Instructions and operation manual for The PUG: Real-time dual comb spectroscopy (DCS) on a graphical processing unit (GPU)

Contents

General information

This instruction manual will guide you on the installation process and how to use The PUG for your real-time DCS averaging. It will also give you some of the limitations of the software and tips to use it properly. There are two different installation processes, one is for only using the compiled C++ code along with the python GUI and the other is for diving into the C++ and python code to see how it works. The latter will be explained at the end of the document. The information presented here is up to date as of May 29th 2024.

The code is available here: https://github.com/MathWalsh/The_PUG_DCS_on_GPU

Contributors:

Mathieu Walsh and Jérôme Genest

Disclaimer:

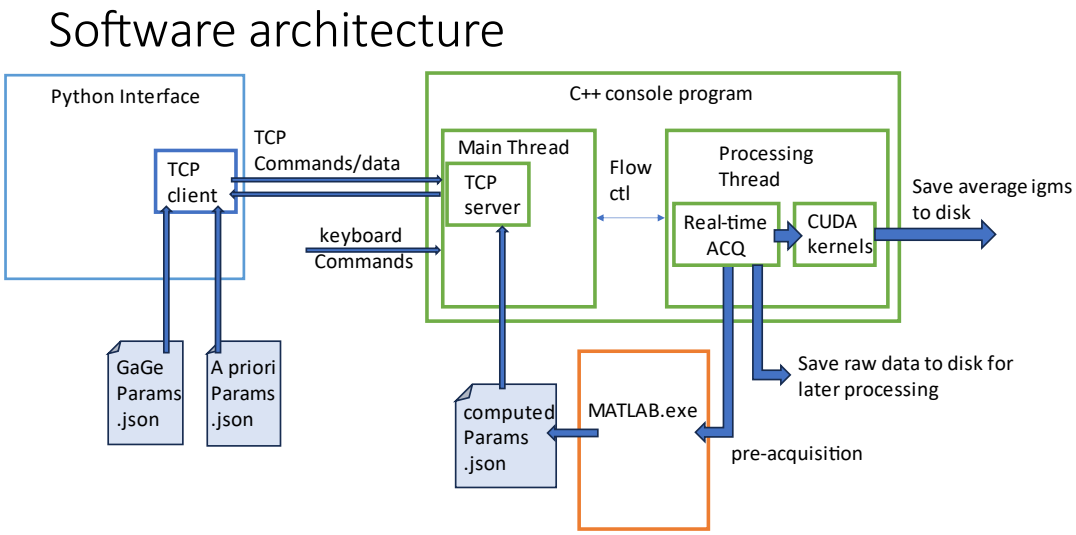
This is not a commercial product, there are still bugs and undetected issues in the code. To improve the quality of the code, please report any issues encountered via this github page:

If you need support to use the code, contact us via email, we will try to respond as fast as possible to get you going!

General flow of the software and GPU algorithm

Before you start using the PUG, we believe it is important for you to understand the basic flow of the software and the GPU algorithm. This information is also presented in this paper: XXX XXX. Below is a block diagram showing the different parts of the python and C++ program. The python GUI sends/receives all of its command to/from the C++ program through our custom TCP server. You will only interact with the python GUI during your real-time acquisitions, but some information such as the current data rate, measured delta fr (dfr), the current running time, etc. are displayed on the C++ program console during the acquisition.

All the parameters necessary to start a real-time acquisition are saved in two json files: "gageCard_params.json" and "apriori_params.json". These are the files modified by the user before starting the pre-acquisition step. These files are then used by the compiled matlab script to calculate all the corrections parameters necessary to perform the fast phase correction and resampling and the self-correction algorithm. These parameters are then saved in the "computed_params.json". The different .json files are accessible in different tabs of the GUI. They are also saved in the output folder along with the output data.



Before we dive into the algorithms, let's understand how the data gets to the GPU. Your digitizer is connected to your computed via a PCIe connection in the computer or in an external box (See hardware configuration). Each channel of the digitizer is sampling a different signal necessary for the corrections

performed on the interferograms. We will start with the most general correction possible that requires 4 optical references (one optical beat note with a CW for each comb and the CEO of each comb). With the interferogram channel, this means we need to digitize 5 different signals. With a 4 channel card, your best option is the following:

1. Interferogram stream
2. Time multiplexing of the two optical beat notes with a CW
3. CEO of comb1 (or beat note with another CW in the future)
4. CEO of comb2 (or beat note with another CW in the future)

There are two other configurations possible: one with only a CW reference and the other with no references. For the former, you will only need 3 channels on your card:

1. Interferogram stream
2. optical beat note of comb1 with CW
3. optical beat note of comb2 with CW

For the latter case, you only need 1 channel:

1. Interferogram stream

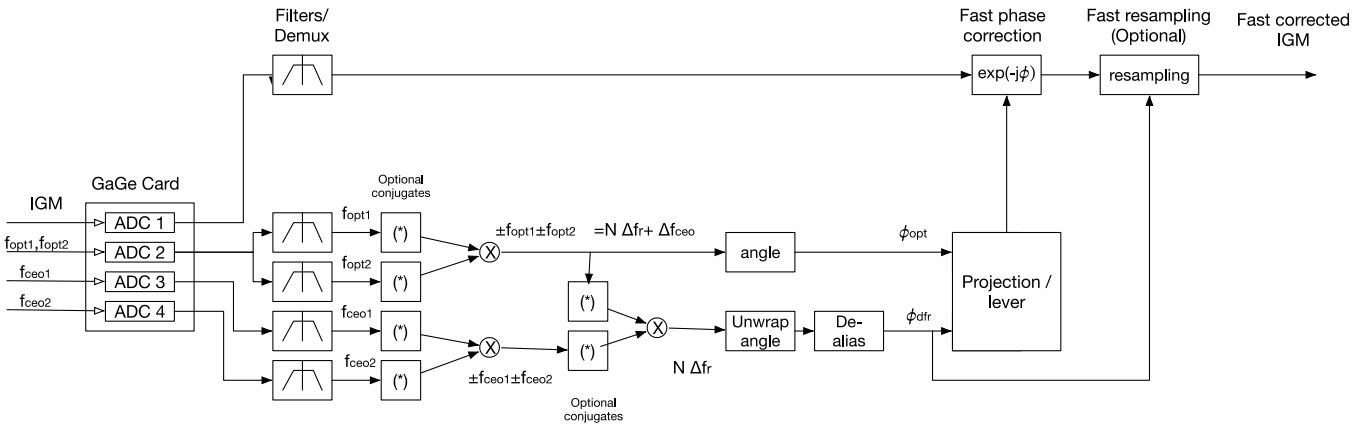
However, in the last two cases, it is important to understand what is the impact of not having two references (4 optical beat notes, 2 for each combs). In all cases, the self-correction will remove the slow out-of-loop noise from your interferograms with a dfr/2 bandwidth (BW). With 1 reference (i.e. two beat notes with a CW laser) you can correct fast phase noise but you may still have fast dfr noise remaining. With no reference, you have both fast phase noise and fast dfr noise remaining. Also, in theory, the self-correction will work if there is <pi rad of phase between two consecutive IGM center bursts. But on the GPU, we put a safety margin, so it will only work if there is <1.5 rad (This safety margin can be modified, contact us for more details). Moreover, depending on the quality and stability of your combs and your locking electronics, the fast noise can introduce a bias on what you are trying to measure. So, it is important to consider what is the best configuration to use given your DCS setup (This is also explained in the paper). If you can reach a fast dfr, the self-correction will correct noise up to dfr/2 BW so you may be able to avoid using two references. Similarly, if your locking allows you to have low phase and dfr noise at high frequencies, then you may be able to go without 2 references. We would recommend always using two references to get the best results. It is also important to note that the references must be locked to the same frequencies during the whole measurement. If the references change frequencies (> 0.5 – 1 MHz), they will go out of the band pass filter and the correction won't work.

The data of all the channels is sent to two RAM buffers on the processing computer. We are currently able to handle a data rate of 1.6 GB/s (200 MS/s per channel with 16 bit per sample) in real-time. The data is then transferred to the GPU to be processed. The data is processed in batches in the GPU to use the parallel capabilities of the GPU and for the self-correction.

For the GPU algorithm, there are two main steps: Fast corrections and slow corrections.

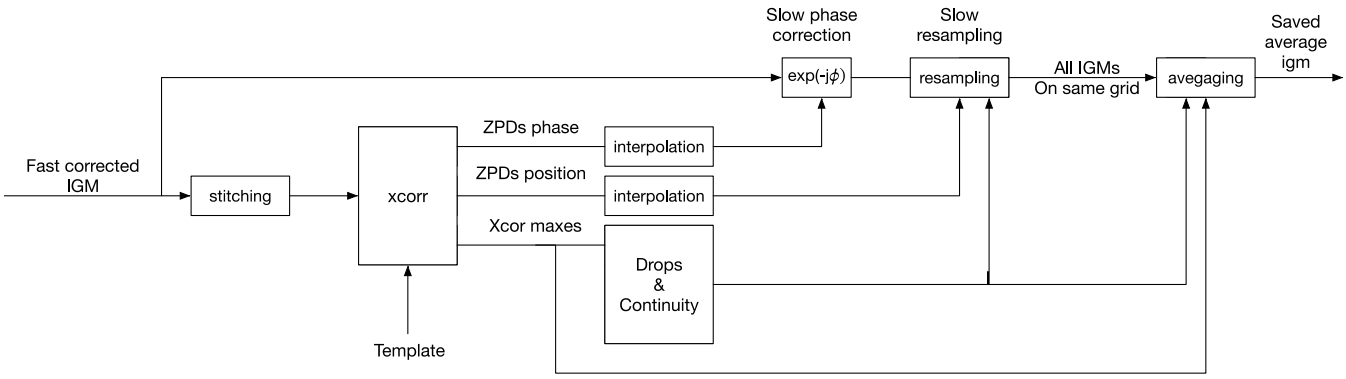
Below is the block diagram for the fast correction algorithm.

- Each signal is demultiplexed and filtered using a complex FIR bandpass filter.
- The optical beat notes (fopt1 and fopt2) are optionally conjugated and then multiplied together to produce the phase evolution of the teeth pair that beat with the CW.
- They are also combined with the two CEOs to produce the phase evolution of N*dfr (N is the number of teeth between the two optical references). These two signals can optionally be combined to do a projection of the phase at any wavelength before the fast phase correction.
- The dfr signal can also be used to do a linear resampling to have a constant dfr grid.



The IGMs coming out of this step should only have slow out-of-loop phase and dfr noise remaining. The second step, the slow corrections (Self-correction), is done to remove this slow remaining noise. Below is the block diagram of the algorithm.

- A cross-correlation (xcorr) against a template IGM calculated in the matlab script is done to retrieve the zero-path difference (ZPDs) phase, position and amplitude. This gives information about the noise at a dfr rate.
- A spline interpolation between the ZPDs phase and between the ZPDs position is performed before the slow phase correction and the slow linear resampling.
- After this step, all the interferograms are in phase and have exactly the same number of points, so a coherent averaging can be performed.
- The average IGMs are saved into binary files in the output folder (All the averaged IGMs will have the same number of points and the same phase, so no correction in post-processing is normally necessary).



Hardware configuration

The PUG requires: A computer, a PCIe fast digitizer and a NVIDIA GPU.

1. Computer
   - Windows operating system (version 10 or 11)
   - At least 32 GB of ram (the code does not use that much RAM, but we have seen issues when trying to allocate the RAM buffer for the digitizer (See TODO_KNOWNBUGS.txt file), so we recommend having more RAM than necessary).
   - At least 4 processing cores (CPU cores). More cores is better in this case. The python GUI has two asynchronous threads running continuously, same thing for the C++ application. This means that the operating system will have a limited number of cores to run other processes if you don't have enough cores.
   - The cards need to be linked to the computer using at least 2 PCIe 3.0 x4 conections. GaGe cards currently (as of 2024) support PCIe 3.0 x8 and your GPU might support PCIe x16 4.0 or even 5.0. Exploiting your card specification to their full extend is preferred. Increments in the PCIe standard (3->4->5) double the transfer speed at each step, so is doubling the number of lanes (x4 -> x8 -> x16). Our test configurations included having both card in external thunderbolt 3.0 eGPU enclosures, each providing PCIe 3.0 x4. The 1.6GB/s data rate is sustained in that case. Make sure, however that each thunderbold port on the computer is connected to DISTINCT sets of x4 PCIe lanes.
   - In theory, our TCP server allows the user to run the python GUI on a different computer than the processing computer running the GPU. This feature still needs to be fully tested.

2. PCIe fast digitizer
   - We are currently only supporting PCIe gages digitizer from GaGe / vitrek. The GaGe C++ code is well separated from the rest of the C++ code, so it will be possible to support other PCIe cards in the future.
   - The code has been successfully tested on a CSE161G4-LR which is a 4 channels, 1 GS/s, 16-bit, +-240 mV range card. A 4 channels 200 MS/s, 14 bit, variable range has also been successfully tested, but there is still some issue depending on the streaming buffer size. According to Gage,

there is an upper limit on the streaming buffer size depending on the firmware of the card. We are in contact with them to figure out this limit
- We currently support a data rate of 1.6 GB/s (200 MS/s on each channel) in real-time. More testing is needed to find what is limiting us (GaGe card can support ~5 GB/s in theory). If you need a faster data rate for your case, this could limit the performance of the DCS processing on the GPU (higher data rate = more calculations on the GPU for the same amount of time).
- To use the Gage card, it is necessary to install the proper drivers provided with the purchase of the card.

3. NVIDIA GPU
   - All the signal processing is done on the graphic card unit (GPU). We have successfully run the code on a GeForce RTX 4090 which is (as of 2024) a top-of-the-line gaming GPU. We have also run the code on a RTX 4070 Super which is ~4 cheaper than the 4090. We will also benefit from the improvement in the GPU market, so the cost of the GPU for this application will become smaller in the near future.
   - If there are multiple GPUs installed on the computer, the code will select the best performing GPU. If there is an issue with the selection of the GPU, please contact us via email. In general, we advise against having several NVIDIA GPU in the computer, we notably saw driver confusion issues with Quadro cards. For general purpose display needs, the built-in Intel GPU (such at Iris / UHD) should be sufficient.

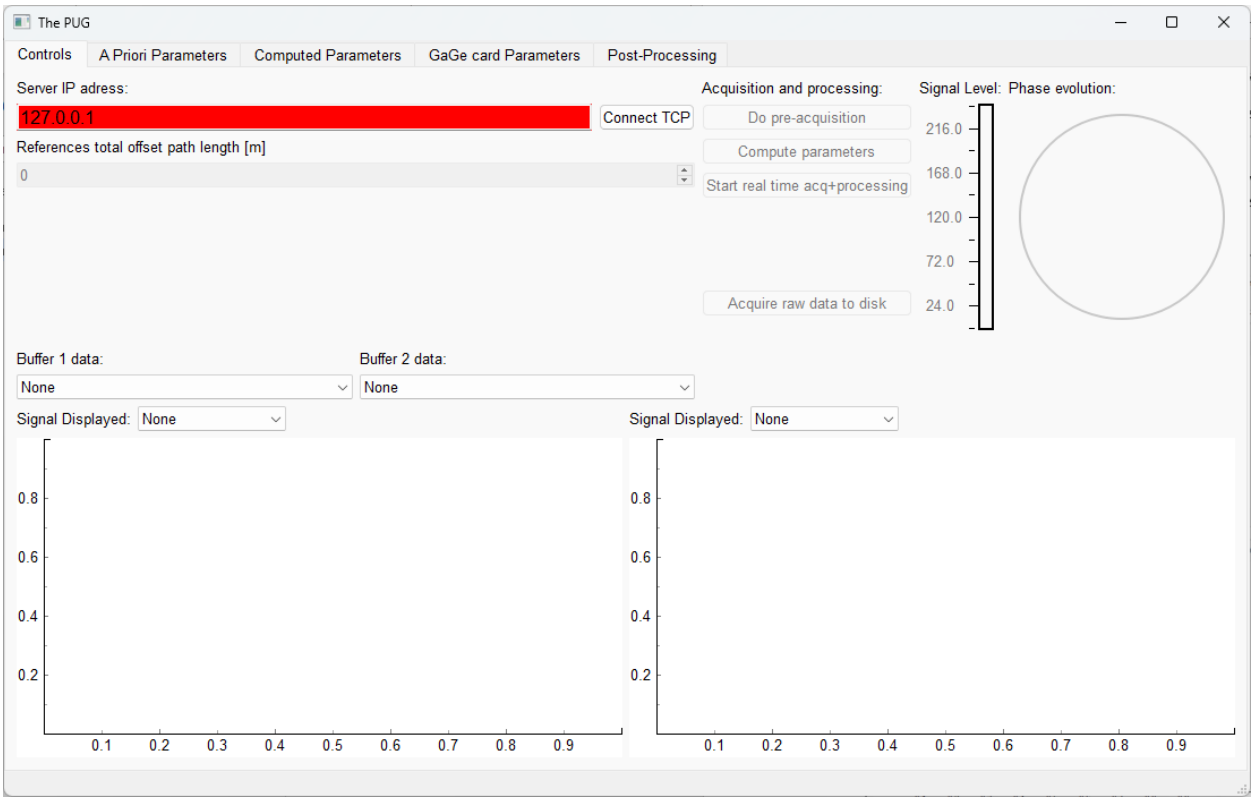
Installation instructions to use the compiled code

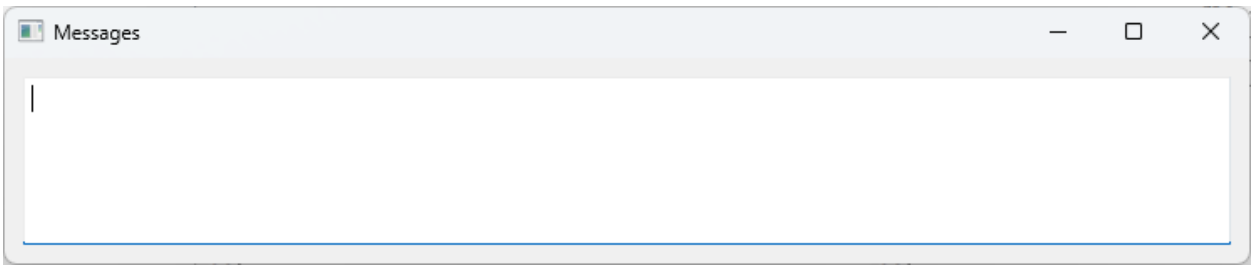
1. Install the gage drivers provided with your card.
2. Install visual studio before the CUDA toolkit to have access to all the CUDA features and to be able to run the C++ executable. In the visual studio installer, you will need to install the following packages : python development and desktop development with C++.
3. Download CUDA Toolkit 12.3 with your windows version (10 or 11) on this NVIDIA page.
4. Download a Python interpreter with the necessary libraries. You need at least python 3.10 to make the code work. **We highly recommend to use spyder in the latest WinPython (version 3.11 as of 2024) distribution because it comes with all the necessary libraries installed**.
   You also need to pip install this specific library: slack_sdk.
   Here is the list of all the necessary libraries to run the python code:
   - slack_sdk
   - collections
   - json
   - statistics
   - PyQt5
   - pyqtgraph
   - os
   - math
   - numpy

- datetime
- enum
- sys

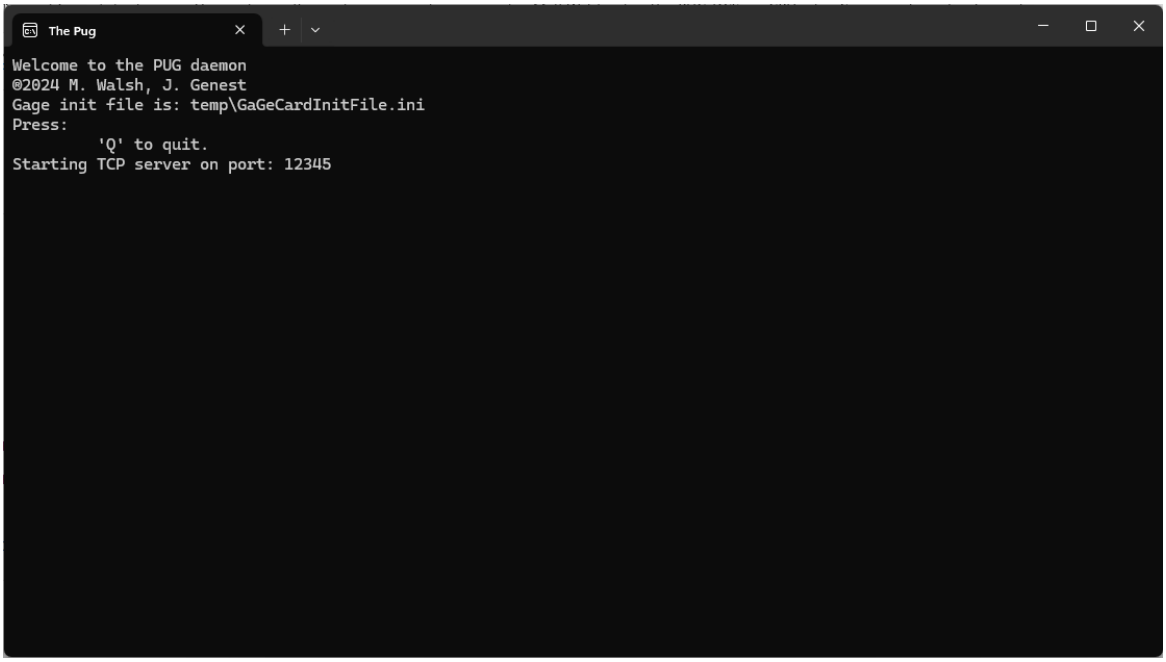
These are standard python libraries, so the code should work with most python interpreters.

5. Download the Matlab Runtime 2023b on windows. This is necessary to run the compiled matlab code “compute_DCS_params_GPU.exe” that computes the DCS parameters necessary for the real-time averaging.
6. Download the latest release or a specific version of the code on github.
7. Open your python interpreter and run “ThePUG_main.py” python file located in the “python_interface” folder. You should see two windows appear (see below). Each tab and buttons will be explained in the “How to use The PUG python GUI” section.

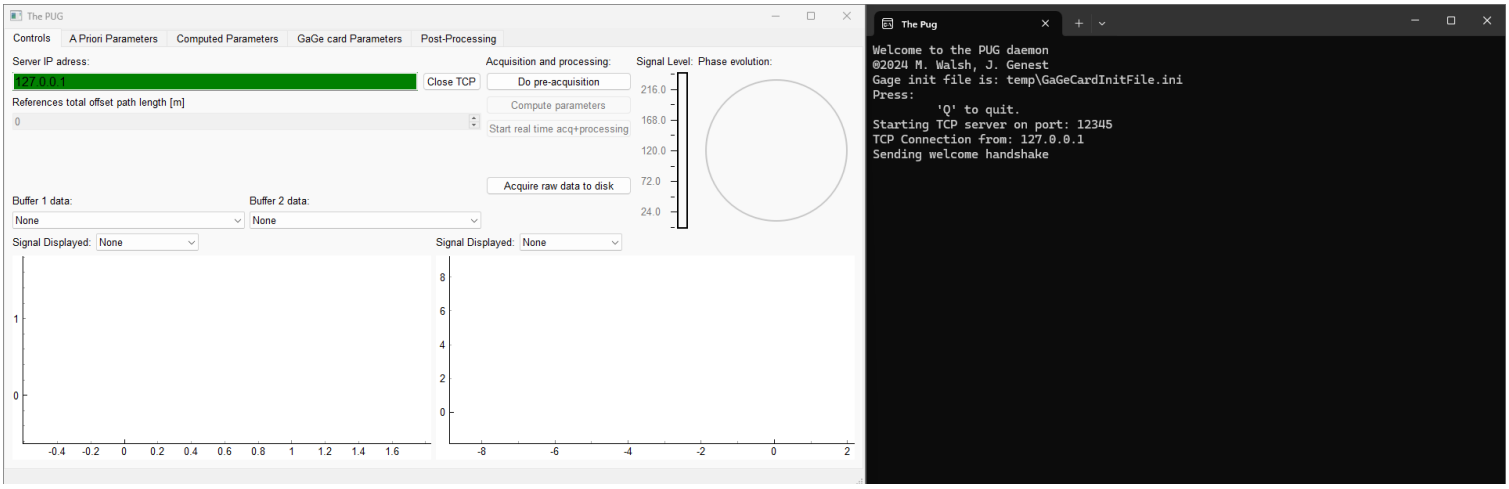




8. Click on the executable “DCS_GPU_Real_Time.exe” located in the “C_app_working_directory” folder. The window shown below should appear. In a future release, a .bat script will be available to launch the GUI and the executable.



9. Click on the “connect TCP” button in the controls tab on the main GUI. The button should become green and a “Sending welcome handshake” message with the IP address should appear in the C++ application window. The two applications are now connected together. In theory, this TCP server allows the user to run the python GUI on a different computer than the processing computer running the GPU. This feature still needs to be tested.



How to use The PUG python GUI

Once the python GUI and the C++ application are connected together through the TCP server, you can start sending commands to the C++ app with the different buttons in the python GUI.

Controls tab

There are four different modes that serve different purposes:

1. **Pre-acquisition (Requires the acquisition card)**
   - In this mode, you acquire a short batch of raw data with the acquisition card that will be used to find the DCS correction parameters. It is important to configure the Apriori and the Gage card Parameters before doing the pre-acquisition because these parameters will be used for the real-time processing. These parameters are explained in the "Parameters tab" section.

2. **Compute parameters (Requires the Matlab Runtime)**
   - This mode calls the matlab executable "compute_DCS_params_GPU.exe" to compute the different DCS correction parameters. The executable uses the raw data acquired in the pre-acquisition mode. The matlab script is available in the "Matlab_computeParamsScripts" folder.
   - This step is crucial for your real-time processing to work properly. In previous real-time versions of similar codes on FPGA, the user had to choose all the parameters manually for the correction to work. In The PUG, we tried to remove as much work as possible from the user with this script. However, this is where the code is most likely to fail. We have tried to make it as robust as possible but there are still some issues with it since it is trying to find the corrections parameters for all the cases possible and for any DCS system. Please report any issues here as soon as possible to make it even better! The script is available in the "Matlab_computeParamsScripts" folder if you want more details.

3. **Real-time acquisition and processing (Requires the acquisition card and the GPU)**
   - After computing the DCS parameters, you can start the real-time averaging with the "Start real time acq+processing" button. This sends the command for the acquisition card to start acquiring in continuous streaming mode and send the data to RAM buffers. The data is then transferred to the GPU to accomplish the real-time processing. The averaged data is then saved into files.
4. **Acquire raw data to disk (Requires the acquisition card)**
   - This mode allows you to save a long stream of continuous raw data in a file to be used in the post-processing feature of the software. The amount of data that can be saved is limited by the RAM available on the processing computer.

During real-time acquisition and processing and during post-processing, it is possible to visualize the data being processed by the GPU. The refresh rate of the TCP server is set to 50ms. The true data rate can vary depending on your buffer size and the speed of your CPU, so you might actually see only a subset of the data in the visualizers. Two different data buffers are available to display two different signals at the same time (Buffer 1 data and Buffer 2 data).

The signals available are:

1. interferogram_filtered: Interferograms after the complex bandpass filtering step
2. fopt1_filtered: Beat note between comb1 teeth and the CW laser after the complex bandpass filtering step. *
3. fopt2_filtered: Beat note between comb2 teeth and the CW laser after the complex bandpass filtering step. *
4. fopt3_filtered: CEO of the comb1 after the complex bandpass filtering step. *, **
5. fopt4_filtered: CEO of comb2 after the complex bandpass filtering step. *, **
6. interferogram_fast_corrected: Interferograms after the fast phase correction and fast resampling step.
7. interferogram_self_corrected: Interferograms after the self-correction step.
8. interferogram_averaged: Average interferogram of a single file. ***

*****comb1 and comb2 could be inverted depending on your hardware configuration of the beat notes.

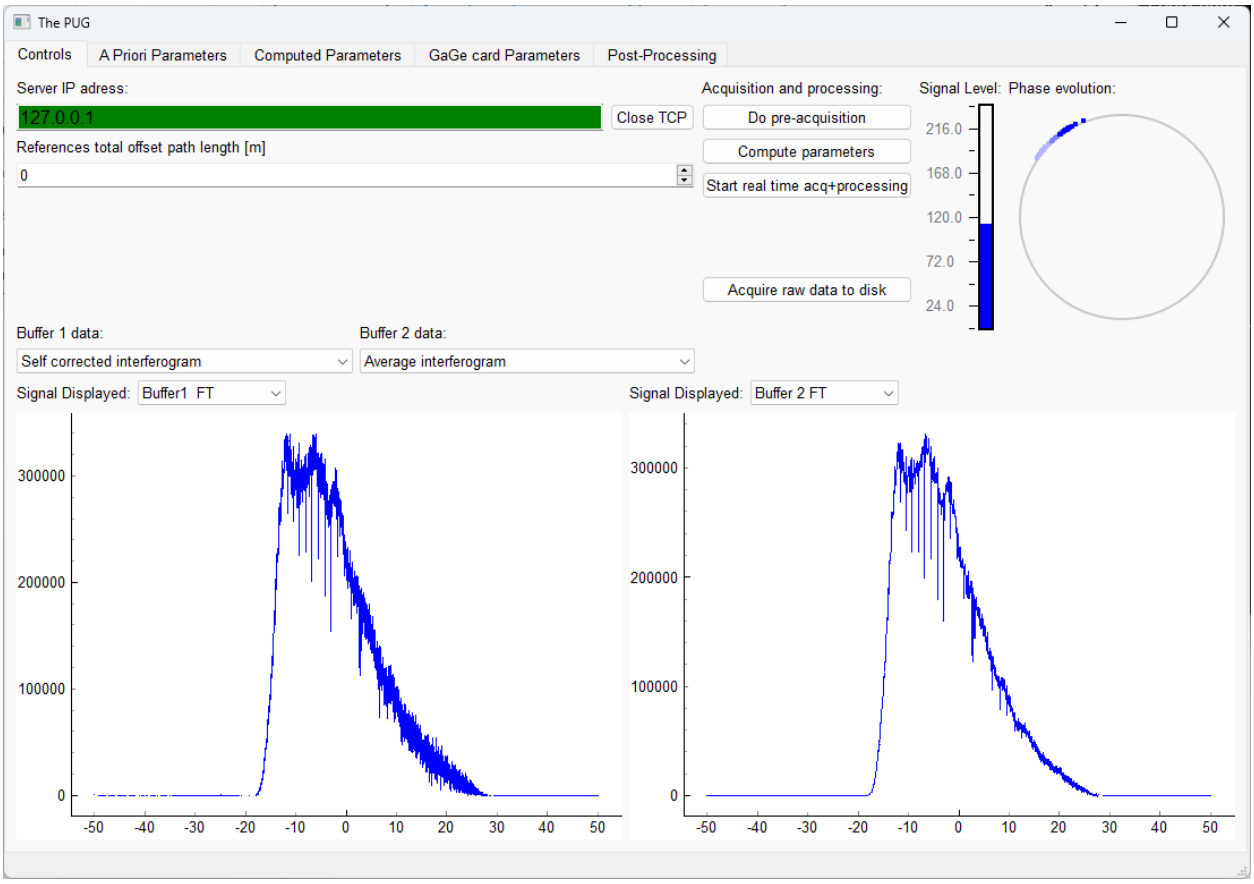
**2 CWs referencing is implemented but still needs to be tester thouroughly

*******The refresh rate of this signal will be slower because it is only refreshed when a file is saved.

Once you have chosen your two buffers, you can choose to display the time domain (real and imaginary part of the signal) or the fft (absolute value) for each signal with the signal displayed buttons.

With these buttons, you can also choose to display the results of the cross-correlations measured in the self-correction step of the algorithm (amplitude, phase and position). These values are good indicators to know if the processing is working properly.

The cross-correlations results are also displayed graphically in the top right corner (Signal level for amplitude and Phase evolution for phase). The signal level is displayed in mV and represents the average amplitude of the interferograms in each processed batch. The phase evolution shows the phase computed in the cross-correlation step between a template and each interferogram. The best-case scenario is a fixed point with minimal spread. Below is an example of good data with minimal spread. Previous buffers are presented with lighter shades of blue to show the time variations of the phase.



Parameters tab (Apriori, Computed and Gage card)

For every real-time acquisition you want to perform, you will need the Apriori and Gage card parameters json files to be filled properly before the pre-acquisition is done. It is possible to save your current configuration and re-use them for later acquisitions. When the GUI opens, it loads the previously saved config files in the “python_interface\parameters” folder. Properly configuring those files might take a bit longer for a new DCS system, but this makes it very convenient for subsequent acquisitions. A small description of each parameters possible values is available in the “Documentation” folder.

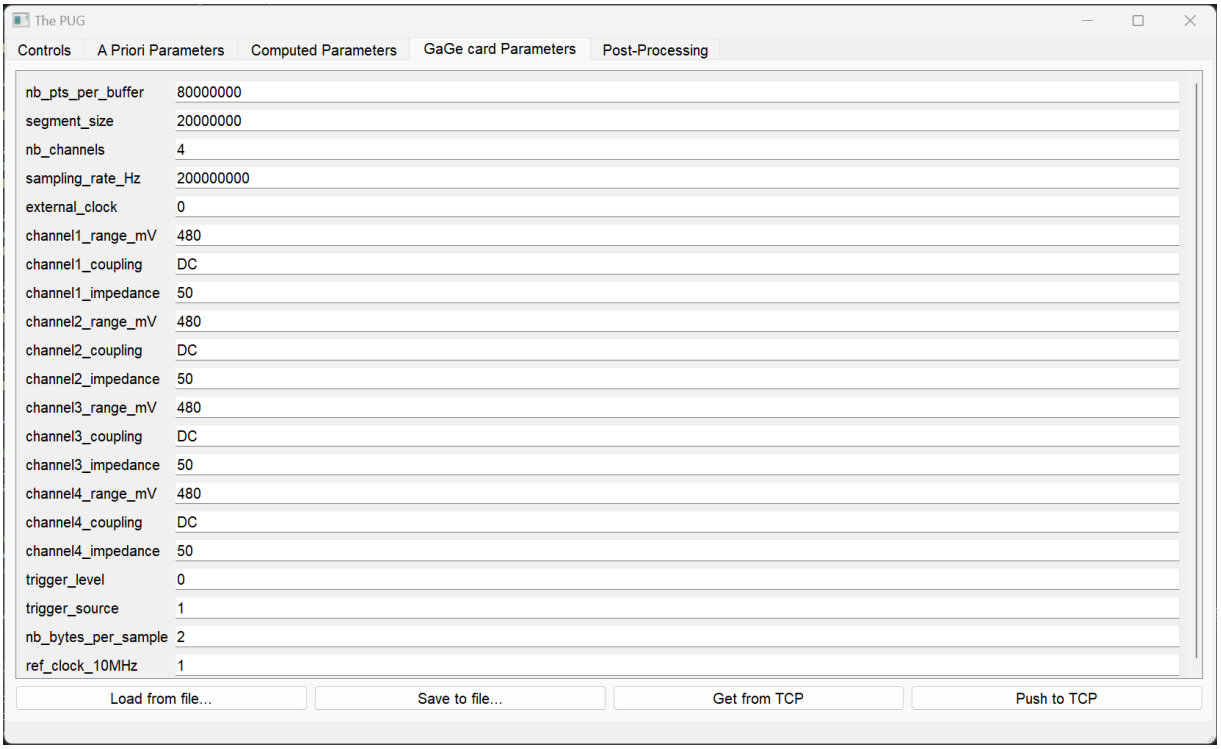
Here, we will present a more detailed description of each parameter that you will need to modify most often to make the real-time acquisition work properly.

The tabs allow loading, editing and saving either of the json files. While you can edit any parameter set at any moment, editing while an acquisition is ongoing is not supported and may lead to unpredictable results. Editing the GaGe and A priori parameters should be done before pre-acquisition. It is at this step that those parameters are sent to the C application. Editing the computed parameters is possible but not supported those are sent from the C application to the Python interface for information purposes only.

For Gage card parameters:

- nb_pts_per_buffer: The data is processed in batches in the GPU. nb_pts_per_buffer allows you to control the total number of points in each buffer. If you put 80e6 points with 4 channels, each channel will have 20e6 pts in each buffer. Let's say we have a sampling rate of 200e6 Hz and 4 channels, we will have 800e6 samples/s. So, depending on the value of nb_pts_per_buffer, you will have more or less buffers each second. We recommend a higher number (60e6-100e6) because this reduces the number of GPU kernels calls by the CPU and thus makes it less demanding for the CPU. We don't recommend going above 100e6 pts per buffer. Also, depending on your dfr value, there is a maximum value possible for nb_pts_per_buffer. Without going into details, the maximum number of IGM allowed in a single buffer is 1024. So, for high dfr cases, you may need to reduce nb_pts_per_buffer to have less than 1024 IGMs in each buffer. For low dfr cases, you need to make sure that you have >3 IGMs per buffer for the self-correction to work. So the code might not be suitable for dfr < 30 Hz. Contact us if you need to operate in this lower dfr range. Different cards can have different buffer size limitations, so this statement might be true only fer certain models. We are in contact with GaGe to figure out the limitations of each models.
- segment_size : This parameter controls the number of points acquired for the pre-acquisition and for the raw data acquisition. This is the number of points acquired PER channel. We usually put 20e6 when we do the pre-acquisition (80e6 points total). You may need to increase the number of points if you operate at a low dfr to have enough IGMs for the script to work (> 3 IGMs).
- nb_channels : number of active channels on your acquisition card. For gage card, it is multiples of 2 only (1,2,4,8).
- sampling_rate_Hz: Sampling rate of the acquisition. SoFor a 200 MHz sampling rate, with 4 channels, you will have 800 MS/s. With 2 bytes per sample, this produces a data rate of 1.6 GB/s (current limit).
- external_clock: Not implemented yet.
- channelX_range_mV: Total range of the channel in mV. Check your card for the available ranges.
- channelX_coupling: DC or AC (We recommend using DC).
- channelX_impedance: 50 or 1e6 (Normally use 50 Ohm).
- trigger_level: This is only relevant for the first buffer, we usually keep it at 0.
- trigger_source: We usually put 1, because it is the IGMs channel.
- nb_bytes_per_sample: For 14 and 16 bit card, this should be 2 bytes.

- ref_clock_10MHz : You can give a 10 MHz reference to the gage card. We recommend using this feature if you have a stable 10 MHz reference.



For Apriori parameters:

- absolute_path: The output data will be saved in this folder. We recommend keeping the default “C:\GPU_acquisition” path. For real-time acquisitions, the “Real_time_processing” folder will be created automatically and for each acquisition, a new folder with the starting time and date will be created. For post-processing, the “Post_processing” folder will be created automatically.
- save_data_to_file: Put 1 to save the average IGMs and the raw data for post-processing to the output folder. This parameter will be used more extensively in the future.
- nb_buffer_average: Each buffer processed by the GPU will produce 1 average IGM. You can choose the number of buffers you want to average before saving to file. Lets say you have 80e6 points per buffer with a sampling rate of 200e6 MS/s and 4 channels, this means you have 10 buffers per second. If you put nb_buffer_average = 10, you will have a file with an average IGM every second.
- fr_approx_Hz: If you have 1 comb going through your sample, this is the fr value of the comb. If both combs go through the sample, this is the average value of the two fr. It can be an approximate value, but more precise value will allow for more reliability at the compute parameters step.
- dfr_approx_Hz: For the matlab script to work, you need to give an approximate (+- 2-3% should work) value of the current dfr of you IGMs.
- minimum_signal_level_threshold_mV: If you have a lot of intensity variations on your interferograms, you can put a threshold of the low signals IGMs that you want to discard.

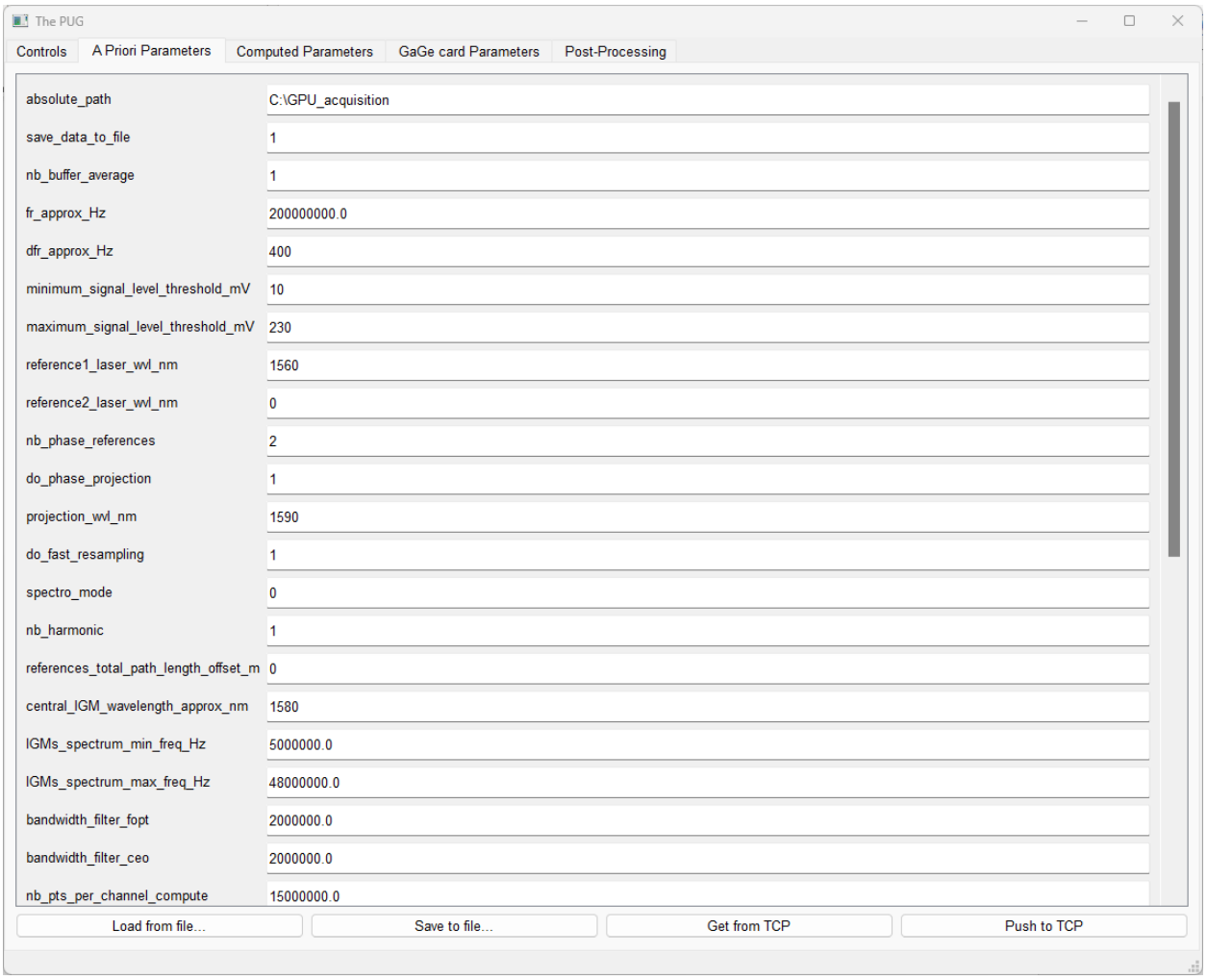
- maximum_signal_level_threshold_mV: If you have a lot of intensity variations, you can put a threshold of the high signals IGMs that you want to discard. This is useful if you know strong signals IGMs are non-linear.
- reference1_laser_wvl_nm: Wavelength of the reference laser used for the fast phase correction and fast resampling. It can be an approximate value, but more precise value will give better results.
  reference2_laser_wvl_nm: Wavelength of the second reference laser used fast resampling. It can be an approximate value, but more precise value will give better results. Put 0 if the second reference is CEO.
- nb_phase_references: number of phase references used for the correction. (0 for no references, 1 for two optical beat notes with a cw and 2 if you have the 2 CEOs.
- do_phase_projection: If you have 2 references, you can to a phase projection to a wavelength of your choice.
- projection_wvl_nm: Projection wavelength. We usually recommend doing the phase projection within your science light wavelength range. Also, the further from your reference you try to project, the worse the projection is going to be. Lets say you have a reference laser at 1565 nm and you want to measure methane around 1650 nm, you would put projection_wvl_nm = 1650 nm so that the phase correction will be optimal around this wavelength.
- do_fast_resampling: If you have two references, you choose if the algorithm does the fast resampling to remove fast dfr noise.
- spectro_mode: You can choose the type of correction you can apply. For now, three modes are implemented:
  1. 0 is for when your CW reference is near or in your IGM wavelength region.
  2. 1 is to correct your IGMs in the MIR after a DFG process with your references in a different region. Let's say you start at 1550 nm and do DFG to reach 3-5 or even 5-10 um. If you have 2 references, for example CEO and a CW around 1550nm, you will be able to recreate the DFG process in software to correct the IGMs in the MIR in real-time.
  3. 2 is to do higher harmonic corrections. The parameter nb_harmonic will be used to do the correction at a specific harmonic. This mode is very recent and has only been tested on one DCS setup
- nb_harmonic: Used to specify the harmonic that is being corrected in spectro_mode 2.
- references_total_path_length_offset_m: If you have a big delay between the time you pick of your references and your IGMs (for example, long outdoor measurements), you need to realign your references and your IGMs (This assumes an index of refraction of 1, so adjust the path length offset with the proper index). For now, this is a common offset for all the references, but in a future release, you will be able to adjust each offset independently.
- central_IGM_wavelength_approx_nm: This parameter is only used to know if the center of mass wavelength is at higher or lower wavelength than the reference1 laser. It does not need to be precise.
- IGMs_spectrum_min_freq_Hz and IGMs_spectrum_max_freq_Hz: These two values are the 3 dB cutoff of the bandpass filter applied to the IGM. Prior to your measurement, you need to know where you IGMs spectrum lies in electrical frequencies. To make sure you capture

the full IGM, you can put looser values for these two parameters. But don't make the filter capture too much out of band noise as this will reduce the performance of the self-correction.

- bandwidth_filter_fopt: This is the bandpass filter bandwidth that will be used to filter fopt1 and fopt2. Depending on the width of the beat note, you will need to adjust this value. Since the maximum number of coefficients available for filtering is 96, the minimal bandwidth is ~2 MHz. We recommend putting 2-5 MHz to capture all the phase noise of your beat notes.
- bandwidth_filter_ ceo: This is the bandpass filter bandwidth that will be used to filter CEO1 and CEO2. Depending on the width of the CEO beat notes, you will need to adjust this value. Since the maximum number of coefficients available for filtering is 96, the minimal bandwidth is ~2 MHz. We recommend putting 2-5 MHz to capture all the phase noise of your CEO beat notes.
- nb_pts_per_channel_compute: This is the number of points per channel that will be used to compute the correction parameters in the matlab script. This value needs to be smaller than segment_size in Gage card parameters. You may need to increase the number of points if you operate at a low dfr to have enough IGMs for the script to work (> 3 IGMs). The default value is 15e6·.
- nb_pts_post_processing: If you are using the post-processing feature of the software, this is the total number of points that will be processed.
- half_width_template: The matlab script needs to find a template to perform the cross-correlation on the GPU. You can specify a half-width value of the template with this parameter. We recommend that you put half_width_template = -1 to let the program choose the appropriate size. This part of the matlab script should work most of the times, but it can fail because it is coded to work for an arbitrary IGM shape which makes it less robust.
- signals_channel_index: This parameter specifies the channel index of each signal. The software expects the signals in the following order : IGMs, fopt1 (beat note between CW and comb1), fopt2 (beat note between CW and comb2), CEO1 and CEO2, For example, if you have 2 references (4 signals) and an IGM on 4 channels, you would put [1, 2, 2, 3, 4] where channel 1 has the IGM, channel 2 has fopt1 and fopt2, channel 3 has CEO1 and channel 4 has CEO2 (channels 3 and 4 could be inverted and the code would still work). If you are in dual-channel mode, you need to put your signals on channel 1 and 3 of your gage card. For single channel, only channel 1 is used.
- decimation_factor: The number of points chosen in the matlab script will always be even so we can always decimate the IGMs by a factor of 2 (Decimation is also why the IGMs spectrum needs to be centered around 0 Hz after the fast phase correction). If you're GPU is fast enough, we recommend not using the decimation feature.
- save_to_float: You can choose to save the data to disk in float32 or in int16. We recommend saving in float32 (save_to_float = 1).
- max_delay_xcorr: For the xcorr in the self-correction, you can specify the number of delays that will be calculated. This specifies how "far" from the expected center burst position the algorithm looks to find each interferogram with the template. Higher number of delays = more calculations on the GPU. If your combs are stable and you are using 2 references for the fast corrections, you can expect the IGMs to move <1 points. For a safety margin, you should put max_delay_xcorr = 10-15 in that case. If you are not doing any fast resampling (1 reference or no references case), you should increase that number (max_delay_xcorr = 30-40

or more depending on your comb stability). If you launch an acquisition and it does not seem to be updating the phase evolution circle properly or you are not getting close to 100% averaging, try increasing this value. Don't increase too much as this directly increases the number of calculations on the GPU. For a given GPU there is therefore a maximum value that you can have before you can't process in real-time.

- nb_coefficients_filters: Number of coefficients used for the FIR bandpass filters. You can choose 32,64 or 96 coefficients. If your GPU can support it, we recommend 96 coefficients for better results. If you have to use 32 coefficients, the filters will be less effective and more noise will be included in the phase references which will lead to worse corrections results.
- measurement_name: Entirely of your choice, to help you remember what this measurement was about. Not used by our code
- Slack_bot parameters: You can implement a slack bot to receive the messages displayed in the messages window
- real_time_display_refresh_rate_ms: You can choose the refresh rate of the different signals on the GUI. The minimum refresh rate is 50 ms. The true refresh rate will depend upon the nb_pts_per_buffer chosen in the Gage card params.
- console_status_update_refresh_rate_s: You can choose the refresh rate of the status update from the C app sent to the message box of the GUI. This is particularly useful if you have a slack bot and want to receive a message of the current status. The minimum refresh rate is 600s to avoid spamming.
- do_post_processing, date_path and input_data_file_name: These are internal parameter, you don't need to change them.

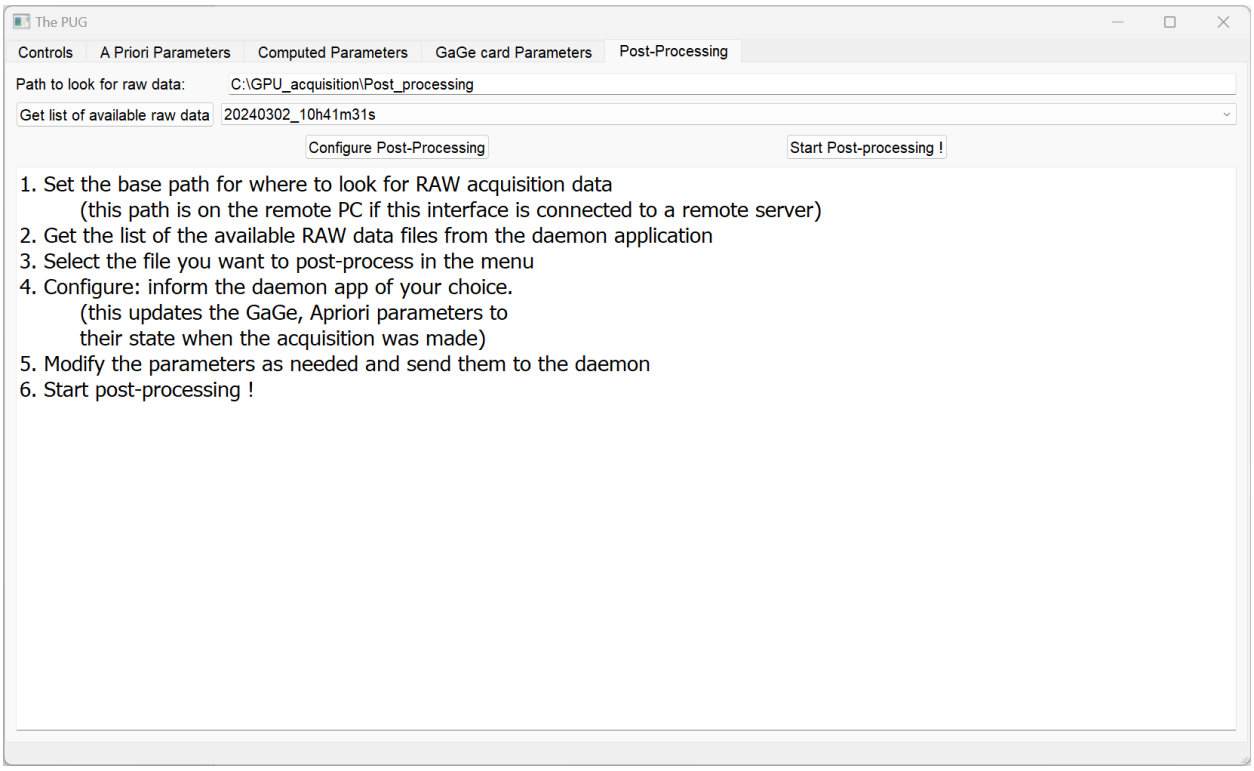


For Computed parameters:

These parameters are computed by the matlab script. After the compute parameters script is executed, the json file is sent to the python GUI and the parameters are shown in the computed parameters tab. You can look at them to see if the script found the proper parameters (You could compare them to a previous successful measurement with similar conditions). You can also use them to retrieve an absolute frequency axis in post-processing (It can be done but it is not trivial).

Post-processing tab

This feature was designed as a development tool for the GPU code, but you can use it to test different corrections configuration with raw data saved to file. To acquire the raw data, you use the "**Acquire raw data to disk"** mode in the "controls" tab. The data will be saved in the "absolute_path/Post_processing/insert_data_and_time' folder. Steps to perform GPU post processing are explained in the GUI. The output data will be saved in a "SimulationX" folder. Make sure to use the appropriate nb_pts_per_buffer , nb_buffer_average and nb_pts_post_processing for the amount of data you have saved.



Tips and tricks to make your experience easier

The previous information gives a good overview of every feature available in the software. Here, we want to give you a general idea of what your experiment flow should be like and how can you make sure the processing code is working properly.

Before making any measurement, here are some things you should think about:

- Are my combs locked properly or stable enough so that the electrical spectrum of my IGMs is fixed?
- Are my references all fixed so that they can be bandpass filtered properly?
- Do I have the proper referencing scheme? If your combs are noisy, you really need to use the two references scheme to have proper results. If you have quiet enough lasers, you

might get away with only 1 or no reference, but this is not recommended to get reliable results

- Am I sampling all the signals properly? We recommend using the GageScope software provided with your gage card to visualize each signal and make sure they are acquired properly.
- If you are multiplexing reference signals on a single channel, when must be at sufficient distant frequencies (10 to 15 MHz) for the filters to isolate them properly without cross talk. This imposes conditions on your locking configuration.
- Make sure you are filling Apriori and Gage parameters properly. A common error we have experienced is not putting the proper dfr or the proper correction configuration before starting the pre-acquisition. An incorrect dfr will lead to an error in the matlab script because it is not finding the IGMs properly.

After doing the pre-acquisition and the compute parameters successfully, how can you make sure the script found the proper parameters and it is working properly?

Here are some indications that it is working:

- The C++ application is showing the proper dfr value
- The BuffAvg value is closed to 100% (This means that 100% of the IGMs measured were averaged)
- The phase evolution has a narrow spread (This means that the phase correction is working well and the phase between consecutive IGMs is small). The phase might turn a lot between different buffers, this is an indication of out-of-loop slow phase noise.
- The fast corrected and the self-correction interferogram spectra are centered around 0 Hz.
- When you are displaying the interferograms signals, if you only see noise, this means that either the matlab script did not find the proper correction parameters, one of your signals is not stable (it is moving out of the filter) or you lost your IGM signal (too low intensity).
- The data rate should be constant around the expected value. For 200 MS/s, 4 channels, 2 bytes per sample, we expect a constant data rate of 1.6 GB/s.
- If the data rate is lower than the expected value, the data will pile up in the Gage card RAM and overflow. This can be caused in 3 ways:
  1. The data rate between the Gage card and the ram is too low.
  2. The data rate between the RAM and GPU is too low (less likely with this data rate)
  3. The GPU is not fast enough to process the data in real-time

  For case 1, make sure that you have a x8 PCIe lane for the gage card. Depending on the model, it should handle 1.6 GB/s without any issue.

  For case 2, this is very unlikely, NVIDIA GPUs should be able to handle >8 of GB/s of data rate

  For case 3, you need to buy a better GPU or reduce the number of calculations in the GPU. This can be achieved by reducing the max_delay_xcorr or nb_coefficients_filters parameters to reduce the number of calculations. Also, if you are doing the phase projection around your region of interest, you could potentially get away by not doing the fast resampling.

Same thing if your reference is in the middle of your ROI. This advice is only good if your optical bandwidth is not large, then you want to do the fast resampling.

The code will run until you stop the acquisition. All the IGMs saved in the output folder are in binary format. It is a complex IGM, so the real and imaginary part are saved consecutively. In the "Matlab_computeParamsScripts" folder, you have a small matlab script "show_DCS_results.m" that opens and shows the resulting IGMs. You could do a similar script in python very easily.

Installation instructions to use and compile the C++ code

1. Download and install a version of visual studio. We have used visual studio 2022 to make this software. In the visual studio installer, you will need to install the following packages : python development and desktop development with C++.
2. Download and install CUDA Toolkit 12.3 for your windows version (10 or 11) on this NVIDIA page.
3. Download and install the Gage drivers provided with the card
4. Link the github project https://github.com/MathWalsh/The_PUG_DCS_on_GPU.git to your visual studio to pull the code (you won't be able to commit to our repository).

This should allow you to look at and run the code behind the C++ application.