This is a CBPM notebook

The goal is to apply frequency filtering to the raw button waveform simulated with CST Microwave studio. The filtering will include the effect from the signal cable itself, as well as the input 1.2 GHz filter.

The simulation focuses on the North Arc vacuum chamber geometry. The beam is centered on (x, y) = (0, 0) and has 0.7 mA current and 17.3 mm length (1 sigma).

Required imports

In [1]:

```
import copy
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
from scipy import signal

import warnings
warnings.filterwarnings('ignore')
```

Defining a styling function for matplotlib plots

In [2]:

```
def style():
    # set global figure parameters
    plt.rcParams['figure.figsize'] = [9, 6]
    plt.rcParams.update({'font.size': 18})
    plt.rcParams['axes.linewidth'] = 1.25

    # create figure and axis objects
    fig = plt.figure()
    ax = fig.add_subplot()

    # lots of cosmetics/style
    ax.yaxis.set_ticks_position('both')
    ax.xaxis.set_ticks_position('both')
    ax.tick_params(axis="y",direction="in", labelleft=True)
    ax.tick_params(axis="x",direction="in", labelleft=True)
    ax.tick_params(axis="y",direction="in", labelleft=True, which='minor')
    ax.tick_params(axis="x",direction="in", labelleft=True, which='minor')
    ax.tick_params('both', width=1.25, length=7.5)
    plt.subplots_adjust(left=0.175, bottom=0.125, right=0.975, top=0.95, wspace=
```

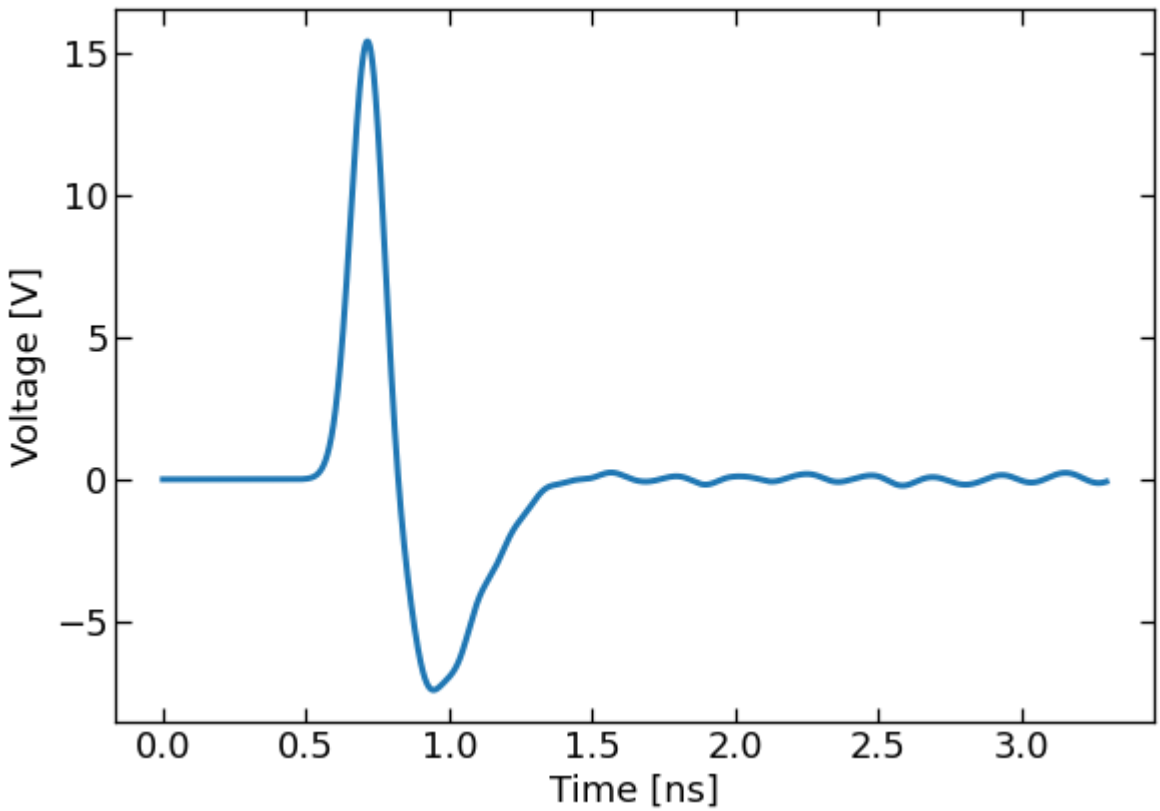
Import raw button waveform an plot it

In [3]:

```
input_file = '/home/atc93/atc93/ngbpm/software/cst_microwave/north_arc_raw_0.7mA

time = np.loadtxt(input_file, skiprows=3, usecols=0)  # importing time array
amp = np.loadtxt(input_file, skiprows=3, usecols=1)  # import amplitude array

style()  # apply plot styling

# do the plotting
plt.plot(time, amp, lw=3)
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
```

Out[3]:

```
Text(0, 0.5, 'Voltage [V]')
```



Cable filtering: from time to frequency domain and back

We are going to apply the signal cable filtering to the FFT spectrum of the waveform. The reason being that the filtering is custom and cannot be applied via built-in tool such as SciPy's butterworth filter.

Sanity check: comparing original waveform with the inverse FFT of the FFT

in theory: A = iFFT(FFT(A))

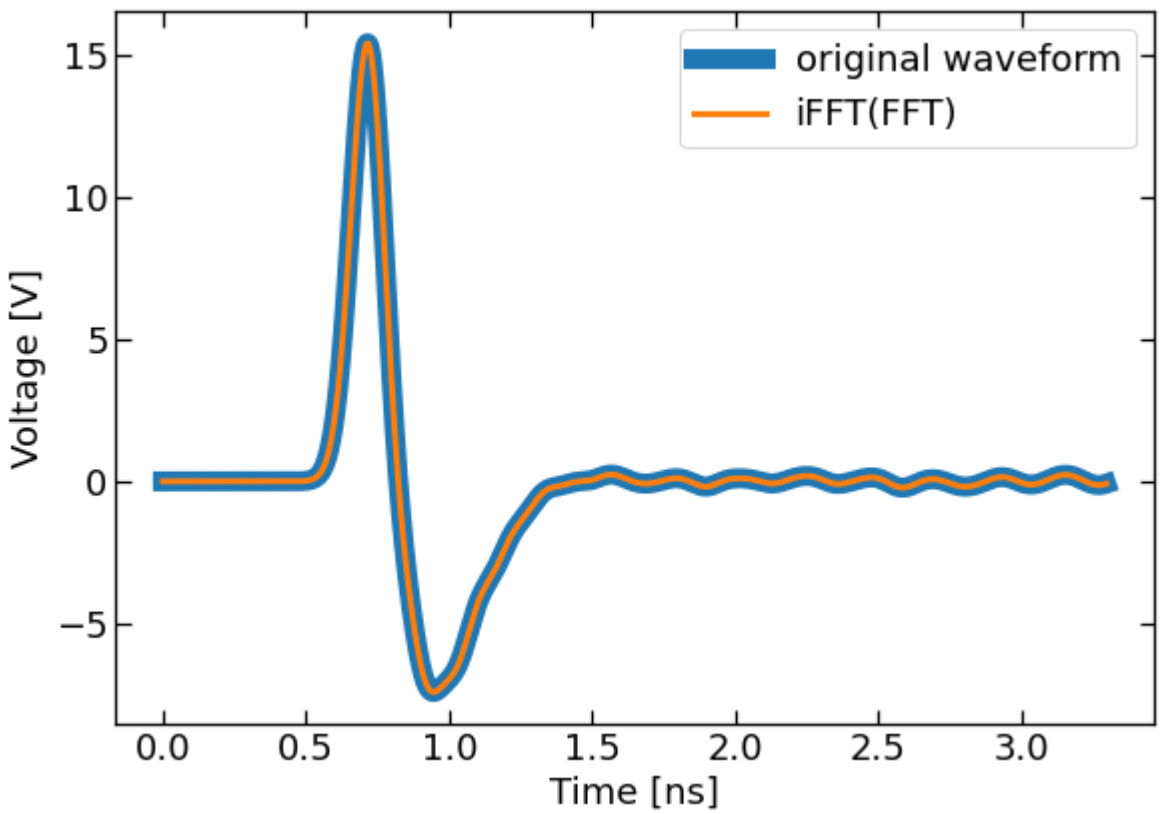
It is an important sanity check since we are going to apply cable filtering to the FFT of the waveform signal and then inverse the modified FFT back to the time domain

In [4]:

```
ifft = np.fft.ifft(np.fft.fft(amp))  # perform inverse FFT on the FFT

style()  # apply plot styling

# do the plotting
plt.plot(time, amp, lw=10, label='original waveform')
plt.plot(time, ifft, lw=3, label="iFFT(FFT)")
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()
```

Out[4]:

```
<matplotlib.legend.Legend at 0x7fc44e6b4a90>
```



Produce the FFT of the waveform and polish it to fit our needs

Because of the Nyquist frequency, the second half of the FFT is the mirror image of the first half. We thus usually discard the second half as it is not physical and provide no extra information. Today though and since we are going to inverse transform the FFT back to the time domain, we want to keep the entire FFT to match back the time domain in term of sampling time interval and other features.

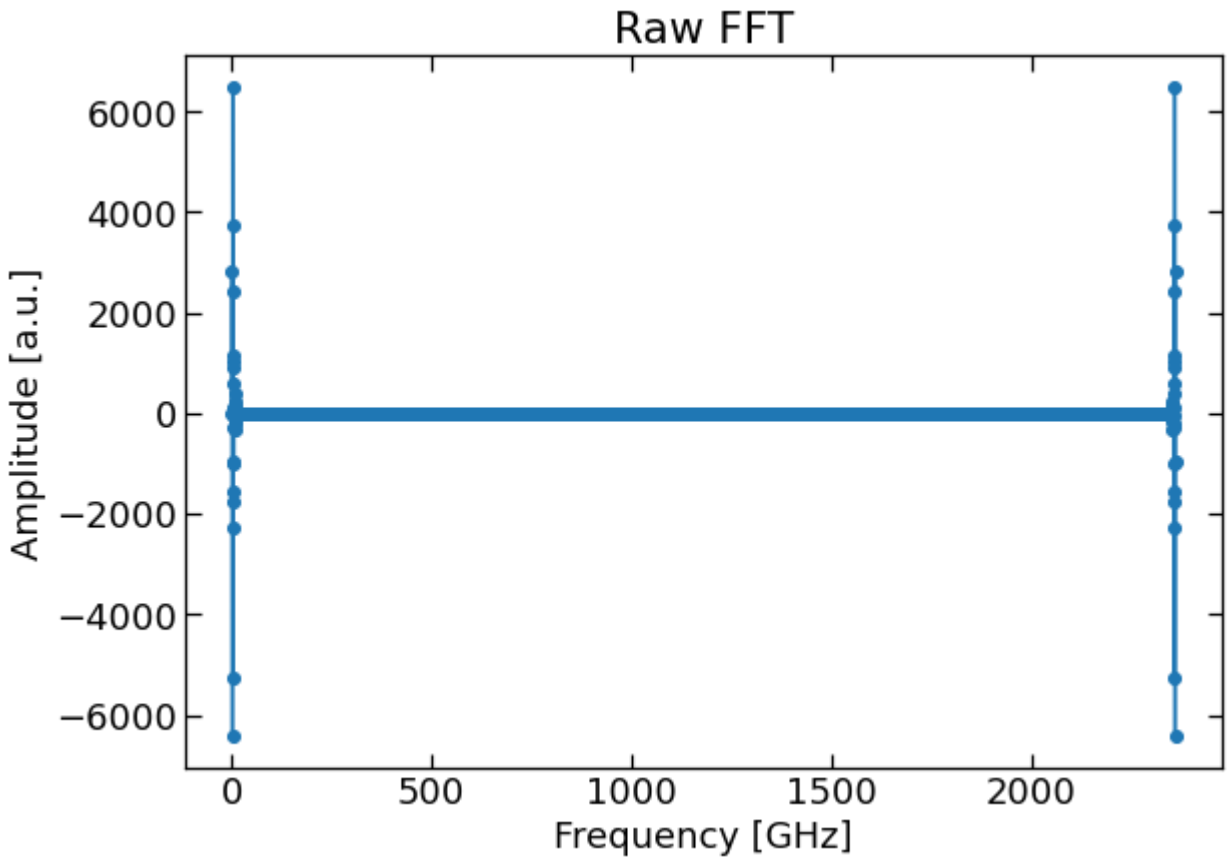
We are going fo apply filtering to the physical frequencies of the FFT between 0 and 10 GHz. In principle, this filtering should also be applied to the mirror frequencies for an accurate FFT inversion. To make our life easier though, we are going to set to zero-amplitude all the frequencies above the Nyquist one. Then, we are going to double the amplitudes for all the frequencies below Nyquist. This way, there will be no loss of information and the normalization back to the time domain will be correct.

In [5]:

```
fft = np.fft.fft(amp)  # produce FFT of waveform

freq_step = 1 / (len(amp) * (time[1]-time[0]))  # retrieve sampling frequency fr
frequencies = np.array([int(i) * freq_step for i in range(0, len(fft))])  # proc

style()  # apply plot styling

# do the plotting
plt.plot(frequencies, fft, 'o-')
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude [a.u.]")
plt.title("Raw FFT")
```

Out[5]:

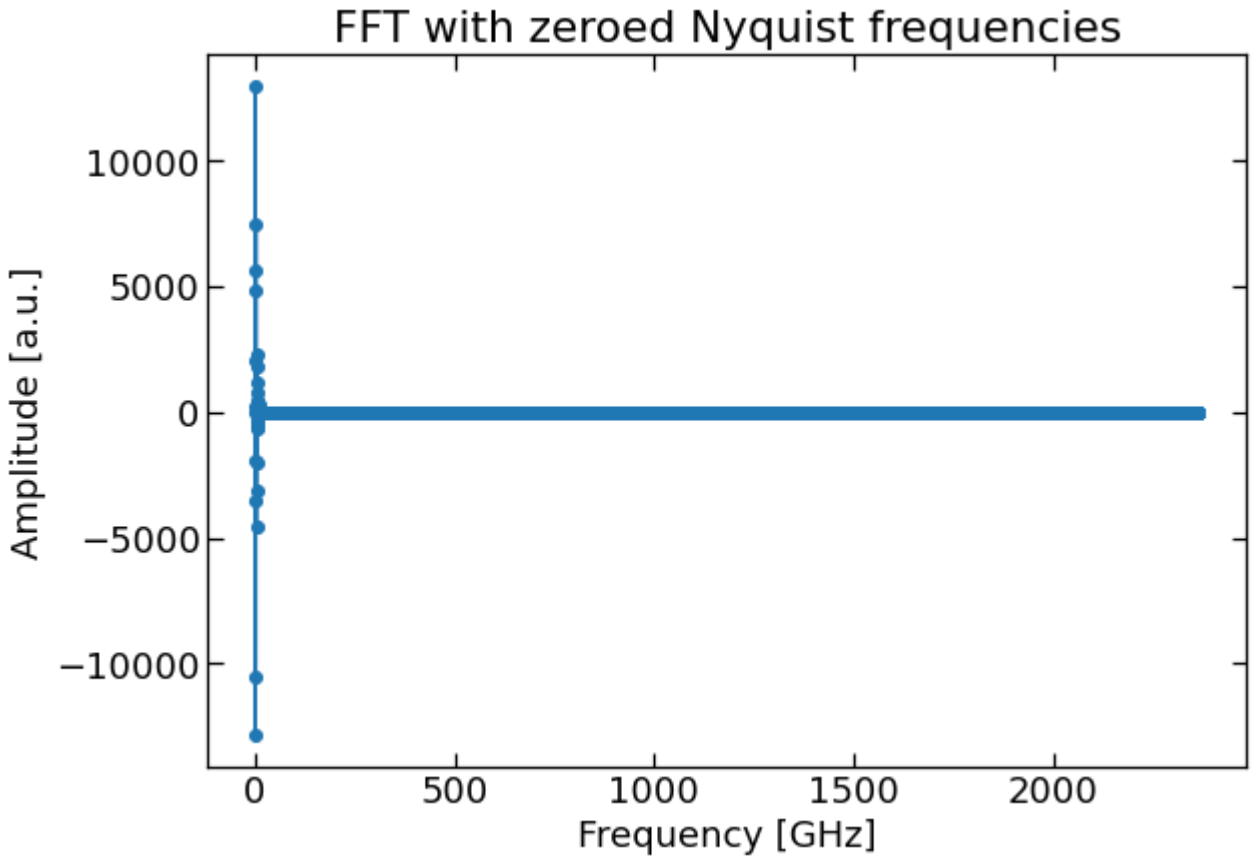
```
Text(0.5, 1.0, 'Raw FFT')
```



In [6]:

```
# zeroing all the frequencies above Nyquist and scale by factor two the ones bel
for idx in range(len(fft)):
    if idx > len(fft)/2:
        fft[idx] = 0
    else:
        fft[idx] *=2

style()  # apply plot styling

# do the plotting
plt.plot(frequencies, fft, 'o-')
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude [a.u.]")
plt.title("FFT with zeroed Nyquist frequencies")
```

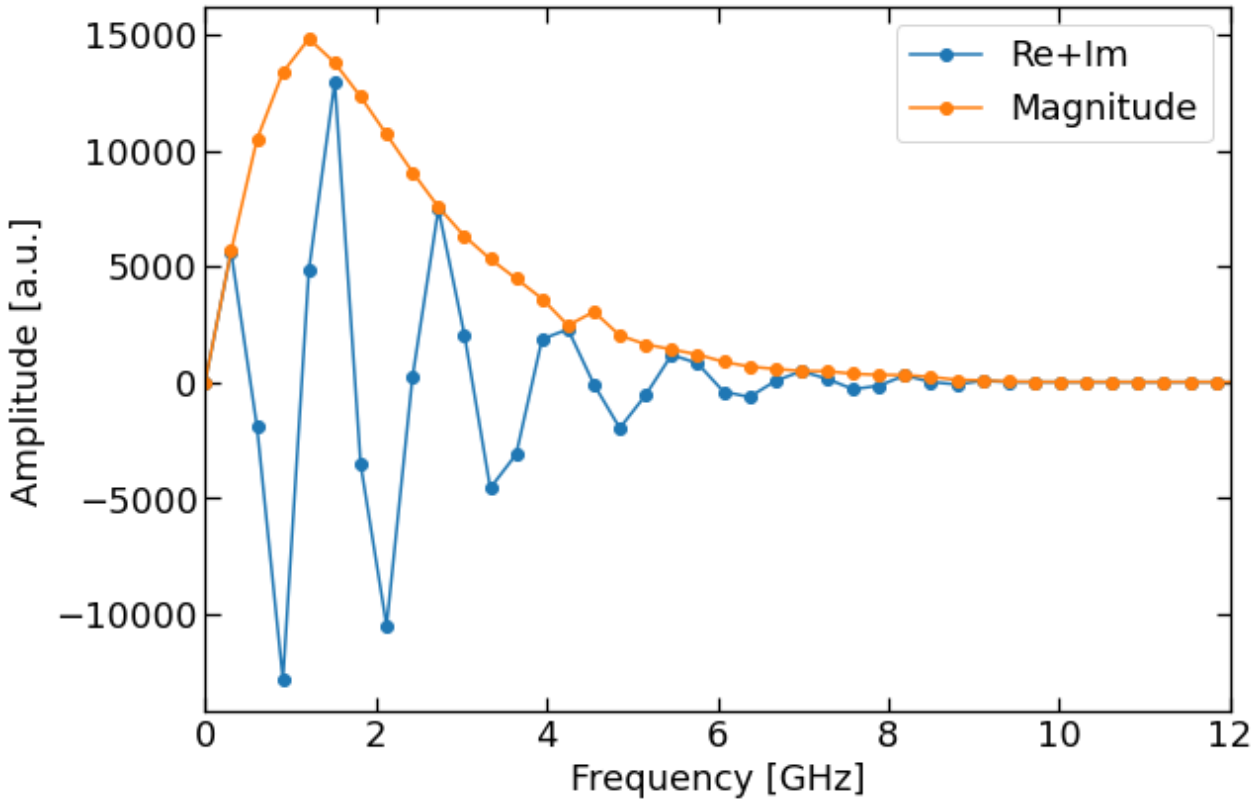
Out[6]: Text(0.5, 1.0, 'FFT with zeroed Nyquist frequencies')



In [7]:

```
style()  # apply plot styling

# do the plotting
plt.plot(frequencies, fft, 'o-', label="Re+Im")
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude [a.u.]")
plt.xlim(0, 12)
plt.plot(frequencies, np.abs(fft), 'o-', label='Magnitude')
plt.legend()
```

Out[7]: <matplotlib.legend.Legend at 0x7fc44e585dc0>

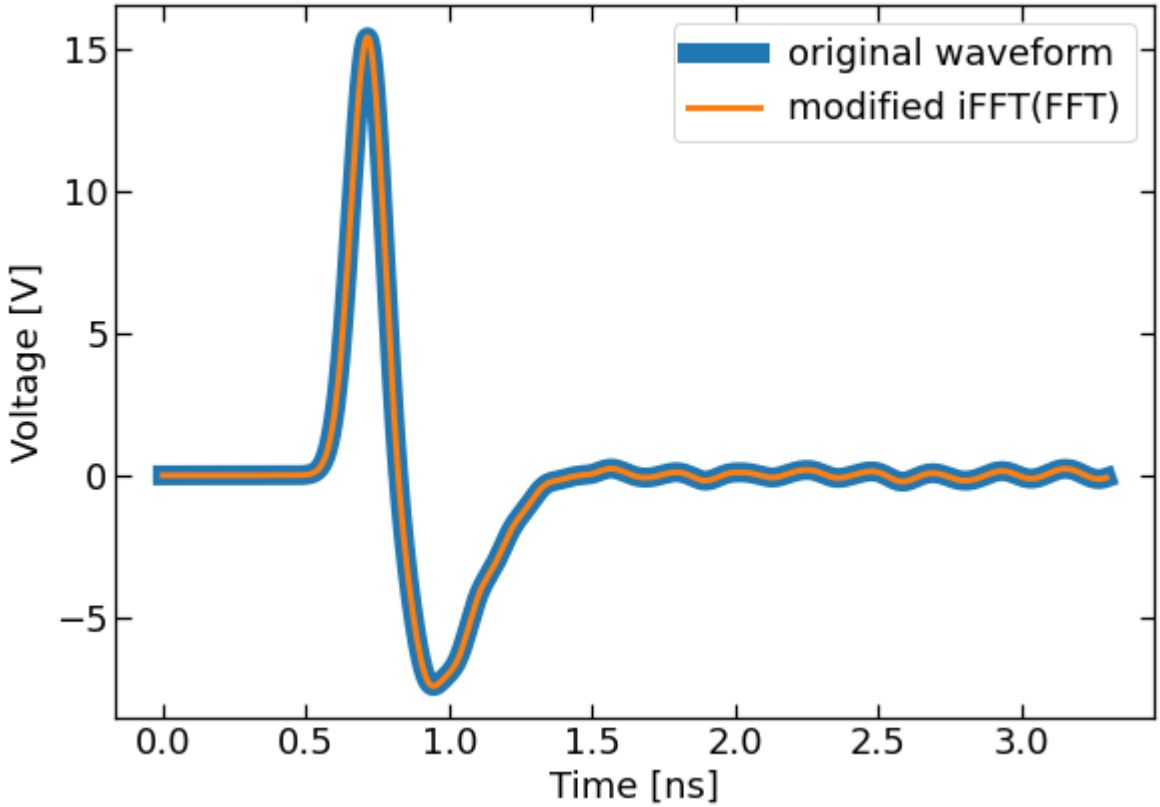


In [8]:

```
# taking the inverse of the modified FFT back to the time domain
ifft = np.fft.ifft(fft)

style()  # apply plot styling

# do the plotting
plt.plot(time, amp, lw=10, label='original waveform')
plt.plot(time, ifft, lw=3, label="modified iFFT(FFT)")
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()
```

Out[8]:

```
<matplotlib.legend.Legend at 0x7fc44e5ade80>
```



The modified FFT yields the correct time domain waveform. We can now apply filtering to the FFT and inverse it to get the waveform as would be measured at the end of the signal cable.

Cable filtering

Bob gave us an analytic expression for cable attenuation at the 12W module:

A = 1.73dB * sqrt(f(GHz))

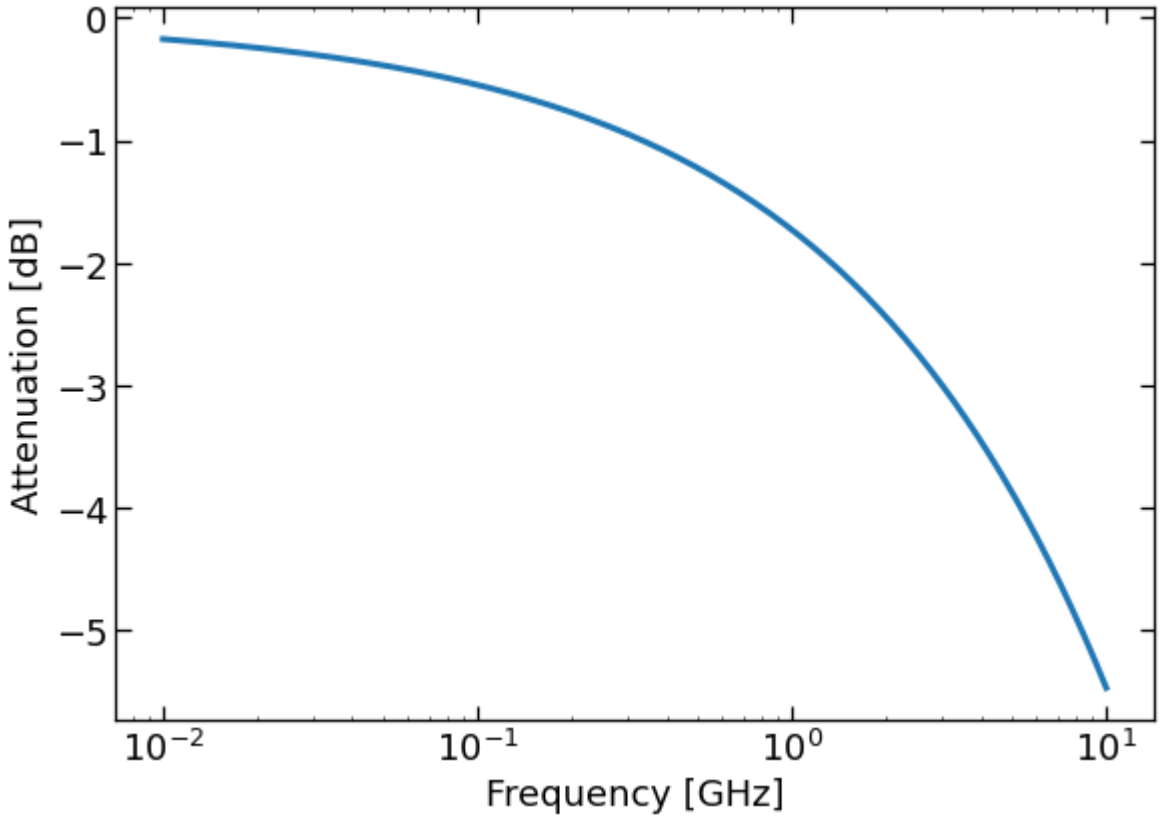
Let's produce the attenuation plot as a function of frequency

In [9]:

```
freq = np.arange(0.01, 10, 0.001)
att = -1.73*np.sqrt(freq)

style()  # apply plot styling

# do the plotting
# plt.plot(freq, att, lw=3)
plt.semilogx(freq, att, lw=3)
plt.xlabel("Frequency [GHz]")
plt.ylabel("Attenuation [dB]")
```

