CUE Python Tool-kit: User Manual

5th February, 2019

1 Introduction

This document aims to provide new users with the insight and understanding of how to use all software developed in association with the PhD Thesis titled: "Exploring the Application of Ultrasonic Phased Arrays for Industrial Process Analysis", by Marcus Ingram, March 2019.

By the end of this document you should be able to install all the necessary software and dependencies associated with these. You should also be able to upload and save data in the format used throughout the Thesis. A high-level overview of each of the functions contained within each software tool is also provided, however the low-level understanding will only be possible through closer inspection of the raw code.

Any queries should be forwarded to Marcus Ingram PhD, University of Strathclyde, Glasgow using m.ingram@strath.ac.uk or Prof Anthony Gachagan, Director of the Centre for Ultrasonic Engineering using a.gachagan@strath.ac.uk. This document begins with an outline of how to install the relevant tool-kits onto a Windows PC. The remainder of the document should be interpreted in chronological order in terms of the image processing algorithm when applied to the PhD Thesis. Such that imaging takes place after data acquisition and image processing takes place after imaging etc.

Enjoy.

1.1 Abbreviations

The following abbreviations are used throughout this document:

- Full Matrix Capture (FMC)
- Full Raw Data (FRD)
- Total Focussing Method (TFM)
- Compute Unified Device Architecture (CUDA)
- General Purpose - Graphics Processing Unit (GP-PGU)

- Diagnostic Sonar Limited (DSL)
- Principal Component Analysis (PCA)
- Sign Coherence Factor (SCF)

2 Installation

The first thing you will need is access to the Python Programming Language. It is recommended to use Anaconda for Windows.

In order to perform the Total Focusing Method imaging, a GP-GPU that can support CUDA must be installed on the PC. And to compile the pyCuda software requires a C compiler such as Visual Studio to be installed.

The key dependencies of the code are:
- pycuda, numpy, Pickle, scipy, matplotlib and skimage

Once these are installed or forked from GitHub, these packages along with the files, found here:

- **Acquisition.py**
- **Imaging.py**
- **TFMKernel.cu**
- **BubbleSizing.py**
- **BubbleTacking.py**

should be stored in a local Python directory such as:

```
C:\ProgramData\Anaconda3\Lib\site-packages\
```

Once **Acquisition.py** is installed, the path to the dynamic link library, **DSLFITstreamFRD.dll**, must be specified, line 114 in **Acquisition.py**:

```
self.DSL = windll.LoadLibrary('C:\\Program Files\\DSLFITscan\\
          DSLFITstreamFRD.dll')
```

Once this is completed the relevant files can be imported into python using:

```
import Acquisition, Imaging, BubbleSizing, BubbleTracking
```

3 Data I/O

All data are saved in binary *.bin* format. The binarisation process is performed using Pickle Serialisation.

To upload data to the kernel we use:

```
FRDs = pickle.load(open('path\\to\\folder\\filename.bin','rb'))
```

where *FRDs* is a list of instances of the *FRD* class, which will be introduced in the next section and 'rb' corresponds to 'read binary'.

To save data from the kernel we use:

```
pickle.dump(FRDs,open('path\\to\\folder\\filename.bin','wb'))
```

where 'wb' corresponds to 'write binary'.

4 Acquisition

4.1 Connecting to the instrument

To connect to the DSL FIToolbox instrument we first create an instance of the DSL class contained within **Acquisition.py**. This launches the 'DSLFITStreamFRD' application developed by Diagnostic Sonar Ltd (DSL). Where upon the user can select a configuration file to load or can manually configure the settings of the FIToolbox. If the user has specified the path and configuration file, as shown below, this specific FIToolbox configuration will be automatically loaded.

```
FIToolbox = Acquisition.DSL(ConfigFile='path\\to\\configFolder
            \\filename.cfg')
```

Once the 'DSLFITStreamFRD' application has been configured we need to initialise the instance of the FIToolbox:

```
FIToolbox.initialise()
```

This process extracts the necessary information from the FIToolbox to build a look-up-table of the sample numbers in the data stream corresponding to the first index of each A-Scan.

4.2 Acquiring ultrasonic data via Dynamic Link Library

The data acquisition can be performed using two different functions:

1. Acquiring a single FRD dataset.

```
FRD = FIToolbox.acquire_single_FRD()
```

2. The second acquires an arbitrary number of *FRDs* set by the value of *'nFRDs'*.

```
nFRDs = 100
FRDs = FIToolbox.acquire_multiple_FRDs(nFRDs)
```

4.3 Translation of data stream into a meaningful dataset

Both of the above approaches acquire data via a Dynamic Link Library that outputs a copy of the data stream, which performed using the function:

```
def _get_u64_stream(self)
```

This function is called for every new FRD dataset to be acquired. Within this function a new instance of the FRD class, also contained within **Acquisition.py**, is generated.

```
newFRD = FRD(self.Fs, self.Ts, self.timeStamp)
```

This requires the sampling frequency, *Fs*, the start time of the A-Scan time axis, *Ts*, and the *timestamp* of when the dataset was acquired as input. These values are stored as attributes of the *FRD* instance. Following this, the data stream is uploaded to the instance of the *FRD* class using:

```
newFRD.upload_stream(I16Stream, self.FRD_LUT, self.SampleStep,
                            self.params['n_samples'])
```

The "upload_stream" function is contained within the *FRD* class rather than the *DSL* class. This reorders the data stream output from the FIToolbox into a coherent dataset with rows corresponding to A-Scans indices and columns corresponding to time samples.

5 Imaging

5.1 Setting the image parameters

The imaging algorithm has been designed for linear phased arrays that have a precise element pitch. The value of this pitch must be declared as it is used to construct the landscape of the image scene. It should be noted that the image construction time is purely dependent on the number of pixels in the image, not the dimensions of the image.

```
pitch = 0.7e-3    # distance between two elements (m)
velocity1 = 5740 # speed of sound in first medium (m/s)
velocity2 = 1496 # speed of sound in second medium (m/s)
zBegin = -5.0e-3 # metres from the front face of the array
nPixels = 1024    # number of pixels in y and z dimensions
```

5.2 Initiation of the imaging module

An instance of the *pyTFM* imaging class is created by uploading a single instance of the *FRD* class and the element pitch.

```
TFM = Imaging.PyTFM(FRD,pitch)
```

The initiation process builds an Numpy array containing the phased array element locations and extracts attributes from the *FRD* instance required to construct the image such as the sampling frequency, *Fs* and the start time, *Ts*. The second stage is to declare the velocities in the image media, **at least one** velocity must be provided. If the sound has propagated through two media then you must declare a second velocity.

```
TFM.setVelocity(velocity1=velocity1,velocity2=velocity2)
```

Next, the number of pixels in each dimension and length of each dimension are set. The images are created using square pixels, enabling accurate quantitative information to be extracted from the processed images. To achieve this, the number of pixels in each dimension is kept constant and the y-extent is restricted to the dimension of the array only and the z-extent is restricted to be the same length as the y-dimension, however the start-point of the z-dimension can be arbitrarily chosen.

```
TFM.setImage(nPixels=nPixels,zBegin=zBegin)
```

Alternatively, if this image geometry is not desired, the user can create a custom image geometry by specifying:

```
TFM.setImage(y0=y0,ny=ny,y1=y1,z0=z0,nz=nz,z1=z1)
```

where y0, y1, z0, z1 relate to the start and end spatial position of each axis (m) and ny & nz relate to the number of pixels on each axis.

5.2.1 Non-invasive Imaging

If the user wishes to image through a flat refracting interface, this feature must be set to "ON". This has the impact of raising the height of the array by an arbitrary distance value, which has been set to the beginning of the z-dimension here. Following this, the coefficients for the CoefficientsTFM modules are calculated, which is required to image into a second medium.

```
distance = zBegin
TFM.setRefractionON(distance)
```

Subspace Analysis and Projection can be performed prior to image construction to remove reverberation artefacts. This performs PCA on the FMC data set and projects the first *nTx* eigenvectors onto the original FMC dataset. This process is somewhat slow on a CPU so if you have a GPU available, set this to **True**.

```
TFM.doSubspaceAnalysis(useGPU=True)
```

5.3 Generate the image matrix

The actual imaging process is performed using the CUDA TFM kernel. There are four different TFM algorithms presented here, each of which can be performed with or without the CoefficentsTFM module.

1. Regular TFM imaging
2. SCF TFM imaging
3. Time Offset TFM imaging
4. Combined SCF and Time Offset TFM imaging

To use either the CoefficientsTFM or SCF imaging, their boolean operators must be set to **True**. However, to apply a time offset to the individual TOF values, an array with a shape of *nTx* rows by *nRx* columns must be provided, where each element in the array pertains to a TOF delay (s). If "txrx_delays" is not provided, the algorithm does not apply any additional time delays.

```
TFM.doTFM(coefficients_TFM=True,SCF=True,txrx_delays=timeOffset)
```

6 Object Sizing

The *BubbleSizing* class relates to an individual frame whereas the *BubbleObject* class relates to each individual bubble object in the current image frame. The *BubbleSizing* class requires an instance of the *pyTFM* class described above to be supplied to it along with the *scf_power*, which is typically set to one. The *lowerThreshold* and *upperThreshold* values (m) relate to the sizing thresholds used during the compilation of the valid *BubbleObjects* in the image frame. The *BubbleSizing* class will mark the frame as invalid if no valid objects are contained within the image or if the image is too noisy. There are a number of assumptions made about the image construction process:

1. The SCF matrix has been determined during image construction
2. The pixels are square
3. The images are orientated with the phased array on the vertical axis and the depth into the image is on the horizontal axis.

The bubble sizing algorithm is then deployed using:

```
processedFrame = BubbleSizing.BubbleSizing(pyTFM,scfPower,
                  lowerThreshold,upperThreshold)
```

7 Object Tracking

7.1 Setting the tracking parameters

The *BubblerTracker* class contained within **BubbleTracking.py** requires one input parameter:

```
# Maximum distance (mm) between two objects to be declared the same object.
maxDistance = 2.0
```

An instance of the *BubbleTracker* class is created by passing it these values.

```
bt = BubbleTracking.BubbleTracker(maxDistance)
```

We then proceed to update the tracker with a sequence of image frames. Each frame must correspond to an instance of the *BubbleSizing* class.

```
for imageFrame in processedFrames:
    bt.update(imageFrame)
```

Once the sequence of images has been processed, the velocity and diameter of the tracked objects can be determined using the functions build into the *TrackableObject* class also contained within **BubbleTracking.py**.

```
velocities = [] # Prepare empty lists to append new values to.
diameters = []

for idx in bt.trackedObjects:
    trackedObj = bt.trackedObjects[idx]

    d_mean, d_std = trackedObj.calc_diameter()
    diameters.append(d_mean)

    v = trackedObj.calc_velocity()
    velocities.append(v)
```

8 Bringing it all together

An example has been provided showing how we can use the above programs to track bubbles rising through a fluid. The rate determining step is the image construction process not the data acquisition process. Therefore, it is recommended that data acquisition is completed prior to image construction and processing. Using this approach, the imaging, image processing and object tracking can be performed in **for** loop as shown:

```
import Acquisition, Imaging, BubbleSizing, BubbleTracking

# Set Imaging Parameters
pitch = 0.7e-3
velocity1 = 1496
zBegin = -5.0e-3
nPixels = 512

# Set Tracking Parameters
maxDisappeared = 1
maxDistance = 2.0
bt = bubbleTracking.BubbleTracker(maxDistance)
```

```
FIToolbox = Acquisition.DSL()
```

```
FIToolbox.initialise()
```

```
nFRDs = 100
FRDs = FIToolbox.acquire_multiple_FRDs(nFRDs)

for nFRD in FRDs:
    TFM = Imaging.PyTFM(nFRD,pitch)
    TFM.setVelocity(velocity1=velocity1)
    TFM.setImage(nPixels=nPixels,zBegin=zBegin)
    TFM.doTFM(coefficients_TFM=True,SCF=True)
    processedFrame = BubbleSizing(TFM)
    bt.update(processedFrame)
```