Faculty of Informatics, Masaryk University in Brno
Centre for Biomedical Image Analysis

FTutor1D

Technical overview

October 11, 2016

CBIA
CENTRE FOR BIOMEDICAL
IMAGE ANALYSIS

Contents

1 Introduction

If you have not read the FTutor1D User Manual, please do so. It is expected that you know what is the purpose of this software and how to use it. The document shall provide an overview on how the FTutor1D is designed and programmed. It is not a replacement for the comments in the source codes. On the contrary, it is a supplement for a programmer to show him how the application is structured, what are the resposibilities of all classes and where to look for something specific. The most important bits are commented in this document, but descriptions for each method must be searched in the source codes. However, one can use Doxygen to generate the documentation from the source codes and read in form of the PDF document or HTML page.

The first chapter provides information on assembling the source codes into a working executable and lists the requirements on preinstalled software. The second chapter provides the overall view of the application and briefly states the resposibilities of each class. All of the following chapters are dedicated each to a signle class. The classes descriptions are ordered alphabetically.

2 Building the application

To build the application from the source codes, download and install the Qt framework. Qt is available on `https://www.qt.io/download/`. You can use the Qt for private use and open-source distribution free of charge. If you want to download the single packages using the repository (common way in most of Linux distributions), download the following (with all dependencies): `libqt5core5a, libqt5widgets5, libqt5printsupport5, libqt5xml5, libqt5xmlpatterns5-dev`. You also need to have a C++ compiler and either *cmake* or *qmake* (packages `cmake` or `qt5-qmake`).

Open the command line, navigate to the root directory with the source codes and type the following:

1. `qmake FTutor1D.pro` or `cmake .`
2. `make`

In case the step one failed, provide the path to the missing libraries, or install the missing packages.

2.1 Compulsory directory structure

You must preserve the directory structure as listed in Figure 1 to have FTutor1D run correctly.

```
root
├── FTutor1D.exe
├── config.ini
├── // runtime dlls
├── platforms
│   └── // QT platform dependent dlls, necessary for users without proper Qt installation
├── resources
│   ├── // the icon should be there
│   └── signals
│       └── // all signal database files, must keep the original names
└── translation
    ├── en.xml
    └── // other translations
```

Figure 1: Directory tree for FTutor1D.

The `resources` folder is the folder where all the predefined signals and images are stored, the `translation` folder is where the language versions are loaded from. The `platforms` folder is important only for Windows users without Qt installation. Executable name, *FTutor1D.exe*, is the file to run, it should be called the same on Linux, just without the

.exe suffix. You probably have *config.ini* file missing. This is OK, it is generated after you first run the application. It searches for the resources at the predefined paths and puts those paths into the *config.ini*. If you do not want to keep the described directory structure, open *config.ini* file in a text editor and change the paths to the application icon, to the folder with translations and to the predefined signals. However, modified *config.ini* must still accompany the executable in the directory so that it could be loaded.

3 Design overview

The application is currently split into 10 classes. Eleventh class, QCustomPlot, was taken from `www.qcustomplot.org` and comprises only minor changes. The diagram in 2 illustrates the relationships between the classes.

