# FFT-Rechteck

November 25, 2022

*Notebook erstellt von A. Naber am 16.11.2022, zuletzt überarbeitet am 24.11.2022*

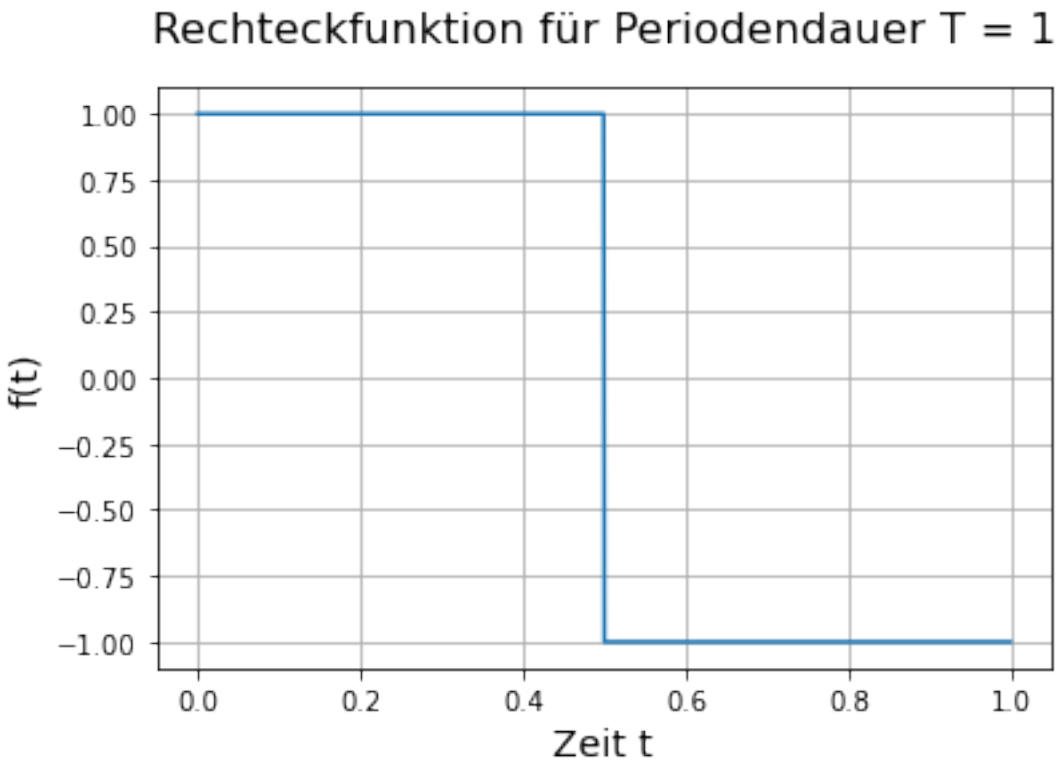
## 1 Fouriertransformation einer periodischen Rechteckfunktion

Eine Rechteckfunktion als Funktion der Zeit $t$ mit Periodendauer $T$ sei gegeben durch

$$f(t) = \begin{cases} +1 & ; \quad 0 \leq t < \dfrac{T}{2} \\ -1 & ; \quad \dfrac{T}{2} \leq t < T \end{cases}$$

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy.fft import fft, ifft, fftfreq

plt.rcParams["axes.grid"] = True

N = 1024      # Anzahl an Punkten
T = 1         # Periodendauer
phi = 0       # Phase
noise = 0.0   # Stärke des Rauschens

# Werte für Zeitachse
t = np.linspace(0, T, N, endpoint=False)

# erzeugt Rechtecksignal
f = signal.square(2 * np.pi * t + phi)

# man kann der Funktion Rauschen überlagern
f += np.random.uniform(-noise / 2, noise / 2, N)

plt.plot(t, f)
plt.xlabel("Zeit t", fontsize=14)
plt.ylabel("f(t)", fontsize=14)
```

