Karlsruher Institut für Technologie

Vorlesung:

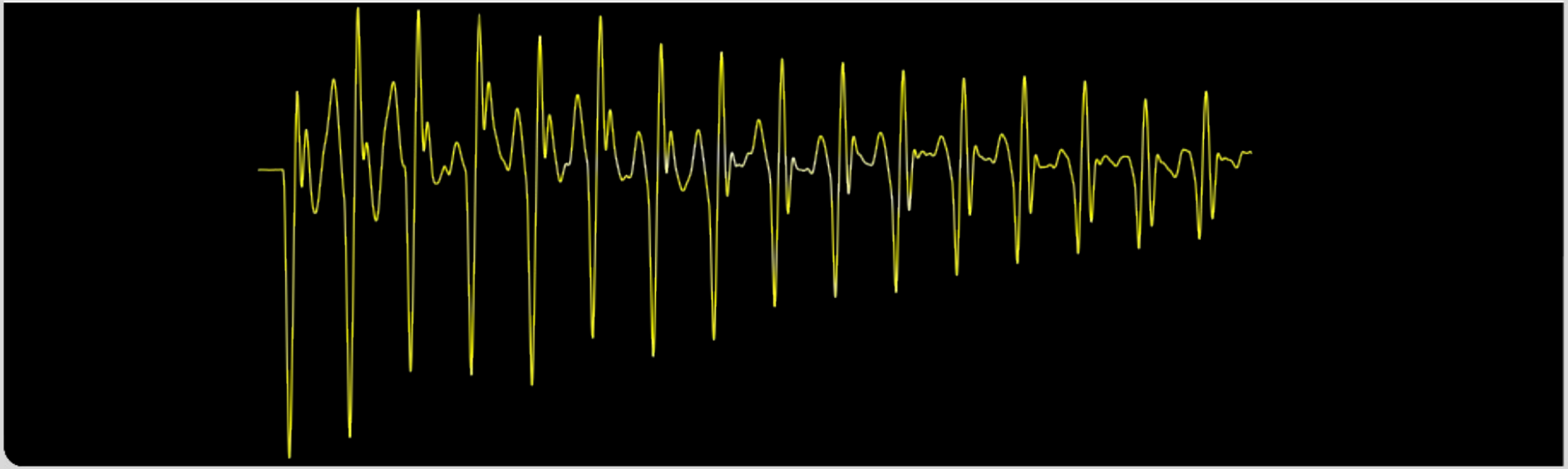
Rechnernutzung in der Physik

Digitale Signalverarbeitung

Günter Quast

Fakultät für Physik
Institut für Experimentelle Kernphysik

WS '23/24



Datenquellen

„Daten“ erreichen den Computer aus verschiedensten Quellen:

- **Direkte Eingabe:** z.B. Tastatur
- **Datei:** auf dem gleichen oder einem anderen System erzeugte Daten; Transfer über mobile Datenträger oder Netzwerk
- **Simulation:** im Rechner selbst erzeugt
- **Peripherie:** von am Rechner über unterschiedlichste Schnittstellen angeschlossenen Peripheriegeräten (Sensoren, Messgeräte, Scanner, Kamera …)

Analoge Daten müssen zur Verwendung im Rechner zunächst „digitalisiert“ werden

(s. Analog-Digitalwandlung in Block 4)

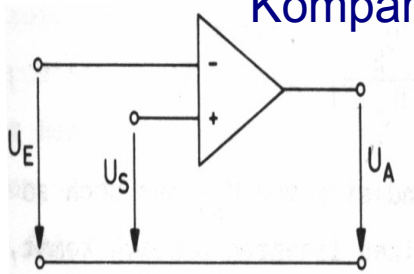
Digital-Analog- bzw. **Analog-Digital**-Wandler sind prinzipiell Abbildungen zwischen Digitalzahlen D und Werten der elektrischen Spannung U

$$0 \leq D < 2^n \quad \leftrightarrow \quad U_{min} \leq U < U_{max}$$

Prinzip Digital-Analog-Wandlung

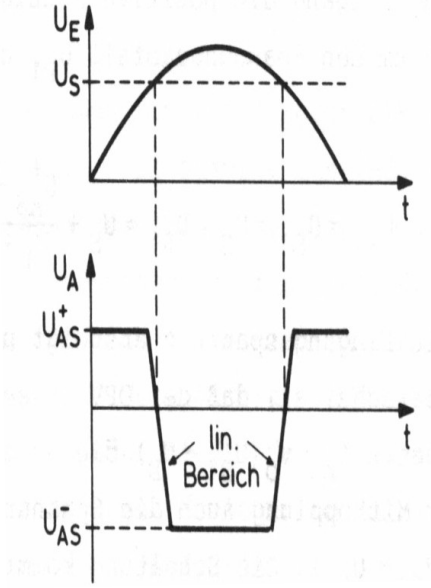
Schlüsselelement:

Komparator



z.B. mit einfachem OP:
$U_E < U_S$: $U_A = U^+$
$U_E > U_S$: $U_A = U^-$
d.h. **digitale Ausgabe !**

Sukzessiv oder parallel erfolgender Vergleich einer Spannung U_E mit Referenz-Spannung(en) U_S ergibt Bitmuster, das auf eine Binärzahl codiert wird.



Grundprinzip einer Analog-Digitalwandlung mittels Komparator

---

erzeugter Datenstrom besteht aus Messwerten zu festen Zeiten t_i:

t_1, m_1
...
t_i, m_i
...
t_N, m_N

üblich:
Text-Datei im **csv**-Format
(=**c**omma **s**eparated **v**alues)

Fast alle modernen digitalen Messgeräte enthalten Exportfunktionen für die aufgezeichneten Rohdaten, die über Schnittstellen **(heute fast ausschließlich USB)** an einen PC übertragen werden.

Digitale Abtastung („sampling“)

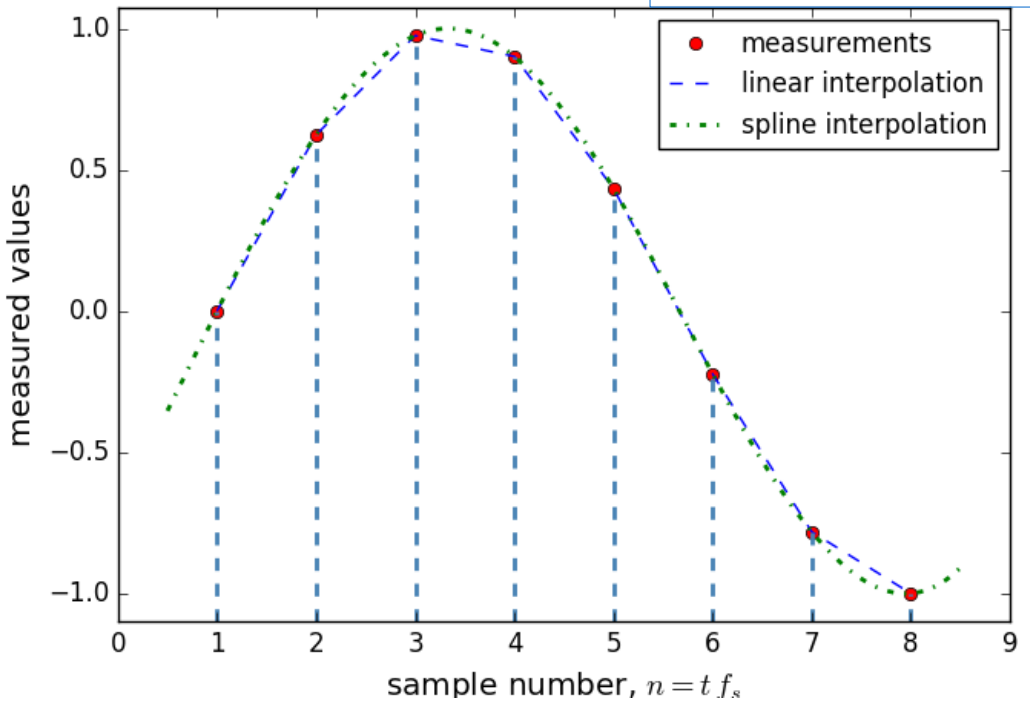
Digitale Messgeräte tasten Signale üblicherweise regelmäßig zu festen Zeiten $t_n = t_0 + n\,T$ mit der Abtast- oder Sampling-Rate

$f_s = 1/T$ ab.

Nummer der Messung

$n = t_n\ f_s$ (für $t_0 = 0$s)

Script `sampling.py`



Datenexport

in Felder mit

- Zeitpunkten t_i (`t[1, …, N]`)
- Messwerten v_i (`v[1, …, N]`)

Grafische Darstellung als Markierungen, evtl.

– verbunden durch gerade Linien „lineare Interpolation“

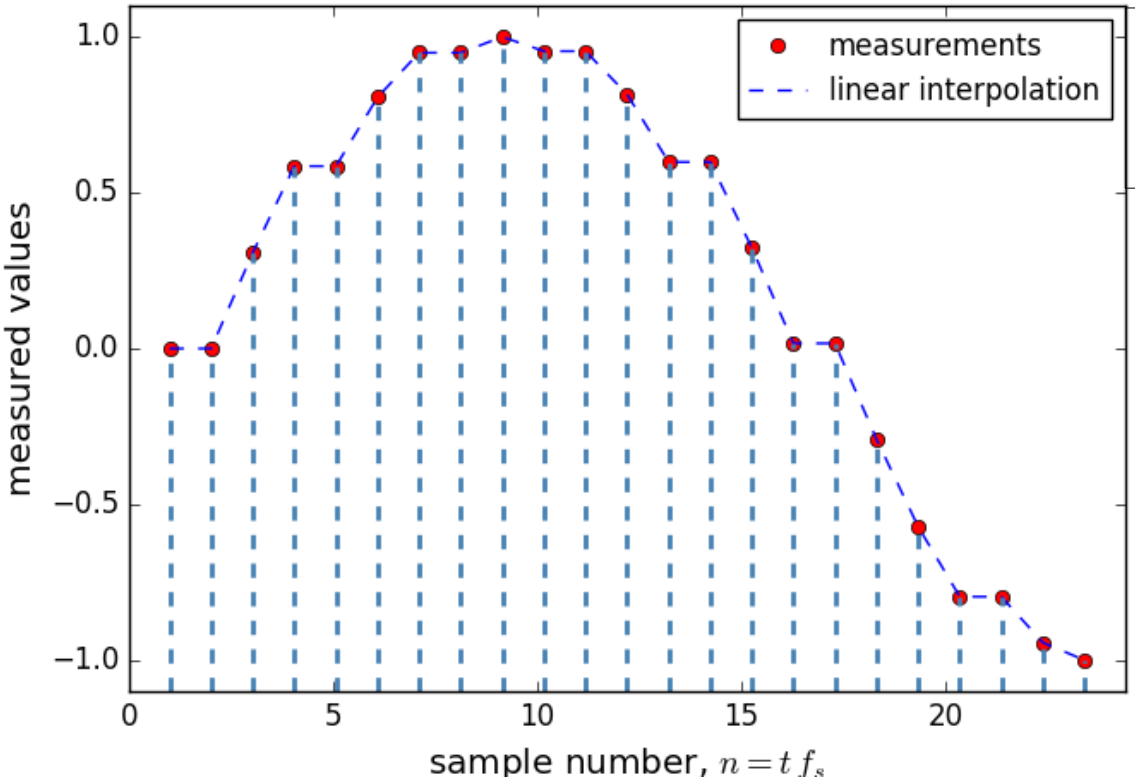
– verbunden durch Kurven, z.B. „Spline-Interpolation“

(kubische Splines:
stetig differenzierbar aneinander anschließende Polynomstücke 3. Grades)

Digitale Abtastung: Oversampling

Wenn man häufiger ausliest, als die Analog-Digitalwandlung Werte liefern kann,

passiert dies:



Stufen durch Überabtastung (**Oversampling**) eines langsamen Signals

Sollte vermieden werden

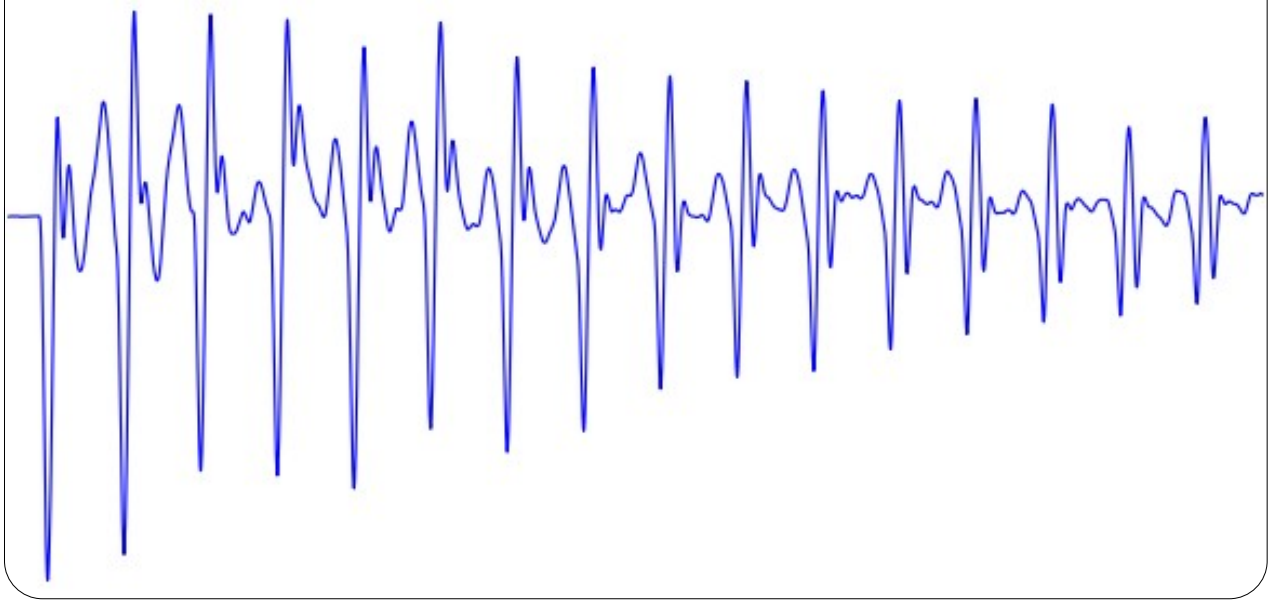
– erfordert ansonsten „Nachbearbeitung“ des digitalisierten Signals

Beispiel: Rohdaten einer Wellenform

```
Time,Channel A
(ms),(V)

-0.34927999,-0.00045778
-0.34799999,-0.00045778
-0.34671999,-0.00045778
-0.34543999,-0.00045778
-0.34415999,-0.00045778
-0.34287999,-0.00033570
-0.34159999,-0.00018311
-0.34031999,-0.00018311
-0.33903999,-0.00018311
-0.33775999,-0.00003052
-0.33647999,0.00006104
-0.33519999,0.00006104
-0.33391999,0.00006104
-0.33263999,0.00006104
-0.33135999,0.00021363
-0.33007999,0.00021363

        . . .

9.63983995,0.02642903
9.64111995,0.02655110
9.64239995,0.02655110
9.64367995,0.02655110
9.64495995,0.02655110
9.64623995,0.02655110
9.64751995,0.02655110
9.64879995,0.02642903
9.65007995,0.02612384
9.65135995,0.02584918
9.65263995,0.02557451
9.65391995,0.02526933
9.65519995,0.02487259
```

7816 Wertepaare exportiert aus USB-Oszilloskop **PicoScope** im **csv**-Format (**c**omma **s**eparated **v**alues)

importiert in numpy-arrays `t` und `v`, dargestellt mit `plt.plot(t,v)`



Software-Werkzeuge

Die im Folgenden gezeigten Funktionen zur Signalbearbeitung (und viele andere mehr) sind typische Bestandteile der mit Messgeräten ausgelieferten Betriebssoftware.

Einfache (und auf das Wesentliche reduzierte) Implementierungen finden sich im Paket

`PhyPraKit`, `http://www.ekp.kit.edu/~quast/PhyPraKit/htmldoc/`

(`PhyPraKit.py` in den Beispielen zu dieser Vorlesung)

Das Paket `scipy.signal` enthält eine sehr große Anzahl frei verfügbarer `python`-Module für eigene Anwendungen.

(übliche) Datenformate

Messgeräte (auch „Datenlogger“) und einige Handy-Apps (z.B. phyphox) nutzen einfache Datenformate in Text-Form:

Beispiel PicoScope, „Comma Separated Values“ (CSV)

```
Time,Channel A
(ms),(V)

-0.349,-0.000458
-0.348,-0.000458
-0.347,-0.000458
    . . .
```

} Kopfzeilen mit sog. „**Meta-Daten**“

} die eigentlichen **Daten** als Dezimal-zahlen, in Spalten durch „,“ getrennt

Üblich sind auch „Tabulator-getrennte“ Dezimalzahlen und - bisweilen – auch Dezimalzahlen mit „ **,** “ statt „ **.**“
(dann müssen für die Verwendung in python-Programmen Dezimalkommata durch Dezimalpunkte ersetzt werden!)

Durchgesetzt haben sich „beschreibende“ Datenformate, z.B.
xml = „**e**xtensible **m**arkup **l**anguage“ oder
json = „**j**ava **s**cript **o**bject **n**otation“ (≙ python dictionary);
gut durch pyhton-Module unterstützt !

Rein „binäre“ Datenformate (also sehr kompakte Darstellungen in Maschinen-abhängigem digitalem Format) werden wegen ihrer Plattformabhängigkeit heute kaum noch verwendet.

Beispiel-Code

```
#  Daten im csv-Format lesen

# Datei zum Lesen öffnen
f = open('AudioData.csv', 'r')

# Kopzeile(n) lesen
header=f.readline()
print "Kopfzeile:", header

# Daten in 2D-numpy-array einlesen
data = np.loadtxt(f,
        delimiter=',', unpack=True)
print "-> Anzahl Spalten",
        data.shape[0]
print "-> Datenzeilen",
        data.shape[1]

# Daten in 1D-arrays speichern
t = data[0]
a = data[1]
l = len(a)
```

s. auch
- PhyPraKit.readCSV()
- PhyPraKit.readtxt()
- PhyPraKit.readColumnData()
- PhyPraKit.labxParser()

u. a.

Signalglättung

Verrauschte bzw. aus anderen Gründen nicht-stetige Signale lassen sich recht einfach **glätten**:

„gleitender Mittelwert“ über 2k+1Punkte

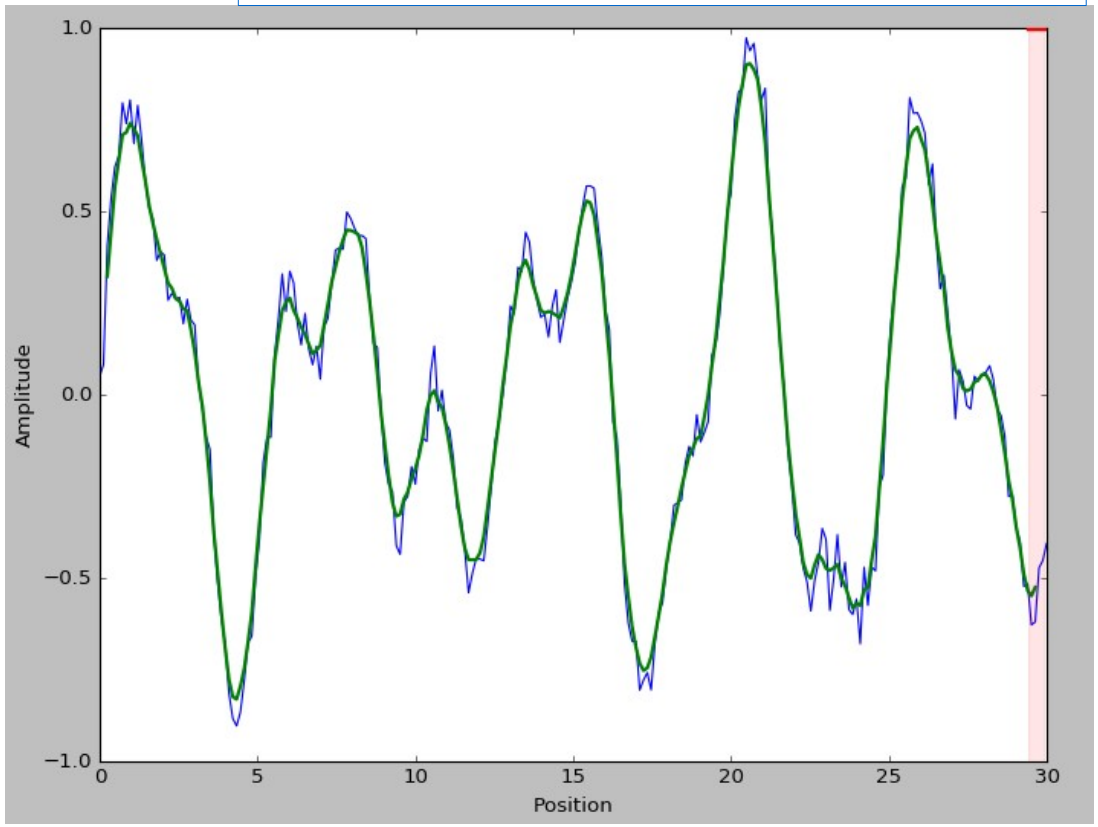
$$v_i = \frac{1}{2k+1} \sum_{j=-k}^{k} v_{i+j}$$

Wert von k dem Störsignal entsprechend anpassen !

Diese Operation stellt einen „Tiefpass“ dar, der hochfrequente Anteile herausfiltert

(ähnlich einer trägen analogen Anzeige)

Script `animated_convolutionFilter.py`



Sample mit 250 Punkten, gleitender Mittelwert über 5 Punkte

Filtern

Formal:

Faltung (engl. „convolution“) des Signals mit einer Filterfunktion f_k

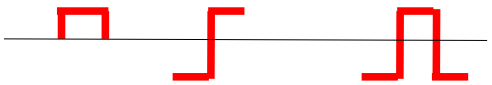
$$v_i = \sum_{j=-k}^{k} f_j \cdot v_{i+j},$$

mit f_k gewichteter Mittelwert über $2k+1$ benachbarte Punkte

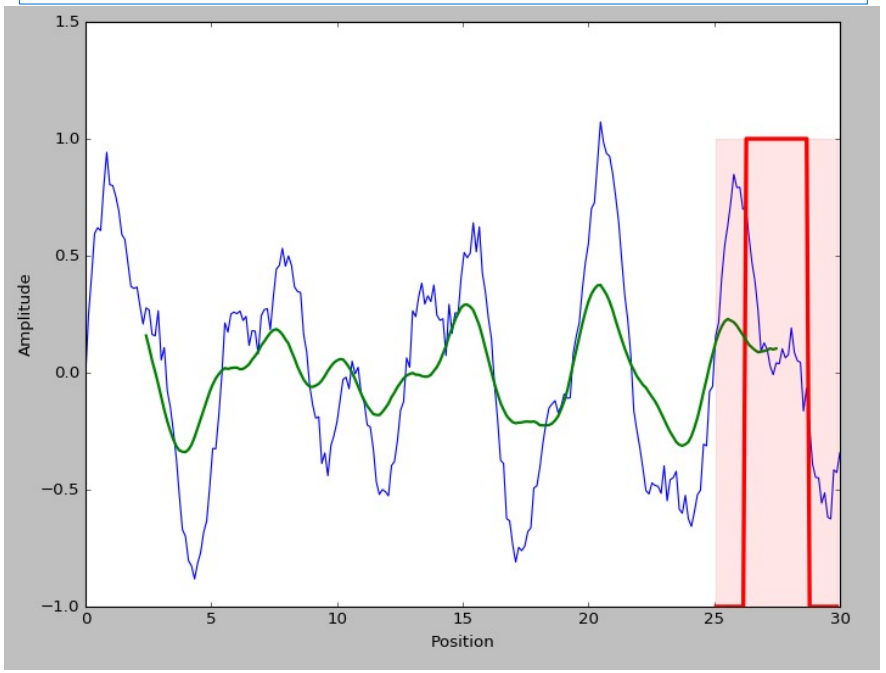
Je nach Wahl der Form von f spricht ein solcher **Filter** auf verschiedene Eigenschaften des Signals an:

glatt Ecke Maximum

Beispiele **f**:



Script `animated_convolutionFilter.py`



Ergebnis einer Maximum-Suche mit der eingezeichneten Filterfunktion

Maxima der glatten grünen Kurve entsprechen den breiten Maxima des Signals

als 2d-Variante werden solche „Filtermatrizen“ auch in der Bildverarbeitung angewandt: Glätten, Randerkennung, Schärfung ...

Beispiel: Analyse einer Wellenform

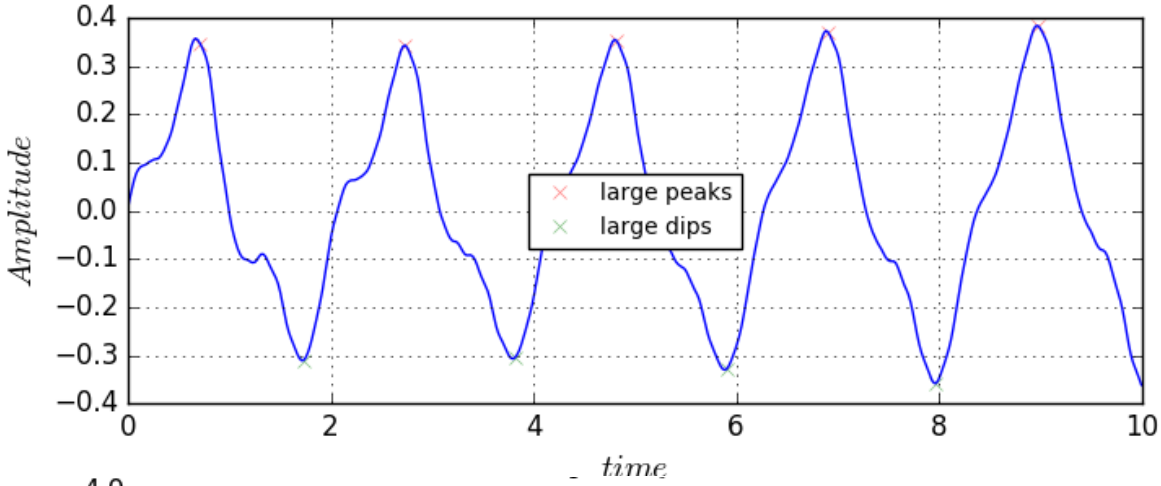
Daten aus der Android-App pyhphox
(Mikrofonaufzeichnung)

im csv-Format mit 1 Kopfzeile

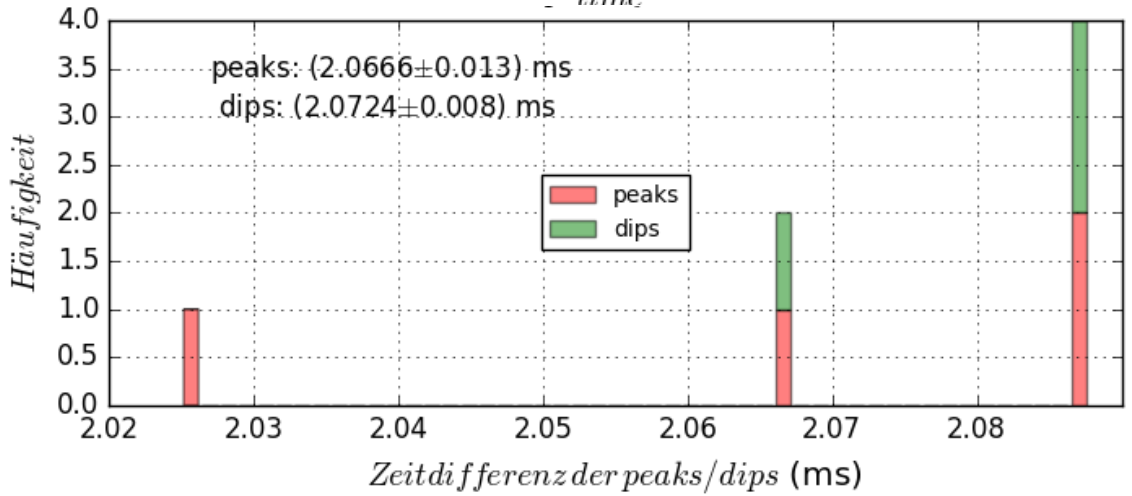
```
"Time (ms)","Recording (a.u.)"
0.000000000E0,6.622516556E-3
2.087682672E-2,2.774132511E-2
4.175365344E-2,4.705954161E-2
          . . .
```

Beispiel:

semi-automatische Suche nach **Maxima** und **Minima** mit **Convolution Filter**



Häufigkeitsverteilung der Zeitdifferenzen zwischen den Extrema

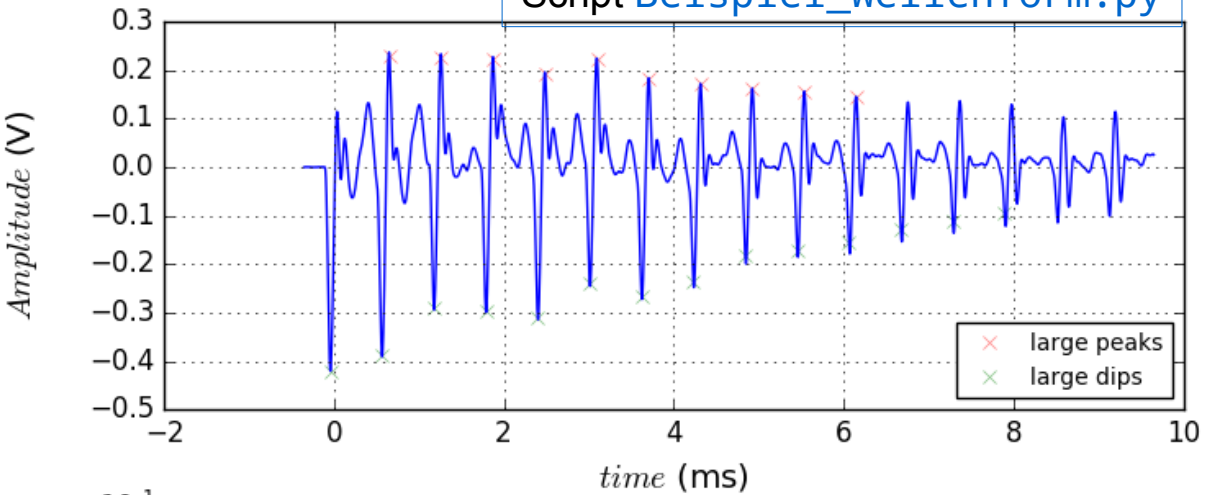


Script `test_convolutionPeakFinder.py`

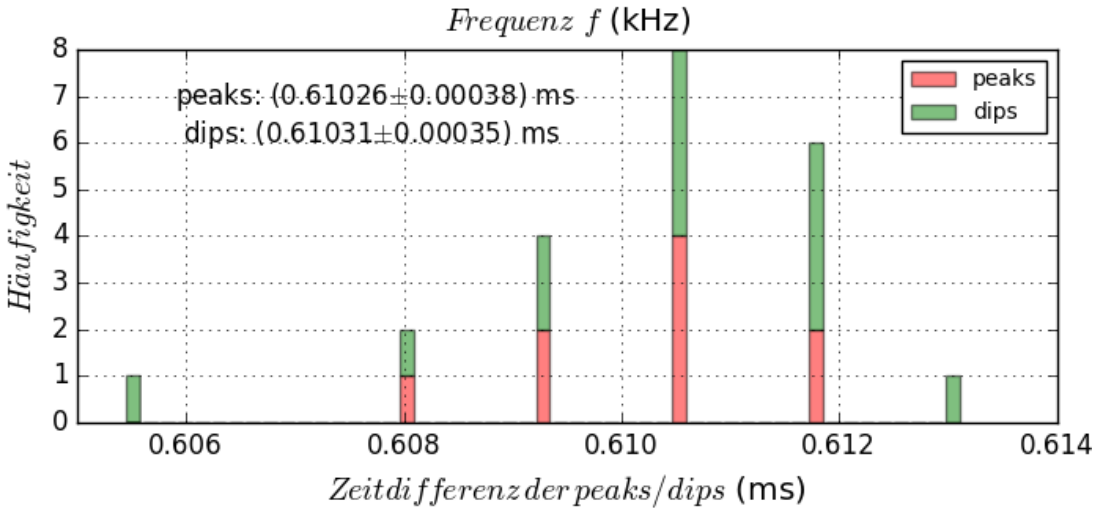
Beispiel: Analyse einer Wellenform

semi-automatische Suche nach Maxima und Minima mit Convolution-Filter

Script `Beispiel_Wellenform.py`



Häufigkeitsverteilung der Zeitdifferenzen zwischen den Extrema



Frequenz einer periodischen Wellenform

Autokorrelation

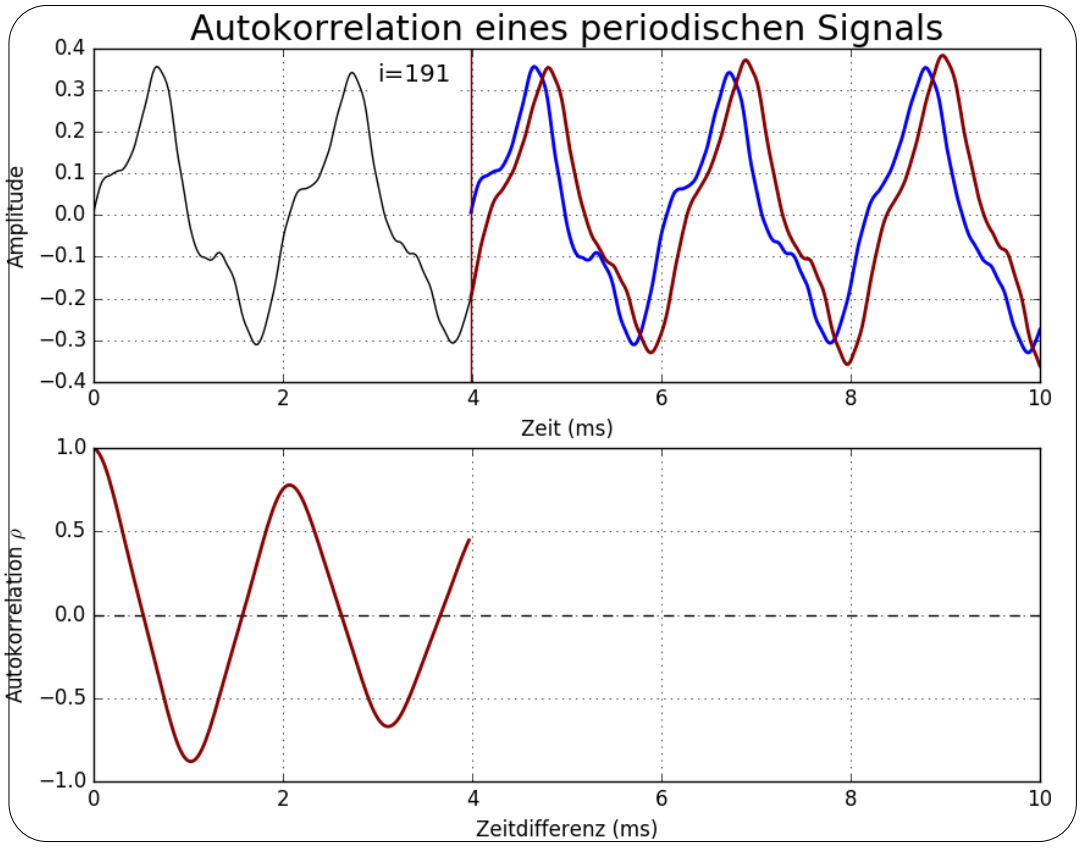
Faltung mit **verschobener Version** von sich selbst:

$$\rho_0 = \sum_{k=0}^{l-1} a_i\, a_i\ ,\ \rho_i = \frac{1}{\rho_0} \sum_{k=0}^{l-i-1} a_k\, a_{i+k},\ i=1,\ \ldots,\ \text{n-1}$$

Wegen der Selbstähnlichkeit der jeweiligen Perioden nimmt ρ Maxima an, wenn die Verschiebung einer Periodenlänge entspricht

```
# einfache Implementierung
  ...
l=len(a)
rho=np.zeros(l)
for i in range(1, l):
  rho[i]=np.inner(a[:-i],a[i:])
rho[0] = np.inner(a, a)
rho = rho/rho[0]
  ...
```

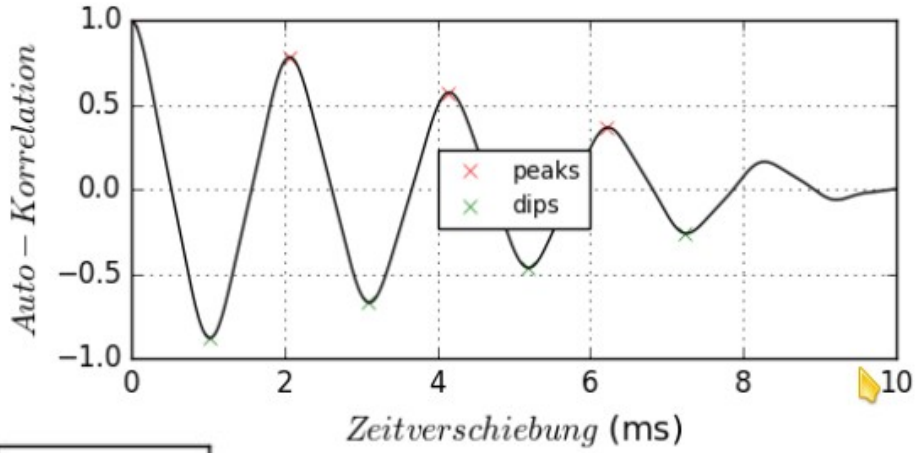
`np.inner(a,b)` *kennen wir in der Mathematik als das Skalarprodukt von a und b*



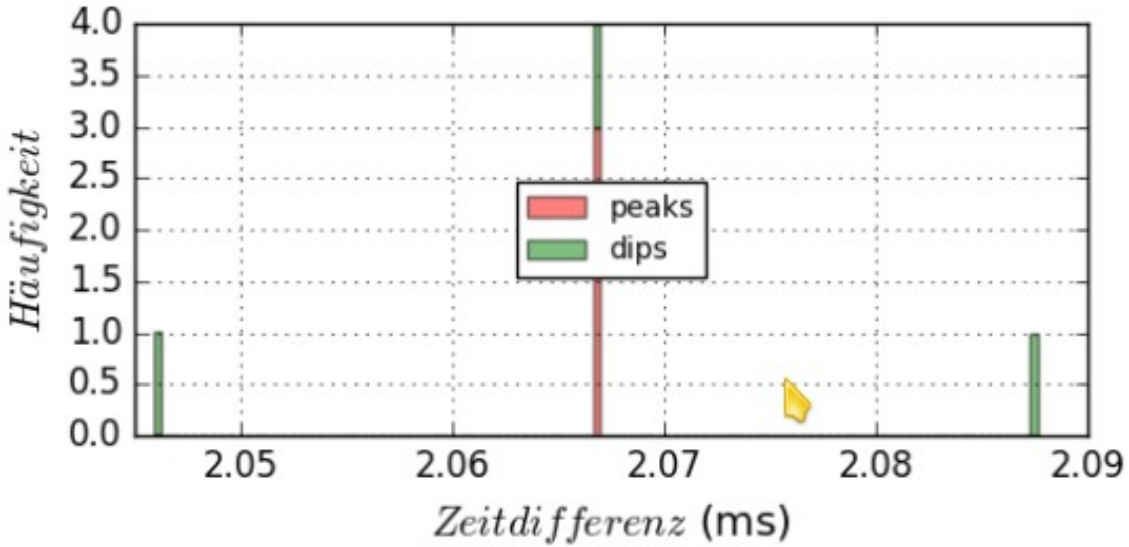
animate_autoCorrelation.py

Frequenz eine periodischen Wellenform (2)

Bestimmung der Maxima und Minima der Autokorrelation und Differenzbildung liefert sehr genaue Bestimmung der



Periodendauer



Beispiel-Skript:
`test_AutoCorrelation.py`

Zeitdifferenz T aus Maxima: (2.0668 +/- 0 [0,0060]*)) ms
bzw. Minima: (2.0668 +/- 0.0049) ms der Autokorrelation

Anm.: Unsicherheiten aus Unsicherheit des Mittelwerts („$\sigma/\sqrt{n}$“);
*) Die Unsicherheit von 0 auf T aus Maxima bedeutet, dass Abweichungen kleiner sind als die Sampling-Zeit t_s = 0.0208 ms; als Unsicherheit wurde daher die Standardabweichung einer Rechteckverteilung mit Breite 0.0208 ms angenommen

Interpolation

Aufgrund der diskreten Natur abgetasteter Daten ergibt sich die Notwendigkeit, Zwischenwerte zu berechnen → **Interpolation**

Einfach und schon aus der Schule bekannt: **Lineare Interpolation**

$n+1$ Datenpunkte $y_j,\ j = 0, \ldots, n$ und n Geradenstücke $l_j(x),\ j = 0, \ldots, n$:

$$l_j(x) = y_{j-1} + \frac{y_j - y_{j-1}}{x_j - x_{j-1}}(x - x_{j-1}),\ \ x \in [x_{j-1}, x_j]$$

Besser: quadratische, oder noch besser, kubische Funktionen, die stetig und differenzierbar aneinander anschließen:

„Kubische Splines“

$$s_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j)^3,\ \ x \in [x_{j-1}, x_j]$$
$$j = 1, \ldots, n$$

mit den $4n$-2 Bedingungen

$$\begin{aligned}
s_j(x_{j-1}) &= y_{j-i}, && j = 1, \ldots, n \\
s_j(x_j) &= y_j, && j = 1, \ldots, n \\
s'_j(x_j) &= s'_{j+1}(x_j), && j = 1, \ldots, n-1 \\
s''_j(x_j) &= s''_{j+1}(x_j), && j = 1, \ldots, n-1
\end{aligned}$$

zwei weitere Bedingungen durch Verhalten am Rand

$4n$ Parameter und $4n$ Bedingungen ergeben ein lösbares Gleichungssystem !

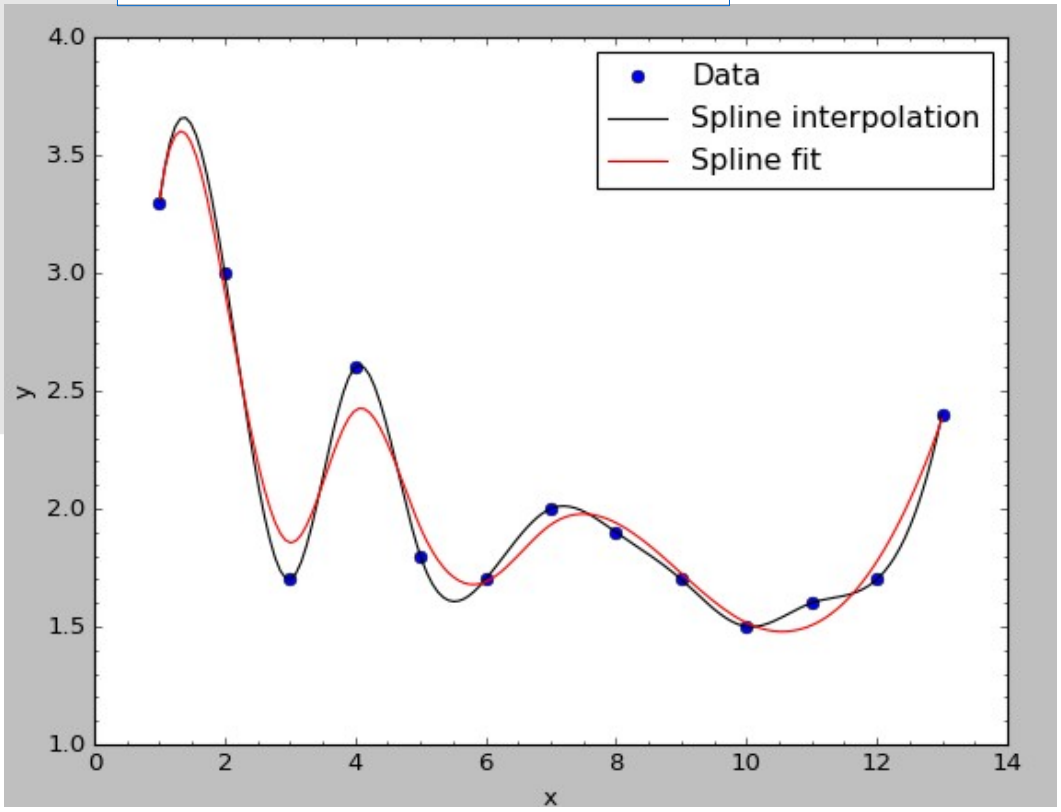
Die effiziente numerische Behandlung für große n ist eine andere Frage …

Spline-Interpolation in python

```
import numpy as np, matplotlib.pyplot as plt
from scipy import interpolate

# set some data
x=np.arange(13)+1
y = np.array([3.3,3.0,1.7,2.6,1.8,1.7,2.0,
    1.9,1.7,1.5,1.6,1.7,2.4])
xplt = np.linspace(1., 13., 150)

# perform spline interpolation and fit
csi_y=interpolate.UnivariateSpline(x,y,s=0.)
csf_y=interpolate.UnivariateSpline(x,y,s=.1)
# plot results
plt.plot (x, y, 'bo', label='Data')
plt.plot (xplt, csi_y(xplt), 'k-',
          label='Spline interpolation')
plt.plot (xplt, csf_y(xplt), 'r-',
          label='Spline fit')
plt.xlabel('x')
plt.ylabel('y')
plt.legend(numpoints=1, loc='best')
plt.minorticks_on()
plt.show()
```

Das python-Paket `scipy.interpolate` enthält die Klasse `UnivariateSpline`

Script `UnivariateSpline.py`



Kubische Splines sind analytisch differenzier- und integrierbar !

Beispiel: Analyse einer Hysterese-Kurve

Abb. a)

geglättete Spannungsverläufe
- proportional zum Strom
- proportional zum B-Feld

an einer Spule mit Eisenkern, Punkte verbunden durch interpolierende kubische Spline-Funktionen

Abb. b)

Kanal a gegen b aufgetragen (Punkte und Splines)

Abb. c)

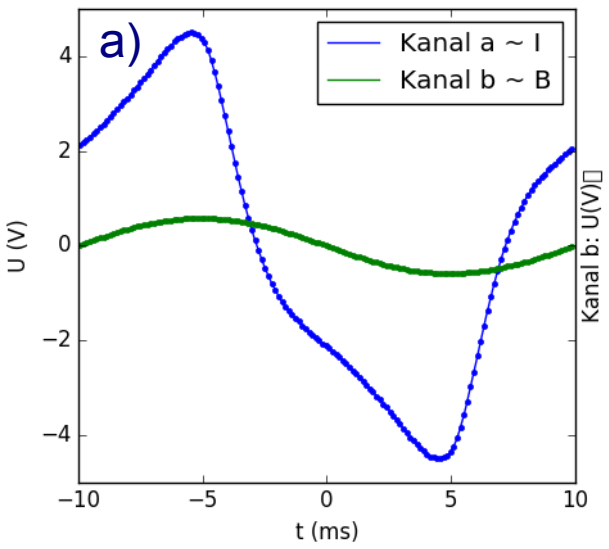
Auftrennung nach Punkten mit steigendem bzw. fallendem Strom,

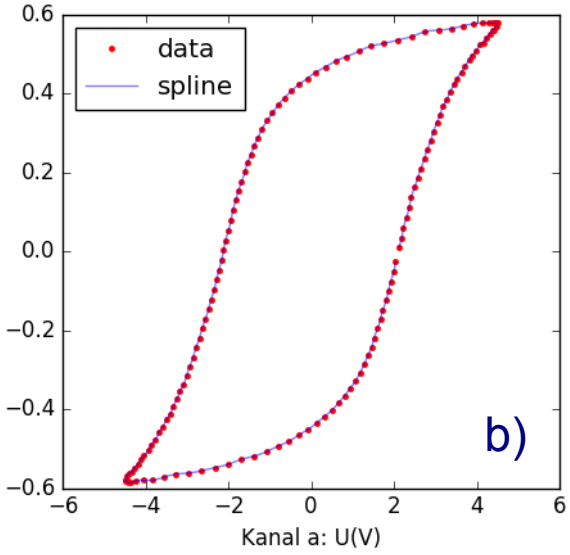
Spline-Anpassung (nicht-interpolierend) an die beiden Kurvenverläufe

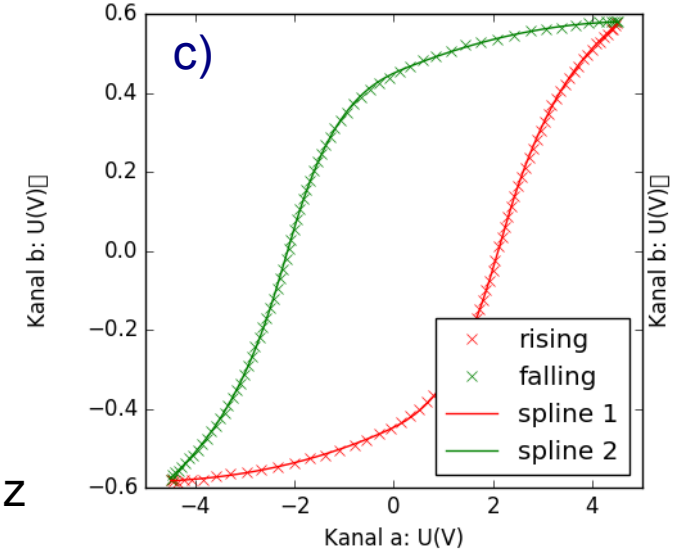
Abb. d)

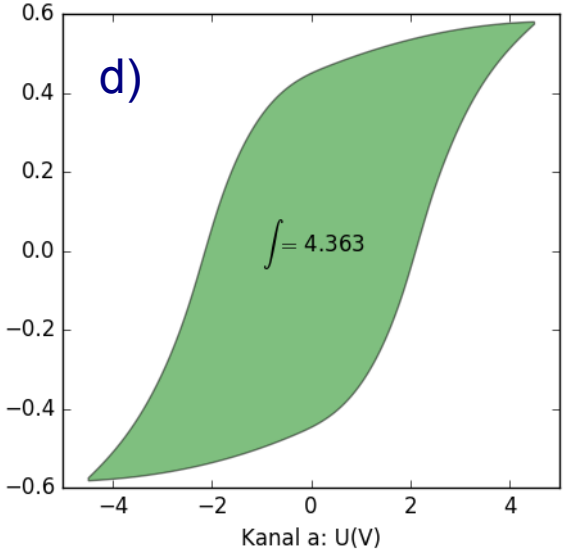
Integral der eingeschlossenen Fläche durch Differenz der Integrale über die beiden Spline-Funktionen

Script: Beispiel_Hysterese.py







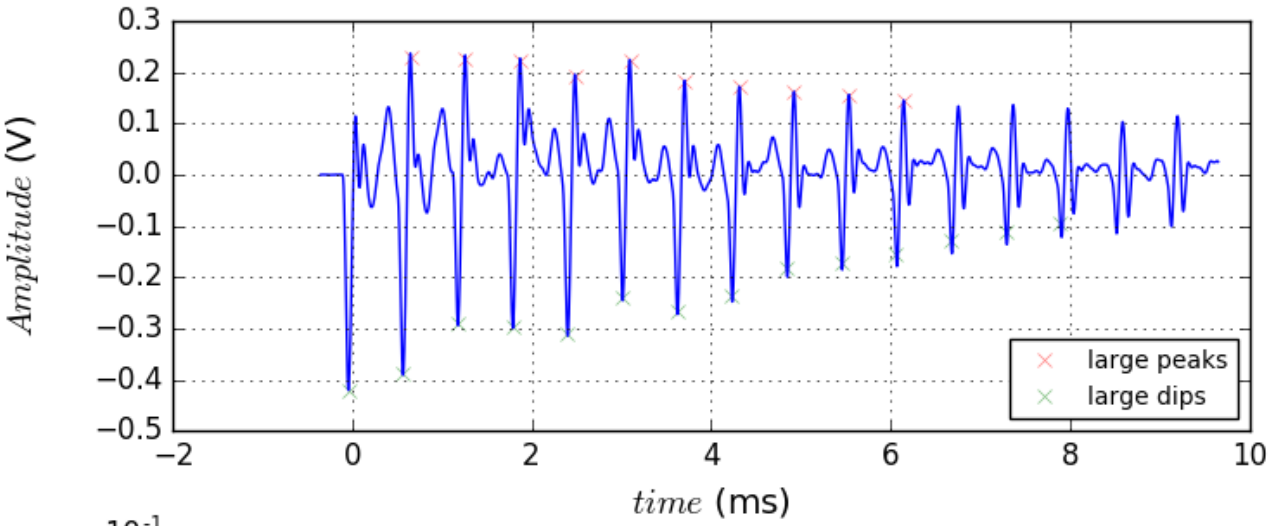


Digitale Abtastung: Frequenzanalyse

Amplitudenverlauf

$$a_i = \text{Amplitude}(t_i)$$

(„time domain“)



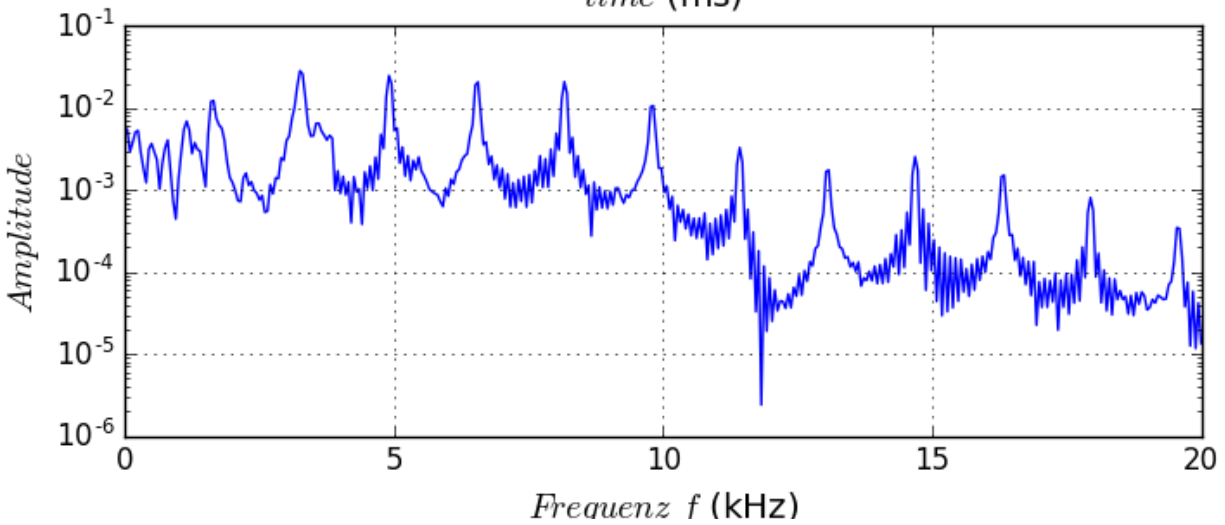
Fourier-Transformation:

$$s_k = \frac{2}{n} \sum_i a_i \sin\left(\frac{2\pi}{f_k} t_i\right)$$

$$c_k = \frac{2}{n} \sum_i a_i \cos\left(\frac{2\pi}{f_k} t_i\right)$$

$$A_k = \sqrt{{s_k}^2 + {c_k}^2}$$

(„frequency domain“)

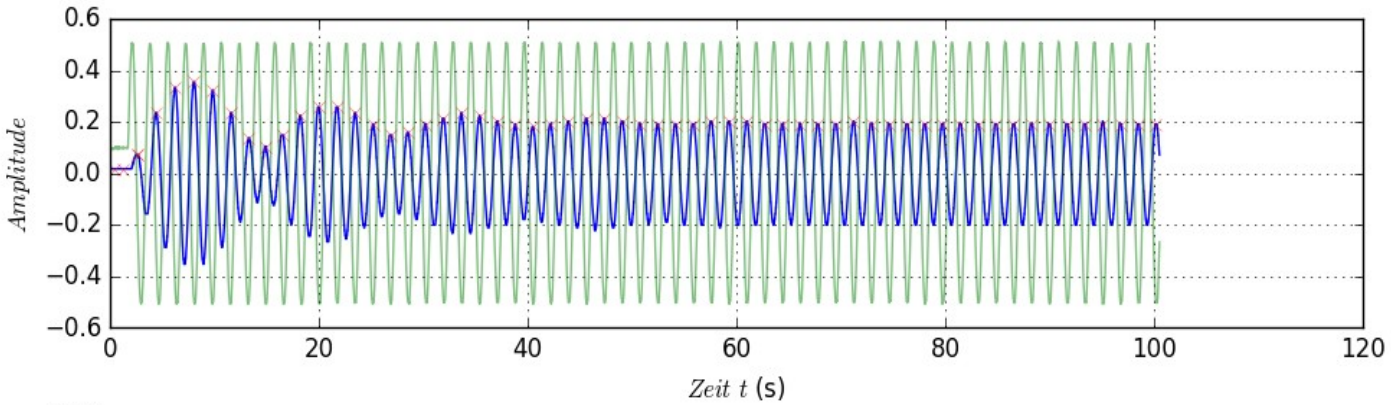


numerisch effizienteres Verfahren: **Fast Fourier Transform, FFT**

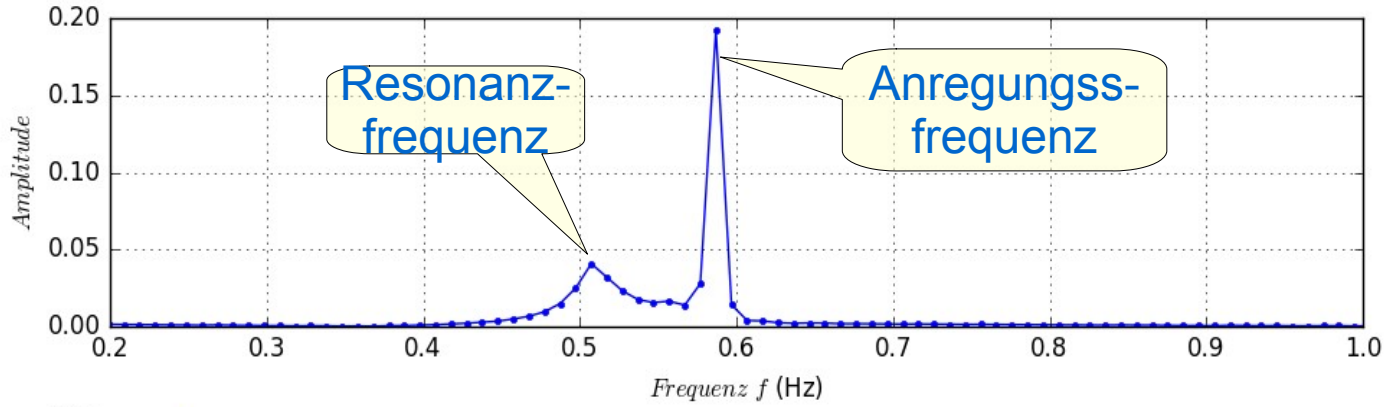
Beispiel: Fourieranalyse

Oberhalb der Resonanzfrequenz angeregtes, gedämpftes Drehpendel:

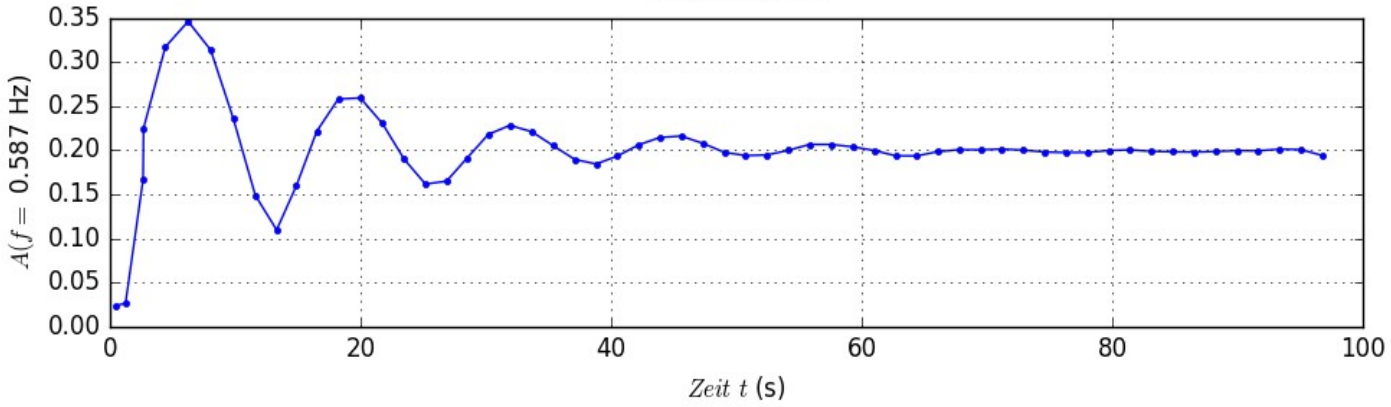
Amplituden-verlauf



Frequenz-spektrum



Zeitlicher Verlauf der Amplitude der Anregungsfrequenz (über je zwei volle Perioden gemittelt)



Toolbox „PhyPraKit“

python-Modul PhyPraKit.py

1. Dateneingabe aus Text-Dateien
   - readColumnData() read data and meta-data from text file
   - readPicoScope() read data from PicoScope
   - readCassy() read CASSY output file in .txt format
   - labxParser() read CASSY output file, .labx format
2. signal processing:
   - offsetFilter subtract an offset in array a
   - meanFilter apply sliding average to smoothen data
   - resample average over n samples
   - Fourier_fft fast Fourier transformation of an array
   - FourierSpectrum Fourier transformation of an array ``(slow, preferably use fft version)``
   - simplePeakfinder find peaks and dips in an array comparing neighbouring samples (``use convolutionPeakfinder``)
   - convolutionPeakfinder find peaks and dips in an array
3. Statistics
4. Histogramm-Tools
5. Funktionsanpassung
6. Erzeugung von simulierten Datensätzen

s. Beispiel-Scripts
im Verz. examples

http://www.ekp.kit.edu/~quast/PhyPraKit/

Zusammenfassung: abgetastete Daten

- Glätten durch gleitenden Mittelwert oder andere Filter
- Datenreduktion
  - durch Mittelwertbildung über *n* Datenpunkte
  - Selektion von Untermengen, d. h.
    nur jeden *n*-ten Punkt verwenden
- Symmetrisieren der Daten, d.h. Offset-Korrektur:
  $y_i \rightarrow y_i$ **Mittelwert**(y_i)
- Extraktion bestimmter Eigenschaften:
  - Minima bzw. Maxima
  - Sprünge bzw. Stellen maximaler Steigung
    Faltung mit geeigneter Filterfunktion
- Frequenzbestimmung bei periodischen Signalen
  Autokorrelation
- Interpolation
  kubische Splines
- Frequenzanalyse:
  Fourier-Transformation

s. Funktionen in PhyPraKit
und die Beispiele
- `test_convolutionPeakFinder()`
- `test_AutoCorrelation.py`
- `test_Fourier.py`
- `UnivariteSpline.py`

u. a.