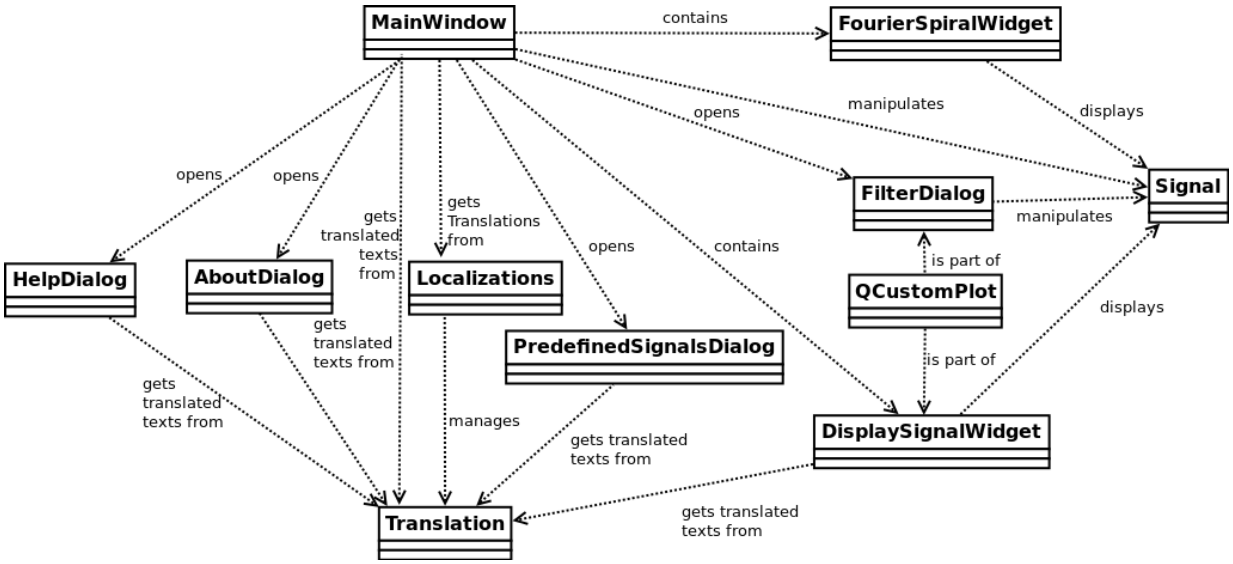

Figure 2: Diagram with classes and their relationships.

The application logic and the window classes are not explicitly splitted. Most of the interactions are bound to a certain window callback. The `MainWindow` is an object, from the user's perspective, that controls basically everything what happens in the application. Application design conforms that. The `MainWindow` contains the signals to be displayed and edited, loads the settings and translations and calls the other windows. Other windows are used either just for displaying some text (`AboutDialog, HelpDialog`), or provide the `MainWindow` with something (`PredefinedSignalsDialog` - file to load, `FilterDialog` - filtered magnitude signal to display). `DisplaySignalWidget` and `FourierSpiralWidgets` are UI elements implementing the signal view and its interactions.

The only classes, which are separed from the UI, are `Signal, Localizations` and `Translation`. `Signal` implements a representation of 1D signal and provides all the necessary methods to work with it. Special methods implemented there are the ones responsible for the forward and inverse Fourier transform. `Localizations` manage `Translation` objects, keeping list of languages and information about currently loaded language. The `MainWindow` must always ask the `Localizations` to provide it a language version. `Translation` class implements a language version itself. The `Translation` objects that the `Localizations` have provide translations for the whole application. Nevertheless, `Translation` provides

methods to get a `Translation` for a specific part of the application (window, widget, ...). One can obtain the localized text only when having a `Translation` corresponding to the specific UI element.

4 AboutDialog class

The `AboutDialog` displays the information about the application and its author. It is recommended to keep this dialog unchanged in the future versions, or to preserve at least all the information contained. `labelAuthorName` and `labelOfficialWebsite` are set to open external links. The action itself is realized using the HTML tag `<a href="...">`. `AboutDialog` implements

```
void setDefaultTexts();
void setLocalizedTexts(const Translation* language);
```

This is common for the all windows and some widgets in FTutor1D, that methods of such names are resposible for setting the window (widget) texts. For the sake of keeping this document short enough, this information will be omitted in the following chapters.

5 DisplaySignalWidget class

`DisplaySignalWidget` is an extensive class, which contains a `QCustomPlot` to display a signal and defines all the event callbacks, which are not enabled by default in the `QCustomPlot`. Interactions for the edit mode and "normal" mode are distinguished, as they are defined separately. The composition of `QCustomPlot` was chosen preferred to inheriting the class. The idea was to make it easier to extend the widget with additional widgets, such as scrollbars or labels (for example in the first stages of the development, the widget contained scrollbars).

