# UnifiedGUI
# Documentation

Benjamin Schiller
benjamin.bs.schiller@fau.de

November 11, 2022
Version: v1.0.0

## Contents

## 1 Summary

UnifiedGUI is an abstract implementation of a graphical user interface which can be used for a variety of different purposes in the context of molecular communication.

## 2 Development

This project is still in its early stages of development and can therefore change heavily in the near future. The list of ideas for features is long. Also, the project may still contain bugs.

Still, if you encounter any bugs or problems or have a feature request, feel free to open an issue on GitHub or write an e-mail (`mailto:benjamin.bs.schiller@fau.de`).

## 3 Motivation

The goal of UnifiedGUI is to avoid redundant implementation of features that are required to visualize a transmission/receiving scheme in molecular communication. When trying out a new way of communication, the user should not have to worry about the visualization and data handling, but only about the hardware itself and the communication to it.

To create a GUI that can be used for a variety of different applications, it requires for a very abstract and dynamic implementation – for example the number of receiver and therefore the number of tables/datalines in the plot is only known at runtime. Because of this, UnifiedGUI may look a bit complicated at the first glance. This document offers a detailed description of the software and hopefully provides a comprehensive understanding.

## 4 Structure

As mentioned in Section 3, the implementation is done in an abstract way to allow for high reusability.

There are exactly four classes that can be implemented for the user's purposes: encoders, transmitters, decoders and receivers. The only files the user should create/edit are the concrete implementations (in the directory `Models/Implementations`).

A model contains up to one encoder and up to one decoder.

- An encoder consists of $n$ transmitters. An encoder also provides an encoding scheme that transforms a sequence (e.g., a string) to symbol values that can then be transmitted.

- A transmitter is used to control the input to a communication channel. For instance, a transmitter may regulate the pressure of a pump or the amount of background flow in the case of fluids.
- A receiver is used to generate sensor data from a communication channel. Each receiver can have multiple sensors where each sensor stores the measured values as a floating point value. For example, a receiver can be a color sensor, where the multiple sensors are the measured colors.
- A decoder consists of $m$ receivers. In the simplest case, the decoder just collects data from the receivers and stores them. A decoder may also provide more functionality, as it allows for further processing to retrieve information from the data generated by the receivers.

Note that the user can individually choose what parts they want to implement. For instance, if the user is just interested in visualizing data, implementing a receiver and simple decoder may be sufficient and an encoder does not have to be defined.

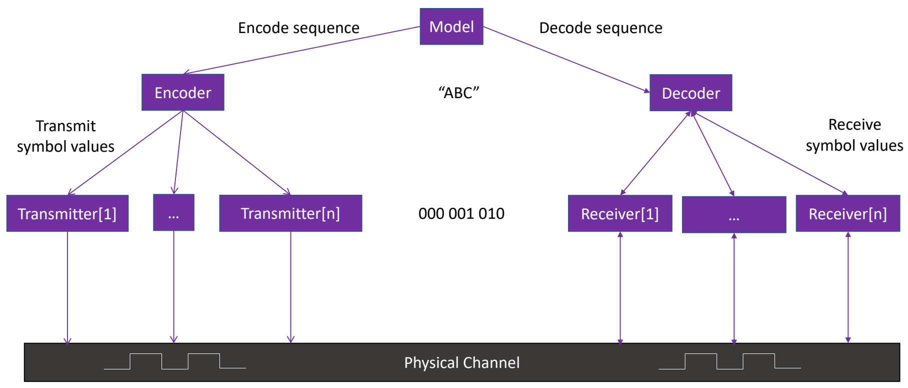


Figure 1: A model consists of three abstraction layers (sequence level, symbol level and physical level). The model can have an encoder, which can consist of multiple transmitter which transmit the information onto a physical channel. The model can also have a decoder, which can consist of multiple receivers which read data from a physical channel.

The model consists of three different levels of information: sequence level, symbol level and physical level (see Table 1). An flow through the different levels could look like this:

1. **Sequence level**: Some information should be transmitted, let us assume a string in this case.

2. **Symbol level**: The information is transformed into a sequence of bits. In this case, each character of the string is stored as a 7-bit ASCII encoding. In this step, we could also add parity bits and other meta-data for the transmission.

3. **Physical level**: The transmitters are set accordingly to transmit the bit sequence. For example, a pump transmits some fluid representing different bits.

4. **Physical level**: The receivers measure some physical property using sensors, in this case, let us assume color sensors.

5. **Symbol level**: Using the physical values picked up by the receivers, the bit sequence is restored.

6. **Sequence level**: The bit sequence is transformed back to a string using the ASCII encoding scheme. If everything went fine, the message is the same one as the one in Step 1.

| Level | Representation | Example |
|---|---|---|
| Sequence | Non-primitive data type | A |
| Symbol | Integer (often binary) | $0, 1$ |
| Physical | Float | $1.0, 1.1, 1.5, 0.9, ...$ |

Table 1: This table gives an overview of the three levels of information present in the model. On the bottom level, data is transmitted/received over some physical medium. On the middle level, data is represented as a list of integers (typically bits, but also more than two symbol values are possible). On the highest layer, multiple symbol values combined yield a high-level representation (such as a string containing a message).

# 5 How To Use

## 5.1 Requirements

1. Download and install the latest version of Python3 (`https://www.python.org/downloads/`).

2. Install the following packages using `python3 -m pip install <package>`.

   - `numpy`
   - `PyQt5`
   - `pyqtgraph`

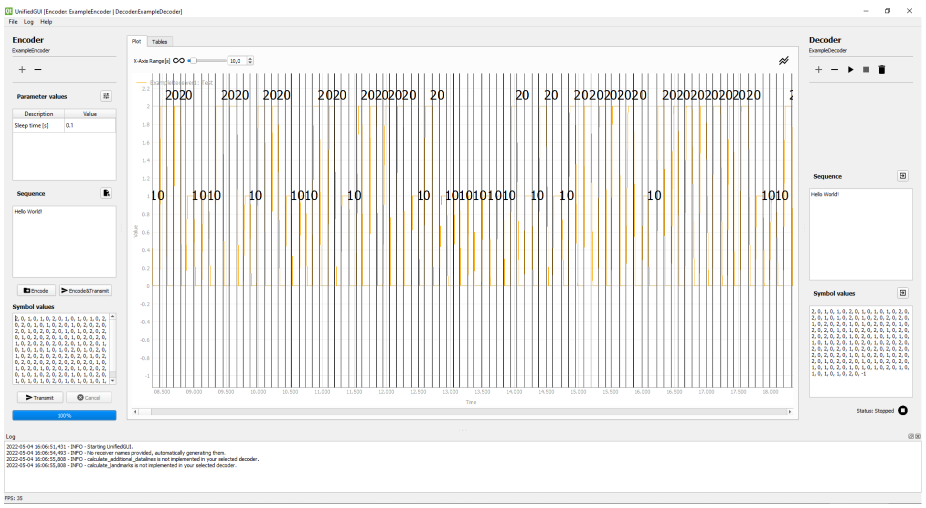
## 5.2 Execution

UnifiedGUI can be run via `python3 UnifiedGUI.py` in the terminal (make sure you are in the correct directory).

Alternatively, you can run the program with the `-O` (the capital letter) flag. This will generate an 'optimized' mode which might be slightly faster. However, **no debug information** is shown in this mode which can make it hard to spot implementation errors!

Especially for debugging purposes, an IDE like PyCharm is recommended.

## 5.3 Using UnifiedGUI



| Description | Value |
|---|---|
| Sleep time [s] | 0.1 |

Sequence:

Hello World!

Symbol values:

```
2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, 0, 2,
0, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 2, 0, 2, 0,
2, 0, 1, 0, 2, 0, 2, 0, 1, 0, 1, 0, 2, 0, 2,
0, 1, 0, 2, 0, 2, 0, 1, 0, 1, 0, 2, 0, 2, 0,
1, 0, 2, 0, 2, 0, 2, 0, 2, 0, 1, 0, 2, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 2, 0, 1, 0, 2, 0,
1, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 1, 0, 2,
0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 1, 0,
1, 0, 2, 0, 1, 0, 2, 0, 2, 0, 1, 0, 2, 0, 2,
0, 1, 0, 1, 0, 2, 0, 2, 0, 1, 0, 1, 0, 2, 0,
1, 0, 1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, 0, 1,
```

Decoder Sequence:

Hello World!

