gain_rel'  : list, relative gain wrt fundamental mode
        'half_bw'   : list, half bandwidth of the mode, rad/s
    plot            - boolean, enable the plot of frequency response
    plot_pno        - int, number of point in the plot
Output:
    status              - boolean, success (True) or fail (False)
    Arf, Brf, Crf, Drf - numpy matrix (complex), continuous cavity model for
                         RF drive
    Abm, Bbm, Cbm, Dbm - numpy matrix (complex), continuous cavity model for
                         beam drive
```

1.10.2 `cav_ss_mech`

Not yet implemented. We suppose to implement a cavity model including mechanical oscillations. This is to simulate the superconducting cavities with Lorenz force detuning and microphonics.

1.10.3 `cav_impulse`

cav_impulse(half_bw, detuning, Ts, order = 20)

```
Derive the impulse response from the cavity equation. We assume that the
system gain has been normalized to 1 and the system phase corrected to 0.
Therefore, the referred cavity equation is
        dvc/dt + (half_bw - 1j * detuning)*vc = half_bw * vd
where vc is the vector of cavity voltage and vd is the vector of cavity
drive (in principle vd = 2 * beta * vf / (beta + 1)). (Refer to LLRF Book
section 4.5.2).

Input:
    half_bw   - float, constant half bandwidth of the cavity, rad/s
    detuning - float, constant detuning of the cavity, rad/s
    Ts        - float, sampling time, s
    order     - int, order of the impulse response
Output:
    status    - boolean, success (True) or fail (False)
    h         - numpy array (complex), impulse response
```

1.10.4 `sim_ncav_pulse`

sim_ncav_pulse(Arfc, Brfc, Crfc, Drfc, vf, Ts, Abmc = None, Bbmc = None, Cbmc = None, Dbmc = None, vb = None)

```
Simulate the cavity response to a pulsed RF drive voltage and beam drive
voltage. This function is for normal conducting cavities with constant QL
and detuning.

Input:
    Arfc, Brfc, Crfc, Drfc - numpy matrix (complex), continuous cavity model
                             for RF drive
    vf                     - numpy array (complex), cavity forward voltage
                             (calibrated to the cavity probe signal reference
                             plane)
    Ts                     - float, sampling frequency, Hz
    Abmc, Bbmc, Cbmc, Dbmc - numpy matrix (complex), continuous cavity model
                             for beam drive
    vb                     - numpy array (complex), beam drive voltage
                             (calibrated to the cavity probe signal reference
                             plane)
Output:
    status - boolean, success (True) or fail (False)
    T      - numpy array, time waveform, s
    vc     - numpy array (complex), cavity voltage waveform
    vr     - numpy array (complex), cavity reflected voltage waveform
```

1.10.5 sim_ncav_step

sim_ncav_step(Arfd, Brfd, Crfd, Drfd, vf_step, state_rf0, Abmd = None, Bbmd = None, Cbmd = None, Dbmd = None, vb_step = None, state_bm0 = None)

```
Simulate the cavity response for a time step using the discrete cavity
state-space function.

Input:
    Arfd, Brfd, Crfd, Drfd - numpy matrix (complex), discrete cavity model
                             for RF drive
    vf_step                - complex, cavity forward voltage of this step
    state_rf0              - numpy matrix (complex), state of RF response
                             model of last step
    Abmd, Bbmd, Cbmd, Dbmd - numpy matrix (complex), discrete cavity model
                             for beam drive
    vb_step                - complex, beam drive voltage of this step
    state_bm0              - numpy matrix (complex), state of beam response
                             model of last step
Output:
    status   - boolean, success (True) or fail (False)
    vc_step  - complex, cavity voltage of this step
    vr_step  - complex, cavity reflected voltage of this step
    state_rf - numpy matrix (complex), state of RF response model of this
               step (should input to next execution)
    state_bm - numpy matrix (complex), state of beam response model of this
               step (should input to next execution)
```

1.10.6 sim_ncav_step_simple

sim_ncav_step_simple(half_bw, detuning, vf_step, vb_step, vc_step0, Ts, beta = 1e4)

```
Simulate the cavity response for a time step using the simple discrete
cavity equation (Euler method for discretization).

Input:
    half_bw   - float, half bandwidth of the cavity (constant), rad/s
    detuning  - float, detuning of the cavity (constant), rad/s
    vf_step   - complex, cavity forward voltage of this step
    vb_step   - complex, beam drive voltage of this step
    vc_step0  - complex, cavity voltage of the last step
    Ts        - float, sampling time, s
    beta      - float, input coupling factor (needed for NC cavities; for SC
                cavities, can use the default value, or you can specify it if
                more accurate result is needed)
Output:
    status   - boolean, success (True) or fail (False)
    vc_step  - complex, cavity voltage of this step
    vr_step  - complex, cavity reflected voltage of this step
```

This function discretize the cavity equation with the Euler method. The accuracy will become bad if the sampling time T_s is close to the time constant of the cavity dynamics (e.g., transient caused by large detuning or the dynamics of pass-band modes).

1.10.7 rf_power_req

rf_power_req(f0, vc0, ib0, phib, Q0, roQ_or_RoQ, QL_vec = None, detuning_vec = None, machine = 'linac', plot = False)

```
Plot the steady-state forward and reflected power for given cavity voltage,
beam current and beam phase, as function of loaded Q and detuning. The beam
phase is defined to be zero for on-crest acceleration (refer to LLRF Book
section 3.3.9).
```

```
Input:
    f0           - float, RF operating frequency, Hz
    vc0          - float, desired cavity voltage, V
    ib0          - float, desired average beam current, A
    phib         - float, desired beam phase, degree
    Q0           - float, unloaded quality factor (for SC cavity, give it a
                     very high value like 1e10)
    roQ_or_RoQ    - float, cavity r/Q of Linac or R/Q of circular accelerator,
                     Ohm (see the note in section 1.5.7)
    QL_vec       - numpy array, QL vector for power calculation
    detuning_vec - numpy array, detuning at which to evaluated the powers
    machine      - string, 'linac' or 'circular', used to select r/Q or R/Q
    plot         - boolean, enable/disable the plotting
Output:
    status - boolean, success (True) or fail (False)
    Pfor   - dictionary, keyed by detuning, forward power at different QL
    Pref   - dictionary, keyed by detuning, reflected power at different QL
```

1.10.8 opt_QL_detuning

opt_QL_detuning(f0, vc0, ib0, phib, Q0, roQ_or_RoQ, machine = 'linac', cav_type = 'sc')

```
Derived the optimal loaded Q and detuning (refer to LLRF Book section 3.3.9).

Input:
    f0           - float, RF operating frequency, Hz
    vc0          - float, desired cavity voltage, V
    ib0          - float, desired average beam current, A
    phib         - float, desired beam phase, degree
    Q0           - float, unloaded quality factor (for SC cavity, give it a
                     very high value like 1e10)
    roQ_or_RoQ    - float, cavity r/Q of Linac or R/Q of circular accelerator,
                     Ohm (see the note in section 1.5.7)
    machine      - string, 'linac' or 'circular', used to select r/Q or R/Q
    cav_type     - string, 'sc' for superconducting or 'nc' for normal
                     conducting
Output:
    status       - boolean, success (True) or fail (False)
    QL_opt       - float, optimal loaded quality factor
    dw_opt       - float, optimal detuning, rad/s
    beta_opt     - float, optimal input coupling factor
```

1.11 rf_misc Routines

1.11.1 save_mat

save_mat(data_dict, file_name)

```
Save a dictionary into Matlab file.
Input:
    data_dict - data dictionary
    file_name - full file name including path
Output:
    status    - boolean, success (True) or fail (False)
```

1.11.2 load_mat

load_mat(file_name, to_list = False)

```
Load a Matlab data file into a dict. this function should be called instead
of direct spio.loadmat as it cures the problem of not properly recovering
python dictionaries from mat files. It calls the function check keys to cure
all entries that are still mat-objects.

Input:
```

```
    file_name - full file name including path
Output:
    status    - boolean, success (True) or fail (False)
    data      - dict, contain the data with the key the variable name in
                Matlab

Note: This implementation can be found here:
https://stackoverflow.com/questions/7008608/scipy-io-loadmat-nested-structures-i-e-dictionaries
```

1.11.3 get_curtime_str

get_curtime_str(format = '%Y-%m-%d %H:%M:%S')

```
Get the current time string.
Input:
    format   - string of format
Output:
    time_str - string, the time string
```

1.11.4 get_bit

get_bit(data, bit_id = 0)

```
Get a bit from the data, the bit index starts from 0.
Input:
    data   - int, the input data
    bit_id - int, index of the bit
Output:
    bit    - int, 1 or 0
```

1.11.5 add_tf

add_tf(num1, den1, num2, den2)

```
Add two transfer functions in num/den polynomial format:
    A/B+C/D = (AD+BC)/BD

Input:
    num1, den1  - list, polynomial coefficients of transfer function1
    num2, den2  - list, polynomial coefficients of transfer function2
Output:
    num_sum, den_sum - list, polynomial coefficients of the sum
```

1.11.6 plot_ellipse

plot_ellipse(n, a = 1.0, b = 1.0, x0 = 0.0, y0 = 0.0, sita = 0.0, plot = False)

```
Draw an ellipse (see "fit_ellipse" function in "rf_fit" module).

Input:
    n      - int, number of points
    a      - float, semi-major
    b      - float, semi-minor
    x0, y0 - float, center of the ellipse
    sita   - float, angle of the ellipse, rad
    plot   - boolean, True for enabling displaying
Output:
    status - boolean, True for success
    X, Y   - numpy array, points on the ellipse
```

1.11.7 plot_Guassian

plot_Guassian(n, a = 1.0, mu = 0.0, sigma = 1.0, plot = False)

```
Draw Gaussian distribution (see "fit_Guassian" function in "rf_fit" module).

Input:
    n      - int, number of points
    a      - float, magnitude scale factor
    mu     - float, mean value
    sigma - float, standard deviation
    plot  - boolean, True for enabling displaying
Output:
    status - boolean, True for success
    X, Y   - numpy array, points on the curve
```

1.12 rf_plot Routines

The routines in this module are used internally by other modules. The purpose is to avoid loading the `matplotlib` library in other modules. Therefore, if an application software does not need to plot the results, the `matplotlib` library does not need to be installed.

2 References

[1] S. Simrock and Z. Geng, Low-Level Radio Frequency Systems, Springer, 2022. https://link.springer.com/book/10.1007/978-3-030-94419-3

[2] Z. Geng and S. Simrock, Intelligent Beam Control in Accelerators, Springer, 2023. https://link.springer.com/book/10.1007/978-3-031-28597-4