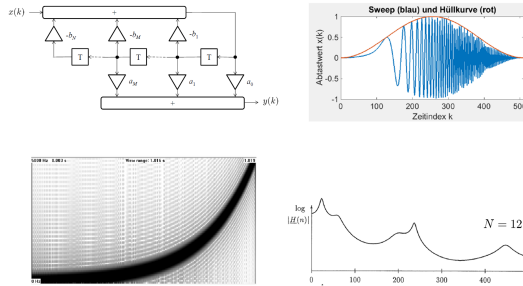


0.1 - Installationsanleitung

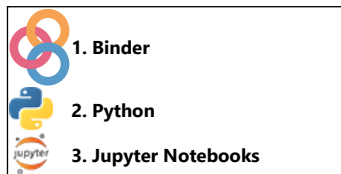
Begleitend zur Übung des Fachs Signalverarbeitung wird Ihnen eine Reihe an Jupyter-Notebooks zur Verfügung gestellt. Diese soll Ihnen sowohl beim Verständnis der in der Vorlesung und Übung behandelten Themen helfen, als auch die Fähigkeit zur selbstständigen Lösung eigener Problemstellungen geben.

Die Bearbeitung der Notebooks ist dabei komplett freiwillig. Falls Sie aber Probleme, Fragen und Anmerkungen haben, posten Sie diese gerne ins Opal-Forum oder senden Sie uns eine Mail.



Dieses Notebook soll Ihnen beim Einstieg mit Python und der Installation von Modulen helfen. Weitere Notebooks werden zeitnah zur jeweiligen Übung online gestellt.

Inhalt



1. Binder

Sie schauen sich dieses Notebook sicherlich zuerst über den Service [Binder](#) an. Dieses Tool erlaubt es, auf Github hochgestellte Jupyter-Notebooks -wie zum Beispiel dieses- über einen externen Server auszuführen. Das gibt einem die Möglichkeit, sich Projekte ohne Download der Daten und ohne lokal installiertes Python anzuschauen. Gestartete Sessions werden aber nach ungefähr 10 Minuten Inaktivität vom Server geschlossen und alle Änderungen gehen damit verloren. [Hier gibt es weitere Informationen dazu.](#)

Falls Sie also die von Ihnen bearbeiteten Notebooks gerne wiederverwenden möchten, ist eine Installation von Python notwendig. Eine Anleitung zur Installation unter Windows gibt es in den folgenden Kapiteln.



2. Python

Allgemeines

Python ist eine Skriptsprache auf hoher Ebene, die Interpretation, Kompilierung, Interaktivität und Objektorientiertheit kombiniert. Python ist leicht zu erlernen, einfach zu lesen, und hat eine breite Palette von Standardbibliotheken. Einer der größten Vorteile von Python ist seine umfangreiche

Bibliothek, die plattformübergreifend und mit UNIX, Windows und Macintosh kompatibel ist. Es gibt viele Tutorials über Python wie zum Beispiel auf [w3schools](http://w3schools.com).

Falls Sie Python auf Ihren Rechner installieren wollen, gibt es im Folgenden eine Anleitung für Windows-Nutzer*innen, da die Installation und Nutzung von Python zum Teil sehr Kommandozeilen-intensiv sind.

Hinweis: Es gibt zudem auch die Software [Anaconda](#), die die Nutzung von Python und Jupyter-Notebooks vereinfacht. Bei Anaconda sind viele weitere externe Module schon vorinstalliert. Wir wollen Ihnen aber im Folgenden eine Anleitung geben, wie sie mit dem normalen Python umgehen. Falls Sie also Probleme bei der direkten Installation von Python haben, können Sie als Option auf die [Anaconda-Software zurückgreifen](#). Sie können sich aber zuerst auch gerne an uns wenden.

Installations- und Nutzungsanleitung von Python

Nun wird Ihnen Schritt für Schritt erklärt, wie Sie Python installieren können. Zuerst sollten Sie auf Ihren PC nachschauen, ob Sie schon eine Pythonversion installiert haben.

1. Kontrolle auf installierte Python-Versionen

Öffnen Sie eine Eingabeaufforderung (cmd) und geben Sie den Befehl `py -0p` ein. Mit diesem Kommando wird -wenn verfügbar- der Python Launcher `py` ausgeführt, der nach ausführbaren Pythoninstallationen (sogenannten executables) sucht und diese nach ihren Versionen auflistet. Wenn dieser Befehl nicht gefunden wurde, haben Sie mit hoher Wahrscheinlichkeit kein Python installiert. Falls doch, schauen Sie über den angezeigten Pfad nach, ob es sich um ein von einem anderen Programm intern installiertes Python handelt. Falls dies der Fall ist, sollten Sie es lieber nicht verändern und ein neues Python installieren. Wenn Sie ein veraltetes Python installiert haben, können Sie die Zeit jetzt nutzen und eine neue Version installieren und Ihre alte Version entweder löschen oder behalten. Sie können nämlich unterschiedliche Python-Versionen nebeneinander installiert haben

2. Installation von Python:

Die neuesten Versionen von Python [finden Sie hier](#). Wählen Sie bei der Installation auch die Option "Add Python 3.X to PATH" aus! Diese Funktion fügt den Speicherort von "python.exe" (..\Python3X) und den der externen Module (..\Python3X\Scripts) in die Umgebungsvariable "PATH", damit bei Funktionsaufrufen über die Eingabeaufforderung auch diese Ordner durchsucht werden.

Hinweis: Bei der Erstellung dieses Notebooks ist die Version 3.9.0 kürzlich veröffentlicht worden. Deshalb waren einige externe Module bis zur Fertigstellung des Notebooks immer noch nicht aktualisiert worden und konnten nicht über den normalen Weg für Python 3.9 installiert werden. Probieren Sie es aber zuerst aus, die neueste Python-Version zu installieren. Falls Sie jetzt immer noch Probleme haben sollten, lassen Sie Python 3.9 einfach installiert und laden sich die Version 3.8 herunter, womit Sie zunächst arbeiten können. Die Probleme mit den inkompatiblen Modulen für Python 3.9 sollten nur eine Frage der Zeit sein, bis diese behoben sind. Zudem ist das zweimalige Installieren von Python eine gute Übung! ;)

3. Starten von Python:

Python können Sie über die Eingabeaufforderung von Windows starten. Zum Öffnen der Eingabeaufforderung drücken Sie dafür zum Beispiel **Windows-Taste + R**, geben Sie "`cmd`" ein und drücken **Enter**. In der sich nun geöffneten Kommando-Shell können Sie Python über zwei unterschiedlichen Kommandos starten:

- Bei dem Befehl `python` sucht die Shell alle im "PATH"-Verzeichnis eingetragenen Ordner nach einer executable (.exe) mit dem Namen python (also python.exe). Dabei werden die Verzeichnisse systematisch von oben nach unten durchgegangen und beim ersten Treffer hört die Suche auf und die .exe wird ausgeführt. Falls Sie nur eine Version installiert haben, sind Sie damit vollkommen zufrieden. Falls Sie aber mehrere Python-Versionen installiert haben, würde dies bedeuten, dass Sie immer die Verzeichnisse (..\Python3X) und (..\Python3X\Scripts) der gewünschten Python-Version nach oben schieben müssen.

- Mit dem Befehl `py` wird der "Python Launcher" aufgerufen, der bei der Installation von Python in das Verzeichnis C:\Windows\ installiert wird. Wenn dieser ohne weitere Bedingungen von Ihnen gestartet wird, läuft der Befehl ab wie der Befehl `python`. Mit dem "Python Launcher" kann aber bestimmt werden, welche Python-Version gestartet werden soll. Um zum Beispiel Python mit Version 3.8 zu starten, hängt man einfach ein `-3.8` an. Also als komplettes Kommando: `py -3.8`. (Wenn Sie mehr Interesse an einzelnen Funktionen haben, können Sie immer ein `-h` hinter jede Funktion schreiben. Z.B.: `py -h` oder `python -h`.)

Ein Python-Kernel startet und Ihnen sollte die Versionsnummer von dem von Ihnen installierten Python angezeigt werden. Falls dies nicht der Fall ist, müssen Sie gegebenenfalls den "PATH" in den Umgebungsvariablen editieren. Eine Anleitung dazu [finden Sie hier](#).

Sie können jetzt Python verwenden, indem Sie Ihren Code zeilenweise in die Eingabeaufforderung eingeben. Probieren Sie zum Beispiel folgenden Code aus:

```
2 + 3 [Enter]

a = 6 [Enter]

b = 12 [Enter]

c = a + b [Enter]

c [Enter] #Kommentar: Ausgabe von c

A = [[1,2,3], [4,5,6]] [Enter] #Kommentar: Erstellen einer 2x3-Matrix

A [Enter] #Kommentar: Ausgabe von A, equivalent zu print(A)
```

Zum Schließen des Kernels können Sie entweder `exit()` oder `Strg + Z` eingeben und mit `Enter` bestätigen.

Nutzung von Modulen

Es gibt built-in und externe Module (auch Packages oder Libraries genannt, hier aber ein [Link über die genaue Wortdefinition](#)).

Built-in Module können direkt verwendet werden und stellen die Basis zur Programmierung in Python dar. Zum Beispiel gibt es das Modul `cmath` für [komplexe Zahlen](#). Daraus können Sie das Objekt `complex()` direkt nutzen, um eine komplexe Zahl zu erzeugen. Testen Sie den folgenden Code hier in der ausführbaren Zelle aus oder in Ihrer Eingabeaufforderung:

```
# Starten Sie den Code, indem sie die Zelle auswählen und den Run-Button drücken
# oder mit Shift + Enter

c = complex(1.4, -3) # Caste die Variable c als komplexe Zahl mit dem Wert 1.4 -3j
print(c)             # Ausgabe des Inhalts von c
```

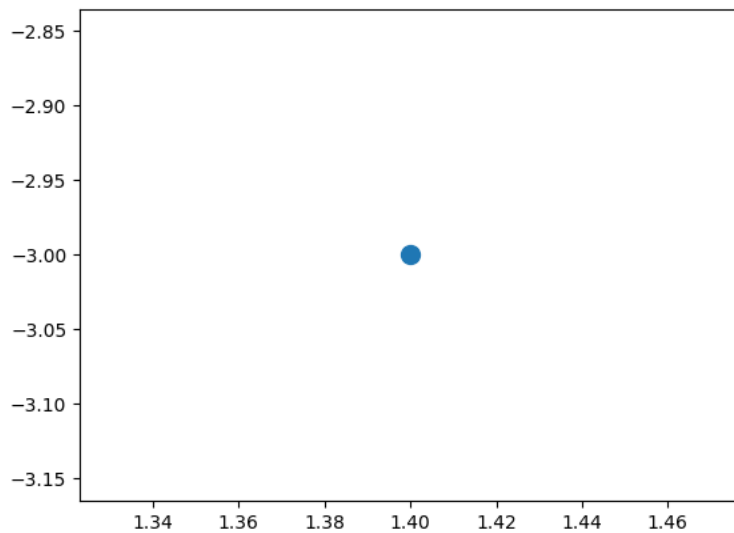
```
(1.4-3j)
```

Die graphische Darstellung des Werts in einem Plot ist mit den internen Modulen aber zum Beispiel nicht mehr möglich. Dafür muss man sich mit einem externen Modul behelfen, zum Beispiel `matplotlib`. Diese externen Module müssen in Python installiert werden und für jedes Projekt importiert werden. Im Folgenden sind zwei Code-Zellen als Beispiel für installierte und nicht installierte Pakete im `Binder`-Projekt dargestellt.

Zuerst ein installiertes Modul zur Erzeugung von Plots:

```
# Starten Sie den Code, indem sie die Zelle auswählen und den Run-Button drücken
# oder mit Shift + Enter
import matplotlib.pyplot as plt

plt.scatter(c.real,c.imag, s=100) # Ausgabe eines Punkts
plt.show()                       # Plot zeigen
```



Hinweis: Falls Sie `matplotlib` schon lokal installiert haben und diesen über die Eingabeaufforderung nutzen, werden die Kommandos ausgeführt, nur wegen fehlender Ausgabemöglichkeit seitens der Eingabeaufforderung kann Ihnen der Plot nicht angezeigt werden.

Als Beispiel für einen nicht ausführbaren Code in dem `Binder`-Projekt ist die folgende Zelle mit dem Modul `pandas`, das nicht mitinstalliert wurde (und im Weiteren auch nicht benötigt wird).

```
# Starten Sie den Code, indem sie die Zelle auswählen und den Run-Button drücken
# oder mit Shift + Enter
import pandas as pd

s = pd.Series([1, 3, 5, np.nan, 6, 8])
# Es sollte ein ModuleNotFoundError entstehen.
```

```
-----
NameError                                Traceback (most recent call last)
Cell In [3], line 5
      1 # Starten Sie den Code, indem sie die Zelle auswählen und den Run-Button drücken
      2 # oder mit Shift + Enter
      3 import pandas as pd
----> 5 s = pd.Series([1, 3, 5, np.nan, 6, 8])

NameError: name 'np' is not defined
```

Für das Einbinden von externen Modulen gibt [es folgende Kommandos](#):

A) Import der kompletten Bibliothek

```
import numpy
```

- Der Befehl importiert die gesamte Bibliothek des Moduls mit dem eigenem Namensraum `numpy`. Das Modul `numpy` besitzt zum Beispiel die Objekte `cos()`, `sin()` oder `pi`. Durch den obigen Import lassen sich die Objekte aufrufen, indem man zuerst das Modul nennt: `numpy.cos()`, `numpy.sin()` oder `numpy.pi`.

B) Import der kompletten Bibliothek mit alias

```
import numpy as np
```

- Wie bei A) importiert der Befehl die gesamte Bibliothek des Moduls, nur heißt der Namensraum nun `np`. Die Objekte `cos()`, `sin()` oder `pi` lassen sich nun folgendermaßen aufrufen: `np.cos()`, `np.sin()` oder `np.pi`.

C) Import einzelner Module aus der Bibliothek

```
from numpy import cos, sin, pi
```

- Mit diesem Befehl werden die einzelnen Objekte `cos()`, `sin()` und `pi` in den globalen Namensraum importiert. Diese können nun direkt aufgerufen werden, also mittels: `cos()`, `sin()` oder `pi`. Passen Sie dabei auf, dass der Name nicht schon belegt ist und es zur Kollision von gleichnamigen Objekten gibt, die dann nicht mehr aufrufbar sind.

D) Import aller Module aus der Bibliothek

```
from numpy import *
```

- Der Befehl bindet alle Objekte und Module von `numpy` direkt in den globalen Namensraum ein. Nun können alle Module direkt aufgerufen werden, also mittels: `cos()`, `sin()` oder `pi`. Dieser Import ist **nicht empfehlenswert**, da so die große Gefahr der Namenskollision besteht, und sollte daher nicht genutzt werden.

E) Import einzelner Module aus der Bibliothek mit Alias

```
import numpy.cos as cosin  
oder  
from numpy import cos as cosin
```

- Bei dieser Art des Imports wird nur jeweils das aufgerufene Modul der Bibliothek mit dem vorgegebenen Alias eingebunden.

Durch das Importieren von externen Modulen kann man also die Funktionalität von Python erweitern. Jedes externe Modul sollte aber nur dann in das Projekt eingebunden werden, wenn dieses auch wirklich verwendet wird. Externe Module können dabei auch in anderen Programmiersprachen geschrieben sein. `numpy` zum Beispiel ist vorwiegend in C geschrieben.

Die in dem folgenden Einführungs-Notebook (0.2) verwendeten Module und deren Funktionen sind:

- `numpy`: für [Arrays und Matrizen](#),
- `matplotlib`: Erstellung von mathematischen Darstellungen,
- `scipy`: Nutzung wissenschaftlicher Funktionen.

Installation von externen Modulen

Für die Installation von externen Modulen steht das Paketverwaltungsprogramm `pip` zur Verfügung. Es lädt die Module aus dem [Python Package Index \(PyPI\)](#) herunter, in dem die wichtigsten Projekte gelistet sind, und installiert diese. `pip` kümmert sich auch darum, bei Abhängigkeiten zu fehlenden Modulen diese automatisch mit zu installieren. Das Paketverwaltungsprogramm nutzt man direkt über die Eingabeaufforderung von Windows. Dabei ist zu beachten, dass Python nicht in derselben Eingabeaufforderung gestartet ist. Alle laufenden Python-Ausführungen können in anderen Eingabeaufforderungen weiterlaufen.

Hinweis: Falls Sie wegen mehreren installierten Python-Version auf den Python-Launcher zurückgreifen, müssen Sie bei allen im Folgenden beschriebenen Befehlen das `python` mit `py -3.X` ersetzen.

