



Quantum Wave ToolBox documentation

BETA version

Q-Wave consortium
October 12, 2015

contact: msira@cmi.cz

Contents

1	Basic description of the toolbox	1
1.1	Toolbox overall scheme	1
1.2	Toolbox use	2
1.2.1	Get list of implemented algorithms	2
1.2.2	Application of an algorithm on the data	2
1.2.3	Running an example of algorithm use	3
1.2.4	Running a test of algorithm	3
1.2.5	Adding or removing algorithm path	3
2	Detailed description of the toolbox	5
2.1	Algorithm directory structure implementation	5
2.1.1	File alg_info.m	5
2.1.2	File alg_wrapper.m	6
2.1.3	File alg_test.m	6
2.1.4	File alg_example.m	6
2.2	Algorithm informations structure	7
2.2.1	.id	7
2.2.2	.longname	7
2.2.3	.desc	7
2.2.4	.citation	7
2.2.5	.remarks	8
2.2.6	.license	8
2.2.7	.requires	8
2.2.8	.reqdesc	8
2.2.9	.returns	8
2.2.10	.retdesc	8
2.2.11	.providesGUF	8
2.2.12	.providesMCM	8
2.2.13	.fullpath	8
2.3	Quantity structure	9
2.3.1	.v	9

2.3.2	.u	9
2.3.3	.d	9
2.3.4	.c	9
2.3.5	.r	9
2.3.6	Quantity structure examples	10
2.4	Calculation settings structure	11
2.4.1	.strict	12
2.4.2	.verbose	12
2.4.3	.unc	12
2.4.4	.cor	12
2.4.5	.dof	13
2.4.6	.mcm	13
2.5	How uncertainty calculation works	14
2.6	How to add a new algorithm	14
3	Bibliography	15
A	Quick reference	16
B	INL – Integral Non-Linearity of ADC	18
C	PSFE – Phase Sensitive Frequency Estimator	23

Abstract

*Press a button with bold title AMPLITUDE
...drink a coffee ...
and get the result*

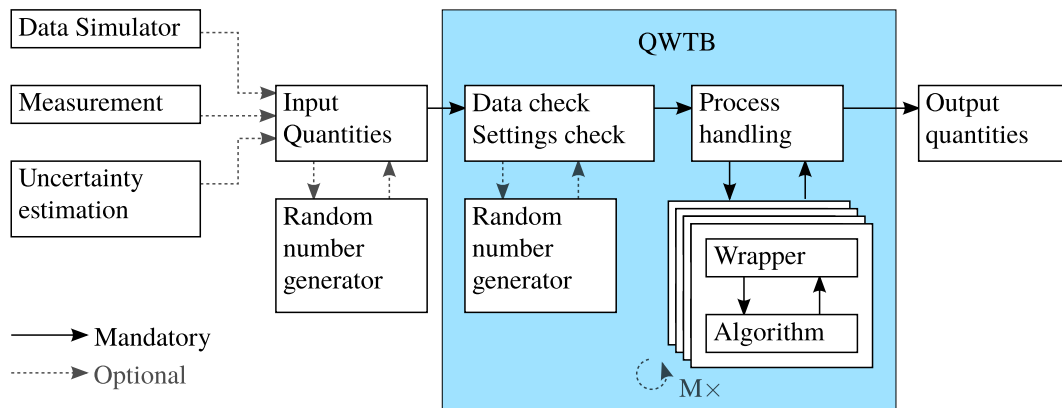
Quantum Wave Tool Box (QWTB) is a toolbox for evaluation of measured data. QWTB consist of data processing algorithms from very different sources and unifying application interface. The toolbox gives the possibility to use different data processing algorithms with one set of data and removes the need to reformat data for every particular algorithm. Toolbox is extensible. The toolbox can variate input data and calculate uncertainties by means of Monte Carlo Method (MCM) [1].

Chapter 1

Basic description of the toolbox

1.1 Toolbox overall scheme

The basic scheme of the toolbox is following:



User have to prepare the data, either based on a real measurement or simulated, into a specified format. If needed, user can generate randomized data for selected quantities (e.g. with special probability density functions) and prepare for Monte Carlo uncertainty calculation. Next user calls toolbox to apply a selected algorithm on the data and review results. Toolbox will:

1. Check user data.
2. Check or generate calculation settings.
3. If required, quantities are randomized according uncertainties to prepare for MCM uncertainty calculation.

4. Data are handled to a wrapper. If needed, wrapper is run multiple times according MCM.
5. Output data are the result of the toolbox.

Another algorithm can be used immediately on the same data. User interface of the toolbox is represented by the function `qwtb` defined in the file `qwtb.m`.

1.2 Toolbox use

The toolbox is used in several modes according to a number and character of input arguments.

1.2.1 Get list of implemented algorithms

```
alginfo = qwtb()
```

With no input arguments, toolbox returns informations on all available algorithms. Result array `alginfo` contains structures for every algorithm found in the same directory as `qwtb.m`. Format of structures is defined in 2.2.

1.2.2 Application of an algorithm on the data

```
dataout = qwtb('algid', datain)
```

The algorithm is selected by first input argument `algid`. It is a string with designator of the algorithm, according structrue 2.2.

The second input argument is the user data. Data have to be formatted in a structure with fields named as quantities required by the algorithm (see 2.3).

The output variable is the structure with fields named as quantities.

In this case, standard calculation settings are used. If the user specifies calculation settings in structure according 2.4, it can be used as third input argument `calcset`:

```
dataout = qwtb('algid', datain, calcset)
```

For some calculation settings some fields of `datain` or `calcset` are generated automatically. To review automatically generated fields, user can get these structure in second and third output argument:

```
[dataout, datain, calcset] = qwtb('algid', datain)
[dataout, datain, calcset] = qwtb('algid', datain,
    calcset)
```

1.2.3 Running an example of algorithm use

Algorithm can have implemented an example of the use. This can be run by following syntax:

```
[] = qwtb('algid', 'example')
```

The algorithm is selected by first input argument `algid`. It is a string with designator of the algorithm, according structrue 2.2. The second argument is a string. Toolbox will run a script `alg_example.m` located in a algorithm directory.

After finish user can review input and output data or resulted figures if any.

1.2.4 Running a test of algorithm

Algorithm can have implemented a self test. This can be run by following syntax:

```
[] = qwtb('algid', 'test')
```

The algorithm is selected by first input argument `algid`. It is a string with designator of the algorithm, according structrue 2.2. The second argument is a string. Toolbox will run a script `alg_test.m` located in a algorithm directory.

Test should prepare data, run algorithm and check results. If implementation of algorithm behaves incorrectly, an error will occur.

1.2.5 Adding or removing algorithm path

Algorithms are stored in different directories, which are not in MATLAB/GNU OCTAVE load path. To add directory with selected path to MATLAB/GNU OCTAVE load path, following syntax is used:

```
[] = qwtb('algid', 'addpath')
```

To remove path, use:

```
[ ] = qwtb( 'algid' , 'rempath' )
```

Adding or removing path should be required only in special cases, such as debugging etc.

Chapter 2

Detailed description of the toolbox

2.1 Algorithm directory structure implementation

Every algorithm is placed in a directory of following name:

`alg_X`

These directories have to be located in the directory containing the toolbox main script `qwtb.m`.

Every algorithm directory contains following files:

`X1, X2, ...` — Mandatory. One or more files with the algorithm itself.

`alg_info.m` — Mandatory. Description of the algorithm. See 2.1.1.

`alg_wrapper.m` — Mandatory. Wrapper of the algorithm. See 2.1.2.

`alg_test.m` — Recommended. Testing function. See 2.1.3.

`alg_example.m` — Recommended. Example script. See 2.1.4.

2.1.1 File `alg_info.m`

File contains a function with definition:

```
function alginfo = alg_info()
```

The output `alginfo` is a structure with informations about the algorithm. Structure is defined in 2.2.

File is mandatory. If file is missing in algorithm directory, QWTB will not recognize this algorithm as part of the toolbox.

2.1.2 File `alg_wrapper.m`

File contains a function with definition:

```
function dataout = alg_wrapper(datain , calcset)
```

The input `datain` is a structure with input data (see), `calcset` is a structure with definition of calculation settings (see 2.4).) and `dataout` is a structure containing output data (see).

The wrapper does following:

1. Formats input data structure `datain` into variables suitable for algorithm.
2. Runs the algorithm.
3. Format results of the algorithm into data structure `dataout`.

File is mandatory. If file is missing in algorithm directory, QWTB will not recognize this algorithm as part of the toolbox.

2.1.3 File `alg_test.m`

File contains a function with following definition:

```
function [] = alg_test(calcset)
```

Test should generate sample data, run algorithm and check results by a function `assert`. QWTB will provide a standard calculation settings structure `calcset` (see 2.4).

This file is not mandatory, however is recommended.

2.1.4 File `alg_example.m`

Example contains a script showing a basic use of the algorithm. The format of the file should conform to the publishing markup defined in Matlab documentation. See matlab help on keyword *Publishing markup*). The QWTB runs this script in base context, thus all variables defined in the example script will be accessible to the user.

To create a documentation of the QWTB, function `publish` is applied to the example script and resulting file is attached to the documentation file.

2.2 Algorithm informations structure

Structure defines properties and possibilities of the algorithm. All fields are mandatory but `.fullpath`.

- `.id` — Designator of the algorithm.
- `.name` — Name of the algorithm.
- `.desc` — Basic description.
- `.citation` — Reference.
- `.remarks` — Any remark.
- `.license` — License of the algorithm.
- `.requires` — Required quantities.
- `.reqdesc` — Short description of required quantities.
- `.returns` — Output quantities.
- `.retdesc` — Short description of output quantities.
- `.providesGUF` — Algorithm/wrapper calculates GUF uncertainty.
- `.providesMCM` — Algorithm/wrapper calculates MCM uncertainty.
- `.fullpath` — Full path to the algorithm. Automatically generated by the toolbox.

2.2.1 `.id`

String. Designator of the algorithm. It is unique identifier, no two algorithms can have same id.

2.2.2 `.longname`

String. Full name of the algorithm.

2.2.3 `.desc`

String. Basic description of the algorithm.

2.2.4 `.citation`

String. A reference to the paper, book or other literature with full description of the algorithm.

2.2.5 .remarks

String. Remarks, license or others related to the algorithm.

2.2.6 .license

String. License of the algorithm. This is not license of the toolbox!

2.2.7 .requires

Cell array of strings. Names of quantities required by the algorithm.

2.2.8 .reqdesc

Cell array of strings. Short description of quantities required by the algorithm.

2.2.9 .returns

Cell array of strings. Names of quantities returned by the algorithm.

2.2.10 .retdesc

Cell array of strings. Short description of quantities returned by the algorithm.

2.2.11 .providesGUF

Boolean. If nonzero, the wrapper or the algorithm calculates uncertainty by means of GUM Uncertainty Framework.

2.2.12 .providesMCM

Boolean. If nonzero, the wrapper or the algorithm calculates uncertainty by means of Monte Carlo Method.

2.2.13 .fullpath

String. Full path to the algorithm. This field is automatically generated by QWTB.

2.3 Quantity structure

Every quantity is a structure with following fields:

- .v — Value.
- .u — Uncertainty.
- .d — Degree of freedom.
- .c — Correlation.
- .r — Randomized uncertainty.

2.3.1 .v

Value of the quantity. Can be a scalar, *row* vector or matrix. More dimensions are not supported.

2.3.2 .u

Standard uncertainty of the quantity. Dimensions are the same as of the value field.

2.3.3 .d

Degrees of freedom the uncertainty according GUM Uncertainty Framework. Dimensions are the same as of the value field.

This field is automatically generated by the toolbox if missing, required and `calcset.dof.gen` is set to nonzero. The value will be set to 50.

2.3.4 .c

Correlation matrix for quantity. 2DO XXX.

This field can be automatically generated by the toolbox if missing, required and `calcset.cor.gen` is set to nonzero. The value will be set to 0.

2.3.5 .r

Randomized uncertainties according Monte Carlo method. In the case of scalar quantity it is *column* vector of length equal to `calcset.mcm.repeats`. For a vector quantity it is a matrix with number of columns equal to length of value of the quantity and number of rows equal to `calcset.mcm.repeats`. For a matrix quantity it is a matrix with three dimensions, first two equal to the dimensions of value quantity, third dimension equal to `calcset.mcm.repeats`.

This field is required if Monte Carlo uncertainty calculation is required. In this case it can be automatically generated by the toolbox if missing and `calcset .mcm .randomize` is set to boolean. The pdf will be normal, sigma will be equal to the standard uncertainty of the quantity.

2.3.6 Quantity structure examples

Example of scalar quantity of mean value 1, standard uncertainty 0.1, degrees of freedom 9, correlation has no sense for scalar quantity, and randomized matrix has number of elements equal to `calcset .mcm.randomize`.

$$\begin{aligned} .v: & (1) \\ .u: & (0.1) \\ .d: & (9) \\ .c: & (0) \\ .r: & \begin{pmatrix} 1.02076 \\ 1.22555 \\ \vdots \\ 0.89727 \end{pmatrix} \end{aligned}$$

Example of vector quantity with i elements, M is equal to `calcset .mcm.randomize` (only symbolic representation):

$$\begin{aligned} .v: & (v_1, v_2, \dots, v_i) \\ .u: & (u_1, u_2, \dots, u_i) \\ .d: & (d_1, d_2, \dots, d_i) \\ .c: & \begin{pmatrix} c_{11} & \dots & c_{1i} \\ \vdots & \ddots & \vdots \\ c_{i1} & \dots & c_{ii} \end{pmatrix} \\ .r: & \begin{pmatrix} r_{11} & \dots & r_{1i} \\ \vdots & \ddots & \vdots \\ r_{M1} & \dots & r_{Mi} \end{pmatrix} \end{aligned}$$

Example of matrix quantity with i times j elements, M is equal to `calcset .`

mcm.randomize (only symbolic representation):

$$\begin{aligned}
 .v: & \begin{pmatrix} v_{11} & \dots & v_{1j} \\ \vdots & \ddots & \vdots \\ v_{i1} & \dots & v_{ij} \end{pmatrix} \\
 .u: & \begin{pmatrix} v_{11} & \dots & u_{1j} \\ \vdots & \ddots & \vdots \\ u_{i1} & \dots & u_{ij} \end{pmatrix} \\
 .d: & \begin{pmatrix} d_{11} & \dots & d_{1j} \\ \vdots & \ddots & \vdots \\ d_{i1} & \dots & d_{ij} \end{pmatrix} \\
 .c: & (XXX???) \\
 .r: & \begin{pmatrix} r_{111} & \dots & r_{1j1} \\ \vdots & \ddots & \vdots \\ r_{i11} & \dots & r_{ij1} \end{pmatrix} \\
 & \vdots \\
 & \begin{pmatrix} r_{11M} & \dots & r_{1jM} \\ \vdots & \ddots & \vdots \\ r_{i1M} & \dots & r_{ijM} \end{pmatrix}
 \end{aligned}$$

2.4 Calculation settings structure

Structure defines calculation methods.

- .strict — (0) If zero, other fields generated automatically.
- .verbose — (1) Display various informations.
- .unc — ('none') How uncertainty is calculated ('none', 'guf', 'mcm').
- .cor.req — (0) Correlation matrix is required for all input quantities.
- .cor.gen — (1) Zero correlation matrix is generated automatically if missing.
- .dof.req — (1) Degrees of freedom are required for all input quantities.
- .dof.gen — (1) Degree of freedom are generated automatically if missing with value 50.
- .mcm.repeats — (100) Number of Monte Carlo iterations.
- .mcm.verbose — (1) Display various informations concerning Monte Carlo method.

`.mcm.method` — ('singlecore') Parallelization method ('multicore', 'multistation').
`.mcm.procno` — (1) Number of processors to use.
`.mcm.tmpdir` — ('.') Directory for temporary data.
`.mcm.randomize` — (1) Randomized uncertainties are generated automatically if missing.

2.4.1 `.strict`

Boolean, default value 0. If set to zero, all other fields of the structure are generated automatically and set to a default value.

2.4.2 `.verbose`

Boolean, default value 1. If set to non-zero value, various messages are displayed during calculation, such as used uncertainty calculation method, automatic generation of matrices etc.

2.4.3 `.unc`

String, default value "". Determines uncertainty calculation method. Only three values are possible:

"" — Uncertainty is not calculated.

'guf' — Uncertainty is calculated by GUM Uncertainty Framework [2].

'mcm' — Uncertainty is calculated by Monte Carlo Method [1].

See chapter XXX for uncertainty calculation details.

2.4.4 `.cor`

Structure sets handling of correlation matrices of quantities. Structure has two fields:

`.req` — Boolean, default value 0. If non-zero, correlation matrices are required for all quantities.

`.gen` — Boolean, default value 1. If non-zero, correlation matrices will be generated automatically if missing in quantity.

Automatically generated correlation matrices has all elements of zero value.

2.4.5 .dof

Structure sets handling of degrees of freedom of quantities. Structure has two fields:

- `.req` — Boolean, default value 0. If non-zero, degrees of freedom are required for all quantities.
- `.gen` — Boolean, default value 1. If non-zero, degree of freedom will be generated automatically if missing in quantity.

Automatically generated degree of freedom has value 50.

2.4.6 .mcm

Structure sets handling of Monte Carlo calculation of uncertainties. Structure has following fields:

- `.repeats` — Positive non-zero integer, default value 100. Number of iterations of Monte Carlo method.
- `.verbose` — Boolean, default value 1. If set to non-zero value, various messages are displayed during calculation of Monte Carlo method such as used parallelization method, number of calculated iterations etc.
- `.method` — String, default value 'singlecore'. Parallelization method used for Monte Carlo method calculation. Only three values are possible:
 - 'singlecore' — No parallelization, all is calculated on one CPU core.
 - 'multicore' — Calculation is divided into cores of one computer.
 - 'multistation' — Calculation is distributed on several computers.

Not all methods are possible to use on all computers. 'singlecore' is always possible to use. 'multicore' use `parfor` in Matlab or `parcellfun` in GNU Octave. 'multistation' use

- `.procno` — Zero or positive integer, default value 0. Number of CPU cores exploitable by the parallelization method 'multicore'. If set to zero, all available CPU cores will be used. If desktop computer is used, it is good practice to set to number of CPU cores minus one, so the computer can be used by other task also. Works only in GNU OCTAVE.
- `.tmpdir` — String, default value '.' (current directory). Temporary directory for storing temporary data needed for some parallelization methods.
- `.randomize` — Boolean, default value 1. If non-zero, randomized uncertainties will be generated automatically if missing, but only if uncertainty calculation method is set to 'mcm' (Monte Carlo) to prevent large memory usage.

2.5 How uncertainty calculation works

2.6 How to add a new algorithm

Chapter 3

Bibliography

- [1] JCGM, *Evaluation of measurement data - Supplement 1 to the “Guide to the expression of uncertainty in measurement” - Propagation of distributions using a Monte Carlo method*, JCGM, Ed. Bureau International des Poids et Mesures, 2008.
- [2] —, *Evaluation of measurement data - Guide to the expression of uncertainty in measurement*, JCGM, Ed. Bureau International des Poids et Mesures, 1995, ISBN: 92-67-10188-9.

Appendix A

Quick reference

Toolbox use:

```
alginfo = qwtb()  
dataout = qwtb('algid', datain)  
[dataout, datain, calcset] = qwtb('algid', datain)  
dataout = qwtb('algid', datain, calcset)  
[dataout, datain, calcset] = qwtb('algid', datain, calcset)  
[] = qwtb('algid', 'example')  
[] = qwtb('algid', 'test')  
[] = qwtb('algid', 'addpath')  
[] = qwtb('algid', 'rempath')
```

Algorithm informations structure:

.id — Designator of the algorithm.
.name — Name of the algorithm.
.desc — Basic description.
.citation — Reference.
.remarks — Any remark.
.license — License of the algorithm.
.requires — Required quantities.
.reqdesc — Description of required quantities.
.returns — Output quantities.
.retdesc — Description of output quantities.
.providesGUF — Algorithm/wrapper calculates GUF uncertainty.
.providesMCM — Algorithm/wrapper calculates MCM uncertainty.
.fullpath — Full path to the algorithm. Automatically generated by the toolbox.

Quantity structure:

.v — Value.
.u — Uncertainty.
.d — Degree of freedom.
.c — Correlation.
.r — Randomized uncertainty.

Calculation settings structure:

.strict — (0) If zero, other fields generated automatically.
.verbose — (1) Display various informations.
.unc — ('none') How uncertainty is calculated ('none', 'guf', 'mcm').
.cor.req — (0) Correlation matrix is required for all input quantities.
.cor.gen — (1) Zero correlation matrix is generated automatically if missing.
.dof.req — (1) Degrees of freedom are required for all input quantities.
.dof.gen — (1) Degree of freedom are generated automatically if missing with value 50.
.mcm.repeats — (100) Number of Monte Carlo iterations.
.mcm.verbose — (1) Display various informations concerning Monte Carlo method.
.mcm.method — ('singlecore') Parallelization method ('multicore', 'multistation').
.mcm.procno — (1) Number of processors to use.
.mcm.tmpdir — ('.') Directory for temporary data.
.mcm.randomize — (1) Randomized uncertainties are generated automatically if missing.

Appendix B

INL – Integral Non-Linearity of ADC

Info file data

.id — INL

.name — Integral Non-Linearity of ADC

.desc — Calculates Integral Non-Linearity of an ADC. ADC has to sample sinewave, ADC codes are required.

.citation — Virosztek, T., Pálfi V., Renczes B., Kollár I., Balogh L., Sárhegyi A., Márkus J., Bilau Z. T., ADCTest project site: <http://www.mit.bme.hu/projects/adctest> 2000-2014

.remarks — Based on the ADCTest Toolbox v4.3, November 25, 2014.

.license — UNKNOWN

.requires

 t — time series of sampled data

 codes — Sampled values represented as ADC codes (not converted to voltage)

.returns

 INL — INL

.providesGUF — no

.providesMCM — no

Example

Integral Non Linearity of ADC

Example for algorithm INL

INL is an algorithm for estimating Integral Non-Linearity of an ADC. ADC has to sample sinewave, ADC codes are required.

See also 'Virosztek, T., Pálfi V., Renczes B., Kollár I., Balogh L., Sárhegyi A., Márkus J., Bilau Z. T., ADCTest project site: <http://www.mit.bme.hu/projects/adctest> 2000-2014';

Contents

- Generate sample data
- Call algorithm

Generate sample data

Suppose a sine wave of nominal frequency 10 Hz and nominal amplitude 1 V is sampled by ADC with bit resolution of 4. First quantities `t` with time of samples and quantity `bits` with number of bits are prepared and put into input data structure `DI`.

```
DI = [];  
DI.t.v = [0:1/1e4:1-1/1e4];  
DI.bits.v = 4;
```

Waveform is constructed.

```
Anom = 1; fnom = 2; phnom = 0;  
wvfrm = Anom*sin(2*pi*fnom*DI.t.v + phnom);
```

Next code values are calculated. It is simulated by quantization and scaling of the sampled waveform. In real measurement code values can be obtained directly from the ADC. Suppose ADC range is -1..1.

```

codes = wvfrm;
rmin = -1; rmax = 1;
levels = 2.^DI.bits.v - 1;
codes(codes<rmin) = rmin;
codes(codes>rmax) = rmax;
codes = round((codes-rmin)./2.*levels);

```

Now lets introduce ADC error. Instead of generating code 2 ADC erroneously generates code 3 and instead of 10 it generates 11.

```

codes(codes==2) = 3;
codes(codes==10) = 11;
codes = codes + min(codes);

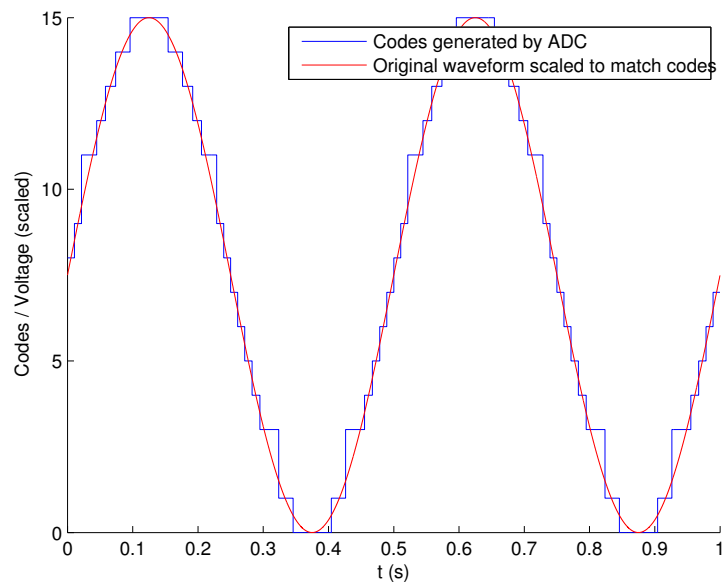
```

Create quantity codes and plot a figure with sampled sine wave and codes.

```

DI.codes.v = codes;
figure
%plot(t, (y+1).*1/2.*levels, t, codes);
hold on
stairs(DI.t.v, codes);
wvfrm = (wvfrm + Anom).*levels./2;
plot(DI.t.v, wvfrm, '-r');
xlabel('t (s)')
ylabel('Codes / Voltage (scaled)');
legend('Codes generated by ADC', 'Original waveform
scaled to match codes');
hold off

```

Call algorithm

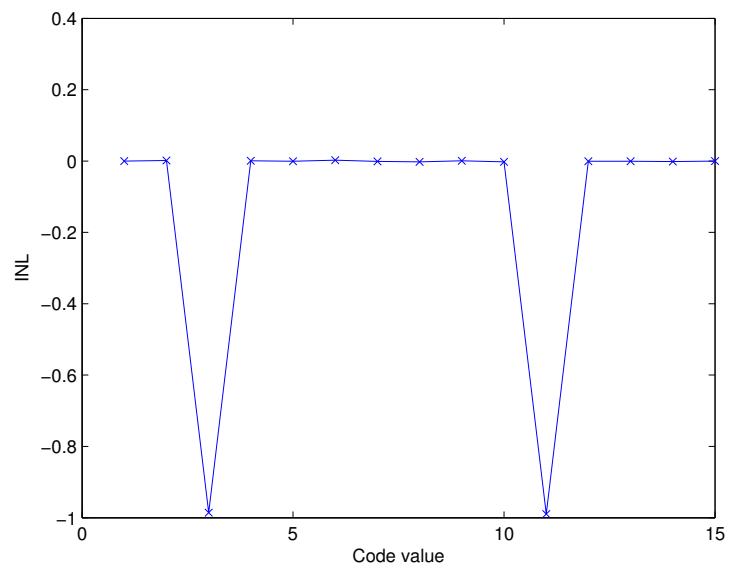
Apply INL algorithm to the input data DI.

```
DO = qwtb('INL', DI);
```

QWTB: no uncertainty calculation

Plot results of integral non-linearity. One can clearly observe defects on codes 3 and 11.

```
figure
plot(DO.INL.v, '-x');
xlabel('Code value')
ylabel('INL')
```



Appendix C

PSFE – Phase Sensitive Frequency Estimator

Info file data

.id — PSFE

.name — Phase Sensitive Frequency Estimator

.desc — An algorithm for estimating the frequency, amplitude, and phase of the fundamental component in harmonically distorted waveforms. The algorithm minimizes the phase difference between the sine model and the sampled waveform by effectively minimizing the influence of the harmonic components. It uses a three-parameter sine-fitting algorithm for all phase calculations. The resulting estimates show up to two orders of magnitude smaller sensitivity to harmonic distortions than the results of the four-parameter sine fitting algorithm.

.citation — Lapuh, R., "Estimating the Fundamental Component of Harmonically Distorted Signals From Noncoherently Sampled Data," Instrumentation and Measurement, IEEE Transactions on , vol.64, no.6, pp.1419,1424, June 2015, doi: 10.1109/TIM.2015.2401211, URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7061456&isnumber=7104190>

.remarks — Very small errors, effective for harmonically distorted signals.

.license — UNKNOWN

.requires

 t — time series of sampled data

 y — sampled values

.returns

f — Frequency of main signal component

A — Amplitude of main signal component

ph — Phase of main signal component

.providesGUF — no

.providesMCM — no

Example

Phase Sensitive Frequency Estimator

Example for algorithm PSFE.

PSFE is an algorithm for estimating the frequency, amplitude, and phase of the fundamental component in harmonically distorted waveforms. The algorithm minimizes the phase difference between the sine model and the sampled waveform by effectively minimizing the influence of the harmonic components. It uses a three-parameter sine-fitting algorithm for all phase calculations. The resulting estimates show up to two orders of magnitude smaller sensitivity to harmonic distortions than the results of the four-parameter sine fitting algorithm.

See also Lapuh, R., "Estimating the Fundamental Component of Harmonically Distorted Signals From Noncoherently Sampled Data," Instrumentation and Measurement, IEEE Transactions on , vol.64, no.6, pp.1419,1424, June 2015, doi: 10.1109/TIM.2015.2401211, URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7061456&isnumber=7104190>

Contents

- Generate sample data
- Call algorithm
- Display results

Generate sample data

Two quantities are prepared: t and y, representing 1 second of sinus waveform of nominal frequency 100 Hz, nominal amplitude 1 V and nominal phase 1 rad, sampled at sampling frequency 10 kHz.

```

DI = [];
Anom = 1; fnom = 100; phnom = 1;
DI.t.v = [0:1/1e4:1-1/1e4];
DI.y.v = Anom*sin(2*pi*fnom*DI.t.v + phnom);

```

Add noise:

```

DI.y.v = DI.y.v + normrnd(0, 1e-3, size(DI.y.v));

```

Call algorithm

Use QWTB to apply algorithm PSFE to data DI.

```

DO = qwtb('PSFE', DI);

```

QWTB: no uncertainty calculation

Display results

Results is the amplitude, frequency and phase of sampled waveform.

```

f = DO.f.v
A = DO.A.v
ph = DO.ph.v

```

```

f =
    100.0000

A =
    1.0000

```

ph =

1.0000

Errors of estimation in parts per milion:

```
ferrppm = (DO.f.v - fnom)/fnom .* 1e6  
Aerrppm = (DO.A.v - Anom)/Anom .* 1e6  
pherrppm = (DO.ph.v - phnom)/phnom .* 1e6
```

ferrppm =

0.1113

Aerrppm =

4.3774

pherrppm =

-13.7952