QWTB documentation

Toolbox description

2024-06-26

https://qwtb.github.io/qwtb/

Contents

1

Introduction

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

QWTB is a toolbox for evaluation of measured data. QWTB consist of data processing algorithms from very different sources and unificating 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].

QWTBLVLib is a set of LabVIEW Virtual Instruments (VI) forming a library and providing an easy link between LabVIEW and QWTB.

Simple QWTB GUI is a fully working example of using QWTBLVLib written in LabVIEW and forms a graphical user interface to QWTB.

QWTBvar is a script to variate input quantities, multiply run algorithm inside of QWTB, and deliver or plot results. It can be also used to precalculate results and interpolate.

Toolbox was realized within the EMRP-Project SIB59 Q-Wave. The EMRP is jointly funded by the EMRP par- ticipating countries within EURAMET and the European Union.

2

Installation

2.1 QWTB & QWTBvar

The toolbox can be downloaded either as a GIT repository or as a zip archive containing documentation and a `qwtb` directory containing all scripts and algorithms. Extract archive into a directory of your selection `YourDirectory`.

Start MATLAB or GNU OCTAVE. To use the toolbox, two methods can be used:

1. Change current working directory of MATLAB or GNU OCTAVE by command:

```
cd('YourDirectory/qwtb')
```

2. Or add toolbox directory into the search path by command:

```
addpath('YourDirectory/qwtb')
```

2.2 QWTBLVLib

Library can be downloaded either as source codes in a GIT repository or as a zip archive containing packed project library with `.lvlibp` and `.dll` libraries. Extract archive into a directory of your selection and import into your LABVIEW project.

2.3 simple QWTB GUI

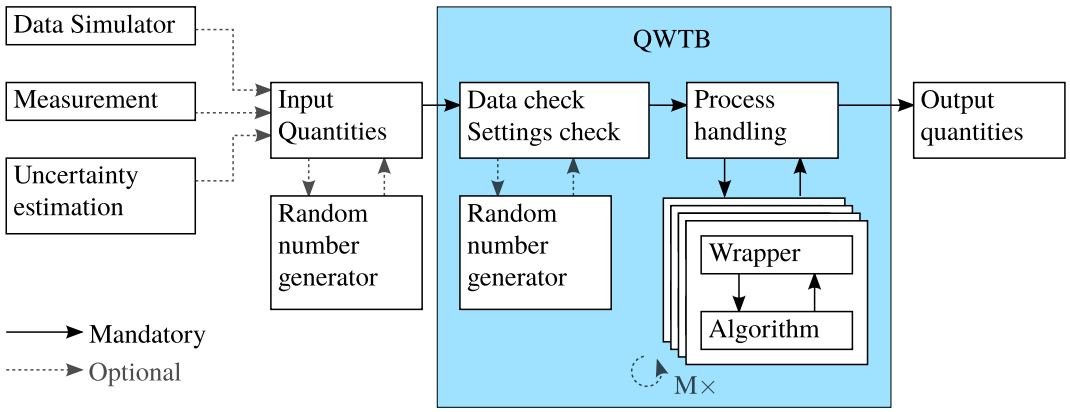
Simple graphical user interace can be downloaded either as source codes in a GIT repository or as a zip archive containing executable and required libraries. Extract archive into a directory of your selection and run the executable file.

3

Basic description of the toolbox

3.1 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 to 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`.

3.2 Toolbox use

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

3.2.1 Get informations of all implemented algorithms

```
alginfo = qwtb()
```

With no input arguments, toolbox returns informations of 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 4.2.

This call can be also used to get *standard* calculation settings used in the toolbox:

```
[alginfo, calcset] = qwtb()
```

For use of calcset, see next chapter.

3.2.2 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 4.2.

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 4.3).

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 4.4, 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)
```

3.2.3 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 4.2. 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.

3.2.4 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 4.2. 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.

3.2.5 Get informations of algorithm

To get informations of only the selected algorithm, following syntax is used:

```
qwtb('algid', 'info')
```

Result structure is defined in 4.2.

3.2.6 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.

3.2.7 Displaying license of an algorithm

To display a license of an algorithm, following syntax is used:

