qc-toolkit Documentation

Release 0.1

TODO: Author

April 15, 2016

Contents

Contents:

CHAPTER 1

Creating a New Pulse Template

This example goes through the workflow of creating a new type of pulse from scratch using the *TablePulseTemplate* class.

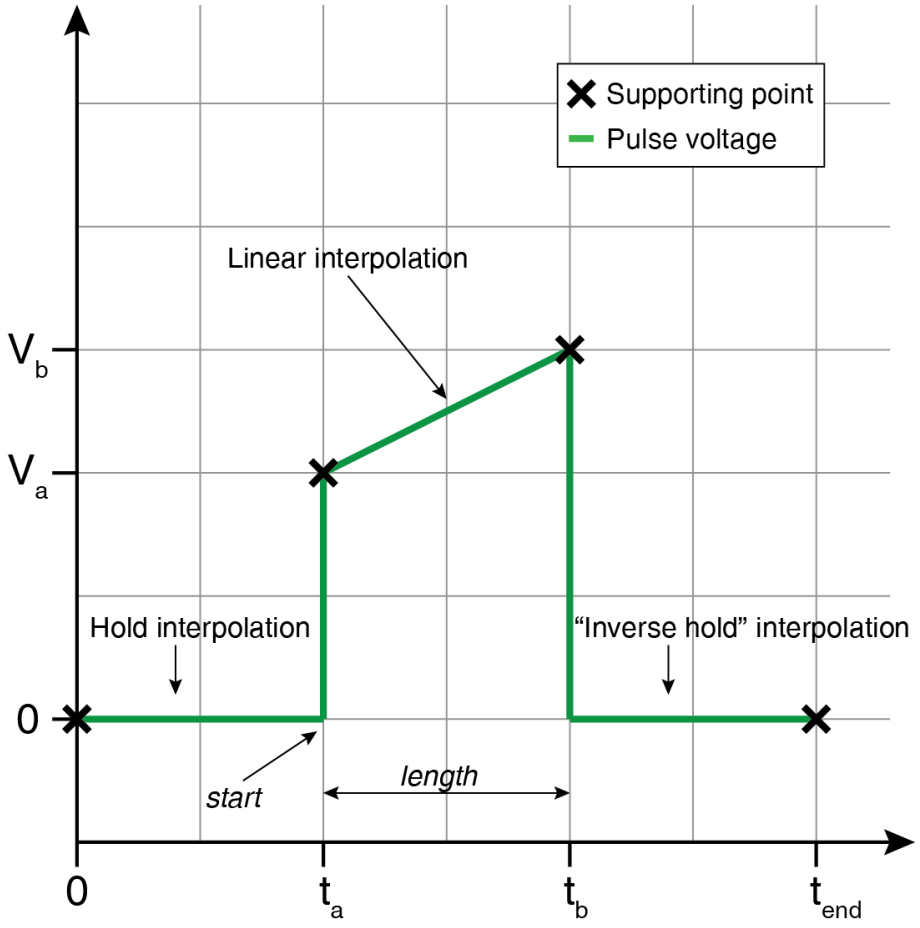

Fig. 1.1: Figure 1: Example pulse shape. The crosses mark the points that go into the table, they are connected using different interpolation strategies.

Figure 1 shows the pulse shape we want to build in this tutorial. The crosses mark the supporting points that are connected using different interpolation strategies. We want to parametrize the position of the supporting points using the time at which the voltage rises (t_a) as *start* and the length of the rise afterwards as *length*.

We start with an empty *TablePulseTemplate* object and add supporting points:

```
squarePulse = TablePulseTemplate() # Prepare new empty Pulse
# Then add pulses sequentially
squarePulse.add_entry(0, 0)
squarePulse.add_entry('ta', 'va', interpolation='hold') # hold is the standard interpolation value
squarePulse.add_entry('tb', 'vb', interpolation='linear')
squarePulse.add_entry('end', 0, interpolation='jump')
```

The supporting points must be added in order of increasing time. Now we can do two things: - leave the pulse and instantiate it using concrete values for the parameters, or - re-parametrize the pulse with more expressive parameters.

```
#The first one is simple, just set up the parameter dictionary and plot the pulse:# We can just plug
parameters = {'ta': 200,
              'tb': 2000,
              'end': 4000,
              'va': 2.2,
              'vb': 3.0}
# with these parameters, we can plot the pulse:
plot(squarePulse, parameters)
```

Re-parametrizing is not difficult, either. We first choose a new set of parameters and then provide functions that can calculate the old/inner parameters from the new/outer ones and collect them in a dictionary.

```
mapping = {}
mapping['t_up']   = 'start'
mapping['t_down'] = 'start + length'
mapping['value1'] = 'value1'
mapping['value2'] = 'value2'
mapping['end']    = 'pulse_length * 0.5'
```

The mapping functions get called with one argument, a dictionary of *outer parameters*, and return a value for the *inner parameter*. This example uses lambda functions, but you are free to use any python function.

We can now wrap our pulse in a *SequencePulseTemplate*:

```
doubleSquare = SequencePulseTemplate([(squarePulse, mapping),
                                      (squarePulse, mapping)], # dictionaries with mapping functions
                                     ['start', 'length', 'value1', 'value2', 'pulse_length']) # decla
```

The first argument to the constructor is a list of tuples *(pulse template, mapping dictionary)*, the second argument is a list of the *SequencePulseTemplate*'s *outer parameters*.

Just like with the simple pulse we can instantiate our new double pulse by providing parameters:

```
params = dict(start=5,
              length=20,
              value1=10,
              value2=15,
              pulse_length=500)

plot(doubleSquare, params)
```

Because *doubleSquare* is a pulse template we can just wrap it again, for convenience. The following block shows you how to define a nested pulse template and how to use the low level plotting function to get the pulse shape as a numpy array:

```
nested_mapping = dict(start='start',
                      length='length',
                      value1='10',
                      value2='20',
```

```
                      pulse_length='pulse_length * 0.5')

nested_pulse = SequencePulseTemplate([(doubleSquare, nested_mapping),
                                      (doubleSquare, nested_mapping)],
                                     ['start', 'length', 'pulse_length'])

params2 = dict(start=10, length=100, pulse_length=1000)
plot(nested_pulse, params2)
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search