Fourier

October 12, 2017

1 Baseband bit transmission and Fourier transforms

In this assignment, we will look at how signals propagate over a bandlimited channel and what that does to bitshapes. We start by looking first at simple sinusoidal functions to get an understanding of forward and inverse Fourier transforms. Then, we will look at actual bit sequences to transmit.

We need numpy (numerics for Python, http://www.numpy.org) and in particular FFT routines (https://docs.scipy.org/doc/numpy/reference/routines.fft.html). We use Matplotlib for plotting.

```
In [59]: # Imports for numerical library and plotting library:

         import numpy as np

         # the following line is necessary to use plotting in Jupyter notebook
         # remove if you use this code outside a notebook
         %matplotlib notebook
         import matplotlib.pyplot as plt
```

1.1 Helper functions

We start with two simple helper functions.

```
In [134]: def fft(signal, dT, periods):
              """For a given signal as an array of samples and the number of
              samples contained in this signal per unit time,
              compute both the values of the Fourier transform and the
              corresponding frequencies at which these values hold.
              """

              t = np.arange(len(signal))
              spectrum = np.fft.fft(signal) * dT/periods
              freq = np.fft.fftfreq(t.shape[-1], d=dT)

              return freq, spectrum

          def plot_signal_fft(time, signal, freq, spectrum, cutoff_freq):
              """Given a signal and its frequency transform, plot both"""
```

```
    # create two figures:
    f, (timeplot, freqplot) = plt.subplots(2)
    f.subplots_adjust(hspace=.5)

    # Plot the time-domain representation:
    # t = np.arange(len(signal))
    timeplot.plot(time, signal)
    timeplot.set_title("Time domain")
    timeplot.set_xlabel("Time")
    timeplot.set_ylabel("Amplitude")

    # concentrate on the positive part,
    # take mirror frequencies into account by doubling the
    # positive part of the specturm,
    # plot POWER (square of absolute value!)
    # upper = int(len(freq)/2)
    cutoff = min(cutoff_freq, len(freq), len(spectrum))

    freqplot.plot(freq[0:cutoff], np.abs(2*spectrum[0:cutoff])**2,
                  drawstyle="steps-mid")
    freqplot.set_title("Frequency domain")
    freqplot.set_xlabel("Frequency")
    freqplot.set_ylabel("Power per frequency")
```

Note that the coefficients of the Fourier transform will typically be complex numbers. For the signals we use here, however, the complex parts are usually negligible (and do to rounding errors), so we will only plot the absolute values.

1.2 Fourier transform for sinusoidal functions

We start by looking at simple sinus-shaped (sinusoidal) functions. Below, you can generate a signal by adding up two sine functions. Play with the parameters!

Add code below to overlap a second sinusoidal function onto the first one!

```
In [137]: # Frequencies for sinusoidals to overlap:
          freq1 = 3
          freq2 = 10

          # samples per time unit (=per period of the signal)
          T = 128

          # how many time units (=periods of the signal) to look at?
          timeunits = 1

          # get the values for the time axis
          # t = np.arange(T*timeunits)
          t, dT = np.linspace(0, timeunits,
```

```
                                num=T*timeunits,
                                endpoint=False, retstep=True)

        # one sinusoidal:
        signal = 2*np.cos(freq1*2*np.pi*t)

        # Hier Code einfügen, um dem Signal eine zweite Sinusoide
        # zu überlagern
        ### BEGIN SOLUTION
        signal += np.cos(freq2*2*np.pi*t)
        ### END SOLUTION

        # Fouriertransformation berechnen:
        freq, signal_spectrum = fft(signal, dT, timeunits)

        # und plotten:
        plot_signal_fft(t, signal, freq, signal_spectrum, 20*timeunits)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.2.1 Undo FFT

Let's undo it by inverse FFT; it should be the same function again. See https://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.ifft.html#numpy.fft.ifft

In [141]:
```
# inverse FFT
# Hier Code ergänzen, so dass in der Variable signal2
# wieder eine Zeitreihe steht. Es sollte signal2 == signal gelten
# (im Rahmen numerischer Genauigkeit, und abgesehen davon,
# dass signal2 komplexe Zahlen beinhalten wird)
# Hinweis: Lesen Sie unter Normalisierung der numpy-fft Routinen nach:
# https://docs.scipy.org/doc/numpy/reference/routines.fft.html#module-nu

### BEGIN SOLUTION
signal2 = np.fft.ifft(signal_spectrum)*len(signal_spectrum)
### END SOLUTION

# Ein neues Bild erzeugen:
plt.figure()

# Und darin plotten Sie den realteil von signal2:
### BEGIN SOLUTION
plt.plot(t, np.real(signal2))
### END SOLUTION
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.3 A simple channel modell

Trivial model: cutoff frequency; any power in higher frequencies is multiplied by a given factor (possibly 0).

Hint: `freq` will contain positive and negative frequencies; watch out when comparing against cutoff. See https://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.fft.html#numpy.fft.fft

```
In [142]: def channel(sp, freq, cutoff, factor=1):
              """Gib eine Liste der Werte zurück, bei denen Werte
              für Frequenzen (dem Betrag nach) kleiner als die cutoff
              Frequenz nicht verändert sind und ansonsten mit dem factor
              mulitpliziert werden"""

              ### BEGIN SOLUTION
              spnew = np.array([
                  s if abs(f) <= cutoff else s*factor
                  for s, f in zip(sp, freq)])
              return spnew
              ### BEGIN SOLUTION
```

1.3.1 Attenuation

Use the function `channel` to attenuate all frequencies larger than 8 by a factor 1/4. Put result into variable `spattenuated`.

```
In [143]: ### BEGIN SOLUTION
          spectrum_attenuated = channel(signal_spectrum, freq, 8, 0.5)
          ### END SOLUTION
```

1.3.2 Plot attenuated frequencies

Create a plot of the *power* spectrum of both the original and the attenuated signal. Watch out to correct for negative frequencies.

```
In [150]: plt.figure()
          ### BEGIN SOLUTIOM
          plt.plot(freq[0:20],
                   np.abs(2*signal_spectrum[0:20])**2,
                   freq[0:20]+0.2,
                   np.abs(2*spectrum_attenuated[0:20])**2)
          ### END SOLUTION
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[150]: [<matplotlib.lines.Line2D at 0x106e85358>,
           <matplotlib.lines.Line2D at 0x106e85438>]
```

1.3.3 The attenuated signal

Use `ifft` to convert the spectrum after attenuation back in a time-series representation. Hint: `ifft` will produce complex numbers; we just need the real part of those numbers (imaginary part is negbiligibly small and just comes from rounding errors). And think about normalization again!

Plot the resulting signal in time.

```
In [153]: ### BEGIN SOLUTION
          signal_attenuated = np.real(np.fft.ifft(spectrum_attenuated)*len(spectrum

          plot_signal_fft(t, signal_attenuated, freq, spectrum_attenuated, 20*timeu
          ### END SOLUTION
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

1.4 Bits statt Sinus-Funktionen

Wiederholen wir das ganze, aber nun nehmen wir eine Folge von Bits, die wir überbtragen wollen.

Die folgenden Parameter beschreiben die Bitfolge:

```
In [154]: bits = np.array([0, 1, 1, 0, 0, 0, 1, 0])
```

Wir brauchen noch mehere Stützpunkte pro Bit, damit wir den Zeitverlauf der Funktion besser approximieren können:

```
In [155]: samples_per_bit = 2**10 # power of two!
```

Die folgende Hilfsfunktion gibt Ihnen eine entsprechende Folge von Werte für die Bitfolge, mit entsprechender Anzahl Samples pro Bits:

```
In [156]: from itertools import chain

          def sample_bits(bits, samples_per_bit):
              return np.array(list((chain.from_iterable(
                      [[x] * samples_per_bit for x in bits ]))))
```

Nutzen Sie die obigen Funktionen: Legen Sie in einer Variable `sampled_bits` eine entsprechende Folge von 0/1 Werte ab. Erstellen Sie ein Fourier-Spektrum und lassen es mitsamt der Zeitdarstellung plotten; nutzen Sie wieder `plot_signal_fft`.

```
In [175]: ### BEGIN SOLUTION
          sampled_bits = sample_bits(bits, samples_per_bit)
          num_bits = len(bits)
          t, dT = np.linspace(0, num_bits,
                              num=samples_per_bit*num_bits,
                              endpoint=False, retstep=True)

          freq, sp = fft(sampled_bits, dT, num_bits)
          plot_signal_fft(t, sampled_bits, freq, sp, 100)
          ### END SOLUTION
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>
```

1.4.1 Dämpfung bei Bit-Übertragung

Nutzen Sie die Funktion `channel`, um die Übertragung eines Bits über einen gedämpften Kanal zu realisieren. Schauen Sie sich das Spektrum aus dem vorherigen Teil an, um sinnvolle Werte für die cutoff-Frequenz zu finden. Nutzen Sie dann `ifft`, um das gedämpfte Signal wieder in der Zeitreihendarstellung zu erhalten (legen Sie es in Variable `attenuated_bits` ab. Plotten Sie.

Experimentieren Sie!

```
In [184]: ### BEGIN SOLUTION
          attenuated_bit_fft = channel(sp, freq, cutoff=2, factor=0)
          attenuated_bits = np.real(np.fft.ifft(attenuated_bit_fft)*len(attenuated_
          ### END SOLUTION

          # In attenuated_bits sollte die Folge der Samples nach gedämpfter
          # Übertragung stehen
          plt.figure()
          plt.plot(np.arange(len(sampled_bits)), np.real(attenuated_bits))
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


Out[184]: [<matplotlib.lines.Line2D at 0x10e8fe5c0>]
```

1.4.2 Rauschen

Fügen Sie Ihren Bits noch zufälliges Rauschen hinzu nach dem AWGN-Modell. (Hinweis: https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.normal.html) Experimentieren Sie mit unterschiedlicehn Werten für die Standardabweichung. Überlegen Sie, ob Sie das Rauschen in der Zeit- oder der Frequenzdarstellung hinzufügen müssen.

Legen Sie die Werte in der Variablen `noisy_bits` ab.

In [188]:
```
# add noise
### BEGIN SOLUTION
noisy_bits = attenuated_bits + np.random.normal(
    scale=0.25, size=len(attenuated_bits))
### END SOLUTION

f, ax = plt.subplots(3, sharex=True)
ax[0].plot(sampled_bits)
ax[0].set_title("Original bits")

ax[1].plot(attenuated_bits)
ax[1].set_title("Attenuated bits")

ax[2].plot(noisy_bits)
ax[2].set_title("Noisy bits")
```

```
<IPython.core.display.Javascript object>


<IPython.core.display.HTML object>


Out[188]: <matplotlib.text.Text at 0x1112d4278>
```