```
plt.title(f"Rechteckfunktion für Periodendauer T = {T}", fontsize=16, y=1.05)
plt.show()
```




### 1.1 Analytische Berechnung der Fouriertransformation

Die Funktion ist periodisch in der Zeit $t$ mit Periodendauer $T$, also $f(t+T) = f(t)$, und ist definiert für *alle Zeiten* $t$ von $-\infty$ bis $+\infty$. Dies ist eine Voraussetzung für die Berechnung der Fourierreihe. Der Graph zeigt nur einen Ausschnitt von 0 bis $T$, in dem aber alle Informationen enthalten sind. Wir werden nun zunächst die Fourierreihe der Rechteckfunktion analytisch berechnen. Es gilt mit $\omega = 2\pi/T$ und $m \in \mathbb{N}$ (vgl. Vorlesung)

$$f(t) = \frac{1}{2}A_0 + \sum_{m=1}^{\infty} A_m \cos(m\omega t) + \sum_{m=1}^{\infty} B_m \sin(m\omega t)$$

mit den Koeffizienten

$$A_m = \frac{2}{T}\int_0^T f(t)\cos(m\omega t)dt \qquad \text{und} \qquad B_m = \frac{2}{T}\int_0^T f(t)\sin(m\omega t)dt \quad .$$

Aufgrund der Punktsymmetrie der Rechteckfunktion zum Ursprung können nur Funktionen gleicher Symmetrie zur Fourierreihe beitragen (Sinusfunktionen), also müssen alle Koeffizienten $A_m$ null werden (rechnen Sie es nach!). Der Koeffizient $A_0$ ist der Mittelwert von $f(t)$, also hier $A_0 = 0$. Wir können uns damit auf die Berechnung von $B_m$ beschränken. Einsetzen von $f(t)$ liefert:

$$\begin{aligned} B_m &= +\frac{2}{T}\left(\int_0^{T/2}\sin(m\omega t)dt - \int_{T/2}^{T}\sin(m\omega t)dt\right) \\ &= -\frac{2}{T}\,\frac{1}{m\omega}\left(\Big[\cos(m\omega t)\Big]_0^{T/2} - \Big[\cos(m\omega t)\Big]_{T/2}^{T}\right) \quad , \end{aligned}$$

also mit $\omega T = 2\pi$

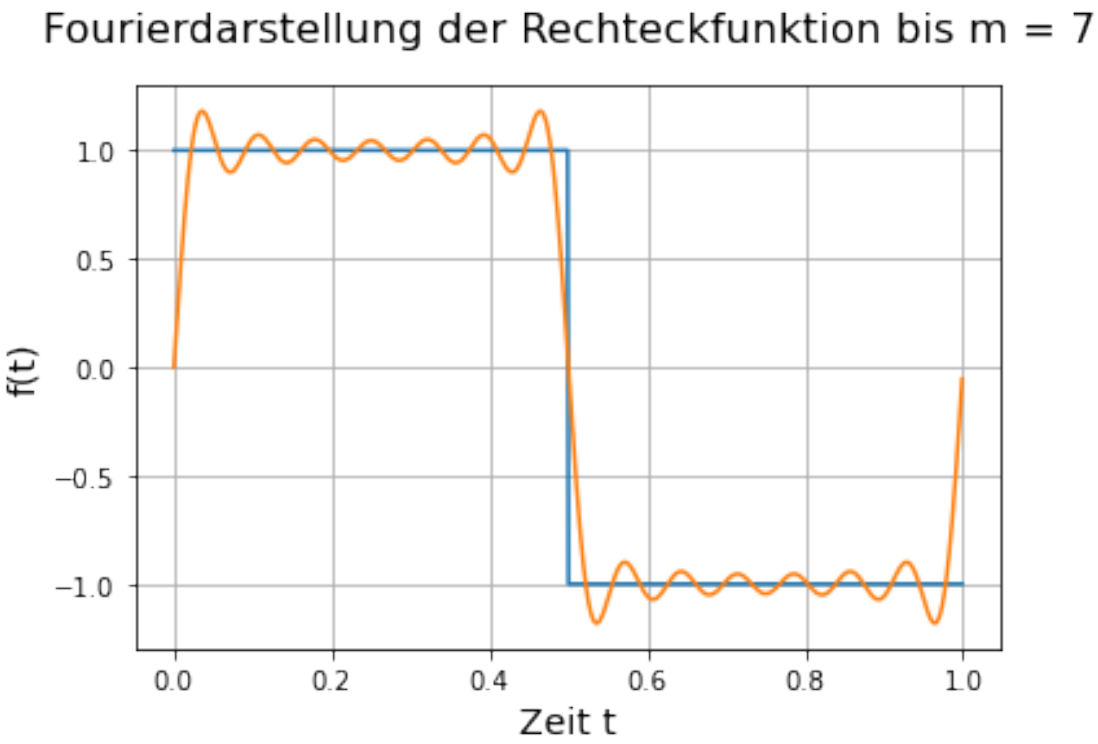
$$B_m = \begin{cases} 0 & \text{gerade } m \\ \dfrac{4}{\pi m} & \text{ungerade } m \end{cases} \quad .$$

Die Rechteckfunktion $f(t)$ kann also mittels Fourieranalyse dargestellt werden als

$$f(t) = \frac{4}{\pi}\left(\sin(\omega t) + \frac{1}{3}\sin(3\omega t) + \frac{1}{5}\sin(5\omega t) + \ldots\right) \quad .$$

Im folgenden Skript wird $f(t)$ näherungsweise mittels Fourierreihe bis zu einer maximalen Ordnung $m$ berechnet und der Rechteckfunktion zum Vergleich überlagert. Öndern Sie $m$ und beobachten Sie, wie sich das auswirkt!

[2]:
```
m = 7                           # Wähle max. Ordnung für Koeffizienten