```
license = qwtb('algid', 'license')
```

For details on licensing, see chapter 6.

4

Detailed description of the toolbox

4.1 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 4.1.1.

`alg_wrapper.m` — Mandatory. Wrapper of the algorithm. See 4.1.2.

`alg_test.m` — Recomended. Testing function. See 4.1.3.

`alg_example.m` — Recomended. Example script. See 4.1.4.

4.1.1 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 4.2.

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

4.1.2 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 4.4).) and `dataout` is a structure containing output data (see).

The wrapper does following:

1. Formats input data structure `datain` into variables wuitable 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.

4.1.3 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 4.4), which is used as a function input variable.

This file is not mandatory, however is recommended.

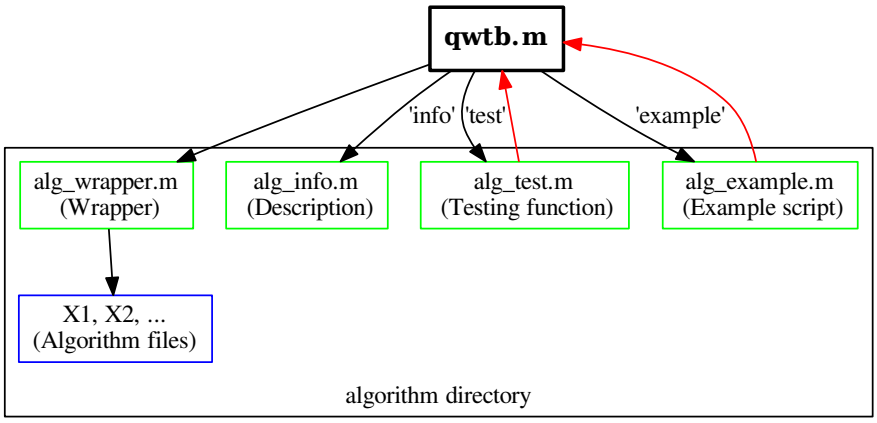
4.1.4 File `alg_example.m`

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.

4.1.5 Overall flow chart

The main toolbox file `qwtb.m` calls files in algorithm directory according following flow chart:



Red arrow marks recursion, blue box represents files of the algorithm itself, green boxes represents files required or recommended by the toolbox.

4.2 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.

.inputs — Input quantities definitions.

.outputs — Output quantities definitions.

.providesGUF — Algorithm/wrapper calculates GUF uncertainty.

.providesMCM — Algorithm/wrapper calculates MCM uncertainty.

.fullpath — Full path to the algorithm. Automatically generated by the toolbox.

4.2.1 .id

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

4.2.2 .name

String. Full name of the algorithm.

4.2.3 .desc

String. Basic description of the algorithm.

4.2.4 .citation

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

4.2.5 .remarks

String. Remarks or others related to the algorithm.

4.2.6 .license

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

4.2.7 .inputs

Array of structures. Every structure define an input quantity. Structure has following mandatory fields:

.name — Name of input quantity.

.desc — Description of input quantity.

.alternative — Index of group of alternative quantities.

.optional — Sets quantity to be optional.

.parameter — Sets quantity to be a parameter.

.name

String. Name of input quantity.

.desc

String. Short description of input quantity.

.alternative

Integer. Index of group of alternative quantities. If several input quantities has the same index, QWTB requires only one of quantities in the group. If set to zero, quantity is not part of any group.

For example, suppose a sampling time `Ts` and sampling frequency `fs` has the same index. The algorithm requires `fs`, but the wrapper can calculate `fs` from `Ts`. User can supply only `fs` to run the algorithm. But he can also supply only the `Ts` and the wrapper calculates `fs` from `Ts` and runs the algorithm. The wrapper should always notice the user that some quantity was calculated from other one. If user supplies both, the wrapper should choose the quantity most suitable for the algorithm.

.optional

Boolean. If set, the quantity is optional. The QWTB do not require user to supply this quantity. If quantities are part of a group of alternative quantities, the group itself is considered optional only if all quantities are optional.

.parameter

Boolean. If set, the quantity is considered as optional. This means the quantity do not have to be a number and is not randomized by QWTB.

4.2.8 .outputs

Array of structures. Every structure define an output quantity. Structure has following mandatory fields:

.name — Name of output quantity.

.desc — Description of output quantity.

.name

String. Name of output quantity.

.desc

String. Short description of output quantity.

4.2.9 .providesGUF

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

4.2.10 .providesMCM

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

4.2.11 .fullpath

String. Full path to the algorithm. This field is automatically generated by QWTB and should not be part of `alg_test.m`.

4.3 Quantity structure

Every quantity is a structure with following fields:

.v — Value.

.u — Uncertainty.

.d — Degree of freedom.

.c — Correlation.

.r — Randomized uncertainty.

4.3.1 .v

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

Vectors should be ordered in a single *row*. If the vector is ordered in a collumn, it is automatically transposed and a warning is generated. Other fields (.u, .d, .r) are transposed if needed to match the value of the quantity.

4.3.2 .u

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

4.3.3 .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.

4.3.4 .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.

4.3.5 .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.

4.3.6 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 radnomized matrix has number of elements equal to `calcset.mcm.randomize`.

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

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

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

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

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

4.4 Calculation settings structure

Structure defines calculation methods.

.strict — (0) If zero, other fields generated automatically.

.verbose — (1) Display various informations.

.checkinputs — (0) Check if inputs are proper.

.unc — ('none') How uncertainty is calculated ('none', 'guf', 'mcm').

.loc — (0.6827) Required level of confidence of output uncertainties ($0 > \text{loc} > 1$).

.cor.req — (0) Correlation matrix is required for all input quantities.

.cor.gen — (1) Zero correlation matrix is generated automatically if missing.

.dof.req — (0) 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 — (0) Number of processors to use.

.mcm.tmpdir — ('.') Directory for temporary data.

.mcm.randomize — (1) Randomized uncertainties are generated automatically if missing.

.var.dir — ('VAR') Only for QWTBvar: Directory for calculations.

.var.fnprefix — ('var') Only for QWTBvar: Prefix for filenames.

.var.cleanfiles — ('0') Only for QWTBvar: Removes files from previous calculations.

.var.smalloutput — ('1') Only for QWTBvar: Minimize size of output files.

.var.method — ('singlecore') Only for QWTBvar: Parallelization method ('multicore', 'multistation').

.var.procno — (1) Only for QWTBvar: Number of processors to use.

.var.chunks_per_proc — (1) Only for QWTBvar: Number of calculation jobs per process.

4.4.1 .strict

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

4.4.2 .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.

4.4.3 .checkinputs

Boolean, default value 1. If set to zero, all checks of inputs (calculation settings, input data) are skipped. However if inputs are not proper, various undocumented errors can happen. This option can be used to speed up qwtb processing. About 13 ms of processing time can be saved for algorithm with 30 input quantities (measured at iCore 5, GNU Octave 4.2.2). User have to be sure all inputs conform proper format and contain all needed data.

4.4.4 .unc

String, default value 'none'. Determines uncertainty calculation method. Only three values are possible:

'none' — 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.

4.4.5 .loc

Number, default value 0.6827. Determines required level of confidence for output uncertainties. If set to 0.6827, uncertainties are calculated for coverage factor 1 (for the case of normal probability density function). Value 0.9545 correspond to a coverage factor 2. Value of loc is respected if uncertainty is calculated by the toolbox. If the uncertainty is calculated by the algorithm itself, one should check capabilities of the algorithm. Currently only standard uncertainties (68.27%) are supported.

4.4.6 .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.

4.4.7 .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.

4.4.8 .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.

4.4.9 .var

Structure sets properties only for QWTBvar. Structure has following fields:

.dir — String, default value 'VAR'. All files needed for QWTBvar calculations will be saved into the selected directory. If path does not exist, it is created.

.fnprefix — String, default value 'var'. All file names needed for QWTBvar calculations will start with this prefix.

.cleanfiles — Boolean, default value 0. Job files and results from previous QWTBvar calculation will be deleted. Do not use if continuation of previously interrupted calculation is required.

.smalloutput — Boolean, default value 1. If set, QWTBvar will not save all results of the calculations. Fields `.c` and `.r` of the input and output quantities will be removed to minimize size of files with results. Check algorithms if these fields are needed.

.method — String, default value 'singlecore'. Parallelization method used for QWTBvar calculations. 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 Be carefull when setting this parameter together with

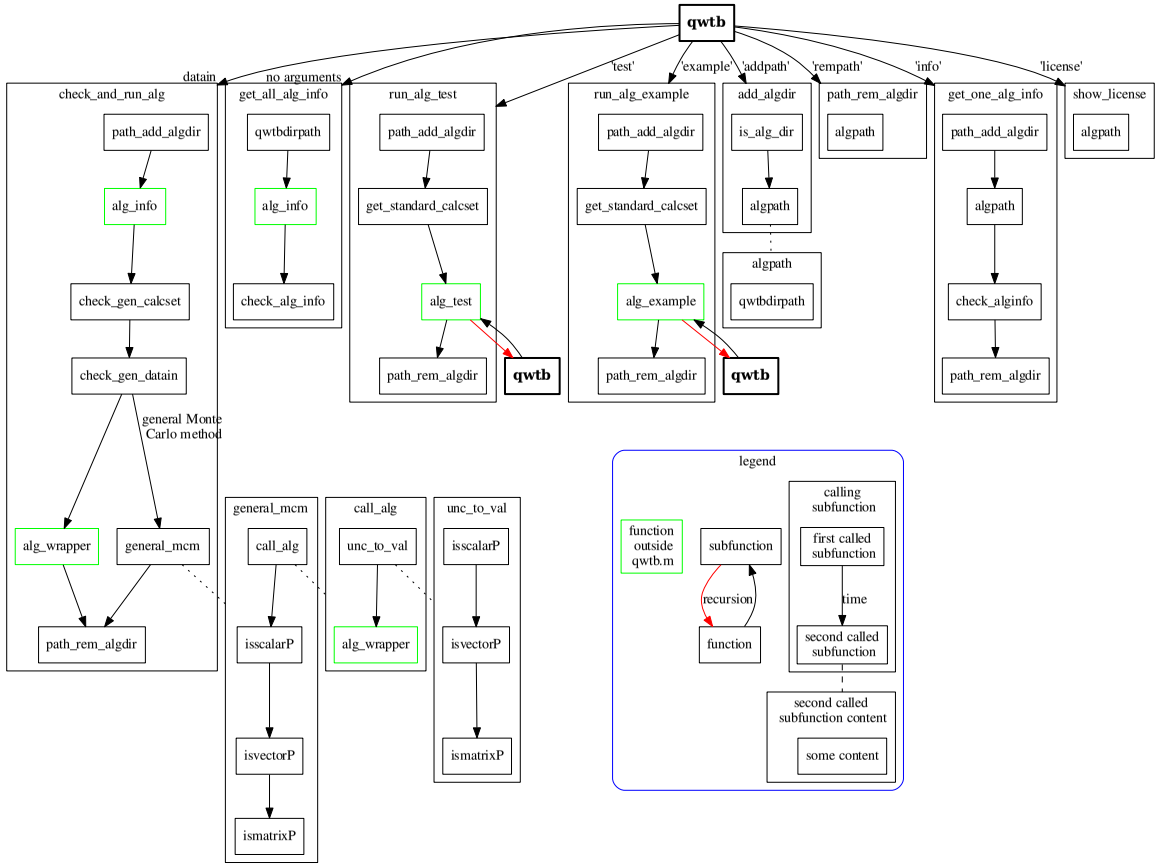
Monte Carlo calculations. Bad settings can result in very large number of processes.

.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.

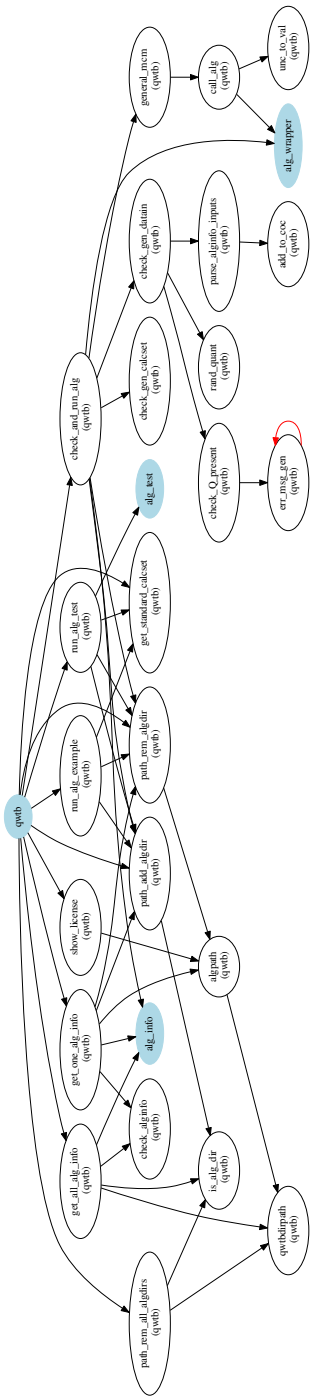
.chunks_per_proc — Positive non-zero integer, default value 1. Specifies number of calculations proceeded by one process. Higher number decreases number of job files on the disk and usage of the disk by the system.

4.5 qwtb.m flow chart

Following figure shows flow chart inside of `qwtb.m` file with subfunctions.


qwtb
datain
no arguments
'test'
'example'
'addpath'
'rempath'
'info'
'license'
check_and_run_alg
path_add_algdir
alg_info
check_gen_calcset
check_gen_datain
general Monte Carlo method
alg_wrapper
general_mcm
path_rem_algdir
get_all_alg_info
qwtbdirpath
alg_info
check_alg_info
run_alg_test
path_add_algdir
get_standard_calcset
alg_test
path_rem_algdir
qwtb
run_alg_example
path_add_algdir
get_standard_calcset
alg_example
path_rem_algdir
qwtb
add_algdir
is_alg_dir
algpath
algpath
qwtbdirpath
path_rem_algdir
algpath
get_one_alg_info
path_add_algdir
algpath
check_alginfo
path_rem_algdir
show_license
algpath
general_mcm
call_alg
isscalarP
isvectorP
ismatrixP
call_alg
unc_to_val
alg_wrapper
unc_to_val
isscalarP
isvectorP
ismatrixP
legend
function outside qwtb.m
subfunction
recursion
function
calling subfunction
first called subfunction
time
second called subfunction
second called subfunction content
some content

Following figure shows dependencies of the `qwtb.m` file.



4.6 How uncertainty calculation works

4.7 How to add a new algorithm

To add a new algorithm, several steps have to be done.

1. Select an algorithm ID. Usually it is an acronym or abbreviation of the new algorithm name.
2. Create a directory named `alg_SOMEID`, where `SOMEID` is a selected ID. For a directory structure, see 4.1.
3. Put all files required by the algorithm (i.e. scripts, libraries) into the directory `alg_SOMEID/`.
4. Create a file `alg_SOMEID/alg_info.m`. An example of such file follows.

```
        % Part of QWTB. Info script for algorithm SOMEALG.
        %
        % See also qwtb

        info.id = 'SOMEID';
        info.name = 'SOMEALG';
        info.desc = 'SOMEID is an super mega hyper
    algorithm for calculation of the ultimate answer to
    everything.';
        info.citation = 'Some nifty paper in some super
    journal.';
        info.remarks = 'Very simple implementation';
        info.license = 'MIT License';
        info.inputs(1).name = 'a';
        info.inputs(1).desc = 'Some important input';
        info.inputs(1).alternative = 0;
        info.inputs(1).optional = 0;
        info.inputs(1).parameter = 0;
        info.inputs(2).name = 'b';
        info.inputs(2).desc = 'Some other input ';
        info.inputs(2).alternative = 0;
        info.inputs(2).optional = 0;
        info.inputs(2).parameter = 0;
        info.outputs(1).name = 'x';
        info.outputs(1).desc = 'Some output';
```

```
        info.outputs(2).name = 'y';
        info.outputs(2).desc = 'Other output';
        info.providesGUF = 1;
        info.providesMCM = 0;
```

f

5. Create a wrapper for the algorithm in a file `alg_SOMEID/alg_wrapper.m`, see 4.1.2. An example of simple wrapper file follows.

