Matlab project structure

General rules

- don't implement anything already implemented by Matlab
  - *reason*: it costs unnecessary time
  - *reason*: own versions usually contain more bugs
  - *reason*: library functions do not require maintenance effort
- use the newest version of Matlab (keep up to date)
  - *reason*: often things are implemented that are already included in newer Matlab versions
  - *reason*: the older the version, the higher the chance newer versions removed support of somthing you use which increases porting effort
- don't use spaces in file and directory names
  - use minus (`-`) instead of spaces
  - except for files for which something else has been explicitly specified (e.g. source code files …)
  - *reason*: spaces are extremely annoying when working on the command line
- don't use umlauts in file and directory names
  - *reason*: can cause problems on computers where a different language is set
- don't use upper letters in file and directory names
  - except for files for which something else has been explicitly specified (e.g. `README.md`, source code files …)
  - *reason*: most but not all file systems distinguish between upper and lower case, all lower case avoids bugs due to different upper and lower case when referencing files (e.g. from Matlab code or script files)
- data should not be uploaded compressed (in a ZIP file)
  - *reason*: it is not possible to link from the documentation to files in archives
  - *reason*: files will be compressed within the Git history anyway, additional manual compression will do the opposite, the files will get larger
- measurement data does not belong in the project
  - unless it is used by the unit tests
  - *reason*: measurement data are not subject to further development and are therefore not relevant for version management.
- unit test data should always be minimal examples
  - used measured data must be adapted accordingly
  - *reason*: unit tests are performed after each change to ensure that the change does not break when the test data is large, these tests take a very long time and are no longer practical

Notice our Matlab gotcha project.

Main directory `project-root`

```
- 📁 project-root
    - 📁 .git
    - 📁 3rdparty
    - 📁 logo
    - 📁 doc
    - 📁 src
    - 📁 lib
    - 📁 test
    - 📁 example
    - 📁 hardware
    - 📁 script
    - 📄 .gitignore
    - 📄 .gitlab-ci.yml
    - 📄 .gitmodules
    - 📄 README.md
```

The root directory should contain `README.md` as the only file. Excluded from this are of course technical help files such as `.gitignore`, `.gitlab-ci.yml` or `.gitmodules`.

File `README.md`

We have two different templates, one for end-user software and one for libraries.

End-user software `README.md` template

Remove the unneeded parts and add project-specific ones if necessary.

~~~~
# Project name

Overview description of the project.

## System requirements

- Windows 10, 64 Bit
- Matlab R2019a Update 3, 64 Bit (or higher)
- …

## Installation

The project does include libraries as submodules. You should clone with `--recursive` to check them out as well.

```bash
git clone --recursive https://gitlab.tu-ilmenau.de/path/to/project-name.git
```

## Startup

The program code is located in the directory `src`. The main component of the program is `UIProjectName`.
~~~~

~~~
```matlab
UIProjectName()
```

## Known bugs

### Bug 1

Description of Bug 1.

## Documentation

The documentation is in the [`doc` directory](doc).

- [Setup](doc/setup.md)
- [Hardware](doc/hardware.md)
- Tools
  - [Extern plot](https://gitlab.tu-ilmenau.de/FakMB/QBV/topics/matlab/extern-plot/blob/master/doc/user-doc.md)
- [End user documentation](doc/user-doc.md)
- [File formats](doc/file-formats.md)

You can find a CAD model of the system in the [`hardware` directory](hardware).

### Examples

Some examples are located in the [`example` directory](example).

### Offline documentation

You can find PDF version on the platin server:

- Platin-Server (141.24.221.155)
- /Austausch/Gitlab-Bot/project-name/

The PDF version can also be downloaded here:

- [Download Manuel as PDF](https://gitlab.tu-ilmenau.de/path/to/project-name/-/jobs/artifacts/master/raw/doc/project-name-Manual.pdf?job=doc)
~~~

Library `README.md` template

Remove the unneeded parts and add project-specific ones if necessary.

````
# Project name

Overview description of the project.

## System requirements

- Matlab R2019a Update 3, 64 Bit (or higher)
- …

## Usage

### Include in your project

Look at our
[Matlab projects guide](https://gitlab.tu-ilmenau.de/FakMB/QBV/topics/compendia/project-structure/blob/master/doc/project-matlab.md#use-a-library)
for how to include it in your project.

### Documentation

The documentation is in the [`doc` directory](doc).

If it is a short description, you can place it directly here.

### Examples

Some examples are located in the [`example` directory](example).
````

Subdirectory `3rdparty`

Place foreign data (third-party data) here. These can be used libraries, configuration data, drivers or documentation of foreign components.

- 📁 `3rdparty`
    - 📁 `submodule1`
    - 📁 `...`

The subdirectory `3rdparty` should contain foreign data as Git submodules if possible.

If the foreign data is not in an accessible git repository, a hard copy can also be placed in a `3rdparty` subdirectory. Check whether it makes sense to create a separate Git project for the foreign data on our GitLab server. This is especially useful if the foreign data is used by several projects or if it is very large.

Note that foreign libraries and data used only by the unit tests should be located in the separate `test/3rdparty` directory.

