Simulating the Best Waveforms for Shuttling Operations

A Tool for Ion Shuttling Methodologies

Oneka Singh

04.08.2025

Contents

1 Overview

1.1 Introduction and Goal

In an ion trap, ions can be shuttled in many ways, each corresponding to a specific voltage ramp. Some of these ways include: swap, merge, split, transport, etc.
This paper focuses on the transport voltage waveform for the experimental red trap setup. When waveforms are passed through the electrodes in the ion trap, there can be noise effects induced by the applied voltages. To combat noise, which is a large issue due to the sensitivity of the ion's state, low-pass FIR filters are applied. In turn, the input voltage waveform becomes distorted in some way. More specifically, for the experiment's filters, the waveform becomes convoluted (see terminology).
To restore the correct waveform while filtering electrical noise, a filter precompensation algorithm is employed. It passes an alternative input waveform, which when filtered, yields the original waveform to a degree of accuracy. In other words, the algorithm finds an input waveform that outputs the original waveform before filtering, essentially "canceling" the convoluted output.
The goal of this paper is to develop an understanding of how the experimental filters can be simulated, convoluted, and precompensated, all while optimizing to minimize the error between the estimation waveform and the original waveform. The best transport waveforms have minimal error while being as small in time as possible. In addition to precompensation in general, precompensating optimally (small and accurate) is the end objective of the paper, in addition to seeing the tradeoff and limits of the precompensation technique on the passed waveforms.

1.2 Algorithm Steps

The filter simulation contains the following main steps:

1. Setup and structure input data: waveform and time arrays, minimum DAQ step function
2. Import the filter kernel as an impulse response
3. Simulate the effect of the desired filter as a convolution on the input waveform
4. Precompensate the input waveform to estimate the desired output result (this is the heart of the algorithm)
5. Optimize parameters: less padding, minimal error, maximum compression, best regularization strength
6. Bound the optimization within the limits of the filter's maximum and minimum voltage range

1.3 Terminology

Factors that are optimized (which are discussed more in the Optimize section) include the compression, padding, regularization, and error. These values come up throughout the paper, and what is meant by them is explained below.

Definition 1.1. Compression: The compression factor, which is 1 for uncompressed waveforms, is a multiplicative value that scales the waveform on the time axis. As part of the goal to minimize the time of the waveform, a larger compression value corresponds to temporally smaller waveforms:

$$\text{Compression} > 1 \rightarrow \text{Larger Waveform than Original}$$
$$\text{Compression} = 1 \rightarrow \text{Equal Waveform to Original}$$
$$\text{Compression} > 1 \rightarrow \text{Smaller Waveform than Original}$$

Codewise, the compression is given by dividing the input time array of the waveform (total time) by the compression factor:

```
time_array / best_compression
```

For a time array that spans 0 to 19.38 μs, for example, a compression of 20x would be:

$$\frac{19.38\mu s}{20} = 0.969\mu s$$

This means the new temporal length of the waveform would be 0.969μs.

Definition 1.2. Padding: The padding is applied on both edges of the waveform, extending its entire length. This is done because the precompensated waveform has edge ringing values. These values arise from there being nothing to precompensate on the ends of the function, and thus need to be suppressed.

$$\text{Larger Padding} \rightarrow \text{Larger Waveform}$$
$$\text{Smaller Padding} \rightarrow \text{Smaller Waveform}$$

Since the goal is to reduce the length of the waveform, the padding should be reduced as much as possible to account for just the edge ringing. In the code, the padding is added to the original waveform at the start, and the ringing edges of the precompensated waveform are then suppressed for the padded values.

Definition 1.3. Error: The error between the original waveform and the precompensated filtered (estimated original) is calculated over the entire waveform, on a case-by-case basis (not taken over multiple runs). The goal is to minimize the error between the two waveforms, making the estimation as close as possible to the original. There are two kinds of errors that are tracked.

The first and most important is the maximum absolute error, which corresponds to the largest error point at a certain time for the entire waveform. Graphically, this point is where the two estimated and original waveforms overlap, so minimizing the maximum absolute error means a closer estimation. This error is minimized in the optimization step to get higher accuracy estimation waveforms.
The second error briefly mentioned in plots and printed outputs is the rmse, which is the root mean squared error of the complete function. This is the root of the mean squared error over the entire function. This value generally decreases when the maximum absolute error decreases.

Definition 1.4. Regularization: The regularization term is explained in detail during the precompensation step of the filter algorithm. Simply, however, it ensures that there is no division by zero during the precompensation mathematics. For different waveform sizes and shapes, the best regularization value extends from $1e^{-1}$ to ke^{-n}, and varies on a case-by-case basis. In the algorithm, the best regularization in dependence on the padding and compression values is found through optimization, which is mentioned later.

2 The Algorithm

2.1 Filter Simulation Class

The filter simulation class is responsible for filtering and precompensating the input waveforms. This starts with setting up the input time and waveform arrays, and getting the

input filter kernel. With the input data in place, the waveform can then be run through the filter, and interpolated to match the length of the time array. At the heart of the algorithm is the precompensation step, which estimates the correct precompensated waveform based on the effect of the filtering. The edge ringing of the precompensated waveform is suppressed by setting these edge values to a constant value, and then the final waveform is plotted.

2.1.1 Importing Libraries

The first step is to import the necessary libraries for the FilterSimulation class.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.fftpack import fft, ifft
from scipy.interpolate import interp1d
import seaborn as sns
from sklearn.metrics import mean_squared_error
import matplotlib.patches as patches
import matplotlib.colors as mcolors
```

Numpy is used very often for constructing arrays, operations, and mathematics. Matplotlib is used to plot results. Scipy applies advanced mathematical operations such as the Fourier transform and interpolation on the precompensated waveform (this is explained in the precompensation section). Seaborn and sklearn are used in plotting and as metrics with which the waveform will be optimized (more on this in the Optimize class section).

2.1.2 Filter Method

After importing, the FilterSimulation() class is defined. The first method mentioned sets up the filter kernel through which the waveforms will be passed. The given filter kernel corresponds to a low-pass FIR filter used experimentally in the red trap quantum computer.
The input data type for the given filter kernel is an array of its coefficients. The associated filter kernel weights for the red trap's filter are given by the graph below:

In the filter() method, the weights are first loaded from an npy file as a step response:

```
def filter(self):
    step_response = np.load('filter_data.npy')
```

Then, the impulse response is defined and returned. The impulse response describes how the step response reacts to the input waveform. It is the way filtering affects the later detailed precompensation function.

```
impulse_response = np.diff(step_response, prepend = 0)
return impulse_response
```

2.1.3 The Setup

With the filter kernel defined, the time scale and waveform arrays can be input. It is important to note here that there are some important initial parameters to clarify.
The time step value, which corresponds to the DAQ's minimum sampling time between varying voltage points, is a lower bound restriction on the resolution of the signal. Smaller time steps allow more data points in the given waveform range, leading to higher accuracy. It is important to know the limitations of the hardware in the simulation. In the red trap's

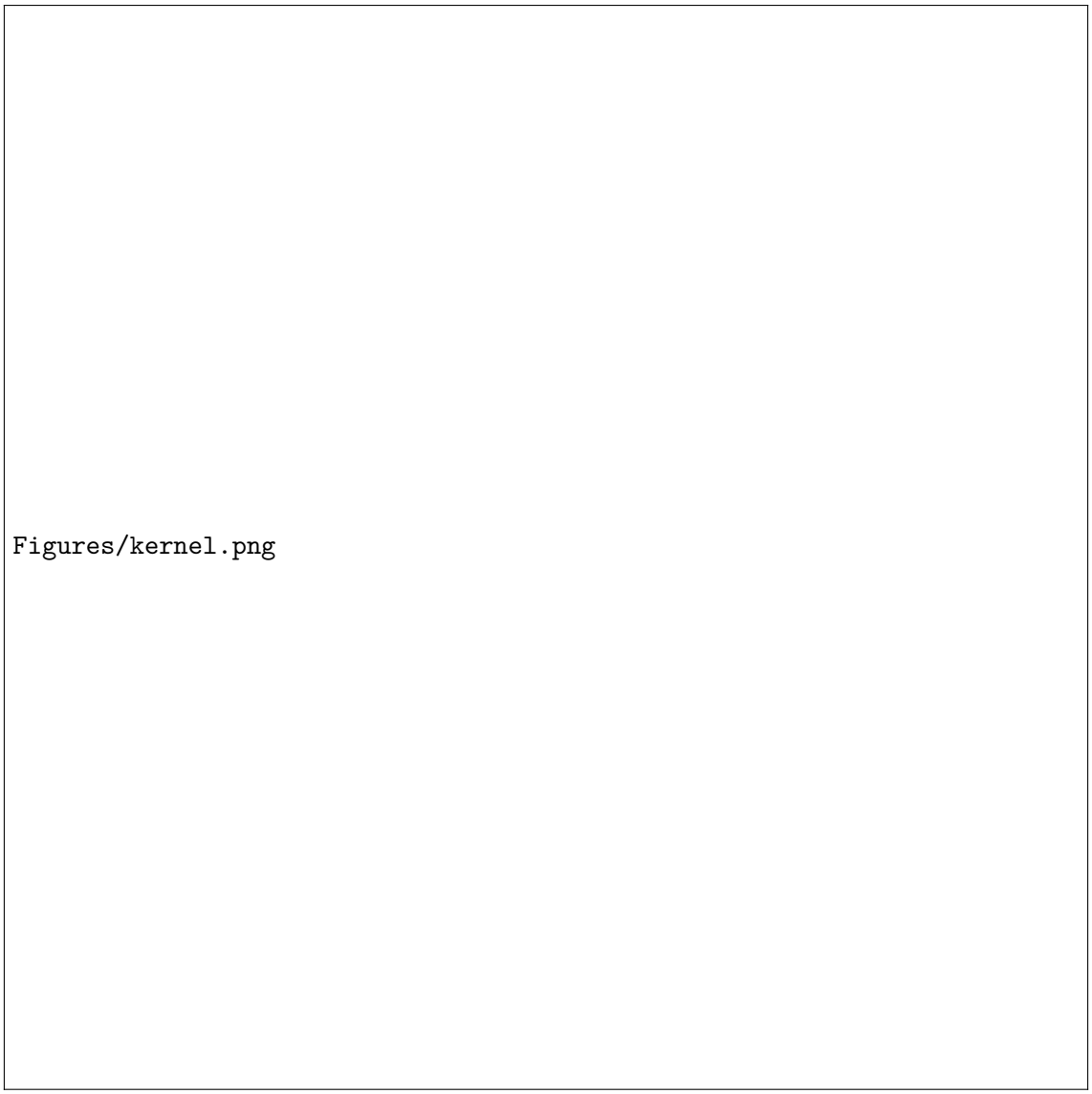


Figure 1: Kernel weights for the FIR filter. These weights correspond to the filtering behavior on the waveforms passed through the electrodes in the red trap.

hardware, the DAQ has a sampling rate corresponding to 380 ns, which sets the time step value in this code.
Additionally, the step size of the waveform (height in volts) is another input factor. The given waveform in this code has a step size of 6 volts, which corresponds to how much the voltage varies over transport.

Both of these parameters are given below:

```
# input values:
waveform_size = -6 # volts
time_step = 0.38 # DAQ lower limit
```

Next comes the input time array and waveform data. Both of these values can range in length, but must match in order for the plotting to align. In the given case, both of these values have a length of 51.