Decoder Symbol values:

```
2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, 0, 2, 0,
2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 2, 0, 2, 0, 2, 0,
1, 0, 2, 0, 2, 0, 1, 0, 1, 0, 2, 0, 2, 0, 1, 0,
2, 0, 2, 0, 1, 0, 1, 0, 2, 0, 2, 0, 1, 0, 2, 0,
2, 0, 2, 0, 2, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, 0,
1, 0, 1, 0, 2, 0, 1, 0, 2, 0, 1, 0, 2, 0, 2, 0,
2, 0, 2, 0, 2, 0, 1, 0, 2, 0, 2, 0, 2, 0, 2, 0,
2, 0, 2, 0, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 2, 0,
2, 0, 1, 0, 2, 0, 2, 0, 1, 0, 1, 0, 2, 0, 2, 0,
1, 0, 1, 0, 2, 0, 1, 0, 1, 0, 1, 0, 2, 0, 1, 0,
1, 0, 1, 0, 1, 0, 2, 0, -1
```

Log

```
2022-05-04 16:06:51,431 - INFO - Starting UnifiedGUI.
2022-05-04 16:06:54,493 - INFO - No receiver names provided, automatically generating them.
2022-05-04 16:06:55,808 - INFO - calculate_additional_datalines is not implemented in your selected decoder.
2022-05-04 16:06:55,808 - INFO - calculate_landmarks is not implemented in your selected decoder.
```

FPS: 35

Figure 2: Screenshot from UnifiedGUI.

Once you started the program, you see everything regarding the encoders on the left side, the live plot/tables in the middle, and everything regarding the decoders on the right side (see Figure 2. Add decoder/encoders by clicking on the + symbol and remove them by clicking on the – symbol.

Start/stop decoders by clicking on the Play/Stop buttons respectively. Clear everything from the live plot by clicking on the trash bin icon. Export decoded symbol values or sequences as `*.txt` or `*.csv` by clicking on the export button (native) above the respective text edits. You can also define a custom export scheme (see Section 5.8 for more information about this).

For the encoders, either enter a sequence directly into the text edit or load one from a file: you can store sequences as `*.txt` files in the directory `Sequences`. Afterwards, either click on the Encode or the Encode&Transmit button. You can also enter symbol values directly into the symbol values text edit. Pressing Transmit will transmit the symbol values currently stored in the symbol values text edit. Pressing the Cancel button cancels an ongoing transmission.

## 5.4 Adding New Implementations

To add a new implementation, create a fork (a local copy) of the GitHub repository and push your changes into the forked repository. Once everything is finished and documented, you can create a pull request so your modifications can be integrated in the main repository.

For more information on forks and pull requests, have a look at the following resources:

- `https://www.atlassian.com/git/tutorials/comparing-workflows/forking-workflow`
- `https://www.atlassian.com/de/git/tutorials/making-a-pull-request`
- `https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests`

The following sections provide more information regarding the implementations.

## 5.5 Implement a New Transmitter

1. Choose a name for your receiver, e.g. "MyTransmitter".
2. In the directory `Models/Implementations/Transmitters`, create a new Python file with the **exact** name chosen in step 1, e.g. `MyTransmitter.py`.
3. Open the receiver template (`Models/Templates/TransmitterTemplate.py`). Select everything in this file and copy it.
4. Open the file created in step 2.
5. Paste the template copied in step 3.
6. Rename the class. Class name must be the **exact** name chosen in step 1.
7. Set the class attributes and implement the required functions as described below.
8. Once you finished your implementation, test it by running the provided testcases (`python3 TestUnifiedGUI.py TestTransmitters`, see Section 5.10 for more information).

**Required**:

- The function `transmit_step` runs a single step of a transmitter. This function will be regularly called by the encoder in a separate thread. If nothing needs to be done, simply return.

**Optional**:

- The member `allowed_sequence_values` can be used to define a list of allowed sequence values to avoid errors while trying to encode invalid sequence values.
- The member `allowed_symbol_values` can be used to define a list of allowed symbol values to avoid errors while trying to transmit invalid symbol values.
- The member `sleep_time` is a float value that represents the time in seconds between the calls of `transmit_step`, i.e., how often the transmitter will transmit new values. Note that due the Python 3 implementation of `time.sleep()`, this is very inaccurate. If, for some reason, you need a more precise timer, you will have to redefine the `transmit` function.

For reference, you can have a look at the already implemented example transmitter (`Models/Implementations/Receivers/ExampleTransmitter.py`).

## 5.6 Implement a New Encoder

1. Choose a name for your receiver, e.g. "MyEncoder".
2. In the directory `Models/Implementations/Encoders`, create a new Python file with the **exact** name chosen in step 1, e.g. `MyEncoder.py`.
3. Open the receiver template (`Models/Templates/EncoderTemplate.py`). Select everything in this file and copy it.
4. Open the file created in step 2.
5. Paste the template copied in step 3.
6. Rename the class. Class name must be the **exact** name chosen in step 1.
7. Set the class attributes and implement the required functions as described below.
8. Once you finished your implementation, test it by running the provided testcases (`python3 TestUnifiedGUI.py TestEncoders`, see Section 5.10 for more information).

**Required**:

- The member `transmitters` is a list that contains all transmitters belonging to the encoder.
- The member `sleep_time` specifies the time between the transmission of the individual symbols. It can either be a `float` value if a uniform sleep time between the symbols is desired, or a list of `float` values if non-uniform sleep times are required. For example, `self.sleep_time = [1.0, 2.0]` will result in alternating sleep times between 1 and 2 seconds after every symbol.
- The function `encode` takes as input a sequence and converts it to a list of symbol values which is returned.
- The function `transmit_single_symbol_value` is used to transmit a given symbol value.

**Optional**:

- The function `parameters_edited` can be used if something needs to be changed when the parameters are edited. This function will be called whenever the user edits the parameters of the encoder.

For reference, you can have a look at the already implemented example encoder (`Models/Implementations/Receivers/ExampleEncoder.py`).

**Hint**: You can also override the function `run_transmit_symbol_values` in your encoder implementation if you need more flexibility for transmitting the symbol values.

## 5.7 Implement a New Receiver

1. Choose a name for your receiver, e.g. "MyReceiver".
2. In the directory `Models/Implementations/Receivers`, create a new Python file with the **exact** name chosen in step 1, e.g. `MyReceiver.py`.
3. Open the receiver template (`Models/Templates/ReceiverTemplate.py`). Select everything in this file and copy it.
4. Open the file created in step 2.
5. Paste the template copied in step 3.
6. Rename the class. Class name must be the **exact** name chosen in step 1.
7. Set the class attributes and implement the required functions as described below.
8. Once you finished your implementation, test it by running the provided testcases (`python3 TestUnifiedGUI.py TestReceivers`, see Section 5.10 for more information).

**Required**:

- The member `num_sensors` is an integer that specifies how many sensors the receiver has.
- The function `listen_step` checks for new values and gets called periodically. If new values are available, they are appended as a tuple or a list to the member `buffer` (inherited by the interface) using the function `append_values` (inherited by the interface). Optionally, you can provide a timestamp corresponding to the time the measurement was generated. If you do not provide a timestamp, the current time is used as a timestamp. **Note**: If the receiver only has one sensor, it is still important to convert the measured value to a tuple or a list (of length 1). This can be done like this: `my_tuple = (value,)` or `my_list = [value]`.

**Optional**:

- The member `sensor_names` is a list of strings that provide a meaningful name for each sensor.
- The member `drop_first_measurements` is an integer value that indicates the amount of measurement values to be ignored in the beginning. This can be useful if a sensor takes some time to measure correctly.
- The member `sleep_time` is a float value that represents the time in seconds between the calls of `listen_step`, i.e., how often the receiver will check for new values. Note that due the Python 3 implementation of `time.sleep()`, this is very inaccurate. If, for some reason, you need a more precise timer, you will have to redefine the `listen` function.

For reference, you can have a look at the already implemented example receiver (`Models/Implementations/Receivers/ExampleReceiver.py`).

