Grundlagen der Nachrichtentechnik

Begleitmaterial zum Buch "Grundlagen der Nachrichtentechnik", Carl Hanser Verlag, 2. Aufl. 2023.

Carsten Roppel (c.roppel@hs-sm.de)

Inhalt

- Basisband-Übertragung und Empfänger mit signalangepasstem Filter
- QPSK-Übertragungsstrecke und Empfänger mit signalangepasstem Filter
- QPSK-Übertragungsstrecke mit Mehrwegekanal und Empfänger mit T/2-Entzerrer

Wir beschreiben die Simulation verschiedener Übertragungsstrecken mit Python. In den vorliegenden Modellen wird die Umsetzung der folgenden Verfahren bzw. Funktionen behandelt:

- Basisbandübertragung: Bipolares NRZ-Signal
- Modulation: QPSK in äquivalenter Basisband-Darstellung
- Wurzel-Kosinus-roll-off-Filter als Pulsformfilter und als signalangepasstes Filter
- Augendiagramm
- Leistungsdichtespektrum
- AWGN-Kanal
- Ermittlung der Bit- und Symbolfehlerhäufigkeit
- Signalraumplot
- Mehrwegekanal in äquivalenter Basisband-Darstellung
- T/2-Entzerrer

Der Python-Code steht in Form eines Jupyter-Notebooks auf den Seiten

https://plus.hanser-fachbuch.de/ und

https://www.hs-schmalkalden.de/nachrichtentechnik zur Verfügung.

1 Basisband-Übertragung und Empfänger mit signalangepasstem Filter

Wir bauen schrittweise die Simulation einer Basisband-Übertragungsstrecke auf. Es handelt sich um eine bipolare NRZ-Codierung. Als Pulsformfilter und dazu passendes signalangepasstes Filter dient ein Wurzel-Kosinus-roll-off-Filter. Nach dem Einfügen eines AWGN-Kanals wird die Bitfehlerhäufigkeit bestimmt und mit dem theoretischen Ergebnis verglichen.

Versuch 1.1: Wurzel-Kosinus-roll-off-Filter

Wir entwerfen ein Wurzel-Kosinus-roll-off-Filter mit einer Impulsantwort gemäß Gl. (5.41) und einem Roll-off-Faktor $\alpha = 0{,}5$. Das Filter wird wie in Abschnitt 5.2.3 beschrieben als FIR-Filter realisiert. Die Impulsantwort wird auf den Bereich $\pm 3\,T_s$ beschränkt und hat 8 Abtastwerte pro Symbol.

Mithilfe von Gl. (4.5) wird die Energie der Impulsantwort bestimmt. Die Impulsantwort wird so skaliert, dass die Energie gleich 1 ist. Plots (ähnlich Bild 5.12 und Bild 5.27) veranschaulichen die Impulsantwort.

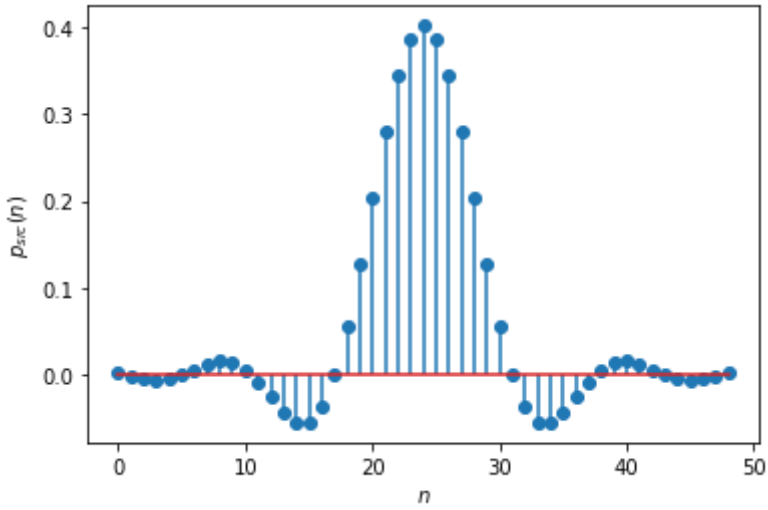
Die Impulsantwort wird entsprechend Gl. (5.41) definiert. Bei $t/T_s = 0$ und bei $t/T_s = 1/(4\,\alpha)$ wird der Nenner und auch der Zähler null. Für diese Werte muss der Funktionswert separat angegeben werden.

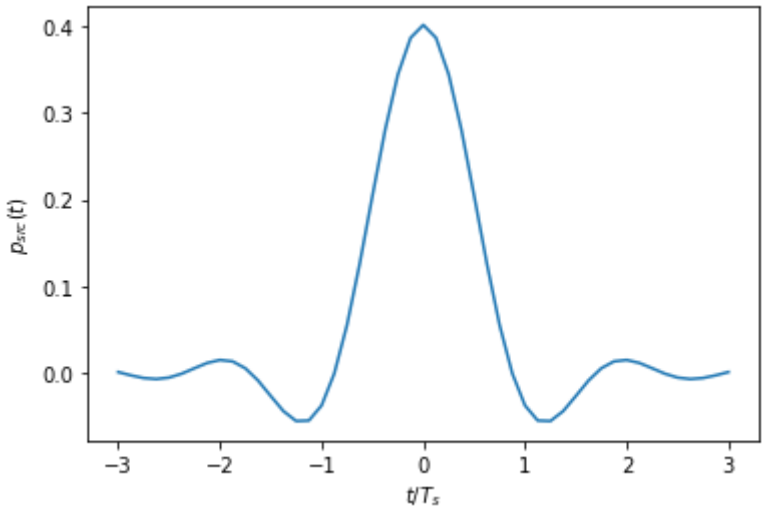
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as scs
from scipy.special import erfc
```

In [2]:
```
# Parameter
alpha = 0.5  # Roll-off-Faktor
os = 8       # Samples pro Symbol (Oversampling Ratio)
dly_rc = 3   # Verzögerung des Wurzel-Kosinus-roll-off-Filters

# Wurzel-Kosinus-roll-off-Filter
t = np.arange(-dly_rc*os, dly_rc*os + 1)/os
psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1 -
# Grenzwert fuer t/T_s = 0
psrc[dly_rc*os] = (4*alpha + np.pi*(1 - alpha))/np.pi
# Grenzwert fuer t/T_s = 1/(4 alpha))
if round(os/(4*alpha)) == os/(4*alpha): # Ist os/(4*alpha) ganzzahlig?
    psrc[int(dly_rc*os + os/(4*alpha))] = (4*alpha*np.cos(np.pi*(1 + alpha)/(4*alpl
                                           - np.pi*(1 + alpha)*np.sin(np.pi*(1 + a
                                           + np.pi*(1 - alpha)*np.cos(np.pi*(1 - a
    psrc[int(dly_rc*os - os/(4*alpha))] = psrc[int(dly_rc*os + os/(4*alpha))]

E = sum(psrc**2)
psrc = psrc/np.sqrt(E)

# Plots
plt.figure()
plt.stem(psrc)
plt.xlabel('$n$')
plt.ylabel('$p_{src}(n)$')
```

```
plt.show()

plt.figure()
plt.plot(t, psrc)
plt.xlabel('$t / T_s$')
plt.ylabel('$p_{src}(t)$')
plt.show()
```

```
C:\Users\roppel\AppData\Local\Temp\ipykernel_2708\3790446892.py:8: RuntimeWarning:
divide by zero encountered in true_divide
  psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1
- (4*alpha*t)**2)*np.pi*t)
C:\Users\roppel\AppData\Local\Temp\ipykernel_2708\3790446892.py:8: RuntimeWarning:
invalid value encountered in true_divide
  psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1
- (4*alpha*t)**2)*np.pi*t)
```





Versuch 1.2: Basisbandsignal

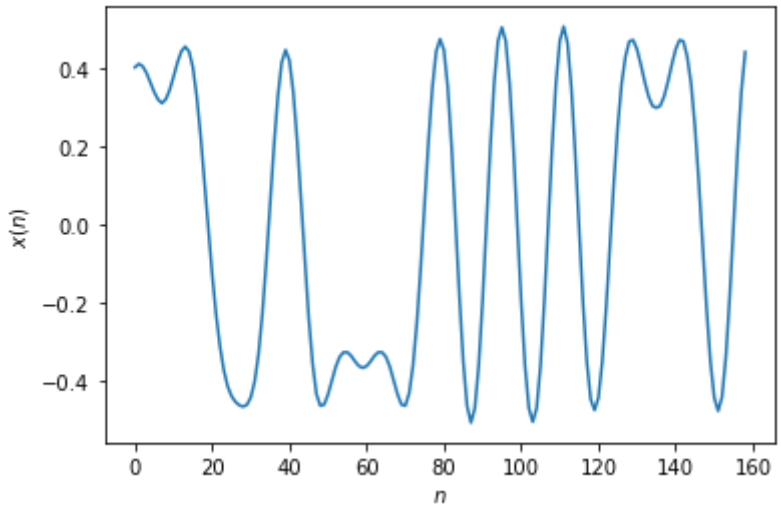
Wir erzeugen 1000 Zufallszahlen $\in \{0, 1\}$, die in binäre bipolare Symbole $\in \{-1, 1\}$ umgewandelt werden. Anschließend wird die Abtastrate auf 8 Abtastwerte pro Symbol erhöht, indem 7 Nullen zwischen den Symbolen eingefügt werden. Dieses Signal wird mit der Impulsantwort des Wurzel-Kosinus-roll-off-Filters gefaltet, und ein Plot der ersten 20 Symbole erstellt. Das Augendiagramm wird erzeugt und das Leistungsdichtespektrum in Form des Periodogramms wird berechnet und geplottet.

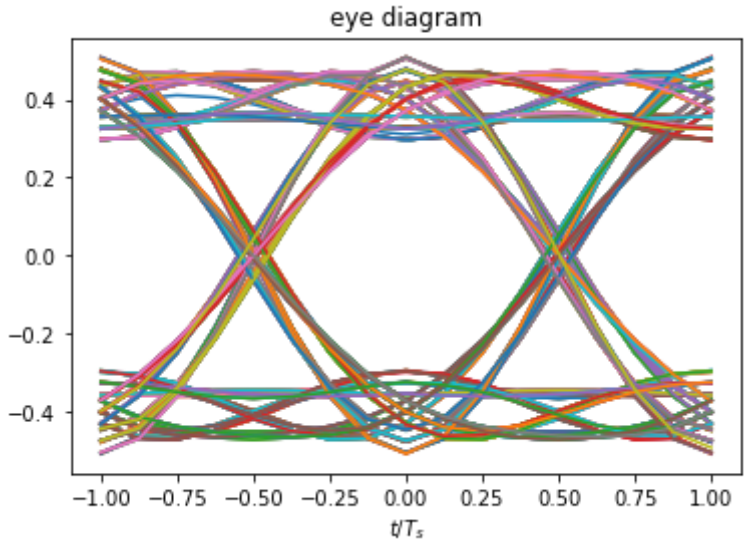
Es ist günstig, den Zufallsgenerator zu initialisieren, so dass immer die gleichen Zufallszahlen erzeugt werden und somit die Ergebnisse reproduzierbar bleiben. Da das Wurzel-Kosinus-roll-off-Filter kein Nyquist-Filter ist, schneiden sich im Augendiagramm die Linien an der Stelle der größten Augenöffnung {\it nicht} in einem Punkt.

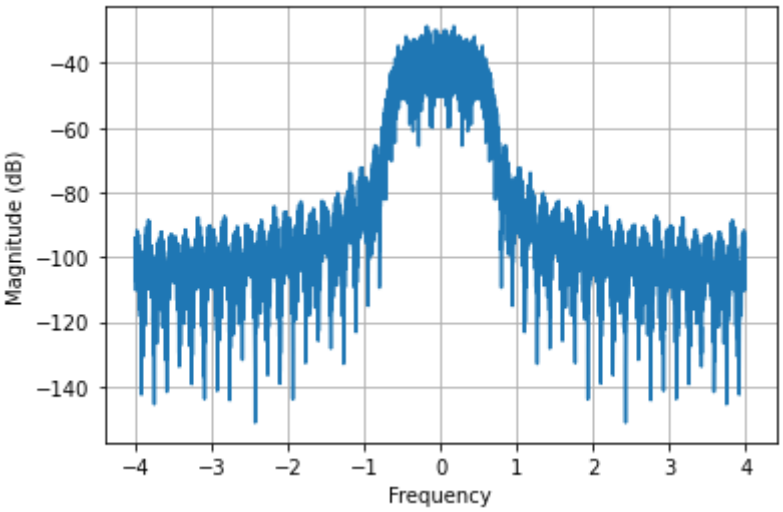
Das Leistungsdichtespektrum wird mit der Matplotlib-Funktion magnitude_spectrum() berechnet. Da wir von einer Symboldauer $T_s = 1$ (Sekunde) und 8 Abtastwerten pro Symbol ausgehen, beträgt die Abtastrate $f_s = 8$\:Hz und das Spektrum erstreckt sich von $-f_s/2 \leq f \leq f_s/2$ oder -4 Hz $\leq f \leq 4$ Hz.

Ferner ist die Symbolrate $r_s = 1$ (Symbole pro Sekunde) und für $\alpha = 0{,}5$ beträgt die Übertragungsbandbreite nach Gl. (5.7) B_K = 0,75 Hz. Das Leistungsdichtespektrum ist deutlich auf diese Bandbreite begrenzt. Die Signalanteile oberhalb von B_K gehen auf die numerische Berechnung des Spektrums mittels der diskreten Fourier-Transformation zurück.

In [3]:
```
nSym = 1000
rng = np.random.default_rng(1234)  # Initialisiere Zufallsgenerator
bits = rng.integers(low = 0, high = 2, size = nSym)  # Array der Länge nSym mit Zu
symbols = 2*bits - 1  # Binäe bipolare Symbolfolge