```
time_array = np.array([0, 0.02, 0.04, 0.06, 0.08, 0.1, 0.12, 0.14, 0.16, 0.18,
0.2, 0.22, 0.24, 0.26, 0.28, 0.3, 0.32, 0.34, 0.36, 0.38, 0.4, 0.42, 0.44, 0.46,
```

```
0.48, 0.5, 0.52, 0.54, 0.56, 0.58, 0.6, 0.62, 0.64, 0.66, 0.68, 0.7, 0.72, 0.74,
0.76, 0.78, 0.8, 0.82, 0.84, 0.86, 0.88, 0.9, 0.92, 0.94, 0.96, 0.98, 1])

waveform = np.array([-1, -0.9990133642, -0.9960573507, -0.9911436254,
-0.9842915806, -0.9755282581, -0.9648882429, -0.9524135262, -0.93815334,
-0.9221639628, -0.9045084972, -0.8852566214, -0.8644843137, -0.842273553,
-0.8187119949, -0.7938926261, -0.7679133975, -0.7408768371, -0.7128896458,
-0.6840622763, -0.6545084972, -0.6243449436, -0.5936906573, -0.5626666168,
-0.5313952598, -0.5, -0.4686047402, -0.4373333832, -0.4063093427,
-0.3756550564, -0.3454915028, -0.3159377237, -0.2871103542, -0.2591231629,
-0.2320866025, -0.2061073739, -0.1812880051, -0.157726447, -0.1355156863,
-0.1147433786, -0.0954915028, -0.0778360372, -0.06184666, -0.0475864738,
-0.0351117571, -0.0244717419, -0.0157084194, -0.0088563746, -0.0039426493,
-0.0009866358, 0]) + 1
```

A '+1' factor is added to the end of the waveform to ensure that the signal begins at zero–otherwise, the precompensation algorithm strays from the input waveform.
The time array is normalized to a length of one. The total length of the filter can be calculated by multiplying the waveform length by that of the time step:

$$51 \times 380ns = 51 \times 0.38\mu s = 19.38\mu s$$

In code, this is:

```
total_time = len(waveform) * time_step # 19.38 microseconds, NOT 17.5 microseconds
```

Then the number of samples is the length of the waveform, which is 51:

```
num_samples = int(total_time / time_step) # len(waveform)
```

The time array and waveform can now be scaled up to the desired size:

```
time_array *= total_time
waveform *= waveform_size
```

Mathematically, this is:

$$\text{time array: } (0,1) \to (0,(1 \times 19.38)) \to (0,19.38) \text{ in } \mu\text{s}$$
$$\text{waveform: } (0,1) \to (0,(1 \times 6)) \to (0,6) \text{ in volts}$$

This gives a time scale of 0 to 19.38 μs and a waveform with 51 points, all spaced at distances of 380 ns.
Next, these values are input to the optimizer to find the best padding size, compression factor, and regularization value that yield the lowest error. This is explained more in the Optimize section later on. For now, the best parameters are taken to complete the setup step.
To structure the input waveform and time array properly to allow for accurate filtering and precompensation, the setup method in the Filter Simulation class is passed:

```
original, time, dt, first_index, last_index = FilterSimulation().setup(time_array
/ best_compression, waveform, best_compression, num_samples, best_padding)
```

This outputs the original waveform, the time, which is an extension of the time array that matches the original waveform, the rate of change, which is dt, and the first and last indices of the original waveform. The next step is to look at the contents of the setup and how it retrieves these values.

```
def setup(self, time_array, waveform, compression, num_samples, padding):
```

The setup method takes in the time array (which can be compressed depending on the compression factor), the waveform, the number of samples, and the padding factor. As mentioned previously, the padding and compression factors are found through optimization and are discussed in more detail later on.
It starts by creating a proper time axis from 0 to the last index of the time array, with the same number of samples as the waveform has.

```
dense_time = np.linspace(0, time_array[-1], num_samples)
```

Then, the waveform is interpolated over the time axis values with x-coordinates being the compressed time array and y-coordinates being the waveform array.

```
dense_waveform = np.interp(dense_time, time_array / compression, waveform)
```

Next the dense waveform is set to the original waveform, which will be used in the algorithm. This is done by adjusting the edge padding of the waveform to practically extend the first and last indices by an amount that is given by the optimizer. As mentioned in the terminology section, the larger the padding, the longer the waveform. We do not want this to happen, so we optimize padding to as small a value as can just mitigate the precompensation edge ringing.

```
original = np.pad(dense_waveform, pad_width = best_compression, mode = 'edge')
```

Next the time axis is sliced to the length of the original waveform if it is larger than the original. This ensures that plotting aligns axes. Then, the dt value, the difference between 2 samples or the rate of change of the time axis is found. This is our 0.38 μs from earlier. To construct the final time array, the array is arranged with the length of the original function and points spaced out at dt intervals. In our experiment, with a waveform length of 51 and dt of 0.38 μs we have a total length of 19.38 μs, which matches our result from the beginning of the section.

```
time_sliced = dense_time[:len(original)]
dt = time_sliced[1] - time_sliced[0]
time = np.arange(len(original)) * dt
```

The first and last indices of the waveform are important for the later used trimming function, which does the actual edge ringing suppression of the precompensated waveform. These value are also calculated, and everything is then returned.

```
first_index = original[0]
last_index = float(original[-1])

return original, time, dt, first_index, last_index
```

2.1.4 Convolution "Filter Simulation"

The next important step is to actually simulate the effect of the filter on the original input waveform. This is done by a method from the FilterSimulation() class called convolution(waveform). It is called with:

```
filtered = FilterSimulation().convolution(original)
```

This returns the filtered waveform. Taking a look at the actual contents of the convolution shows its relation to the specific FIR filter coefficients from the Filter Kernel section.

```
def convolution(self, waveform):
    impulse_response = self.filter()
    filtered_waveform = np.convolve(waveform, impulse_response, mode = 'full')
    return filtered_waveform
```

First, the impulse response is defined, which is retrieved from the filter method. As detailed previously, this method sets up the filter kernel depending on the input coefficients. The impulse response shapes how the convolution works, and is based upon the filter kernel. Then, numpy's convolve() method is used on the waveform with the given impulse response. The mode 'full' ensures that the convoluted waveform's size is not warped, which would significantly distort the waveform in relation to the time axis. The output filtered waveform is then returned.

2.1.5 Interpolation Method

This output waveform is shifted from the time axis in some way after the convolution operation, so it is "reanchored" to the current time axis by creating a shifted time array, and interpolating over the time axis values, with the x-axis being the shift and the y-axis being the filtered waveform.

```
shift_f = np.arange(len(filtered)) * dt
filtered = FilterSimulation().interpolation(time, shift_f, filtered)
```

This calls the Filter Simulation's interpolation method. The precompensated and precompensated filtered waveforms must also be interpolated for the same reason as mentioned above (time axis shifts), so this method is created for cleanliness.

```
def interpolation(self, time, shift, filtered):
    interp_func = interp1d(shift, filtered, kind='cubic', bounds_error=False,
        fill_value="extrapolate")
    filtered_interp = interp_func(time)  # Now same length as time_original
    return filtered_interp
```

Note that the interp1d() function from scipy is called, which is different than np.interp(). The scipy interp1d() constructs a function that estimates values between a set of known data points–in this case, between the shift and filtered arrays. The 'cubic' kind is the shape of the interpolation, and is smoother than other kinds, like 'linear'.

2.1.6 The Filter Precompensation

The main charm of this algorithm lies in the mathematics of the precompensation. As explained earlier, the original waveform is distorted (convoluted) when it undergoes filtering. To combat this, the waveform is precompensated to take on a different input shape so that the output matches the original waveform.
The key point to precompensation and having to estimate the original waveform with a margin of accuracy arises from the fact that the function is injective. This means that it is not invertible, and you cannot restore its input based on the output. To estimate this function, a series of Fourier transformations in the frequency domain.

The precompensation is called with:

```
precompensated = FilterSimulation().precompensation(original, last_index,
    best_reg_strength, best_padding)
```

Inside the actual precompensation method, the impulse response is first retrieved from the filter method, similar to what is done earlier.

```
def precompensation(self, waveform, last_index, reg_strength, best_padding):
    impulse_response = self.filter()
```

Since the original waveform is now padded, the impulse response must know how to work the greater-length waveform, and the inputs are both padded to sync their length.

```
total_length = len(impulse_response) + len(waveform)
ir_padded = np.pad(impulse_response, (best_padding, total_length -
    len(impulse_response) - best_pading), mode='constant')
waveform_padded = np.pad(waveform, (best_padding, total_length -
    len(waveform) - best_pading), mode='constant', constant_values=last_index)
```

After this initial setup step begins the pure mathematics. The function is now ready to begin the precompensation estimation.
Let $h(t)$ be the function of the original waveform, $g(t)$ be the effect of the filter on the input waveform, and $f(t)$ be the output convoluted waveform. Then, the relation below becomes apparent:

$$f(t) = g(t) * h(t) \tag{2.1}$$

Here, the $*$ corresponds to the *convolution* operation of the filter on the input waveform. This corresponds perfectly to the convolution() method defined earlier to simulate the filter effect.
Codewise, we set the padded waveform to the input $h(t)$ value:

```
h_t = waveform_padded
```

Since the mathematics is being done in the frequency domain, the complex transfer function of the filter $G(\omega)$ (called G in the code) is given by:

```
G = fft(ir_padded)
```

As one can see, the complex transfer function of the filter directly relates the filter's effect to the padded impulse response, where the filter kernel comes into play.
Here, another realization arises, since $g(t)$ is the effect of the filter function, and $G(\omega)$ is the complex transfer function of the filter.

$$g(t) = F^{-1}(G(\omega)) \tag{2.2}$$

This fact is useful in the grand scheme of the mathematics. If we know how we want the output waveform $f(t)$ to look (like the original signal), then the precompensated waveform to solve for is $h(t)$. For this reason, we isolate and solve for the correct precompensation waveform that yields the original waveform after convolution.

$$h(t) = F^{-1}\left(\frac{F(\omega)}{G(\omega)}\right) = F^{-1}\left(\frac{F(f(t))}{G(\omega)}\right) \tag{2.3}$$

The code then follows with taking the fourier transform of $f(t)$, which will be called $F(\omega)$:

```
h_w = fft(h_t)
```

There is a big problem here. If any of the complex transfer function's values are zero, the value of $h(t)$ becomes undefined. This is because $G(\omega)$ is in the denominator. To ensure this never happens, the function undergoes regularization, which prevents division by zeros or near-zeros.

2.1.7 Tikhonov Regularization

In the precompensation algorithm, we invert the filter transfer function $G(\omega)$ to compute the input waveform $h(t)$ that will produce a desired output after filtering. To avoid getting close to or being zero, $G(\omega)$ is regularized. This begins by defining the complex transfer function in terms of its amplitude attenuation $A(\omega)$ and phase $\phi(\omega)$.

$$G(\omega) = A(\omega)e^{i\phi(\omega)} \tag{2.4}$$

In the code, numpy's abs() and angle() methods are used.