## 5.8 Implement a New Decoder

1. Choose a name for your receiver, e.g., "MyDecoder".
2. In the directory `Models/Implementations/Decoders`, create a new Python file with the **exact** name chosen in step 1, e.g., `MyDecoder.py`.
3. Open the decoder template (`Models/Templates/DecoderTemplate.py`). Select everything in this file and copy it.
4. Open the file created in step 2.
5. Paste the template copied in step 3.
6. Rename the class. Class name must be the **exact** name chosen in step 1.
7. Set the class attributes and implement the required functions as described below.
8. Once you finished your implementation, test it by running the provided testcases (`python3 TestUnifiedGUI.py TestDecoders`, see Section 5.10 for more information)).

**Required**:

- The member `receivers` is a list containing the the receivers belonging to the decoder. A decoder must contain at least one receiver.

**Optional**:

- The constant `PARAMETERS` can be used to specify additional parameters which the user has to provide values for upon adding the decoder and can later edit. See Section 5.9 for more information.
- Plot settings such as which datalines are shown, the color of the datalines or whether the symbol intervals are shown can be selected by the user during the execution of the program and are saved once the program is closed so they can be restored during the next execution (on the same device). You can provide default settings by setting the member `plot_settings` accordingly. `plot_settings` is a dictionary that supports the following keys:
  - `additional_datalines_active`: A list of `bool` values that specify which additional datalines are active.
  - `additional_datalines_colors`: A list of `string` values that specify which color each additional dataline has. Color values can be specified using a hex value.
  - `additional_datalines_style`: A list of `string` values that specify the style of each additional dataline. For a list of available styles, have a look at the Qt specification. Specify the style by its constant name without the Qt, for example `'SolidLine'`.

- `additional_datalines_width`: An `int` value that specifies the width of the additional datalines. Note that for some reason, a width larger than 1 often causes the live plot to be significantly slower.
- `datalines_active`: A list of lists of `bool` values that specify which datalines are active. First 'dimension' is the receivers while the second 'dimension' refers to the sensors.
- `datalines_color`: A list of lists of `string` values that specify which color each dataline has. First 'dimension' is the receivers while the second 'dimension' refers to the sensors. Color values can be specified using a hex value.
- `datalines_style`: A list of lists of `string` values that specify the style of each dataline. For a list of available styles, have a look at the Qt specification. First 'dimension' is the receivers while the second 'dimension' refers to the sensors. Specify the style by its constant name without the Qt, for example `'SolidLine'`.
- `datalines_width`: An `int` value that specifies the width of the datalines. Note that for some reason, a width larger than 1 often causes the live plot to be significantly slower.
- `landmarks_active`: A list of `bool` values that specify which landmarks are active.
- `landmarks_size`: An `int` value that specifies the size of the landmark symbols.
- `landmarks_symbols`: A list of `string` values that specify the symbol for each landmark set. Supported symbols can he found here.
- `show_grid`: A `string` value that indicates whether a grid should be shown in the live plot. Can have the values 'None', 'x-axis only', 'y-axis only', 'x-axis and y-axis'.
- `step_size`: An `int` value that can be used to achieve downsampling of the live plot data. A step size of $s$ means that only every $s^{\text{th}}$ value is drawn in the live plot which might result in better performance.
- `symbol_intervals`: A `bool` value that specifies whether symbol intervals are shown as vertical lines.
- `symbol_intervals_color`: A `string` value that specifies the color of the symbol interval vertical lines. Color values can be specified using a hex value.
- `symbol_intervals_width`: An `int` value that specifies the width of the symbol interval vertical lines. Note that for some reason, a width larger than 1 often causes the live plot to be significantly slower.
- `symbol_values`: A `bool` value that specifies whether symbol values are shown as text items.
- `symbol_values_fixed_height`: A `float` value that specifies the height for the symbol values. Only relevant if `symbol_values_position` is set to `'fixed'`.

  - `symbol_values_height_factor`: A `float` value that specifies the relative height of the symbol values above or below the plot. Only relevant if `symbol_values_position` is set to `'above'` or `'below'`. For example, `symbol_height_factor = 1.1` means that the symbol value is shown at 10% above or below the maximum value in this interval.
  - `symbol_values_position`: A `string` value that specifies where the symbol values are shown. Allowed values are `'above'`, `'below'` and `'fixed'`.
  - `symbol_values_size`: An `int` value that specifies the size of the symbol values.
  - `x_range_active`: An `bool` value that is True when the x range of the plot should be limited, otherwise False.
  - `x_range_decimals`: An `int` value that specifies the granularity at which the x range of the plot can be adjusted, e.g., `x_range_decimals = 3` allows for values like 0.123.
  - `x_range_min`: A `float` value that specifies the minimum x range of the plot.
  - `x_range_max`: A `float` value that specifies the maximum x range of the plot.
  - `x_range_value`: A `float` value that specifies the actual x range of the plot.

  **Important**: If you modify your decoders or receivers (such as change the number of sensors for the receivers), it may be necessary to reset the plot settings. You can do this by deleting the respective file in `Utils/PlotSettings/DecoderPlotSettings`. A new one will be generated the next time you add the decoder.

- Additional datalines can be used to visualize further information, e.g., derivates or means of the values from the receiver. If you want to use this, set the member `additional_datalines_names` accordingly (a list of string items). Implement the function `calculate_additional_datalines` that calculates the additional datalines and stores them in the member `additional_datalines`. Additional datalines are in the form of a dictionary containing the three keys `length` (`int`), `timestamps` and `values` (`numpy array`) (`numpy array`). Since multiple additional datalines are supported, `additional_datalines` is a list of dictionaries.
  **Note**: If you only want to display one additional datalines, make sure `additional_datalines` is a list of dictionaries (of length 1). This can be done like this: `my_list = [my_dataline]`.

- The function `calculate_landmarks` can be used to generate landmarks for edges, peaks etc. The landmarks can then be shown in the plot. If you want to use this, set the member `landmarks` accordingly. Landmarks are in the form of a dictionary with keys `x` for the position of the landmark on the x-axis (timestamp) and `y` for the position on the y-axis respectively. Since multiple landmarks sets are supported, `landmarks` is a list of dictionaries.

**Note**: If you only want to display one set of landmarks, make sure `landmarks` is a list of dictionaries (of length 1). This can be done like this: `my_list = [my_landmark]`.

- The function `calculate_symbol_intervals` can be used to divide the signal into different interval where each intervals corresponds to one symbol. The symbol intervals can then be shown in the plot as vertical infinite lines. If you want to use this, set the member `symbol_intervals` to a list of timestamps. For instance, symbol intervals can either be constructed from fixed time intervals (for example every $t$ milliseconds), from a fixed amount of samples (for example every $n$ samples) or from landmarks (edges, peaks, etc.).

  **Note**: The first symbol intervals begins at the first timestamp provided and the last symbol intervals ends at the last timestamp provided, resulting in a total of $n-1$ intervals where $n$ is the length of `symbol_intervals`.

- The function `calculate_symbol_values` can be used to assign a symbol to each symbol intervals. If you want to use this, first make sure to implement `calculate_symbol_intervals` correctly and set the member `symbol_values` accordingly (expects a list of strings). Also pay close attention where the first intervals starts and the last intervals ends. If implemented correctly, the length of `symbol_values` should be 1 smaller than the length of `symbol_intervals`.

- The function `calculate_sequence` can be used to decode the symbol values to a sequence.

- The function `decoder_removed` can be used if something needs to be done when the decoder is removed, e.g., close serial ports.

- The function `decoder_stopped` can be used if something needs to be done when the decoder is stopped, e.g., resets.

- The function `export_custom` can be used to used to configure some export scheme. The custom export can then be accessed in the File menu. Have a look at the basic export function provided in the interface that stores the data from the receivers as well as the additionally calculated datalines in `.csv` files.

- The function `parameters_edited` can be used if something needs to be changed when the parameters are edited. This function will be called whenever the user edits the parameters of the decoder.

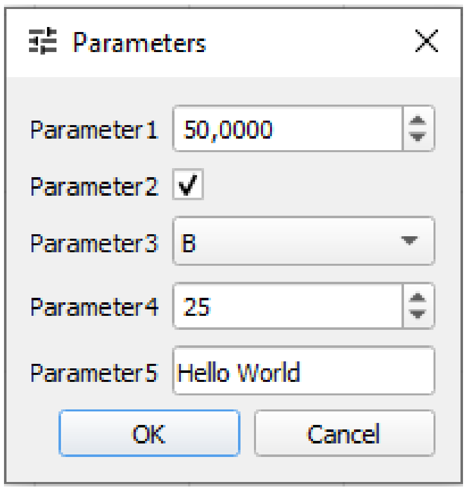
- The function `pre_processing` can be used if some pre-processing needs to be applied to the measurement values from the receivers, e.g., apply filters for smoothing.

For reference, you can have a look at the already implemented example decoder (`Models/Implementations/Receivers/ExampleDecoder.py`).

**Hint**: You can use the function `get_received(receiver_index, sensor_index)` to get measurement values for a given receiver and sensor. Set `sensor_index = -1` to get all measurement values of a receiver.

## 5.9 Parameters

Parameters can be used to specify additional attributes which the user has to provide values upon adding a decoder/encoder and can later edit as well.



Parameter1 50,0000
Parameter2 ✓
Parameter3 B
Parameter4 25
Parameter5 Hello World

Figure 3: Example parameters (`float`, `bool`, `item`, `int`, `string`).

Parameters are optional, if you do not want to use any parameters, let `get_parameters` return `None` in your implementation or simply do not define it in the first place.

Parameters are defined as a list of dictionaries where each dictionary represents one parameter. Every parameter dictionary must contain the following keys:

- `description` (`string`): Description of the parameter.
- `default`: Default value of the parameter.
- `type` (`string`): Datatype of the parameters = {`bool`, `int`, `float`, `string`, `item`}
- `editable` (`bool`): Whether the parameter can be edited after the first initialization.

Depending on the datatype, additional keys may be required:

- `bool`:
  - No additional keys required.
- `int`:
  - `min` (`int`): Minimum value.

    - `max` (`int`): Maximum value.

- `float`:
    - `min` (`float`): Minimum value.
    - `max` (`float`): Maximum value.
    - `decimals` (`int`): Number of decimals.
- `string`:
    - `max_length` (`int`): Maximum length.
- `item`:
    - `items` (`list`): A list of selectable `string` items.

For datatypes `float` and `int` you can optionally provide a conversion function that returns a `string` value that will get displayed in the parameter dialogue. You can use lambda functions, for example, `'conversion_function':lambda x:str(1/x) + "s"` will convert a frequency to seconds. You can also define regular functions and pass a reference to them (without the round brackets after the function name).

For reference, have a look at the parameters defined in the example decoder (`Models/Implementations/Decoders/ExampleDecoder.py`).

## 5.10 Testcases

To test your implementations, you can run the provided testcases using the following syntax:

`TestUnifiedGUI.py {TestTransmitters, TestEncoders, TestReceivers, TestDecoders} {file}`

You can run all the testcases by providing no further argument (`python3 TestUnifiedGUI.py`).

You can run testcases only for a specific type (e.g., `python3 TestUnifiedGUI.py TestDecoders`).

You can run testcases only for a single file (e.g., `python3 TestUnifiedGUI.py TestDecoders ExampleDecoder`.

## 6 Troubleshooting

- **Live plot is lagging or freezing.**

  For the visualization of the live plot, the `pyqtgraph` module is used which is designed to quickly update live data like in this context. However, for large amount of data, the live plot might still lag or freeze. If you experience this problem, try the following suggestions (sorted by expected effectiveness):

  1. **Hardware**: As simple as it may sound, check if you have access to a better computer you can use — the crucial component is most likely the CPU.
  2. **Datalines width**: For some reason, it makes a huge difference whether the width of the datalines is 1 or larger. Try setting it to 1 in the plot settings of your decoder.
  3. **Symbol intervals and symbol values**: Drawing symbol values (infinite vertical lines) as well as symbol values (text items) is very resource expensive and can quickly lead to the plot becoming laggy if they have to be drawn in large numbers. To reduce lag, consider disabling them in the plot settings.
  4. **Inefficient implementation**: Especially if the live plot becomes increasingly laggy over time, the reason might be due to an inefficient implementation in your decoder. Make sure that you do not calculate stuff again that has not changed, e.g., do not calculate symbol values for every interval again every step — only do it for the new intervals.
  5. **Downsampling using step size**: In the plot settings of your decoder, set step size to an `int` value larger than 1. A step size of $s$ means that only every $s^{\text{th}}$ value is drawn in the live plot.
  6. **Optimized mode**: You can run the program with the `-O` (the capital letter) flag. This will generate an 'optimized' mode which might be slightly faster. However, **no debug information** is shown in this mode which can make it hard to spot implementation errors!