symbols = scs.upfirdn([1], symbols, os)
x = scs.convolve(psrc, symbols, method = 'direct')
x = x[dly_rc*os:-1 - dly_rc*os]

plt.figure()
plt.plot(x[1:20*os])
plt.xlabel('$n$')
plt.ylabel('$x(n)$')
plt.show()

teye = np.arange(-os, os + 1)/os
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, x[i1:i2])

plt.title('eye diagram')
plt.xlabel('$t / T_s$')
plt.show()

plt.magnitude_spectrum(x, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()
```





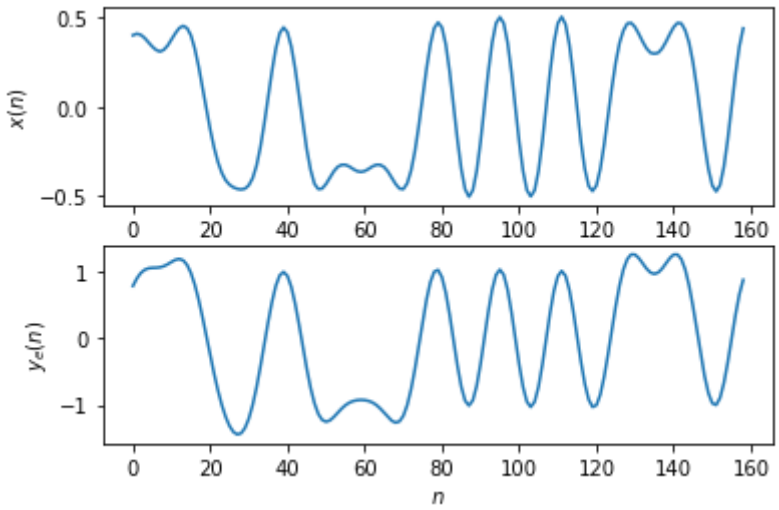


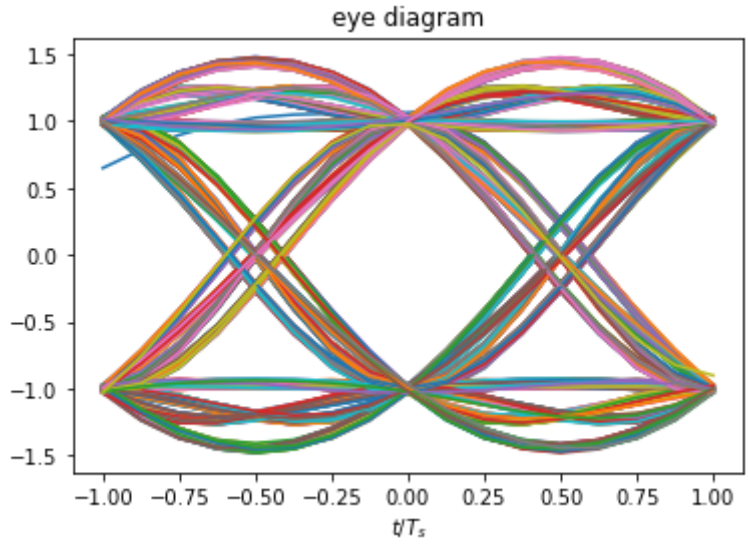
Versuch 1.3: Signalangepasstes Filter

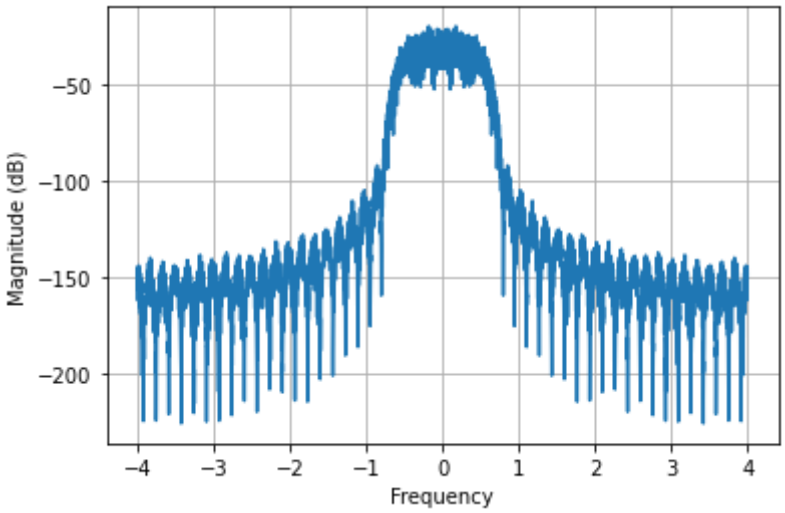
Da beim Wurzel-Kosinus-roll-off-Filter das signalangepasste Filter identisch zum Pulsformfilter ist, genügt es, das Basisbandsignal mit der Impulsantwort des Wurzel-Kosinus-roll-off-Filters zu falten. Das Signal wird vor und nach dem Empfangsfilter für die ersten 20 Symbole geplottet. Nach dem Empfangsfilter wird das Augendiagramm erstellt und das Leistungsdichtespektrum in Form des Periodogramms berechnet und geplottet.

Am Ausgang des Empfangsfilters entstehen Nyquist-Impulse, und im Augendiagramm schneiden sich die Linien an der Stelle der größten Augenöffnung alle in einem Punkt. Bei genauerem Hinsehen erkennt man allerdings, dass die Linien nicht exakt durch den gleichen Punkt verlaufen. Dies hängt damit zusammen, dass wir die Wurzel-Kosinus-roll-off-Filter nur näherungsweise implementiert haben, da die Impulsantwort der Filter zeitlich begrenzt ist. Die Näherung wird besser, wenn wir die Impulsantwort verlängeren (z. B. mit dly_rc = 5). Das Leistungsdichtespektrum hat die Form entsprechend Beispiel 5.1.

In [4]:
```
ye = scs.convolve(psrc, x, method = 'direct')
ye = ye[dly_rc*os:-1 - dly_rc*os]

plt.figure()
plt.subplot(2, 1, 1)
plt.plot(x[1:20*os])
plt.ylabel('$x(n)$')
plt.subplot(2, 1, 2)
plt.plot(ye[1:20*os])
plt.xlabel('$n$')
plt.ylabel('$y_e(n)$')
plt.show()

teye = np.arange(-os, os + 1)/os
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, ye[i1:i2])

plt.title('eye diagram')
plt.xlabel('$t / T_s$')
plt.show()

