TexoMatlab library V2.5

for the UltraSonix RP 500

Documentation

Imperial College London, UK, 2009
INSERM U556, France, 2010

This documentation is to be used alongside that provided with the UltraSonix scanner, and assumes familiarity with that material.

Credits and acknowledgments:

- Original creation (Version 1.0): Jean Martial Mari, PhD, 2007.
- Library upgrade (Version 2): Richard Criddle / Jean Martial Mari 2009
- Library upgrade (Version 2.5): Jean Martial Mari 2010.
- This library was created with the support of
  - Dr MengXing Tang and Emeritus Prof. Colin G. Caro, Department of Bioengineering, Imperial College London, UK.
  - Dr Cyril Lafon, INSERM U556, Lyon France.

OUTLINE

1 TERMS AND CONDITIONS 4

2 OVERVIEW 5

3 PUBLIC FUNCTIONS IN THE C++ LIBRARY 7

3.1 TexoTools functions 7

3.2 Scanner functions 10

3.3 Sequence creation functions 13

4 DATA FORMATS 19

4.1 Configuration files 19

4.2 Returned settings 19

4.3 Saved image data 20

5 CONTROLLING THE LIBRARY AND SCANNER FROM MATLAB 23

5.1 Standard sequences 23

5.2 Two-pulsed sequences 26

5.3 Doppler sequences 27

5.4 Custom sequences 27

5.5 Multiple sequences 28

1 TERMS AND CONDITIONS

a) This software is provided as is, and is for research purpose only.

b) The authors and the Imperial College London cannot be held responsible for any loss or damages occurring in relation with the use of this library, or related programs and code.

c) This library is provided for free and it is strictly forbidden to sell it alone or as part of a package.

d) This library is intended to be used by professionals of ultrasound imaging that are aware of the risks linked with ultrasound imaging to both living being and equipment.

e) This library is not intended for clinical use.

2 OVERVIEW

The TexoMatlab llibrary and its TexoTools stands between the programmable UltraSonix RP 500 and a Matlab interface, granting a user full control over the functions provided with the scanner in a way that requires no programming knowledge to operate. The basic concept is shown in Fig. 1. All that the user has to do is write simple Matlab scripts (examples are provided with the library and explained in section 5) and the library will give them access to the power and flexibility of the programmable scanner.

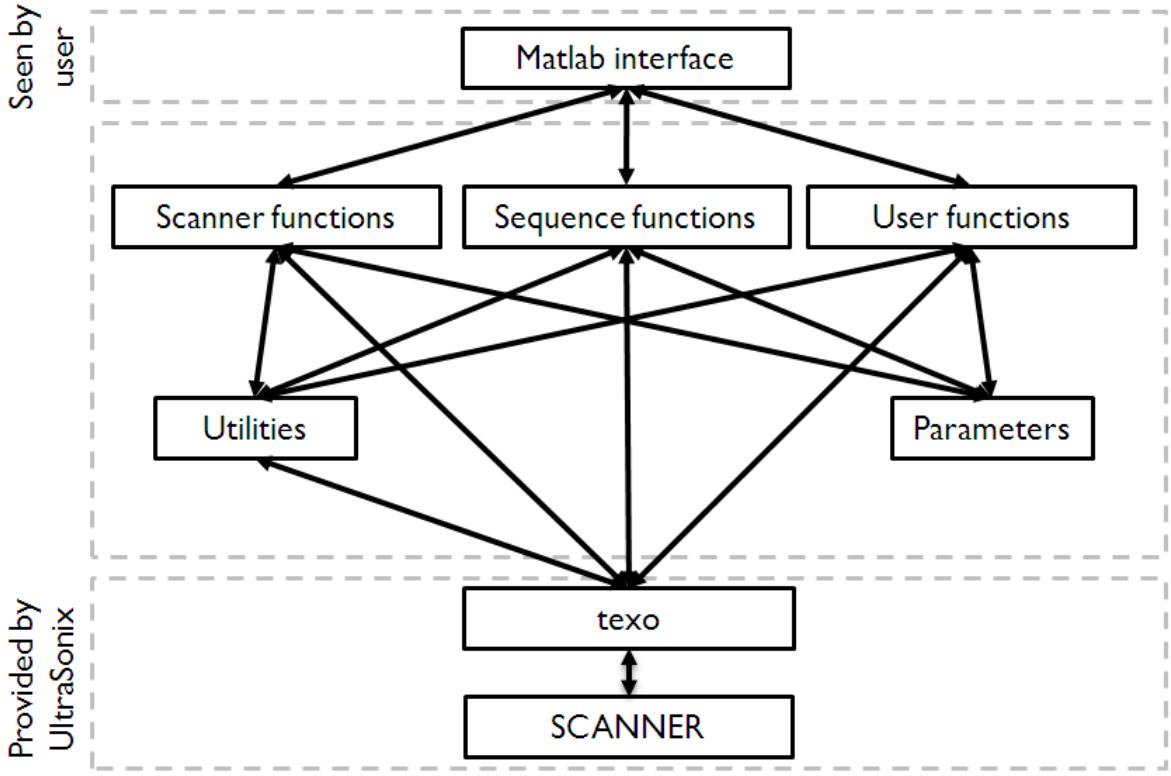

Figure 1: The architecture of the TexoMatlab library. From the Matlab interface, a user can access user functions with basic operations required for every image, scanner functions which directly control the functions provided by UltraSonix and sequence functions which select pre-defined or customised sequences. Using these, he can control the scanner, and get back the data that it produces.

Installation of the library to a new machine requires two steps. First, the zip file containing the library files must be opened and extracted to a suitable folder. Now there is one file that needs to be updated with the new location of the library. In the folder `TexoProcesses` there is a file called `openPaths.m`. Open this file in Matlab

or a text editor and replace the old location with the new folder containing the library files in this line: `rootPath = 'folder name';`

For example: `rootPath = 'D:\SonixProgramming\TexoLib';`

Now new scripts can be written, saved in this folder and executed.

3 PUBLIC FUNCTIONS IN THE C++ LIBRARY

3.1 TexoTools functions

The TexoTools object contains the basic functions for preparing the scanner and taking a scan. All of these functions will need to be called in nearly every Matlab script which uses the library to take images, as they provide the fundamental control which the user will need over the scanner.

`int loadConfigData(const char * configFilePathName)`

Reads the configuration file specified by the user and sets the library variables and the transmit/receive parameters accordingly. The function also communicates with the scanner to send the correct power values, add the appropriate TGC curve and activate the specified probe connector, so the scanner is ready for sequence definition.

[in]	configFilePathName	A path to the settings file

[out]	0	Indicates a successful function call
[out]	-210	Unexplained error occurred in the function call
[out]	-211	The scanner isn't ready to receive settings: imaging or not initialised
[out]	-212	Failed to add power values to scanner
[out]	-213	Failed to add TGC curve to scanner
[out]	-214	Failed to activate probe connector
[out]	-215	Failed to add power values and TGC curve to scanner
[out]	-216	Failed to add power values to scanner and activate probe connector
[out]	-217	Failed to add TGC curve to scanner and activate probe connector
[out]	-218	Failed to add power values and TGC curve and activate probe connector
[out]	-219	Unexplained error occurred while sending settings to the scanner
[out]	-221	The specified configuration file failed to open
[out]	-222	No configuration file was specified, and default file failed to open
[out]	-223	Unexplained error occurred while reading configuration file

`int takeScanForDuration(int duration)`

Takes a scan for the specified duration.

`[in]`	`duration`	The scan duration in milliseconds

`[out]`	`0`	Indicates a successful function call
`[out]`	`-230`	Unexplained error occurred in the function call
`[out]`	`-231`	The scanner is not ready: not initialised or already scanning
`[out]`	`-232`	Call to UltraSonix function `runImage` failed
`[out]`	`-233`	Call to UltraSonix function `stopImage` failed

`int takeScan(void)`

Takes a scan for a pre-specified duration.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-240`	Unexplained error occurred in the function call
`[out]`	`-241`	The scanner is not ready: not initialised or already scanning
`[out]`	`-242`	Call to UltraSonix function `runImage` failed
`[out]`	`-243`	Call to UltraSonix function `stopImage` failed
`[out]`	`-244`	The scan duration has not been pre-set, so no scan can be taken

`int saveData(char * fileName, int maxNumberOfFramesToSave)`

Saves the results of the scan to a file. The first 1kb of the file contains various important settings that are needed to correctly interpret the scan. Next is the data from each frame of the scan. Then it writes the centre element, the line angle and the pulse shape for each line of the image, because they can change from line to line. See section 3.3 for the structure of the saved file.

`[in]`	`fileName`	The destination file to save data to
`[in]`	`maxNumberOfFramesToSave`	However many frames have been imaged, the number saved will be capped to this

[out]	0	Indicates a successful function call
[out]	-250	Unexplained error occurred in the function call
[out]	-251	The scanner is not ready: not initialised or is currently scanning
[out]	-252	The scanner has not imaged any frame to save
[out]	-253	The specified save file will not open
[out]	-254	The header has exceeded the 1024 byte size limit
[out]	-255	The save file will not close

`int returnScannerSettings(bool allSettingsRequested)`

Uses a Matlab engine to put the scan settings directly into the Matlab workspace so they can be examined by a user, who can request either a full list of settings or a brief list of the important ones. The engine creates a variable called `TML_ScannerSettings` in Matlab which contains the settings. See section 3.2 for the format of this variable.

[in]	`allSettingsRequested`	`true` for complete list, `false` for short list

[out]	0	Indicates a successful function call
[out]	-260	Unexplained error occurred in the function call
[out]	-261	The Matlab engine did not open
[out]	-262	The settings could not be saved into the Matlab environment
[out]	-263	The Matlab engine could not be closed

3.2 Scanner functions

This object contains the functions that directly access the UltraSonix functions or variables without doing any computation of their own, so provide direct control over the scanner. The first set of functions use UltraSonix functions to start and stop the engine, start and stop images and set the power values for the scan. The second set each change a parameter in the scanner to obtain a different image. This allows more precise control over the data being obtained.

`int startEngine(char * settingsPath, int freq)`

Initialises the scanner's electronics.

`[in]`	`settingsPath`	Folder with the scanner settings data
`[in]`	`freq`	Initialisation frequency in MHz (20 or 40)

`[out]`	`0`	Indicates a successful function call
`[out]`	`-110`	TexoTools object couldn't be created – the library is malfunctioning
`[out]`	`-310`	Unexplained error in the function call
`[out]`	`-311`	The scanner is not ready: already initialised
`[out]`	`-312`	Call to UltraSonix function `init` failed
`[out]`	`-313`	The specified frequency is invalid, 20MHz used as default

`int setPower(int power, int maxPositive, int maxNegative)`

Sets the power values for the sequence, all values must be between 0 and 15.

`[in]`	`power`	The overall power
`[in]`	`maxPositive`	Max power allowed on a positive transmit
`[in]`	`maxNegative`	Max power allowed on a negative transmit

`[out]`	`0`	Indicates a successful function call
`[out]`	`-320`	Unexplained error in the function call
`[out]`	`-321`	The scanner is not ready: already imaging or not initialised
`[out]`	`-322`	Power values are outside the required range (0-15)
`[out]`	`-323`	Call to UltraSonix function `setPower` failed

`int startImaging(void)`

Begins a scan.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-330`	Unexplained error in the function call
`[out]`	`-331`	The scanner is not ready: already imaging or not initialised
`[out]`	`-332`	Call to UltraSonix function `runImage` failed

`int stopImaging(void)`

Stops the scan.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-340`	Unexplained error in the function call
`[out]`	`-341`	The scanner is not ready: not currently scanning
`[out]`	`-342`	Call to UltraSonix function `stopImage` failed

`int stopEngine(void)`

Powers down the scanner and all electronics.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-350`	Unexplained error in the function call
`[out]`	`-351`	The scanner is not ready: not initialised
`[out]`	`-352`	Call to UltraSonix function `shutdown` failed

`int setFrequency(int freq)`

Changes the value of `tx.frequency` to `freq`, returns 0 if successful, -361 if not.

`int setFocusDistance(int focus)`

Changes the value of `tx.focusDistance` to `focus`, returns 0 if successful, -362 if not.

`int setAcquisitionDepth(int depth)`

Changes the value of `rx.acquisitionDepth` to `depth`, returns 0 if successful, -363 if not.

`int setAngle(int angle)`

Changes the value of `tx.angle` to `angle`, returns 0 if successful, -364 if not.

`int setCenterElement(int centerElement)`

Changes the value of `tx.centerElement` and `rx.centerElement` to `centerElement`, returns 0 if successful, -365 if not.

`int setTxCenterElement(int centerElement)`

Changes the value of `tx.centerElement` only to `centerElement`, returns 0 if successful, -366 if not.

`int setRxCenterElement(int centerElement)`

Changes the value of `rx.centerElement` only to `centerElement`, returns 0 if successful, -367 if not.

`int setPulseShape(char * pulseShape)`

Changes the value of `tx.pulseShape` to `pulseShape`, returns 0 if successful, -368 if not.

3.3 Sequence creation functions

These functions are used to define sequences, so must be run before the scan is performed. They come in two groups. The first set of functions (`create…Sequence`) each create pre-defined sequences, so the Matlab user can have a sequence defined automatically. The second set (`beginSequence`, `addLine` and `endSequence`) is for users who need more control over the sequences they are running, and will allow customised sequences to be defined line-by-line.

`int createStandardSequence(void)`

Creates a standard scan sequence using the parameter values which have already been set in the library and loads it into the scanner ready for imaging.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-410`	Unexplained error in the function call
`[out]`	`-411`	The scanner is not ready: already imaging or not initialised
`[out]`	`-412`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-413`	Call to UltraSonix function `addLine` failed for sector scan
`[out]`	`-414`	Call to UltraSonix function `addLine` failed for linear scan
`[out]`	`-415`	Call to UltraSonix function `endSequence` failed

`int createStandardSequenceWithShape(char * pulseShape)`

Creates a standard scan sequence using the specified pulse shape and loads it into the scanner ready for imaging.

`[in]`	`pulseShape`	Must consist of `'+'`, `'-'` and `'0'`

`[out]`	`0`	Indicates a successful function call
`[out]`	`-420`	Unexplained error in the function call
`[out]`	`-421`	The scanner is not ready: already imaging or not initialised
`[out]`	`-422`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-423`	Call to UltraSonix function `addLine` failed for sector scan
`[out]`	`-424`	Call to UltraSonix function `addLine` failed for linear scan
`[out]`	`-425`	Call to UltraSonix function `endSequence` failed

`int createAlternatingSequence(void)`

Creates an alternating sequence with two pulses fired along each line with different pulse shapes, so any non-linear effects can be detected. Pre-set library parameter values are used, and the sequence is loaded into the scanner ready for imaging.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-430`	Unexplained error in the function call
`[out]`	`-431`	The scanner is not ready: already imaging or not initialised
`[out]`	`-432`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-433`	Call to UltraSonix function `addLine` failed for sector scan, pulse A
`[out]`	`-434`	Call to UltraSonix function `addLine` failed for sector scan, pulse B
`[out]`	`-435`	Call to UltraSonix function `addLine` failed for linear scan, pulse A
`[out]`	`-436`	Call to UltraSonix function `addLine` failed for linear scan, pulse B
`[out]`	`-437`	Call to UltraSonix function `endSequence` failed

`int createAlternatingSequenceWithShape (char * pulseShapeA, char * pulseShapeB)`

Creates an alternating sequence in which two pulses are fired along each line with different pulse shapes, so any non-linear effects in the target can be detected. The specified pulse shapes are used and pre-set library values for the other parameters, and the sequence is loaded into the scanner ready for imaging.

`[in]`	`pulseShapeA`	Must consist of `'+'`, `'-'` and `'0'`
`[in]`	`pulseShapeB`	Must consist of `'+'`, `'-'` and `'0'`

`[out]`	`0`	Indicates a successful function call
`[out]`	`-440`	Unexplained error in the function call
`[out]`	`-441`	The scanner is not ready: already imaging or not initialised
`[out]`	`-442`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-443`	Call to UltraSonix function `addLine` failed for sector scan, pulse A
`[out]`	`-444`	Call to UltraSonix function `addLine` failed for sector scan, pulse B
`[out]`	`-445`	Call to UltraSonix function `addLine` failed for linear scan, pulse A
`[out]`	`-446`	Call to UltraSonix function `addLine` failed for linear scan, pulse B
`[out]`	`-447`	Call to UltraSonix function `endSequence` failed

`int createInvertingSequence(void)`

Creates an inverting sequence, a special subset of an alternating sequence in which the second pulse shape is the inverse of the first. Pre-set library parameter values are used, and the sequence is loaded into the scanner ready for imaging.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-450`	Unexplained error in the function call
`[out]`	`-451`	The scanner is not ready: already imaging or not initialised
`[out]`	`-452`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-453`	Call to UltraSonix function `addLine` failed for sector scan, pulse A
`[out]`	`-454`	Call to UltraSonix function `addLine` failed for sector scan, pulse B
`[out]`	`-455`	Call to UltraSonix function `addLine` failed for linear scan, pulse A
`[out]`	`-456`	Call to UltraSonix function `addLine` failed for linear scan, pulse B
`[out]`	`-457`	Call to UltraSonix function `endSequence` failed

`int createInvertingSequenceWithShape (char * pulseShape)`

Creates an inverting sequence, a special subset of an alternating sequence in which the second pulse shape is the inverse of the first. The specified shape is used for the first pulse, and its inverse is found and used for the second. Otherwise, pre-set library parameter values are used, and the sequence is loaded into the scanner ready for imaging.

`[in]`	`pulseShape`	Must consist of `'+'`, `'-'` and `'0'`

`[out]`	`0`	Indicates a successful function call
`[out]`	`-460`	Unexplained error in the function call
`[out]`	`-461`	The scanner is not ready: already imaging or not initialised
`[out]`	`-462`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-463`	Call to UltraSonix function `addLine` failed for sector scan, pulse A
`[out]`	`-464`	Call to UltraSonix function `addLine` failed for sector scan, pulse B
`[out]`	`-465`	Call to UltraSonix function `addLine` failed for linear scan, pulse A
`[out]`	`-466`	Call to UltraSonix function `addLine` failed for linear scan, pulse B
`[out]`	`-467`	Call to UltraSonix function `endSequence` failed

`int createHighSpeedSequence (char * pulseShape,int numberLines, int depth, int txFrequency, int FocusDistance)`

Creates a sequence with one pulse shape which is more customisable than a standard sequence as a variety of parameters can be passed as arguments.

`[in]`	`pulseShape`	Must consist of `'+'`, `'-'` and `'0'`
`[in]`	`numberLines`	The number of lines to make the image
`[in]`	`depth`	The max acquisition depth of the image
`[in]`	`txFrequency`	The transmitted ultrasound frequency
`[in]`	`focusDistance`	The focus depth of the image

`[out]`	`0`	Indicates a successful function call
`[out]`	`-470`	Unexplained error in the function call
`[out]`	`-471`	The scanner is not ready: already imaging or not initialised
`[out]`	`-472`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-473`	Call to UltraSonix function `addLine` failed for sector scan
`[out]`	`-474`	Call to UltraSonix function `addLine` failed for linear scan
`[out]`	`-475`	Call to UltraSonix function `endSequence` failed

`int createDopplerSequence(char *pulseShape, int lineNumber, int repeat)`

Creates a Doppler sequence – for determining the velocity of blood flow – and loads it into the scanner ready for imaging. The total number of lines will be the product of `lineNumber` and `repeat`.

`[in]`	`pulseShape`	Must consist of `'+'`, `'-'` and `'0'`
`[in]`	`lineNumber`	The number of lines to be transmitted
`[in]`	`repeat`	How many times to transmit each line

`[out]`	`0`	Indicates a successful function call
`[out]`	`-480`	Unexplained error in the function call
`[out]`	`-481`	The scanner is not ready: already imaging or not initialised
`[out]`	`-482`	Call to UltraSonix function `beginSequence` failed
`[out]`	`-483`	Call to UltraSonix function `addLine` failed for sector scan

[out]	-484	Call to UltraSonix function addLine failed for linear scan
[out]	-485	Call to UltraSonix function endSequence failed

int createMDopplerSequence(char *pulseShape, int repeatCenter)

Creates a multi-mode Doppler image – first a standard B-mode image is taken and then the central line is repeated several times, so Doppler information can be found for that line. The sequence is loaded into the scanner ready for imaging.

[in]	pulseShape	Must consist of '+', '-' and '0'
[in]	repeatCenter	The number of times the central line is to be repeated after the B-mode image

[out]	0	Indicates a successful function call
[out]	-490	Unexplained error in the function call
[out]	-491	The scanner is not ready: already imaging or not initialised
[out]	-492	Call to UltraSonix function beginSequence failed
[out]	-493	Call to UltraSonix function addLine failed for sector scan, B-mode
[out]	-494	Call to UltraSonix function addLine failed for sector scan, Doppler
[out]	-495	Call to UltraSonix function addLine failed for linear scan, B-mode
[out]	-496	Call to UltraSonix function addLine failed for linear scan, Doppler
[out]	-497	Call to UltraSonix function endSequence failed

int beginSequence(void)

Begins a sequence definition, ready for the user to define a customised scan sequence.

[out]	0	Indicates a successful function call
[out]	-510	Unexplained error in the function call
[out]	-511	The scanner is not ready: already imaging or not initialised
[out]	-512	Call to UltraSonix function beginSequence failed

`int addLine(void)`

Adds a single line to the sequence definition, using the parameter values that have been loaded into the library. These can be changed between lines using the variable updating functions in the scanner object (section 2.2).

`[out]`	`0`	Indicates a successful function call
`[out]`	`-520`	Unexplained error in the function call
`[out]`	`-521`	The sequence definition hasn't begun
`[out]`	`-522`	Call to UltraSonix function `addLine` failed

`int endSequence(void)`

Ends the definition of a customised sequence, so the scanner is ready to scan.

`[out]`	`0`	Indicates a successful function call
`[out]`	`-530`	Unexplained error in the function call
`[out]`	`-531`	The sequence definition hasn't begun
`[out]`	`-532`	Call to UltraSonix function end`Sequence` failed

4 DATA FORMATS

4.1 Configuration files

The configuration files used by the library must have a specific format. The parameter name must be written in full, followed by `" = "`, followed by the parameter's value (integer, Boolean or string, depending on the parameter), followed by `";"`. An example with three parameters (one of each data type) is shown below, this will set the scan duration to 2 seconds (2000 ms), sectorial probe to `false` and the primary pulse shape to `"+-"`.

```
scanDuration = 2000;  // Comments can be written here
sectorialProbe = 0;   // For bool, use 0 for false and 1 for true
pulseShapeA = +-;
```

Because the configuration files must have the correct format, the simplest solution is to make a copy of `DefaultConfigFile.txt` (in the `ConfigFiles` folder of the library files) and alter the values there as required.

4.2 Returned settings

The library function `returnScannerSettings` creates an array in the Matlab environment (`TML_ScannerSettings`) containing integers with each setting value. For a partial list (`allSettingsRequested` = `false`), the format of this array is:

```
(1)  scannerPresent
(2)  scanDuration
(3)  sectorialProbe *
(4)  tx.focusDistance
(5)  tx.frequency
(6)  rx.acquisitionDepth
(7)  rx.angle
(8)  rx.applyFocus
(9)  rx.decimation
```

* Boolean, will be displayed as 1 for true or 0 for false

For a full list (`allSettingsRequested = true`), the format of this array is:

```
(1)  samplingFreq              (22) tx.focusDistance
(2)  power                     (23) tx.angle
(3)  maxPositivePower          (24) tx.frequency
(4)  maxNegativePower          (25) tx.speedOfSound
(5)  TGC                       (26) tx.useManualDelays *
(6)  inputSignalConfig         (27) tx.tableIndex
(7)  outputSignalConfig1       (28) tx.useDeadElements *
(8)  outputSignalConfig2
(9)  scannerPresent *          (29) rx.centerElement
(10) readyToScan *             (30) rx.aperture
(11) sectorialProbe *          (31) rx.angle
(12) probeConnector            (32) rx.maxApertureDepth
(13) numberFramesToSave        (33) rx.acquisitionDepth
(14) scanDuration              (34) rx.channelMask[0]
(15) imagingMode               (35) rx.channelMask[1]
(16) subMode                   (36) rx.applyFocus *
(17) numberDopplerLines        (37) rx.useManualDelays *
(18) highSpeedLineNumber       (38) rx.customLineDuration
(19) totalLineNumber           (39) rx.lgcValue
                               (40) rx.tgcSel
(20) tx.centerElement          (41) rx.tableIndex
(21) tx.aperture               (42) rx.decimation
```

* Boolean, will be displayed as 1 for true or 0 for false

For the definitions of these variables see section 4.4 for library variables (1-19) and UltraSonix documentation for transmit (tx) and receive (rx) variables.

4.3 Saved image data

Every file saved by the library will have the same format, meaning that it is easy to read old files and get the data from them. There are three components to the file, a 1kb header containing the settings that were used to take the image, all the saved frames and then a series of values of central elements, steering angle and pulse shape for every line transmitted. The last component means that any other researcher can correctly interpret the images saved in the file, because they will know where and how each line of the image was taken.

The header contains thirteen integers, each of which requires one byte. They are:

```
(1)  frame size
(2)  number of frames
(3)  total number of lines
(4)  number of Doppler lines
(5)  frame rate
(6)  number of elements on the probe
(7)  whether a sectorial probe was used *
(8)  imaging mode
(9)  imaging sub mode
(10) sampling frequency
(11) transmit frequency
(12) focus distance
(13) acquisition depth
```

* Boolean, will be displayed as 1 for true or 0 for false.

The remainder of the 1kb header is filled with zeros. This ensures that if later versions of the library require more data to be stored in this header there will be room for the additional data. It is important that frame size and number of frames are the first two pieces of data, as they give information about the format of the frames so are required to decode the rest of the file.

The mode and sub mode values are decoded to give the operation mode of the scanner according to Table 1:

Mode	Sub mode	Operation
1 (basic RF)	1	Basic RF mode
1 (basic RF)	2	High speed sequence
2 (multi-pulse)	1	Inverting sequence
2 (multi-pulse)	2	Alternating sequence
3 (Doppler)	1	B-mode with Doppler repeat of centre
3 (Doppler)	2	Doppler
4 (custom)	1	Custom sequence

Table 1: Decoding the mode and sub mode numbers to give the scan sequence that was used to obtain the data.

After this will come the imaged frames and, after that, the details of each line. First is a series of integers which are the centre element for each line. There will be the same number as the value for number of lines in the header. Next will be a series of integers which are the steering angle for each line. Finally, there will be a series of pulse shapes. Because these do not all have to be the same length, each pulse shape will have a semicolon at the end to designate where one ends and the next begins. For example, a three-line pulse as described in Table 2 would have the following line definitions:

```
40 80 120 -100 0 100 +-;-+;++00--;
```

(There would be no spaces, these are just shown for clarity).

Line number	Centre element	Steering angle	Pulse shape
1	40	-100	+-
2	80	0	-+
3	120	100	++00--

Table 2: An example of a customised sequence composed of three lines.

5 CONTROLLING THE LIBRARY AND SCANNER FROM MATLAB

There are nine sample Matlab scripts in the `TexoProcesses` folder, the folder where all new Texo processes should be stored. These are designed to demonstrate how to write Matlab scripts which will use the TexoMatlab library to take images and process the data, and are also a useful starting point for writing new scripts to carry out new tasks. Four of them (examples 1, 2, 3 and 8) are discussed in detail here, the rest are briefly mentioned and should be readily understandable from these principles. There are five steps which will be required in every script: initialising the library, preparing the scanner for the image, defining the imaging sequence that will be used, running a scan to gather data on the target and shutting down the scanner and library. Then the data can be retrieved and processed in Matlab.

5.1 Standard sequences

The first example, `eg1_StandardSequence1`, is the simplest of the examples, and each of the steps outlined above is reproduced below and then discussed. As it is run in Matlab, the progress of the process will be displayed in the command window, as will any errors which are encountered.

```
% Initialise the library:
[saveFolder, configFolder, libraryFolder, settingsFolder] = openPaths();
initTexoMatlabLibrary(libraryFolder);
```

This first step initialises the library and these two lines must always be included unchanged. They create pointers to the various folders where the library will find files it will need, as well as to the files that Matlab will need to use to access the library functions.

```
try
    % Prepare the scanner:
    startEngine(settingsFolder, 40);
    configFile = [configFolder, 'DefaultConfigFile.txt'];
    loadConfigData(configFile);
```

The second step is preparing the scanner. The calls to `startEngine` (switching on the scanner) and `loadConfigData` (setting up the scan parameters) should be left unchanged for any Matlab script. The only thing which can be changed is the configuration file. If a different set of configurations are required a new file should be created with the changes and its file name should be added here.

```
    % Create a sequence:
    createStandardSequence();
```

The third step is defining the imaging sequence. This is the step where there is the most flexibility, as many different sequences can be used. Hence, the main difference in the following examples will be in this step. This example is the simplest way to use the scanner, a standard image with no pulse shape defined. The library will use the pulse shape that was given in the configuration file, and prepare a B-mode sequence for imaging.

```
    % Take the image and save the data:
    takeScan();
    saveFile = [saveFolder, 'StandardSequence1.bin'];
    saveData(saveFile, 1);
```

The forth step is to take the image and save the data. The file name for saving should obviously be changed for a new script so previous data is not overwritten.

```
    % Stop the engine and close the library:
    stopEngine();


catch
    disp('   - Error: Texo procedure failed.');
    disp('   - Attempting to stop engine and recover...');
    stopEngine();
    pause(2);
end
closeTexoMatlabLibrary;
```

The fifth step is stopping the scanner and shutting down the library. This should always be done as it has been here.

```
% Process the data:
[data, properties] = readSavedFile(saveFile);

figure;
imagesc(log(1+abs(hilbert(data'))));
colormap(gray);
```

Finally, the data can be retrieved and processed. The imaged frames are copied into `data` and the settings for the scan (the data header and the series of centre elements, angles and pulse shapes) are copied into `properties`. Looking at the properties should show that a B-mode image has indeed been taken – the centre elements will increase for each line while the steering angle is always 0, because a linear probe is specified in the configuration file. The figure will be a greyscale image where the brightness at any point shows the intensity of the reflection at that point.

In `eg2_StandardSequence2.m`, the scan is again done using a standard sequence. However, this example demonstrates more of the functionality of the library. The general principles are the same, so only the changes need be explained. The first changes relate to the preparation of the scanner. A different configuration file (`HalfDepthConfig.txt`) is called. In this case, the only difference is that the focus and acquisition depths are halved, but in general this is how a user would change the parameters for a scan to get better data in their particular application. Secondly, use has been made of the functions `returnScannerSettings` and `displayScannerSettings`. These two must always be used together, and sending the Boolean `true` to the first means that a complete list of settings is returned and so displayed. The reason it has been called twice is to demonstrate the effect of the next change, a call to `setPower` which will change the scanner's power values. In this example, the sequence is created with a specified pulse shape (+), and any required pulse shape could be passed in this way. That this has been successful can be verified by looking at `properties.pulseShape`.

There are several changes to the way the data is obtained, saved and processed. `takeScan` is now passed a value, so the duration of the scan will be 2500 ms rather than the 2000 ms specified in the configuration file. The data is stored in a different save file (this will be true for all the examples so they do not overwrite each other). More interestingly, the call to `saveData` is passed the value three for the number of frames to save. This means that when the save file is read, `data` will contain three frames rather than only one as before. This means that before a greyscale image of the target can be displayed, a single frame must be extracted from the image data. This is done by taking the first lines of `data`, up to and including the number of lines in a frame (`properties.nbLines`).

5.2 Two-pulsed sequences

`eg3_InvertingSequence` demonstrates all the additional commands that are needed to run two-pulsed sequences. It is very similar to `eg1`, with the primary difference being that `createInvertingSequence` is used for the sequence definition. This can be called with no arguments, in which case the pulse shapes will be `pulseShapeA` in the configuration file and its inverse. In this case, though, a pulse shape is passed as an argument of the function. The library will determine the inverse of this shape and transmit both pulses along each line. Looking at `properties.centerElement` and `properties.angle` should demonstrate that every line is transmitted twice, because the central elements and angles come in pairs. `properties.pulseShape` shows that these two transmissions use the given pulse shape and its inverse (`'+0-0+0-0'` and `'-0+0-0_0'`).

Because the pulse shape is inverting, a frame of the B-mode image will be comprised of alternate lines of `data`. So `frame` is extracted by taking the odd numbered lines and is then displayed. If a linear target is being imaged, the responses to the positive and negative pulses should be identical but opposite in sign. This can be verified by plotting the traces of the response from one line using each pulse shape. This is done in the second figure, using lines 127 and 128 from `data` as these will be near the centre if the probe has 128 elements and should be inverted. To check this, the negative of the second line is plotted so the red and black lines on the plot should overlap.

`eg4_AlternatingSequence` is much the same as `eg3`, but in the definition of an alternating sequence any two pulse shapes may be passed, they do not have to be inverses. If `createAlternatingSequence` is used with no arguments, the two pulse shapes will be `pulseShapeA` and `pulseShapeB` of the configuration file.

5.3 Doppler sequences

`eg5_DopplerSequence` and `eg6_MDopplerSequence`, the two Doppler examples, contain little that is new. In each example, the parameters which will be sent to the sequencing function have been specified by name for clarity, but it would be more usual to give these 'hello' values directly in the function call, for example: `createDopplerSequence('+-', 128, 3)`.

5.4 Custom sequences

Although the definition of a high speed sequence allows more parameters to be passed, so this sequence is more customisable than previous examples, the scripts to call it are almost identical to those for standard sequence (`eg1` and `eg2`). An example is given in `eg7_HighSpeedSequence`. That the parameters have been passed correctly can be confirmed by looking at the values displayed by `displayScannerSettings`, and the values in the `properties` structure.

The fully customised sequence requires more control from Matlab, but obviously gives more flexibility in the sequences it makes possible. In the case of `eg8_CustomSequence` there are only nine lines in the sequence, but it would be straightforward to use this procedure to add many more lines. In this example, we change the power values and the focus and acquisition depth before the sequence definition is begun – these parameters should not be changed during a sequence definition. `setLibraryVariable` is a Matlab function which is passed the name of the parameter to change (`frequency`, `focus`, `depth`, `angle`, `centerElement`, `txCenterElement`, `rxCenterElement` or `pulseShape`) and then the new value. It then calls the corresponding function in the library to change the value of that parameter (see section 2.2). The two calls to `returnScannerSettings` and `DisplayScannerSettings` should verify that these changes have been effective.

The sequence must be defined between `beginSequence` and `endSequence`. Between these points, any number of lines may be added using `addLine`, and scan parameters can be changed between these lines. In this example, there are three positions imaged (so the `for` loop runs three times), and each position has each of three pulse shapes transmitted. To achieve this, the central element and steering angle are changed once per loop, while the pulse shape is changed before every call to `addLine`. To see the effect of these – or any other – changes to the line position or pulse shape, the user can look at the relevant values in the `properties` variable to see how they change with each line.

Once the sequence definition is completed, the scan is executed as normal, the scanner shut down and the library closed. Any required processing can be applied to the data which is loaded into Matlab using the `readSavedFile` function.

5.5 Multiple sequences

The final example, `eg9_AllSequences`, demonstrates that several images can be taken using one Matlab script, and each can have a different sequence or parameters. This example calls each of the pre-defined sequences twice, once with a transmission frequency of 2MHz and once with a transmission frequency of 9.5MHz. The data from each scan is saved to a different file, the data is loaded into Matlab with a different variable name and greyscale frames are plotted so that the user can see in real-time whether they are getting the results they expect. For speed, these are displayed during the scanning process rather than waiting until the scanner and library have been closed down to process and display all of the data.