```
G_magnitude = np.abs(G) # amplitude attenuation
G_phase = np.angle(G) # phase shift
```

As noted above, the filter inversion becomes unstable when $G(\omega)$ is very small (at frequencies strongly attenuated by the filter). Here, dividing by zero leads to large amplification of noise or numerical artifacts (excess ringing) in the precompensated signal.
Tikhonov regularization stabilizes signals at high frequencies that cause near-zero function values. This technique is demonstrated in our setup using the Tikhonov regularization equation below:

$$H(\omega) = \frac{F(\omega) \cdot G^*(\omega)}{|G(\omega)|^2 + \lambda} \tag{2.5}$$

The λ value here corresponds to the regularization strength, called $reg_strength$ in the code. It is a small positive constant that stabilizes the inversion, and can be tuned to a certain value (usually in the range e^{-5} to e^{-1} for the best precompensation fit. This tunable parameter is optimized in the Optimizer for this reason, which is detailed in the later Optimize class section.
Tikhonov regularization suppresses high-frequency components that interfere with the precompensated waveform, and provides a controlled trade-off between exact inversion and smoothness of the precompensated waveform.

In the precompensation method, this step is given by:

```
regularized_magnitude = G_magnitude / (G_magnitude**2 + reg_strength)
regularized_G = regularized_magnitude * np.exp(-1j * G_phase)
```

Here, the amplitude attenuation and phase shift become apparent. Mathematically speaking, these lines correspond to:

$$\left(\frac{A(\omega)}{A(\omega)^2 + \lambda}\right) \cdot e^{-i \cdot \phi(\omega)} \tag{2.6}$$

With this regularized value of $G(\omega)$, the Fourier transfomation sequence can be completed:

```
    f_w = h_w * regularized_G
    f_t = np.real(ifft(f_w))

    return f_t
```

Thus, the correct output $f(t)$ is found. This function represents the precompensated waveform that returns the original input waveform with the degree of accuracy determined by the regularization trade-off (larger regularization means smoother function, and smaller regularization means higher frequency spikes).

2.1.8 Trimming Method

Before the precompensated waveform is fully ready for use, it still has one more problem: edge ringing. As mentioned previously edge ringing on the precompensated waveform occurs because there is no waveform to precompensate on the ends of the function. This is suppressed by padding the original waveform's edges by a constant value matching that of the first and last indices. Over these extended edges, precompensated waveform's (ringing) values are suppressed. The trimming method is responsible for this suppression.
First, the precompensated waveform is sliced to the length of the original to ensure matching lengths, and then the the method:

```
precompensated = precompensated[:len(original)]
precompensated = FilterSimulation().trimming(precompensated, best_padding,
    first_index, last_index)
```

As mentioned in the setup section, this is where the first and last index values of the original waveform become important.
From the beginning of the precompensated waveform (index zero) to the end of the padded segment, the precompensated array is set to a constant value of the first index. The same is done for the end, with the padded end period.

```
def trimming(self, precompensated, best_padding, first_index, last_index):
        for i in range(0, int(best_padding*0.3)):
            precompensated[i] = first_index
        for i in range(len(precompensated) - int(best_padding*0.5),
            len(precompensated)):
            precompensated[i] = last_index
        return precompensated
```

Notice that there is one explicit difference that is added to the code. This is the 'tuning' parameter in the trimming function. For the beginning padding segment, instead of going from the beginning of the signal all the way up to the end of the padding period, the suppression only goes up to the 0.3 of the way. This is because of the behavior of the precompensation waveform.
In order to overcompensate accurately, the waveform begins to take shape before the start of the original waveform, meaning that it cuts into the padding zone. The 0.3 is the real 'edge' of the precompensated waveform, where the edge artifacts (ringing) become apparent.
A similar reason is given to the ending pad segment, but the waveform only starts to produce ringing 0.5 the way of the padded segment–suppressing it any earlier would warp the effect of the precompensation.

2.1.9 Filtering the Precompensated Waveform

With a fully precompensated waveform, simulating this new input through the filter simulation provides a precompensated filtered waveform. This is the end result, and should be optimized to attain an estimation with as minimal error as possible to the original, unfiltered input waveform.

This begins by running the precompensated waveform through the convolution method, just like the filtered waveform.

```
precompensated_filtered = FilterSimulation().convolution(precompensated)
```

As mentioned previously, the convolution can distort the waveform, which calls for interpolation to realign the waveform with the central time axis. The same process is used for the precompensated filtered waveform.

```
shift_p = np.arange(len(precompensated_filtered)) * dt
precompensated_filtered = FilterSimulation().interpolation(time, shift_p,
precompensated_filtered)
```

The same steps are applied to also visualize the precompensated waveform before filtering.

```
shift_p_unfiltered = np.arange(len(precompensated)) * dt
precompensated_unfiltered = FilterSimulation().interpolation(time,
shift_p_unfiltered, precompensated)
```

The result is four waveforms: original, filtered, precompensated unfiltered, and precompensated filtered. These can be visualized through matplotlib plotting.

2.1.10 Plotter Method

The plotter method takes the four different waveforms and the cohesive time axis and plots them together. The convoluted waveform is shallower than the original and starts later, while the precompensated unfiltered waveform starts before the original. The hope is to maximize overlap between the original and precompensated filtered waveforms.

The plotter method is first called:

```
FilterSimulation().plotter(original, time, filtered, precompensated_unfiltered,
    precompensated_filtered)
```

As seen, it takes in the 4 waveforms and the time axis. Inside the method, all four waveforms are plotted on the central time axis and are labeled accordingly.

```
def plotter(self, original, time, filtered, precompensated_unfiltered,
precompensated_filtered):
    plt.figure()
    plt.plot(time, original, label = 'Original')
    plt.plot(time, filtered, label = 'Filtered')
    plt.plot(time, precompensated_unfiltered, label = 'Precompensated Unfiltered',
        linestyle = '--')
    plt.plot(time, precompensated_filtered, label = 'Precompensated Filtered',
        linestyle = '--')
```

The time is still in μs, which is on the x-axis, and the y-axis is in volts.

```
    plt.title('Filter Waveforms')
    plt.xlabel('Time in µs')
    plt.ylabel('Voltage')
    plt.grid(True)
    plt.legend()
    plt.show()
```

The result of this algorithm is shown below, after running the plotter method at the end of the code. These results correspond to the aforementioned filter kernel, time array, and waveform input. The padding and regularization strength λ are optimized later on but are set to a constant value for simplicity in explanation here. Compression is not yet applied here.

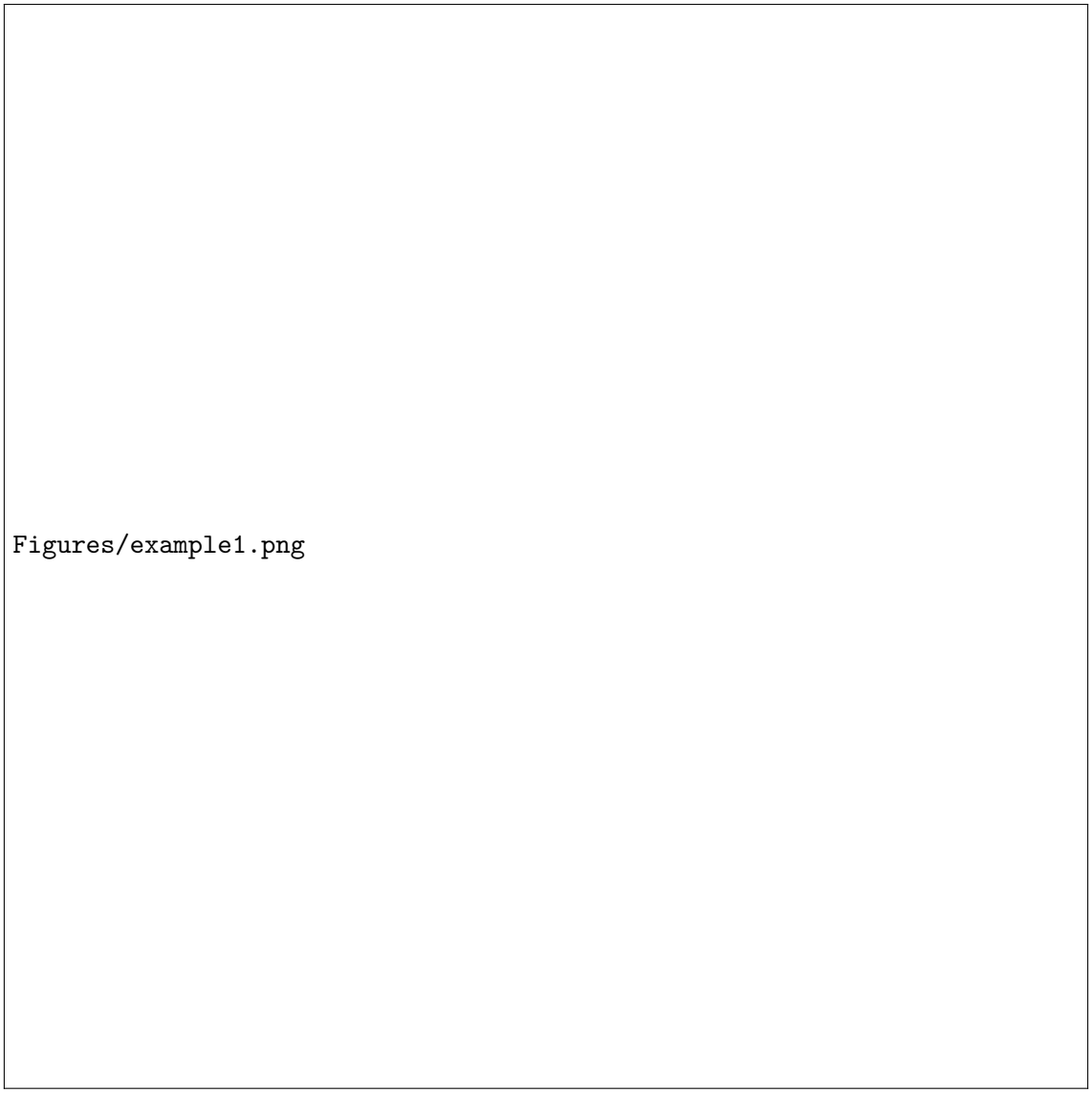


Figure 2: A simple example of applying the filter simulation algorithm on an input waveform.

The specific given values for this result are listed below:

Input Waveform Array	[-1, -0.9990133642, ..., 0] + 1
Input Time Array	[0, 0.02, ..., 1]
Filter Kernel Coefficients	[4.65175217e-05, ..., 1.00004649e+00]
Minimum DAQ Step	380 ns
Total Duration	19.38 μs
Waveform Size	6 volts
Padding	30
Regularization Strength	5e-05
Min/Max Voltage Strength	$\pm$40 volts

Here, the calculated error is:

```
Root Mean Squared Error (RMSE): 0.15461 V
Max Absolute Error: 0.28321 V
```

Graphically, the maximum absolute error over time is represented as follows.

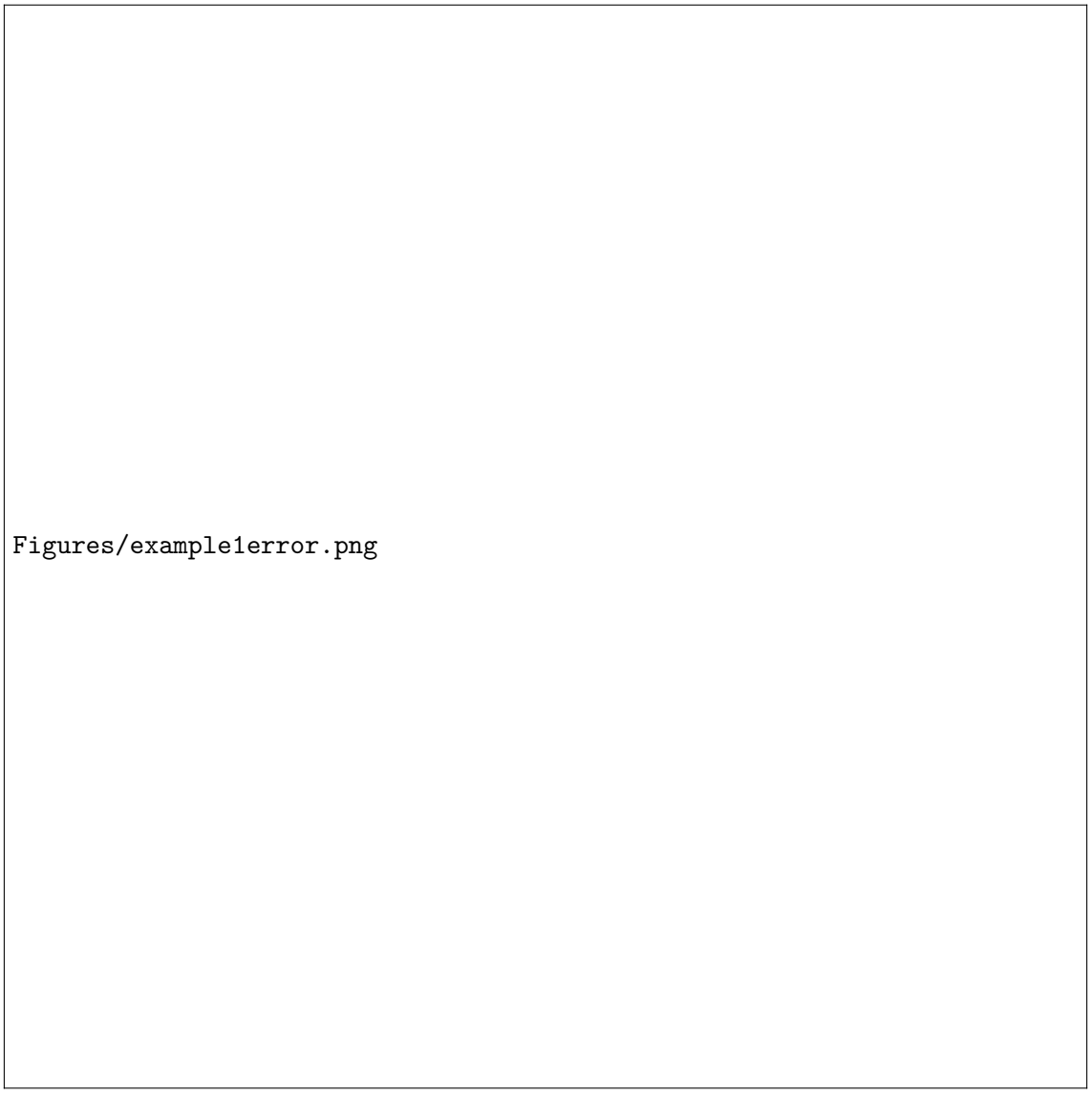


Figure 3: A simple example of applying the filter simulation algorithm on an input waveform.

As seen, the peak error is at 0.28321 volts, a bit above 0.25 volts. The root mean square error (rmse) takes the root of the average error, and is always lower than the max absolute error. The smaller the error margin (of rmse and max absolute error), the greater the accuracy of the estimated precompensated waveform, and the greater the overlap between the original (blue) and precompensated filtered (red) waveforms.

2.2 Optimize Class

The optimize simulation class optimizes the precompensated waveform to achieve:

1. As low an error margin as possible → best precompensation approximation of original waveform
2. As small a waveform as possible → faster transport sequences

Optimizing the waveform provides a roadmap to understanding the boundaries and limitations of the precompensation approach taken in this paper. It provides a threshold given by the varying parameters that are finely tuned for filter optimization:

- Minimize **padding** → smaller waveforms
- Maximize **compression** → smaller waveforms
- Minimize **max absolute error** (consequently rmse) → more accurate waveforms
- Find best **regularization strength** → more accurate waveforms

These parameters are further detailed in the terminology section at the beginning of this paper. The optimization class has several methods for plotting relevant data (such as the error plot from the section above), along with a main, large method that approximates the best parameters depending on an input waveform and tunable weights.

2.2.1 Parameters and Optimization Logic

The algorithm finds the best matching case according to the weight of three main parameters: max absolute error (alpha), padding (beta), and compression (gamma).
Each parameter is given a tunable *weight* according to its relevance in the result. For example, to get a waveform with minimal max absolute error, one has to increase the weight on alpha, which yields a more accurate precompensation at the stake of a larger function. Exploring the effect of different weights helps demonstrate the tradeoff between waveform size and waveform accuracy, and the true "estimation" limitation of the precompensated waveform.
In code, the class is defined and initial parameters are attributes of the class, as follows:

```
class Optimize:

    def __init__(self):
        self.padding_range = list(range(xx, xx))
        self.reg_exponents = list(range(-x, x))
        self.reg_coeff = list(range(xx, xx))
        self.compression_range = list(range(xx, xx))

        self.lower_voltage_bound = -40
        self.upper_voltage_bound = 40

        self.alpha = xx.xx # max abs error
        self.beta = xx.xx # padding
        self.gamma = xx.xx # compression
```

The values marked with x or xx are variable parameters that are to be tuned depending on the waveform. Usually regularization is between 9e-1 to 1e-6. Padding also ranges, and usually falls between 20 and 100 sample units.

The weights alpha, beta, and gamma are also defined here.
The range of the padding and compression corresponds to how many values the optimizer should test. Larger ranges mean a longer runtime, so it is best to benchmark the waveform for appropriate padding and compression values beforehand, to get a general feel for its limits.
The weighted parameters are summed up to a penalty value. The penalty value for each combination of parameters in the given range is stored, and the lowest penalty is chosen

at the end of the loop to yield the best result.
Additionally, the lower and higher voltage bounds that are restricted by the hardware of the device are also defined attributes. Together, these values are constantly used in the Optimize class, and set up the initial tweakable bounds for the behavior of the methods in the class.

2.2.2 Optimizer Method

The optimizer method is the heart of the optimize class. It was briefly mentioned before the setup method of the filter simulation class, as this is where it is located in the code.

```
...
waveform *= waveform_size

best_error_grid, bounds_grid, best_compression, best_padding, best_reg_strength =
Optimize().optimizer(waveform, time_array, num_samples) # the method of focus

Optimize().heatmap(best_error_grid, bounds_grid, best_padding, best_reg_strength,
...
```

The optimizer method takes in the initial waveform and time arrays before properly setting them up as the original waveform and time axis. This is because the parameters need to be found before the original waveform–which is padded and compressed by a certain degree–can be set up. Output are the best compression, padding, and regularization values, plus two additional values used to plot results (used by other methods in Optimize).
The optimizer method is defined below, and some initial variables are defined:

```
def optimizer(self, waveform, time_array, num_samples):
    best_penalty = float('inf')
    best_params = (None, None, None, None, None) # compression, padding,
    reg_strength, penalty, bounds
    all_best_params = []
    reg_array = []

    compression_grids = {}
```

Initially, the best penalty is set to infinity, so that later in the code, the first iteration will take the place of the best penalty thus far. Any penalty smaller than the initial run is then tracked, and at the end, all tracked penalties are compared to find the absolute minimum penalty case.
The second defined variable is a blueprint of the five parameters to be stored for the tracked best penalty case scenarios. The five values to be stored are the compression, padding, regularization strength, calculated penalty (where the weight comes into play), and the bounds (checks voltage bounds). If the voltage of the precompensated waveform reaches above 40 volts or below -40 volts for a certain parameter combination, that result is voided. This is done by setting the bounds for all voided results to one, and all inclusive results to zero. The function later checks the bound value first, to further decrease the sample size, it needs to search for the lowest penalty.
The best parameters with respect to the run are saved in a variable as well as the regularization coefficient c corresponding to the current regularization exponent n. These values are later used to construct the full regularization strength $c \cdot 10^{-n}$.
For the later mentioned heatmap method, an error grid and bounds grid are also created. The error grid finds the magnitude of the absolute max error per combination for a set

of padding and regularization values. The bounds grid shows for which combinations the results are out of the hardware range of ±40 volts. Each run has a corresponding error and bound grid, two arrays which are stored in the compression grid dictionary below:

```
compression_grids = {}
```

Furthermore, the compression grid iterates over all regularization strengths and padding values for a specific compression. Since the heatmap is two-dimensional, only the heatmap corresponding to the resulting chosen compression value is displayed. This means first defining the compression range, and nesting the grids inside of this compression, but before the regularization and padding ranges are defined.

```
# generate all possibilities:
for k, compress in enumerate(self.compression_range):

    error_grid = np.full((len(self.padding_range), len(self.reg_exponents) *
        len(self.reg_coeff)), np.nan)
    bounds_grid = np.full((len(self.padding_range), len(self.reg_exponents) *
        len(self.reg_coeff)), np.nan)

    for i, padding in enumerate(self.padding_range):
        reg_index = 0
        for j, reg in enumerate(self.reg_exponents):
            for l, coeff in enumerate(self.reg_coeff):
```

In the example run, the following range values are used for the padding, regularization, and compression:

Padding Range	$6 \rightarrow 61$
Regularization Coefficient	$1 \rightarrow 8$
Regularization Exponent	-6 $\rightarrow$ 0
Compression Range	$1 \rightarrow 2$

With these ranges, the total combinations are:

$$54 \cdot 6 \cdot 7 \cdot 19 = 44,688 \text{ combinations total}$$

Increasing the number of combinations also increases the processing time. A size of 44,688 combinations takes approximately 1.5 seconds to run, although this can vary for different hardware. For some large combination values, over 1 minute of computation time may be required.
The central loop consists of three nested loops, starting with the regularization, inside padding, inside compression.

```
for k, compress in enumerate(self.compression_range):
        for i, padding in enumerate(self.padding_range):
            for j, reg in enumerate(self.reg_exponents):
```

For each loop iteration, the code runs through a possibility for the precompensated waveform, changing the compression (k), padding (i) and regularization (j) values on each run, until all 44,688 cases are exhausted.
As mentioned previously, the regularization exponent n corresponds to the full regularization strength $c \cdot 10^{-n}$. Because of this, the full regularization strength, which is the value used in precompensation mathematics, is defined.

```
reg_strength = coeff * 10 ** reg
```

Then, the code trials the current iteration case:

```
dense_time = np.linspace(0, time_array[-1], num_samples)
dense_waveform = np.interp(dense_time, time_array / compress, waveform)
original = np.pad(dense_waveform, pad_width=padding, mode='edge')
time_sliced = dense_time[:len(original)]
dt = time_sliced[1] - time_sliced[0]
time = np.arange(len(original)) * dt
first_index = original[0]
last_index = float(original[-1])

precompensated = FilterSimulation().precompensation(original, last_index,
    reg_strength, padding)
precompensated = precompensated[:len(original)]
precompensated = FilterSimulation().trimming(precompensated, padding,
    first_index, last_index)
precompensated_filtered = FilterSimulation().convolution(precompensated)
shift_p = np.arange(len(precompensated_filtered)) * dt
precompensated_filtered = FilterSimulation().interpolation(time, shift_p,
    precompensated_filtered)
```

This code is mentioned in more detail in the previous section on the filter simulation class, and will not be explained here.
The next part of the method is interpreting the results. At the end of the iteration, the maximum absolute error between the original and precompensated filtered waveforms is calculated.

```
error = np.max(np.abs(original - precompensated_filtered))
```

Then, the penalty calculator comes into play. As mentioned previously, the penalty is the sum of the weighted values, as given.

$$\text{penalty} = \alpha \cdot \text{error} + \beta \cdot \text{padding} + \gamma \cdot \text{compression}$$

This is given by:

```
penalty = self.alpha * error + self.beta * padding - self.gamma * compress
```

Now, four of the five stored values are filled. Lastly, the iteration needs to be checked to see if it is outside the voltage bounds.

```
if np.max(precompensated) > self.upper_voltage_bound or np.min(precompensated)
    < self.lower_voltage_bound:
    bounds = 1
else:
    bounds = 0
```

If the maximum precompensated waveform (y-axis) value is over 40 volts or under -40 volts, the bound tally becomes 1 and will be further disregarded. Otherwise, the iteration is still valid.
After calculating both the error amount and the bounds value for the given iteration, these contents are stored as indices in the respective grids. The regularization index keeps track of the iteration corresponding to the input values.

```
error_grid[i, reg_index] = error
bounds_grid[i, reg_index] = bounds
reg_index += 1
```

Lastly, per iteration, the value of the best parameters is constructed. This conditional statement filters out any cases beyond the bounds by checking if the bounds value is one, which is the invalid case. The initially empty parameters array stores each current best value case. Each case is then reviewed later for the best absolute case. The current lowest penalty that passes the condition is then set to the next standard, filtering out cases that are not below the current threshold.

```
if bounds == 0:
    if penalty < best_penalty:
        best_penalty = penalty
        best_params = (compress, padding, reg_strength, penalty, bounds)
        all_best_params.append(best_params)
```

To later plot the regularization strength axis, which depends both on the c and n values, the regularization array initially defined at the beginning of the function is filled with the values spanning the x-axis of the later used heatmap.

```
if len(reg_array) < (len(self.reg_coeff) * len(self.reg_exponents)):
reg_array.append(reg_strength)
```

After the regularization and padding, nested loops have been exhausted, the compression grid dictionary is filled with the current compression iteration corresponding to the current error grid and bound grid. This can be thought of as a "snapshot" of the heatmap for each compression value, and the best case is then returned.

```
compression_grids[compress] = (error_grid.copy(), bounds_grid.copy())
```

After all iterations are complete, the best cases are reviewed based on their penalty value. In the case that no valid result passed the bounds test and the function fails, an error message is displayed. Otherwise, the lowest penalty case is found, as the goal is to minimize the penalty as much as possible. This optimal case is then chosen as the best overall case.

```
valid_results = [res for res in all_best_params if res[4] == 0]
    if not valid_results:
        print("No valid results found within bounds!
            Returning least-penalty violating case.")
        best_params = min(all_best_params, key=lambda x: x[3])
    else:
        best_params = min(valid_results, key=lambda x: x[3])

    return compression_grids, best_params, reg_array
```

Lastly, the grids for the heatmap are returned, the best parameters, which store the values corresponding to the optimal case, and the regularization axis.

```
return compression_grids, best_params, reg_array
```

2.2.3 Heatmap Method

After the optimizer method, which does the heavy lifting for optimization, the results need to be viewed and verified. The values returned from the optimizer are next fed into the heatmap method:

```
Optimize().heatmap(compression_grids, best_params, reg_array)
```

Here, the compacted best parameter value and compression dictionary are expanded to represent their nested values for the best case iteration.

```
def heatmap(self, compression_grids, best_params, reg_array):
    best_compression = best_params[0]
    best_padding = best_params[1]
    best_reg_strength = best_params[2]
    error_grid, bounds_grid = compression_grids[best_compression]
```

In the example case, the returned values are represented below:

Best Padding	31
Best Regularization Coefficient	5
Best Regularization Exponent	-5
Best Compression Value	1
Best Penalty	0.729

Then, these values are expected to be represented on the heatmap generated.
The figure is generated as a 12 by 7 landscape-style graph, but these values can be adjusted depending on the given range size.

```
plt.figure(figsize = (12, 7))
```

The seaborn (sns) library is used to generate this plot. The coolwarm colorscheme is also chosen. The x-axis represents the regularization strengths in scientific notation, and the y-axis represents the maximum absolute error difference between the original and precompensated filtered waveform. This difference is in volts.

```
cmap = sns.color_palette("coolwarm", as_cmap=True)
ax = sns.heatmap(
    error_grid,
    yticklabels = self.padding_range[::10],  # compression values
    cmap = cmap,
    cbar_kws = {'label': 'Error (in V)'},
    annot = False, # set to true for values displayed on each cell
    fmt = '.2e', # scientific notation format for annotations
)
```

The bounds grid is also displayed, and grays out the iterations for which the results are beyond the bounds. The color value for these cells does not have a full opacity to demonstrate how the error hot spots would look for hardware that could potentially exceed the $\pm$ 40 volts boundary.

```
# gray overlay where bounds = 1:
overlay = np.where(bounds_grid == 1, 1, np.nan)
sns.heatmap(
```

```
        overlay,
        mask = np.isnan(overlay),
        map = mcolors.ListedColormap([(0.5, 0.5, 0.5, 0.8)]),
        cbar = False,
        ax = ax
    )
```

The design code lines will not be discussed in detail, as they are standard from the seaborn library. The chosen best case iteration is highlighted in a golden frame, making it easy to tell what the optimizer chose as 'optimal'.

```
    plt.title(f'Heatmap at Compression x{best_compression}')
    plt.xlabel('Regularization Strength')
    plt.ylabel('Padding Samples')

    n_yticks = 7
    ytick_positions = np.linspace(0, len(self.padding_range) - 1, n_yticks,
        dtype = int)
    plt.yticks(ytick_positions, [self.padding_range[i] for i in ytick_positions])

    n_xticks = 12
    xtick_positions = np.linspace(0, len(reg_array) - 1, n_xticks, dtype = int)
    labels = [f'{coeff}e{exponent}' for exponent in self.reg_exponents for
        coeff in self.reg_coeff]
    selected_labels = [labels[i] for i in xtick_positions]
    plt.xticks(xtick_positions, selected_labels, rotation = 45)

    ax.grid(False)
    rect = patches.Rectangle((reg_array.index(best_reg_strength),
        self.padding_range.index(best_padding)), 1, 1, fill = False,
            edgecolor = 'gold', lw = 3)

    ax.add_patch(rect)
    plt.tight_layout()
    plt.show()
```

Thus, the heatmap display is completed. Below is the output figure corresponding to the example case's outputs. The output values given in the aforementioned table above correspond perfectly with the highlighted optimal case, ensuring that the heatmap is showing the correct result.

As seen in the displayed heatmap, the selected optimal tile is at (5e-5, 31) with a maximum absolute error magnitude of 0.729 volts. For different weights, the selected tile would change; lighter weights on padding would present a darker blue, lower tile, while different values for compressions would show completely different heatmap snapshots. The main point here is that this result is quite tunable. Different alterations of parameters and consequently different output heatmaps are pondered in more depth in a later section.
The heatmap is a great visualization of the developed optimization algorithm in practice, and it also showcases the options for precompensation. These options are further illustrated and explored in the other two Optimize methods: $error_plotterandtradeoff_plot$.

2.2.4 Error Plotter Method

The error plotter, which was already showcased in the Plotter Method section to give an elaborate explanation of error in the results, is explained in detail here.

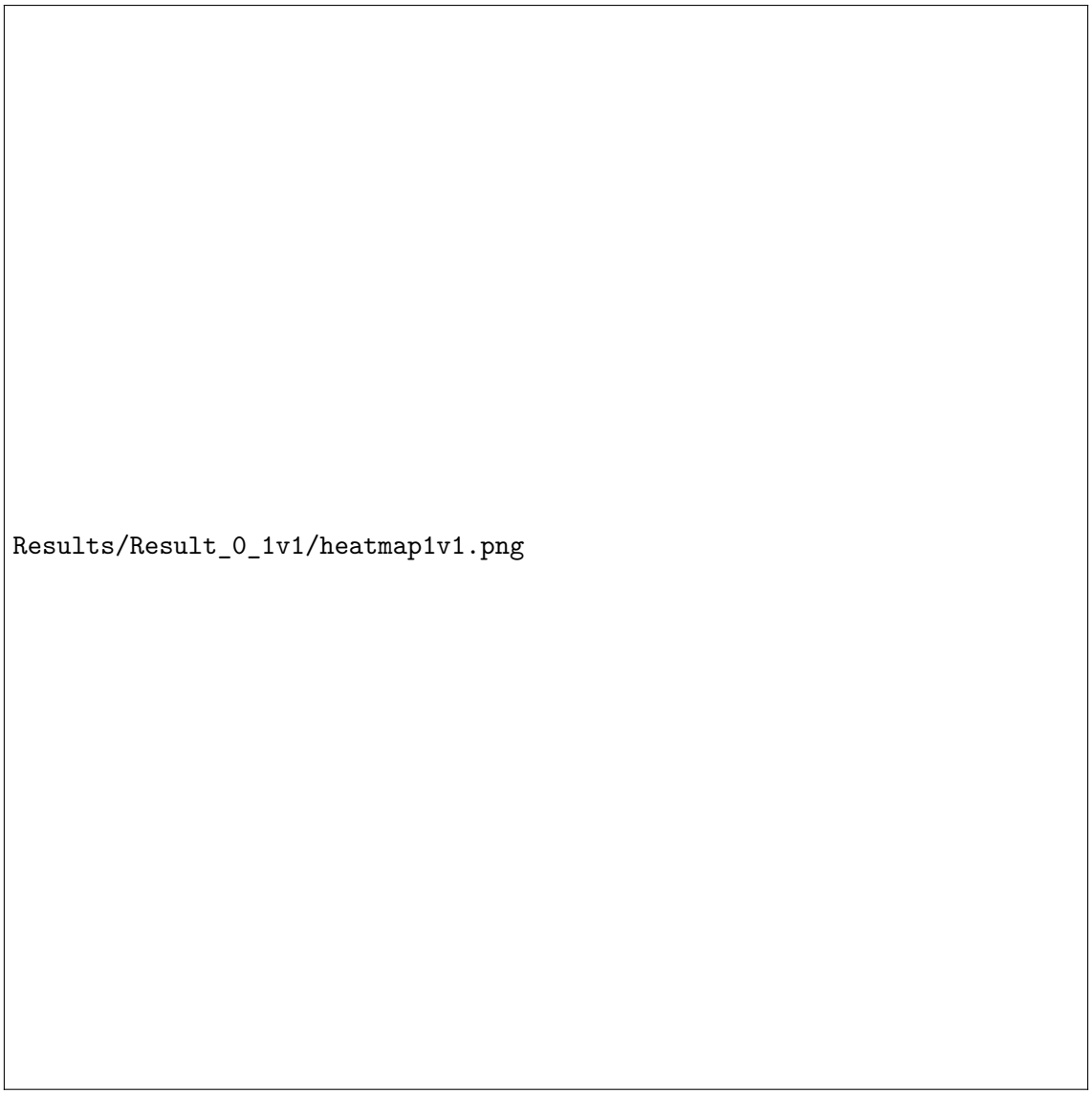


Figure 4: The best compression value with the current set weights is 1. This results from having a much higher weight on low error rather than high compression. As seen in a later section, higher compressions generally lead to higher error rates, which further explains why the lowest possible compression (in this case, 1) is chosen. The padding rang,e as previously mentioned, is between 5 and 61, and the regularization strength is between 1e-6 and 7e-1. The blue areas correspond to fewer errors, and the red areas have the most errors (in terms of maximum absolute error per iteration). The grayed-out areas correspond to values outside the upper and lower voltage bounds of the function and cannot be accessed. The illuminated golden cell is the chosen "best case" of the general heatmap.

The method begins by calling the following line of code:

```
Optimize().error_plotter(time, original, precompensated_filtered, best_params)
```

Like the heatmap method, it also decomposes the best parameters variable.

```
def error_plotter(self, time, original, precompensated_filtered, best_params):
```

The error plotter finds the values of rmse and maximum absolute error (which is the pointwise error per time point of the function).

```
    rmse = np.sqrt(mean_squared_error(original, precompensated_filtered))
    pointwise_error = np.abs(original - precompensated_filtered)
```

```
best_compression = best_params[0]
best_padding = best_params[1]
best_reg_strength = best_params[2]
best_penalty = best_params[3]
```

Next, the method plots the pointwise error concerning the time coordinate.

```
# absolute error over time:
plt.plot(time, pointwise_error, color='red', label='|Original -
    Precompensated Filtered|')
plt.title('Error between Original and Precompensated Filtered Signal')
plt.xlabel('Time in µs')
plt.ylabel('Absolute Error (V)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Additionally, to verify the results of the heatmap match the correct error output, the values of padding, regularization, compression, penalty, rmse, and maximum absolute error are also printed.

```
print(f'The best padding is:', best_padding)
print(f'The best regularization is: {best_reg_strength:.0e}')
print(f'The best compression is: {best_compression}')
print(f'The best penalty is: {best_penalty}')
print(f"Root Mean Squared Error (RMSE): {rmse:.3f} V")
print(f"Max Absolute Error: {np.max(pointwise_error):.3f} V")
```

The result corresponding to the example yields the following values (which are where the table values originate from):

```
#The best padding is: 31
#The best regularization is: 5e-05
#The best compression is: 1
#The best penalty is: 0.7289897979212718
#Root Mean Squared Error (RMSE): 0.075 V
#Max Absolute Error: 0.179 V
```

Then, the corresponding error plot is also shown below:

The error plotter shows the difference in volts between the original and precompensated filtered waveforms at every instance in time. The maximum absolute error is the absolute peak (or maximum) of this graph, and visually shows the magnitude of the maximum error relative to the rest of the function. The root mean squared error (rmse), which is also printed in the chain of results, gives further context to the error plot. The difference in rmse and maximum absolute error gives insights on how sharp the error peak is.

So far, the results within the defined bounds have been showcased. The results thus far show the possibilities within the padding and regularization range (shown in the heatmap), and the error variance throughout the function (shown in the error plot). However, the compression result has been held constant. To fully demonstrate the effect of multiple compression values, and the heart of the tradeoff between error and waveform temporal length, the tradeoff plot method is utilized.

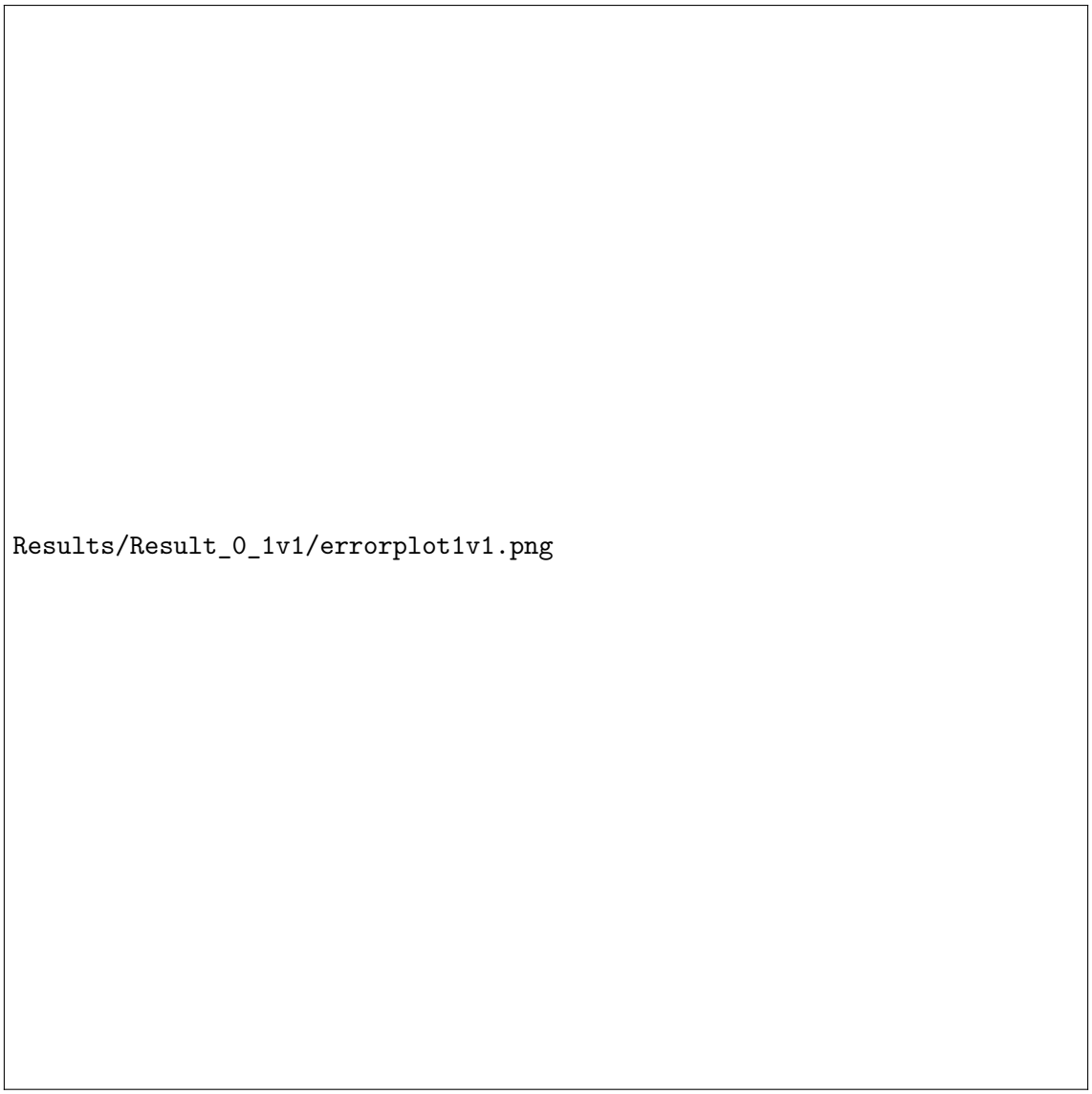


Figure 5: The following plot shows the pointwise error graphically corresponding to the example run.

2.2.5 Tradeoff Plot Method

The tradeoff plot is at the heart of the tradeoff between precompensation error (always in max absolute) and temporal waveform size. As mentioned in the opening section of this paper, the goal of the precompensation optimization is to maximize the accuracy of the precompensated filtered waveform while minimizing its length, in order to speed up adiabatic shuttling transport operations.
The method simply takes in the following parameters:

```
Optimize().tradeoff_plot(time_array, num_samples, waveform, best_params)
```

Some initial parameters defined, which are used to parameterize the for loop used and store iteration results.

```
def tradeoff_plot(self, time_array, num_samples, waveform, best_params):
    error_array = []
    compression_array = []
    best_padding = best_params[1]
    best_reg_strength = best_params[2]
```

The method by which the tradeoff is calculated over a range of compression values is very similar to the optimizer method detailed previously. For this reason, a thorough description of this piece of the method is left out.

```
for k, compress in enumerate(self.compression_range):
    dense_time = np.linspace(0, time_array[-1], num_samples)
    dense_waveform = np.interp(dense_time, time_array / compress, waveform)
    original = np.pad(dense_waveform, pad_width=best_padding, mode='edge')
    time_sliced = dense_time[:len(original)]
    dt = time_sliced[1] - time_sliced[0]
    time = np.arange(len(original)) * dt
    first_index = original[0]
    last_index = float(original[-1])

    precompensated = FilterSimulation().precompensation(original, last_index,
        best_reg_strength, best_padding)
    precompensated = precompensated[:len(original)]
    precompensated = FilterSimulation().trimming(precompensated, best_padding,
        first_index, last_index)
    precompensated_filtered = FilterSimulation().convolution(precompensated)
    shift_p = np.arange(len(precompensated_filtered)) * dt
    precompensated_filtered = FilterSimulation().interpolation(time, shift_p,
        precompensated_filtered)

    error = np.max(np.abs(original precompensated_filtered))
```

At the end of each iteration, the compression and consequent error values are stored for plotting.

```
compression_array.append(compress)
error_array.append(error)
```

The plotting piece of the method is defined below:

```
plt.plot(compression_array, error_array)
plt.xlabel("Compression")
plt.ylabel("Max Absolute Error")
plt.title("Compression vs. Max Error")
plt.grid(True)
plt.tight_layout()
plt.show()
```

Below, the plot of compression to error is given, ultimately showcasing this tradeoff between error and time:

This tradeoff corresponds to the compression factor of the given input waveform. As seen from very early on, the compression factor quickly causes a large error spike of around 11 volts just at a low compression of around 4 times the original waveform size (the exact compression scheme is detailed in the compression terminology section at the beginning of this paper). What is important to realize here is that the error begins to converge to a constant value after a compression factor of around 13 times. The initial spike and convergence behavior of the tradeoff are the two important factors to notice here.
First, if a minimum error is to be achieved, a compression factor of 1 (original size) or 2 (half the original size) should be employed. If an error of around 9 volts is acceptable,

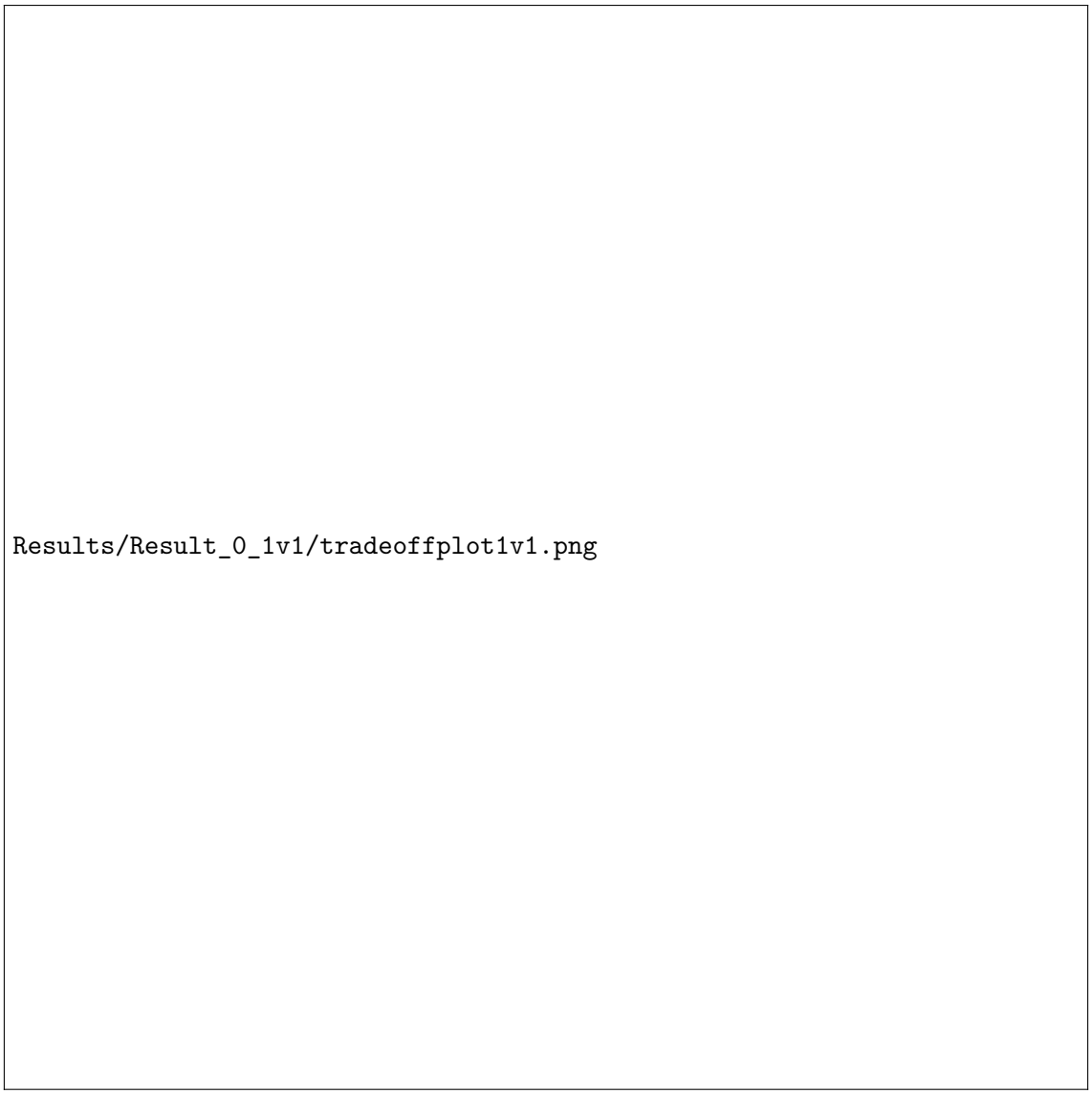


Figure 6: The tradeoff between error values and waveform size is shown above. The range of values for this plot is given between a compression factor of 1 to 20, as this is where the rate of change of the error begins to settle. The maximum absolute error is around 11 volts, and spikes early on at around a compression factor of 4.

due to the convergence behavior with larger compressions, very small waveforms can be achieved with minimal differences in error. This makes it easy to achieve a suitable temporal length at the cost of around 9 volts of error.
In the context of a ± 40 volt bound, this 9-volt difference is very large, and smaller error rates are preferred. For this reason, a compression factor of 1 or 2 is best suited for the example case.

3 Use Cases

3.1 Neighboring Segment Transport

For neighboring segments, the general scheme of the voltage ramp waveform is sinusoidal, as seen in the example used throughout the previous sections. Here, the algorithm's results on different use cases and an analysis of each interpretation given different parameters are the subjects of focus.

3.1.1 Example Case 1: Good for Low Error or Large Compressions

In the first case, the given input parameters are provided in the table below:

Input Waveform Array	[-1, -0.9990133642, ..., 0] + 1
Input Time Array	[0, 0.02, ..., 1]
Filter Kernel Coefficients	[4.65175217e-05, ..., 1.00004649e+00]
Minimum DAQ Step	380 ns
Total Duration	19.38 μs
Waveform Size	6 volts
Minimum Voltage Size	-40 volts
Maximum Voltage Size	+40 volts

These weights correspond to the following optimized parameters used:

Error Weight	1.00
Padding Weight	0.05
Compression Weight	1.00

The optimization parameters are given as:

Padding Range	$5 \rightarrow 61$
Regularization Coefficient	$1 \rightarrow 8$
Regularization Exponent	$-6 \rightarrow 0$
Compression Range	$1 \rightarrow 2$

These parameters will not be explained here, as they were detailed in the previous sections leading up to this point.
The following output is as given:

The corresponding printed results are:

```
#The best padding is: 31
#The best regularization is: 5e-05
#The best compression is: 1
#The best penalty is: 0.72898979792127 18
#Root Mean Squared Error (RMSE): 0.075 V
#Max Absolute Error: 0.179 V
```

This neighboring transport ramp has a compression factor of one, meaning it is not at all compressed. No compression allows the waveform to maximize accuracy, which is shown by the low error rate of ≈ 0.729, which is even below 1 volt–this is a large difference compared with the peak error at 10 volts as shown in image (d).
For the given weights, the selected iteration (in gold in image (c)) is weighted equally between error and compression, explaining why it is located on a low error square that is not necessarily at a high or low padding value. This cell borders the voltage bounds of +40 volts, as shown in the peak near 40 volts in image (a). Here, the precompensation is not yet failing, but instead, the limitation on error versus padding comes from the voltage bound.
This example case section showcases the same input as the previous case, but with different weights. The goal is to show the effect of different weights on the optimizer.
Here, the given weights are:

Error Weight	0.50
Padding Weight	0.09
Compression Weight	0.01

(a) (b)

(c) (d)

Figure 7: (a) The output result of the four different waveforms generated. The original in blue and precompensated filtered in red are what is supposed to match. The precompensated waveform that is not yet filtered is given in green, and the orange is the filtered but unprecompensated waveform. (b) The error between the original and precompensated filtered waveforms at every given point in time. The peak is the maximum absolute error overall. (c) The heatmap snapshot of the best case compression shows the range of padding and regularization values. The golden bordered cell marks the optimal chosen combination of values. The more blue the result, the less error, corresponding to a larger overlap in red and blue in (a). The gray areas are cases over the given voltage bounds. (d) The tradeoff between much larger compression values and error rates for the given waveform.

Then, the results are as follows:
The corresponding printed results are:

```
#The best padding is: 26
#The best regularization is: 4e-05
#The best compression is: 1
#The best penalty is: 2.249
#Root Mean Squared Error (RMSE): 0.451 V
#Max Absolute Error: 0.649 V
```

In this case, the chosen optimal iteration has a smaller padding value of 22 instead of 31 as in case one. Corresponsively, the output (red) waveform in image (a) is also smaller

(a) **(b)**

(c) **(d)**

Figure 8: (a) The output result of the four different waveforms generated. The original in blue and precompensated filtered in red are what is supposed to match. The precompensated waveform that is not yet filtered is given in green, and the orange is the filtered but unprecompensated waveform. (b) The error between the original and precompensated filtered waveforms at every given point in time. The peak is the maximum absolute error overall. (c) The heatmap snapshot of the best case compression shows the range of padding and regularization values. The golden bordered cell marks the optimal chosen combination of values. The more blue the result, the less error, corresponding to a larger overlap in red and blue in (a). The gray areas are cases over the given voltage bounds. (d) The tradeoff between much larger compression values and error rates for the given waveform.

than the first case, at almost 35 μs instead of around 43 μs. In consequence however, the rmse and max absolute error has almost doubled. The rmse from case one to case two is: 0.075v $\rightarrow$ 0.942v. This is a significant increase. Likewise, for the max absolute error: 0.179v $\rightarrow$ 1.264v.

Comparing example cases one and two provides a clear demonstration of the tradeoff between waveform length and accuracy. Waveform one is much more accurate, but some almost 10 μs longer, while waveform two is shorter but not as accurate.

A very important realization for this case is in image (d). If the waveform wants to be scaled down and a $\approx$ 6.1 volt difference is acceptable, large compression values can be achieved due to the conversion behavior.

3.1.2 Example Case 2: Shorter Waveforms with Padding and Weights Reparameterization

In this second example, the same results with different optimization parameters are shown. The goal in this example is to show how different compression and padding values attempt to shorten the waveform, changing the shape of the heatmap in conjunction with the optimizer.
Here, the input values are the same as the first example. The key is to show different optimization parameter cases of the same input to see how the algorithm behaves.

Input Waveform Array	[-1, -0.9990133642, ..., 0] + 1
Input Time Array	[0, 0.02, ..., 1]
Filter Kernel Coefficients	[4.65175217e-05, ..., 1.00004649e+00]
Minimum DAQ Step	380 ns
Total Duration	19.38 μs
Waveform Size	6 volts
Minimum Voltage Size	-40 volts
Maximum Voltage Size	+40 volts

The weights for heatmaps 1 and 2 are:

Error Weight	1.00
Padding Weight	0.05
Compression Weight	2.00

Error Weight	1.00
Padding Weight	0.05
Compression Weight	5.00

These optimization parameters for heatmaps 1 and 2 are:

Padding Range	10 $\rightarrow$ 61
Regularization Coefficient	1 $\rightarrow$ 8
Regularization Exponent	-6 $\rightarrow$ 0
Compression Range	1 $\rightarrow$ 3

Padding Range	30 $\rightarrow$ 81
Regularization Coefficient	1 $\rightarrow$ 8
Regularization Exponent	-6 $\rightarrow$ 0
Compression Range	1 $\rightarrow$ 3

The corresponding results are shown in the figure below:
The corresponding printed results are:

```
#The best padding is: 47
#The best regularization is: 3e-04
#The best compression is: 2
#The best penalty is: -0.615
#Root Mean Squared Error (RMSE): 0.487 V
#Max Absolute Error: 1.035 V
```

In this case, the compression value is 2 instead of 1. Intrinsically, this increases the error. To combat this, larger padding value and regularization strength is employed by the optimizer to keep the waveform in control. In image (a), the effect of larger regularization strength corresponds with the more oscillatory precompensation signal as opposed to the first example with a regularization strength of 5e-5. It is also worth noting that the max absolute error is also much larger than the first case of 0.179 volts (this also applies to the rmse, which is larger). As mentioned in the tradeoff method section, the price to pay for a larger compression value is also a larger error.

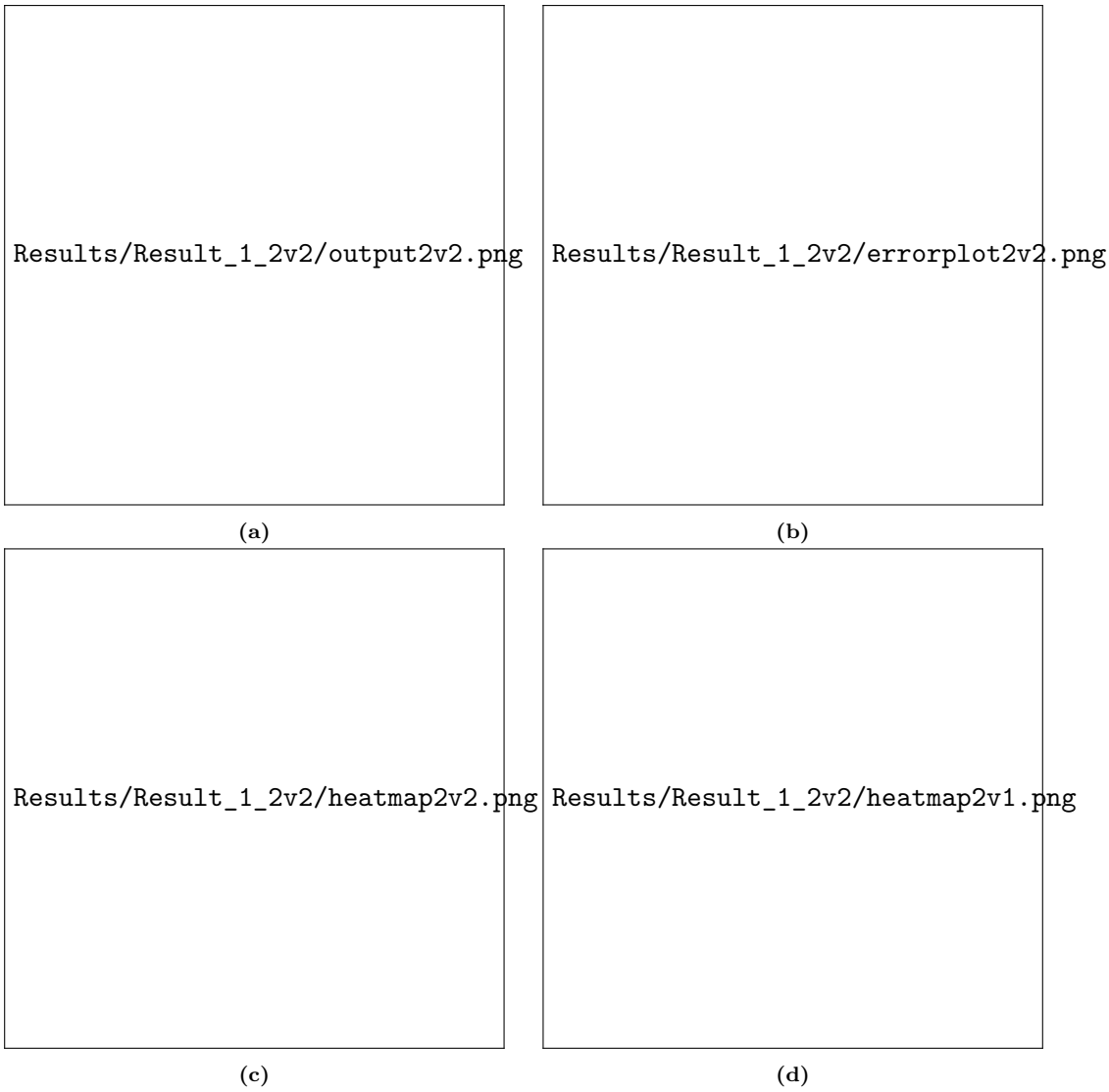


Figure 9: (a) The output result of the four different waveforms generated. The original in blue and precompensated filtered in red are what is supposed to match. The precompensated waveform that is not yet filtered is given in green, and the orange is the filtered but unprecompensated waveform. (b) The error between the original and precompensated filtered waveforms at every given point in time. The peak is the maximum absolute error overall. (c) The first heatmap snapshot of the best case compression shows the range of padding and regularization values. The golden bordered cell marks the optimal chosen combination of values. The more blue the result, the less error, corresponding to a larger overlap in red and blue in (a). The gray areas are cases over the given voltage bounds. (d) A second heatmap snapshot of the same case.

3.1.3 Example Case 3: The Nature of Large Compressions

This example continues on the tradeoff section, utilizing the convergence of large compression values to a constant error value. The point here is to show that, at the cost of in this case ≈ 7 volts, high compressions can be achieved with minimal difference in error compared to a much lower compression. In fact, as seen in the tradeoff graph from the respective section, some higher compressions actually even yield lower error rates than the inital spike at around a compression factor of 12. In the given image (d), the compression range is larger to further amplify the nature of this convergence behavior.
In this case, the setup is as follows (the input data is the same):

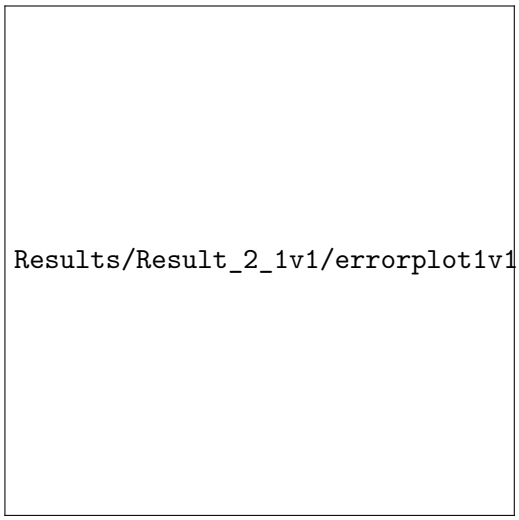
Padding Range	$30 \rightarrow 101$
Regularization Coefficient	$1 \rightarrow 8$
Regularization Exponent	$-6 \rightarrow 0$
Compression Range	$1 \rightarrow 5$

The respective weight values are:

Error Weight	1.00
Padding Weight	0.05
Compression Weight	5.00

Here, the output is:

(a)



(b)

(c)

(d)

Figure 10: (a) The output result of the four different waveforms generated. The original in blue and precompensated filtered in red are what is supposed to match. The precompensated waveform that is not yet filtered is given in green, and the orange is the filtered but unprecompensated waveform. (b) The error between the original and precompensated filtered waveforms at every given point in time. The peak is the maximum absolute error overall. (c) The heatmap snapshot of the best case compression shows the range of padding and regularization values. The golden bordered cell marks the optimal chosen combination of values. The more blue the result, the less error, corresponding to a larger overlap in red and blue in (a). The gray areas are cases over the given voltage bounds. (d) The tradeoff between much larger compression values and error rates for the given waveform.

The corresponding printed results are:

```
#The best padding is: 53
#The best regularization is: 4e-04
#The best compression is: 4
#The best penalty is: -15.317
#Root Mean Squared Error (RMSE): 0.757 V
#Max Absolute Error: 2.033 V
```

With a much larger padding range, higher compression values can be achieved without a much higher max absolute error. Compared to a previous example with a compression of 2 and error of 1.035 volts, here higher compressions can be achieved without a sudden error "spike". Although the max absolute error is twice the magnitude than that of a factor 2 compression case, the waveform is also half the size. For cases in which waveforms want to be kept some factor shorter without a reasonably large compression (such as 9 volts or more for the spike case) the padding is of a higher magnitude. A larger padding may at first sound like a bad idea, but with larger compression values, the division scales this large padding down significantly, as seen in image (a).

3.2 Multiple Segment Transport

In the previous examples, transport voltage ramps are used across neighboring segments. This sinusoidal shape corresponds to the voltage change in the electrode segments is an extreme case of precompensation. For multiple-segment transport, a more bell-shaped curve is employed, which has more precompensation potential. For this reason, larger compressions and lower errors are demonstrated.

Here, the input data is different:

Input Waveform 1 Array	[-1, -0.9990133642, ..., 0] + 1
Input Waveform 2 Array	[0, -0.0009866358, ..., 0] + 1
Input Time Array	[0, 0.02, ..., 1]
Filter Kernel Coefficients	[4.65175217e-05, ..., 1.00004649e+00]
Minimum DAQ Step	380 ns
Total Duration	19.38 μs
Waveform Size	6 volts
Minimum Voltage Size	-40 volts
Maximum Voltage Size	+40 volts

The following optimization parameters are used:

Padding Range	$5 \rightarrow 61$
Regularization Coefficient	$1 \rightarrow 8$
Regularization Exponent	$-6 \rightarrow 0$
Compression Range	$1 \rightarrow 2$

These weights correspond to the following optimized parameters used:

Error Weight	1.00
Padding Weight	0.05
Compression Weight	1.00

Keeping in mind that multiple-segment transport has a different shape than neighboring segments, the resulting output is shown below:

The corresponding printed results are:

(a) (b)

(c) (d)

Figure 11: (a) The output result of the four different waveforms generated. The original in blue and precompensated filtered in red are what is supposed to match. The precompensated waveform that is not yet filtered is given in green, and the orange is the filtered but unprecompensated waveform. (b) The error between the original and precompensated filtered waveforms at every given point in time. The peak is the maximum absolute error overall. (c) The heatmap snapshot of the best case compression shows the range of padding and regularization values. The golden bordered cell marks the optimal chosen combination of values. The more blue the result, the less error, corresponding to a larger overlap in red and blue in (a). The gray areas are cases over the given voltage bounds. (d) The tradeoff between much larger compression values and error rates for the given waveform.

```
#The best padding is: 33
#The best regularization is: 3e-06
#The best compression is: 1
#The best penalty is: 2.493
#Root Mean Squared Error (RMSE): 0.107 V
#Max Absolute Error: 0.193 V
```

A big difference between this non-neighboring segment case is tradeoff graph, which has a much higher initial spike that only begins to taper off at a compression of 10. The 9 volt convergence value is still the same as with neighboring segments. Notable, the heatmap is quite different, favoring larger padding values and smaller regularization strengths much more heavily. At no compressions, the function has quite a nice low error rate of 0.193

volts (max absolute). The padding is also quite small, at 33.

4 Conclusion

4.1 Closing Thoughts

In all cases of transport waveform ramps, a maximum absolute error of well below 1 volt is best achieved by no compression, and a moderate padding of around 30 to 50. The regularization strength for these values is usually also in the 5e-4 to 2e-2 range, which lies in the middle of the heatmap. A compression factor of 2 yields around 1 volt of error, which can be appropriate depending on the waveform optimization priorities (if the speedup is more important than the 1 volt error).
For much larger compressions, a larger error must be accounted for, but this incremental error rate begins to converge as compressions increase, which can be a useful fact if the voltage error (in this case, 9 volts) is small relative to the use case.
To see some compression while still holding on to as much accuracy as possible, larger padding values morph the tradeoff graph to tolerate a larger range of small compression values (with minimal error).

4.2 Hardware Limitations

The DAQ used in our experiment can only update every 380ns (or 0.38 μs), leaving a bound on resolution. For high compressions and high error rates, this is a major problem. For a waveform of 19.38 μs, which is the uncompressed duration of the above two plots, there are only

$$\frac{19.38\mu s}{380ns} = \frac{19.38\mu s}{0.38\mu s} \approx 51 \tag{4.1}$$

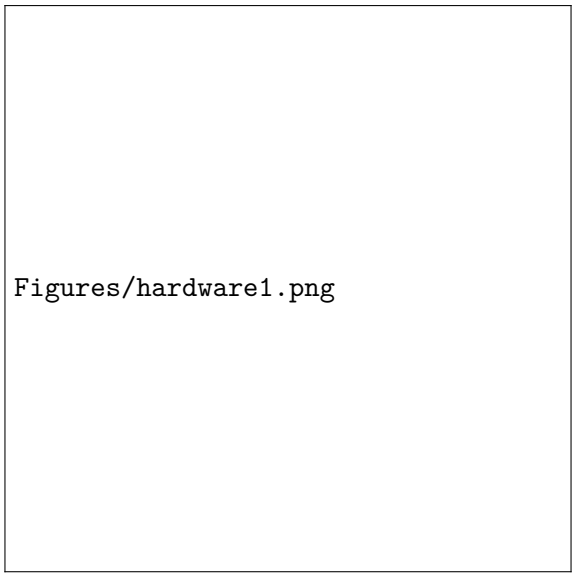
51 samples in total.
To further visualize this restraint on resolution, a graph of plots is shown to highlight the spacing limitation:

Because of this limit of 380 ns or higher resolution (more samples), the waveform would need to be longer in order to have a higher accuracy, which is the opposite of what compression does, and what is desired. This also explains why multiple-segment transport ramps, which are longer than neighboring ramps, have better accuracy. To increase the resolution and decrease the waveform size while still retaining accuracy, a DAQ below a sampling rate corresponding to 380 ns would need to be used.
Another limitation of the waveform precompensation is the voltage bound of ± 40 volts. In some heatmaps from the previous section, the "grayed" area represents out-of-bounds cells, as mentioned before. Some of these values hold low error-low padding values, which could be better case scenarios than the optimal case. For these to be accessible, the hardware used would need to have values above and below 40 volts.
However, within the selected parameters, the precompensation and optimization provided the optimal 'best' case scenario waveform.

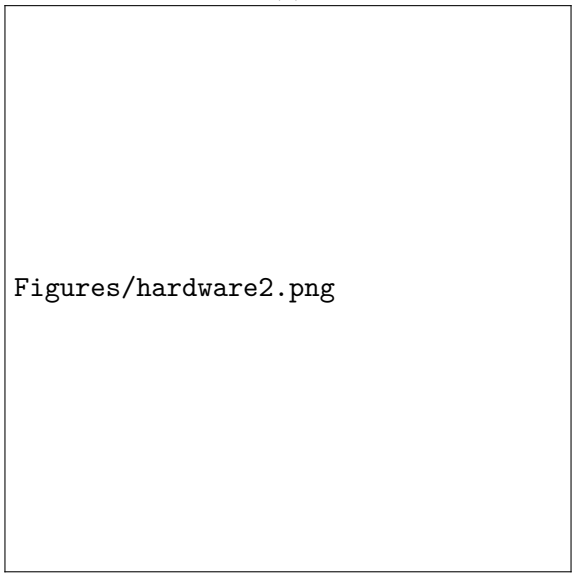
5 Code Reference

For reference to my direct algorithm code used to build this system, you can click here, or visit the following link:

`https://github.com/OnekaSingh/Qiskit-Tutorial-Projects/blob/main/transport_ramps.ipynb`



(a)



(b)

Figure 12: A scatter plot showing the 0.38 μs step between points.

References

Maximilian Heinrich Orth (2021), *Advanced Ion Shuttling Operations for Scalable Quantum Computing*, Master's thesis, Department of Physics, Mathematics and Computer Science (FB 08), Johannes Gutenberg University Mainz, November 2, 2021.

R. Bowler, J. Gaebler, Y. Lin, T. R. Tan, D. Hanneke, J. D. Jost, J. P. Home, and D. Leibfried (2012), *Coherent Diabatic Ion Transport and Separation in a Multizone Trap Array*, Phys. Rev. Lett. **109**, 080501.

R. B. Blakestad (2020), *Shuttling-based trapped-ion quantum information processing*, AVS Quantum Sci. **2**, 014101.

Sebastian Weidt (2012), *Quantum Information Processing with Trapped Ions in Microstructured Paul Traps*, Ph.D. thesis, Johannes Gutenberg-Universität Mainz.