The most important methods are:

- `void displaySignal(Signal* signal, bool shadowPrevious = false)`

  - displays new signal in the widget. If `shadowPrevious` is true, the previously displayed signal will be visible in grey colour until the next zoom or viewport translation

- `void setSibling(DisplaySignalWidget*& other);`

  - sibling is a `DisplaySignalWidget` object, whose displaying area shall be zoomed or translated together with this object's displaying area

- `void setInteractionsEnabled(bool val);`

  - if `val` is false, the zooming, scaling or anything else will be inactivated

- `void placePlotBackground(QCPItemRect*& section);`

  - the argument is the object representing the green background of the original part of the signal. The size of it and the position is given by the length of the original signal, and whether the centering is enabled or not.

- `bool event(QEvent* e) override;`

  - overriding of `QWidget` events. This is always executed before the actual callbacks, so you can modify the default events or define your own.

The slots defined by this class are:

- `void plotDefaultScale();`

  - rescales the graph to appropriately fit the area dedicated to `DisplaySignalWidget`

- `void displayWithLines(bool value);`

  - toggles displaying with lines

- `void enableCentering(bool enabled);`

  - toggles centering

The signals, which the class emits, are:

- `void needFrequencyUpdate(int idx, double value);`

  - mouse was moved to a new x coordinate and so the basis function plot must be updated

- `void needUpdateFiltered();`

  - a value in the magnitude or phase graph was edited and so filtered graph must be updated

- `void editModeNeedUpdate();`

  - need to replot all the plots so that they will correspond with edited original signal

- `void callForSaveState();`
  `void callForSaveEditModeState();`

  - save the current situation so that the user can return to this point using the `Undo` action in the menu

- `void openEditMode();`

  - open the edit mode

- `void displayValueStatusBar(int x, int index);`

  - display value at position `x` in the status bar. `index` is the order of the value at position `x`

- `void mouseLeave();`

  - mouse left the widget, clear the status bar and possibly the selected frequency

Together with `DisplaySignalWidget`, the enumeration `DisplaySignalWidgetType` is defined. It is used in the constructor of the `DisplaySignalWidget` object. Values it can take on are: `ORIGINAL, FILTERED, MAGNITUDE, PHASE, EDIT_MODE`. `MAGNITUDE` and `PHASE` force the frequency spectrum axis labels, otherwise the widgets works in the time domain. `EDIT_MODE` has separate interactions allowing adding points to the graph, other widget types differ only in the amount of allowed interactions. `ORIGINAL` and `FILTERED` allow no signal editing, but `ORIGINAL` has an option to open the edit mode. `MAGNITUDE` and `PHASE`, on the other hand, have an option to edit the displayed signal. Distinguishing the two frequency spectrum usages allows to restict the interval of possible values for the points.

6 FilterDialog class

`FilterDialog` is a window, which enables user to set the filter properties and apply the filter. It is programmed as a single window, which can work in different modes based on the type of filter that it represents.

Type of the filter is controlled by the `FilterType`, which is a parameter of the constructor. The available filters are: ideal low, high and band pass filters, Gaussian low and high pass filters and Butterworth low and high pass filters.

The important methods are:

- ```
  void initLowPass();
  void initHighPass();
  void initBandPass();
  void initGaussianLowPass();
  void initGaussianHighPass();
  void initButterworthLowPass();
  void initButterworthHighPass();
  ```

  - *init* prefixed functions create the layout of the window for the specific filter

- ```
  void ilpfGraph(int omega0);
  void ihpfGraph(int omega0);
  void bpfGraph(int omega1, int omega2);
  void glpfGraph(int omega0);
  void ghpfGraph(int omega0);
  void blpfGraph(double omega0, int n);
  void bhpfGraph(double omega0, int n);
  ```

  - *Graph* suffixed functions plot the visualisation of the filter in the `QCustomPlot`

- ```
  void idealLowPassFilter(int omega0);
  void idealHighPassFilter(int omega0);
  void bandPassFilter(double from, double to);
  void gaussianLowPassFilter(double omega0);
  void gaussianHighPassFilter(double omega0);
  void butterworthLowPassFilter(double omega0, int n);
  void butterworthHighPassFilter(double omega0, int n);
  ```

  - *Filter* suffixed functions create the full-length filter and apply it on the magnitude signal (which is as stored in the dialog as one of the atributes). This is basically an implementation of *OK* button click.

The *MainWindow* is notified that the filter was applied by accepting the dialog. Rejecting
  ```

is regarded as clicking the *Cancel* button.

FilterDialog is the only class that currently works with `<UseCase>` tags in the *Translation*. The reason is the fact that the window is reusing the same UI elements for the different filters.

7 FourierSpiralWidget class

The `FourierSpiralWidget` class is a blank widget, on top of which the Fourier spiral, the visualisation of Fourier basis functions, is drawn using `QPainter`. `FourierSpiralWidget` provides no interactions, it only responds to the change of its settings.

The coordinate system, into which is the scene drawn, is illustrated in the Figure 3.

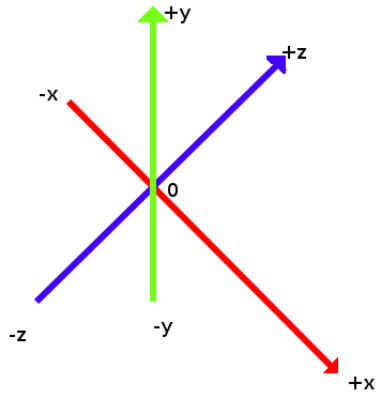

Figure 3: Coordinate system.

In the class attribute `projection`, the transform matrix from the coordinates in 3 to the widget coordinates is defined. This part of the widget might require rewriting. The engineering approach was chosen when programming it for the first version. Currently, the projection matrix is fixed and the plot fits the widget, if the coordinates of x are limited to the interval $[-1, 7]$, the coordinates of y to the interval $[-2.5, 2.5]$ and the coordinates of z to the interval $[-2.5, 2.5]$. To display the points out of these intervals, the linear interpolation is currently used, which scales the requested interval into these intervals.

The important functions of this class are:

- `void displayFrequency(...);`

  - display the basis function with given properties

- `void setMagnitudeAndPhase(double mag, double pha);`
  `void newSignal(int length);`

  - update attributes of the signal. The second function updates the length.

- `void paintEvent(QPaintEvent *event) Q_DECL_OVERRIDE;`
  `void drawAxes(QPainter& painter);`

```
void drawBackground(QPainter& painter);
void drawTexts(QPainter& painter);
```

- Methods responsible for drawing. The three *draw*-prefixed functions together draw only the basic coordinate system without any basis function. The basis function and its projections onto the real, imaginary and time plane are added directly in the `paintEvent`.

8 HelpDialog class

The `HelpDialog` class is a dialog, which displays the help message. Currently the class offers no advanced interaction.

The way of displaying the help text in `QTextBrowser` is quite untypical, as it is formatted using the HTML. The text is the concatenation of the HTML header, the text paragraphs defined by the translation file and the HTML footer.

9 Localizations class

The `Localizations` class holds a list (key-value map) of all available localizations and controls which localization is currently enabled. This list was created by scanning the translations directory and adding all the well-formed XML files containing the root tag `<FTutor1DLocalization>`.
Important functions of `Localizations` class:

- `QList<QString> getAvailableLanguages() const;`

  - returns the list of available language names

- `bool setLanguage(const QString& language);`

  - set the current language to the given value. Return value indicates whether a language of given name exists or not

- `Translation* getCurrentLanguage() const;`

  - returns the `Translation` object corresponding to the currently set language. The pointer returned is not newly allocated.

10 MainWindow class

The `MainWindow` is the heart of the application. This is the object that is created as the first, displays the first and creates all the other objects. Basically everything the window does is implemented inside the event callbacks.

The `MainWindow` has many attributes, which are mostly the UI elements. Except those, there are six `Signal`s (`original`, `magnitude`, `phase`, `filtered`, `editMode`, `prevOriginal`). While the usage of the first five is obvious, the `prevSignal` is a variable used to store the original signal when opening the edit mode, to enable going back to it if the user discards the edit mode changes. There is also `QSettings` instance called `settings`, which is responsible for loading and storing the configuration from the *config.ini* file. Loading is a part of `MainWindow` constructor and storing a part of the destructor. There is a `Localizations` object called `localization` for managing the translations. Finally, there are two signal stacks, `history` stack storing edits of `magnitude` and `phase` signals and `editModeHistory`, storing the changes in the `editMode`.

One of the most important parts of the `MainWindow` is the constructor. Loading settings, creating widgets, setting callbacks, reading translations and setting one of them, all of that is implemented there. Most of the rest methods implement only the reactions to some user actions and many of the callbacks are anonymous hidden in the constructor. I believe that they are pretty self-explanatory. The only method I want to comment is

```
void undo();
```

This method must take into account whether the edit mode is enabled, and in such case take values from the `editModeHistory`. If it is not enabled, it works with `history`. If a new signal is loaded, the history stacks are reset. If edit mode was canceled, the history of edits of frequency domain is still available. Note that the pointers in the stacks are allocated when adding the items and must be deleted in `undo` (or in `emptyHistoryStacks()`).

11 PredefinedSignalsDialog class

The `PredefinedSignalsDialog` class represents the dialog, which serves for loading signals from the predefined signals library. The necessary files are searched in *resources/signals* folder (see section). Even if the path is changed and reconfigured in `config.ini`, the files inside must keep their original names.

There is no intersting logic in this class. The only thing to note is that, compared to the `FilterDialog`, this class uses a signal

```
void signalChosen(QString resourcePath);
```

to pass the path to the chosen predefined signal file to the `MainWindow`. Which button opens which signal file is hardcoded.

12 Signal class

The `Signal` class is a representation of real 1D signal. The main part, called `original`, is saved in a `QMap`, where points' x coordinates are the keys and y coordinates are the values. All computations over the signal are performed with values stored there. There are also two `QVectors`, `extended_x` and `extended_y`, which are the coordinates of points to display. The programmer should make sure these vectors contain at least the copy of the data stored in the `original` map. More copies can be stored there, as the Fourier transform has the effect of replicating the band-limited signal.

The class methods mostly access the `original` or `extended_*` data structures and return some information about the data. The ones that modify the signal are:

- `void extend_left();`
  `void extend_right();`

  - add one copy of the original signal to the left or to the right (from `original` map to `extended_*` vectors)

- `void shrink_left();`
  `void shrink_right();`

  - remove one copy of the original signal from the left or from the right – this is currently not used

- `void reset();`

  - clear `extended_*` vectors and add only one copy of the original data to them

- `void updateAll(int index, double value);`

  - change original signal `index`th value and all its copies in the `extended_y` vector

- `void clear();`

  - delete all points from both the `original` map and `extended_*` vectors, leaving the signal empty

There are three special methods:

- `Signal applyFilter(Signal& filter) const;`

  - applies the `filter` signal to this signal. The application of the signal is realized by the point-wise multiplication. The result is stored in the new signal, which is finally returned. The lengths (`original` length) of both signals must be equal, otherwise an empty signal is returned.

- `static void fourierTransform(Signal& input, Signal& magnitude, Signal& phase);`

- computes the Fourier transform of the input signal, storing magnitude and phase signals. Fast Fourier Transform algorithm is used to accelerate the computation.

- `static void inverseFourierTransform(Signal& magnitude, Signal& phase, Signal& output, QVector<double> x = QVector<double>());`

  - computes the inverse Fourier transform using the magnitude and phase signals, storing the real part of the result as the output. Fast Fourier Transform algorithm is used. Parameter x can be used to assign x-axis coordinates to the resulting signal. The reason of providing such option is that the Fourier transform works with positions instead of real coordinates, so if the original coordinates were different than integers that are set in `magnitude` and `phase` signals and it would not be possible to recover them. If parameter `x` is an empty vector or its length differs from the length of the signal, positions are used anyway.

It can be shown that the k-th coefficient in the Fourier transform can be computed using the following formula:

$$\begin{aligned} F(k) =& F^0(k \mod m) + \psi^k \cdot F^1(k \mod m) + \psi^{2\cdot k} \cdot F^2(k \mod m) + \\ & \cdots + \psi^{(j-1)\cdot k} + \cdot F^j(k \mod m) \end{aligned}$$

In this formula, we asume the signal length n is divisible by j and so we divided the signal into j parts, each having m positions. The first part contains the positions which are divisible by j, the second part contains the positions which have a remainder from division by j equal to 1, ..., the last part contains the positions which have a remainder from division by j equal to $j-1$. F^i denotes the Fourier transformed part i. $\psi = e^{\frac{-2\pi i}{n}}$.

This formula is used by the Fast Fourier transform algorithm, which essentially recursively splits the signal into smaller parts until hitting the one sample signal. Then, itassembles the parts back while steppping out of the recursion. The inverse algorithm is basically the same, the only change is that $\psi = e^{\frac{2\pi i}{n}}$. In the implementation used in FTutor1D, the number of parts is chosen as the smaller integer divisor of the length of the input signal, which is greater than 1.

The recursive routines are performed by the following functions:

```
QVector<complex> fft_recursion(QVector<double>& input);
QVector<complex> ifft_recursion(QVector<complex>& input);
```

The routines are managed by the following functions:

```
QVector<complex> fft(QVector<double> input);
QVector<double> ifft(QVector<complex> input);
```

They convert the vectors of real (or complex) values into required format. Also, normalization by $\frac{1}{\sqrt{N}}$, where N is the length of the input signal, is done there. The main functions, `fourierTransform()` and `inverseFourierTransform()` in fact prepare the vectors of real or complex numbers from the input, call `fft()` or `ifft()`, and convert the result back into signals.

13 Translation class

The `Translation` class provides the information loaded from the localization XML file. The root `Translation` object, containing the localization for the whole application, is created and maintained by the `Localizations` class. However, for example to obtain a text for a single `QLabel`, it is necessary to obtain a subset of the root `Translation` and finally, only after having the `Translation` for the user interface element (`<UIElement>`) representing this label, the text can be retrieved.

Most important functions:

- `Translation* getTranslationForWindow(QString& windowName);`

  - obtain the `Translation` for the window of given name

- ```
  Translation* getTranslationForElement(QString& elementName);
  Translation* getTranslationForElement(int id);
  Translation* getTranslationForUseCase(QString& name);
  ```

  - obtain the `Translation` for the specific element or use case

- ```
  QString getTitle() const;
  QString getText() const;
  ```

  - obtain the text or title in the `<UIElement>` to which this `Translation` refers to

- ```
  QString getChildElementText(const QString& elementName) const;
  QString getChildElementText(const int elementIndex) const;
  ```

  - `getTranslationForElement()` and `getText()` in one function call

Note that the new `Translation` objects are dynamically allocated and returned as pointers, so their memory must be deleted later. Internally, the document is represented as a DOM tree using `QDomDocument` (`QDomElement` and the other related classes respectively). This class requires `Qt5Xml` packages to be installed and linked.
  ```

14 QCustomPlot class

The `QCustomPlot` is a class, which serves for drawing the 2D graph and provides some basic interactions, such as zooming, dragging, etc. It was reused from the project of Emanuel Eichhammer. It is available for free on `http://www.qcustomplot.com`. However, the class had to be modified to allow some additional callbacks, such as `mouseLeave()`. *Friend* access had to be granted for `DisplaySignalWidget` in several `QCustomPlot` classes. Because of this fact, it is not recommended to upgrade the `QCustomPlot` to a newer version, but rather maintain the old one.