Subdirectory `logo`

If your project has a logo, it should be in this directory. Preferably it should be an SVG file. If it is an SVG file, a rendered version as a PNG file with the dimensions `600x600px` should also be located there. This should also be set as logo in the project settings under General.

Note our hints about minimizing PNG file size.

Subdirectory `doc`

Any **documentation you created** should be located in the `doc` directory.

```
- doc
    - images
        - image1.png
        - image2.svg
        - ...
    - user-doc.md
    - file-formats.md
    - ...
```

The primary documentation should be named `user-doc.md`. Additional documentation should be in separate files. This applies for example to own file formats. E.g. the structure of Matlab `.mat` files to be loaded or saved, which are to be documented in the file `file-formats.md`. Images should always be placed in the `images` subdirectory.

Note our hints about image file formats.

Some examples can be found in the `doc` directories of our sample projects.

Subdirectories `src` or `lib`

If your project is an end-user software, your source code should be located in the `src` directory. If your project is a library that should be used by other Matlab projects, your source code should be located in the `lib` directory instead.

You should have *either* a `src` *or* a lib `directory`.

```
- src
    - AProgramClass.m
    - aProgramFunction.m
    - ...
```

- the Hyperspectral system is an example for an end-user project

External libraries do *not* belong in the `lib` directory, but in the `3rdparty` directory!

```
- lib
    - ALibraryClass.m
    - aLibraryFunction.m
    - ...
```

- the Mosaic tools are an example for a library project

The file names should match the CamelCase style. Classes start with a large letter, functions with a small letter.

If the project has a main component, it should have the same name as the project.

Some examples can be found in the `src` and `lib` directories of our sample projects.

Subdirectory `test`

The `test` directory should contain your unit tests. Necessary reference data should be located in appropriate subdirectories.

```
- test
    - testData
        - refData.mat
        - refImage.png
        - ...
    - 3rdparty
        - submodule1
        - ...
    - UnitTest1.m
    - ...
```

We recommend using class based unit tests.

- Class-Based Unit Tests
- Assertation functions

Some examples can be found in the `test` directories of our sample projects.

Especially for testing GUI classes our unit test library can be useful.

Subdirectory `example`

If the project is a library, it may be useful to store some examples for use. These belong in the subdirectory `example`.

If the example only consists of one Matlab `.m` file, it can be located directly in the `example` directory. If the example consists of several source files or needs additional data, it should be located in its own subdirectory.

```
- example
    - complexExample
```

```
        - 📁 data
            - 🗎 refData.mat
        - 🗎 example.m
        - 🗎 ...
    - 🗎 simpleExample.m
    - 🗎 ...
```

Subdirectory `hardware`

Project **data you created** for the hardware should be stored in the "hardware" directory. This could printed circuit boards (PCBs), computer-aided design data (CAD) or data sheets.

This does not apply to hardware documentation created by you. This belongs in the `doc` directory.

```
- 📁 hardware
    - 📁 pcb
    - 📁 cad
    - 📁 ...
```

If the data is very large (`> 20 MB`) it is better to place it in a separate project which might be referenced as Git submodule.

Subdirectory `script`

The `script` directory contains all data relevant for a continuous integration (CI) process.

```
- 📁 script
    - 🗎 setup.sh
    - 🗎 test.sh
    - 🗎 doc.sh
    - 🗎 ...
```

Important links:

- **Turn on** in your project settings
- **Git submodules**
- Convert **Markdown to PDF**
- Find the process **log-file**

To start a Continuous Integration process when pushing, a Runner must be enabled in the project settings. We have a guild about how to enable a GitLab-CI-Runner.

Now GitLab will process the file `.gitlab-ci.yml` in your project root directory with each push.

Typically, we define two processes for Matlab projects. The first executes the unit tests. The second converts the documentation from Markdown format to PDF.

The conversion to PDF is done to summarize the documentation in one file that can be displayed locally on a computer independently of GitLab. This is especially useful on computers without internet access.

You can copy the file content unaltered.

```
test:
  image: gitlab-registry.rz.tu-ilmenau.de/fakmb/qbv/topics/docker/matlab:R2019a
  before_script:
    - script/setup.sh
  script:
    - script/test.sh

doc:
  image: tuiqbv/tex-live:latest
  before_script:
    - script/setup.sh
  script:
    - cd doc
    - DATETIME=$(date '+%Y%m%d_%H%M%S')
    - ../script/doc.sh $DATETIME
  artifacts:
    paths:
      - $CI_PROJECT_DIR/doc/*.pdf
```

The Matlab image is stored on the GitLab-Server and created by out Matlab Docker image project. You may define multiple test sections to run your tests on different Matlab version. To do this, copy the test section and name `text` unique for each section (e.g. `test_2018b` and `test_2019a`).

The documentation conversion image is stored on Docker Hub and created by out Tex-Live project.

The `script/setup.sh` script enables the docker image to checkout all project submodules as well. We have a guild about it.

The `script/doc.sh` script will convert your Markdown files to PDF. We have a guild about it.

Write and use a library

A library is characterized by the fact that the files are not located in the `src` but in the `lib` directory.