Out[9]:

```
Text(0, 0.5, 'Attenuation [dB]')
```



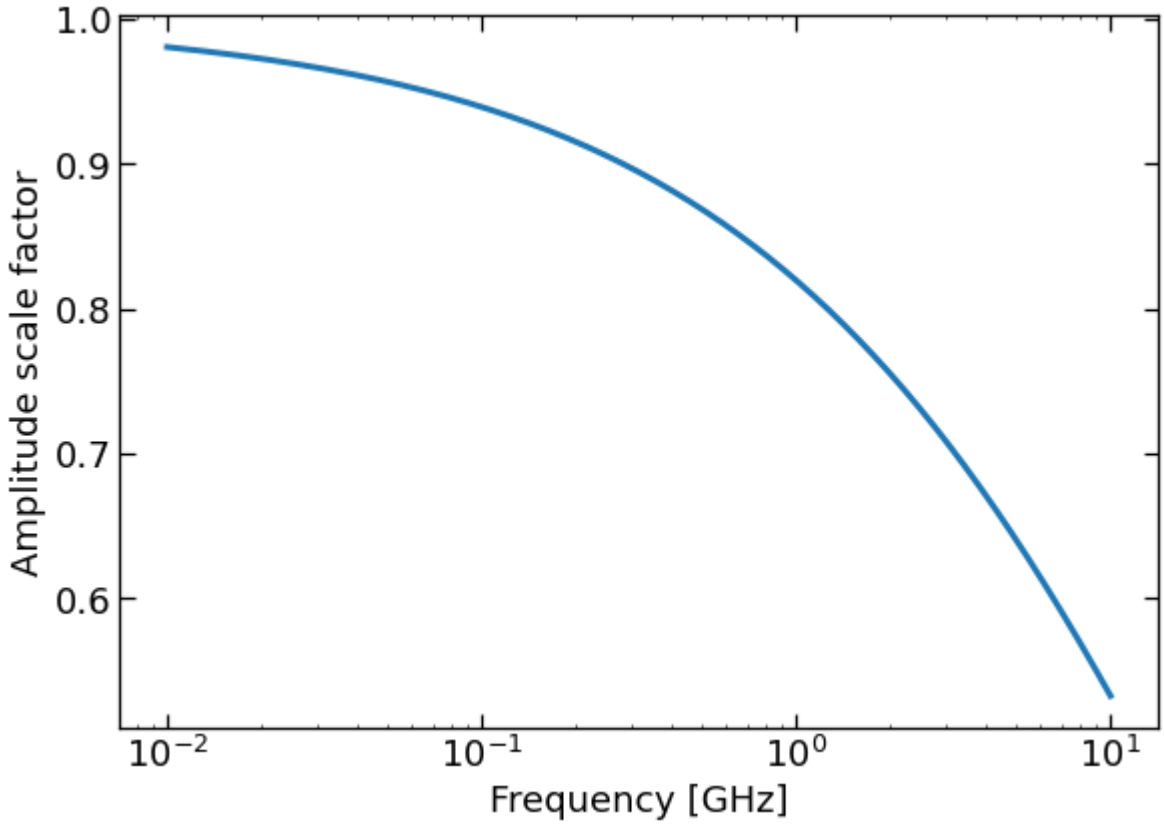
And convert that plot into the amplitude scaling (0.5 for instance mean that the amplitude is halved)

In [10]:

```
amplitude = np.power(10, att/20)

style()  # apply plot styling

# do the plotting
plt.semilogx(freq, amplitude, lw=3)
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude scale factor")
```

Out[10]:

```
Text(0, 0.5, 'Amplitude scale factor')
```



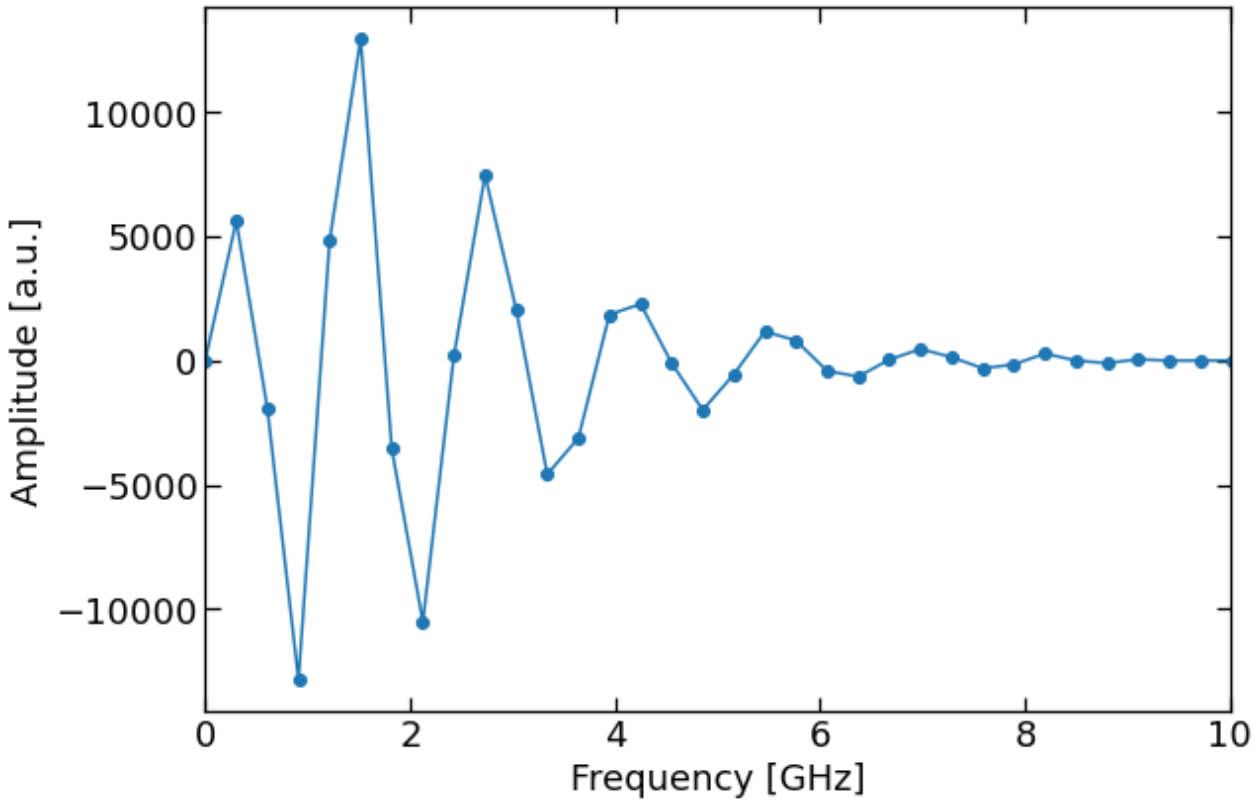
We now have to convoluate that amplitude scaling factor with the FFT where non-zero frequencies are below 10 GHz

In [11]:

```
style()  # apply plot styling

# do the plotting
plt.plot(frequencies, fft, 'o-')
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude [a.u.]")
plt.xlim(0, 10)
```

Out[11]:

```
(0.0, 10.0)
```

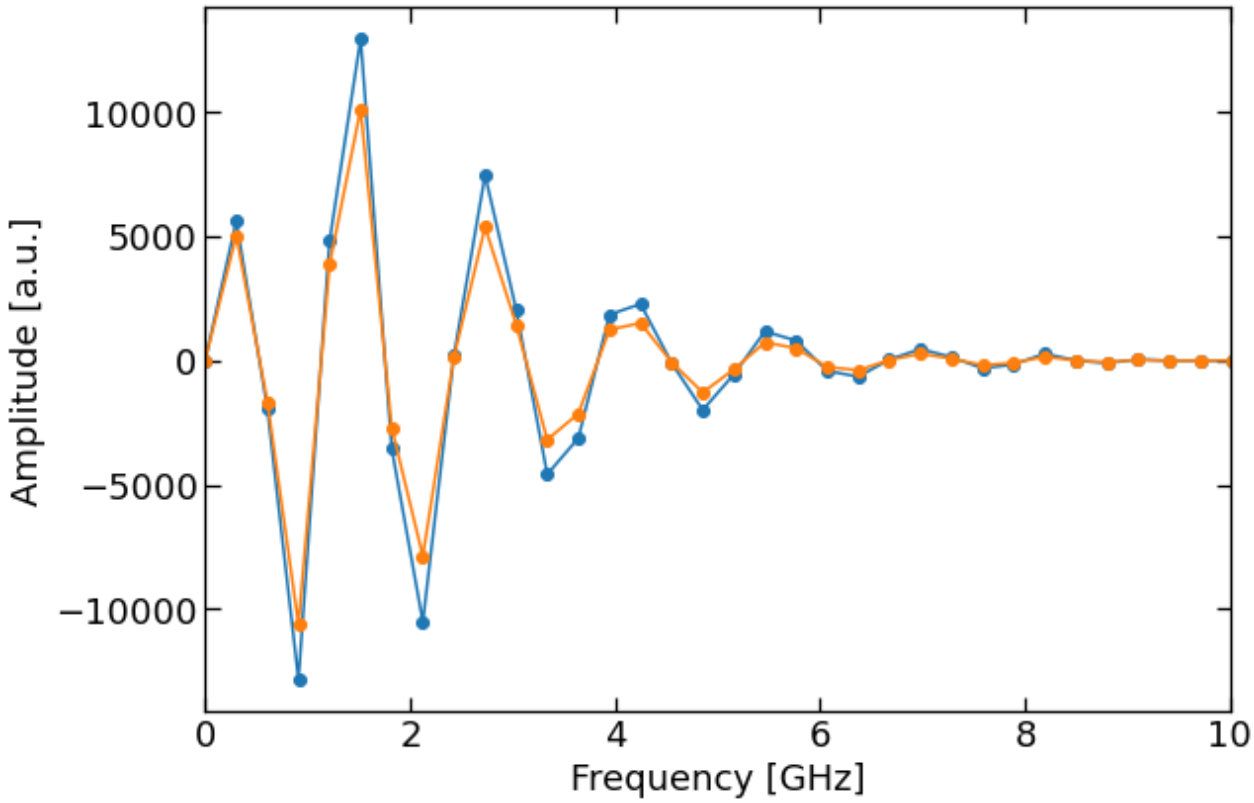


In [12]:

```
filtered_fft = copy.deepcopy(fft)
for idx in range(len(fft)):
    if frequencies[idx] > 15:
        break
    else:
        filtered_fft[idx] *= np.power(10, (-1.73*np.sqrt(frequencies[idx]))/20)

style()  # apply plot styling

# do the plotting
plt.plot(frequencies, fft, 'o-')
plt.plot(frequencies, filtered_fft, 'o-')
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude [a.u.]")
plt.xlim(0, 10)
```