def fRect(t, T, m):             # Funktion zur Berechnung von f(t) mittels
    y = 0                       # Fourierkoeffizienten (nur für phi=0 !)
    for i in range(1, 2 * m, 2):
        y += np.sin(i * 2 * np.pi / T * t) / i
    return 4 / np.pi * y

plt.plot(t, f)
plt.plot(t, fRect(t, T, m))
plt.xlabel("Zeit t", fontsize=14)
plt.ylabel("f(t)", fontsize=14)
plt.title(f"Fourierdarstellung der Rechteckfunktion bis m = {m}",␣
 ↪fontsize=16, y=1.05)
plt.show()
```




*Anmerkung:* Diese Darstellung ist natürlich nur korrekt, solange $f(t)$ ungeändert bleibt, da `fRect` feste Werte für $A_m$ und $B_m$ benutzt. Insbesondere führt die Önderung der Phasenverschiebung von $f(t)$ mit dem Parameter `phi` zu falschen Ergebnissen. Wir werden weiter unten zeigen, wie man das verbessern kann.

### 1.2 Numerische Berechnung der Fouriertransformation

Statt die Fourierreihe analytisch zu berechnen, kann man auch numerische Verfahren dazu verwenden. Das mit Abstand am häufigsten verwendete Verfahren ist die *Fast-Fourier-Transformation (FFT)*. Die hohe Effektivität der numerischen Berechnung beruht u.a. darauf, dass die Zahl der verwendeten Daten eine Potenz von 2 sein muss, also z.B. 16, 512, oder 524288. Wir haben oben für das Array $t$ die Anzahl zu $N = 1024$ festgelegt. Die *FFT*-Routine in Python verwendet komplexe Zahlen zur Berechnung, also die Darstellung aus der Vorlesung (für eine kontinuierliche Funktion ist $N$ unendlich groß)

$$f(t) = \sum_{m=-\infty}^{+\infty} f_m \mathrm{e}^{im\omega t} \quad .$$

Die Amplituden $f_m$ sind komplexe Zahlen $f_m = f'_m + if''_m$. Für eine *reelle Funktion* $f(t)$ gelten die Beziehungen

$$f'_{-m} = f'_m \; ; \quad f''_{-m} = -f''_m$$

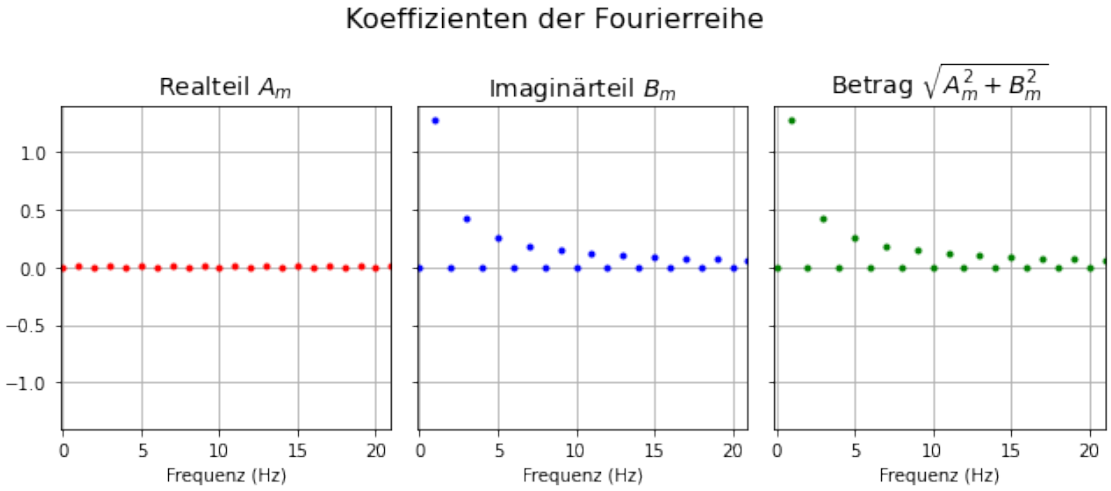
und

$$A_m = 2f'_m \; ; \quad B_m = -2f''_m \quad .$$

Da $f(t)$ für numerische Zwecke aus diskreten Werten besteht (also $f_n(t_n)$), genügt eine endliche Zahl $m \in [0 \dots N-1]$ von Sinus- und Kosinusfunktionen für die Fourierreihe. In der untenstehenden Grafik dargestellt werden der Realteil $A_m$ (rot), der Imaginärteil $B_m$ (blau) sowie der Betrag $\sqrt{A_m^2 + B_m^2}$.

[3]:
```
# FFT
yfo = fft(f, norm="forward")  # berechnet FFT mit Normierungsfaktor 1/N
                              # zum besseren Vergleich mit A_m und B_m
xfo = fftfreq(N, T / N)       # erzeugt die zugehörigen Frequenzen

# Kopiere die Arrays, da wir sie verändern werden und später die Originale␣
 ↪noch brauchen
yf = yfo.copy()
xf = xfo.copy()

# Sortiere Freq
xf = np.fft.fftshift(xf)      # sortiert die Werte so, dass die Frequenz 0
yf = np.fft.fftshift(yf)      # und ihre Amplitude im Zentrum der Arrays ist.
mf = np.abs(yf)               # Absolutewert der Amplituden

m_max = 21                    # Maximale Ordnung für die Darstellung des␣
 ↪Plots
f_max = m_max / T             # Maximale Frequenz

fig, ax = plt.subplots(1, 3, sharey=True, figsize=(9, 4))

fig.suptitle("Koeffizienten der Fourierreihe", fontsize=16)

ax[0].set_title("Realteil $A_m$", fontsize=14)
ax[0].plot(xf, 2 * yf.real, "r.")

ax[1].set_title("Imaginärteil $B_m$", fontsize=14)
ax[1].plot(xf, -2 * yf.imag, "b.")

ax[2].set_title("Betrag $\sqrt{A_m^2+B_m^2}$", fontsize=14)
ax[2].plot(xf, 2 * mf, "g.")

plt.setp(ax, xlim=[-0.2, f_max], xlabel="Frequenz (Hz)")
fig.tight_layout()
plt.show()
```

```
result = signal.argrelextrema(mf, np.greater, order=1)[0]   # Finde Maxima
print("Maxima des Betrages:")
for i in result:
    if 0 < xf[i] <= f_max:
        print(f"{xf[i]:10.3f} {2*mf[i]:10.3f}")
```




```
Maxima des Betrages:
     1.000      1.273
     3.000      0.424
     5.000      0.255
     7.000      0.182
     9.000      0.141
    11.000      0.116
    13.000      0.098
    15.000      0.085
    17.000      0.075
    19.000      0.067
    21.000      0.061
```

Da zu Anfang $N = 1024$ gesetzt wurde, haben wir gleich 1024 Koeffizienten bestimmt, von denen hier aber nur einige dargestellt sind! Die numerisch berechneten Werte stimmen mit denen der analytischen Berechnung überein.

Was passiert, wenn man die Rechteckfunktion "verschiebt", z.B. um 45°, also $\pi/4$? Probieren Sie es aus! Setzen Sie oben bei der Definition der Rechteckfunktion die Phasenverschiebung auf phi $= \pi/4$ oder auf einen beliebigen anderen Wert. Die Symmetrie der Funktion ändert sich und damit auch die Koeffizienten $A_m$ und $B_m$, nicht aber der Betrag $\sqrt{A_m^2 + B_m^2}$.

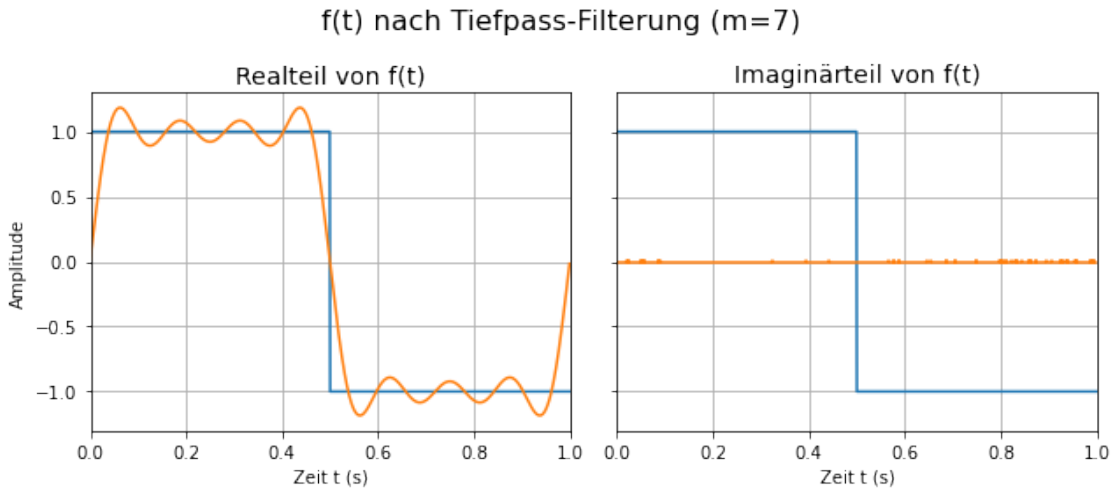
### 1.3 Inverse Fouriertransformation (Umkehrtransformation)

Wendet man auf das Resultat der Fouriertransformation die *inverse Fast-Fourier-Transformation (iFFT)* an, dann erhält man die ursprünglichen Eingangsgrößen, hier also die Funktion $f(t)$, wieder zurück. Man spricht daher auch von einer Rück- oder Umkehrtransformation. Sinnvoll ist so eine Rücktransformation natürlich nur dann, wenn dadurch die ursprünglichen Eingangsgrößen in gezielter Weise verändert, also "verbessert" werden. Eine häufige Anwendung ist es, Komponenten hoher Frequenz zu entfernen, wenn diese nur noch *Rauschen* und somit keine Informationen mehr beinhalten. So eine Manipulation der Daten nennt man *Filter*. Wir können das hier demonstrieren, indem wir die Amplituden hoher Frequenzen zu null setzen - das haben wir ja oben im Beispiel der Rekonstruktion der Rechteckfunktion mittels Sinusfunktionen bis zu einer maximalen Ordnung $m$ schonmal gemacht. Zum Vergleich werden wir das hier wiederholen, allerdings nun mit Hilfe der inversen FFT für die komplexwertigen Resultate der Hintransformation. Dazu setzen wir alle komplexen Amplituden des Arrays `yf` oberhalb der Frequenz $m/T$ und unterhalb $-m/T$ zu null.

Probieren Sie wieder aus, wie sich der Output ändert mit Önderung von $m$! Für $m > N/2$ wird die Rechteckfunktion vollständig rekonstruiert - es ist also keine Information bei Hin- und Rücktransformation verloren gegangen.

[4]:
```
# Cutoff-Frequenz bei Ordnung +- m  (m < N/2)
m = 7

# Wir verwenden wieder Kopien der Arrays yfo und xfo, da wir xf und yf
# zwischendurch verändert haben wurden (mit fftshift)
yf = yfo.copy()
xf = xfo.copy()

# Setze Amplituden mit Frequenzen xf > m/T und < -m/T zu null (Filter).
# Das geht in Python sehr elegant - der Ausdruck in der runden Klammer ist
# die Bedingung für die Zuweisung.
yf[xf > +m/T] = 0
yf[xf < -m/T] = 0

# inverse FFT
iyf = ifft(yf, norm="forward")  # berechnet iFFT ohne Normierungsfaktor
                                # da dieser schon bei der Hintransformation
                                # angewendet wurde

# Plot
fig, ax = plt.subplots(1, 2, sharey=True, figsize=(9, 4))
fig.suptitle(f"f(t) nach Tiefpass-Filterung (m={m})", fontsize=16)

ax[0].set_title("Realteil von f(t)", fontsize=14)
ax[0].set_ylabel("Amplitude")
ax[0].plot(t, f)
ax[0].plot(t, iyf.real)
```

```
ax[1].set_title("Imaginärteil von f(t)", fontsize=14)
ax[1].plot(t, f)
ax[1].plot(t, iyf.imag)

plt.setp(ax, xlabel="Zeit t (s)", xlim=[0, 1])
fig.tight_layout()
plt.show()
```




Die inverse FFT liefert dasselbe Ergebnis wie die die Berechnung mit den analytisch gewonnenen Koeffizienten, allerdings (für hohe Ordnungen) viel schneller, da der *iFFT-Algorithmus* weit effizienter ist. Im Vergleich zur obigen Berechnung liefert diese auch korrekte Ergebnisse, wenn die Phasenverschiebung `phi` von $f(t)$ in der Definition geändert wird. Der Grund ist, dass neben dem Realteil auch der Imaginärteil der Lösung verwendet wird.

Der Sinn der hier diskutierten Filterung erschließt sich leichter, wenn man als Beispiel "verrauschte" Daten verwendet. Setzten Sie dazu in der Definition von $f(t)$ den Parameter `noise` von 0 auf z.B. 0.5. Der Rechteckfunktion wird dann ein Rauschen mit Standardabweichung 0.5 überlagert. Das Rauschen wird umso stärker unterdrückt, je kleiner $m$ für den *Filter* im obigen Skript gewählt wird. Das geschieht allerdings auch auf Kosten der hochfrequenten *Informationen* in $f(t)$: die Kante wird weit weniger steil. Statt des simplen Löschens der Koeffizienten hoher Frequenzen werden in "echten" Anwendungen die Koeffizienten kontinuierlich mit der Frequenz abgeschwächt - dann sind die Überschwinger an den Kanten ("Ringing") weniger ausgeprägt.

Probieren Sie es aus! Vergessen Sie nicht, alle Zellen im Skript nochmal zu berechnen, wenn Sie in der ersten Zelle eine Önderung machen (Menü *Run / Run All Cells*).