We use Git submoduls to use a library in a project. The library should be included in the `3rdparty` directory. If the library is only used by the unit tests, it should be located in the `test/3rdparty` directory.

In Matlab, each function and class to be used in another file will be in a separate file. The file must have the same name as the function or class. To find the file, Matlab searches each directory in its search path for a file with the same name. The current directory is always searched first.

To detect name collisions directly when including a file, a function with the name scheme `verifyLibNameInclude.m` is created for each library.

Write a library

Add a file with the name `verifyLibNameInclude.m` to your `lib` directory, where the name part `LibName` is the name of your library. This function shall have the following content:

```
function verifyLibNameInclude()
    % extract path and name of this file
    [fn_path, fn_name, ~] = fileparts(mfilename('fullpath'));

    % verify this function is properly named
    lib_name = regexp(fn_name, "^verify([a-zA-Z_]+)Include$", "tokens");
    assert(isequal(size(lib_name), [1, 1]), ...
        "TUIQBV:InvalidVerifyFunctionName", ...
        "This is a bug in the included library, contact the author.");

    % get the paths and names of all m-files in the lib directory
    files = dir(string(fn_path) + filesep + "*.m");
    folders = string({files.folder});
    filenames = string(cellfun(@(name) name(1:end-2), {files.name}, ...
        'UniformOutput', false));

    % iterate over all files
    for entry = [folders; filenames]
        folder = entry(1);
        filename = entry(2);

        % concat the full file-path-name of the current file
        expected = folder + filesep + filename + ".m";

        % find the fill file-path-name file in matlabs search path
        found = which(filename);

        % verify both are identical
        assert(isequal(found, expected), ...
            "TUIQBV:IncludeLibrary:ExternPlot", ...
            "library include failed because which('" + filename + ...
            "') returned '" + found + "' instead of the expected '" ...
            + expected + "'.");
    end
end
```

Do not edit this code! (Except for the `LibName` part of the function name.) Just place it in your `lib` directory. It makes sure that none of the files in your lib directory is hidden by another file that is higher in the searchpath of the library user.

Use a library

To use a library, it is added to the `3rdparty` directory as a Git submodule.

```
cd 3rdparty
git submodule add https://gitlab.tu-ilmenau.de/path/to/lib-name.git
```

Changes to the library do not directly affect projects in which the library is used. If a commit has been added to the library, it must be explicitly adopted. To do this, the changes for the library must first be pulled from the server. Then the changed submodule directory is added as usual and made a commit.

```
cd 3rdparty/lib-name
git pull
cd ..
git add lib-name
git commmit -m "adopt lib-name changes"
```

To use functions from the library submodule, the `lib` directory must first be added to the search path.

```
addpath(fullfile(fileparts(mfilename('fullpath')), ...
    '../3rdparty/lib-name/lib'));
```

The actual path is given relative to the `src`/`lib` directory of your current project where you want to use the library. In our case it is `'../3rdparty/lib-name/lib'`.

If you change the current directory during the program execution or if you write a library where you use another library, the directory might no longer be relative to the `src`/`lib` directory. This can be fixed by converting the relative path to an absolute path. This is done by the following steps:

1. `mfilename('fullpath')` get you the name of the current file including its absolut path
2. `fileparts` will strip the filename from the path
3. `fullfile` will append your relative path to the absolute path of your current file

The resulting path is absolut and is added to the serach path by `addpath`.

If you include multiple libraries, add them all to the search path as described.

Then call the verify include function of each library to detect possible name collisions of library function.

```
verifyLibNameInclude();
```

If one of your own functions collides with a library function, rename your function. If two functions of different libraries collide, contact the authors and ask for a renaming.

Graphical user interface

The **use** of **AppDesigner components** is recommended for the implementation of graphical user interfaces (GUIs). It is recommended to ***not*** **use AppDesigner itself**.

In this way, the entire GUI can be implemented as text files. Changes are therefore easily traceable within the Git version history. The explicitly set properties are immediately traceable in the source code. This is not possible via the property editor of AppDesigner, which is why errors that occur due to differently set properties of a UI component are very difficult to analyze. Furthermore the implementation of the GUI can be distributed over several files.

Splitting a large GUI into independent components has many advantages. Each component is clearly defined, which makes the program code clearer. Each component can be tested individually. Components can possibly be used multiple times, which avoids code duplication.

AppDesigner components always start with an `ui`, for example `uifigure`.

The positioning of elements should always and strictly be done by `uigridlayout`. This ensures that the size of the window can be changed by the user. It also ensures that a change of the theme by an operating system or Matlab update or even an entirely different operating system can not destroy the GUI design.

- overview of all AppDesigner components
- how to use grid view

A simple example can be found in our extern plot libary and more compex ones are in the projects:

- Hyperspectral system
- Spectral reconstruction

All classes beginning with an `UI` are GUI classes.

Project examples that follow the guidelines

In the following some of our projects are listed, which implement the conventions described here.

End-user software

- Hyperspectral system
- Spectral reconstruction

Libraries

- Extern plot
- Mosaic tools
- BIG I/O
- Unittest