Out[12]:

```
(0.0, 10.0)
```

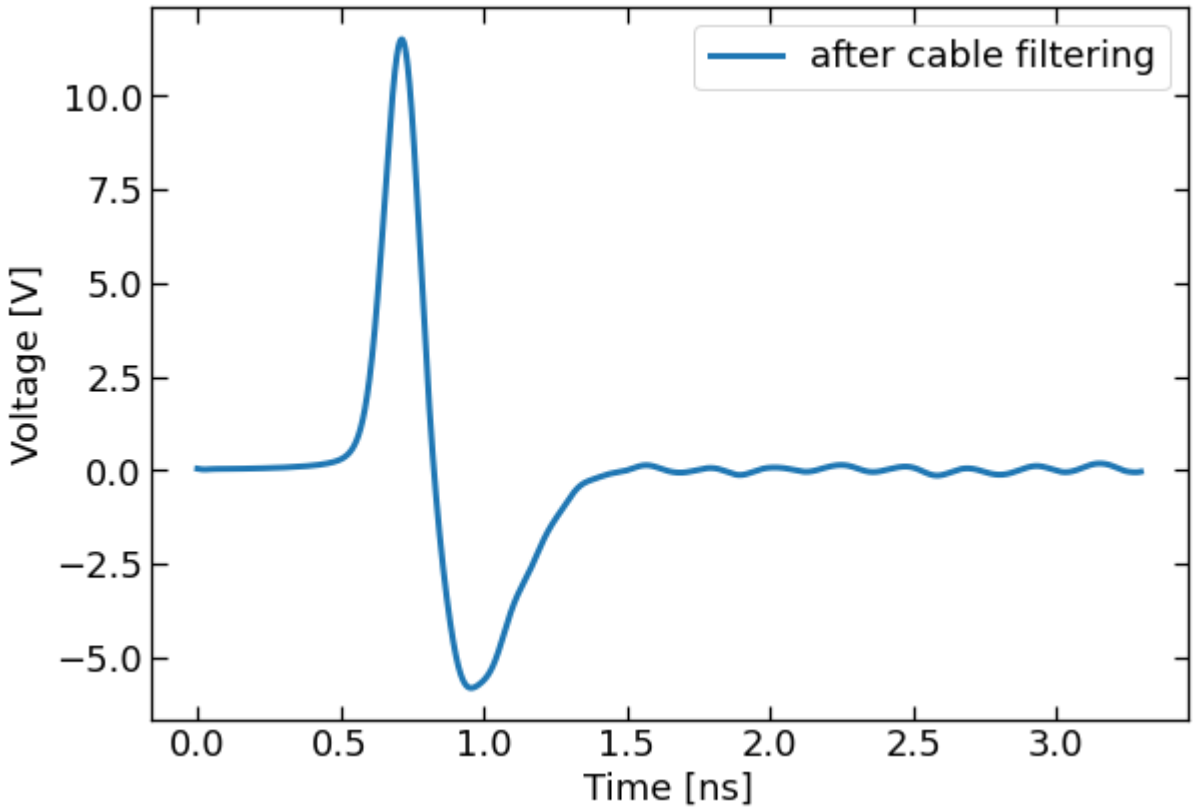


In [13]:

```
# taking the inverse of the filtered FFT back to the time domain
waveform = np.fft.ifft(filtered_fft)

style()  # apply plot styling

# do the plotting
plt.plot(time, waveform, lw=3, label="after cable filtering")
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()
```

Out[13]:

```
<matplotlib.legend.Legend at 0x7fc44e079460>
```



1.2 GHz filtering

This is external to the module, a filter connected to the end of the signal cable. Let's apply a 1.2 GHz Butterworth low-pass filter to the waveform that was already filtered for the cable effect

Butterworth low-pass filter

SciPy provides a built-in Butterworth low-pass filter.

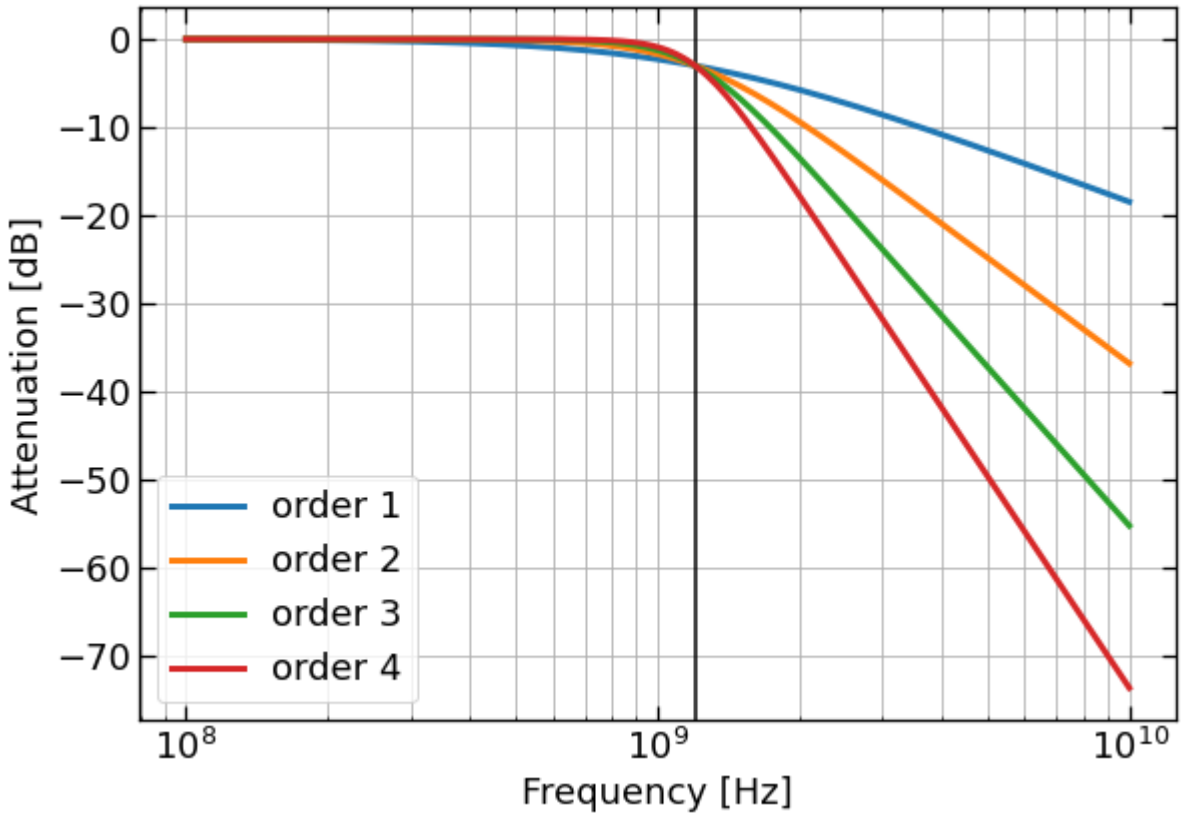
See: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.butter.html

The main parameter is the order of the filter. CST Microwave studio uses second order for instance. Let's plot the attenuation as a function of frequency for a 1.2 GHz cut-off and several orders.

In [14]:

```
style()  # apply plot styling

# do the plotting
plt.grid(which='both', axis='both')
plt.xlabel("Frequency [Hz]")
plt.ylabel("Attenuation [dB]")
plt.axvline(1.2e9, color='black') # cutoff frequency

for order in range(1, 5):  # iterate over filter's order
    b, a = signal.butter(order, 1.2e9, analog=True, output='ba')
    w, h = signal.freqs(b, a)#, worN=10000)
    plt.semilogx(w, 20 * np.log10(abs(h)), lw=3, label="order " + str(order))
#     plt.semilogx(w, np.abs(h), lw=3, label="order " + str(order))

plt.legend()
```

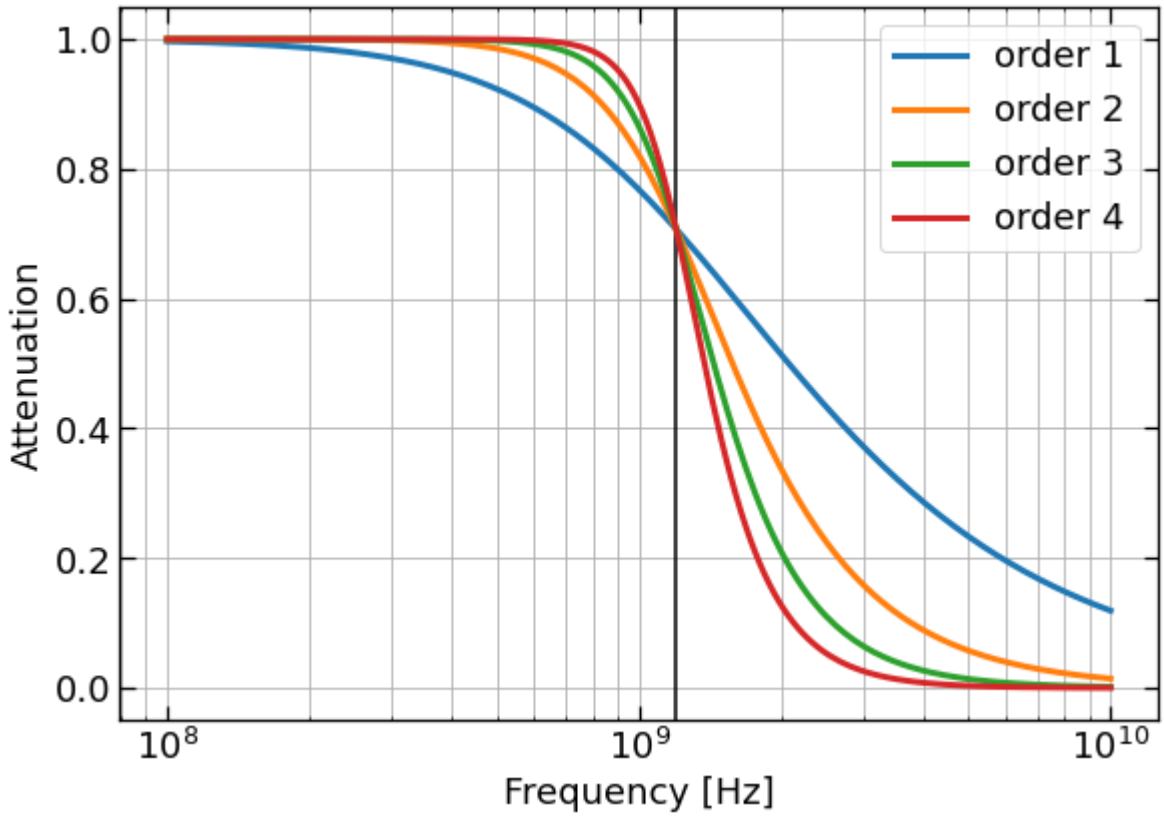
Out[14]: <matplotlib.legend.Legend at 0x7fc44e05cb80>



In [15]:

```
style()  # apply plot styling

# do the plotting
plt.grid(which='both', axis='both')
plt.xlabel("Frequency [Hz]")
plt.ylabel("Attenuation")
plt.axvline(1.2e9, color='black') # cutoff frequency

for order in range(1, 5):  # iterate over filter's order
    b, a = signal.butter(order, 1.2e9, analog=True, output='ba')
    w, h = signal.freqs(b, a)#, worN=10000)
    plt.semilogx(w, np.abs(h), lw=3, label="order " + str(order))

plt.legend()
```

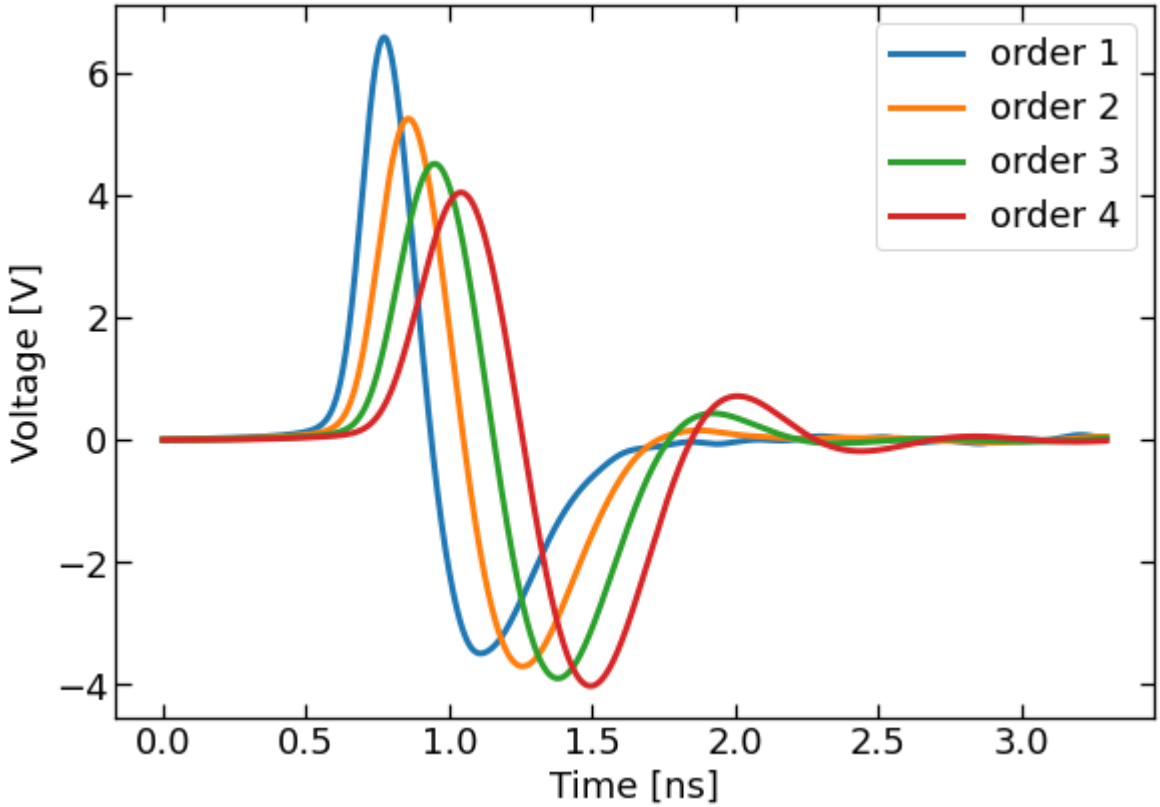
Out[15]: <matplotlib.legend.Legend at 0x7fc44df18eb0>



In [16]:

```
style()

# plt.plot(time, amp, lw=3)
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")

for order in range(1, 5):
    sos = signal.butter(order, 1.2e9, 'lp', fs=1e9/(time[1]-time[0]), output='so
    filtered = signal.sosfilt(sos, waveform)
    plt.plot(time, filtered, lw=3, label="order " + str(order))

plt.legend()
print(sos)
```

```
[[ 6.46549815e-12  1.29309963e-11  6.46549815e-12  1.00000000e+00
  -1.99410820e+00  9.94118358e-01]
 [ 1.00000000e+00  2.00000000e+00  1.00000000e+00  1.00000000e+00
  -1.99754936e+00  9.97559540e-01]]
```



Comparing results

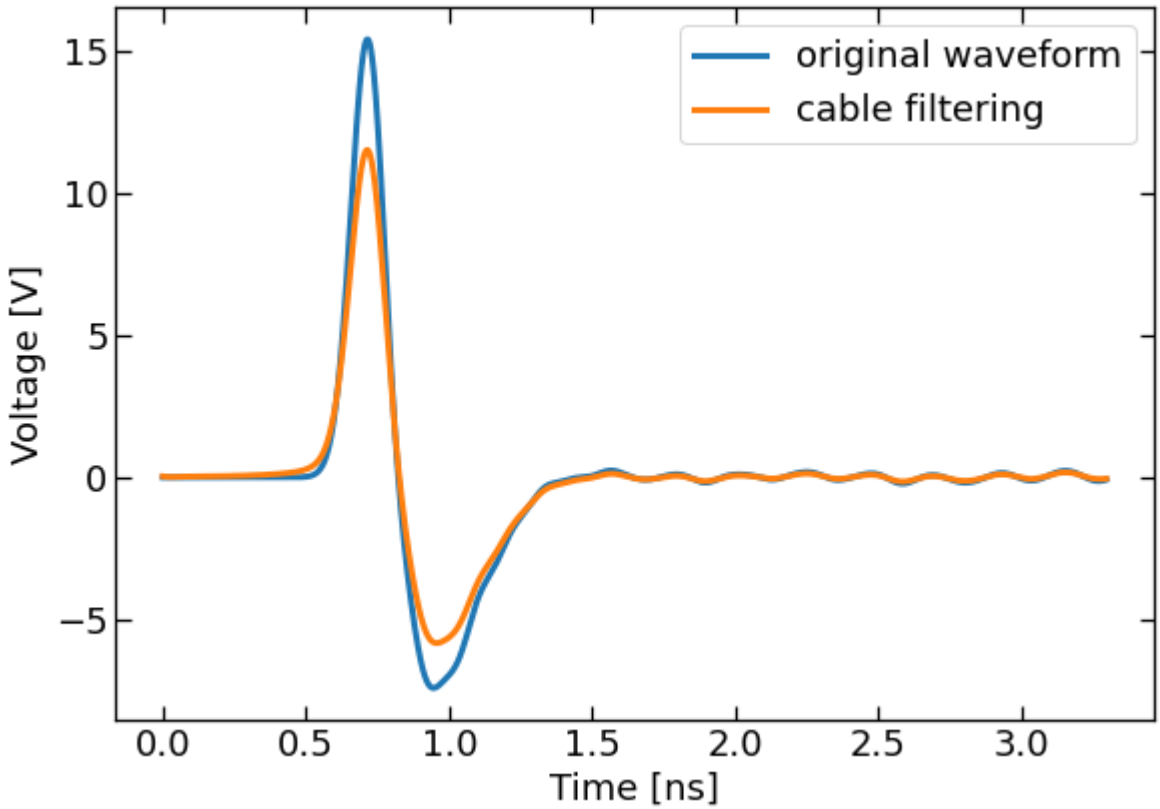
Waveform before after cable filtering

In [17]:

```
style()  # apply plot styling

# do the plotting
plt.plot(time, amp, lw=3, label='original waveform')
plt.plot(time, waveform, lw=3, label="cable filtering")
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()
```

Out[17]:

```
<matplotlib.legend.Legend at 0x7fc44e2139a0>
```



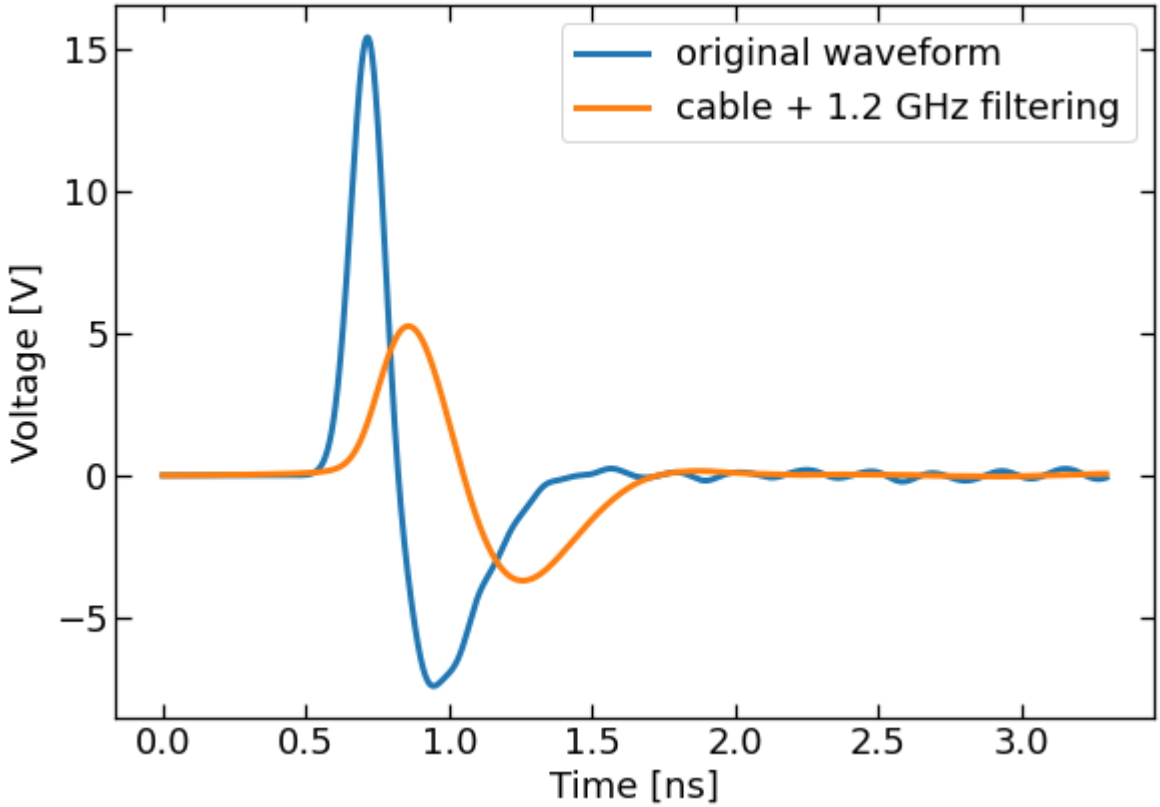
Waveform before filtering, and after cable + 1.2 GHz filtering (second order filter)

In [18]:

```
style()  # apply plot styling

# do the plotting
plt.plot(time, amp, lw=3, label='original waveform')

sos = signal.butter(2, 1.2e9, 'lp', fs=1e9/(time[1]-time[0]), output='sos')
filtered = signal.sosfilt(sos, waveform)
plt.plot(time, filtered, lw=3, label="cable + 1.2 GHz filtering")

plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()
```

Out[18]:

```
<matplotlib.legend.Legend at 0x7fc44e105f10>
```



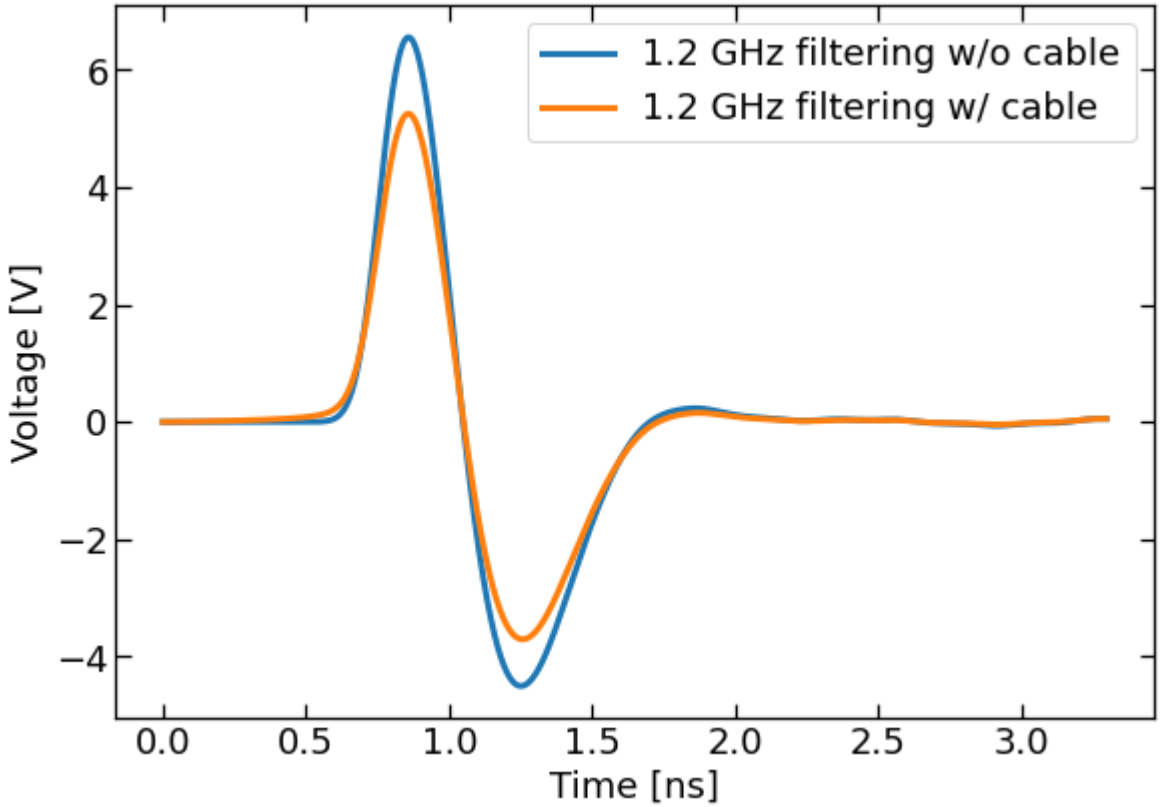
Waveform after 1.2 GHz filter, with and without cable filtering

In [19]:

```
style()  # apply plot styling

# do the plotting
sos = signal.butter(2, 1.2e9, 'lp', fs=1e9/(time[1]-time[0]), output='sos')

filtered = signal.sosfilt(sos, amp)
plt.plot(time, filtered, lw=3, label="1.2 GHz filtering w/o cable")

filtered = signal.sosfilt(sos, waveform)
plt.plot(time, filtered, lw=3, label="1.2 GHz filtering w/ cable")

plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()
```

Out[19]:

```
<matplotlib.legend.Legend at 0x7fc44ddbfeb0>
```



FFT filtering cross-check

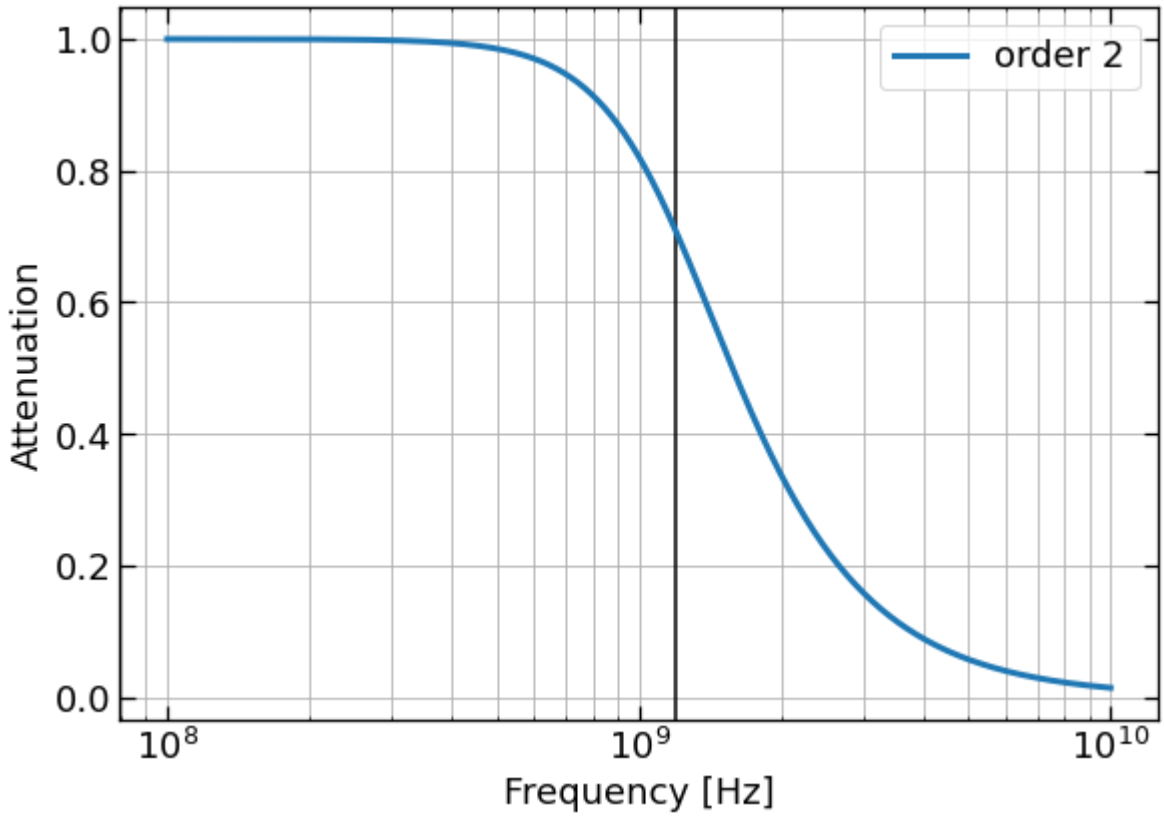
Let's make sure that the FFT filtering is doing the right thing by applying the 1.2 GHz filter the FFT route and the SciPy route. Let's use the second order Butterwoth filter.

In [20]:

```
style()  # apply plot styling

# do the plotting
plt.grid(which='both', axis='both')
plt.xlabel("Frequency [Hz]")
plt.ylabel("Attenuation")
plt.axvline(1.2e9, color='black') # cutoff frequency

for order in range(2, 3):  # iterate over filter's order
    b, a = signal.butter(order, 1.2e9, analog=True, output='ba')
    w, h = signal.freqs(b, a, worN=20000)
    plt.semilogx(w, np.abs(h), lw=3, label="order " + str(order))

plt.legend()
w *= 1e-9  # make GHz the natural unit
len(w)
```

Out[20]:

```
20000
```

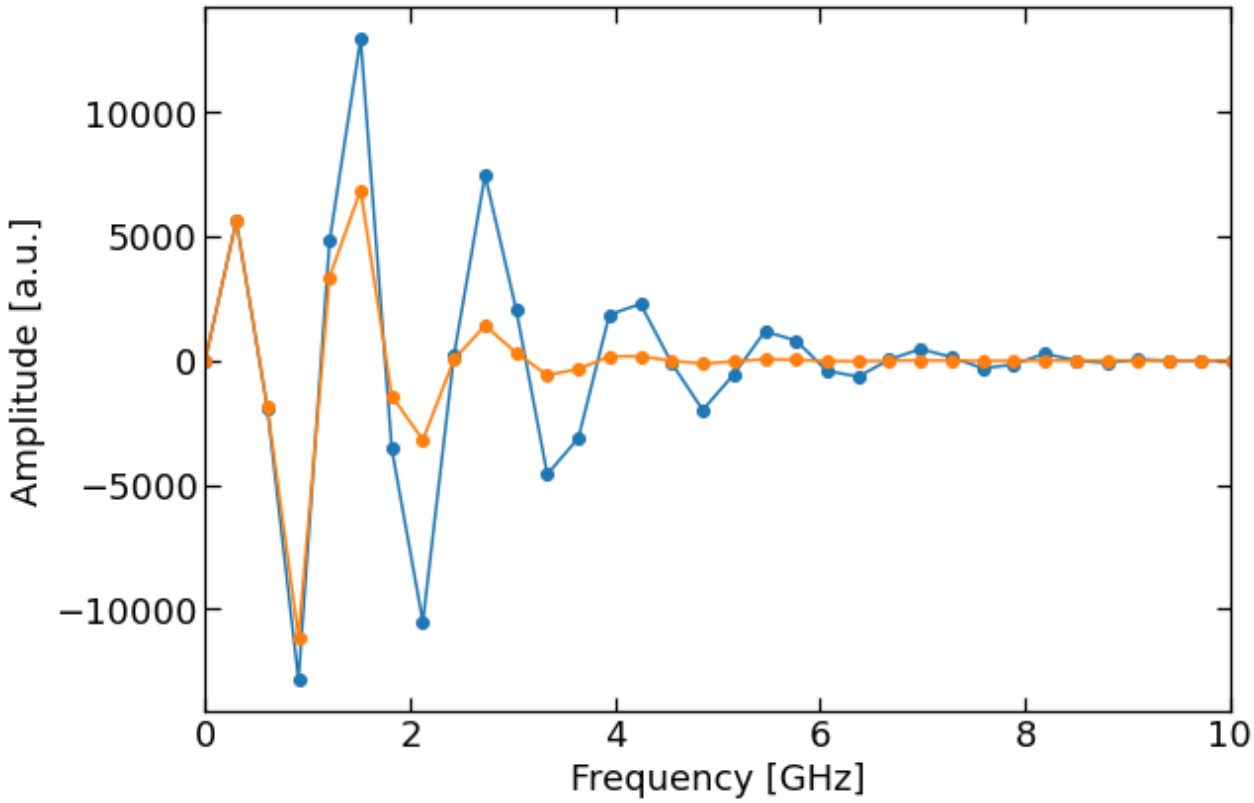


In [21]:

```
fft = np.fft.fft(amp)

# Getting rid of Nyquist image and double amplitude of physical frequencies
for idx in range(len(fft)):
    if idx > len(fft)/2:
        fft[idx] = 0
    else:
        fft[idx] *=2

filtered_fft = copy.deepcopy(fft)

def find_nearest_idx(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx

for idx in range(len(fft)):
    if frequencies[idx] > 15:
        break
    else:
        filtered_fft[idx] *= np.abs(h[find_nearest_idx(w, frequencies[idx])])  #

style()  # apply plot styling

# do the plotting
plt.plot(frequencies, fft, 'o-')
plt.plot(frequencies, filtered_fft, 'o-')
plt.xlabel("Frequency [GHz]")
plt.ylabel("Amplitude [a.u.]")
plt.xlim(0, 10)
```

Out[21]:

```
(0.0, 10.0)
```

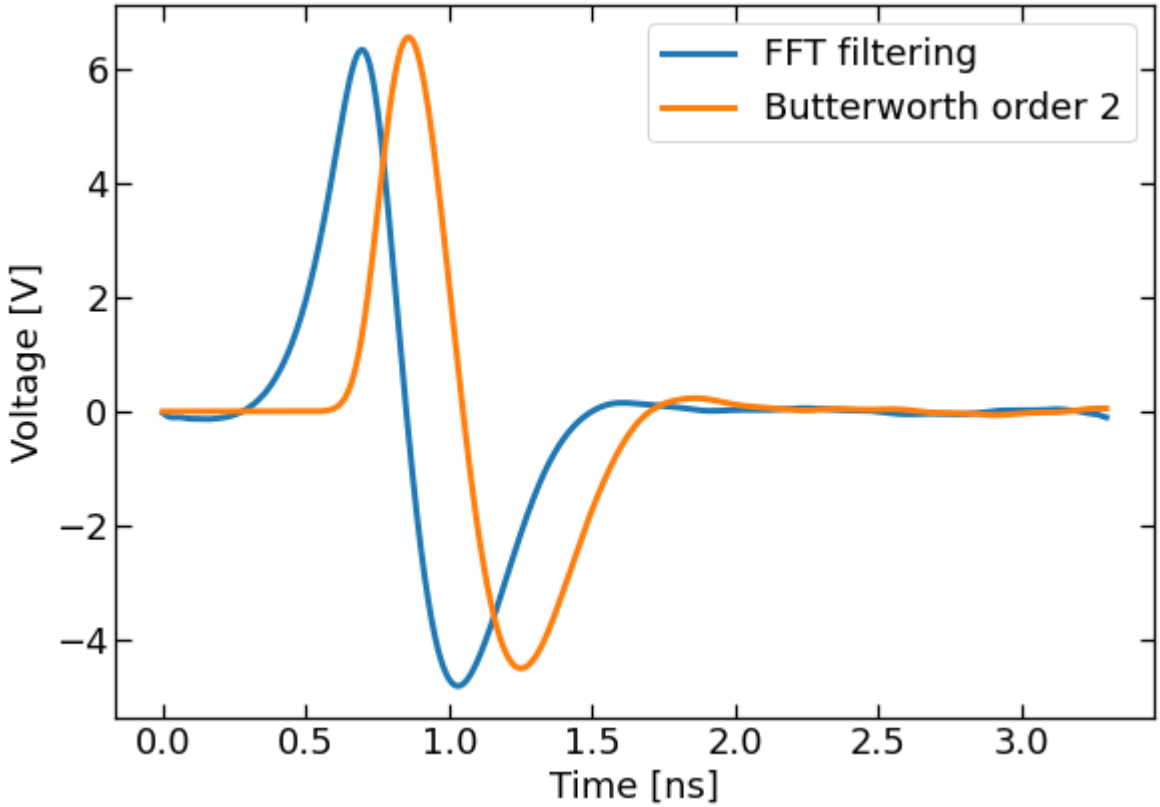


In [22]:

```
# taking the inverse of the filtered FFT back to the time domain
waveform = np.fft.ifft(filtered_fft)

style()  # apply plot styling

# do the plotting
plt.plot(time, waveform, lw=3, label="FFT filtering")
plt.xlabel("Time [ns]")
plt.ylabel("Voltage [V]")
plt.legend()

for order in range(2, 3):
    sos = signal.butter(order, 1.2e9, 'lp', fs=1e9/(time[1]-time[0]), output='so
    filtered = signal.sosfilt(sos, amp)
    plt.plot(time, filtered, lw=3, label="Butterworth order " + str(order))

plt.legend()
```

Out[22]:

```
<matplotlib.legend.Legend at 0x7fc44db85e80>
```


FFT filtering
Butterworth order 2
Voltage [V]
Time [ns]
6
4
2
0
−2
−4
0.0
0.5
1.0
1.5
2.0
2.5
3.0

In [ ]: