TeraFAST software. Python API reference

Terasense Group, Inc
2033 Gateway Place, Suite 500 San Jose, CA 95110, USA

January 15, 2024

Contents

1 Installation

TeraFAST software is a Python package. It requires Python and the following extension modules to be installed:

- Python 3.10 http://www.python.org/download/
- NumPy 1.24 http://www.numpy.org/
- OpenCV 4.7 http://opencv.org with Python bindings
- wxPython 4.2 http://www.wxpython.org
- Pyserial 3.5 http://sourceforge.net/projects/pyserial/files/

It also depends on Microsoft Visual C++ Redistributable 2015

Installers for all required software are provided in the default installation package. Download it from our software distribution page, unzip it to a temporary folder, and execute *install.bat* script by double-clicking it in Windows Explorer. It will run all necessary installers for you. Install all packages with default settings, agreeing to EULA where required.

If you already have a Python installation you may download only the wheel file (or get it from the default zip file, which also contains documentation, C/C++ software, etc.) and install it using PIP or your favorite package manager; appropriate dependencies should be installed automatically.

TeraFAST software only guaranteed to work with Python 3.10 or later. Since Python dropped support for Windows 7 at version 3.9, it can't be installed on Windows 7.

2 Overview

Two main modules are supplied in the *teraFAST* package, *teraFAST.processor* and *teraFAST.worker*. The first provides *processor* class for data acquisition and basic processing (including background compensation and normalizaion); the second one provides *Worker* class, which can be used to convert data to a RGB image.

teraFAST.processor module works by spawning a separate process for asynchronous data acquisition using Python's *multiprocessing* module, while data processing is running in parallel in its own thread using Python's *threading* module. The process and the thread are created when acquisition is started and destroyed when it is stopped so that computer resources are not spent unnecessary.

Data processing consists of two main parts — background compensation and normalization. It is performed within *teraFAST.processor* module.

By default, background data are read from a device-specific config file. They can be re-recorded and it should be done if external temperature is changed. Obviously, incoming radiation should be switched off. Normalizaion data are read from the same config file and stored in a dictionary; there is a default set, supplied from the factory, but you can record your own (which will be stored under the label "recorded") to take into account distribution of the incoming radiation and effectively flatten the field. Pixels that produce too low signal-to-noise ratio during recording (either due to defect or to being in a dark spot) are marked as non-performing (the threshold separating performing from non-performing pixels can be changed after the recording). Readouts from non-performing pixels are substituted by zeros.

3 Module *teraFAST.processor*

class *teraFAST.processor*.**processor**([**threaded, config, defaults, flags**])

This class provides main data acquisition and processing capabilities.

Parameters:

threaded deprecated (boolean, default: *True*)

config path to a configuration file containing background and normalization data (default:*None*)

defaults path to a file with auxilliary default settings (default:*None*)

flags data processing flags (default: *teraFAST.processor*.DEFAULT_FLAGS)

Data processing flags:

DIFFERENCE turn on the difference mode

DEFAULT_FLAGS default value, equivalent to none, 0x0.

3.1 Methods

The module provides methods in several groups:

Main:

- start([callback, errorcallback, resume])
- stop([join])
- read()
- read_raw()

Acquisition settings:

- SetFrameLength(value)
- SetRate(rate)
- GetRate()
- GetRateRange()
- SetDifference([on])

Synchronization settings:

- SetSyncOut(on)
- GetSyncOut()
- SetExternalSync(on,[edge, divider])
- GetExternalSync()

- SetEdge(rise)
- GetEdge()
- SetSyncDivider(value)
- GetSyncDivider()
- SetSyncHoldoff(value)
- GetSyncHoldoff()

Background and normalization:

- SetBG(data)
- SelectBG(val)
- SetNorm([data, mask])
- GetNormList()
- SelectNorm([val])
- GetSelectedNorm()
- LoadConfig([zipName])
- SaveConfig(filename)
- RecordBG([count, callback])
- RecordNorm([count, callback])

Processing

- SetThreshold(val)
- GetThreshold()
- SetAccumulation([on])
- SetAccuLength(val)
- GetAccuLength()
- ResetAccumulation()

3.1.1 Main

start([callback, errorcallback, resume])

Starts acquisition process and corresponding threads. Callback and errorcallback are functions that should be called when new data became available or an error occurs.

Parameters:

callback a target to be called on each aquisition cycle as *callback(data)*, where *data* is a *numpy.ndarray* containing processed data within [0,1] interval (Notice, that *data* may be *None* if the data queue is empty for too long). Callback function is executed in the same thread as data processing, it should return *False* during normal operation and *True* if acquisition should be aborted (default: *None*);

errorcallback a target to be called in case of an error as *errorcallback(error, critical)*, where *error* is an instance of *Exception*, and *critical* is a boolean value indicating wheather operation should be stopped or may continue (default: *None*);

resume if *True* and *callback* is *None* the value of *callback* from previous call to this function will be used (default: *False*).

Returns:

None

stop([join])

Stops acquisition and processing threads. If acquisition is not running it has no effect.

Parameters:

join if *True*, it will try to join acquisition and processing threads, possibly causing delay. (default: *True*).

Returns:

None

read()

Reads processed data. The method will work only if acquisition have been started using *start*, otherwise it will return *None*. This method will not read the same frame twice and it will block if the new frame is not available yet.

Returns:

data *numpy.ndarray* containing processed data within [0,1] interval.

read_raw()

Reads raw unprocessed data. If **processor** instance have been created in a multi-threaded mode, the method will work only if acquisition was started using *start*, otherwise it will return *None*. This method is not intended for end-user.

Returns:

data *numpy.ndarray* containing unprocessed data within [-32768, 32767] .

3.1.2 Acquisition settings

SetFrameLength(value)
Sets length (number of lines) in a frame that is returned by **read()** and **read_raw()**.

Parameters:

value new value for the frame length (integer, value> 1).

None

SetRate(rate)
Changes current acquisition rate.

Parameters:

rate new rate value, lines per second.

Returns:

err If acquisition is not running returns *None* for success or *Exception* instance for failure; If acquisition is running always returns *None* and in case of failure **errorcallback()** is called by the processing thread.

GetRate()
Gets current acquisition rate.

Returns:

rate The acquisition rate, lines per second. The return value may differ from the one used at **SetRate()** due to rounding errors.

GetRateRange()
Gets admissible rate range.

Returns:

(min, max) a tuple with minimal and maximal rate values, lines per second.

SetDifference([on])
Turns the difference mode on or off.

Parameters:

on *True* to turn on, *False* to turn off (default: *True*).

Returns:

None

3.1.3 Synchronization settings

SetSyncOut(on)

Turns the synchronization output signal on or off on the corresponding connector (see Manual). The signal changes level at the moment of acquisition of each line; it is on by default.

Parameters:

on *True* to turn on, *False* to turn off.

Returns:

None

GetSyncOut()

Returns current state of the synchronization ouptut signal.

Returns:

state *True* is on, *False* is off.

SetExternalSync(state,[edge=*None*, divider=*None*])

Turns the external synchronization mode on or off. If it is on, the device would acquire data (a line) only when a synchronization pulse arrives to SYNC IN pin. If it is off, it works from the internal synchronization source. Additional parameters sets up the details of the sync signal (see description of the corresponding functions below).

Parameters:

state *True* is on, *False* is off.

rise *True* corresponds to rising edge, *False* to the falling edge; *None* means no change.

divider *integer* the divider value. It is coerced to 1 to 32768 interval; *None* means no change.

Returns:

None

GetExternalSync()

Returns current state of the external synchronization mode.

Returns:

state *True* is on, *False* is off.

SetEdge(rise)

Sets the active edge for external synchronization signal.
The setting change takes effect on the next acquisition start (or on switching on of the external synchronization mode).

Parameters:

rise *True* corresponds to rising edge, *False* to the falling edge.

Returns:

None

GetEdge()

Returns which edge of the external synchronization is active at the moment.

Returns:

rise *True* corresponds to rising edge, *False* to the falling edge.

SetSyncDivider(n)

Sets the frequency divider for the external synchronization signal. It means that the acquisition would be run only on each **n**-th pulse at the synchronization input (it is not required for them to be equally sapced in time).
The setting change takes effect on the next acquisition start (or on switching on of the external synchronization mode). Parameters:

n *integer* divider value. It is coerced to 1 to 32768 interval

Returns:

None

GetSyncDivider()

Returns current value of the frequency divider for the external synchronization signal.

Returns:

n *integer*the value of the divider in 1 to 32768 range.

SetSyncHoldoff(n)

Sets value of the holdoff paramete for the external synchronization signal which roughly filters the synchronization input. Larger values correspond to slower expected synchronzization rate. Generally, your pulse width would be longer than 2^n microseconds and the period should be at least $2^(n+1)$ microseconds. The setting change takes effect on the next acquisition start (or on switching on of the external synchronization mode).

Parameters:

n *integer* the holdoff parameter value (0 to 15).

Returns:

None

GetSyncHoldoff()

Returns current value of the holdoff parameter.

Returns:

n *integer*the value of the divider in 0 to 15 range.

3.1.4 Background and normalization-related

SetBG(data)

Sets data as current backgound data to be used in processing.

Parameters:

data *numpy.ndaray* with data to be used as a background data. If it is *None*, empty array is used (i.e. no background compensation is performed).

Returns:

None

SelectBG(val)

Selects a background data from existing list. The data are set as current backgound data to be used in processing.

Parameters:

val index of the data in the list (currently only 0 is a valid choice); if it is out of range, empty array is used.

Returns:

None

GetSelectedBG()

Gets index of a currently selected background data within the list.

Returns:

idx index of a currently selected background; *None* if no background compensation is performed.

SetNorm([data, mask])

Sets current normalization data to be used in processing.

Parameters:

data *numpy.ndaray* with data to be used for normalization. If it is *None*, empty array is used (i.e. no normalization is performed).

mask *numpy.ndaray* with corresponding mask data to be used for normalization. If it is *None*, all pixels are assumed to be good.

Returns:

None

SelectNorm([val])

Selects normalization from the dictionary by the key (either "default", "recorded", or *None*)

Parameters:

val key value. If the key is *None* or does not exists normalization is switched off.

Returns:

key value of the key on success or *None* otherwise.

GetNormList()

Gets a list of normalizations as dictionary. It includes all possible keys and boolean values indicate whether the corresponding normalizaion is available at the moment.

Returns:

dict dictionary of a form {"default": *True— False*, "recorded": *True— False*}

GetSelectedNorm()

Gets key for currently selected normalization.

Returns:

key key for the currently selected normalization ("default" or "recorded") or *None* if none is selected.

LoadConfig([zipName])

Loads previously saved background and normalization data from a configuration file and puts them into the corresponding list and dictionary.

Parameters:

zipName filename for a configuration file. If it is not provided, default configuration file is loaded. If it is a relative path, it is relative to the module folder.

Returns:

None

SaveConfig(filename)

Saves complete list of backgrounds and normalization data dictionary to a configuration file to be loaded later.

Parameters:

filename filename for a configuration file. If it is a relative path, it is relative to the module folder.

Returns:

None

RecordBG([count, callback])

Record background data for all available exposures and places them into the background list.

Parameters:

count number of repetitions used for averaging. If 0 or not provided, then default value is used (30, may be changed in defaults file).

callback callback to indicate progress. It is called with completed percentage and it is expected to return tuple (*continue*, *skip*), where continue is *True* unless the process should be aborted (see wx.ProgressDialog from wxPython package).

Returns:

success *True* if completed successfully, *False* if canceled.

RecordNorm([count, callback])

Record normalization data and put them into the normalization dictionary under "recorded" key.

Parameters:

count number of repetitions used for averaging. If 0 or not provided, then default value is used (30, may be changed in defaults file).

callback callback to indicate progress. It is called with completed percentage and it is expected to return tuple (continur, skip), where continue is *True* unless the process should be aborted (see wx.ProgressDialog from wxPython package).

Returns:

success *True* if completed successfully, *False* if canceled.

3.1.5 Processing

SetThreshold(val)

Set threshold value for the signal-to-noise ratio to declare pixels unusable with recorded normalization.

Parameters:

val new value for threshold (default: 10.0, may be changed in defaults file; invalid values less than 1 defaults to 10).

Returns:

None

GetThreshold()

Returns current threshold value.

Returns:

val current threshold value.

SetAccumulation([on])

Turns accumulation on or off.

Parameters:

on *True* to turn on, *False* to turn off (default: *True*).

Returns:

None

SetAccuLength(val)

Sets accumulation length (window size for time-domain filtering).

Parameters:

val accumulation length (coerced to [1,100]).

Returns:

None

This method does not turn accumulation on! Use SetAccumulation().

GetAccuLength()
Returns current accumulation length (window size for time-domain filtering). Result does not depend on whether the accumulation is on or off.

Returns:

val current accumulation length.

ResetAccumulation()
Resets accumulated data (i.e. starts accumulation anew).

Returns:

None

3.2 Properties

3.2.1 General

X_SIZE X dimension of the sensor array (integer, read-only).

Y_SIZE Y dimension of the sensor array (integer, read-only).

bgList background list containing background information for each available exposure. Each item is an instance of *teraFAST.ref.RefData*. See SelectBG, GetSelectedBG, LoadConfig, RecordCurrentBG, RecordBG methods.

normDict normalization dictionary containing normalization information. By default it contains "default" and "recorded" keys (with values possibly being *None*). Each item is an instance of *teraFAST.ref.RefData*. See GetNormList, SelectNorm, GetSelectedNorm, LoadConfig,

RecordNorm methods.

3.2.2 Multi-threading

Generally, it is recommended to use either callback of the start method or read method to get access to the data in multi-threaded mode. However, if you want to have a direct access, here are several properties to do that.

result *numpy.ndarray* with the shape (X_SIZE, Y_SIZE), which contains processed data during multi-threaded operation.

datalock instance of *threading.Lock()*. Acquire it if you access *result* property directly.

ready instance of *threading.Event()*. It is set when *result* property is renewed.

4 Module *teraFAST.worker*

class ***teraFAST.worker.*Worker(size, [flags])**

This class provides means for converting data array to an image with some additional processing. It relies on *Numpy* and *OpenCV*.

Parameters:

size a tuple with dimensions of the imag/data array (width, height)

flags processing flags (default: *teraFAST.processor.*DEFAULT_FLAGS)

Data processing flags:

FALSECOLOR produce image in false colors (rainbow) instead of b/w (tinted)

SMOOTH smooth image (space-domain filtering)

NEGATIVE invert image

MEDIAN use median filtering instead of gaussian blurring for smoothing

MIRROR mirror the image

DEFAULT_FLAGS default value, equivalent to FALSECOLOR|SMOOTH

4.1 Methods

The module provides the following methods:

- makeImg(data)
- SetBrightness(black,white)
- SetContrast(black,white)
- SetGamma(val)
- SetSmoothness(val)
- GetSmoothness()
- data2RGB(data)

makeImg(data) Generates image from data according to the current settings.

Parameters:

data input data (one-channel [0,1] array of floats).

Returns:

img three-channel RGB image.

SetBrightness(black,white) Sets brightness using black and white points.

Parameters:

black black point value in [0, white)

white white point value in (black, 1]

Returns:

None

SetContrast(black,white) Sets contrast using black and white points.

Parameters:

black black point value in [0, white)

white white point value in (black, 1]

Returns:

None

SetGamma(val) Sets gamma value.

Parameters:

val gamma value (gamma ¿ 0).

Returns:

None

SetSmoothness(val) Sets smoothness parameter.

Parameters:

val smoothness parameter (0, 100]. For gaussian blur smoothing it is the standard deviation $\times 100$, for median smoothing it sets 3×3 kernel if $val \leq 50$ and 5×5 kernel otherwise.

Returns:

None

GetSmoothness() Gets smoothness value.

Returns:

val value of the smoothness parameter (0, 100]

data2RGB(data) Converts one-channel BW data to three-channel RGB data, output depends on the presense of FALSECOLOR flag in *processFlags* property. Parameters:

data input data (one-channel [0,1] array of floats).

Returns:

img three-channel RGB image.

4.2 Properties

processFlags mask, which defines processing options according to data processing flags (see description of the constructor). It may be changed at any time.

brightness brightness parameter, float, possible values in $[-1, 1]$ range.

contrast contrast parameter, float, possible values in $(0, \text{inf})$ range.

size tuple with 2D dimensions of the data (width, height) (read-only).

5 Examples

5.1 The simplest program

Data acquisition from the camera is performed in a separate thread. You need to start acquisition explicitly, otherwise *read()* would return *None*.

```
from teraFAST import processor as tp
def main():

    frameLength = 256 #number of lines returned per frame

    source = tp.processor(rows=frameLength)

    #You need to start data acquisition explicitly
    source.start()

    for i in range(100):
        data = source.read()
        #do something with data
        print data.shape

    #Call stop() to stop acquisition and join the acquisition and processing thread
    source.stop()

if __name__ == '__main__':
    main()
```

5.2 A multithreaded version using callbacks

```
from teraFAST import processor as tp
import time
count = 0
def callback(data):
    global count

    if data is not None:
        #do something with data
        print count, data.shape
    else:
        print "Data queue is empty."
    count +=1
    if count >= 100:
        # you may stop acquisition by returning True from the callback (instead of calling processor.st
        return True

def main():
```

```
    frameLength = 256 #number of lines returned per frame

    source = tp.processor(rows=frameLength)

    #You need to start data acquisition explicitly
    source.start(callback)

    # Sleep for some time or do something while acquisition is going on
    time.sleep(3)
##  time.sleep(10)

    #stop() function should be called from the main thread, not from the callback
    source.stop()


if __name__ == '__main__':
    main()
```

5.3 Image generation

```
from teraFAST import processor as tp
from teraFAST import worker as tw

def main():
    frameLength = 256 #number of lines returned per frame

    source = tp.processor(rows=frameLength)
    convert = tw.Worker(size = (source.X_SIZE,source.Y_SIZE))


    #You need to start data acquisition explicitly
    source.start()
    for i in range(100):
        data = source.read()
        img = convert.makeImg(data)
        #do something with image
        print img.shape

    #Call stop() to stop acquisition and join the acquisition and processing thread
    source.stop()

if __name__ == '__main__':
    main()
```

5.4 Use of background and normalization

```
from teraFAST import processor as tp
from teraFAST import worker as tw

def ticker(progress):
    """Progress indicator. To abort process, return (False, False) tuple. """
    if progress < 100:
        print ".",
    else:
        print " "
    return (True, False)

def main():
    frameLength = 256 #number of lines returned per frame

    source = tp.processor(rows=frameLength)

    raw_input("Prepare to record background. Switch off incoming radiation and press Enter")

    # Background is recorded.
    source.RecordBG(callback = ticker)

    raw_input("Prepare to record normalization. Switch on incoming radiation and press Enter")


    # Normalization is recorded. The recorded normalization is automatically selected.
    source.RecordNorm(callback = ticker)

    # You may want to use SaveConfig/LoadConfig to avoid repeating recording background/normalization pr

    #You need to start data acquisition explicitly
    source.start()

    for i in range(100):
        data = source.read()
        #do something with image
        print data.shape

    #Call stop() to stop acquisition and join the acquisition and processing thread
    source.stop()

if __name__ == '__main__':
    main()
```