```
        % Part of QWTB. Wrapper script for algorithm
    SOMEALG.
        %
        % See also qwtb

        % Format input data
        % SOMEALG definition is:
        % function [x, y, z] = SOMEALG(a, b);
        a = datain.a.v;
        b = datain.b.v;

        % Call algorithm
        [x, y, z] = SOMEALG(a, b);

        % Format output data:
        dataout.x.v = x;
        dataout.y.v = y;
        dataout.z.v = z;

        end % function
```

f

6. Put a license of the algorithm into the file `alg_SOMEID/LICENSE.txt`.

7. Create a testing script `alg_SOMEID/alg_test.m`. This is optional, however recomended. An example follows.

```
        % Part of QWTB. Test script for algorithm SOMEALG
        %
        % See also qwtb
```

```
        % Generate sample data
        DI = [];
        U = 1; V = 2;
        DI.a.v = [U:V];
        DI.b.v = U/V;

        % Call algorithm
        DO = qwtb('SOMEID', DI);

        % Check results
        assert((DO.x.v > U.*(1-1e6)) & (DO.x.v < U.*(1+1e6
)));
        assert((DO.y.v > V.*(1-1e6)) & (DO.y.v < V.*(1+1e6
)));
        assert((DO.z.v > sqrt(U).*(1-1e6)) & (DO.z.v <
sqrt(U).*(1+1e6)));

        end % function
```

f

8. Create an example script `alg_SOMEID/alg_example.m`. This is optional, however recomended. An example follows.

```
        %% SOMEALGNAME
        % Example for algorithm SOMEID.
        %
        % SOMEID is an super mega hyper algorithm for
calculation of the ultimate answer to
        % everything.
        %

        %% Generate sample data
        % Two quantities are prepared: |a| and |b|,
representing something and something even more
        % important.
        DI = [];
        U = 1; V = 2;
        DI.a.v = [U:V];
        DI.b.v = U/V;
```

```
        %% Call algorithm
        % Use QWTB to apply algorithm |SOMEID| to data |DI
    |.
        CS.verbose = 1;
        DO = qwtb('SOMEID', DI, CS);

        %% Display results
        % Results is the very answer.
        x = DO.x.v
        y = DO.y.v
        z = DO.z.v
        %%
        % Errors of estimation in parts per milion:
        xerrppm = (DO.x.v - U)/U .* 1e6
        yerrppm = (DO.y.v - V)/V .* 1e6
        zerrppm = (DO.z.v - sqrt(U)/sqrt(U) .* 1e6
```

9. Check and test everything. Send your contribution to QWTB authors. Ask them to generate a new documentation. Celebrate.

5

QWTBvar

QWTBvar is a script used to variate input quantities, run algorithm or user function for all combinations of inputs, and evaluate or plot results.

User have to define values of input quantities, values of variated input quantities and algorithm or user function to run.

After finish of the calculations, user can obtain results, plot them, or convert the results to Look up table (LUT). LUT can be used to interpolate results and quickly estimate outputs without calculating the results again.

5.1 QWTBvar use

5.1.1 Run calculation

The main usage is repeated run of QWTB calling algorithm `algid`:

```
[jobfn] = qwtbvar('calc', 'algid', datain, datainvar, [calcset
   ])
```

The structure `datain` contains nominal values of input quantities, the same as when using QWTB, see 4.2.

The structure `datainvar` contains multiple values of input quantities that will be variated for different runs of QWTB (see 5.3).

If `datainvar` contains more input quantities, QWTBvar will make all possible permutations.

Settings for the calculations is defined in `calcset` (see 4.4). For QWTBvar, only sub-structure `calcset.var` is important. The rest is transferred to QWTB.

The output `jobfn` is a path to a file with stored informations needed to perform calculations.

5.1.2 Continue interrupted calculation

```
[jobfn] = QWTBVAR('cont', jobfn)
```

Continues interrupted calculation according calculation plan `jobfn`. QWTBvar will read all settings and calculation plan, check what is already calculated and will finish calculations.

5.1.3 Calculate particular job

```
[jobfn] = QWTBVAR('job', jobfn, jobids)
```

Mostly internal use. Calculates actual calculation job or multiple calculation jobs listed in `jobids`. This is useful for distribution of the work for parallel processing.

5.1.4 Get result

```
[ndres, ndresc, ndaxes, consts] = QWTBVAR('result', jobfn, [
   consts])
```

Obtain calculated output quantities from calculation `jobfn`. Optionally, the varied input quantities can be set to the values in `const` and output matrices will be sliced.

Output `ndres` contains structure of n-dimensional matrices with output quantities. Output `ndresc` contains cell of structures with particular results. Output `ndaxes` contains values for dimensions (axes) of `ndres` and `ndresc`. These *axes* are the same as the varied quantities. See 5.4 and 5.5.

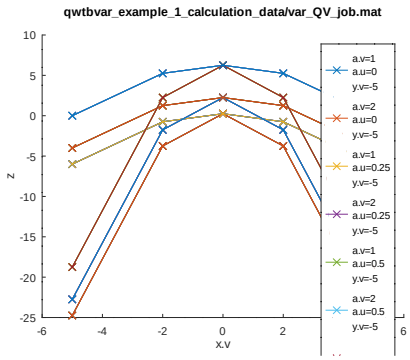
5.1.5 Plot result in 2D

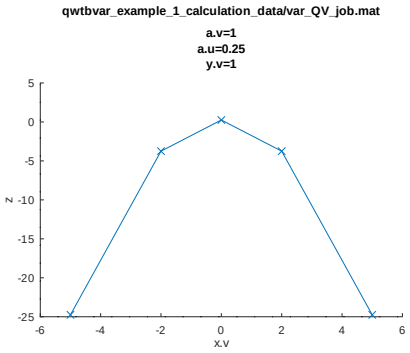
```
[H, x, y] = QWTBVAR('plot2D', jobfn, varx, vary, [consts])
```

Plots dependence of output quantity `vary` on varied input quantity `varx` in 2-dimensional plot, based on results in `jobfn`. Fills in properly title and axes labels. If results contain more varied quantities than `varx`, multiple dependences are plotted. If only single dependence is sought, the values of other varied input quantities can be defined in `consts`.

Output H is a handle to the figure. Outputs x and y are structures containing the plotted values and relevant labels (see 5.6).

As an example, consider calculation with varied input quantities x.v, y.v, a.v, a.u and scalar output quantity z.v. Plotting all dependencies of z.v on x.v results in too many plot lines, as shown in following figure on the left. However setting the consts variable to carry values of a.v, a.u and y.v, only one line is plotted, as shown in figure on the right:





5.1.6 Plot result in 3D

```
[H, x, y, z] = QWTBVAR('plot3D', jobfn, varx, vary, [consts])
```

Plots dependence of output quantity vary on varied input quantity varx in 3-dimensional plot, based on results in jobfn. Fills in properly title and axes labels. If results contain more varied quantities than varx, multiple dependences are plotted. If only one dependence is sought, the values of other varied input quantities can be defined in consts.

Output H is a handle to the figure. Outputs x, y and z are structures containing the plotted values and relevant labels (see 5.6).

5.1.7 Create look-up table

```
[lut] = QWTBVAR('lut', jobfn, axset, rqset)
```

Create look-up table based on the results of the job specified in input jobfn. Settings of the axis are defined in structure axset. Settings of the output quantity are defined in structure rqset. QWTBvar will create a n-dimensional look-up table

of the results. Number of dimensions is equal to the number of varied inputs. Interpolation methods for all varied quantities (axes) has to be defined together with description of boundary cases.

5.1.8 Interpolate look-up table

```
[ival] = QWTBVAR('interp', lut, ipoint)
[ival] = QWTBVAR('interp', lutfn, ipoint)
```

Interpolate output quantity `Q` at point `ipoint` using look-up table `lutfn`.

5.2 QWTBvar workflow

1. Select either QWTB algorithm or a user-defined script `'algid'`.
2. Create new structure with all needed input quantities and set values `datain`.
3. Create new structure with varied input quantities `datainvar` and set a range of values.
4. Run calculation of all results using 5.1.1.
5. If calculation was interrupted, continue using 5.1.2.
6. Obtain results of the calculation using 5.1.4.
7. Plot results of the calculation using 5.1.5 or 5.1.6.
8. Prepare a look-up table using 5.1.7.
9. Interpolate output quantity without actual calculation using 5.1.8.

5.3 Structure with varied quantities `datainvar`

Structure `datainvar` should contain only the quantities that should be varied by QWTBvar. Every field of the quantity in `datainvar` (e.g. `.v`, `.u` etc.) must contain one more dimension than the value of the same quantity in `datain`. If input quantity in `datainvar` is a vector, than input quantity in `datain` must be scalar. Matrix in `datainvar` means `datain` contains vector, etc.

QWTBvar will slice the input quantity in `datainvar` and one after another set this sliced value into input quantity `datain`.

5.3.1 Example with scalar quantity

Consider scalar input quantity `x`:

$$\begin{aligned} .\mathsf{v}: &\quad (v_1) \\ .\mathsf{u}: &\quad (u_1) \end{aligned}$$

If only value of the input quantity `x` should be varied, than `datainvar` must be of following:

$$\begin{aligned} .\mathsf{v}: &\quad (v_1, v_2, \ldots, v_i) \\ .\mathsf{u}: &\quad\quad (u_1) \end{aligned}$$

QWTBvar will do i calculations (or more if other quantities will be varied).

If only uncertainty of the input quantity `x` should be varied, than `datainvar` must be of following:

$$\begin{aligned} .\mathsf{v}: &\quad\quad (v_1) \\ .\mathsf{u}: &\quad (u_1, u_2, \ldots, u_j) \end{aligned}$$

QWTBvar will do j calculations (or more if other quantities will be varied).

If both value and uncertainty has to be varied by QWTBvar:

$$\begin{aligned} .\mathsf{v}: &\quad (v_1, v_2, \ldots, v_i) \\ .\mathsf{u}: &\quad (u_1, u_2, \ldots, u_j) \end{aligned}$$

For the last case, the QWTBvar will do $i \times j$ calculations (or more if other quantities will be varied).

5.3.2 Example with vector quantity

Consider scalar input quantity `x`:

$$.\mathsf{v}: \quad (v_1, v_2, \ldots, v_i)$$

`datainvar` must be of following:

$$\texttt{.v:} \quad \begin{pmatrix} v_{11} & \dots & v_{1j} \\ \vdots & \ddots & \vdots \\ v_{i1} & \dots & v_{ij} \end{pmatrix}$$

QWTBvar will do j calculations.

5.4 Outputs of QWTBvar `ndres`, `ndresc`, `ndaxis`

The outputs of the QWTBvar calculations are the output quantities of the used algorithm or user function. The outputs are in a form of n-dimensional matrices, where every dimension of the matrix is related to one varied input quantity.

5.4.1 `ndres`

Structure `ndres` contains all output quantities as matrices.

As an example, consider calculation with varied input quantities `x.v` (i elements), `y.v` (j elements), `a.v` (k elements), `a.u` (l elements), and scalar output quantity `z.v`. QWTBvar proceeded $i \times j \times k \times l$ calculations.

The `ndres` structure contains field of quantity `z`, that contains field value `.v`, that contains 4-dimensional matrix Z:

$$\texttt{ndres.z.v} = \underset{i \times j \times k \times l}{Z}, \tag{5.1}$$

where dimensions of the Z are i, j, k, l. If the output quantity z is not scalar, but vector or more, the `ndres.z.v` contains a cell of dimensions i, j, k, l.

5.4.2 `ndresc`

The `ndresc` cell contains cell of structures. Each cell element contains a structure with all output quantities.

For the previous example, the size of the cell `ndresc` dimensions are i, j, k, l. Each cell element `,j,k,lndresci` contains a structure with value of `z.v`:

$$\texttt{ndresc\{i,j,k,l\}.z.v} = Z_{i,j,k,l}. \tag{5.2}$$

5.4.3 ndaxes

This structure describes dimensions of n-dimensional matrix `ndres` or n-dimensional cell `ndresc`.

Fields of the structure are:

.names — Full names of varied quantities.
.values — Values of varied quantities in a matrix.
.valuesc — Values of varied quantities in a cell.
.Q — Names of varied quantities.
.f — Fields of varied quantities.

.names

Cell of strings. Full names of varied quantities. Example:

```
ndaxes.names = {'x.v', 'y.v', 'a.v', 'a.u'}
```

.values

Cell of values of the varied quantities. The values are matrix for scalar values, cells for other (in the last case the quantity `.values` is identical to a `.valuesc`). First value of `.values` cell represent a varied quantity listed first in `.names` etc.

.valuesc

Cell of cells with values of the varied quantities. First value of `.values` cell represent a varied quantity listed first in `.names` etc.

.Q

Cell of strings. Quantity name of the varied quantity. Part of the `.names`. Example:

```
ndaxes.Q = {'x', 'y', 'a', 'a'}
```

.f

Cell of strings. Field name of the varied quantity. Part of the `.names`. Example:

```
ndaxes.Q = {'v', 'v', 'v', 'u'}
```

5.5 Structure with constants `consts`

The structure is used to limit the results in the outputs to a selected subset. The results are sliced through the n-dimensional matrices at points selected in `consts`. The structure should contain one or more quantities in the same way as in `datain`. The quantities and values of the quantities has to be one of the `datainvar`.

Example: consider calculations proceeded with following `datainvar`:

```
datainvar.a.v = [1 2];
datainvar.a.u = [0 0.25 0.5];
datainvar.x.v = [-5  -2   0   2   5];
datainvar.y.v = [-5; -3; -1;  1;  3;  5];
```

The output quantities got dimensions $2 \times 3 \times 5 \times 6$. Following value of `consts`

```
consts.a.v = 1;
consts.x.v = -2;
```

limit output results and/or figures to dimensions 3×6.

5.6 Output structure with plot data

The outputs of the QWTBvar, when used in mode `'plot2D'` or `'plot3D'` (see 5.1.5 or 5.1.6), is a structure with following fields:

.name — Full name of the plotted quantity.
.data — Values of the plotted quantity.
.bars — Values of the plotted error bars.
.lbl — Label of the axis.
.ticklbl — Labels for plot ticks.
.Q — Name of the plotted quantity.
.f — Field of the plotted quantity.
.uncbar — Nonzero if error bars are to be plotted.

5.7 Structure for look-up table axis properties `axset`

2DO

5.8 Structure for look-up table XXXxxxxxxxxxxx `rqset`

2DO

5.9 Structure for look-up table interpolation `ipoint`

2DO

6

Licensing

Every algorithm has its own license. License of every algorithm is placed in the directory of the algorithm in a file named `LICENSE.txt`. Type of the license is included in the algorithm information structure, see chapter 4.2. The license of an algorithm can be displayed by following syntax:

```
license = qwtb('algid', 'license')
```

The license of the toolbox itself is MIT License, please see file `LICENSE.txt` in the directory containing script `qwtb.m`.

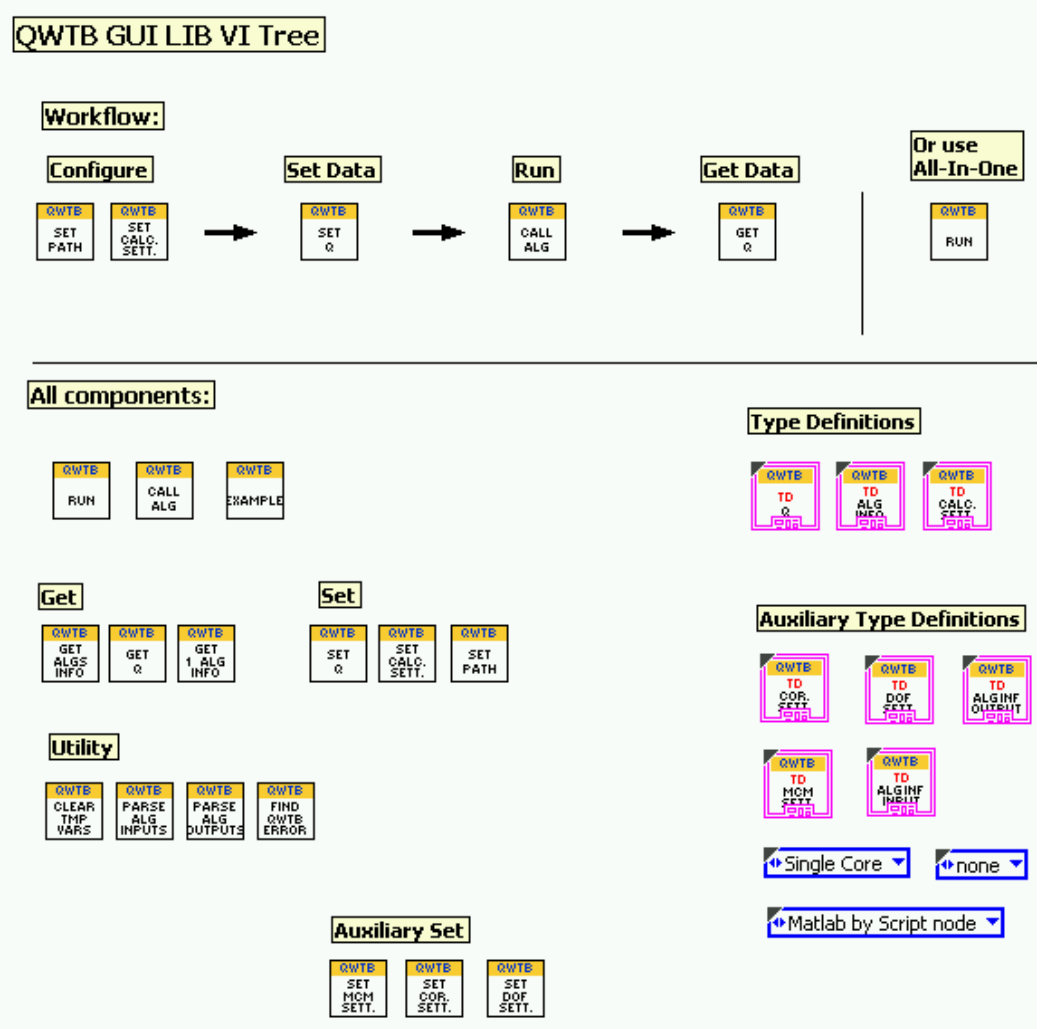
7

QWTBLVLib & simple QWTB GUI

7.1 QWTBLVLib

QWTBLVLib is a set of LabVIEW Virtual Instruments (VI) forming a library and providing an easy link between LabVIEW and QWTB. It uses *MATLAB Script Node* to make *ActiveX* calls to MATLAB and run QWTB. In future a use of GOLPI is intended to run QWTB by means of GNU Octave.

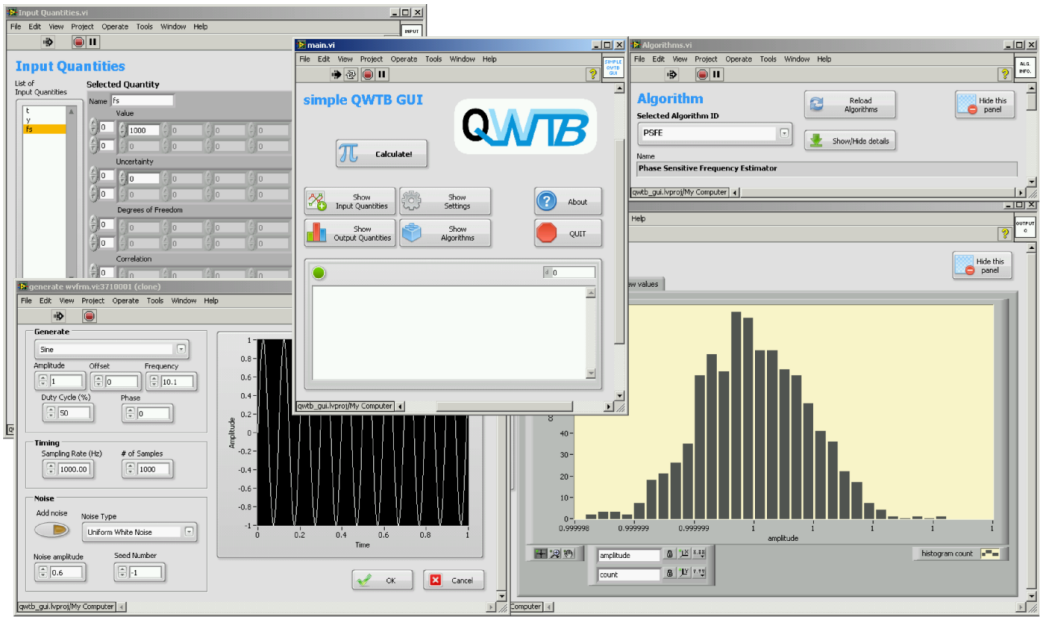
Description of particular VIs and example of use is a part of the library. Following figure shows the content of `VI Tree.vi`.



7.2 simple QWTB GUI

simple QWTB GUI (Graphical User Interface) is a fully working example of using QWTBLVLib written in LabVIEW. User can browse available algorithms and its descriptions, load or simulate data, run calculations with selected algorithm and view results.

See following figure with a screen shot of the Simple QWTB GUI.



After a start of simple QWTB GUI a window with several buttons is shown. To show a window with a list of input quantities, to set input quantities or to generate input quantities press button `Show Input Quantities`. To list of available algorithms and algorithm informations or to select algorithm press button `Show Algorithms`. To set calculation settings or GUI settings pres button `Show Settings`. To run calculation with set input quantities and selected algorithm press button `Calculate!`. To view output quantities press button `Show Output Quantities`. To quit the simple QWTB GUI press button `QUIT`.

The work flow is following. After the start user have to set the directory containing QWTB in window `Settings`. Next user have to set input quantities in window `Input Quantities`. After selecting required algorithm in window `Algorithms` the user can pres button `Calculate!`. The results can be reviewed in window `Output quantities`.

8

Bilbiography

[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 Measures, 2008.

[2] JCGM, *Evaluation of measurement data - Guide to the expression of uncertainty in measurement*, JCGM, Ed. Bureau International des Poids et Measures, 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')
                   alginfo = qwtb('algid', 'info')
                               qwtb('algid', 'addpath')
                               qwtb('algid', 'rempath')
                   license = qwtb('algid', 'license')
```

Algorithm informations structure (4.2):

.id — Designator of the algorithm (4.2.1)
.name — Name of the algorithm (4.2.2).
.desc — Basic description (4.2.3).
.citation — Reference (4.2.4).
.remarks — Any remark (4.2.5).
.license — License of the algorithm (4.2.6).
.inputs — Input quantities definitions (4.2.7).
.outputs — Output quantities definitions (4.2.8).
.providesGUF — Algorithm/wrapper calculates GUF uncertainty (4.2.9).
.providesMCM — Algorithm/wrapper calculates MCM uncertainty (4.2.10).
.fullpath — Full path to the algorithm. Automatically generated by the toolbox (4.2.11).

Quantity structure (4.3):

.v — Value (4.3.1).
.u — Uncertainty (4.3.2).
.d — Degree of freedom (4.3.3).
.c — Correlation (4.3.4).
.r — Randomized uncertainty (4.3.5).

Calculation settings structure (4.4):

.strict — (0) If zero, other fields generated automatically (4.4.1).
.verbose — (1) Display various informations (4.4.3).
.unc — ('none') How uncertainty is calculated ('none', 'guf', 'mcm') (4.4.4).
.loc — (0.6827) Required level of confidence of output uncertainties (0, 1) (4.4.5).
.cor.req — (0) Correlation matrix is required for all input quantities (4.4.6).
.cor.gen — (1) Zero correlation matrix is generated automatically if missing (4.4.6).
.dof.req — (0) Degrees of freedom are required for all input quantities (4.4.7).
.dof.gen — (1) Degree of freedom are generated automat. if missing with value 50 (4.4.7).
.mcm.repeats — (100) Number of Monte Carlo iterations (4.4.8).
.mcm.verbose — (1) Display various informations concerning Monte Carlo method (4.4.8).
.mcm.method — ('singlecore') Parallelization method ('multicore', 'multistation') (4.4.8).
.mcm.procno — (0) Number of processors to use (4.4.8).
.mcm.tmpdir — ('.') Directory for temporary data (4.4.8).
.mcm.randomize — (1) Randomized uncert. are generated automat. if missing (4.4.8).

Appendix B

Simple example of QWTB use

Contents

Overview

Sample data are simulated. QWTB is used to apply two different algorithms on the same data. Uncertainty of the results is calculated by means of Monte Carlo Method.

Generate sample data

Two quantities are prepared: `t` and `y`, representing 0.5 second of sinus waveform of nominal frequency 1 kHz, nominal amplitude 1 V and nominal phase 1 rad, sampled at sampling frequency `fsnom` 10 kHz.

```
DI = [];
Anom = 1; fnom = 1e3; phnom = 1; fsnom = 1e4;
DI.t.v = [0:1/fsnom:0.5];
DI.y.v = Anom*sin(2*pi*fnom*DI.t.v + phnom);
```

Add noise of standard deviation 1 mV:

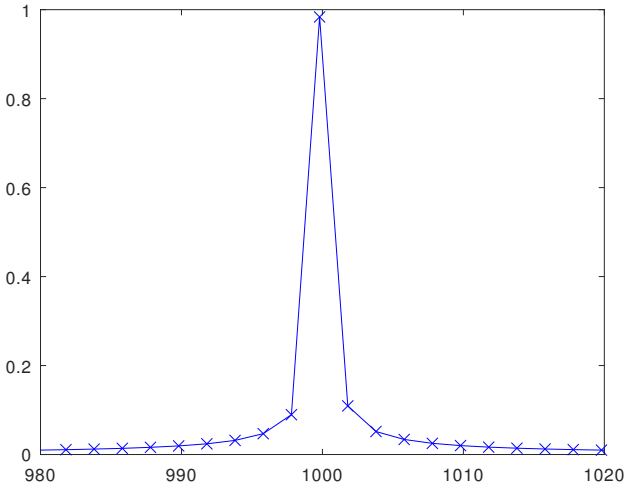
```
DI.y.v = DI.y.v + 1e-3.*randn(size(DI.y.v));
```

Analyzing data

To get a frequency spectrum, algorithm `SP-WFFT` can be used. This algorithm requires sampling frequency, so third quantity `fs` is added.

```
DI.fs.v = fsnom;
DO = qwtb('SP-WFFT', DI);
plot(DO.f.v, DO.A.v, '-xb'); xlim([980 1020])
```

```
QWTB: no uncertainty calculation
```



One can see it is not a coherent measurement. Therefore to get 'unknown' amplitude and frequency of the signal algorithm `PSFE` can be used:

```
DO = qwtb('PSFE', DI);
f = DO.f.v
A = DO.A.v
```

```
QWTB: no uncertainty calculation
QWTB: PSFE wrapper: sampling time was calculated from sampling
    frequency
f = 1000.0
A = 1.0000
```

Uncertainties

Uncertainties are added to the `t` (time stamps) and `y` (sampled data) structures.

```
DI.t.u = zeros(size(DI.t.v)) + 1e-5;
DI.y.u = zeros(size(DI.y.v)) + 1e-4;
```

Calculations settings is created with Monte Carlo uncertainty calculation method, 1000 repeats and singlecore calculation. The output of messages is supressed to increase calculation speed.

```
CS.unc = 'mcm';
CS.mcm.repeats = 1000;
CS.mcm.method = 'singlecore';
CS.verbose = 0;
```

An uncertainty of sampling frequency has to be added. Let suppose the value:

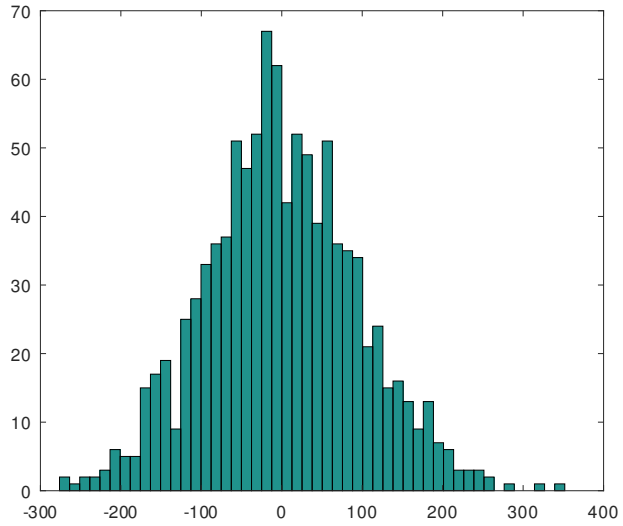
```
DI.fs.u = 1e-3;
```

Run PSFE algorithm on input data `DI` and with calculattion settings `CS`.

```
DO = qwtb('PSFE',DI,CS);
```

Result is displayed as a histogram of calculated frequency (error from mean value multiplied by 1e6)

```
figure; hist((DO.f.r - mean(DO.f.r)).*1e6, 50);
```



One can see the histogram maybe is, maybe is not a Gaussian function. To get correct uncertainties, a shortest coverage interval has to be calculated.

Appendix C

Long example of QWTB use

Contents

Overview

Data are simulated, QWTB is used to estimate quanties using different algorithms.

Generate ideal data

Sample data are generated, representing 1 second of sine waveform of nominal frequency `fnom` 1000 Hz, nominal amplitude `Anom` 1 V and nominal phase `phnom` 1 rad. Data are sampled at sampling frequency `fsnom` 10 kHz, perfectly synchronized, no noise.

```
Anom = 1; fnom = 1000; phnom = 1; fsnom = 10e4;
timestamps = [0:1/fsnom:0.1-1/fsnom];
```

```
ideal_wave = Anom*sin(2*pi*fnom*timestamps + phnom);
```

To use QWTB, data are put into two quantities: `t` and `y`. Both quantities are put into data in structure `DI`.

```
DI = [];
DI.t.v = timestamps;
DI.y.v = ideal_wave;
```

Apply three algorithms

QWTB will be used to apply three algorithms to determine frequency and amplitude: `SP-WFFT`, `PSFE` and `FPNLSF`. Results are in data out structure `DOxxx`. Algorithm `FPNLSF` requires an estimate, select it to 0.1% different from nominal frequency. `SP-WFFT` requires sampling frequency.

```
DI.fest.v = fnom.*1.001;
DI.fs.v = fsnom;
DOspwfft = qwtb('SP-WFFT', DI);
DOpsfe = qwtb('PSFE', DI);
DOfpnlsf = qwtb('FPNLSF', DI);
```

```
QWTB: no uncertainty calculation
QWTB: no uncertainty calculation
QWTB: PSFE wrapper: sampling time was calculated from sampling
    frequency
QWTB: no uncertainty calculation
Fitting started
Fitting finished
```

Compare results for ideal signal

Calculate relative errors in ppm for all algorithm to know which one is best. `SP-WFFT` returns whole spectrum, so only the largest amplitude peak is interesting. One can see for the ideal case all errors are very small.

```
disp('SP-WFFT errors (ppm):')
[tmp, ind] = max(DOspwfft.A.v);
```

```
ferr  = (DOspwfft.f.v(ind) - fnom)/fnom .* 1e6
Aerr  = (DOspwfft.A.v(ind) - Anom)/Anom .* 1e6
pherr = (DOspwfft.ph.v(ind) - phnom)/phnom .* 1e6

disp('PSFE errors (ppm):')
ferr  = (DOpsfe.f.v - fnom)/fnom .* 1e6
Aerr  = (DOpsfe.A.v - Anom)/Anom .* 1e6
pherr = (DOpsfe.ph.v - phnom)/phnom .* 1e6

disp('FPNLSF errors (ppm):')
ferr  = (DOfpnlsf.f.v - fnom)/fnom .* 1e6
Aerr  = (DOfpnlsf.A.v - Anom)/Anom .* 1e6
pherr = (DOfpnlsf.ph.v - phnom)/phnom .* 1e6
```

```
SP-WFFT errors (ppm):
ferr = 0
Aerr = 0
pherr = -6.5503e-09
PSFE errors (ppm):
ferr = -2.2737e-10
Aerr = -3.2196e-09
pherr = 4.7518e-08
FPNLSF errors (ppm):
ferr = -6.4733e-07
Aerr = -1.4322e-08
pherr = 8.8987e-05
```

Noisy signal

To simulate real measurement, noise is added with normal distribution and standard deviation `sigma` of 100 microvolt. Algorithms are again applied.

```
sigma = 100e-6;
DI.y.v = ideal_wave + 100e-6.*randn(size(ideal_wave));
DOspwfft = qwtb('SP-WFFT', DI);
DOpsfe = qwtb('PSFE', DI);
DOfpnlsf = qwtb('FPNLSF', DI);
```

```
QWTB: no uncertainty calculation
QWTB: no uncertainty calculation
QWTB: PSFE wrapper: sampling time was calculated from sampling
    frequency
QWTB: no uncertainty calculation
Fitting started
Fitting finished
```

Compare results for noisy signal

Again relative errors are compared. One can see amplitude and phase errors increased to several ppm, however frequency is still determined quite good by all three algorithms. FFT is not affected by noise at all.

```
disp('SP-WFFT errors (ppm):')
[tmp, ind] = max(DOspwfft.A.v);
ferr  = (DOspwfft.f.v(ind) - fnom)/fnom .* 1e6
Aerr  = (DOspwfft.A.v(ind) - Anom)/Anom .* 1e6
pherr = (DOspwfft.ph.v(ind) - phnom)/phnom .* 1e6

disp('PSFE errors:')
ferr  = (DOpsfe.f.v - fnom)/fnom .* 1e6
Aerr  = (DOpsfe.A.v - Anom)/Anom .* 1e6
pherr = (DOpsfe.ph.v - phnom)/phnom .* 1e6

disp('FPNLSF errors:')
ferr  = (DOfpnlsf.f.v - fnom)/fnom .* 1e6
Aerr  = (DOfpnlsf.A.v - Anom)/Anom .* 1e6
pherr = (DOfpnlsf.ph.v - phnom)/phnom .* 1e6
```

```
SP-WFFT errors (ppm):
ferr = 0
Aerr = -0.1921
pherr = 3.5831
PSFE errors:
ferr = -5.9336e-03
Aerr = -0.1910
pherr = 5.4498
FPNLSF errors:
```

```
ferr = -2.4370e-03
Aerr = -0.1917
pherr = 4.3516
```

Non-coherent signal

In real measurement coherent measurement does not exist. So in next test the frequency of the signal differs by 20 ppm:

```
fnc = fnom*(1 + 20e-6);
noncoh_wave = Anom*sin(2*pi*fnc*timestamps + phnom);
DI.y.v = noncoh_wave;
DOspwfft = qwtb('SP-WFFT', DI);
DOpsfe = qwtb('PSFE', DI);
DOfpnlsf = qwtb('FPNLSF', DI);
```

```
QWTB: no uncertainty calculation
QWTB: no uncertainty calculation
QWTB: PSFE wrapper: sampling time was calculated from sampling
    frequency
QWTB: no uncertainty calculation
Fitting started
Fitting finished
```

Compare results for non-coherent signal

Comparison of relative errors. Results of PSFE or FPNLSF are correct, however FFT is affected by non-coherent signal considerably.

```
disp('SP-WFFT errors (ppm):')
[tmp, ind] = max(DOspwfft.A.v);
ferr  = (DOspwfft.f.v(ind) - fnc)/fnc .* 1e6
Aerr  = (DOspwfft.A.v(ind) - Anom)/Anom .* 1e6
pherr = (DOspwfft.ph.v(ind) - phnom)/phnom .* 1e6

disp('PSFE errors:')
ferr  = (DOpsfe.f.v - fnc)/fnc .* 1e6
Aerr  = (DOpsfe.A.v - Anom)/Anom .* 1e6
```

```
pherr = (DOpsfe.ph.v - phnom)/phnom .* 1e6

disp('FPNLSF errors:')
ferr  = (DOfpnlsf.f.v - fnc)/fnc .* 1e6
Aerr  = (DOfpnlsf.A.v - Anom)/Anom .* 1e6
pherr = (DOfpnlsf.ph.v - phnom)/phnom .* 1e6
```

```
SP-WFFT errors (ppm):
ferr = -20.000
Aerr = -2.8780
pherr = 6291.9
PSFE errors:
ferr = -1.1368e-10
Aerr = 6.6613e-10
pherr = 2.9754e-08
FPNLSF errors:
ferr = -6.0583e-07
Aerr = -1.5765e-08
pherr = 8.3299e-05
```

Harmonically distorted signal.

In other cases a harmonic distortion can appear. Suppose a signal with second order harmonic of 10% amplitude as the main signal.

```
hadist_wave = Anom*sin(2*pi*fnom*timestamps + phnom) + 0.1*Anom
   *sin(2*pi*fnom*2*timestamps + 2);
DI.y.v = hadist_wave;
DOspwfft = qwtb('SP-WFFT', DI);
DOpsfe = qwtb('PSFE', DI);
DOfpnlsf = qwtb('FPNLSF', DI);
```

```
QWTB: no uncertainty calculation
QWTB: no uncertainty calculation
QWTB: PSFE wrapper: sampling time was calculated from sampling
    frequency
QWTB: no uncertainty calculation
Fitting started
```

```
Fitting finished
```

Compare results for harmonically distorted signal.

Comparison of relative errors. SP-WFFT or PSFE are not affected by harmonic distortion, however FPNLSF is thus is not suitable for such signal.

```
disp('SP-WFFT errors (ppm):')
[tmp, ind] = max(DOspwfft.A.v);
ferr  = (DOspwfft.f.v(ind) - fnom)/fnom .* 1e6
Aerr  = (DOspwfft.A.v(ind) - Anom)/Anom .* 1e6
pherr = (DOspwfft.ph.v(ind) - phnom)/phnom .* 1e6

disp('PSFE errors:')
ferr  = (DOpsfe.f.v - fnom)/fnom .* 1e6
Aerr  = (DOpsfe.A.v - Anom)/Anom .* 1e6
pherr = (DOpsfe.ph.v - phnom)/phnom .* 1e6

disp('FPNLSF errors:')
ferr  = (DOfpnlsf.f.v - fnom)/fnom .* 1e6
Aerr  = (DOfpnlsf.A.v - Anom)/Anom .* 1e6
pherr = (DOfpnlsf.ph.v - phnom)/phnom .* 1e6
```

```
SP-WFFT errors (ppm):
ferr = 0
Aerr = 0
pherr = -6.6613e-09
PSFE errors:
ferr = -2.2737e-10
Aerr = -4.3299e-09
pherr = 4.4853e-08
FPNLSF errors:
ferr = -0.7820
Aerr = 0.1103
pherr = 236.67
```

Harmonically distorted, noisy, non-coherent signal.

In final test all distortions are put in a waveform and results are compared.

```
err_wave = Anom*sin(2*pi*fnc*timestamps + phnom) + 0.1*Anom*sin
   (2*pi*fnc*2*timestamps + 2) + 100e-6.*randn(size(ideal_wave
   ));
DI.y.v = err_wave;
DOspwfft = qwtb('SP-WFFT', DI);
DOpsfe = qwtb('PSFE', DI);
DOfpnlsf = qwtb('FPNLSF', DI);
```

```
QWTB: no uncertainty calculation
QWTB: no uncertainty calculation
QWTB: PSFE wrapper: sampling time was calculated from sampling
    frequency
QWTB: no uncertainty calculation
Fitting started
Fitting finished
```

Compare results for harmonically distorted, noisy, non-coherent signal.

```
disp('SP-WFFT errors (ppm):')
[tmp, ind] = max(DOspwfft.A.v);
ferr  = (DOspwfft.f.v(ind) - fnc)/fnc .* 1e6
Aerr  = (DOspwfft.A.v(ind) - Anom)/Anom .* 1e6
pherr = (DOspwfft.ph.v(ind) - phnom)/phnom .* 1e6

disp('PSFE errors:')
ferr  = (DOpsfe.f.v - fnc)/fnc .* 1e6
Aerr  = (DOpsfe.A.v - Anom)/Anom .* 1e6
pherr = (DOpsfe.ph.v - phnom)/phnom .* 1e6

disp('FPNLSF errors:')
ferr  = (DOfpnlsf.f.v - fnc)/fnc .* 1e6
Aerr  = (DOfpnlsf.A.v - Anom)/Anom .* 1e6
pherr = (DOfpnlsf.ph.v - phnom)/phnom .* 1e6
```

```
SP-WFFT errors (ppm):
ferr = -20.000
Aerr = 0.3775
pherr = 6295.2
PSFE errors:
ferr = 1.1917e-04
Aerr = 2.7827
pherr = 1.8512
FPNLSF errors:
ferr = -0.7626
Aerr = 2.8970
pherr = 233.03
```

Appendix D

Simple example of QWTBvar use

D.1 The example

Contents

Overview

A function qwtbvar_example_1_process() calculating hyperparaboloid is used to demonstrate the QWBTvar. Input quantities will be varied to calculate outputs for multiple series of inputs.

Inputs

Input quantities:

```
DI.x.v = 0;
DI.y.v = 0;
DI.a.v = 1;
DI.a.u = 0;
DI.b.v = 2;
```

Varied input quantities:

```
DIvar.a.v = [1 2];
DIvar.a.u = [0 0.25 0.5];
DIvar.x.v = [-5  -2   0   2   5];
DIvar.y.v = [-5; -3; -1;  1;  3;  5];
```

Some optional settings:

```
CS.verbose = 1;
CS.mcm.repeats = 1e3;
CS.var.dir = 'qwtbvar_example_1_calculation_data';
CS.var.cleanfiles = 1;
CS.var.method = 'singlecore';
CS.var.procno = 1;
```

Calculation

Run whole calculation. Input quantities in DI will be varied according DIvar:

```
[jobfn] = qwtbvar('calc', 'qwtbvar_example_1_process', DI,
   DIvar, CS);
```

```
### QWTBVAR: Number of variation calculations is: 180
### QWTBVAR: All calculations finished. The calculations took
    0.0573 s (0.000956 m, 1.59e-05 h)
```

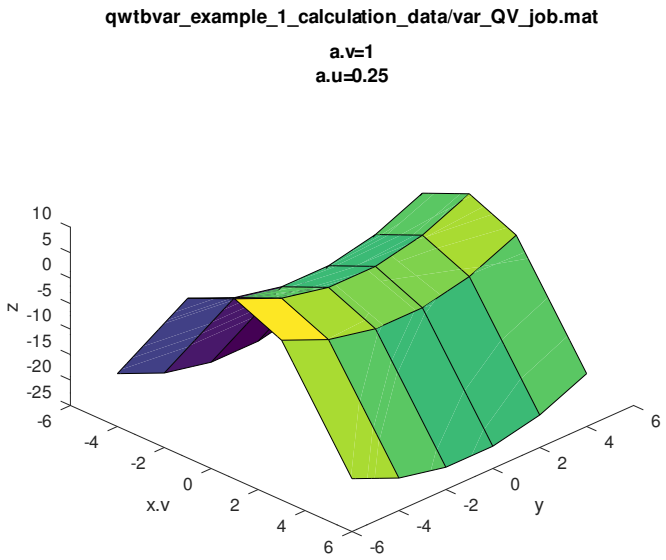
Plotting

Plot results. Results are multidimensional. To plot 3D or 2D plots, the consts will contain 'constant' values that will be used to select data from multidimensional results to decrease dimensions and simplify plotting:

```
consts.a.v = 1;
consts.a.u = 0.25;
```

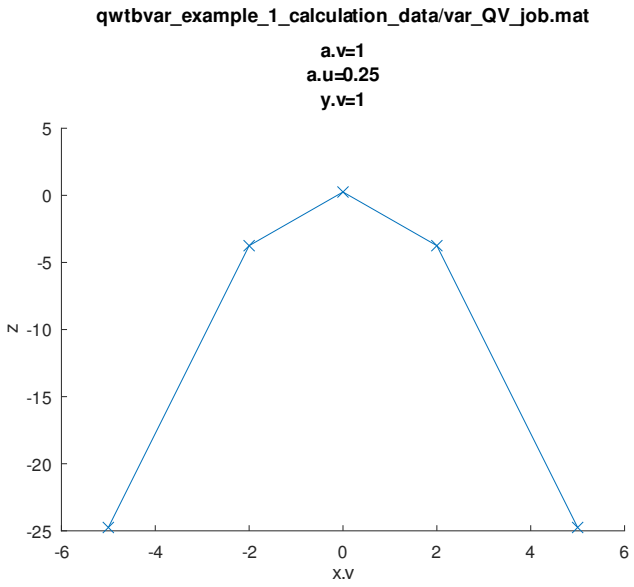
3 dimensional plot:

```
qwtbvar('plot3D', jobfn, 'x', 'y', 'z', consts);
```



Add another constant to plot simple 2 dimensional figure:

```
consts.y.v = 1;
qwtbvar('plot2D', jobfn, 'x', 'z', consts);
```



Look Up Table

Set up additional information for axes of the multidimensional results that will be used to create look-up table.

```
axset.a.v.min_ovr = min(DIvar.a.v)-0.1;
axset.a.v.max_ovr = max(DIvar.a.v)+0.1;
axset.a.v.min_lim = 'error';
axset.a.v.max_lim = 'error';
axset.a.v.scale = 'lin';
axset.a.u.min_ovr = min(DIvar.a.u);
axset.a.u.max_ovr = max(DIvar.a.u);
axset.a.u.min_lim = 'error';
axset.a.u.max_lim = 'error';
axset.a.u.scale = 'lin';
axset.x.v.min_ovr = min(DIvar.x.v)-3;
axset.x.v.max_ovr = max(DIvar.x.v)+3;
axset.x.v.min_lim = 'const';
axset.x.v.max_lim = 'const';
axset.x.v.scale = 'lin';
axset.y.v.min_ovr = min(DIvar.y.v)-3;
axset.y.v.max_ovr = max(DIvar.y.v)+3;
axset.y.v.min_lim = 'const';
axset.y.v.max_lim = 'const';
axset.y.v.scale = 'lin';
```

Set type of scaling for output quantity.

```
rqset.z.v.scale = 'lin';
```

Create look up table that will be used for interpolation.

```
lut = qwtbvar('lut', jobfn, axset, rqset);
```

Interpolation of results

Set parameters for interpolation:

```
ipoint.a.v = 1.5;
ipoint.a.u = 0;
```

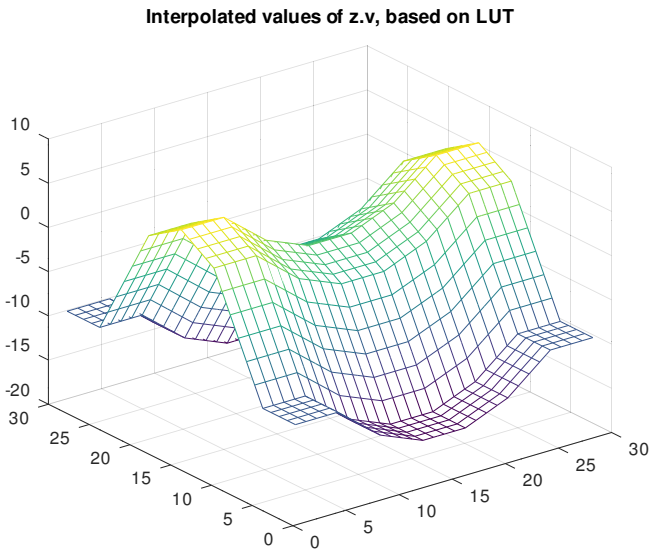
```
ipoint.x.v = 0;
ipoint.y.v = 0;
```

Create a range of input quantities for which the interpolation will be done:

```
x = -7:0.5:7;
y = -7:0.5:7;
% For all input quantities values, interpolate:
for j = 1:numel(x)
    ipoint.x.v = x(j);
    for k = 1:numel(y)
        ipoint.y.v = y(k);
        tmp = qwtbvar('interp', lut, ipoint);
        ival(j, k) = tmp.z.v;
    end
end
```

Plot figure with interpolated values based on the LUT:

```
figure
mesh(ival)
title('Interpolated values of z.v, based on LUT')
xlabel = 'x.v';
ylabel = 'y.v';
```



D.2 The varied function `qwtbvar_example_1_process.m`

```
function [DO, DI, CS] = qwtbvar_example_1_process(DI, CS)
% Calculates hyperparaboloid.
% Input quantities: x, y, a, b
% Output quantities: z

    if isfield(DI.x, 'u') && isfield(DI.y, 'u') && isfield(DI.a
    , 'u') && isfield(DI.b, 'u') && isfield(CS.mcm, 'repeats')
        x = normrnd(DI.x.v, DI.x.u, CS.mcm.repeats, 1);
        y = normrnd(DI.y.v, DI.y.u, CS.mcm.repeats, 1);
        a = normrnd(DI.a.v, DI.a.u, CS.mcm.repeats, 1);
        b = normrnd(DI.b.v, DI.b.u, CS.mcm.repeats, 1);
        z = y.^2./b.^2 - x.^2./a.^2;
        DO.z.v = mean(z);
        DO.z.u = std(z);
        DO.z.r = z;
    else
        x = DI.x.v;
        y = DI.y.v;
        a = DI.a.v;
        b = DI.b.v;
        DO.z.v = y.^2./b.^2 - x.^2./a.^2;
    end
end % function
```

../qwtb/qwtbvar_example_1_process.m

Appendix E

Complex example of QWTBvar use

E.1 The example

Contents

Overview

The example shows how nonscalar quantities can be used in variations and plotting.

Inputs

All input quantities are vectors. First value is x commponent, second value is y component. Gravity acceleration is approx. [0, -9.81]. Starting velocity (m/s):

```
DI.v.v = [0 0];
DI.v.u = [0 0];
```

Evaluation will be conducted at times (s):

```
DI.t.v = [0 1 2 3];
DI.t.u = [0.1 0.2 0.3 0.4 0.5].*0.0001;
```

Varied input quantities: Starting velocity is for 3 cases: 1, still object, 2, object thrown upward, 3, object thrown forward, 4, object thrown backward:

```
DIvar.v.v = [0 0; 0 10; 10 0; -10 0];
```

Variation for velocity uncertainties: none, small, large:

```
DIvar.v.u = [0 0; 0.2 0.2; 10 100];
```

Variation: Results calculated for short range of times and distant times:

```
DIvar.t.v = [0 1 2 3; 4 5 6 7];
```

Some optional settings:

```
CS.verbose = 1;
CS.mcm.repeats = 10;
CS.var.dir = 'qwtbvar_example_2_calculation_data';
CS.var.cleanfiles = 1;
CS.var.method = 'singlecore';
CS.var.procno = 1;
```

Calculation

Run whole calculation. Input quantities in DI will be varied according DIvar:

```
[jobfn] = qwtbvar('calc', 'qwtbvar_example_2_process', DI,
   DIvar, CS);
```

```
### QWTBVAR: Number of variation calculations is: 24
### QWTBVAR: All calculations finished. The calculations took
   0.00832 s (0.000139 m, 2.31e-06 h)
```

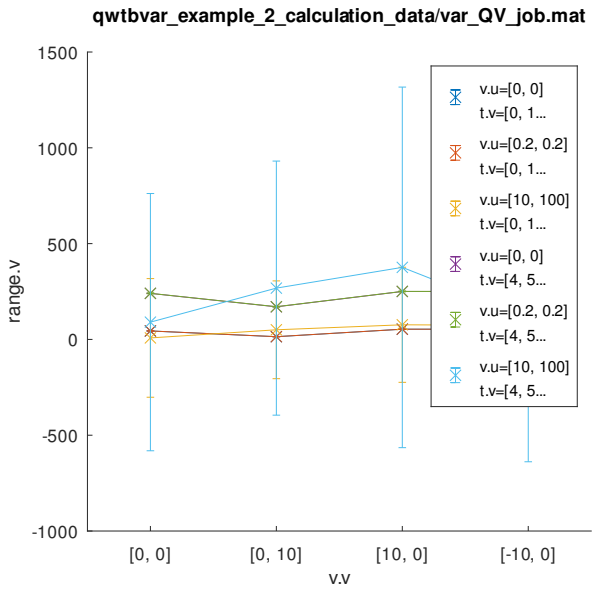
Results

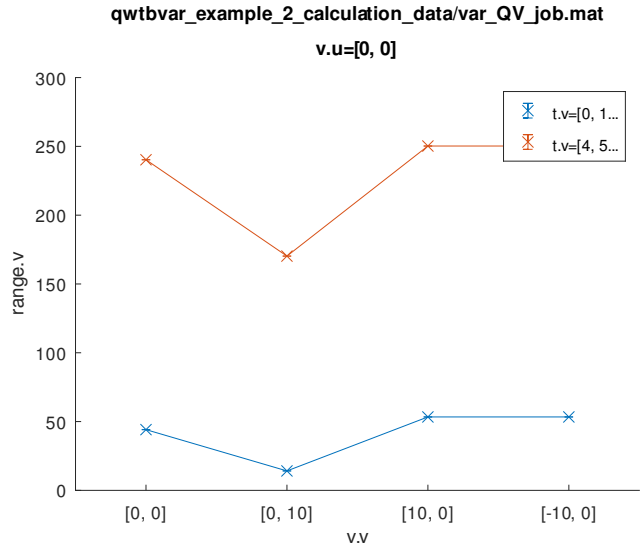
Get results as multidimensional arrays, where every dimension represents one varied quantity. The result will be for selected settings of consts.

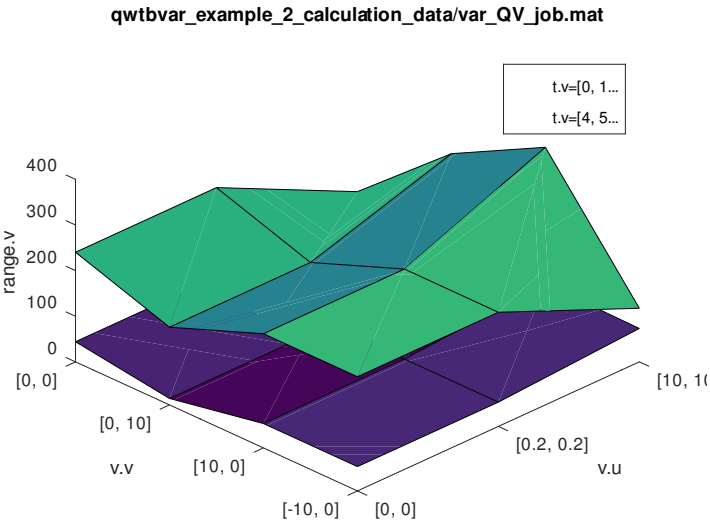
```
qwtbvar('plot2D', jobfn, 'v.v', 'range.v');
consts.v.v = [0   0];
consts.v.u = [0   0];
qwtbvar('result', jobfn, consts);
qwtbvar('plot2D', jobfn, 'v.v', 'range.v', consts);
qwtbvar('plot3D', jobfn, 'v.v', 'v.u', 'range.v', consts);



% % consts.v.v = [1 2];
% consts.v.u = [0.1 0.2];
% [ndres, ndresc, ndaxes] = qwtbvar('result', jobfn, consts);
% % ndres is a structure with results as matrices. Although,
   because results are not scalars,
% % it contains cells.
% % ndresc is a cell with structures of results.
% % ndaxes is description of which quantity was varied in which
    dimension.
```







E.2 The varied function `qwtbvar_example_2_process.m`

```
% Example for qwtbvar with 2 vector inputs and 1 vector output.
% Calculates position and acceleration of an object thrown at t
    = 0 from
% coordinates (0,0) with velocity (v_x, v_y) in gravitational
   field of magnitude
% g = (0, 9.8067) +- (0, 0.0001). Position is evaluated for all
    times specified
```

```
% in vector t.
%
% Inputs:
%   'v' is vector of initial velocity of two elements with
   vector components [v_x, v_y] in in 2D plane.
%   't' is vector of times for which the position will be
   evaluated [t_1, t_2, ..., t_n].
% Outputs:
%   's' is position vector of size 2*n: [s_x1, s_y1; s_x2, s_y2
   ; ..., s_xn, s_yn];
%   'a' is acceleration vector size 2: [a_x, a_y];
%
% Example of inputs:
%   DI.v.v = [1 20]; % speed 1 m/s in x coordinate, 2 m/s in y
   coordinate
%   DI.v.u = [0.1 0.2]; % speed uncertainties
%   DI.t.v = [1 2 3 4 5]; % evaluation times 1 s, 2 s, 3 s, 4 s
%   DI.t.u = [0.1 0.2 0.3 0.4 0.5]; % uncertainties of time 0.1
    s etc.
%   CS.mcm.repeats = 1e3;
% Run:
%   [DO, DI, CS] = qwtbvar_example_2_process(DI, CS);
% Output:
%   DO.s.v =
%      0.9991   15.0641
%      1.9942   20.1867
%      2.9776   15.5779
%      4.0281    0.5859
%      4.9917  -23.7004
%   DO.s.u =
%      0.1338    0.9857
%      0.2862    0.4841
%      0.4058    2.7508
%      0.5637    7.9105
%      0.7121   14.3013
%   DO.a.v =
%          0   9.8067
%   DO.a.u =
%          0   9.8345e-5

function [DO, DI, CS] = qwtbvar_example_2_process(DI, CS)
```

```
    % acceleration vector:
    Gx =  0;
    uGx = 0;
    Gy =  -9.8067; % negative because falling down
    uGy = 0.0001;

    DI.v.v = DI.v.v(:)'; % ensure row vector - is it needed?
   QWTB does it on its own.
    DI.t.v = DI.t.v(:)';

    % initialize output variables:
    DO.s.v = nan.*zeros(numel(DI.t.v), 2);
    DO.a.v = DO.s.v;

    % with or without uncertainties?
    if isfield(DI.v, 'u') && isfield(DI.t, 'u') && isfield(CS.
   mcm, 'repeats')
        DI.v.u = DI.v.u(:)'; % ensure row vector - is it needed
   ? QWTB does it on its own.
        DI.t.u = DI.t.u(:)';
        % randomize G, v, t:
        gx = normrnd(Gx, uGx, CS.mcm.repeats, 1);
        gy = normrnd(Gy, uGy, CS.mcm.repeats, 1);
        vx = normrnd(DI.v.v(1), DI.v.u(1), CS.mcm.repeats, 1);
        vy = normrnd(DI.v.v(2), DI.v.u(2), CS.mcm.repeats, 1);
        % using for loop because mvnrnd is not in basic matlab:
        for j = 1:numel(DI.t.v)
            % randomize time vector:
            t = normrnd(DI.t.v(j), DI.t.u(j), CS.mcm.repeats,
   1);
            % calculate coordinates:
            sx = vx.*t + 0.5.*gx.*(t.^2);
            sy = vy.*t + 0.5.*gy.*(t.^2);
            DO.s.v(j, 1) = mean(sx);
            DO.s.v(j, 2) = mean(sy);
            % calculate coordinate uncertainties:
            DO.s.u(j, 1) = std(sx);
            DO.s.u(j, 2) = std(sy);
        end % for
        % uncertainty of acceleration
        DO.a.u = [std(gx) std(gy)];
```

```
        % uncertainty of final distance from start:
        DO.range.u = sqrt(sum(DO.s.u(end, :).^2, 2));
    else
        gx = Gx;
        gy = Gx;
        t = DI.t.v;
        DO.s.v(:, 1) = DI.v.v(1).*t - 0.5.*gx.*(t.^2);
        DO.s.v(:, 2) = DI.v.v(2).*t - 0.5.*gy.*(t.^2);
        DO.a.v(:, 1) = Gx.*ones(size(DO.s.v(:, 1)));
        DO.a.v(:, 2) = Gy.*ones(size(DO.s.v(:, 2)));
    end
    % value of acceleration
    DO.a.v = [mean(gx) mean(gy)];
    % final distance from start:
    DO.range.v = sqrt(sum(DO.s.v(end, :).^2, 2));
end % function DO = qwtbvar_example_process2(DI)

%!test
%! DI.v.v = [1 20];
%! DI.t.v = [1 2 3 4 5];
%! CS.mcm.repeats = 1000;
%! [DO, DI, CS] = qwtbvar_example_2_process(DI, CS);
%! assert(all(size(DO.s.v) == [5, 2]))
%! assert(all(size(DO.a.v) == [1, 2]))
%! DI.v.u = [0.1 0.2];
%! DI.t.u = [0.1 0.2 0.3 0.4 0.5];
%! [DO, DI, CS] = qwtbvar_example_2_process(DI, CS);
%! assert(all(size(DO.s.u) == [5, 2]))
%! assert(all(size(DO.a.u) == [1, 2]))

% vim settings modeline: vim: foldmarker=%<<<,%>>> fdm=marker
   fen ft=matlab textwidth=80 tabstop=4 shiftwidth=4
```

../qwtb/qwtbvar_example_2_process.m