plt.magnitude_spectrum(ye, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()
```







Versuch 1.4: AWGN-Kanal

Wir fügen vor dem signalangepassten Filter einen AWGN-Kanal mit $E_b/N_0 = 6$ dB ein. Das entsprechende Signal-Rausch-Verhältnis im Kanal wird gemäß Gl.(5.47) bestimmt. Wir müssen noch beachten, dass sich in dieser Gleichung die Rauschleistung auf die Bandbreite des Nutzsignals bezieht. Die Rauschleistung ist das Produkt Rauschleistungsdichte mal Bandbreite. Da wir in unserer Simulation die Bandbreite aufgrund der Überabtastung mit os = 8 Abtastwerten pro Symbol um den Faktor 8 erhöht haben, muss die Rauschleistung um den Faktor 8 größer und der Störabstand um $10 \lg 8$ dB kleiner sein.

Wir bestimmen die Leistung $\overline{x^2(n)}$ des Basisbandsignals. Mithilfe des zuvor ermittelten Signal-Rausch-Verhältnises ergibt sich die erforderliche Rauschleistung N. Als Rauschsignal werden mittelwertfreie normal verteilte Zufallszahlen der Leistung N erzeugt. Diese werden zum Basisbandsignal addiert. Das Augendiagramm und das Leistungsdichtespektrum wird erzeugt. Aufgrund des Rauschens ist im Augendiagramm keine Öffnung mehr zu erkennen, und im Spektrum ragt das Nutzsignal nur noch wenig über das Rauschen hinaus.

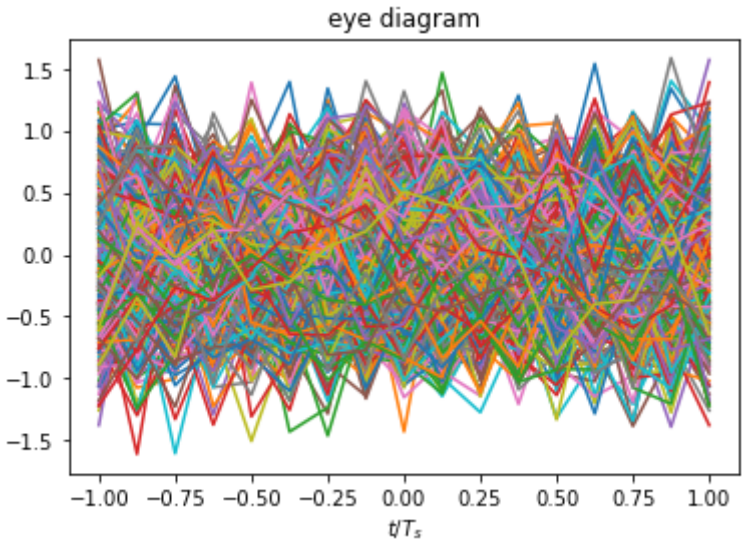
Wir fügen wieder das signalangepasste Filter ein und erzeugen das Augendiagramm. Nach dem Empfangsfilter ist wieder eine Augenöffnung zu erkennen. Die Abtastrate wird auf 1 Abtastwert pro Symbol herabgesetzt. Dazu wird nur jeder achte Wert des Signals nach dem Empfangsfilter beibehalten. Dies müssen die Werten bei der größten Augenöffnung sein. Es

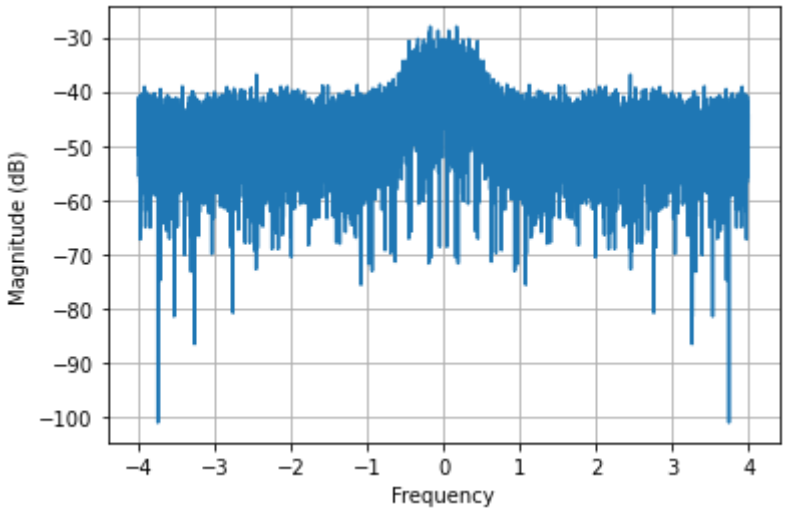
folgt der Entscheider, der den Wert 0 oder 1 liefern soll, je nachdem ob der Abtastwert größer oder kleiner 0 ist.

Die Anzahl der Bitfehler sowie die Bitfehlerhäufigkeit wird durch Vergleich mit der zuvor erzeugten Bitfolge bestimmt. Zum Vergleich wird die theoretische Bitfehlerwahrscheinlichkeit für $E_b/N_0 = 6$ dB mithilfe von Gl. (5.39) berechnet.

Der Entscheider wird mithilfe der Signum-Funktion ysym = sign(ysym) realisiert, es ist ysym $\in \{-1, 0, 1\}$. Durch die nachfolgende Anweisung ybit = round((ysym + 1)/2) wird 1 auf 1, -1 auf 0 und 0 auf 1 abgebildet. Abschließend werden die Anzahl der Bitfehler und die Bitfehlerhäufigkeit ermittelt.

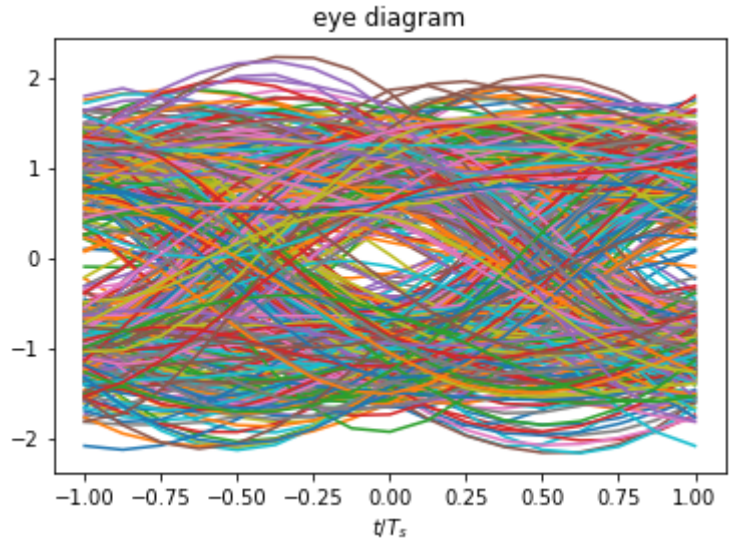
In [5]:
```
EbN0 = 6
snr = EbN0 + 10*np.log10(2) - 10*np.log10(os)
sigPwr = np.mean(x**2)
noisePwr = sigPwr/(10**(snr/10))
rng = np.random.default_rng(1234)
noise = rng.standard_normal(len(x))*np.sqrt(noisePwr)
y = x + noise

teye = np.arange(-os, os + 1)/os
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, y[i1:i2])

plt.title('eye diagram')
plt.xlabel('$t / T_s$')
plt.show()

plt.magnitude_spectrum(y, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()
```

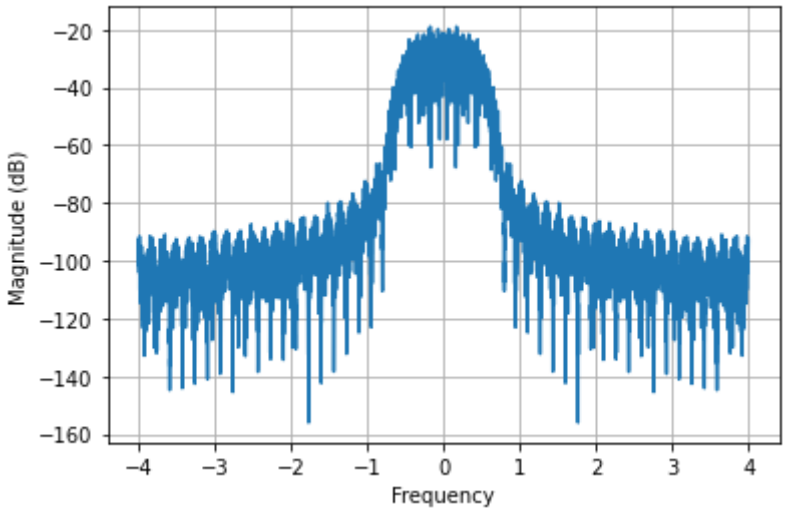




In [6]:

```
ye = scs.convolve(psrc, y, method = 'direct')
ye = ye[dly_rc*os:-1 - dly_rc*os]

teye = np.arange(-os, os + 1)/os
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, ye[i1:i2])

plt.title('eye diagram')
plt.xlabel('$t / T_s$')
plt.show()

plt.magnitude_spectrum(ye, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()
```





In [7]:
```
ysym = scs.decimate(ye, os, n = 0, ftype = 'fir')
ysym = np.sign(ysym)
ybit = (ysym + 1)/2
nerror = sum(abs(ybit - bits[0:len(ybit)]))
ber = nerror/len(ybit)
berTheo = 0.5*erfc(np.sqrt(10**(EbN0/10)))
print("Eb/N0:", EbN0, "dB", "    Pb (Theo): %1.4e" % berTheo, "    Bitfehler: %4i"
```

```
Eb/N0: 6 dB     Pb (Theo): 2.3883e-03     Bitfehler:    2     Bitfehlerhäufigkeit:
2.0020e-03
```

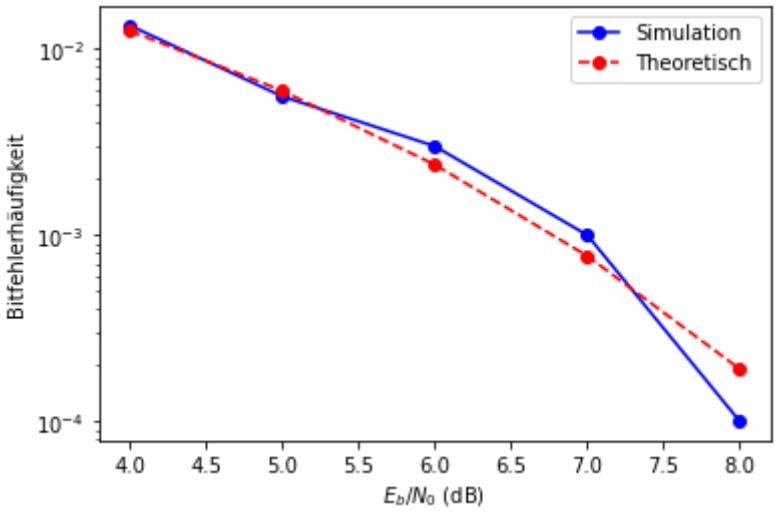
Versuch 1.5: Bitfehlerhäufigkeit

Wir wollen abschließend die Bitfehlerhäufigkeit für $E_b/N_0 = 4,\ 5,\ \ldots,\ 8$ dB bestimmen. Dazu legen wir einen entsprechenden Vektor für E_b/N_0 an. Die Anzahl der Symbole beträgt 20000. Innerhalb einer for-Schleife werden die Anweisungen für den AWGN-Kanal, den Empfänger mit signalangepasstem Filter (ohne das Augendiagramm) und die Bestimmung der Bitfehlerhäufigkeit ausgeführt. Die Schleife wird für jeden der E_b/N_0-Werte einmal durchlaufen. Die Werte für die Bitfehlerhäufigkeit und die theoretische Bitfehlerwahrscheinlichkeit werden jeweils in einem Vektor abgespeichert. Nachdem die for-Schleife durchlaufen wurde, wird die ermittelte Bitfehlerhäufigkeit und die theoretische Bitfehlerwahrscheinlichkeit in einem halblogarithmischen Plot über E_b/N_0 in dB geplottet.

In [8]:
```
nSym = int(2e4)
EbN0List = np.arange(4, 9)
berTheoList = np.zeros(len(EbN0List))
berList = np.zeros(len(EbN0List))

rng = np.random.default_rng(1234)  # Initialisiere Zufallsgenerator
bits = rng.integers(low = 0, high = 2, size = nSym)  # Array der Länge nSym mit Zu
symbols = 2*bits - 1  # Binäe bipolare Symbolfolge
symbols = scs.upfirdn([1], symbols, os)
x = scs.convolve(psrc, symbols, method = 'direct')
x = x[dly_rc*os:-1 - dly_rc*os]
sigPwr = np.mean(x**2)

for i in range(0, len(EbN0List)):
    EbN0 = EbN0List[i]
    snr = 10*np.log10(2) + EbN0 - 10*np.log10(os)
    noisePwr = sigPwr/(10**(snr/10))
    noise = rng.standard_normal(len(x))*np.sqrt(noisePwr)
    y = x + noise
```

```
    ye = scs.convolve(psrc, y, method = 'direct')
    ye = ye[dly_rc*os:-1 - dly_rc*os]
    ysym = scs.decimate(ye, os, n = 0, ftype = 'fir')
    ysym = np.sign(ysym)
    ybit = (ysym + 1)/2
    nerror = sum(abs(ybit - bits[0:len(ybit)]))
    ber = nerror/len(ybit)
    berList[i] = ber
    berTheo = 0.5*erfc(np.sqrt(10**(EbN0/10)))
    berTheoList[i] = berTheo
    print("Eb/N0:", EbN0, "dB", "    Pb (Theo): %1.4e" % berTheo, "    Bitfehler: %

plt.figure()
plt.semilogy(EbN0List, berList, 'bo-', label = 'Simulation')
plt.semilogy(EbN0List, berTheoList, 'ro--', label = 'Theoretisch')
plt.grid()
plt.xlabel('$E_b/N_0$ (dB)')
plt.ylabel('Bitfehlerhäufigkeit')
plt.legend()
plt.show()
```

```
Eb/N0: 4 dB     Pb (Theo): 1.2501e-02     Bitfehler:  267     Bitfehlerhäufigkeit:
1.3351e-02
Eb/N0: 5 dB     Pb (Theo): 5.9539e-03     Bitfehler:  111     Bitfehlerhäufigkeit:
5.5503e-03
Eb/N0: 6 dB     Pb (Theo): 2.3883e-03     Bitfehler:   60     Bitfehlerhäufigkeit:
3.0002e-03
Eb/N0: 7 dB     Pb (Theo): 7.7267e-04     Bitfehler:   20     Bitfehlerhäufigkeit:
1.0001e-03
Eb/N0: 8 dB     Pb (Theo): 1.9091e-04     Bitfehler:    2     Bitfehlerhäufigkeit:
1.0001e-04
```



2 QPSK-Übertragungsstrecke und Empfänger mit siganlangepasstem Filter

Wir verwenden QPSK (Quadrature Phase-Shift Keying) als Modulationsverfahren. Zur Pulsformung dient wieder ein Wurzel-Kosinus-roll-off-Filter, und im Empfänger kommt das passende signalangepasste Filter zum Einsatz. Die Signale werden mit Hilfe des Augendiagramms, des Konstellationsdiagramms und des Leistungsdichtespektrums untersucht. Die Symbolfehlerhäufigkeit wird ermittelt und mit dem theoretischen Ergebniss verglichen.

Das System wird im Basisband simuliert, d. h. es werden die zum Bandpasssignal äquivalenten Tiefpasssignale $x_i(t)$ und $x_q(t)$ erzeugt und verarbeitet. Dadurch werden die bei einer Simulation im Bandpassbereich auftretenden hohen Frequenzen und Abtastraten vermieden. Andererseits können alle Einflüsse auf das Bandpasssignal (Kanalübertragungsfunktion, Störungen) durch äquivalente Tiefpassmodelle in die Simulation einbezogen werden.

Versuch 2.1: QPSK-Modulator und Demodulator

Wir erzeugen als Symbolfolge 1000 Zufallszahlen $\in \{0, \ldots, m-1\}$ mit $m = 4$. Für $\lambda = \pi/4$ werden den Symbolen gemäß Gl. (6.55) bei der QPSK-Modulation folgende Phasenwinkel zugeordnet:

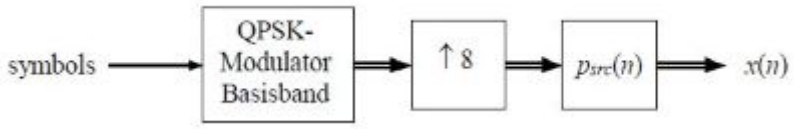
$a_k = 0 : \varphi_k = \pi/4$

$a_k = 1 : \varphi_k = 3\pi/4$

$a_k = 2 : \varphi_k = 5\pi/4$

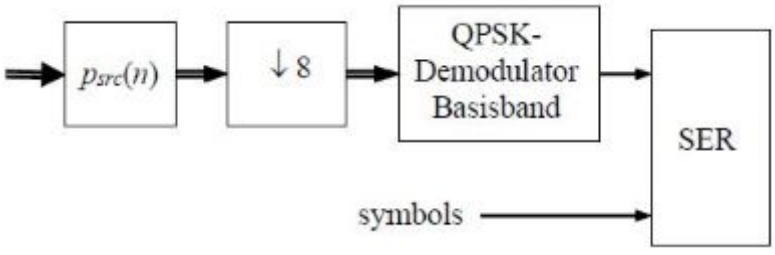
$a_k = 3 : \varphi_k = 7\pi/4$

Ein Basisbandmodell eines QPSK-Modulators erzeugt die komplexen Werte $I_k + j\,Q_k = \cos\varphi_k + j\,\sin\varphi_k = \exp(j\,\varphi_k)$. Diese entsprechen den vier Punkten im Signalraum. Anschließend wird wie in Versuch 1.2 die Abtastrate auf 8 Abtastwerte pro Symbol erhöht und das Signal mit der Impulsantwort des Wurzel-Kosinus-roll-off-Filters gefaltet. Es ergibt sich folgendes Simulationsmodell zur Erzeugung des QPSK-Signals:



Nun kann das Augendiagramm und das Leistungsdichtespektrum erzeugt werden. Da das Basisbandsignal jetzt ein komplexes Signal ist, werden zwei Augendiagramme jeweils für den Realteil (die I-Komponente) und den Imaginärteil (die Q-Komponente) erstellt.

Im Empfänger wird das Signal zuerst mit der Impulsantwort des signalangepassten Filters gefaltet und das Augendiagramm erzeugt. Anschließend wird die Abtastrate auf 1 Abtastwert pro Symbol herabgesetzt (die Werte bei der größten Augenöffnung werden beibehalten) und ein Signalraumplot (Scatterplot) erzeugt. Es folgt noch der Basisband-Demodulator, und wir erhalten folgendes Simulationsmodell zur Demodulation des QPSK-Signals:



Im Signalraumplot sind die vier Punkte der QPSK-Signalraumkonstellation zu erkennen wie in Bild 6.33 mitte, mit einer zusätzlichen Drehung um $\lambda = \pi/4$. Die Punkte liegen nicht exakt übereinander, da wir die Wurzel-Kosinus-roll-off-Filter nur näherungsweise implementiert haben (vgl. die Bemerkung zu Versuch 1.3).

Für die Demodulation wird zunächst die Drehung um λ durch Multiplikation mit $\exp(-j\lambda)$ rückgängig gemacht. Anschließend wird die Phase mithilfe der arctan-Funktion bestimmt, diese liegt im Bereich $-\pi$ bis π. Durch Multiplikation mit $m/(2\,\pi)$ und Runden auf ganze Zahlen erhält man Werte $\in \{-2, -1, 0, 1, 2\}$. Zu den negativen Werten wird $m = 4$ addiert und man erhält die demodulierten Symbole $\in \{0, 1, 2, 3\}$. Dieses Vorgehen entspricht Entscheidungsgrenzen, die gleich den Winkelhalbierenden zwischen den Punkten im Signalraum sind. Es kann auch für $m > 4$ angewendet werden. Für $m = 8$ erhält man beispielsweise 8-PSK mit einer Signalraumkonstellation und Entscheidungsgrenzen entsprechend Bild 6.55 a.

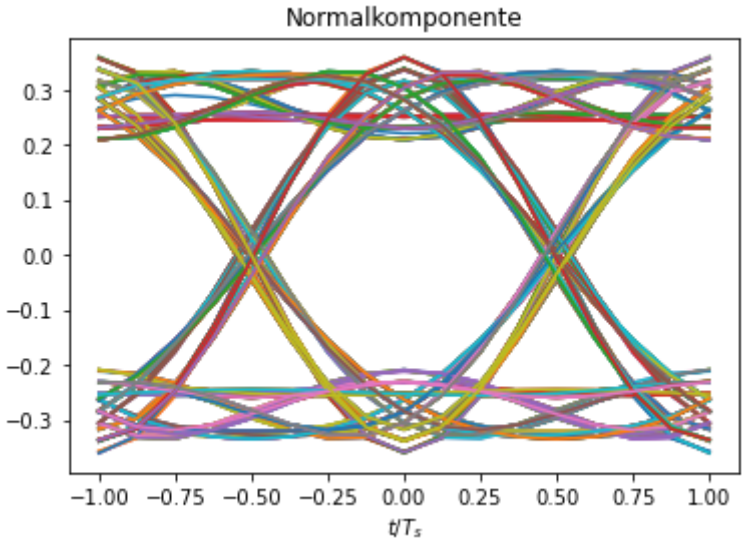
Abschließend vergleichen wir die sendeseitig erzeugte Symbolfolge mit der demodulierten Symbolfolge. Hier kann es zu einer Verschiebung der Folgen relativ zueinander kommen, die für die Ermittlung der Symbolfehler kompensiert werden muss.

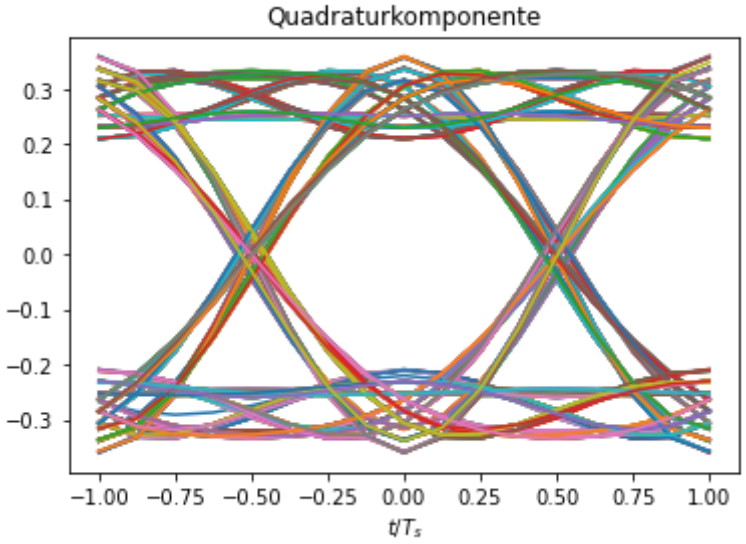
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as scs
from scipy.special import erfc
```

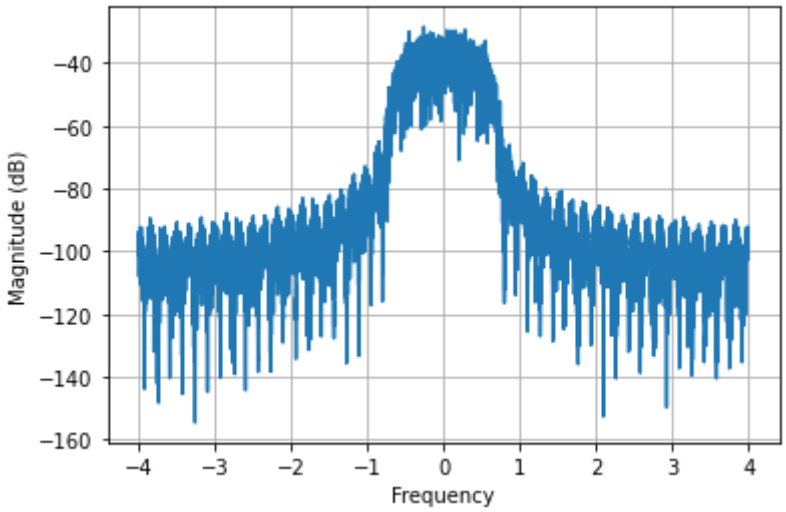
In [2]:
```
# Parameter
alpha = 0.5  # Roll-off-Faktor
os = 8       # Samples pro Symbol (Oversampling Ratio)
dly_rc = 3   # Verzögerung des Wurzel-Kosinus-roll-off-Filters
nSym = 1000
m = 4
k = np.log2(m)
lambda_psk = np.pi/4

# Wurzel-Kosinus-roll-off-Filter
t = np.arange(-dly_rc*os, dly_rc*os + 1)/os
psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1 -
# Grenzwert fuer t/T_s = 0
psrc[dly_rc*os] = (4*alpha + np.pi*(1 - alpha))/np.pi
# Grenzwert fuer t/T_s = 1/(4 alpha))
if round(os/(4*alpha)) == os/(4*alpha): # Ist os/(4*alpha) ganzzahlig?
    psrc[int(dly_rc*os + os/(4*alpha))] = (4*alpha*np.cos(np.pi*(1 + alpha)/(4*alpl
                                           - np.pi*(1 + alpha)*np.sin(np.pi*(1 + a
                                           + np.pi*(1 - alpha)*np.cos(np.pi*(1 - a
    psrc[int(dly_rc*os - os/(4*alpha))] = psrc[int(dly_rc*os + os/(4*alpha))]

E = sum(psrc**2)
psrc = psrc/np.sqrt(E)

rng = np.random.default_rng(1234)  # Initialisiere Zufallsgenerator
symbols = rng.integers(low = 0, high = m, size = nSym)  # Array der Länge nSym mit
xmod = np.exp(1j*((2*np.pi/m)*symbols + lambda_psk))
xmod = scs.upfirdn([1], xmod, os)
x = scs.convolve(psrc, xmod, method = 'direct')
x = x[dly_rc*os:-1 - dly_rc*os]

teye = np.arange(-os, os + 1)/os
xi = np.real(x)
```

```
xq = np.imag(x)
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, xi[i1:i2])
plt.title('Normalkomponente')
plt.xlabel('$t / T_s$')
plt.show()
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, xq[i1:i2])
plt.title('Quadraturkomponente')
plt.xlabel('$t / T_s$')
plt.show()

plt.magnitude_spectrum(x, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()
```

```
C:\Users\roppel\AppData\Local\Temp\ipykernel_17624\791639686.py:12: RuntimeWarnin
g: divide by zero encountered in true_divide
  psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1
- (4*alpha*t)**2)*np.pi*t)
C:\Users\roppel\AppData\Local\Temp\ipykernel_17624\791639686.py:12: RuntimeWarnin
g: invalid value encountered in true_divide
  psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1
- (4*alpha*t)**2)*np.pi*t)
```

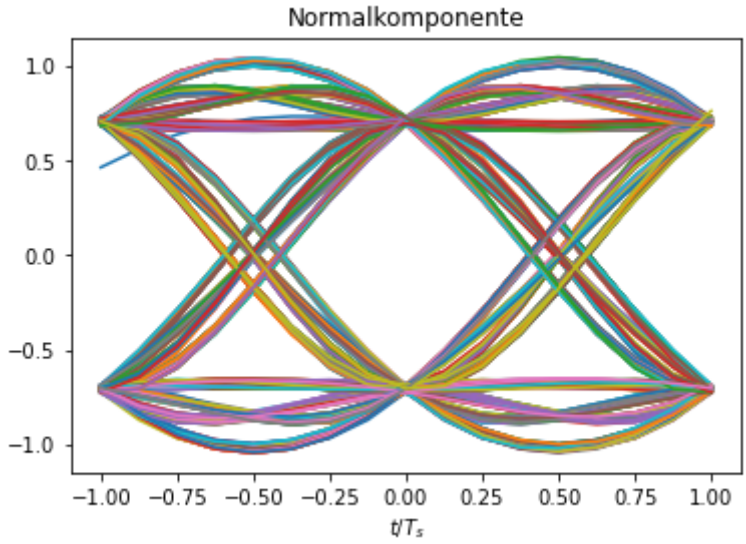


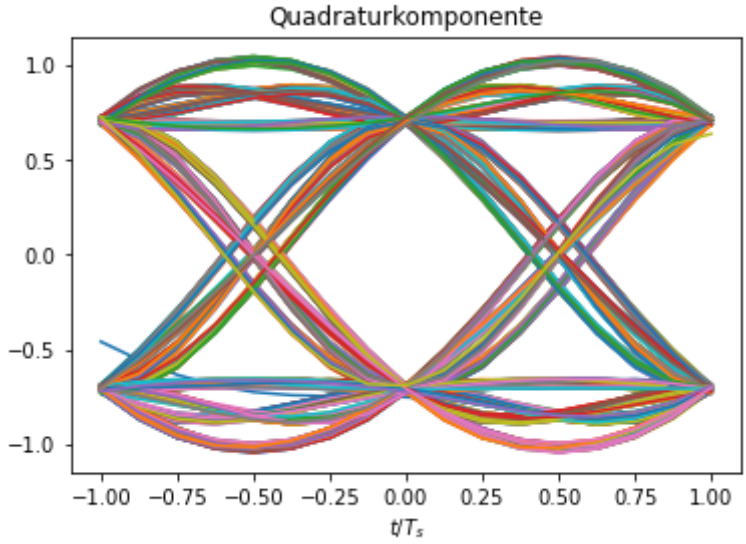


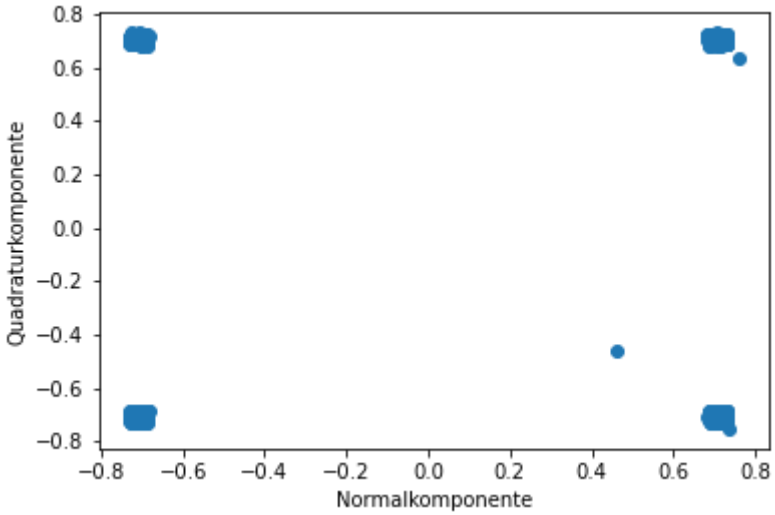


In [3]:

```
ye = scs.convolve(psrc, x, method = 'direct')
ye = ye[dly_rc*os:-1 - dly_rc*os]

teye = np.arange(-os, os + 1)/os
yi = np.real(ye)
yq = np.imag(ye)
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, yi[i1:i2])
plt.title('Normalkomponente')
plt.xlabel('$t / T_s$')
plt.show()
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, yq[i1:i2])
plt.title('Quadraturkomponente')
plt.xlabel('$t / T_s$')
plt.show()

ye = scs.decimate(ye, os, n = 0, ftype = 'fir')
plt.scatter(np.real(ye), np.imag(ye))
plt.xlabel('Normalkomponente')
plt.ylabel('Quadraturkomponente')
plt.show()

ye = ye*np.exp(-1j*lambda_psk)
ysym = np.rint(m/(2*np.pi)*np.angle(ye))
ysym[ysym < 0] = ysym[ysym < 0] + m
for i in range(10):
    print('xsym = %i' % symbols[i], '    ysym = %i' % ysym[i])
```







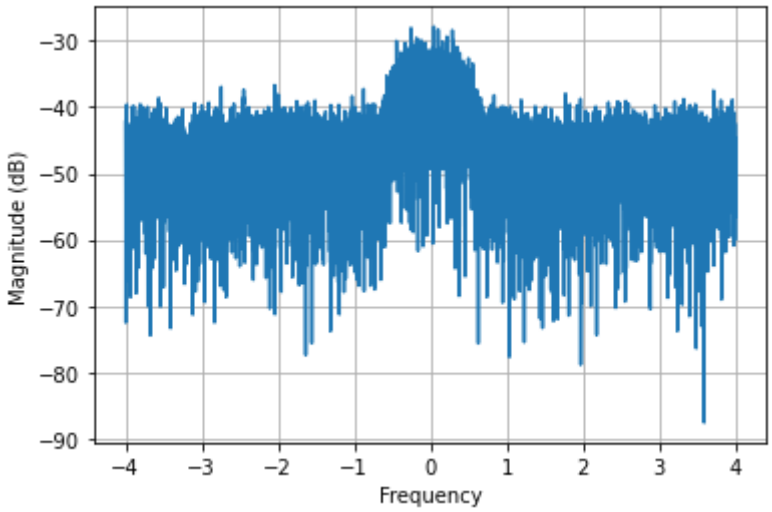

```
xsym = 3     ysym = 3
xsym = 3     ysym = 3
xsym = 3     ysym = 3
xsym = 1     ysym = 1
xsym = 0     ysym = 0
xsym = 3     ysym = 3
xsym = 0     ysym = 0
xsym = 1     ysym = 1
xsym = 0     ysym = 0
xsym = 1     ysym = 1
```

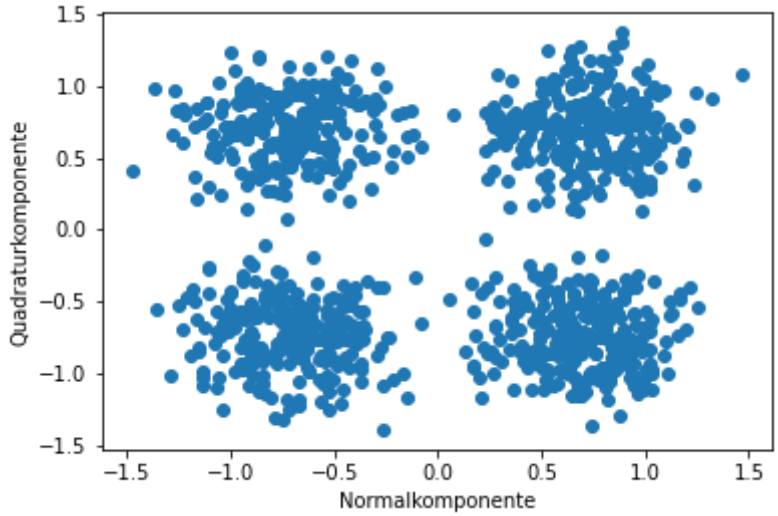
Versuch 2.2: Versuch 2.2: AWGN-Kanal für komplexes Basisbandsignal

Analog zu Versuch 1.4 ergänzen wir unser Simulationsmodell um einen AWGN-Kanal mit $E_b/N_0 = 6$ dB. Für den Zusammenhang zwischen S/N und E_b/N_0 gilt nun Gl. (6.99). Zur Bestimmung der Signalleistung bilden wir den Mittelwert über das Betragsquadrat der komplexen Werte. Da jede der Quadraturkomponenten die gleiche Leistung wie das Bandpasssignal selbst hat, müssen wir noch mit 1/2 multiplizieren. Wie in Versuch 1.4 erzeugen wir normal verteilte Zufallszahlen mit der erforderlichen Rauschleistung. Dies muss hier getrennt für die I- und die Q-Komponente mit unabhängig voneinander erzeugten Zufallszahlen erfolgen.

Nach dem signalangepassten Filter und der Demodulation bestimmen wir die Symbolfehler und die Symbolfehlerhäufigkeit, und es folgt der Vergleich mit dem theoretischen Wert $P_{s,QPSK} \approx 2\, P_{b,QPSK}$ mit der Bitfehlerwahrscheinlichkeit $P_{b,QPSK}$ nach Gl. (6.89).

In [4]:
```
EbN0 = 6
snr = EbN0 + 10*np.log10(k) - 10*np.log10(os)
sigPwr = np.mean(abs(x)**2)
noisePwr = 0.5*sigPwr/(10**(snr/10))
rng = np.random.default_rng(1234)
noise_i = rng.standard_normal(len(x))*np.sqrt(noisePwr)
rng = np.random.default_rng(4321)
noise_q = rng.standard_normal(len(x))*np.sqrt(noisePwr)
y = x + noise_i + 1j*noise_q

plt.magnitude_spectrum(y, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()

ye = scs.convolve(psrc, y, method = 'direct')
ye = ye[dly_rc*os:-1 - dly_rc*os]

ye = scs.decimate(ye, os, n = 0, ftype = 'fir')
plt.scatter(np.real(ye), np.imag(ye))
plt.xlabel('Normalkomponente')
plt.ylabel('Quadraturkomponente')
plt.show()

ye = ye*np.exp(-1j*lambda_psk)
ysym = np.rint(m/(2*np.pi)*np.angle(ye))
ysym[ysym < 0] = ysym[ysym < 0] + m
diff = abs(ysym - symbols[0:len(ysym)])
nerror = len(diff[diff > 0])
ser = nerror/len(ysym)
if m <= 4:
    serTheo = erfc(np.sqrt(10**(EbN0/10)))
else:
    serTheo = erfc(np.sqrt(np.log2(m)*10**(EbN0/10))*np.sin(np.pi/m))
print("Eb/N0:", EbN0, "dB", "    Ps (Theo): %1.4e" % serTheo, "    Symbolfehler: %
```






```
Eb/N0: 6 dB     Ps (Theo): 4.7766e-03     Symbolfehler:    2     Symbolfehlerhäufi
gkeit: 2.0020e-03
```

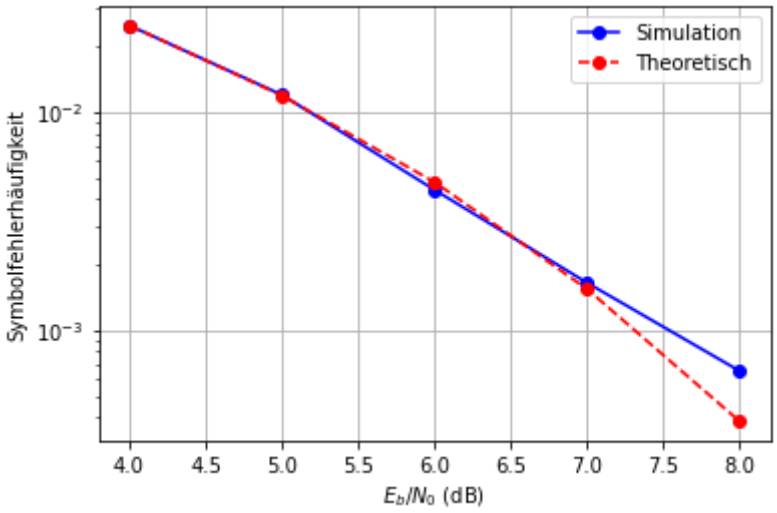
Versuch 2.3: Symbolfehlerhäufigkeit

Abschließend wird wieder die Symbolfehlerhäufigkeit für $E_b/N_0 = 4 \ldots 8$ dB bestimmt und die ermitttelten Werte mit den theoretischen Werten in einem halblogarithmischen Plot verglichen (vgl. Versuch 1.5).

In [5]:
```
nSym = int(2e4)
EbN0List = np.arange(4, 9)
serTheoList = np.zeros(len(EbN0List))
serList = np.zeros(len(EbN0List))

rng = np.random.default_rng(1234)  # Initialisiere Zufallsgenerator
symbols = rng.integers(low = 0, high = m, size = nSym)  # Array der Länge nSym mit
xmod = np.exp(1j*((2*np.pi/m)*symbols + lambda_psk))
xmod = scs.upfirdn([1], xmod, os)
x = scs.convolve(psrc, xmod, method = 'direct')
x = x[dly_rc*os:-1 - dly_rc*os]
sigPwr = 0.5*np.mean(abs(x)**2)

for i in range(0, len(EbN0List)):
    EbN0 = EbN0List[i]
    snr = EbN0 + 10*np.log10(k) - 10*np.log10(os)
    noisePwr = sigPwr/(10**(snr/10))
    rng = np.random.default_rng(1234)
    noise_i = rng.standard_normal(len(x))*np.sqrt(noisePwr)
    rng = np.random.default_rng(4321)
```

```
    noise_q = rng.standard_normal(len(x))*np.sqrt(noisePwr)
    y = x + noise_i + 1j*noise_q
    ye = scs.convolve(psrc, y, method = 'direct')
    ye = ye[dly_rc*os:-1 - dly_rc*os]

    ye = scs.decimate(ye, os, n = 0, ftype = 'fir')

    ye = ye*np.exp(-1j*lambda_psk)
    ysym = np.rint(m/(2*np.pi)*np.angle(ye))
    ysym[ysym < 0] = ysym[ysym < 0] + m
    diff = abs(ysym - symbols[0:len(ysym)])
    nerror = len(diff[diff > 0])
    ser = nerror/len(ysym)
    serList[i] = ser
    if m <= 4:
        serTheo = erfc(np.sqrt(10**(EbN0/10)))
    else:
        serTheo = erfc(np.sqrt(np.log2(m)*10**(EbN0/10))*np.sin(np.pi/m))
    serTheoList[i] = serTheo
    print("Eb/N0:", EbN0, "dB", "    Ps (Theo): %1.4e" % serTheo, "    Symbolfehle

plt.figure()
plt.semilogy(EbN0List, serList, 'bo-', label = 'Simulation')
plt.semilogy(EbN0List, serTheoList, 'ro--', label = 'Theoretisch')
plt.grid()
plt.xlabel('$E_b/N_0$ (dB)')
plt.ylabel('Symbolfehlerhäufigkeit')
plt.legend()
plt.show()
```

```
Eb/N0: 4 dB     Ps (Theo): 2.5002e-02     Symbolfehler:  499     Symbolfehlerhäufi
gkeit: 2.4951e-02
Eb/N0: 5 dB     Ps (Theo): 1.1908e-02     Symbolfehler:  240     Symbolfehlerhäufi
gkeit: 1.2001e-02
Eb/N0: 6 dB     Ps (Theo): 4.7766e-03     Symbolfehler:   88     Symbolfehlerhäufi
gkeit: 4.4002e-03
Eb/N0: 7 dB     Ps (Theo): 1.5453e-03     Symbolfehler:   33     Symbolfehlerhäufi
gkeit: 1.6501e-03
Eb/N0: 8 dB     Ps (Theo): 3.8182e-04     Symbolfehler:   13     Symbolfehlerhäufi
gkeit: 6.5003e-04
```



3 QPSK-Übertragungsstrecke mit Mehrwegekanal und Empfänger mit T/2-Entzerrer

Wir ergänzen die QPSK-Übertragungsstrecke um einen Mehrwegekanal, der als äquivalentes Basisbandmodell simuliert wird. Die durch den Kanal hervorgerufenen Verzerrungen führen zu Intersymbolinterferenz (ISI). Zur Kompensation wird ein T/2-Entzerrer verwendet.

Versuch 3.1: Mehrwegekanal

Der Mehrwegekanal sowie dessen Impulsantwort und Übertragungsfunktion wird in Beispiel 2.9 beschrieben. Zu der Impulsantwort $h(t) = \delta(t) + a\,\delta(t - t_0)$ gehört die Impulsantwort des äquivalenten Tiefpasssystems

$$h_{TP}(t) = \delta(t) + a\,\delta(t - t_0)\ \exp(-j2\pi f_c t_0)$$

wie man sich durch Einsetzen in Gl. (6.3) überzeugen kann. Die Übertragungsfunktion lautet:

$$H_{TP}(f) = 1 + a\ \exp(-j2\pi f_c t_0)\ \exp(-j2\pi f t_0)$$

Der Betrag der Übertragungsfunktion wird geplottet und das Basisbandsignal wird mit der zeitdiskreten Version der Impulsantwort gefaltet. Das Leistungsdichtespektrum des Signals nach dem Mehrwegekanal wird erzeugt, ebenso wie das Augendiagramm nach dem signalangepassten Filter.

Ist $b = f_c\,t_0$ ganzzahlig, so liegt die Trägerfrequenz bei einem Maximum von $|H(f)|$ (siehe Bild 2.20). Für $b = f_c\,t_0 = 1/2$ liegt die Trägerfrequenz dagegen bei einem Minimum. Die Übertragungsfunktion des äquivalenten Tiefpasssystems hat entspechend ein Maximum bzw. ein Minimum bei $f = 0$. In beiden Fällen ist $h_{TP}(t)$ reell und $|H_{TP}(f)|$ ist eine gerade Funktion. Für andere Werte von b ist die Impulsantwort komplex und $|H_{TP}(f)|$ ist nicht mehr symmetrisch bezüglich $f = 0$.

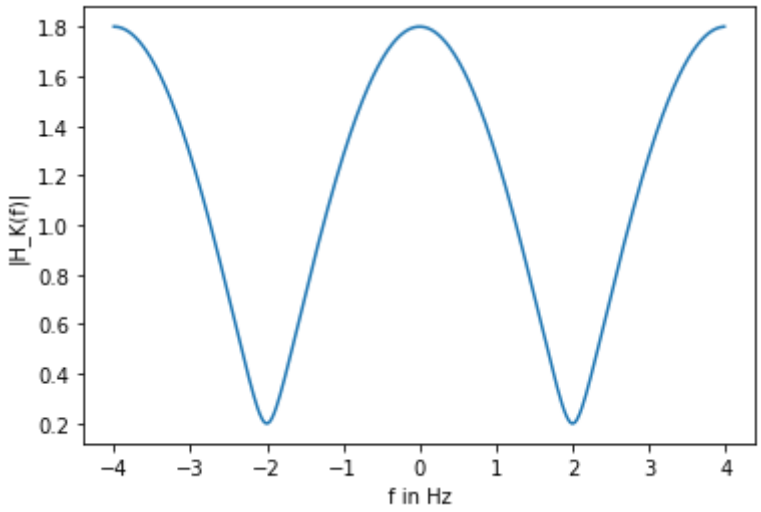
Für $b = 1$ und eine Laufzeitdifferenz t_0 entsprechend zwei Abtastperioden liegt der erste Einbruch der Übertragungsfunktion bei 2 Hz und damit außerhalb des Signals mit der Bandbreite $B_K = 0{,}75$ Hz. Die Intersymbolinterferenz und damit die Auswirkung auf die vertikale Augenöffnung sind nur gering. Für eine Laufzeitdifferenz entsprechend acht Abtastperioden liegt der erste Einbruch der Übertragungsfunktion bei 0,5 Hz, also innerhalb des Signals. Entsprechend gravierend ist die Intersymbolinterferenz.

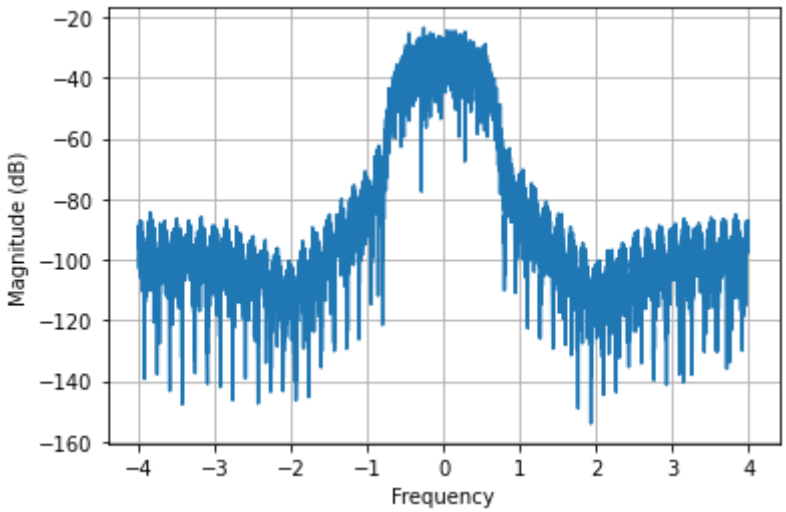
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as scs
from scipy.special import erfc
from scipy.linalg import toeplitz
```

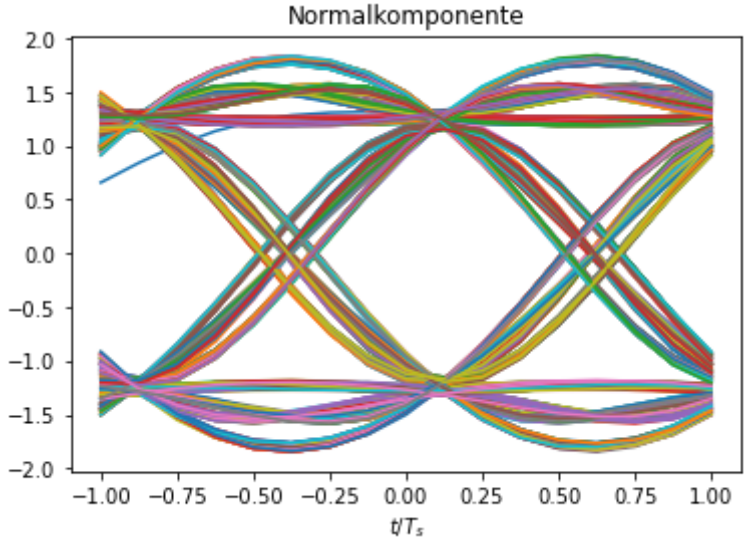
In [2]:
```
# Parameter
alpha = 0.5  # Roll-off-Faktor
os = 8       # Samples pro Symbol (Oversampling Ratio)
dly_rc = 3   # Verzögerung des Wurzel-Kosinus-roll-off-Filters
nSym = 1000
m = 4
k = np.log2(m)
lambda_psk = np.pi/4

# Wurzel-Kosinus-roll-off-Filter
t = np.arange(-dly_rc*os, dly_rc*os + 1)/os
psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1 -
# Grenzwert fuer t/T_s = 0
```

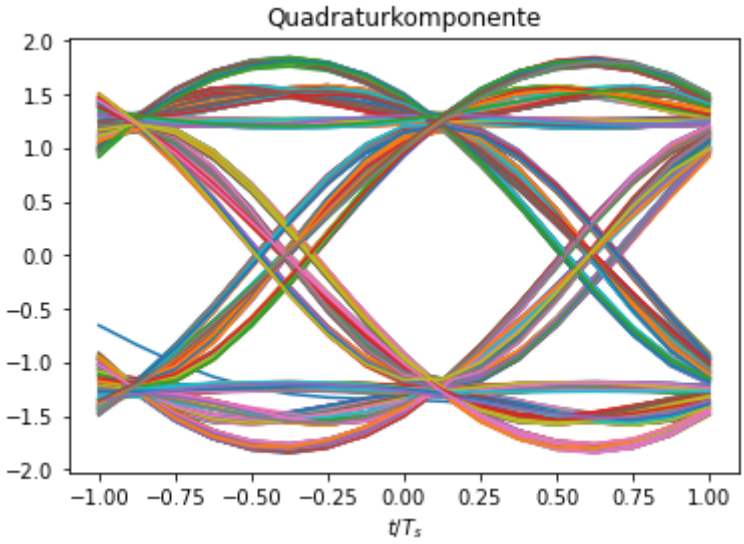
```
psrc[dly_rc*os] = (4*alpha + np.pi*(1 - alpha))/np.pi
# Grenzwert fuer t/T_s = 1/(4 alpha))
if round(os/(4*alpha)) == os/(4*alpha): # Ist os/(4*alpha) ganzzahlig?
    psrc[int(dly_rc*os + os/(4*alpha))] = (4*alpha*np.cos(np.pi*(1 + alpha)/(4*alpl
                                          - np.pi*(1 + alpha)*np.sin(np.pi*(1 + a
                                          + np.pi*(1 - alpha)*np.cos(np.pi*(1 - a
    psrc[int(dly_rc*os - os/(4*alpha))] = psrc[int(dly_rc*os + os/(4*alpha))]

E = sum(psrc**2)
psrc = psrc/np.sqrt(E)

rng = np.random.default_rng(1234)  # Initialisiere Zufallsgenerator
symbols = rng.integers(low = 0, high = m, size = nSym)  # Array der Länge nSym mit
xmod = np.exp(1j*((2*np.pi/m)*symbols + lambda_psk))
xmod = scs.upfirdn([1], xmod, os)
x = scs.convolve(psrc, xmod, method = 'direct')
x = x[dly_rc*os:-1 - dly_rc*os]

# Mehrwegekanal
dt = 2   # Laufzeitdifferenz (in Samples)
a = 0.8  # Dämpfung des indirekten Pfades
b = 1    # Produkt f_c * t_0

# Impulsantwort
hKanal = np.zeros(dt + 1) + 1j*np.zeros(dt + 1)
hKanal[0] = 1
hKanal[dt] = a*np.exp(-1j*2*np.pi*b)

# Übertragungsfunktion
f = np.arange(-256, 256)*os/512
HKanal = 1 + a*np.exp(-1j*2*np.pi*b)*np.exp(-1j*2*np.pi*f*dt/os)
plt.figure()
plt.plot(f, abs(HKanal))
plt.xlabel('f in Hz')
plt.ylabel('|H_K(f)|')
plt.show()

r = scs.convolve(hKanal, x, method = 'direct')

plt.magnitude_spectrum(r, Fs = os, scale = 'dB', sides = 'twosided')
#plt.title('sine signal + noise')
plt.grid()
plt.show()

ye = scs.convolve(psrc, r, method = 'direct')
ye = ye[dly_rc*os:-1 - dly_rc*os]

teye = np.arange(-os, os + 1)/os
yi = np.real(ye)
yq = np.imag(ye)
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, yi[i1:i2])
plt.title('Normalkomponente')
plt.xlabel('$t / T_s$')
plt.show()
for i in range(0, round(nSym/2) - 1):
    i1 = 2*i*os
    i2 = 2*(i + 1)*os + 1
    plt.plot(teye, yq[i1:i2])
plt.title('Quadraturkomponente')
plt.xlabel('$t / T_s$')
plt.show()
```

```
C:\Users\roppel\AppData\Local\Temp\ipykernel_764\1830563861.py:12: RuntimeWarning:
divide by zero encountered in true_divide
  psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1
- (4*alpha*t)**2)*np.pi*t)
C:\Users\roppel\AppData\Local\Temp\ipykernel_764\1830563861.py:12: RuntimeWarning:
invalid value encountered in true_divide
  psrc = (4*alpha*t*np.cos(np.pi*(1 + alpha)*t) + np.sin(np.pi*(1 - alpha)*t))/((1
- (4*alpha*t)**2)*np.pi*t)
```









Versuch 3.2: T/2-Entzerrer

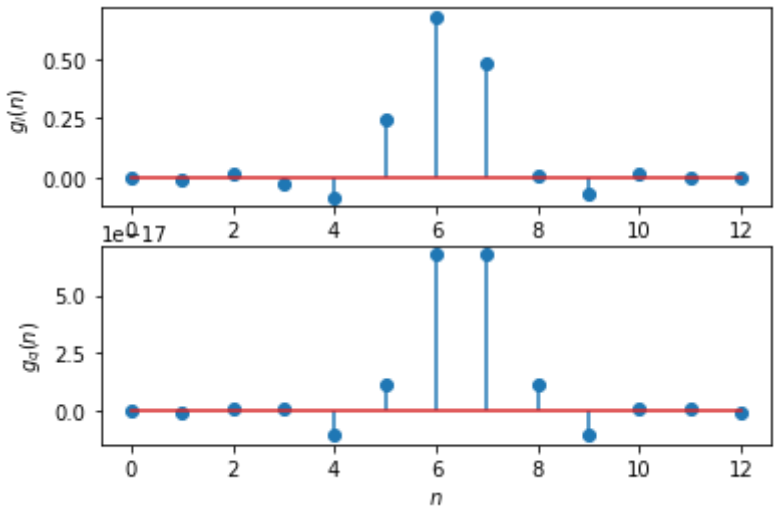
Der Einfluss des Mehrwegekanals wird durch einen Entzerrer kompensiert, der mit 2 Abtastwerten pro Symbol arbeitet (T/2-Entzerrer). Er übernimmt gleichzeitig die Funktion des signalangepassten Filters (siehe Abschnitte 5.4.2 und 6.5). Für die Bestimmung der Koeffizienten wird die MMSE-Lösung verwendet.

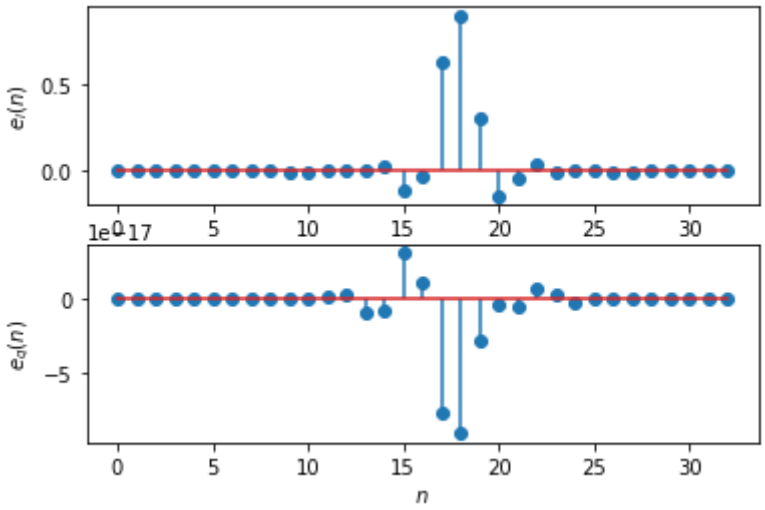
Der Entzerrer hat 33 Koeffizienten. Zunächst wird die vollständige Faltungsmatrix aufgestellt, anschließend wird jede zweite Zeile gestrichen. Die Berechnung der Koeffizienten erfolgt gemäß den Gln. (5.62), (5.63) und (5.64). Für die Rauschleistung σ_r^2 wird zunächst ein sehr kleiner Wert (10^{-6}) angenommen.

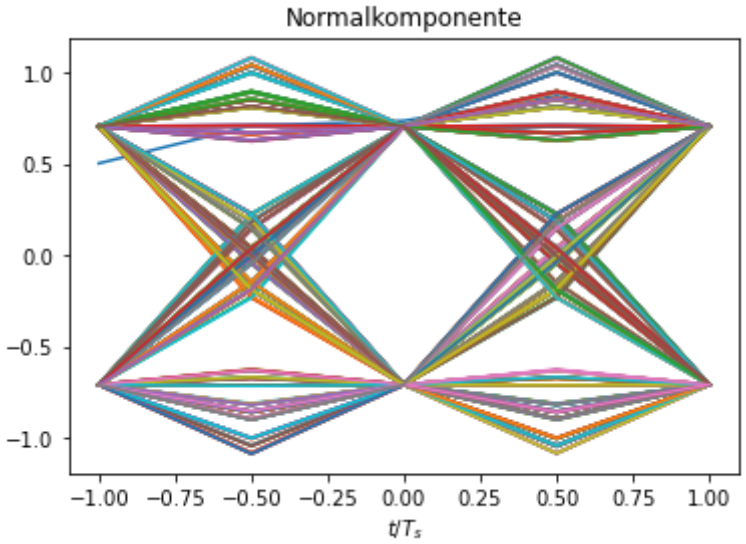
Der zu entzerrende Impuls $g(n)$ und die Impulsantwort des Entzerrers wird geplottet. Die Abtastrate des Eingangssignals wird auf 2 Abtastwerte pro Symbol herabgesetzt. Nun kann das Signal mit der Entzerrer-Impulsantwort gefaltet werden, und das Augendiagramm wird erzeugt.

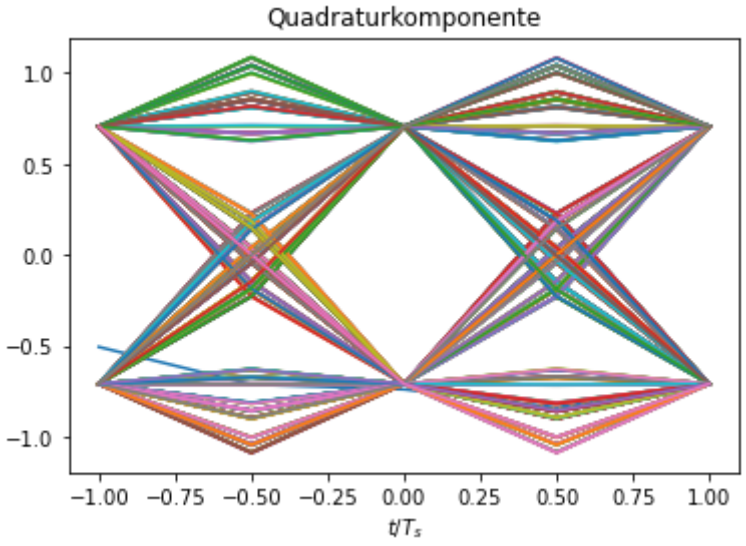
In [3]:
```
#### T/2-Entzerrer
Ne = 33                    # Anzahl der Koeffizienten
dly_eq = round((Ne-1)/2)   # Verzögerung des Entzerrers

# Zu entzerrender Impuls g(n)
gn = scs.convolve(psrc, hKanal)
gn = scs.decimate(gn, round(os/2), n = 0, ftype = 'fir')

# Rauschleistung
var_r = 1e-6
# Faltungsmatrix mit m + 1 + N Zeilen und m + 1 Spalten
first_col = np.concatenate((gn, np.zeros(Ne - 1)))
first_row = np.concatenate(([gn[0]], np.zeros(Ne - 1)))
F = toeplitz(first_col, first_row)
# F hat z Zeilen und s Spalten, Weglassen jeder zweiten Zeile
[z, s] = F.shape
F = np.delete(F, np.arange(1, z, 2), 0)
# Ideales Ausgangssignal
[z, s] = F.shape
yid = np.zeros(z)
yid[round(z/2)] = 1
# Berechnung der Koeffizienten
I = np.identity(s)
```

```
F_H = np.transpose(np.conjugate(F))
en = np.matmul(np.matmul(np.linalg.inv(np.matmul(F_H, F) + var_r*I), F_H), yid)

plt.figure()
plt.subplot(2, 1, 1)
plt.stem(np.real(gn))
plt.ylabel('$g_i(n)$')
plt.subplot(2, 1, 2)
plt.stem(np.imag(gn))
plt.xlabel('$n$')
plt.ylabel('$g_q(n)$')
plt.show()

plt.figure()
plt.subplot(2, 1, 1)
plt.stem(np.real(en))
plt.ylabel('$e_i(n)$')
plt.subplot(2, 1, 2)
plt.stem(np.imag(en))
plt.xlabel('$n$')
plt.ylabel('$e_q(n)$')
plt.show()

# Eingangssignal, 2 Samples/Symbol, Entzerrung
r_ds = scs.decimate(r, round(os/2), n = 0, ftype = 'fir')
y = scs.convolve(en, r_ds)
y = y[dly_eq + 2:-1 - dly_eq]

teye = np.arange(-2, 3)/2
yi = np.real(y)
yq = np.imag(y)
eyemax = round(np.floor(len(y)/4)) - 1
for i in range(0, eyemax):
    i1 = 2*i*2
    i2 = 2*(i + 1)*2 + 1
    plt.plot(teye, yi[i1:i2])
plt.title('Normalkomponente')
plt.xlabel('$t / T_s$')
plt.show()
for i in range(0, eyemax):
    i1 = 2*i*2
    i2 = 2*(i + 1)*2 + 1
    plt.plot(teye, yq[i1:i2])
plt.title('Quadraturkomponente')
plt.xlabel('$t / T_s$')
plt.show()
```









Versuch 3.3: Symbolfehlerwahrscheinlichkeit

Wir ermitteln wieder die Symbolfehlerhäufigkeit für $E_b/N_0 = 4 \ldots 8$ dB (siehe Versuch 2.3). Am Ausgang des Mehrwegekanals erfolgt die Reduzierung der Abtastrate auf 2 Samples pro Symbol. Für die Berechnung der Entzerrerkoeffizienten verwenden wir einen mittleren Wert der Rauschleistung (0.04).

Das Rauschen wird am Ausgang des Mehrwegekanals addiert, also bei einer Abtastrate von 2 Samples pro Symbol. Dies muss bei der Berechnung des Störabstandes berücksichtigt

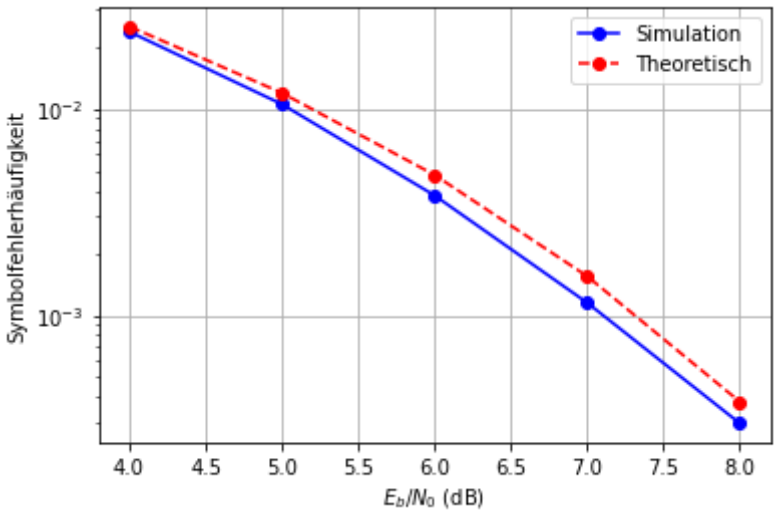
werden.

Für $b = 1$ und eine Laufzeitdifferenz von 2 Abtastperioden kann der Entzerrer den Einfluss des Mehrwegekanals vollständig kompensieren und wir erhalten für die Symbolfehlerhäufigkeit die gleichen Werte wie im Falle des AWGN-Kanals und dem idealen Empfänger mit signalangepasstem Filter. Dies gilt aber nicht für beliebige Konfigurationen des Mehrwegekanals, insbesondere wenn der Einbruch der Übertragungsfunktion noch innerhalb der Signalbandbreite B_K liegt.

In [4]:
```
nSym = int(2e4)
EbN0List = np.arange(4, 9)
serTheoList = np.zeros(len(EbN0List))
serList = np.zeros(len(EbN0List))

rng = np.random.default_rng(1234)  # Initialisiere Zufallsgenerator
symbols = rng.integers(low = 0, high = m, size = nSym)  # Array der Länge nSym mit
xmod = np.exp(1j*((2*np.pi/m)*symbols + lambda_psk))
xmod = scs.upfirdn([1], xmod, os)
x = scs.convolve(psrc, xmod, method = 'direct')
x = x[dly_rc*os:-1 - dly_rc*os]

# Mehrwegekanal
r = scs.convolve(hKanal, x, method = 'direct')

# Eingangssignal, 2 Samples/Symbol
r_ds = scs.decimate(r, round(os/2), n = 0, ftype = 'fir')
sigPwr = 0.5*np.mean(abs(r_ds)**2)

# Rauschleistung
var_r = 0.04
# Entzererkoeffizienten
en = np.matmul(np.matmul(np.linalg.inv(np.matmul(F_H, F) + var_r*I), F_H), yid)

for i in range(0, len(EbN0List)):
    EbN0 = EbN0List[i]
    snr = EbN0 + 10*np.log10(k) - 10*np.log10(2)
    noisePwr = sigPwr/(10**(snr/10))
    rng = np.random.default_rng(1234)
    noise_i = rng.standard_normal(len(r_ds))*np.sqrt(noisePwr)
    rng = np.random.default_rng(4321)
    noise_q = rng.standard_normal(len(r_ds))*np.sqrt(noisePwr)
    rn = r_ds + noise_i + 1j*noise_q
    # Entzerrer und Demodulator
    y = scs.convolve(en, rn)
    y = y[dly_eq + 2:-1 - dly_eq]

    y = scs.decimate(y, 2, n = 0, ftype = 'fir')

    y = y*np.exp(-1j*lambda_psk)
    ysym = np.rint(m/(2*np.pi)*np.angle(y))
    ysym[ysym < 0] = ysym[ysym < 0] + m
    diff = abs(ysym - symbols[0:len(ysym)])
    nerror = len(diff[diff > 0])
    ser = nerror/len(ysym)
    serList[i] = ser
    serTheo = erfc(np.sqrt(10**(EbN0/10)))
    serTheoList[i] = serTheo
    print("Eb/N0:", EbN0, "dB", "    Ps (Theo): %1.4e" % serTheo, "    Symbolfehle

plt.figure()
plt.semilogy(EbN0List, serList, 'bo-', label = 'Simulation')
```

```
plt.semilogy(EbN0List, serTheoList, 'ro--', label = 'Theoretisch')
plt.grid()
plt.xlabel('$E_b/N_0$ (dB)')
plt.ylabel('Symbolfehlerhäufigkeit')
plt.legend()
plt.show()
```

```
Eb/N0: 4 dB     Ps (Theo): 2.5002e-02     Symbolfehler:  471     Symbolfehlerhäufi
gkeit: 2.3552e-02
Eb/N0: 5 dB     Ps (Theo): 1.1908e-02     Symbolfehler:  210     Symbolfehlerhäufi
gkeit: 1.0501e-02
Eb/N0: 6 dB     Ps (Theo): 4.7766e-03     Symbolfehler:   76     Symbolfehlerhäufi
gkeit: 3.8004e-03
Eb/N0: 7 dB     Ps (Theo): 1.5453e-03     Symbolfehler:   23     Symbolfehlerhäufi
gkeit: 1.1501e-03
Eb/N0: 8 dB     Ps (Theo): 3.8182e-04     Symbolfehler:    6     Symbolfehlerhäufi
gkeit: 3.0003e-04
```



In [ ]: