microwave SQUID multiplexer user manual

v1.1

Justus A. Brevik

December 17, 2013

1 introduction

This document is designed to instruct a user on how to operate the microwave SQUID multiplexer (μMUX) using the ROACH software defined radio (SDR) system. The ROACH is operated using the python-based μMUX software suite, which includes umuxlib.py and tune.py.

2 firmware description

Need to fill this in.

3 hardware setup

Need to fill this in.

3.1 setting up the ROACH

The ROACH needs to be configured to talk to the DAQ computer. This will be done through the ethernet port on the ROACH (not the XPort), but it's likely the default IP address will need to be edited on the SD card that contains the file system for the ROACH .

There are two ways to edit the default filesystems. The easiest way is to remove the SD card from the ROACH and plug it in to a linux system (the file system seems to be incompatible with OS X) using an SD card reader. Then the files can be edited as described below. The other method is to connect the ROACH to the DAQ computer using an RS232 serial cable and null modem. Once the proper serial port on the DAQ is identified (this can be done using 'ls -lt /dev/serial/by-id') picocom can be used to communicate with the ROACH :
picocom /dev/your_device_here --baud=115200

Now the appropriate IP address must be configured in either '/etc/rcSimple' or '/etc/networks/'. It will be possible to ping and ssh directly into the ROACH as root once this has been properly configured. This is also a good time to set up '/etc/resolv.conf' to make sure that the ROACH can properly resolve hostnames, which may be necessary for the data transmission code.

4 software setup

The μMUX software suite must first be installed on the DAQ computer. The steps to install the code and necessary directories are:

1. check out the code in the the desired code directory (e.g. '∼/code/') using github. The github repository 'umux' includes umuxlib.py, tune.py, setupFG.py, default_frex.txt, roachIP.text and a directory of bof files.
   https://github.com/justusbrevik/umux

2. create the directories ('/data' and '/data/raw/') as sudo:
   mkdir /data/
   mkdir /data/raw/

3. move default_freqs.txt and roachIP.txt to '/data/raw/'

4. edit roachIP.txt to reflect the IP address or hostname of each ROACH

5. edit the function generator wrapper setupFG.py to reflect the interface needed for the function generator connected to the flux ramp lines

6. install ipython (or equivalent) and the modules needed to run the ROACH and katcp (install the corr module, then attempt to import corr and install modules as they fail to load)

7. copy the contents of the 'boffiles' directory to the roach:
   scp -r ./boffiles/ root@ip_your_roach:/boffiles/

5 loading the μMUX python module

The μMUX python module (umuxlib.py) can then be loaded in at the python prompt using the code below. Instatiating the python module will automatically load in the IP address set for your roach in roachIP.txt, connect to the ROACH using katcp, configure the 512 MHz clock on the IF board and program the ROACH with the default bof file set in '__init__' of umuxlib.py. The μMUX python module is loaded and instantiated in ipython using:

```
import umuxlib.py    #import the library
um=umuxlib.util()    #instantiate the util subclass
```

6 finding resonances and VNA mode

The approximate resonant frequency of each channel must be entered in '/data/raw/default_freqs.txt' along with the local oscillator (LO) frequency that will be used by the IF board to mix the baseband DAC signals up to the GHz frequency range of interest. The LO frequency should be selected so that all resonant frequencies are within $\pm$256 MHz and no resonance is closer than $\sim$10 MHz. For the 35-channel μMUX version 10b chips this is easily accomplished by placing the LO 10 MHz below the lowest resonant frequency.

6.1 vna mode

The resonant frequencies are determined from the complex S21 data from a vector network analyzer (VNA) sweep. To avoid changing the RF connections between the ROACH and a commercial VNA, a rudimentary VNA is provided in umuxlib.py. The 'vna' function returns frequency points and

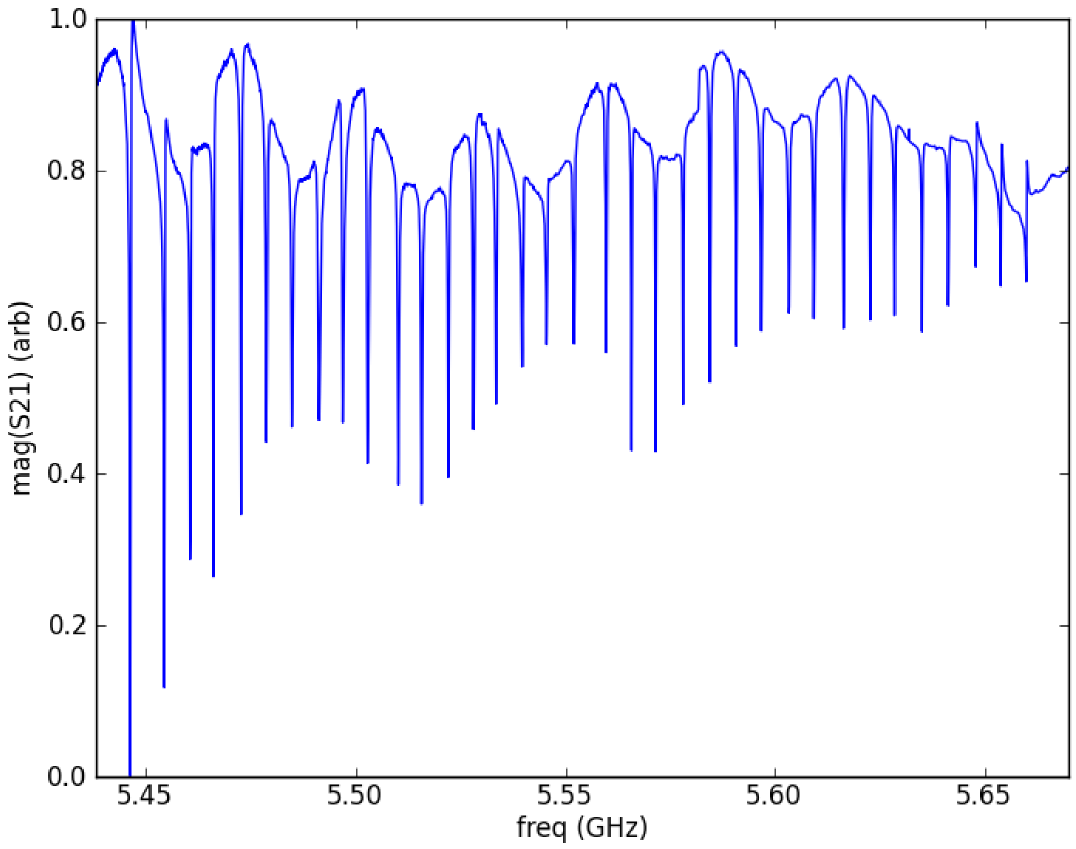

Figure 1: An example vna sweep from the 'vna' function. The data show the 35 resonance dips expected for the version 10b μMUX . The mag(S21) data that are returned have been heavily processed, so are returned scaled from 0–1 so that the magnitude of the IQ signal is not inferred from them.

the corresponding mag(S21) data. The function programs the output DAC with a comb of equally-spaced tones and then sweeps the local oscillator (LO) frequency by the spacing of the tones. There are fluctuations in the power of each tone which results in discontinuities in the mag(S21) data, which are corrected in the code. There is also a natural roll-off in power for tones further from the LO, which is corrected by removing a second-order polynomial from the data. Finally, the output magnitude data are rescaled to the interval 0–1 to remind the user that the output data do not reflect the actual magnitudes of the IQ data.

To operate the 'vna' function choose a range of frequencies to sweep over (start_freq to stop_freq) that is sufficient to include all of the resonators, but does not exceed the 256 MHz half-bandwidth of the DAC. The 'vna' function is run in ipython with:

```
fdat,mag=um.vna(start_freq,stop_freq)
```

NOTE: Using the 'vna' function will program the output DAC with a comb of equally-spaced tones. If you want to return to probing the resonators you will need to either tune again as in section 7 (slow) or reload a previous tuning as in section 8 (fast).

6.2 saving default frequencies

The umuxlib also includes a function 'um.findRes' which will automatically detect the resonant frequencies from the results of 'um.vna'. Finally, once the resonant frequencies and LO are determined they can be automatically written to '/data/raw/default_freqs.txt'. The code to perform these steps in ipython are:

```
freqs=um.findRes(fdat,mag)    #find the resonant frequencies from the vna sweep
um.saveFreqsAttens(loFreq,freqs,zeros([len(freqs)]),'/data/raw/default_freqs.txt')
```

7 tuning

The tune.py script is used to tune up the μMUX by selecting the frequency and amplitude of the readout tones – one per resonator – that maximizes the signal to noise of each readout channel. The script saves six diagnostic plots, one text file containing the optimal probe frequency and relative attenuation (power) setting for each readout tone ('tuned_freqs.txt') and a data file containing all of the tuning data ('tune.mat') to a tuning directory.

The basic steps the script follows are:

1. The approximate resonant frequencies are read in from '/data/raw/default_freqs.txt' along with the LO. The LO and output DAQ waveform are then programmed.

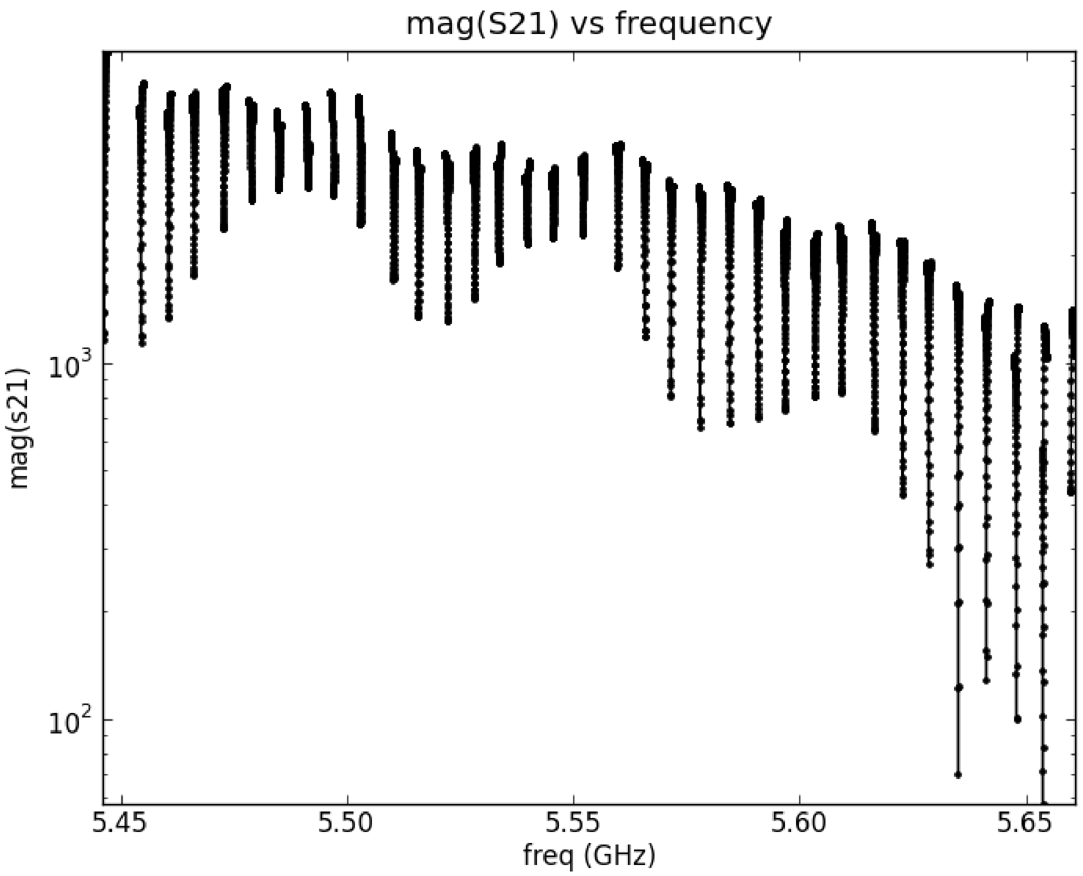

Figure 2: A tuning plot of the mag(S21) versus frequency. The rough resonant frequencies in '/data/raw/default_freqs.txt' are programmed and the LO is stepped ±1 MHz about the default LO frequency. This plot should be checked after tuning to verify that the off-resonance magnitudes are > 1000, and that the ripple from standing waves on the feedlines is adequately low. When the circulators at the RF input and output lines to the cryostat are removed, the ripple can be large and resonance dips can also appear as spikes. If the I and Q RF coxes between the ADC/DAC and IF board are swapped the resonances can also appear as spikes.

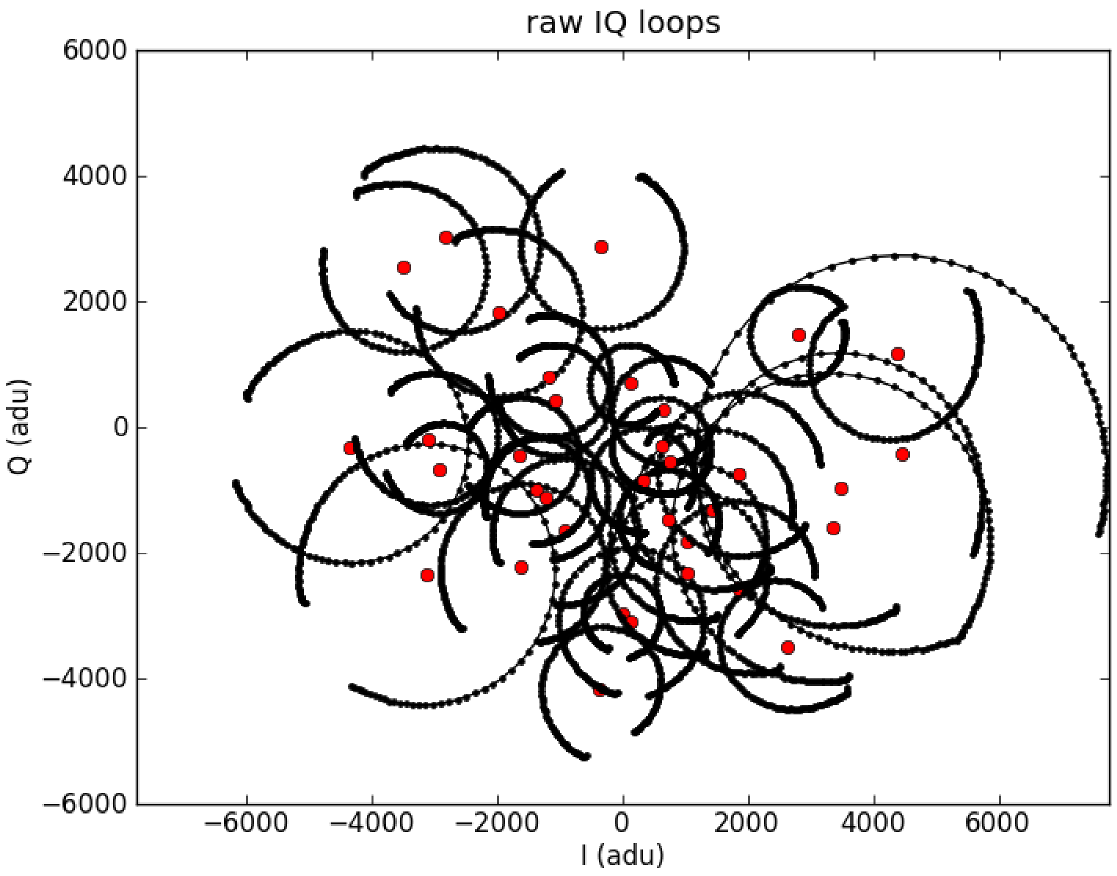

Figure 3: A tuning plot of the complex S21 parameters in the IQ plane acquired by LO sweep explained in figure 2. The complex S21 data form arcs or IQ loops when the frequency is swept close to resonance. Some IQ loops appear somewhat elliptical due to the effect of the cable delay on the frequency sweep. The rough centers of the IQ loops, indicated by the red circles, are initially determined from the frequency sweep so that the phase response of the channel can be determined with respect to the center of the IQ loop and not the IQ origin. Plots of IQ loops that form too closely to the origin (closer than 1000) may indicate an issue with the setup.

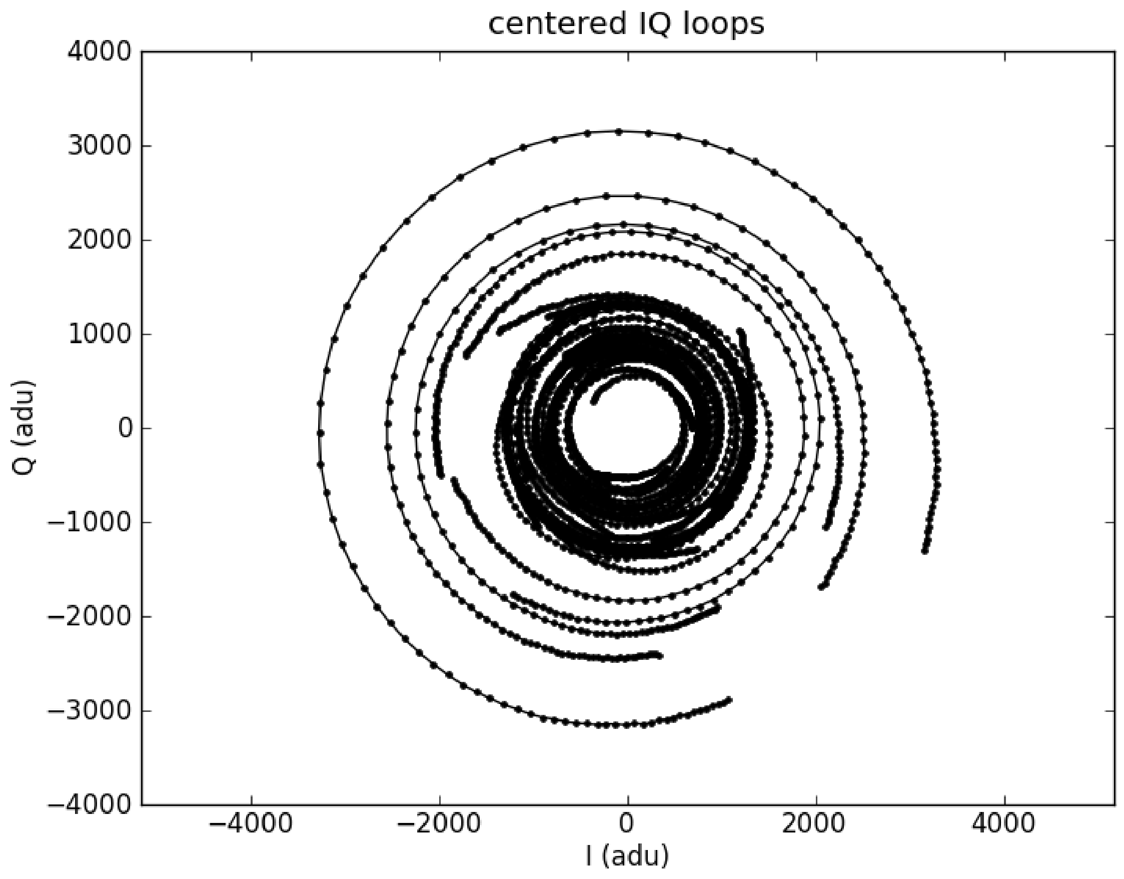

Figure 4: A tuning plot of the IQ loops after the centers indicated in figure 3 have been programmed in to firmware and the LO is swept again. The centers are subtracted from the raw IQ data so that the firmware calculates the phase response with respect to the center of the IQ loops.

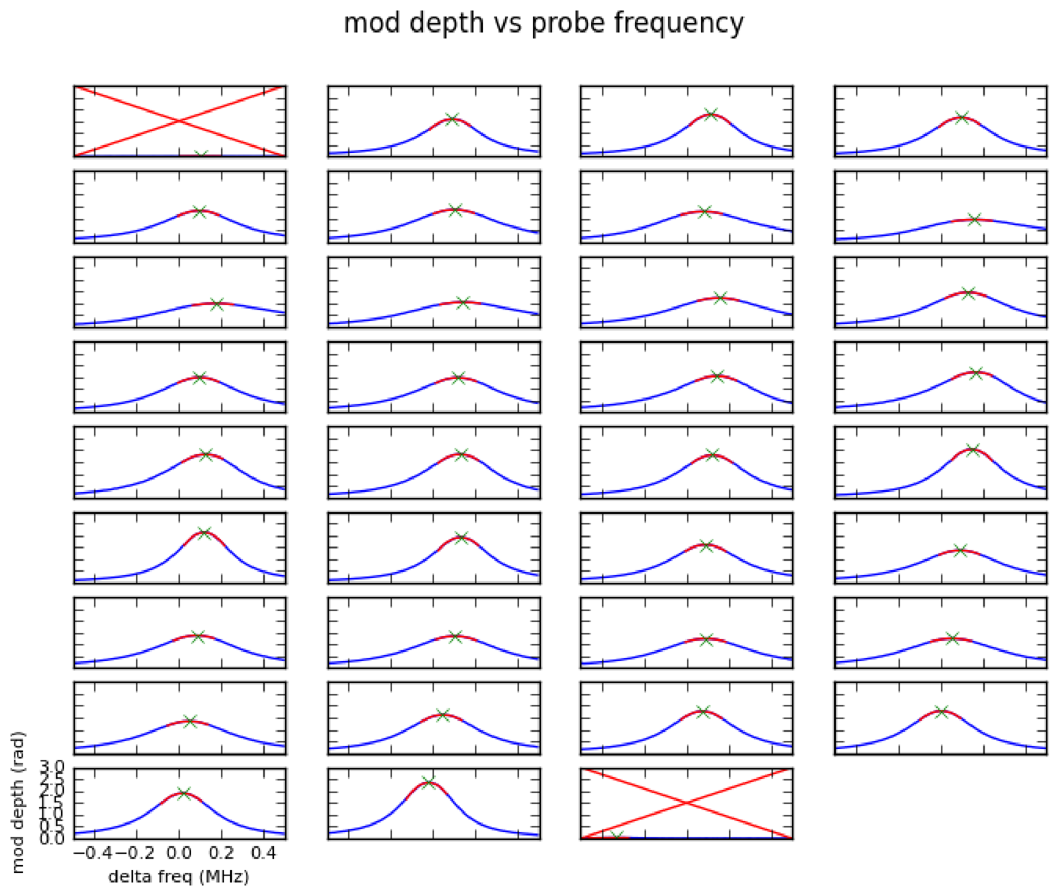

Figure 5: A tuning plot of the modulation depth of the SQUID response versus frequency. The frequency is the offset from the rough resonant frequency for each channel in '/data/raw/default_freqs.txt'. These plots are used to determine the optimal readout frequency for each channel, which correspond to the peaks. The peaks are not found from maxima of these plots, since noise or spikes could corrupt that determination. Instead the response versus frequency is fit with a parabola (shown in red) and the maxima of those fits (green crosses) are used. The first and last channels with the red crosses correspond to the dark resonator and dark SQUID and have no response to flux ramp modulation. This plot is the most useful for assessing the health of the tuning and the μMUX system. Flux trapping events can result in noisy features in this plot, diminished modulation depths or large shifts in the peak modulation depths of groups of channels. This plot should be monitored over time to monitor the functionality of the readout system.

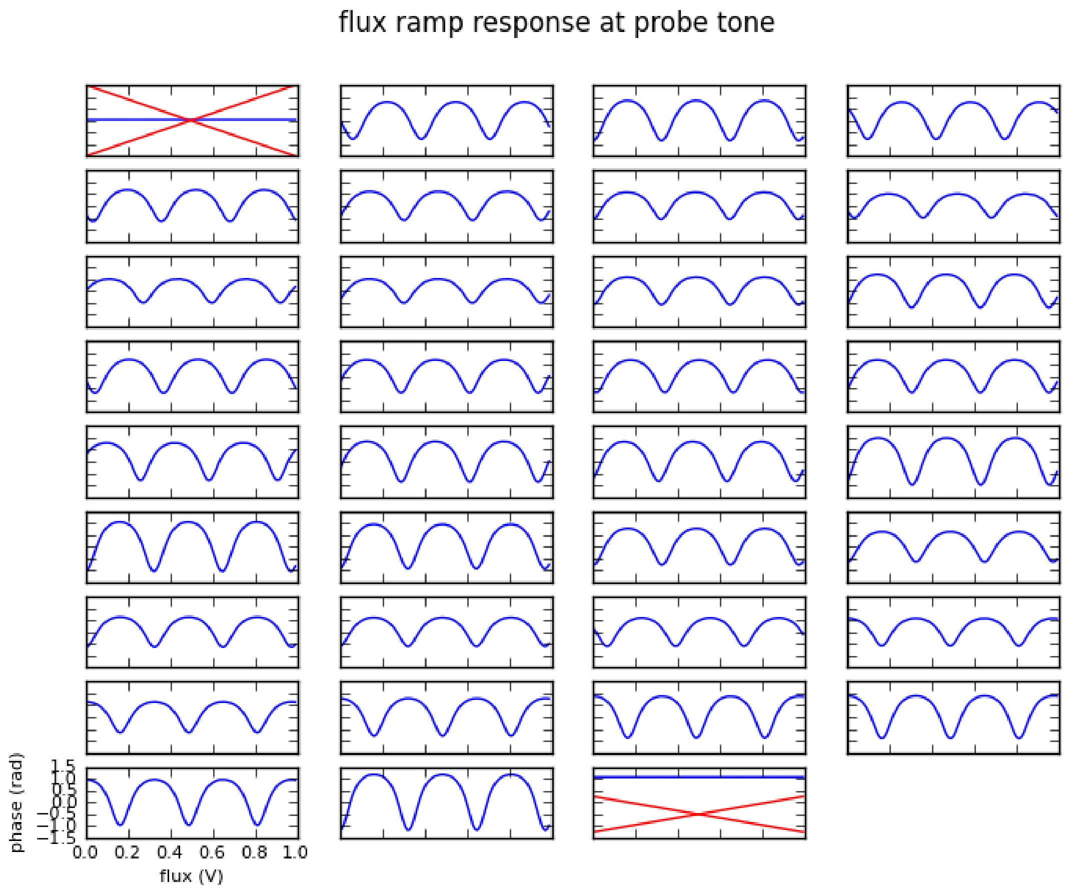

Figure 6: A tuning plot of the phase response to a slow flux ramp for each readout channel at optimal probe frequency selected by the tuning. The modulation depths of the response reflect the noise quality the user should expect for the readout channel. The periodicity of the sinusoidal SQUID response can be determined from this plot, which can be used to measure the mutual inductance of the flux ramp coil and SQUID. The relative phase shifts (along the x-axis) of the sinusoids are an indication of differences in the magnetic flux at each SQUID. The first and last channels with the red crosses have no response to the flux ramp.

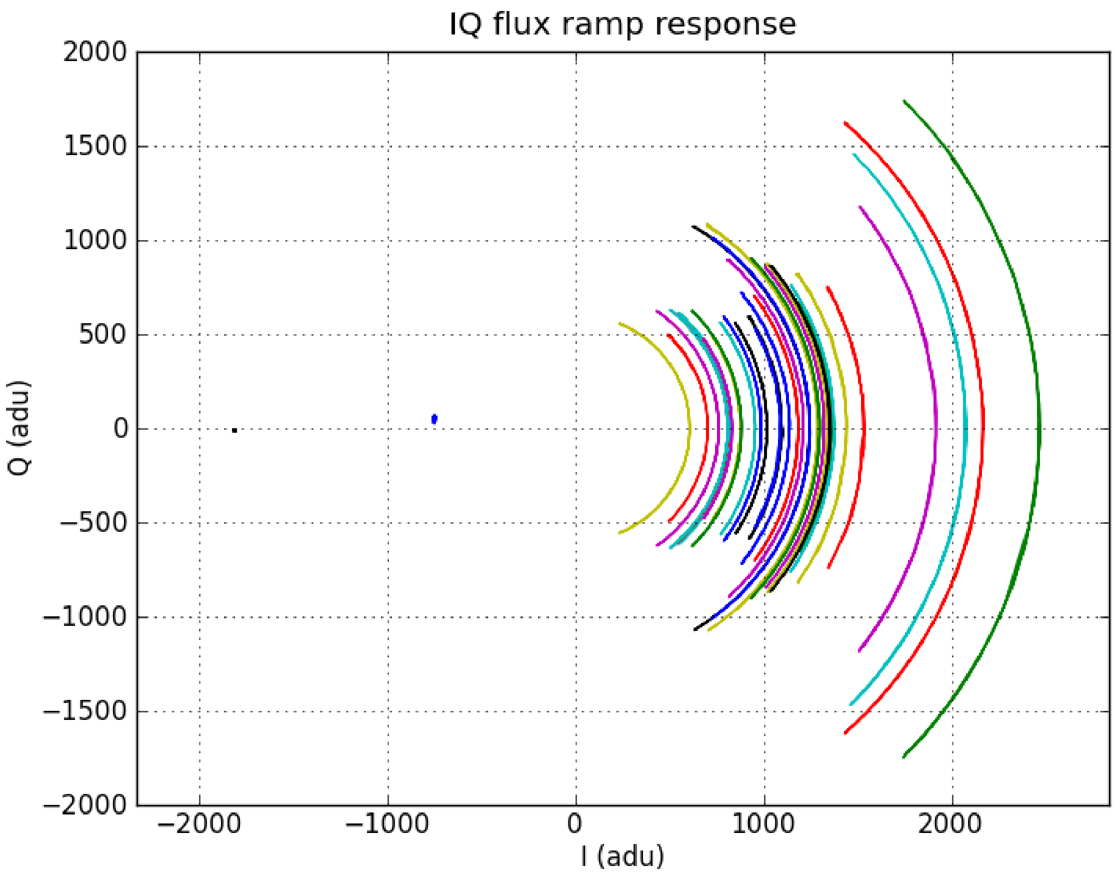

Figure 7: The final tuning plot is of the response in the IQ plane to a slow flux ramp for each readout channel at the probe frequency selected by the tuning. The tuning script centers the channel's flux ramp response in the IQ plane so that the firmware calculates the phase response is with respect to the center of a channel's flux ramp response. The script also selects the phase of the output probe tone so that the IQ response is centered on the x-axis. This prevents the response from wrapping across the $\pm\pi$ boundaries in the CORDIC ATAN operation that converts response in the IQ plane to phase in firmware. This plot and the plot of the modulation depth versus frequency in figure 5 are the two most important diagnostic plots produced by the tuning code and should be carefully monitored. Note the very circular appearance of flux ramp response, as opposed to the elliptical shapes of the IQ response to frequency sweep shown in figures 3 and 4. The first and last readout channels, which have no flux ramp response, appear as two small blobs in the IQ plane.

2. The LO is stepped by ± 1 MHz about its nominal value so that each tone makes a small sweep about its default value in order to find the rough resonance locations. This step creates the first two plots, 'mag(S21) vs Freq' and 'raw IQ loops', shown in figures 2 and 3.
3. The centers of the resonance circles in the IQ plane are determined from the LO sweep. The centers are then programmed into the firmware which subtracts them from each resonators' IQ response. Another LO sweep is made to produce the 'centered IQ loops' plot in figure 4 to verify the IQ data is being properly centered[1] in firmware.
4. The code now performs power sweeps in which a DC flux ramp input is stepped and a LO sweep of 1 MHz is performed at each step. For a given LO frequency step there are a family of points corresponding to the steps in the DC flux ramp input. This family of points sweeps out the $V - \Phi$ response of the SQUID, from which the modulation depth in radians is determined. Thus, there is a measurement of modulation depth made for a number of points ± 0.5 MHz about each of the default readout frequencies that results in the 'mod depth vs probe frequency' plot in figure 5. The code then fits a parabola the to modulation depth versus frequency response data to identify the peak response. The fit technique is used instead of simply finding the maximum in case the data are noisy or spiky. (Optionally, the code can loop over output attenuation in order to determine the optimal probe power. This is not currently done since we are power starved.)
5. The optimal probe frequency (and relative output attenuation) are saved to the 'tuned_freqs.txt' file and the output DAC waveform is programmed with these values.
6. A slow (1 Hz) flux ramp is turned on and the response of each channel in the IQ plane is recorded. The centers of the resulting IQ loops are fit, programmed in firmware for subtraction, and the output phase of the waveforms is corrected to rotate the IQ flux ramp response so that it is centered on the x-axis. This rotation step prevents the IQ response from wrapping across the $\pm\pi$ boundary in the firmware's CORDIC ATAN block that converts the IQ flux ramp response to phase response.
7. The IQ response to the slow flux ramp is recorded one last time after the centering and rotation step. The 'flux ramp response at probe tone' plot in figure 6 is made to show the periodic SQUID response and the 'IQ flux ramp response' plot in figure 7 is made to demonstrate the IQ response has been properly centered and rotated.
8. Finally, the tuning script saves all of the data recorded during the tuning to 'tune.mat' so that the tuning paramters may be monitored over time.

The tuning script can either be from a umuxlib wrapper at the ipython command line as:

```
um.tune()  #tune directory will be set to '/data/raw/todays_date'
#or
um.tune('/data/raw/specific_directory')
```

[1]Frequency sweeps give distorted IQ loops due to cable delay, so are only used to find the approximate centers. Flux ramp response at a fixed frequency gives circular loops and is ultimately used to program the IQ centers.

or directly from your code directory as:

```
python tune.py  #tune directory will be set to '/data/raw/todays_date'
#or
python tune.py /data/raw/specific_directory
```

The script takes approximately four minutes to tune up the system, when the relative probe powers are not optimized. It is a first pass script that could be improved to run more quickly and do fewer intermediate steps.

In general, a tuning should be performed after every fridge cycle in order to maintain optimal performance of the readout system. Resonant frequencies may shift slightly (or a lot!) due to trapped flux. Resonant frequencies also shift when the base temperature of the multiplexer chip changes.

NOTE: The ROACH can sometimes get in to a funk (see section 13), which can almost always be fixed by tuning.

8 loading a tuning

The ROACH can be reprogrammed with the results of a previous tuning, which is considerably faster than running the tuning script again. For example, it is necessary to retune after the vna function from section 1 is run. The retuning operation is performed by the function 'um.retune'. It loads in the 'tuned_freqs.txt' file from the specified tuning directory, and programs the output DAC waveform and LO. The IQ centers and phase data that are used to center and rotate the flux ramp response during tuning are not reused. The centers may shift over time and new centers and phase data can be quickly remeasured and applied.

To reload a tuning at the command line in ipython:

```
um.retune('/data/raw/desired_tuning_directory')
```

NOTE: Consider performing a new tuning (section 7) instead of retuning if the base temperature of the multiplexer chip has changed substantially or if the cryostat has been cycled. A retune is always required after using the VNA function.

9 demodulation

After tuning or retuning, the system still needs to be configured for online (in firmware) flux ramp demodulation. The user must select the frequency and amplitude of the flux ramp. The flux ramp frequency determines the sampling rate of the detectors, also known as the frame rate. The flux ramp amplitude determines the number of flux quanta the ramp sweeps through. This determines the fundamental harmonic of the sinusoidal SQUID response or the carrier frequency of the frequency modulation that occurs in the readout. The carrier frequency is the product of the flux ramp frequency and number of flux quanta that are swept through. For example, if the flux ramp frequency is set at 20 kHz and the amplitude is set to sweep through 5 Φ_0, the carrier frequency will be 100 kHz. The choice of carrier frequency is based on the bandwidth of the resonator and the spectrum of the noise in the frequency direction for the resonators.

In the homodyne readout for the μMUX system the frequency noise is dominated by two-level systems and improves with increasing frequency, eventually reaching the noise floor set by the noise temperature of the HEMT amplifier. Thus, in the homodyne system the carrier frequency should be made as high as the single-sideband bandwidth of the resonator allows. In the μMUX version 10b chips the single-sideband bandwidth is around 150 kHz, so the carrier frequency is typically set to 100 kHz. At this point, the noise floor in the SDR system is higher than the floor given by two-level system or the HEMT. Thus, as discussed in section 9.3, the carrier frequency is not currently set as close to the resonator bandwidth as possible.

The demodulation operation takes the raw phase time streams and determines the offset in the phase argument to the sinusoidal SQUID response. The raw phase time stream can be represented by i samples acquired at a rate of 1 MHz. The sinusoidal response of the SQUID produces a fundamental harmonic at carrier frequency $\omega_c = 2\pi N_{\Phi 0} f_{fr}$, where N_{Φ_0} is the number of flux quanta per ramp period and f_{fr} is the flux ramp rate. For simplicity, assume that the input flux signal to the μMUX channel is quasi-DC and contributes a static phase change φ_j to the argument of the sinusoid for every ramp period j. Thus, the raw phase response of the μMUX channel can be modeled as:

$$\phi_i = Asin(\omega_c t_i + \varphi_j) \tag{1}$$

The demodulation algorithm recovers the phase argument through the algorithm:

$$\varphi'_j = arctan(\frac{\sum_i \phi_i \cdot sin(\omega_c t_i)}{\sum_i \phi_i \cdot cos(\omega_c t_i)}) \tag{2}$$

It produces one measurement of the input phase, and therefore the input flux, per ramp period. Thus, while the raw phase time streams are produced at a rate of 1 MHz, the output demodulated time streams are produced at the flux ramp rate f_{fr}. The phase and input flux are related by:

$$\varphi = 2\pi \frac{\Phi_{in}}{\Phi_0} \tag{3}$$

NOTE: Both the SQUID response (ϕ) and the demodulated signal of interest (φ') are phase data. To avoid confusion the former will be referred to as 'raw phase time streams' and the latter as 'demodulated phase time streams' in this document.

9.1 flux ramp control

The flux ramp is controlled by a remote-controllable function generator as described in section 3. A resistor (typically 10 kΩ) is placed in series with the flux ramp lines. The value of the resistor sets the current swing at SQUIDs and the mutual inductance (M_{fr}) between the flux ramp inductor and SQUID sets the number of flux quanta that are swept for a given flux ramp amplitude. In the μMUX version 10b chips $M_{fr} = 88\ pH$.

The flux ramp function generator can be remotely controlled through the generic script setupFG.py in the code directory. This script sets the frequency, amplitude, DC offset, output waveform and output state of the flux ramp funtion generator. The type of function generator being used and the type of connection is configured by the user at the top of the script. The default function generator used at NIST is the SRS DS345 over a GPIB connection. If a different function generator is being used the syntax for controlling it should be added to the bottom of the script.

The script can be called by a wrapper function in umuxlib at the ipython command line as:

```
#e.g. um.setupFG(10000,1.64,0,ramp,on)
um.setupFG(frequency,amplitude,offset,waveform,output)
```

or directly from the code directory as:

```
#e.g. um.setupFG 10000 1.64 0 ramp on
python setupFG.py frequency amplitude offset waveform output
```

The input variable ‘waveform’ accepts the following arguments: ramp, sine, square, triangle, noise and arbitrary. The input variable ‘output’ accepts: on and off.

NOTE: In the SRS DS345 function generator the amplitude and offset voltages that appear on the output are double the commanded value.

9.2 raw phase time streams

The firmware takes the centered, rotated IQ data for each channel and creates phase time streams sampled at 1 MHz. These raw phase time streams have the sinusoidal response of the SQUIDs that is demodulated to produce a measurement of the input flux. The raw phase time streams can be read in with the functions ‘readRawPhase’ or ‘readRawPhaseLong’. The former loads in 2048 samples stored to a BRAM buffer and the latter loads 2**20 samples stored to a QDR buffer. They return phase timestreams for one channel at a time, as shown in figure 8.

The flux ramp signal must be turned on before these functions are called. Their acquisition is trigged by the flux ramp TTL signal from the ‘sync’ output of the flux ramp function generator. This is done to ensure that the raw phase timestreams are properly synchronized with the flux ramp signal so that the ramp reset transient can be properly blanked out. If the flux ramp is turned off the ‘getRawPhaseLong’ function will stall indefinitely while waiting for the data to fill the QDR. The flux ramp can be some slow frequency, but the QDR may take a long time to populate with samples.

The raw phase time streams can be read in at the command line of ipython with:

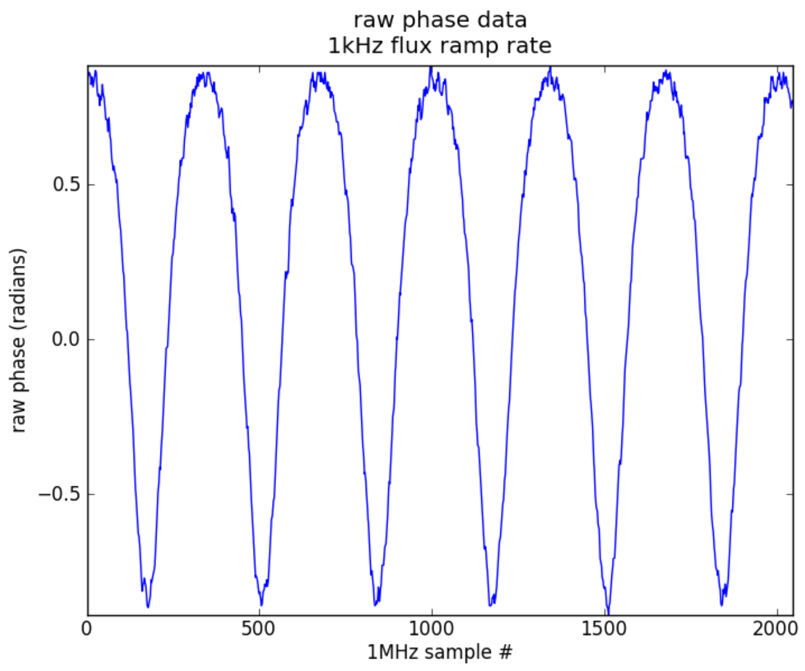

Figure 8: An example raw phase time stream for a single channel with a 1 kHz, 0.97 V_{pp} ramp applied to the flux ramp lines through a 10 $k\Omega$ series resistor. These data were read in using the 'getRawPhase' function. This function reads in 2048 samples written to a BRAM at 1 MHz.

```
ph=um.getRawPhase(channel_number)  #2048 samples for one channel from BRAM
#or
ph=um.getRawPhaseLong(channel_number)  #2**20 samples for one channel from QDR
```

9.3 SDR noise

9.4 firmware demodulation

The μMUX firmware produces 16-bit demodulated phase time streams for up to 256 readout channels at the flux ramp rate. There are currently two methods for transferring demodulated data from the ROACH to the DAQ computer. Up to 2**20 samples can be written to QDR memory ('demodqdr'), which can then be read in by the DAQ over katcp. Alternatively, a udp server can be established between the ROACH (power pc) and the DAQ computer, which can continuously stream data to the DAQ computer. The method for setting up and streaming data over the udp server is described in section 9.4.3.

The ROACH may be connected to a μMUX that has fewer than 256 channels, but the firmware automatically reports the demodulated output for all 256 channels. There are several reasons the user may want to restrict the data output to a limited selection of channels. For example, the

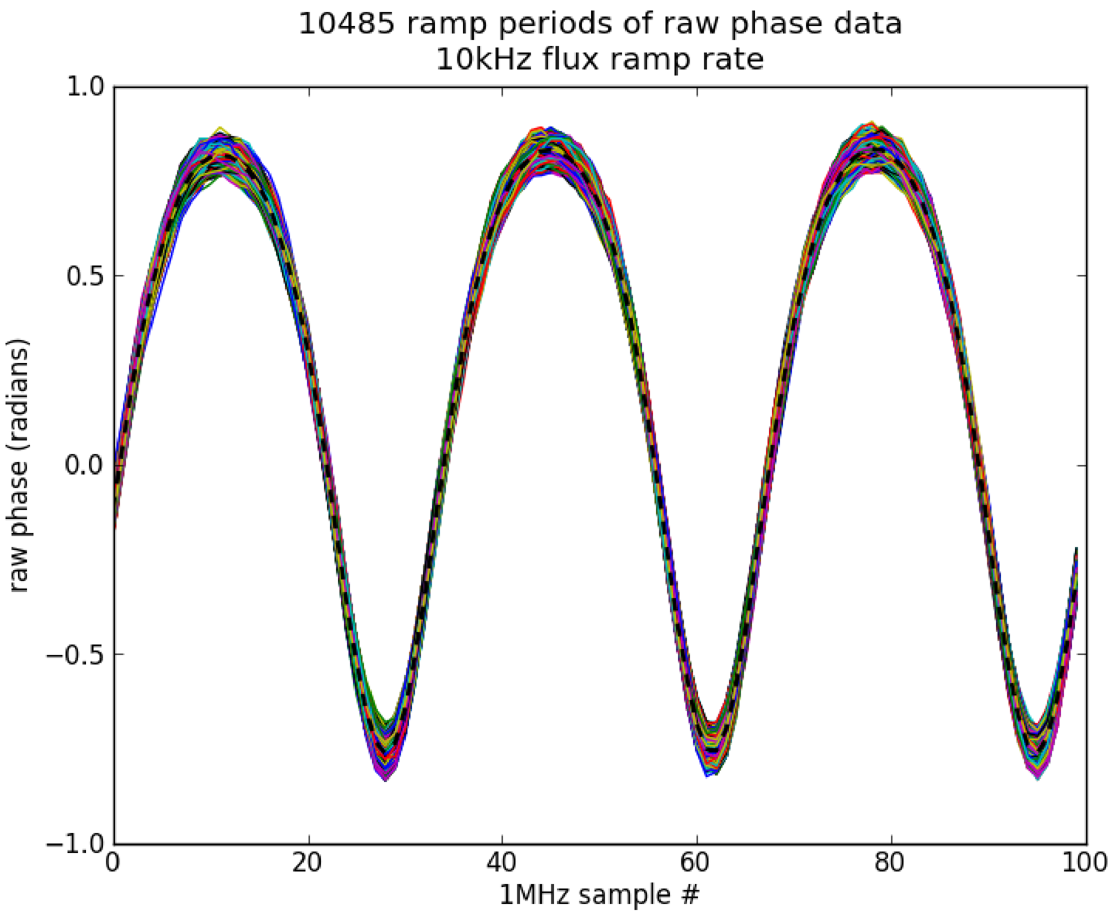

Figure 9: An example raw phase time stream for a single channel with a 10 kHz flux ramp. The 10485 full framp periods have been overplotted (multicolor lines) along with their average (dashed black line). This average is used to calculate the carrier frequency in the 'setupDemod' function.

μMUX 10b chips have only 33 channels that respond to flux ramp modulation. The additional 223 channels that are read out by default in firmware do not contain useful data. Therefore, there are options in the firmware to report data for only a selection of channels.

9.4.1 setting up demodulation

The umuxlib can be used to automatically set up the firmware for flux ramp demodulation using 'setupDemod'. However, it is useful to detail the steps that are taken in order to better understand the procedure. The steps taken in 'setupDemod' are:

1. The flux ramp function generator is turned on with a specific rate and amplitude. The rate sets the readout rate of the multiplexer and the combination or rate and amplitude sets the carrier frequency of the demodulation.

2. Raw phase time streams are then read in from the QDR to determine each channel's carrier frequency. The number of samples per frame is determined and the phase data is averaged over all the full frames, as shown in figure 9. The mean phase response per frame is then fit with a sinusoid to determine the carrier frequency. This procedure is performed for each operational channel. The average measured carrier frequency is reported to the user.

3. The array of carrier frequencies measured for each channel is then programmed into the firmware. The firmware includes a multiplexer that records each channels carrier frequency.

4. The default blank period is programmed in the 'blank_period' software register of the firmware, so that the first 'blank_period' samples per frame are excluded from the sum in equation 2.

5. A FIR filter is generated so that the higher-order frequency content in the products of sin or cos and ϕ do not contribute to the sums in equation 2. The FIR filter has 26 taps and is set to have a cutoff frequency 50% higher than the mean carrier frequency. The stop band of the FIR filter is made wide – from the cutoff frequency to the 500 kHz Nyquist frequency of the phase data – in order to minimize in-band ripple. The FIR filter is then loaded in to the demodulation block.

6. Finally, the channel selection for the demodulated data is programmed in firmware. The function 'readDemodQDR' relies on this step so that it knows how to properly unpack the data it reads.

The steps above can be automatically set up by executing the function 'setupDemod' at the command line in ipython, the user needs only to specify the rate and amplitude of the flux ramp signal. This step can take a while, even for a modest number of readout channels. Alternatively, the user can indicate an array of carrier frequencies that are programmed into the carrier frequency multiplexer in the firmware. This may be done in order to avoid the time-consuming step of measuring the carrier frequencies for each channel, or to keep the carrier frequencies fixed over time. It is critical that all input flux signals, such as the TES bias, be turned off while using this function (see note below).

To automatically set up demodulation at the command line in ipython run (with all input flux signals turned off!):

```
carrier_freq_array=um.setupDemod(flux_ramp_rate,flux_ramp_amplitude)
#or to manually specify the carrier frequency
um.setupDemod(flux_ramp_rate,flux_ramp_amplitude,carrier_freq_array)
```

NOTE: If firmware demodulation is automatically configured using 'setupDemod' it is critical that the input signals to the channels be turned off (e.g. function generators connected directly to the input coils or TES bias signals). These signals will cause the raw phase time streams to shift from frame to frame, which when averaged over all full frames will lead to an incorrect fitting of the carrier frequency.

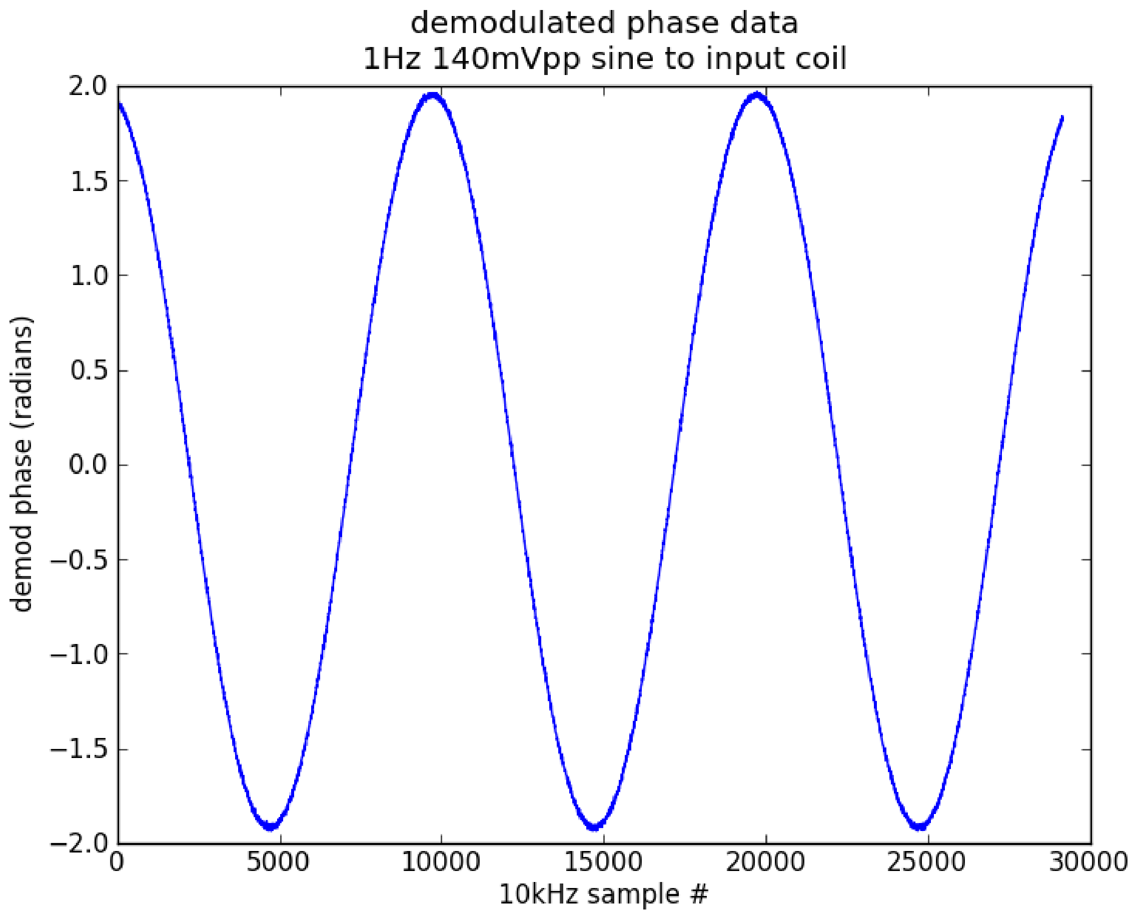

Figure 10: An example demodulated phase time stream for a single channel with a 1 Hz, 140 mV$_{\mathrm{pp}}$ sinusoid applied through a 10 kΩ resistor to the input coil of a single channel. These data were read in using the 'readDemodQDR' function. The 'NchRead' was set to 36 using the function 'setNchRead(36)', so that the channel received floor(2**20/36)=29127 of the 2**20 QDR samples.

9.4.2 acquiring demodulated data from QDR

A limited amount of data can be read in to the DAQ computer from the QDR memory 'demod_qdr', but this buffer is limited to 2**20 samples. The data written to the QDR can be restricted to a subset of channels in order to allow more samples per channel to be written to the limited QDR buffer. Due to the method of packing the frames, the channel selection must be set to a number divisible by 4 to prevent the samples from becoming out of sequence.

By default, the channel selection is set to the default readout channels by the function 'setupDemod'. The channel selection can be changed on-the-fly at any point after the demodulation has been started using the function 'setNchRead'. Currently the firmware is designed to read out the first 'NchRead' set by the function, but an upcoming firmware revision should allow for a specific range of channels to be set. The choice of 'NchRead' affects the number of samples that are read in for each channel by sampsPerChan = floor(2**20/NchRead). For longer time streams 'NchRead' should be set lower. If shorter acquisition times are desired 'NchRead' should be set to 256.

To change the number of channels read in to the QDR after the demodulation has been started, use the function 'setNchRead' at the ipython command line:

```
# read in the first NchRead channels
# NchRead must be 4–256 and divisible by 4
um.setNchRead(NchRead)
```

The demodulated data can be read in from the command line in ipython using the function 'readDemodQDR' (at this point it is safe to turn on input flux signals or the TES bias):

```
# read in 2**20 samples of demodulated data from QDR
# the data will have the shape (2**20/NchRead) x NchRead
demodData=um.readDemodQDR()
```

An example demodulated time stream is shown in figure 10 in which a 1 Hz, 140 mV peak-to-peak sinusoid has been fed directly to the input coil of a single μMUX channel. The demodulated data for that channel clearly show the sinusoid sampled at 10 kHz with a period of 10000 samples. The choice of amplitude has resulted in a demodulated phase amplitude of nearly 4 radians.

9.4.3 data transmission

More information here shortly. This is a new feature and it's going through its paces!

9.5 offline demodulation

An alternative way to acquire demodulated data is to demodulate offline. This technique can be very useful as a sanity check and for debugging the firmware-demodulated data. It is performed by reading in the raw phase time streams (see section 8) and demodulating them in python. The umuxlib code provides functions that allow the user to implement the demodulation algorithm given in equation 2. This procedure will yield a limited number of demodulated samples for only one channel at a time.

To perform offline demodulation the sine and cosine signals used in the sums in equation 2 must be generated. There are two ways to do this using the umuxlib functions. The waveforms can either be determined from a raw phase timestream acquired with no input flux signals, or if the carrier frequency is already known, they can be directly defined. In both cases the number of samples per frame must be determined from the flux ramp rate using the function 'getSampPerFrame'.

To measure the carrier frequency from a raw phase times stream and define the sine and cosine waveforms, input the following commands in ipython (with all input flux signals turned off!):

```
phDat=um.getRawPhaseLong('desired_channel')
sampPerFrame=um.getSampPerFrame('flux_ramp_rate')
LOsin,LOcos,carrierFreq=um.calcLO(phDat,sampPerFrame,'flux_ramp_rate')
```

or to define it explicitly with a specific carrier frequency (with all input flux signals turned off!):

```
sampPerFrame=um.getSampPerFrame('flux_ramp_rate')
LOsin,LOcos=um.defineLO('desired_carrier_frequency',sampPerFrame)
```

After defining the sine and cosine waveforms offline demodulation can be performed at the command line in ipython as (now with input flux signals turned on):

```
phDat=um.getRawPhaseLong('desired_channel')
demodDataOffline=um.demodTOD(phDat,LOsin,LOcos)
```

9.6 useful conversions

The demodulated phase time streams can be referred to current at the input coil knowing only the mutual inductance between the input coil and SQUID. This value will depend on the model of μMUX chip, and has been measured in the version 10b chips to be M_{in}= 88 pH.

The conversion from demodulated phase to flux and current at the input coil is derived from the equation 3:

$$\Phi_{in} = \frac{\varphi \Phi_0}{2\pi} \tag{4}$$

$$I_{in} = \frac{\varphi \Phi_0}{2\pi M_{in}} \tag{5}$$

since $\Phi_{in} = M_{in} I_{in}$ and where $\Phi_0 = 2.0678 \cdot 10^{-15}$ Wb is the magnetic flux quantum.

As an example we can verify the amplitude of the phase response we expect for the sinusoid applied direclty to the input coil of a channel in figure 10. The phase response is given by $\varphi = 2\pi \frac{M_{in} V_{in}}{\Phi_0 R_{in}}$. In this example $V_{in} = 140$ mV$_{\text{pp}}$ and $R_{in} = 10$ kΩ, which gives $\varphi = 3.74$ radians of peak-to-peak response, which is in agreement with the amplitude in the figure.

The conversions can be performed at the command line in ipython using:

```
inputFlux=um.ph2inputFlux(demodData)
inputCurr=um.ph2inputCurr(demodData)
```

NOTE: The M_{in} calibration value is hard-coded in the initialization of umuxlib.py and must be modified if a version of μMUX chips with different coupling than that of 10b is used.

10 diagnostics

11 function descriptions

12 software registers

13 troubleshooting