Mit `python -m pip -v` können Sie sich den Speicherort der `pip`-Version ausgeben lassen. Kontrollieren Sie, dass der richtige Pfad angezeigt wird, indem Sie die Python-Version installiert haben. Falls dies nicht der Fall ist, müssen Sie die Umgebungsvariablen editieren oder auf den Python Launcher `py` zurückgreifen (siehe dafür vorheriges Unterkapitel: Installation von Python).

Über den Befehl `python -m pip list` oder in einem Pythonkernel über `help("modules")` kann man sich eine Liste der installierten Module ausgeben lassen.

Bevor Sie Module installieren, sollte pip geupdated werden. Dies kann mit folgendem Befehl getan werden.

```
python -m pip install --upgrade pip
```

Zudem ist es sinnvoll, daraufhin das Modul [wheel](#) zu installieren. Mit diesem Modul kann pip vorgepackte und -kompilierte Packages von PyPi herunterladen, was den Download und die Installation beschleunigt (siehe [Link für mehr Erklärung](#)). Installiert wird wheel über folgenden Befehl:

```
python -m pip install wheel
```

1. Installation von Numpy:

Da [numpy](#) primär in C geschrieben ist, werden Libraries und Interpreter auf dem PC benötigt, um numpy zu installieren. Probieren Sie aber zuerst, die Installation mit folgendem Code zu starten. Öffnen Sie eine Eingabeaufforderung und geben Sie folgenden Code ein und drücken **Enter**:

```
python -m pip install numpy
```

Falls dies mit einem Fehler abbricht, liegt das wahrscheinlich daran, dass Sie die C-Libraries installieren müssen. Diese können Sie mit [C++-Buildtools](#) installieren. Wählen Sie nach Starten des Installationsassistenten im Installationsmenü C++-Buildtools aus. Hinweis: entfernen Sie keine Häkchen bei den vorausgewählten optionalen Modulen. Und ja, leider sind das um die 5 Gigabyte an Speicherplatz..

Danach geben Sie nochmal den obigen Code ein, um die Installation ein weiteres Mal zu starten.

1. Installation von matplotlib:

Wenn numpy und wheel installiert sind, kann die Installation von [matplotlib](#) über folgenden Befehl gestartet werden:

```
python -m pip install matplotlib
```

Wenn Sie sich nun die Liste der installierten externen Module anschauen (`pip list`), wird Ihnen auffallen, dass durch die Installation von matplotlib noch weitere Module mit installiert wurden. Die sind automatisch mitinstalliert worden, da matplotlib sich auf diese bezieht.

1. Installation von scipy:

Scipy auf einem Windows-Betriebssystem über den regulären Weg zu installieren [ist sehr aufwendig](#). Um diesen Aufwand zu umgehen, wird über den Dienst wheel eine vorgepackte Version heruntergeladen und installiert. Dies wird über folgenden Befehl gestartet:

```
python -m pip install scipy
```

Falls dies mit Fehlern endet, kann dies an folgenden Problemen liegen: - das Package [mkl](#) muss vorinstalliert werden.

```
python -m pip install mkl
```

Starten Sie danach nochmal die Installation von `scipy`. - Falls Sie die Pythonversion 3.9 installiert haben, kann das Wheel gegebenenfalls noch nicht zur Verfügung stehen. Dafür stehen Ihnen zwei Möglichkeiten zur Verfügung. Zum einen können Sie die Python-Version 3.8. installieren und darüber nochmal alle externen Module installieren, was wir Ihnen empfehlen.

Zum anderen gibt es die Möglichkeit, sich kompilierte Binary-Dateien von anderen Anbietern zu downloaden und installieren. Eine Webseite mit solchen Binary-Dateien ist über [\[diesen Link zu finden\]](#) (<https://www.lfd.uci.edu/~gohlke/pythonlibs/>). Um nun `scipy` darüber zu installieren, müssen Sie zunächst `numpy` und `mkl` deinstallieren und ein vorgepacktes .whl für `numpy + mkl` und eines für `scipy` herunterladen. Dazu müssen Sie aus den Listen die korrekten binary-Files aussuchen. [Hier die Liste für scipy mit einem Link zu numpy + mkl](<https://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy>). Die Deinstallation der schon installierten Module und die Installation der lokalen `whl` -Dateien folgt über folgende Kommandos. Dabei müssen Sie den Pfad und den Dateinamen noch anpassen.

```
python -m pip uninstall numpy
```

```
python -m pip uninstall mkl
```

```
python -m pip install C:\Users\~User\Downloads\numpy-1.19.2+mk1-cp3X-cp3X-win_amd64.whl
python -m pip install C:\Users\~User\Downloads\scipy-1.5.X-cp3X-cp3X-win_amdXX.whl
```

Diese Möglichkeit sollte aber immer nur als "letzte Option" gesehen werden, da es experimentelle Binaries sind, und wird Ihnen in diesem Fall auch abgeraten. Falls Sie also (noch) nicht normal `scipy` installieren können, dann nutzen Sie lieber Python v3.8. Lassen Sie Python 3.9 aber einfach installiert und probieren Sie die Installation einfach ein paar Tage später aus. Vielleicht ist bis dahin ja ein Wheel erstellt worden.

Eine Installation von mehreren Packages kann man auch mit nur einem `pip install`-Kommando durchführt, indem man die externen Pakete nacheinander schreibt. Zum Beispiel über: `python -m pip install wheel numpy matplotlib scipy`.

Nun sollten alle externen Module - bis auf Jupyter Notebook - auf Ihrem Computer installiert sein, die Sie für das Einführungs-Notebook (0.2) benötigen. Falls Sie weitere Informationen benötigen, welche Möglichkeiten Ihnen bei der Installation und Update mit pip zur Verfügung stehen, können Sie sich noch folgende Links anschauen [\(1\)](#) [\(2\)](#) [\(3\)](#).



3. Jupyter Notebook

Jupyter Notebook ist eine interaktive Computerumgebung, mit der Benutzer Notizbuchdokumente erstellen können, die Folgendes umfassen: - Live-Code - Interaktive Widgets - Diagramme - Erzähltext - Gleichungen - Bilder - Video usw.

Über die Kernel- und Messaging-Architektur von Jupyter ermöglicht das Notebook die Ausführung von Code in verschiedenen Programmiersprachen. Für jedes Notizbuchdokument (auch Notebook), das ein Benutzer öffnet, startet die Webanwendung einen Kernel, der den Code für dieses Notizbuch ausführt. Hier benutzen wir Python3 als Kernel.

Folgende Informationen zum Umgang sind wichtig zu wissen:

- Der Aufbau ist in Zellen. Dabei kann eine Zelle zum Beispiel Pythoncode (**Code**) oder Text/Bilder (**Markdown**) beinhalten.
- Zellen mit Code können ausgeführt werden, indem man auf den Run-button klickt oder auf **Umschalt + Enter**.
- Auch Zellen mit Text können (wenn sie im Bearbeitungsmodus sind) mittels Run-Button oder **Umschalt + Enter** ausgeführt werden.
- Die Ergebnisse aller ausgeführten Zellen bleiben dabei gespeichert und können in der nächsten Zelle wieder verwendet werden.
- Löschen der kompletten Ausgabe geht über das Menüband unter **Kernel -> Restart & Clear All Output**.

Weitere Hinweise zu dessen Verwendung finden Sie [hier](#).

Installation von Jupyter Notebook

Zur Installation von [Jupyter Notebook](#) wird wieder `pip` verwendet. Nutzen Sie dafür folgendes Kommando:

```
python -m pip install jupyter
-- beziehungsweise --
py -3.X -m pip install jupyter
```

Starten können Sie **Jupyter Notebook** genau wie `python` oder `pip` über die Eingabeaufforderung von Windows mittels folgendem Kommando:

```
jupyter notebook
```

Daraufhin startet in der Eingabeaufforderung **Jupyter Notebook** und Ihr Standardbrowser öffnet sich mit dem Notebook-Fenster. Sie können das Programm auch mit einem anderen Browser verwenden, indem Sie den in der Eingabeaufforderung mit ausgegebenen http-Link nutzen. Wenn Sie nun im Browser ein oder mehrere Notebooks (.ipynb) öffnen, wird in der Eingabeaufforderung für jedes Notebook ein neuer Python-Kernel gestartet.

Hilfe bei mehreren Python-Versionen

Wenn Sie mehrere Python-Versionen installiert haben, können auch mehrere Python-Versionen den Kernel für die Notebooks stellen. Das Problem daran ist, dass der Kernel immer gleichnamig mit **python3** betitelt wird. Das sorgt dafür, dass es keine wirkliche Kontrolle gibt, welche Version gerade für den Kernel genutzt wird. Es wird in dieser Konfiguration meistens die Python-Version genommen, die im "PATH" zuerst kommt. Um eine Kontrolle über die Python-Kernel-Version einzurichten, können Sie folgende Befehle verwenden.

Zeigen Sie sich alle Kernelspezifikationen an über:

```
jupyter kernelspec list
```

Diese Liste muss dabei nicht stimmen, da Namensdopplungen nicht angezeigt werden!
Installieren Sie weitere Kernelspezifikationen mit anderem Namen über folgenden Befehl (hier für Python 3.8):

```
py -3.8 -m ipykernel install --name python_3.8 --display-name "Python 3.8"
```

Wiederholen Sie das für alle Python-Versionen, die mit Jupyter Notebook verwendet werden sollen.
Sie können die nun redundant gewordene Spezifikation **python3** über folgenden Befehl löschen:

```
jupyter kernelspec uninstall python3
```

Wenn Sie dann ein Notebook geöffnet haben, können Sie über das Menüband unter **Kernel** -> **Change Kernel** die Python-Version auswählen, mit der der Kernel laufen soll. Auch können Sie bei einer Erstellung eines neuen Notebooks direkt auswählen, mit welchem Kernel dieser bei der Erstellung gestartet wird. Für die weitere Nutzung ist diese Auswahl aber irrelevant, da der Python-Code zwischen den Versionen zum großen Teil kompatibel ist.

Falls Sie jetzt erfolgreich Python mit den externen Packages installieren konnten und Jupyter Notebook lokal nutzen können, sind Sie gewappnet für die nächsten Notebooks! Herzlichen Glückwunsch und viel Spaß mit den neu gewonnen Möglichkeiten durch ein frisch installiertes Python.

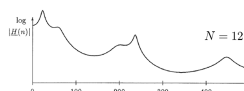
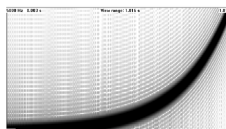
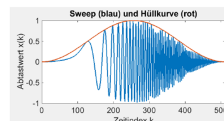
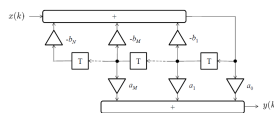
Melden Sie sich gerne, wenn Sie Anmerkungen zu dieser Anleitung haben.

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)

0.2 - Einführung

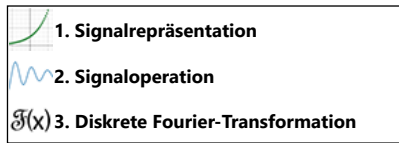
Begleitend zur Übung des Fachs Signalverarbeitung wird Ihnen eine Reihe an Jupyter-Notebooks zur Verfügung gestellt. Diese soll Ihnen sowohl zum besseren Verständnis der in der Vorlesung und Übung behandelten Themen helfen, als auch die Fähigkeit zur selbstständigen Lösung eigener Problemstellungen geben.

Die Bearbeitung der Notebooks ist dabei komplett freiwillig. Falls Sie aber Probleme und Fragen haben, posten Sie Ihre Fragen gerne ins Opal-Forum.



Dieses Notebook soll Ihnen beim Einstieg in die Programmiersprache Python helfen. Weitere Notebooks werden zeitnah zur jeweiligen Übung online gestellt.

Inhalt



1. Signalrepräsentation

Zunächst schauen wir uns an, wie man mit Python typische Eingangssignale erzeugt und sich diese graphisch darstellen lassen kann. Dafür benötigen wir aus dem Modul `numpy` Funktionen zur Erzeugung von Arrays. Zur Visualisierung verwenden wir aus dem Modul `matplotlib` das Objekt `pyplot`.

Die Module und Objekte lassen sich wie folgt importieren:

```
import numpy as np
import matplotlib.pyplot as plt
```

Probieren Sie es in der nächsten Zelle gleich mal aus, indem sie die mit `[.]` gekennzeichneten Zeilen ersetzen!

```
# Lösung:
# Hier können Sie Python-Code hineinschreiben und mit Shift+Enter ausführen!
# Module importieren (numpy und matplotlib.pyplot)
import numpy as np
import matplotlib.pyplot as plt

# Erzeugung eines linearen Arrays als Zeitbereich
t = np.linspace(-5, 5, 1001)
```

Das Objekt `np.linspace(a,z,res)` erzeugt ein äquidistantes Zahlenarray zwischen der unteren Schranke `a` und der oberen Schranke `z` mit der Auflösung von `res`. Den Inhalt von Variablen lässt sich anzeigen, indem man das `print()`-Kommando verwendet oder die Variable alleine ans Ende der Zelle in eine Zeile schreibt.

Lassen Sie sich also nun in der folgenden Zelle das Array ausgeben! (Da die Auflösung sehr groß gesetzt ist, ist auch das Array dementsprechend groß und wird nicht vollständig angezeigt. Deshalb kann man sich auch nur einen Ausschnitt des Arrays mit z.B. `t[0:9]` anzeigen lassen.)

```
# Lösung
# Betrachten des Inhalts der Variable 't'.
t[0:9]
```

```
array([-5.    , -4.99, -4.98, -4.97, -4.96, -4.95, -4.94, -4.93, -4.92])
```

Es lassen sich noch weitere Eigenschaften der Variablen und des Inhalts anzeigen:

```
# Betrachtung der Eigenschaften:
t_length = len(t)
t_type = type(t)
t_entry_type = type(t[0])

# Setzen Sie in der print()-Eingabe anstelle [...] ein.

print("Array-Länge: {}; Arraytyp: {}; Elementtyp: {}".format( t_length, t_type,
t_entry_type))
```

```
Array-Länge: 1001;   Arraytyp: <class 'numpy.ndarray'>;   Elementtyp: <class 'numpy.float64'>
```

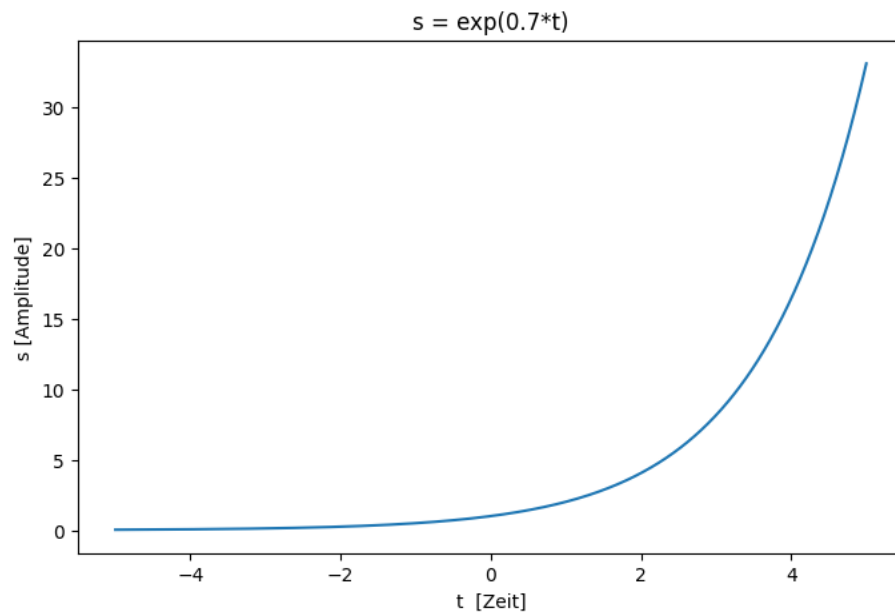
Exponentielle Funktion

Zur Erzeugung einer exponentiellen Funktion kann aus dem Modul `numpy` das Objekt `exp` verwendet werden. Mit dem Objekt `plt` kann dann die exponentielle Funktion geplottet werden.

```
# Erzeugung eines Arrays mit exponentiellen y-Werten für alle Einträge in t.

m = 0.7                # Wachstumsrate
s = np.exp(m*t)        # Exponentialfunktion
```

```
# Graphische Darstellung
plt.gcf().set_size_inches(8, 5) # Fenstergröße festlegen
plt.title(u's = exp(0.7*t)')    # Titel einfügen
plt.xlabel(u't [Zeit]')        # x-Achsenbeschriftung einfügen
plt.ylabel(u's [Amplitude]')   # y-Achsenbeschriftung einfügen
plt.plot(t, s)                 # Festlegen der Variablen x und y
plt.show()
```



Das Objekt `np.exp()` kann auch mit komplexen Exponenten verwendet werden.

Zum aneinanderfügen der Plots wird hier zusätzlich die Funktion `subplot` verwendet.

```
#Lösung
# komplexe Exponentialkomponente

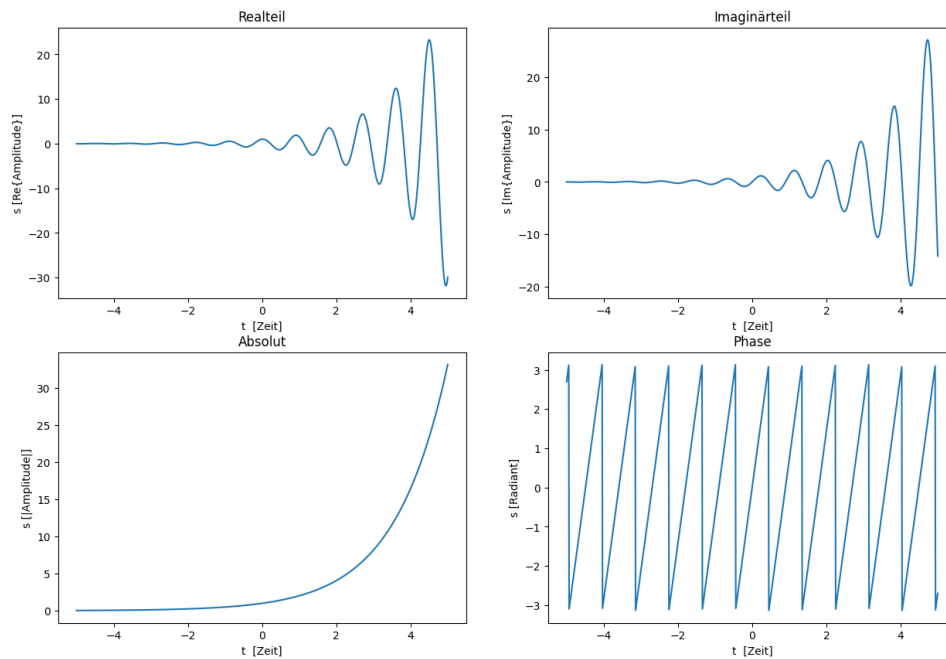
m = complex(0.7, 7)    # 0.7 = Realteil, 7 = Imaginärteil

# Erzeugung eines Arrays mit exponentiellen y-Werten mit t als x-Werten.

s = np.exp(m*t) # Ergänzen Sie die korrekte Funktion.
```

```
# Graphische Darstellung mittels Subplots
plt.subplot(221)
plt.title(u'Realteil')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's [Re{Amplitude}]')
plt.plot(t,np.real(s)) # Plot des Realteils über np.real()
plt.subplot(222)
plt.title(u'Imaginärteil')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's [Im{Amplitude}]')
plt.plot(t,np.imag(s)) # Plot des Imaginärteils über np.imag()
plt.subplot(223)
plt.title(u'Absolut')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's [|Amplitude|]')
plt.plot(t,np.abs(s)) # Plot des Absolutwerts über np.abs()
plt.subplot(224)
plt.title(u'Phase')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's [Radiant]')
plt.plot(t,np.angle(s)) # Plot der Phase über np.angle()

plt.gcf().set_size_inches(15, 10)
plt.show()
```



Sinus und Cosinus

Auch Sinus- und Cosinussignale lassen sich über Objekte aus dem numpy-Modul erzeugen. Folgende Objekte stehen dabei zur Verfügung:

- `np.pi`: Wert von π ,
- `np.sin()`: Sinusfunktion,
- `np.cos()`: Cosinusfunktion.

Implementieren Sie nun im Folgenden ein Array mit den Werten der Sinusfunktion mit der Kreisfrequenz von $\pi/3$ und eine Cosinusfunktion mit der Kreisfrequenz von $2/3 * \pi$ im zeitlichen Bereich von -5 bis 5 und einer Auflösung von 1001 Einheiten.

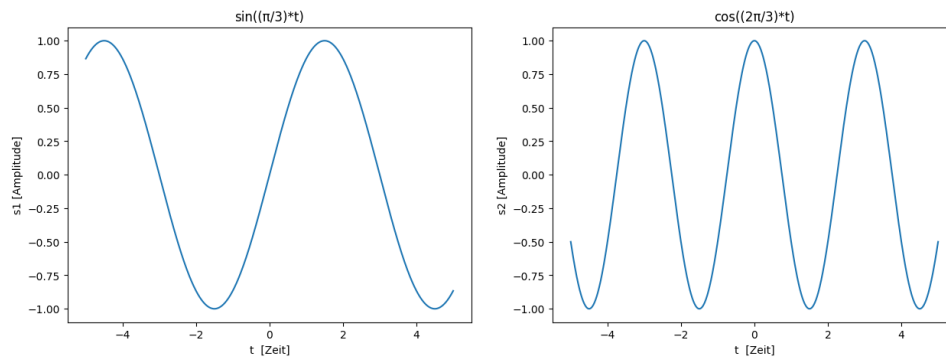
```
# Lösung
# Implementieren sie ein Sinus- und Cosinussignal mit den vorgegebenen Kreisfrequenzen

s1 = np.sin(np.pi/3 * t) # Sinus mit Kreisfrequenz  $\pi/3$ 
s2 = np.cos(2/3 * np.pi * t) # Cosinus mit Kreisfrequenz  $2\pi/3$ 
```

```
# Graphische Darstellung mittels Subplots
# w =  $\pi/3$ , Sinus:
plt.subplot(121)
plt.xlabel(u't [Zeit]')
plt.ylabel(u's1 [Amplitude]')
plt.plot(t, s1)
plt.title(u'sin(( $\pi/3$ )*t)')

# w =  $2\pi/3$ , Cosinus:
plt.subplot(122)
plt.xlabel(u't [Zeit]')
plt.ylabel(u's2 [Amplitude]')
plt.plot(t, s2)
plt.title(u'cos(( $2\pi/3$ )*t)')

plt.gcf().set_size_inches(15, 5)
plt.show()
```



Impuls- und Sprungsignale

Mit dem Objekt [np.where](#) können Impulse und Sprünge implementiert werden. Das Objekts ist wie folgt aufgebaut: `np.where(Bedingung, True, False)`. Über die `Bedingung` lässt sich zwischen den vorgegebenen Variablen `True` und `False` schalten. Für einen Sprung benötigt man eine Bedingung, zum Beispiel `(t >= 0)`. Für einen Impuls werden zwei Bedingungen benötigt (Beispiel: `(t >= 0) & (t <= 1)`).

Implementieren Sie im Folgenden die vorgegebenen Signale im zeitlichen Bereich von -5 bis 5 und einer Auflösung von 1001 Werten.

```
# Lösung

# Impulsbreite = 0.1 um den Nullpunkt, Amplitude = 1, Vorzustand = 0.
s1 = np.where((t >= -0.05) & (t <= 0.05), 1, 0)

# Impuls mit Breite = 3 mit dem Mittelpunkt bei t = -0.5, Amplitude = 1, Vorzustand = 0.
s2 = np.where((t >= -2) & (t <= 1), 1, 0)

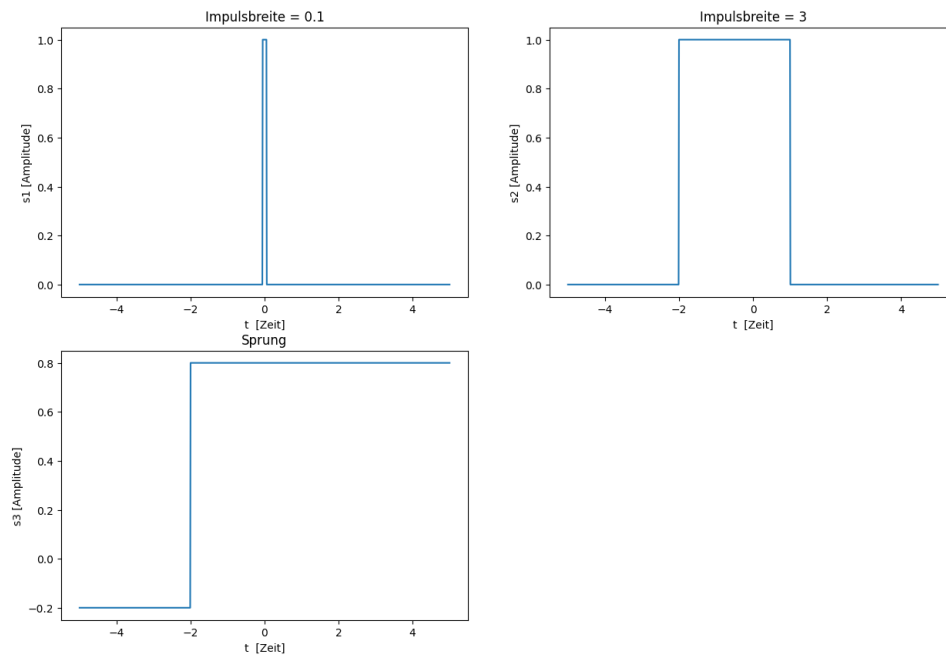
# Sprung bei t >= -2, Amplitude = 1, Vorzustand = -0.2.
s3 = np.where(t >= -2, .8, -.2)
```

```
# Graphische Darstellung mittels Subplots
plt.subplot(221)
plt.title(u'Impulsbreite = 0.1')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's1 [Amplitude]')
plt.plot(t, s1)

plt.subplot(222)
plt.title(u'Impulsbreite = 3')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's2 [Amplitude]')
plt.plot(t, s2)

plt.subplot(223)
plt.title(u'Sprung')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's3 [Amplitude]')
plt.plot(t, s3)

plt.gcf().set_size_inches(15, 10)
plt.show()
```



Diskrete Signaldarstellung

Alle bisherigen Beispiele sind trotz ihres Aussehens zeitdiskret, zwischen deren Punkten eine durchgehende Linie interpoliert wurde. Nur durch die große Anzahl an berechneten Punkten scheinen die Darstellungen zeitkontinuierlich zu sein. Um die zeitdiskrete Eigenschaft graphisch besser darzustellen, kann `plt.stem()` anstelle von `plt.plot()` verwendet werden.

Um ein einfaches inkrementelles Array zu erzeugen, wird `np.arange()` verwendet.

`np.arange()` und `plt.stem()` werden im Folgenden auf eine Exponentialfunktion angewendet:

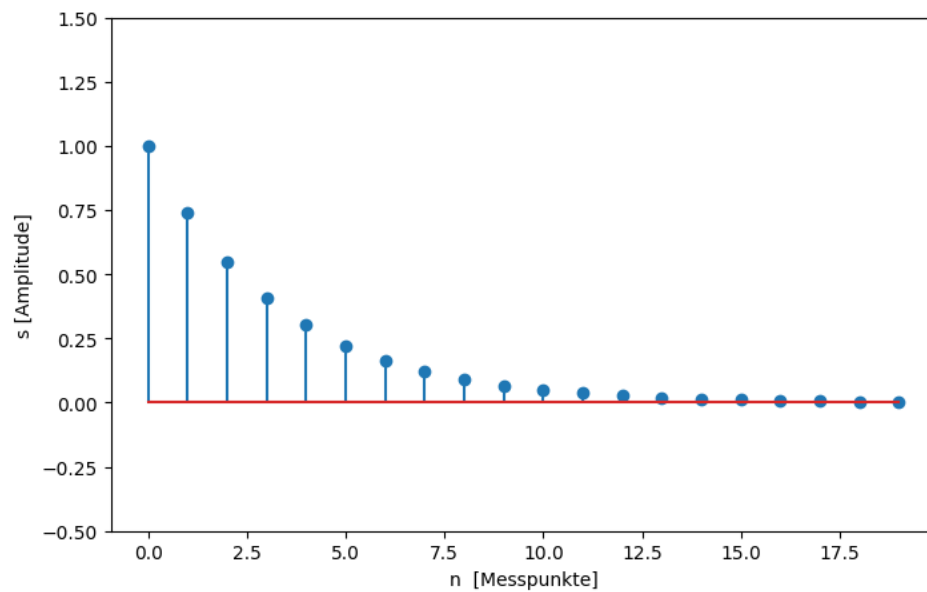
Anmerkung: Die Option `use_line_collection = True` erzeugt die Graphik effizienter. Mehr Informationen dazu finden Sie in [der Dokumentation](#).

```
# Erzeugen von 21 Samples
n = np.arange(0, 20)

# Erzeugen eines Arrays mit dem exponentiellen Werten
s = np.exp(-0.3*n)

# Graphische Darstellung
plt.ylim(-0.5, 1.5)
plt.xlabel(u'n [Messpunkte]')
plt.ylabel(u's [Amplitude]')
plt.stem(n, s, use_line_collection=True) # Erstellung des Stem plot

plt.gcf().set_size_inches(8, 5)
plt.show()
```



Implementieren Sie nun im Folgenden die diskreten Signaldarstellungen für folgende Sinus- und Cosinusfunktionen:

- $s_1 = \sin(n \cdot \pi)$
- $s_2 = \sin(n \cdot \pi/6)$
- $s_3 = \cos(n \cdot \pi/9)$
- $s_4 = \cos(2n)$

```
# Lösung
# Implementieren sie zwei Sinus- und zwei Cosinussignal mit den vorgegebenen Werten.

n = np.arange(51)          # Es sollen insgesamt 51 Samples (0-51) angezeigt werden.

s1 = np.sin(n*np.pi) # sin(n · π)
s2 = np.sin(n*np.pi/6) # sin(n · π/6)
s3 = np.cos(n*np.pi/9) # cos(n · π/9)
s4 = np.cos(2*n) # cos(2n)
```

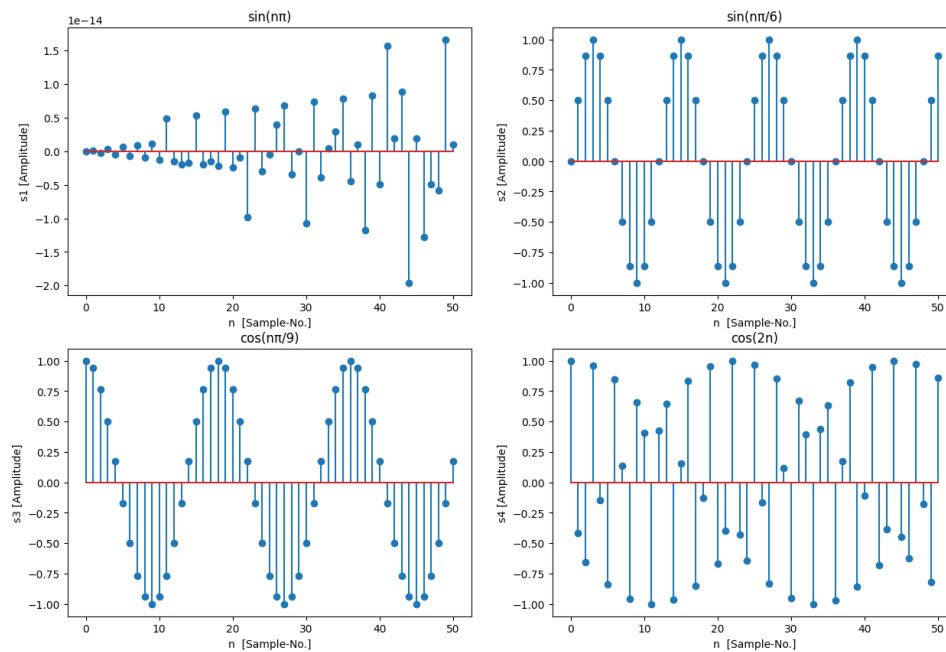
```
# Graphische Darstellung
plt.subplot(221)
plt.title(u'sin(nπ)')
plt.xlabel(u'n [Sample-No.]')
plt.ylabel(u's1 [Amplitude]')
plt.stem(n, s1, use_line_collection=True)

plt.subplot(222)
plt.title(u'sin(nπ/6)')
plt.xlabel(u'n [Sample-No.]')
plt.ylabel(u's2 [Amplitude]')
plt.stem(n, s2, use_line_collection=True)

plt.subplot(223)
plt.title(u'cos(nπ/9)')
plt.xlabel(u'n [Sample-No.]')
plt.ylabel(u's3 [Amplitude]')
plt.stem(n, s3, use_line_collection=True)

plt.subplot(224)
plt.title(u'cos(2n)')
plt.xlabel(u'n [Sample-No.]')
plt.ylabel(u's4 [Amplitude]')
plt.stem(n,s4, use_line_collection=True)

plt.gcf().set_size_inches(15, 10)
plt.show()
```



In den Graphen s2 und s3 erkennt man gut den Sinus-/Cosinus-Verlauf. Als Ausgabepunkte für s1 sind die Nulldurchgänge gewählt worden und sollten alle theoretisch den Wert Null haben. Durch Rundungsfehler ist das Ergebnis der Rechnung nur nahe Null. Und in Graph s4 ist die Abtastung nicht synchronisiert, weshalb man die Frequenz nicht ansehen kann.

Implementieren Sie nun einen Impuls und ein Sprungsignal mit `np.where` und lassen Sie sich den mit `plt.stem` ausgeben:

- s1 = Impuls an Position n = -2
- s2 = Sprung an Position n = -2 von 0 auf 1

```
# Lösung
# Implementieren Sie.

n = np.arange(-10, 10) # Es sollen insgesamt 21 Samples (-10 bis +10) angezeigt werden.

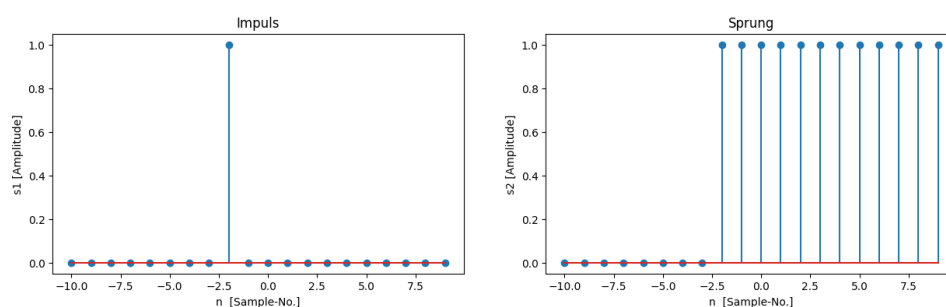
s1 = np.where(n == -2, 1, 0)

s2 = np.where(n >= -2, 1, 0)
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title(u'Impuls')
plt.xlabel(u'n [Sample-No.]')
plt.ylabel(u's1 [Amplitude]')
plt.stem(n, s1, use_line_collection=True)

plt.subplot(122)
plt.title(u'Sprung')
plt.xlabel(u'n [Sample-No.]')
plt.ylabel(u's2 [Amplitude]')
plt.stem(n, s2, use_line_collection=True)

plt.gcf().set_size_inches(15, 4)
plt.show()
```





2. Signaloperation

In Python können Signale leicht über die Grundrechenarten verknüpft werden oder aber auch mit externen Modulen gefaltet werden. Für die Grundrechenarten stehen interne Objekte zur Verfügung, die mit den zugehörigen einfachen Rechner-Zeichen verwendet werden, also $+$, $-$, $*$ & $/$.

Ein paar Beispiele zur Verwendung der Grundrechenarten sind nun gegeben:

```
# Lösung
# Implementierung der Grundrechenarten
```

```
t = np.linspace(-5, 5, 1000)
```

```
s1 = np.sin(2*np.pi*t)
s2 = np.exp(-t/2)
```

```
s_add = s1+s2
s_mult = s1*s2
```

```
# Graphische Darstellung
```

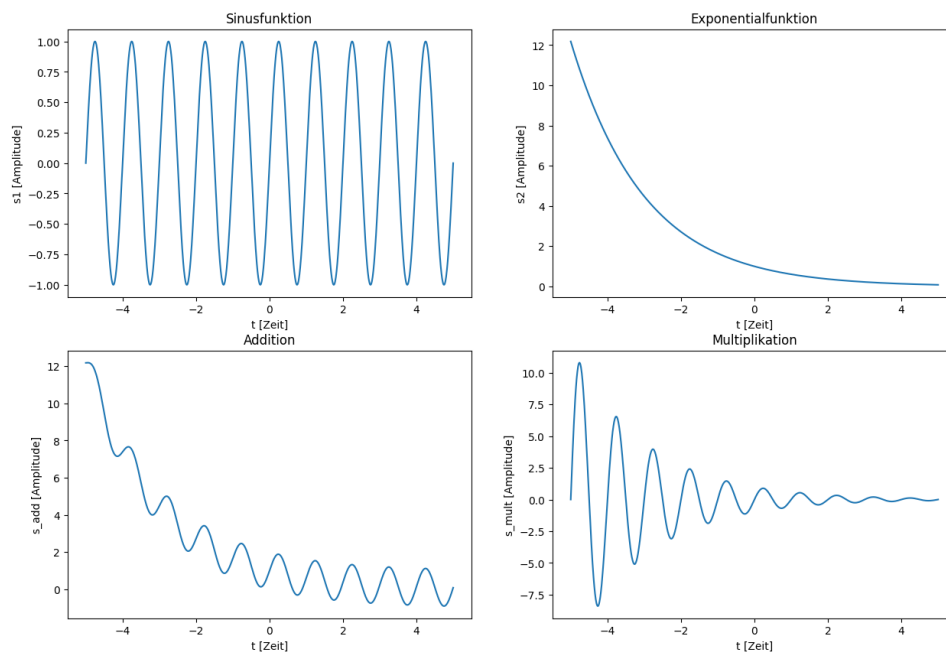
```
plt.subplot(221)
plt.title(u'Sinusfunktion')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's1 [Amplitude]')
plt.plot(t, s1)
```

```
plt.subplot(222)
plt.title(u'Exponentialfunktion')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's2 [Amplitude]')
plt.plot(t, s2)
plt.subplot(223)
```

```
plt.title(u'Addition')
plt.plot(t, s_add)
plt.xlabel(u't [Zeit]')
plt.ylabel(u's_add [Amplitude]')
plt.subplot(224)
```

```
plt.title(u'Multiplikation')
plt.plot(t, s_mult)
plt.xlabel(u't [Zeit]')
plt.ylabel(u's_mult [Amplitude]')
```

```
plt.gcf().set_size_inches(15, 10)
plt.show()
```



Eine Faltung von Arrays kann mit Hilfe von `numpy` mit dem Objekt `convolve()` durchgeführt werden. Lesen Sie dafür in der verlinkten Doku nach, wie Sie das Faltungs-Objekt nutzen können. Um Signale in einen Bereich zuzuschneiden, kann das schon für die Impulse und Sprünge verwendete Objekt `where` verwendet werden.

```
# Lösung
# Implementierung einer e-Funktion über den Bereich t mit dem Verlauf e_fun von -5 bis +4
und von +4 bis 5 einen Wert von 0.

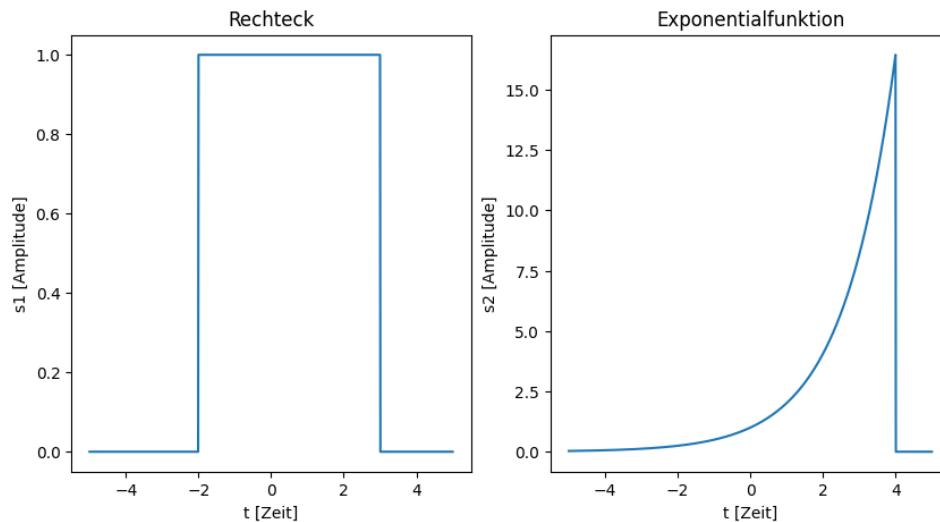
t = np.linspace(-5, 5, 1001)

s1 = np.where((t >= -2) & (t <= 3), 1, 0) # Rechteckfunktion
e_fun = np.exp(0.7*t)
s2 = np.where(t <= 4, e_fun, 0) # Exponentialfunktion
```

```
#Graphische Darstellung
plt.subplot(121)
plt.title(u'Rechteck')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's1 [Amplitude]')
plt.plot(t, s1)

plt.subplot(122)
plt.title(u'Exponentialfunktion')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's2 [Amplitude]')
plt.plot(t, s2)

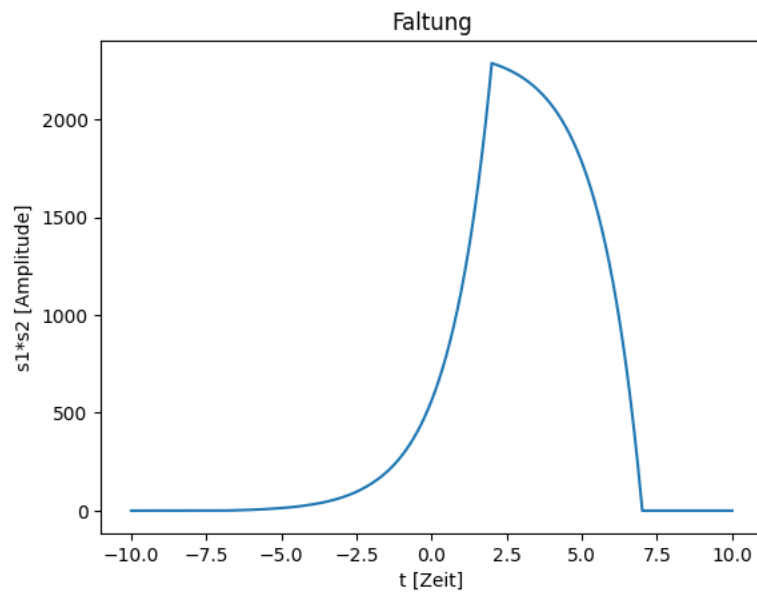
plt.gcf().set_size_inches(10, 5)
plt.show()
```



```
# Lösung
# Implementierung des Operators convolve aus der Bibliothek numpy

s_conv = np.convolve(s1, s2)
t_conv = np.linspace(-10, 10, len(s_conv))
```

```
# Graphische Darstellung
plt.title(u'Faltung')
plt.xlabel(u't [Zeit]')
plt.ylabel(u's1*s2 [Amplitude]')
plt.plot(t_conv, s_conv)
plt.show()
```



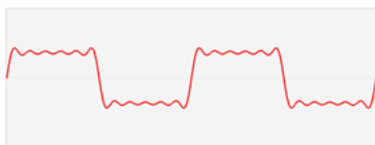
$\mathcal{F}(x)$

3. Diskrete Fourier-Transformation

Type of Transform	Example Signal
Fourier Transform <i>signals that are continuous and aperiodic</i>	
Fourier Series <i>signals that are continuous and periodic</i>	
Discrete Time Fourier Transform <i>signals that are discrete and aperiodic</i>	
Discrete Fourier Transform <i>signals that are discrete and periodic</i>	

FIGURE 8-2
Illustration of the four Fourier transforms. A signal may be continuous or discrete, and it may be periodic or aperiodic. Together these define four possible combinations, each having its own version of the Fourier transform. The names are not well organized; simply memorize them.

Hinweis: Die Beziehungen zwischen FS, FT, DTFT, DFT bzw. diskret vs. kontinuierlich und periodisch vs. nicht periodisch schauen Sie sich bitte die entsprechenden Folien in der Vorlesung an.



In diesem Beispiel zur Fouriertransformation soll eine sinc-Funktion Fourier-transformiert und invers rücktransformiert werden. Dabei entsteht ideal im Frequenzbereich eine Rechteckfunktion. Da wir uns durch die Nutzung von Zahlenarrays im Zeit- und Wertdiskreten bewegen, ist die Fouriertransformation diskret. Die wichtigen Formeln sind

(1) Diskrete Fouriertransformation (DFT):
$$H(n) = \frac{1}{N} \sum_{k=0}^{N-1} h(k) e^{-j2\pi n \frac{k}{N}},$$

(2) Inverse DFT (IDFT):
$$h(k) = \sum_{n=0}^{N-1} H(n) e^{j2\pi k \frac{n}{N}},$$

(3) sinc-Funktion:
$$\text{sinc}\left(\frac{x}{T}\right) = \frac{\sin\left(\frac{\pi x}{T}\right)}{\left(\frac{\pi x}{T}\right)},$$

(4) Rechteckfunktion:
$$\text{rect}\left(\frac{x}{\tau}\right) = \begin{cases} A & \text{if } |x| \leq \frac{\tau}{2} \\ 0 & \text{else} \end{cases},$$

(5) Beziehungen bei Fourier zwischen τ und T :
$$\tau = \frac{1}{T}.$$

Zur Umsetzung der Transformationen wird die Fast-Fourier-Transformation durch die Objekte [fft.fft](#) und [fft.ifft](#) aus dem Modul `scipy` verwendet. Die Verwendung der FFT beschleunigt die Berechnungszeit der Fouriertransformation, soll aber hier nicht weiter vertieft werden. Im Modul 1.1 wird näher die Umsetzung der Diskreten Fouriertransformation beschrieben.

Importieren Sie zunächst die Objekte `fft` und `ifft` aus der Bibliothek `scipy.fftpack`

```
# Lösung
# Import der Objekte fft und ifft aus der Bibliothek scipy.fftpack in den globalen
# Namensraum

from scipy.fft import fft, ifft
```

Zur Beschreibung der Sinc-Funktion werden zunächst die benötigten Variablen initialisiert:

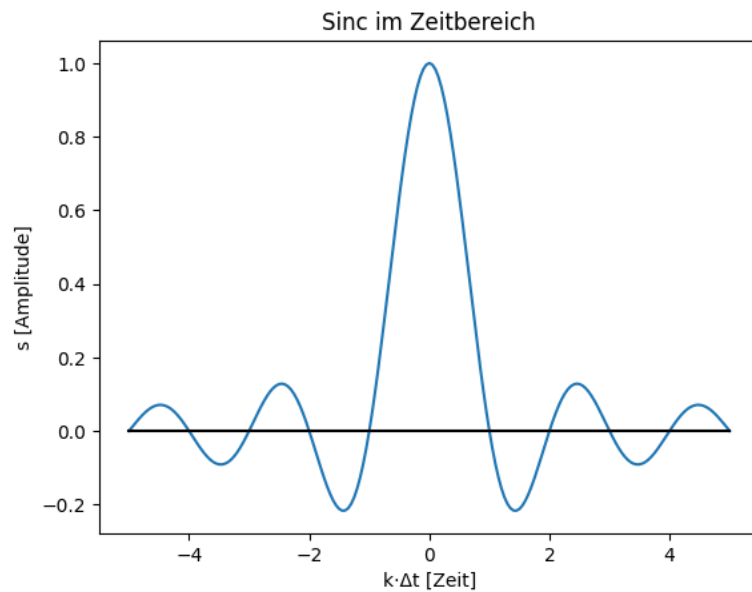
```
# Variablendefinition für sinc-Funktion

N          = 201                # Anzahl Abtastwerte
t_total = 10                    # zeitlicher Ausschnitt der Sinc-Funktion [-
T_total/2, ..., T_total/2]
A_sinc  = 1                     # Flächeninhalt ("Amplitude") der Sinc-Funktion
T_sinc  = 1                     # 'Periode' der Sinc-Funktion (erster
Nulldurchgang)

# Erzeugung der Sinc-Funktion

t = np.linspace(-t_total/2, t_total/2, N) # Erzeugung der äquidistanten x-Abtastpunkte
sinc = A_sinc * np.sinc(t/T_sinc)          # Definition der Sinc-Funktion
```

```
#Graphische Darstellung
plt.title(u'Sinc im Zeitbereich')
plt.xlabel(u'k·Δt [Zeit]')
plt.ylabel(u's [Amplitude]')
plt.plot(t, sinc)
plt.plot([-t_total/2, t_total/2], [0, 0], 'k') # Nulllinie
plt.show()
```

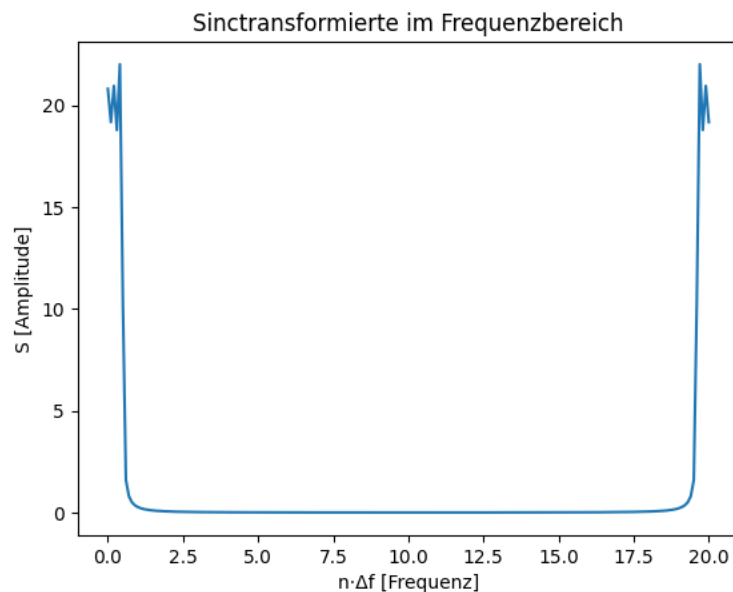


Nun wird die erzeugte Sinc-Funktion mittels der FFT-Funktion diskret Fourier-transformiert: Dazu muss zudem noch der Frequenzbereich berechnet werden.

```
# Diskrete Fourier-Transformation

# Erzeugung der Fouriertransformation
f = np.linspace(0, (N-1)/t_total, N)      # Bestimmung des Frequenzbereichs
H = fft(sinc)                             # Fouriertransformation der Sinc-Funktion
```

```
# Graphische Darstellung
plt.title(u'Sinctransformierte im Frequenzbereich')
plt.xlabel(u'n·Δf [Frequenz]')
plt.ylabel(u'S [Amplitude]')
plt.plot(f, np.abs(H))                    # Betrachtung des Absolutwerts der nun
komplexen Funktion
plt.show()
```



Die Funktion sieht aus wie eine in der Mitte halbierte Rechteckfunktion. Wenn man nun die Periodizität mit bedenkt, kann man die Funktion in der Mitte teilen und den rechten Hälfte in den negativen Frequenzbereich verschieben. Dies kann mit der Funktion [fftshift](#) aus der Bibliothek `numpy.fft` umgesetzt werden. Das Array für den Frequenzbereich muss dazu auch angepasst werden. Zum Vergleich mit dem gewünschten Frequenzgang wird in den Plot noch ein ideales Rechteck eingefügt.

```
# Lösung
# Darstellung der Fouriertransformation unter Berücksichtigung der Periodizität
# und Vergleich mit idealem Rechteck

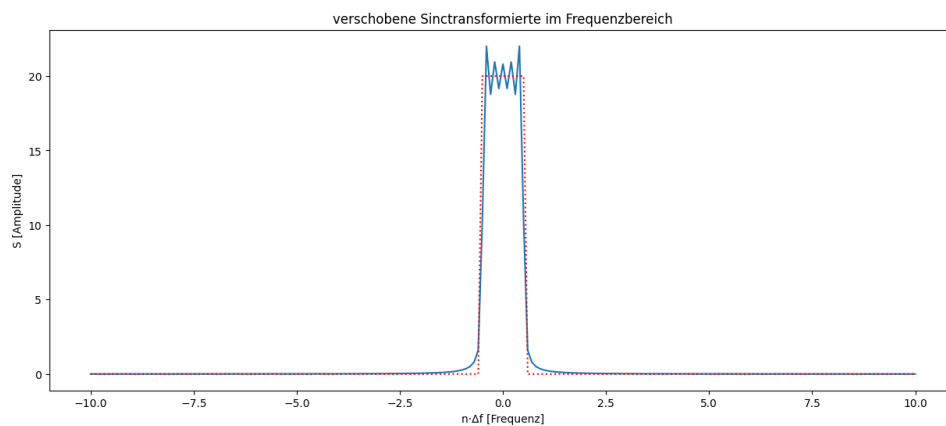
# Verschieben des Frequenzbereichs
f_shift = np.linspace(-(N-1)/(2*t_total), (N-1)/(2*t_total), N)

# Verschieben der Fouriertransformierten
H_shift = np.abs(np.fft.fftshift(H))

# Berechnung der Variablen für das ideale Rechteck
tau_rect = 1/T_sinc
Amp_rect = A_sinc * T_sinc * (N-1) / t_total

rect_ideal = np.where(abs(f_shift)<=tau_rect/2, Amp_rect, 0)
```

```
#Graphische Darstellung
plt.gcf().set_size_inches(15, 6)
plt.title(u'verschobene Sinctransformierte im Frequenzbereich')
plt.xlabel(u'n·Δf [Frequenz]')
plt.ylabel(u'S [Amplitude]')
plt.plot(f_shift, H_shift)
plt.plot(f_shift, rect_ideal,ls=':', c='r')
plt.show()
```



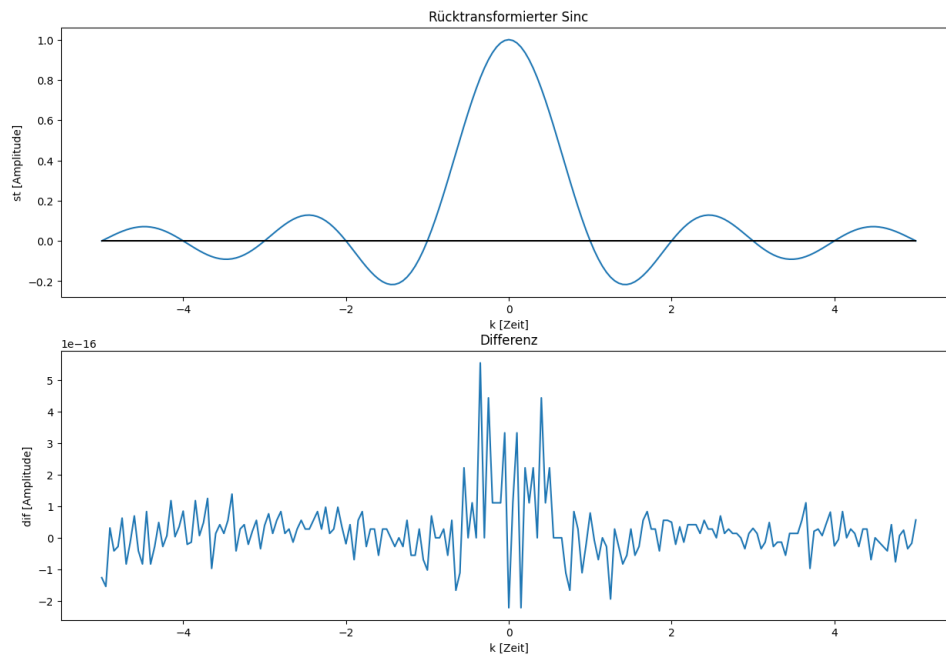
Rücktransformation und Vergleich mit Start-Sinc:

```
# inverse diskrete Fouriertransformation der Fouriertransformation
h = ifft(H)

h_real = np.real(h)
# Wertevergleich
dif = sinc - h_real
```

```
# Graphische Darstellung
plt.subplot(211)
plt.title(u'Rücktransformierter Sinc')
plt.xlabel(u'k [Zeit]')
plt.ylabel(u'st [Amplitude]')
plt.plot(t, h_real)
plt.plot([-t_total/2, t_total/2], [0,0], 'k') # Nulllinie

plt.subplot(212)
plt.title(u'Differenz')
plt.xlabel(u'k [Zeit]')
plt.ylabel(u'dif [Amplitude]')
plt.plot(t, dif)
plt.gcf().set_size_inches(15, 10)
plt.show()
```



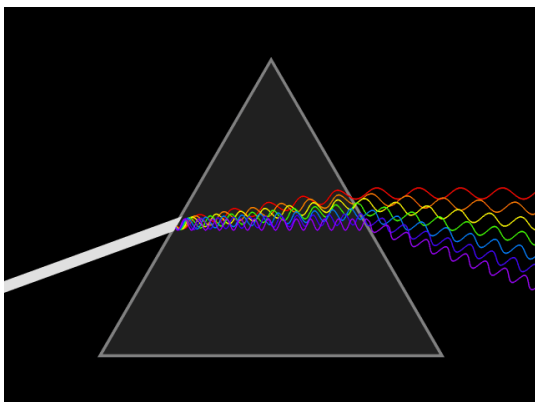
Der maximale Wertunterschied zwischen dem Original-Sinc und der Rücktransformierten liegt unter dem Bereich von 10^{-15} . Die Sinc-Funktion konnte dadurch nahezu perfekt wieder rekonstruiert werden.

References

1. Bild (Type of transformation) von [Steven W. Smith](#), Bild (Fourier Series) von [Lucas V. Barbosa](#)
2. [Offizielle Tutorials](#) über Python
3. [Einführung](#) von Jupyter Notebook
4. Eine visuelle Einführung von Fourier-Transformation: [What is the Fourier Transform](#)
5. DSP Guide: [The Scientist and Engineer's Guide to Digital Signal Processing](#)

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)

1.1 - FIR Filterentwurf



Ein FIR-Filter (Finite Impulse Response) ist ein Filter, dessen Impulsantwort (oder Antwort auf eine Eingabe mit endlicher Länge) von endlicher Dauer ist, da sie sich in endlicher Zeit auf Null einstellt. Dies steht im Gegensatz zu IIR-Filtern (Infinite Impulse Response), die interne Rückkopplungen aufweisen und möglicherweise unbegrenzt weiter reagieren (normalerweise abklingend). Die Entwurfsmethoden umfassen Least MSE, Minimax, Frequenzabtastung usw. (siehe die Vorlesung: Thema1-Filter). Hier diskutieren wir die Frequenzabtastung - und damit den Filterentwurf - mittels inverser Fourier-Transformation, da er gleichzeitig einen grundlegenden Einblick in die Signaltheorie bietet.

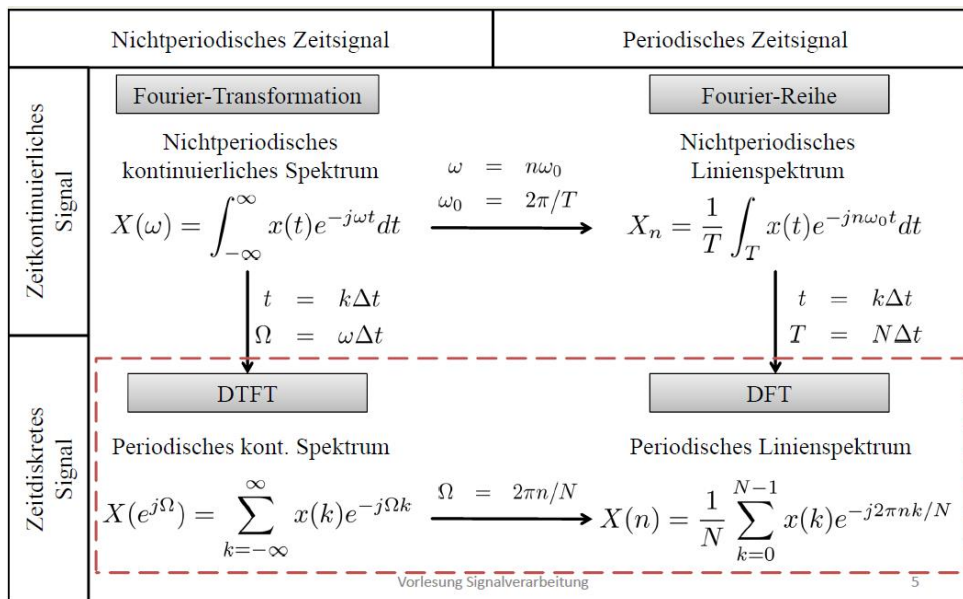
Inhalt

	1. Wiederholung
	2. Inverse DTFT
	2. Inverse DFT

?!

1. Wiederholung

Die Zusammenhänge zwischen Fourier-Transformation, Fourier-Reihe, DTFT und DFT, wie sie in der Vorlesung Thema1-Filter zusammengefasst wurden, sind in der nachfolgenden Grafik noch einmal dargestellt:



- **Fourier-Reihe:**

Die Fourier-Reihe beschreibt, dass jede periodische Funktion $x(t)$ durch eine unendliche Reihe dargestellt werden kann, die aus Sinusfunktion und Cosinusfunktion besteht (die Sinusfunktion und die Cosinusfunktion werden als Basisfunktionen gewählt, weil sie orthogonal zueinander sind). Nach der Euler-Formel können diese Funktionen wie nachfolgend auch als Exponentialform geschrieben werden
$$x_T(t) = \sum_{n=-\infty}^{\infty} X_n e^{j\omega_0 n t},$$
 weswegen die Fourier-Reihe auch als Exponentialreihe bezeichnet wird. Index n verdeutlicht die Periodizität in T , wobei T die zeitliche Periode des Signals $x(t)$ ist, über welche die Fourier-Koeffizienten X_n bestimmt werden. $\omega_0 = \frac{2\pi}{T}$ ist die Grundfrequenz (in [rad/s]).

- **Fourier-Transformation:**

Die Fourier-Transformation ist eigentlich eine Verallgemeinerung der Fourier-Reihe, da das Integral die Grenzform der Summe ist, wenn $T \rightarrow \infty \rightarrow \omega_0 \rightarrow 0$ ist [3]. Die Fourier-Transformation wird hauptsächlich zur Analyse kontinuierlicher nichtperiodischer Signale verwendet. Für ein kontinuierliches (= analoges), nichtperiodisches Signal kann das Spektrum durch Fourier-Transformation berechnet werden. Da das Signal kontinuierlich und nicht-periodisch ist, kann es Anteile aller möglichen Frequenzen von $\omega = 0$ bis $\omega \rightarrow \infty$ enthalten, wodurch das Spektrum ebenfalls nicht-periodisch und kontinuierlich (in ω) ist.

- **DTFT:**

Da ein digitaler Computer nur digitale Signale verarbeiten kann, ist es zunächst erforderlich, das ursprüngliche analoge Signal im Zeitbereich zu diskretisieren, d.h., im Zeitbereich abzutasten (z.B. über einen Dirac-Impuls an den Zeitpunkten $k\Delta t$) und eine Fourier-Transformation für das

abgetastete zeitdiskrete Signal, die sogenannte *zeitdiskrete Fourier-Transformation*, durchzuführen. Das Zeitsignal ist damit nicht-periodisch und diskret, wogegen das Spektrum periodisch und kontinuierlich ist. Dies ist der symmetrische Gegenfall zu der Fourier-Reihe, wo das analysierte Zeitsignal periodisch und kontinuierlich ist, das Spektrum dagegen diskret und nicht-periodisch ist (= Linienspektrum). Für Theorie-Enthusiasten ist eine umfangreichere Ausführung dieses Zusammenhanges in Dokument **impulseSamplingAndDTFT.pdf** in OPAL zu finden.

- **DFT:**

Die *diskrete Fourier-Transformation* erhält man, indem die DTFT bzw. das daraus resultierende, kontinuierliche Spektrum, an N festen Frequenzpunkten $f = \frac{n}{N} \cdot f_s$ im Intervall $[0, f_s]$ abgetastet wird. $\Omega = \frac{2\pi}{N} f$ wird dadurch zu $\Omega = \frac{2\pi}{N} n$, da sich f_s herauskürzt.

$x(k)$

2. Inverse DTFT

Die inverse DTFT ist eine Möglichkeit, um die Impulsantwort im Zeitbereich aus einem vorgegebenen Frequenzgang $H(\omega)$ im Bild-/Spektralbereich zu berechnen. Der gegebene Amplitudenfrequenzgang ist kontinuierlich, nicht-periodisch und reell (siehe Übung 2, Aufgabe 1).

Unter der Verwendung der inversen DTFT,

$$x(k) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H(e^{j\Omega}) e^{jk\Omega} d\Omega$$

wird implizit die Annahme getroffen, dass das ursprünglich nicht-periodische Spektrum $H(\omega)$ nun periodisch ist (mit einer spektralen Grundperiode, und weiterhin kontinuierlich + reell), über $H(e^{j\Omega})$ ausgedrückt und in eine nicht-periodische, diskrete Impulsantwort $x(k)$ transformiert wird. Wie in der Übung soll nachfolgend ein Hochpassfilter über die Methode der inversen DTFT entworfen.

Importieren Sie nun alle dafür benötigten externen Module:

```
# Lösung
# Importieren Sie:

import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate, fft, signal
```

Zuerst werden die Variablen für den gewünschten Hochpass definiert und ein Array mit dem idealen Frequenzgang H_{soll} erzeugt und geplottet. Dafür eignet sich die Numpy-Funktion [numpy.where\(\)](#). Das erste Argument ist eine Bedingung, gefolgt von zwei Skalaren oder Arrays als Argument zwei und drei. Abhängig von der Bedingung im ersten Argument wird entweder der erste oder zweite Wert bzw. aus dem ersten oder zweiten Array ausgewählt.

```
'''
Beispiel: Endliche Impulsantwort eines Hochpassfilters mittels inverser DTFT
'''

# Initialisierung aller wichtigen Variablen:
fs_Hz = 800          # Abtastfrequenz in [Hz]
ws_rad_s = 2*np.pi*fs_Hz # Abtastfrequenz in [rad]

fc_Hz = 200          # Grenzfrequenz (= corner frequency) des Filters in [Hz]
wc_rad_s = 2*np.pi*fc_Hz # Grenzfrequenz des Filters in [rad]


A = 1                # Amplitude im Passband

# idealer Frequenzgang:
H_soll = 0
w_rad_s = np.linspace(-ws_rad_s/2, ws_rad_s/2, fs_Hz) # Omega [-pi, pi]: Linear und
äquidistant aus [-ws_rad_s/2, ws_rad_s/2]
H_soll = np.where((w_rad_s >= -wc_rad_s) & (w_rad_s <= wc_rad_s), 0, A)
```



```
# Graphische Darstellung
plt.title('Idealer Frequenzgang über die spektralen Grundperiode [-$\omega_s$, $\omega_s$]')
plt.xlabel('Frequenz $\omega$ [rad/s]')
plt.ylabel('Amplitude $|H(e^{j\omega})|$')
plt.plot(w_rad_s, H_soll)

plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_9_0.png

Die Impulsantwort $h(k)$ wird über die IDTFT aus dem Frequenzverhalten des idealen Hochpass berechnet und geplottet. Dabei wird die Integration in der IDTFT mit dem Objekt [integrate.quad](#) über numerische Integration angenähert.


Hinweis: Der `//`-Operator führt eine Division durch und gibt eine ganzzahliges Ergebnis aus, indem der Rest hinter dem Komma abgeschnitten wird.

```
# Endliche Impulsantwort (reell) mit der IDTFT berechnen:
k = np.arange(-fs_Hz//2, fs_Hz//2)

# Über alle Sample-Zeitpunkte in k iterieren und numerisch integrieren (Annäherung an das Integral der IDTFT):
h = [] # Impulsantwort (Allokation des Arrays unbestimmter Länge)
for i in k:
    an, err = integrate.quad(lambda w_rad_s: A*np.cos(i*w_rad_s/fs_Hz), wc_rad_s, 3*wc_rad_s, limit=fs_Hz)
    h.append(an / ws_rad_s)
```

```
# Graphische Darstellung:
kLimMin = -200
kLimMax = 200
plt.title('Impulsantwort h(k) (Sichtbar für k=[%d, %kLimMin + %d])' %kLimMax)
plt.xlabel('Abtastwertindex k')
plt.ylabel('Amplitude h(k)')
plt.xlim(kLimMin, kLimMax)
plt.stem(k, h, use_line_collection=True)

plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_12_0.png

Anschließend wird die Impulsantwort $h(k)$ mittels der DTFT zurück in den Frequenzbereich transformiert, geplottet und mit dem vorgegebenen, idealen Frequenzgang verglichen. Dabei erkennt man sehr gut ein Überschwingen bei den Sprüngen im idealen Hochpass durch die Rücktransformation mit einer *endlichen* Anzahl an Frequenzen/Abtastwerten.


```
# Realer Frequenzgang der berechneten, abgeschnittenen Impulsantwort h(k) mittels der DTFT:
W_rad = np.linspace(-np.pi, np.pi, 10000) # Laufvariable Omega definieren
H_ist = np.zeros(W_rad.size, dtype='complex128') # H_ist für das Ergebnis der DTFT allokalieren als 2x64bit float

# Über alle h(k) iterieren und jeweils X(k) berechnen:
for i in k:
    H_ist += h[i]*np.exp(-1j*W_rad*k[i])
```

```
# Graphische Darstellung:
plt.subplot(211)
plt.title('realer Frequenzgang der Impulsantwort h(k) für den Bereich k=[%d' %np.min(k) +
', %d]' %np.max(k))
plt.xlabel('Frequenz $\omega$ [rad/s]')
plt.ylabel('$|H(e^{j\omega})|$')
plt.plot(W_rad*fs_Hz, np.abs(H_ist))

plt.subplot(212)
plt.title('Idealer Frequenzgang über die spektrale Grundperiode [-$\omega_s$,
$\omega_s$]')
plt.xlabel('Frequenz $\omega$ [rad/s]')
plt.ylabel('$|H(e^{j\omega})|$')
plt.plot(w_rad_s, H_soll)

plt.gcf().set_size_inches(15, 10)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR
Filterentwurf_solved_15_0.png

Um den Einfluss der Anzahl an Abtastwerten L der Impulsantwort auf den Frequenzgang zu veranschaulichen, wird nun über eine for-Schleife die soeben berechnete Impulsantwort $h(k)$ mittels `np.where`-Operator beschnitten ($h_w(k)$). Das hat zur Folge, dass die Impulsantwort mit einem Rechteckfenster gefenstert wird. Die Auswirkung dessen kann über Fouriertransformation im Frequenzbereich $H_w(e^{j\omega})$ betrachtet werden.

```
# Lösung
'''
Aufgabe: Beschränkte Impulsantwort mit verschiedenen Fensterlängen L = 10, 20, 50, 200
berechnen
(Hinweis: Sie können die berechnete Impulsantwort aus dem vorherigen Beispiel direkt
weiterverwenden.
Probieren Sie auch aus, was passiert, wenn die Impulsantwort nicht symmetrisch um k = 0
gefenstert wird.)
'''

for L in (10, 20, 50, 100):
    # Fensterung mit einem Rechteckfenster (auf Wertebereich der Länge +-L beschränken):
    hw = np.where((k<=(L/2)) & (k>=(-L/2)), h, 0)


    # Realer Frequenzgang der berechneten, abgeschnittenen Impulsantwort h(k) mittels der
    DTFT für k = [-L/2, L/2]:
    W_rad = np.linspace(-np.pi, np.pi, 1000)
    H_ist = np.zeros(W_rad.size, dtype='complex128') # H_ist für das Ergebnis der DTFT
    allokieren


    # Über alle h(k) iterieren und jeweils X(k) berechnen und aufsummieren:
    for i in k:
        H_ist += hw[i]*np.exp(-1j*W_rad*k[i])


    plt.subplot(121)
    plt.title('Impulsantwort mit Fensterlänge L=%d' %L)
    plt.xlabel('Abtastwertindex k')
    plt.ylabel('Amplitude h(k)')
    plt.xlim(fs_Hz//2-L, fs_Hz//2+L)
    plt.xticks(np.arange(fs_Hz//2-L, fs_Hz//2+L+1, L/5), range(-L, L+1, L/5))
    plt.stem(hw, use_line_collection=True)


    plt.subplot(122)
    plt.title('Resultierender Frequenzgang für L=%d' %L)
    plt.xlabel('Frequenz $\omega$ [rad/s]')
    plt.ylabel('Amplitude $|H(e^{j\omega})|$')
    plt.plot(W_rad*fs_Hz, np.abs(H_ist))

    plt.gcf().set_size_inches(16, 4)
    plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_17_0.png

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_17_1.png

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_17_2.png

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_17_3.png

$h(k)$

3. Inverse DFT

Für die IDTFT in Abschnitt 2 musste ein Integral über die (logischerweise kontinuierliche) Variable Ω ausgewertet werden, was über eine numerische Integration gelöst wurde, aber für Rechner nicht besonders elegant ist. Die IDFT eignet sich für digitale Computer deutlich besser, da sie mit diskreten Abtastwerten arbeitet.

Im Folgenden soll daher dasselbe Vorgehen mit der IDFT anstatt der IDTFT durchgeführt werden. Die IDFT ist definiert durch
$$h(k) = \sum_{n=0}^{N-1} H(n) e^{j2\pi kn/N},$$
 und ergibt sich direkt aus der IDTFT, wenn ω bzw. f mit N Punkten abgetastet wird (siehe Abschnitt 1, Wiederholung).

Um die Gleichung der IDFT zunächst besser zu veranschaulichen, wird nachfolgend einmal ein Tiefpassfilter mit einer selbst definierten IDFT nach der obigen Gleichung entworfen.

Funktionen in Python starten mit dem Präfix **def**, gefolgt vom Funktionsnamen (idft z.B.) und den Argumenten in Klammern (arg1, arg2, ...), getrennt durch Kommata. Der Ausgabewert (in diesem Falle die Impulsantwort $h(k)$) folgt nach einem **return** Statement.

```
'''
Funktion definieren: idft(func, N)
Impulsantwort mit IDFT, Verschiebung und Fensterung

    param func: Funktion für Impulsantwort
    param N: Sample Zahl
'''

# IDFT
def idft(func, N):
    h = []
    k = np.arange(N)
    for i in range(N):
        an = np.sum(func * np.exp(1j*2*np.pi*k*i/N))
        h.append(an)
    # Verschiebung
    h = h[N//2:] + h[:N//2]
    # Normalisierung
    h = h / (2*np.max(h))
    return h
```

Wenden Sie nun die neu definierte Funktion zur Berechnung der Impulsantwort h aus dem Frequenzgang des Tiefpassfilters H_{soll} an, der eine Grenzfrequenz bei $f_s/4$ haben soll. Für den idealen Frequenzgang H_{soll} wird die erste Hälfte des Spektrums (= bis zur Nyquistfrequenz $f_s/2$) vorgegeben und der (um die y-Achse gespiegelte) Abschnitt für die negativen Frequenzen (entspricht $f=0$ bis $f=-f_s/2$) bzw. $[f=f_s/2$ bis $f_s]$, für reelle Signale) daran angehängt, d.h. er wird über den Bereich $\Omega = [0, 2\pi]$ festgelegt.

```
# Lösung

'''
Beispiel: Endliche Impulsantwort eines Tiefpassfilters mittels inverser DFT
'''


# Initialisierung
N = 512 # Sample Zahl
A = 1 # Amplitude
fs_Hz = 800
fc_Hz = fs_Hz/4
Wc_rad = 2*np.pi*fc_Hz/fs_Hz

# Berechnung des idealen Frequenzganges:
W_rad = np.linspace(0, 2*np.pi, N)
H_soll = np.where((W_rad >= Wc_rad) & (W_rad <= 2*np.pi - Wc_rad), A, 0)

# Impulsantwort
h = idft(H_soll, N)
```

```
# plot
plt.subplot(121)
plt.title('Idealer Amplitudenfrequenzgang in der Grundperiode')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|H(n)|')
plt.plot(W_rad*fs_Hz/(2*np.pi), H_soll)
plt.plot([fs_Hz/2, fs_Hz/2],[0, 1], '--k')

plt.subplot(122)
plt.title('Impulsantwort')
plt.xlabel('Sample Nummer k')
plt.ylabel('Amplitude h(k)')
plt.xlim(150, 360)
plt.stem(np.real(h), use_line_collection=True)
plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR
Filterentwurf_solved_22_0.png

Für die DFT und ihre inverse IDFT gibt es im SciPy-Paket die fertigen Objekte [fft.fft](#) und [fft.ifft](#), wobei immer die FFT verwendet wird (generell wird in der Praxis für die Berechnung der DTF ausschließlich die FFT verwendet. Falls die Anzahl an Abtastwerten keine Zweierpotenz ist, wird das Signal bis zur nächsthöheren Zweierpotenz mit Nullen aufgefüllt). Die selbst definierte IDFT soll nun mit der `ifft`-Funktion des SciPy-Paketes verglichen werden. Ebenfalls sehr nützlich ist die [ifftshift\(\)](#), welche das DFT/FFT Spektrum in der Mitte um $\Omega = 0$ zentriert, indem die beiden Hälften der Impulsantwort $h(0:N/2-1)$ und $h(N/2:N-1)$ vertauscht werden.


```
# Lösung
# Impulsantwort mittels ifft, ifftshift
# Initialisierung
N = 512 # Länge der IFFT
A = 1 # Amplitude
fs_Hz = 8000;
fc_Hz = fs_Hz/4;
Wc_rad = 2*np.pi*fc_Hz/fs_Hz

# Berechnung des idealen Frequenzganges:
W_rad = np.linspace(0, 2*np.pi, N)
H_soll = np.where((W_rad >= Wc_rad) & (W_rad <= 2*np.pi - Wc_rad), A, 0)

h = np.fft.ifftshift(fft.ifft(H_soll, N))
k = np.arange(0, N)
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Grundperiode des idealen Frequenzganges')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|H(n)|')
plt.plot(W_rad*fs_Hz/(2*np.pi), H_soll)
plt.plot([fs_Hz/2, fs_Hz/2],[0, 1], '--k')

plt.subplot(122)
plt.title('Impulsantwort')
plt.xlabel('Sample Nummer k')
plt.ylabel('h(k)')
plt.xlim(150, 350)
plt.stem(k, np.real(h), '-b', use_line_collection=True)
plt.stem(k, np.imag(h), '-r', use_line_collection=True)
plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_25_0.png

Wenn alles richtig gemacht wurde, sind beide Ergebnisse identisch.

Nun soll (wie in Kapitel 2) die Auswirkung der Länge L der berechneten Impulsantwort $h(k)$ auf den Frequenzgang des Tiefpasses und der Einfluss von verschiedenen Fensterfunktionen betrachtet werden. Zur Erzeugung des Fensters wird nun das Objekt [signal.get_window\(\)](#) genutzt, welches Funktionen für die wichtigsten Fenster zur Verfügung stellt. Die berechneten Abtastwerte der Fensterfunktion werden mit der Impulsantwort punktweise multipliziert.

```
# Lösung
# Fensterung der berechneten Impulsantwort h(k) mit verschiedene Längen L

for L in (10, 20, 50, 100):


    # Erzeugung des Fensters:
    window = signal.get_window('boxcar', L+1)
    N = 512 # Länge der FFT
    # Erzeugung einer Maske, mit der nur der Bereich [-L/2, L/2] des Signals ungleich Null ist:
    mask = np.zeros(N)
    mask[(N-L)//2:(N+L)//2+1] = window
    h_list = h * mask

    # Frequenzgang berechnen mit der FFT:
    H_list = np.abs(fft.fft(h_list))
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Impulsantwort mit Fensterlänge L=%d' %L)
plt.xlabel('Sample Nummer k')
plt.ylabel('h(k)')
plt.xlim(N//2-L, N//2+L)
plt.xticks(np.arange(N//2-L, N//2+L+1, L//5), range(-L, L+1, L//5))
plt.stem(np.real(h_list), use_line_collection=True)
plt.plot(mask, ls=':', c='r')

plt.subplot(122)
plt.title('Realer Amplitudenfrequenzgang in der Grundperiode')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|H(n)|')
plt.plot(W_rad*fs_Hz/(2*np.pi), H_list)

plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR Filterentwurf_solved_28_0.png

Zum Schluss sollen unterschiedliche Fenstertypen auf die Impulsantwort h angewendet und die Veränderung auf den Frequenzgang gegenübergestellt werden.

```

# Lösung
# mit verschiedenen Fenstern
L = 50 # Hälfte der Fensterlänge um h(0) herum
N = 512 # Länge der FFT
for windowType in ('boxcar', 'triangle', 'blackman', 'hanning'):
    mask = np.zeros(N)
    mask[(N-L)//2:(N+L)//2+1] = signal.get_window(windowType, L+1)
    h_list = h * mask




    # Frequenzgang berechnen mit der FFT:
    H_list = np.abs(fft.fft(h_list))

    # Plotten:
    plt.subplot(121)
    plt.title('Impulsantwort durch %s-Fenster' %windowType)
    plt.xlabel('Sample Nummer k')
    plt.ylabel('Amplitude h(k)')
    plt.xlim(150, 350)
    plt.stem(np.real(h_list), use_line_collection=True)
    plt.plot(mask, ls='--', c='r')

    plt.subplot(122)
    plt.title('Frequenzgang nach Fensterung')
    plt.xlabel('Frequenz [Hz]')
    plt.ylabel('|H(n)|')
    plt.plot(W_rad*fs_Hz/(2*np.pi), H_list)

plt.gcf().set_size_inches(15, 5)
plt.show()

```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
 sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR
 Filterentwurf_solved_30_0.png
 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
 sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR
 Filterentwurf_solved_30_1.png
 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
 sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.1 - FIR
 Filterentwurf_solved_30_2.png

```

-----
ValueError                                Traceback (most recent call last)
File ~\AppData\Local\Programs\Python\Python310\lib\site-
packages\scipy\signal\windows\_windows.py:2214, in get_window(window, Nx, fftbins)
    2213 try:
-> 2214     beta = float(window)
    2215 except (TypeError, ValueError) as e:

ValueError: could not convert string to float: 'hanning'

During handling of the above exception, another exception occurred:

KeyError                                Traceback (most recent call last)
File ~\AppData\Local\Programs\Python\Python310\lib\site-
packages\scipy\signal\windows\_windows.py:2232, in get_window(window, Nx, fftbins)
    2231 try:
-> 2232     winfunc = _win_equiv[winstr]
    2233 except KeyError as e:

KeyError: 'hanning'

The above exception was the direct cause of the following exception:

ValueError                                Traceback (most recent call last)
Cell In [16], line 7
     5 for windowType in ('boxcar', 'triangle', 'blackman', 'hanning'):
     6     mask = np.zeros(N)
----> 7     mask[(N-L)//2:(N+L)//2+1] = signal.get_window(windowType, L+1)
     8     h_list = h * mask
    10     # Frequenzgang berechnen mit der FFT:

File ~\AppData\Local\Programs\Python\Python310\lib\site-
packages\scipy\signal\windows\_windows.py:2234, in get_window(window, Nx, fftbins)
    2232 winfunc = _win_equiv[winstr]
    2233 except KeyError as e:
-> 2234     raise ValueError("Unknown window type.") from e
    2236 if winfunc is dpss:
    2237     params = (Nx,) + args + (None,)

ValueError: Unknown window type.

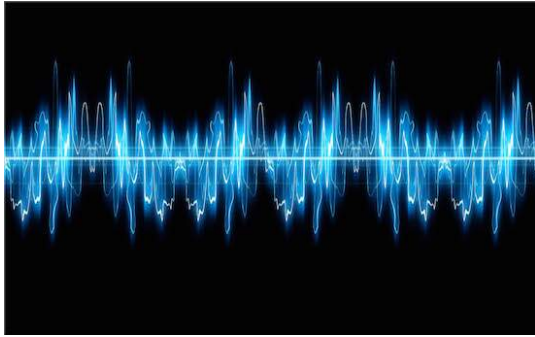
```

References





1. Titelbild von [Lucas Vieira](#)
 2. DSP Guide: [The Scientist and Engineer's Guide to Digital Signal Processing](#)
 3. http://fourier.eng.hmc.edu/e101/lectures/Fourier_Transform_C/node1.html
-

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)

1.2 - IIR-Filterentwurf



Inhalt

	1. Wiederholung
	2. Butterworth-Filter
	3. Chebyshev-Filter
	4. ToDo: Vorverzerrung

?!

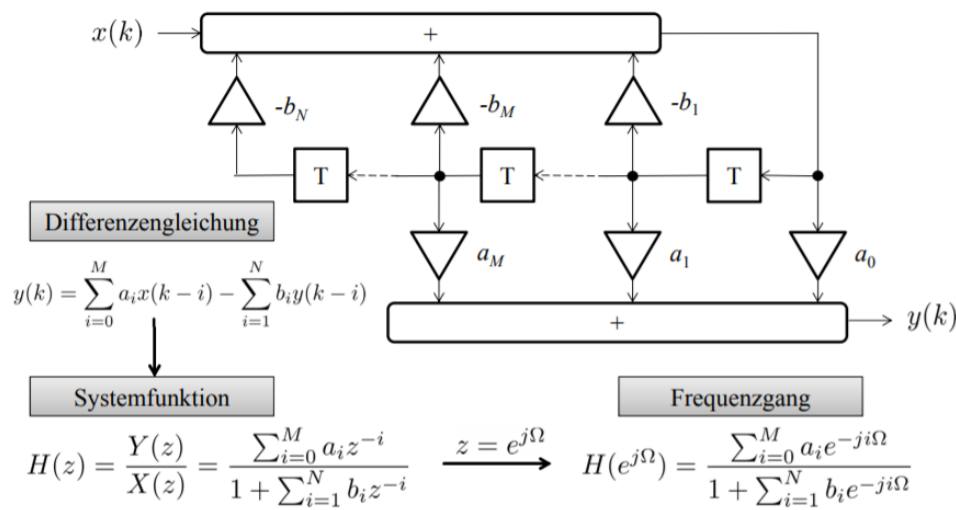
1. Wiederholung

Im Notebook 1.1 haben wir die Implementierung von FIR-Filtern diskutiert. FIR-Filter bieten viele Vorteile wie einen linearen Phasengang, außerdem sind sie immer stabil. Zur Erinnerung: eine typische FIR hat die Differenzengleichung
$$y_{\text{FIR}}(k) = \sum_{i=0}^M a_i x(k-i).$$

$\end{equation*}$

Je länger der Filter/je höher die Filterordnung ($M \rightarrow \infty$), desto besser kann der ideale Amplitudenfrequenzgang angenähert werden. Die z.T. hohe Anzahl an Koeffizienten kann ein Problem werden, wenn dadurch die Einschwingzeit des Filters zu lang ist. Eine Alternative bieten die IIR-Filter (Infinite Impulse Response Filter). Der größte Unterschied zwischen IIR- und FIR-Filtern besteht darin, dass die Ausgabe $y(k)$ zum aktuellen Abtastwertindex k nicht nur von der Eingabe $x(k)$, sondern auch von vorherigen Abtastwerten der Ausgabe $y(k-1), y(k-2), \dots$ selbst abhängt. Die Standardform eines IIR-Filters lautet damit
$$y_{\text{IIR}}(k) = \sum_{i=0}^M a_i x(k-i) - \underbrace{\sum_{i=0}^N b_i y(k-i)}_{\text{neuer, rekursiver Teil}}$$

Die grundlegende Struktur von IIR-Filtern und dessen Darstellungsformen sind ebenfalls zur Erinnerung in der folgenden Abbildung noch einmal zusammengefasst (siehe dazu auch die Vorlesungsfolien Thema1-Filter sowie die erste Übung).



Der Unterschied zwischen FIR- und IIR-Systemen:

- FIR:
 - Impulsantwort hat eine endliche Länge
 - keine Rückkopplungsschleife/rekursiven Teil
 - keine Pole (immer stabil)
- IIR:
 - Impulsantwort hat eine unendliche Länge
 - rekursive Struktur (Rückkopplungsschleife)
 - mindestens ein Pol ($b_i \neq 0$)

Im Allgemeinen wird beim Entwerfen eines digitalen IIR-Filters zuerst ein entsprechender analoger Filter entworfen und dann wird der analoge Filter durch eine [bilineare Transformation](#) oder [Impulsantwort-Invarianzmethode](#) in einen digitalen Filter umgewandelt. Dies bedeutet, dass die Systemfunktion $H(s)$ in der komplexen s -Ebene entworfen wird. Wenn sich alle Pole von $H(s)$ in der linken Halbebene von s befinden, ist der Filter stabil. Die Komplexität liegt in der Bestimmung der Übertragungsfunktion des Analogfilters.

Die einzelnen Implementierungsschritte umfassen dabei:

- Auswahl des Filtertyps, z.B. Butterworth oder Chebyshev
- Wahl der Entwurfsparameter (ω_c , Dämpfung, u.a.),
- Pole mit den Gleichungen für ϕ_n , γ und p_n bestimmen (siehe Vorlesung Thema1-Filter)

Nachfolgend soll der IIR-Filterentwurf für den Typ Butterworth und Chebyshev durchgeführt und diskutiert werden. Die Gleichungen zur Bestimmung der Pole werden im Foliensatz zu Thema1-Filter erwähnt. In der Anwendung macht es mehr Sinn, die entsprechenden Funktionen aus dem Modul [scipy.signal](#) zu benutzen.

Importieren Sie zunächst alle für dieses Notebook nötigen externen Module:

```
# Lösung

# Importieren Sie aus dem scipy-Modul die Teilbibliothek "signal"
# Importieren Sie aus dem matplotlib-Modul die Teilbibliothek "pyplot" mit dem Alias "plt"
# Importieren Sie das numpy-Modul mit dem Alias "np"

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

2. Butterworth-Filter

Butterworthfilter besitzen einen maximal steilen Übergang zwischen Durchlass- und Sperrbereich bei absolut glattem Amplitudengang im Durchlassbereich (kein Überspringen). Ein analoger Butterworth-Filter besitzt keine Nullstellen (sog. "All-Pole-Filter"), bei der Transformation zum Digitalfilter kommen aber - abhängig von der Transformation (Bilinear, Pol-Nullstellen-Abbildung z.B.) - Nullstellen hinzu.

In diesem Abschnitt soll ein Bandpassfilter mittels `signal.butter()` entworfen. Die Abtastfrequenz f_s beträgt 1000 Hz. Frequenzkomponenten unter 130 Hz und über 440 Hz sollen herausgefiltert werden, d.h., die Grenzfrequenzen f_{lower} und f_{upper} des Bandpassfilters betragen 130 Hz bis 440 Hz. Das Passbandintervall ist dann $f_{\text{pass}} = [130, 440]$ Hz. Die Einheit der Frequenzgrenzen in `signal.butter()` richten sich nach der Einheit der Abtastfrequenz, die als Argument übergeben wird. Oft werden sämtliche Frequenzen in Ω/π angegeben, wodurch die Frequenzachse von 0 Hz bis zur Nyquistfrequenz ($f_s/2$) auf den Bereich $[0, 1]$ normiert ist. z.T. es ist aber übersichtlicher, wenn die Frequenzachse weiterhin in Hz oder rad/s dargestellt ist. Unabhängig von der Wahl der Einheit für die Frequenzachse können die Filterpolynome jederzeit in eine andere Einheit skaliert werden, d.h. es ist nur wichtig, dass Abtast- und Grenzfrequenz(en) zusammenpassen. Gerade bei höheren Filterordnungen und hohen Abtastfrequenzwerten kann es zu numerischen Problem kommen, da die Polynomkoeffizienten schnell explodieren.

Zur graphischen Darstellung werden folgende Objekte verwendet:

1. `signal.freqz()` kann dann der Frequenzgang des Filters aus dem Zähler- und Nennerpolynom aus `signal.butter()` berechnet werden.
2. `signal.tf2zpk` rechnet das Zähler- und Nennerpolynom in Pol- und Nullstellen um.

```
'''
Beispiel: Butterworth Bandpassfilterentwurf mit verschiedenen Ordnungen
'''

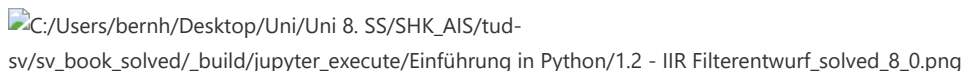
for filterOrder in (2, 10, 20): # Ordnung

    # Parameter
    fs_Hz = 1000                # Abtastfrequenz
    fPass_Hz = [130/fs_Hz, 440/fs_Hz] # Grenzfrequenzen normalisiert

    # Filterentwurf
    a, b = signal.butter(filterOrder, fPass_Hz, 'Bandpass', analog=False, fs=fs_Hz/fs_Hz)
# Filterkoeffizienten
    f_Hz, H = signal.freqz(a, b, fs=fs_Hz) # Frequenzgang des Filters
    z, p, k = signal.tf2zpk(a, b)          # Pol-Nullstellen-Verteilung
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Frequenzgang des Butterworth-Filters mit Ordnung=%d' %filterOrder)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|H(e^{j\Omega})|')
plt.plot(f_Hz, abs(H))

plt.subplot(122)
plt.title('Null- und Polverteilung des Butterworth-Filters mit Ordnung=%d'
%filterOrder)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='g', lw=0.2) # Erzeugung des Einheitskreises
plt.plot(np.real(p), np.imag(p), 'x', label=u"Pol")    # Polstellen
plt.plot(np.real(z), np.imag(z), 'o', label=u"Null")    # Nullstellen
plt.axis("equal")
plt.legend(loc="upper left")                          # Erzeugung einer Legende
plt.gcf().set_size_inches(15, 6)
plt.show()
```

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR Filterentwurf_solved_8_0.png

Zusätzlich gibt es die Funktion `signal.buttord()` zur Auswahl einer geeigneten Filterordnung in Abhängigkeit von festgelegten Randbedingungen bezüglich der Dämpfung im Durchlass- und Sperrbereich. Nachfolgend soll ein Bandpassfilter mittels dieser Funktion entworfen werden, damit die

Dämpfung im Durchlassbereich innerhalb von 10 dB von 140 Hz bis 430 Hz liegt, während sie im Sperrbereich (außerhalb von [130, 440] Hz) mindestens -40 dB beträgt.


```
'''
Beispiel: Butterworth Bandpassfilterentwurf mittels Ordnungselektion
'''

# Ordnungselektion
fs_Hz = 1000 # Abtastfrequenz
optOrder, WPass_rad = signal.buttord([140/fs_Hz, 430/fs_Hz], [130/fs_Hz, 440/fs_Hz], 10,
40, False, fs_Hz/fs_Hz)

# Filterentwurf
b, a = signal.butter(optOrder, WPass_rad, 'Bandpass', False, fs=fs_Hz/fs_Hz) #
Filterkoeffizienten
f_Hz, H = signal.freqz(b, a, fs=fs_Hz) # Frequenzgang des Filters
z, p, k = signal.tf2zpk(b, a) # Null-Pol Verteilung

# Graphische Darstellung
plt.subplot(121)
plt.title('Frequenzgang des Butterworth-Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$|H(e^{j\Omega})|$')
plt.plot(f_Hz, abs(H))

plt.subplot(122)
plt.title('Null- und Polverteilung des Butterworth-Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='g', lw=0.2) # Einheitskreis
plt.plot(np.real(p), np.imag(p), 'x', label=u"Pol")
plt.plot(np.real(z), np.imag(z), 'o', label=u"Null")
plt.axis("equal")
plt.legend(loc="upper left")
plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR
Filterentwurf_solved_11_0.png

3

3. Chebyshev-Filter

Beim Chebyshev-Filter fällt die Amplitude im Übergangsbereich steiler ab als beim Butterworth-Filter. Dieser Vorteil wird durch eine gewisse (einstellbare) Welligkeit im Durchlassbereich erkauft (unterhalb der Grenzfrequenz).

Ähnlich wie Butterworth-Filterentwurf, gibt es auch für den Chebyshev-Filterentwurf Funktionen zur Ordnungselektion, daher können diese ganz analog dazu verwendet werden. Alle Details können Sie unter [scipy.signal](#) finden.

Außerdem gibt es eine allgemeine Funktion [signal.iirfilter\(\)](#) zum IIR-Filterentwurf. Im Nachfolgenden sollen diese Funktionen benutzt werden, um einen Chebyshev-Typ-I und -II Digitalfilter mit einer Abtastfrequenz von $f_s = 8000$ Hz zu entwerfen. Chebyshev Filter Typ I weisen eine Welligkeit im Passband auf, während Typ II Filter die Welligkeit im Stopband haben.

- **Chebyshev Typ I**

Entwerfen Sie eine for-Schleife zur Berechnung von Chebyshev-Hochpassfiltern des Typ I mit den unterschiedlichen Ordnung 2, 5 und 10 (wie es für den Butterworth-Filter in Kapitel 2 schon umgesetzt wurde). Die Grenzfrequenz f_c soll dabei 1500 Hz betragen und die Abtastfrequenz f_s 8000 Hz.

Hinweis: Der In- und Output der Objekte `signal.iirfilter()` und `signal.butter()` ähneln sich sehr. Zur Erzeugung des Chebyshev-HP-Filters benötigt es im Vergleich zu Kapitel 2 noch folgende Variableneinstellungen: (`btype='highpass'`, `ftype='cheby1'`, `rp=1`).


```
# Lösung
'''
Beispiel: Chebyshev Typ-I Hochpassfilterentwurf mit verschiedenen Ordnungen
'''

for n in (2, 5, 10): # Ordnung

    # Parameter
    fc_Hz = 1500 # Grenzfrequenz
    fs_Hz = 8000 # Abtastfrequenz

    # Filterentwurf
    a, b = signal.iirfilter(n, fc_Hz/fs_Hz, rp=1, btype='highpass', analog=False,
ftype='cheby1', fs=fs_Hz/fs_Hz)
    f_Hz, H = signal.freqz(a, b, fs=fs_Hz) # Frequenzgang
    z, p, k = signal.tf2zpk(a, b)          # Null-Pol Verteilung
```

```
# Plot
plt.subplot(121)
plt.plot(f_Hz, abs(H))
plt.title('Frequenzgang des Chebyshev-Typ-I-Filters mit Ordnung=%d' %n)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$|H(e^{j\Omega})|$')
plt.subplot(122)
plt.title('Null- und Polverteilung des Chebyshev-Typ-I-Filters mit Ordnung=%d' %n)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='g', lw=0.2) # Einheitskreis
plt.plot(np.real(p), np.imag(p), 'x', label=u"Pol")
plt.plot(np.real(z), np.imag(z), 'o', label=u"Null")
plt.axis("equal")
plt.legend(loc="upper left")
plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR
Filterentwurf_solved_14_0.png

Nachfolgend soll ein Chebyshev-Bandstopp-Filter vom Typ 1 entworfen werden. Wie in Kapitel 2 soll auch hier Ordnungsselektion durchgeführt werden. Dies wird mit dem Objekt `signal.cheb1ord()` umgesetzt. Die Abtastfrequenz soll wieder bei $f_s = 8000$ Hz liegen, dessen Dämpfung im Sperrbereich von 1590 Hz bis 2990 Hz mindestens 40 dB beträgt, während die Dämpfung im Durchlassbereich (außerhalb von [1500,3000] Hz) innerhalb von -10 dB liegen soll.

Hinweis: ändern Sie für `signal.iirfilter()` die Variablen wie folgendermaßen: (`btype='bandstop'`).


```
# Lösung
'''
Beispiel: Chebyshev Typ-I Bandsperrfilterentwurf mit Ordnungsselektion
'''

# Ordnungselektion
fs_Hz = 8000 # Abtastfrequenz
optOrder, WPass_rad = signal.cheb1ord([1500/fs_Hz, 3000/fs_Hz], [1590/fs_Hz, 2990/fs_Hz],
10, 40, False, fs_Hz/fs_Hz)

# Filterentwurf
a, b = signal.iirfilter(n, WPass_rad, rp=1, btype='bandstop', analog=False,
ftype='cheby1', fs=fs_Hz/fs_Hz)
f_Hz, H = signal.freqz(a, b, fs=fs_Hz) # Frequenzgang
z, p, k = signal.tf2zpk(a, b)          # Null-Pol Verteilung
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Frequenzgang des Chebyshev-Typ-I-Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$|H(e^{j\omega})|$')
plt.plot(f_Hz, abs(H))

plt.subplot(122)
plt.title('Null- und Polverteilung des Chebyshev-Typ-I-Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='g', lw=0.2) # Einheitskreis
plt.plot(np.real(p), np.imag(p), 'x', label=u"Pol")
plt.plot(np.real(z), np.imag(z), 'o', label=u"Null")
plt.axis("equal")
plt.legend(loc="upper left")
plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR Filterentwurf_solved_17_0.png

• Chebyshev Typ II

Der Chebyshev-Filter des Typ 2 soll nun genutzt werden, um einen Tiefpass mit der Grenzfrequenz $f_c = 1500$ Hz bei einer Abtastfrequenz von $f_s = 8000$ Hz zu entwerfen. Es sollen analog zum vorherigen Chebyshev-Filter-Entwurf 1. Typs wieder Filter mit 2., 5. und 10. Ordnung ausgegeben werden.

Hinweis: Ändern Sie für `signal.iirfilter()` die Variablen folgendermaßen: (`rs=20`, `btype='lowpass'`, `analog=False`, `ftype='cheby2'`).

```
# Lösung
'''
Aufgabe: Chebyshev Typ-II Tiefpassfilterentwurf mit verschiedenen Ordnungen
'''

for filterOrder in (2, 5, 10): # Ordnung


    # Parameter
    fc_Hz = 1500 # Grenzfrequenz
    fs_Hz = 8000 # Abtastfrequenz

    # Filterentwurf
    a, b = signal.iirfilter(filterOrder, fc_Hz/fs_Hz, rs=20, btype='lowpass',
analog=False, ftype='cheby2', fs=fs_Hz/fs_Hz)
    w, H = signal.freqz(a, b, fs=fs_Hz) # Frequenzgang
    z, p, k = signal.tf2zpk(a, b) # Null-Pol Verteilung
```

```
# Plot
plt.subplot(121)
plt.title('Frequenzgang des Chebyshev-Typ-II-Filters mit Ordnung=%d' %filterOrder)
plt.xlabel('Frequenz [rad/s]')
plt.ylabel('Amplitude')
plt.plot(w, abs(H))

plt.subplot(122)
plt.title('Null- und Polverteilung des Chebyshev-Typ-II-Filters mit Ordnung=%d' %filterOrder)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='g', lw=0.2) # Einheitskreis
plt.plot(np.real(p), np.imag(p), 'x', label=u"Pol")
plt.plot(np.real(z), np.imag(z), 'o', label=u"Null")
plt.axis("equal")
plt.legend(loc="upper left")

plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR Filterentwurf_solved_20_0.png

Auch hier soll noch einmal die Ordnungsselektion für den Chebyshev-Filter vom Typ 2 verwendet werden. Dafür soll ein Chebyshev-Bandpass-Filter vom Typ 2 entworfen werden. Dies wird mit dem Objekt `signal.cheb2ord()` umgesetzt. Die Abtastfrequenz soll bei $f_s = 8000$ Hz liegen, dessen Dämpfung im Durchlassbereich von 1590 Hz bis 2990 Hz innerhalb von 10 dB, während sie im Sperrbereich (außerhalb von [1500,3000] Hz) mindestens -40 dB beträgt.

Hinweis: ändern Sie für `signal.iirfilter()` die Variablen wie folgendermaßen: `(btype='bandpass')`.


```
# Lösung
'''
Aufgabe: Chebyshev Typ-II Bandpassfilterentwurf mit Ordnungsselektion
'''

# Ordnungsselektion
fs_Hz = 8000 # Abtastfrequenz
optOrder, wPass_rad = signal.cheb2ord([1590/fs_Hz, 2990/fs_Hz], [1500/fs_Hz, 3000/fs_Hz],
10, 40, False, fs_Hz/fs_Hz)

# Filterentwurf
a, b = signal.iirfilter(optOrder, wPass_rad, rs=20, btype='bandpass', analog=False,
ftype='cheby2', fs=fs_Hz/fs_Hz)
f_Hz, H = signal.freqz(a, b, fs=fs_Hz) # Frequenzgang
z, p, k = signal.tf2zp(a, b) # Null-Pol Verteilung
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Frequenzgang des Chebyshev-Typ-II-Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Frequenz [rad/s]')
plt.ylabel('Amplitude')
plt.plot(f_Hz, abs(H))

plt.subplot(122)
plt.title('Null- und Polverteilung des Chebyshev-Typ-II-Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='g', lw=0.2) # Einheitskreis
plt.plot(np.real(p), np.imag(p), 'x', label=u"Pol")
plt.plot(np.real(z), np.imag(z), 'o', label=u"Null")
plt.axis("equal")
plt.legend(loc="upper left")
plt.gcf().set_size_inches(15, 6)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR Filterentwurf_solved_23_0.png



4. Vorverzerrung

Wenn ein analoger Filter $H_{lp}(s)$ mit einer festgelegten Grenzfrequenz mittels der Bilineartransformation in einen digitalen Filter $H_{d,lp}(z)$ überführt werden soll, wird diese durch die Achsenverzerrung von der s - in die z -Ebene verzerrt und die Grenzfrequenz des Digitalfilters ist u.U. deutlich daneben (die Verzerrung wird stärker für höhere Grenzfrequenzen). Aus diesem Grund wird die analoge Grenzfrequenz zunächst *vorverzerrt*, danach die Filterkoeffizienten berechnet und anschließend der analoge Filter in einen digitalen Filter transformiert. Für die Transformation in den z -Bereich kann die `signal.cont2discrete()` Funktion verwendet werden.

Im Nachfolgenden wird zunächst ein analoger Butterworth-Filter 3. Ordnung erstellt mit der bereits kennengelernten Funktion `signal.butter()`, wobei die `analog=True` flag gesetzt werden muss. Zum Plotten des analogen Frequenzganges eignen sich die `signal.TransferFunction()` und `signal.bode()` Funktionen.

Für die Vorverzerrung wird die neue Grenzfrequenz $\omega_c'(\text{prime})$ über den in der Vorlesung kennengelernten Ausdruck
$$\omega_c' = 2f_s \cdot \tan\left(\frac{\omega_c}{2 \cdot f_s}\right)$$
 berechnet.

```
'''
Beispiel für das Vorverzerren und der anschließenden Transformation vom Analog- zum
Digitalfilter
'''

fs_Hz = 2000          # Samplingfrequenz
ws_rad_s = 2*np.pi*fs_Hz # Samplingfrequenz
dt = 1/fs_Hz          # Abtastintervall
fc_Hz = 500           # Grenzfrequenz des Filters
wc_rad_s = fc_Hz*2*np.pi # Grenzfrequenz des Filters
Wc_rad = wc_rad_s/fs_Hz # Normalisierte Grenzfrequenz
# Inline function für Umrechnung der vorverzerzten Frequenz VOR der Transformation:
analogFreqPreWarping = lambda wc_rad_s, fs_Hz: 2*fs_Hz*np.tan(wc_rad_s/(fs_Hz*2))

# Grenzfrequenz vorverzerren:
wcPw_rad_s = analogFreqPreWarping(wc_rad_s, fs_Hz)
print("{} {} {}".format("Vorverzernte Grenzfrequenz: ", wcPw_rad_s, "rad/s"))


# Koeffizienten eines Butterworth-Filters berechnen:
filterOrder = 3
a, b = signal.butter(filterOrder, wcPw_rad_s, 'lowpass', analog=True) #
Filterkoeffizienten des analogen Tiefpasses
Hlp = signal.TransferFunction(a, b) # Systemfunktion des Analogfilters

# Analog zu diskret Transformation des Tiefpasses:
ad, bd, dt = signal.cont2discrete((a, b), 1/fs_Hz, 'bilinear')
ad = ad.flatten() # sonst spinnen die Dimensionen von ad...
```

Vorverzernte Grenzfrequenz: 3999.999999999995 rad/s

```
# Plotten der Frequenzgänge:
wPlot_rad_s = np.arange(0, fs_Hz*2*np.pi, 1) # Frequenzachse zum Plotten
w, mag, phase = Hlp.bode(w=wPlot_rad_s) # Frequenzgang des Analogfilters
wd, Hdlp = signal.freqz(ad, bd, fs=fs_Hz) # Frequenzgang des Digitalfilters

plt.subplot(121)
plt.plot(wPlot_rad_s/(2*np.pi), 10**(mag/20), '-k')
plt.plot([wc_rad_s/(2*np.pi), wc_rad_s/(2*np.pi)], [0, 1], '-b')
plt.plot([0, ws_rad_s/(2*np.pi)], [1/np.sqrt(2), 1/np.sqrt(2)], '--b')
plt.title('Analoger Butterworth-Tiefpassfilter %d. Ordnung' %filterOrder)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('Amplitude')
plt.subplot(122)
plt.plot(wd/fs_Hz*2*np.pi, abs(Hdlp), '-k')
plt.plot([Wc_rad, Wc_rad], [0, 1], '-b')
plt.plot([0, np.pi], [1/np.sqrt(2), 1/np.sqrt(2)], '--b')
plt.title('Digitaler Butterworth-Tiefpassfilter %d. Ordnung' %filterOrder)
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.subplots_adjust(left=0.1, right=2, top=0.9)
```

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.2 - IIR
Filterentwurf_solved_26_0.png

Im ersten Fall (unverzerrt) sollte der Amplitudenfrequenzgang des Analogfilters bei der Grenzfrequenz $\omega_c \frac{1}{\sqrt{2}}$ betragen, die äquivalente Grenzfrequenz des Digitalfilters aber kleiner als gewollt sein. Wenn der Digitalfilter dagegen mit der vorverzernten Grenzfrequenz $\omega_{c,pw}$ berechnet wird, ist die Grenzfrequenz des Analogfilters höher als gewollt (was in diesem Falle egal ist, da ja der Digitalfilter interessiert), die des Digitalfilters passt aber genau.

References

1. Titelbild von [Avon Ampo](#)
2. DSP Guide: [The Scientist and Engineer's Guide to Digital Signal Processing](#)
3. Python-Modul von Signalverarbeitung: [scipy.signal](#)

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)

1.3 - Anwendungsbeispiel



Diese Notebook baut auf die beiden Vorherigen auf, in denen der FIR und IIR Filterentwurf mit verschiedenen Methoden ausführlich behandelt wurde.

Im Folgenden soll ein Praxisbeispiel für einer Filteranwendung mit konkreten Randbedingungen bearbeitet werden. Auf ein Audiosignal (akustik.wav, zu finden im Ordner 'data') sollen zunächst Störsignale aufaddiert werden, die anschließend mit verschiedenen Filtern wieder entfernt werden, um das ursprüngliche Signal wiederherzustellen. Dabei sollen die Vor- und Nachteile der verschiedenen Filtertypen aufgezeigt werden.

Außerdem sollen einige grundlegende Python Module zur Verarbeitung speziell von Audiosignalen vorgestellt und angewendet werden.

Inhalt



1. Audiosignale in Python



2. FIR-Filterung



3. IIR-Filterung



1. Audiosignale in Python

1.1 Import von Audiodateien

Um ein Audiosignal zu bearbeiten, muss es zunächst geladen/importiert werden. Dafür werden drei verschiedene Methoden vorgestellt:

```
path = 'data/akustik.wav'

# 1. wavfile:
from scipy.io import wavfile
sr, audio = wavfile.read(path)

# 2. wave:
import wave
wf = wave.open(path, 'rb')
audio = wf.readframes(1024)

# 3. Librosa:
import librosa
audio, sr = librosa.load(path)
```

In diesem Notebook verwenden wir die erste Option `wavfile()`, da es schon in dem Modul `scipy` integriert und somit installiert ist. Schauen Sie sich nun die Audioaufnahme `data/akustik.wav` im Zeit- und Frequenzbereich an. Importieren Sie dafür zuerst die dafür notwendigen Module:

```
# Lösung
# Importieren Sie aus scipy die Teilbibliothek "fftpack"
# Importieren Sie aus scipy.io die Teilbibliothek "wavfile"
# Importieren Sie aus matplotlib die Teilbibliothek "pyplot" mit dem Alias "plt"
# Importieren Sie das numpy-Modul mit dem Alias "np"

import numpy as np
import matplotlib.pyplot as plt
from scipy import fftpack
from scipy.io import wavfile
```


```
# Lösung
'''
Beispiel: Audiodatei einlesen und visualisieren
'''

# Datei einlesen
fs_Hz, audioSignal = wavfile.read('data/akustik.wav') # Sample Rate, Audiosignal im
Array-Form
audioSignal = audioSignal/np.max(np.abs(audioSignal)) # Normalisierung

# Signalparameter:
signalLength = len(audioSignal) # Länge des Audiosignals
T_s = signalLength/fs_Hz - 1/fs_Hz # Zeit
t_s = np.linspace(0, T_s, signalLength) # Zeitbereich
f_Hz = np.linspace(0, fs_Hz/2, signalLength//2) # Frequenzbereich

# Spectrale
audioSignal_fft = fftpack.fft(audioSignal)
```

```
# plot
plt.subplot(121)
plt.title('Audiosignal im Zeitbereich')
plt.xlabel('Zeit [s]')
plt.ylabel('Signalamplitude')
plt.plot(t_s, audioSignal)
plt.subplot(122)
plt.title('%d Punkte FFT des Audiosignals' %signalLength)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|X(f)|')
plt.ylim(0, 200)
plt.plot(f_Hz, np.abs(audioSignal_fft[:int(signalLength/2)]))
plt.gcf().set_size_inches(15, 5)
plt.show()
```

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/1.3 -
Anwendungsbeispiel_solved_5_0.png

1.2 Abspielen von Audiodateien

Die Ausgabe von Audiosignalen kann in Python auch mit unterschiedlichen externen Modulen realisiert werden. Falls Sie das Modul nicht installiert haben, können Sie dies via `pip install "module"` durchführen. Hier werden Ihnen drei mögliche Module mit Ihren individuellen Vorzügen vorgestellt:

- [playsound](#)

Das PlaySound-Modul ist ein plattformübergreifendes Modul, das Audiodateien abspielen kann.

```
from playsound import playsound
```

```
playsound('data/akustik.wav')
```



```
Error 259 for command:
    play data/akustik.wav wait
Unbekannter Befehlsparameter.
```

```
-----
PlaysoundException                                Traceback (most recent call last)
Cell In [5], line 1
----> 1 playsound('data/akustik.wav')

File ~\AppData\Local\Programs\Python\Python310\lib\site-packages\playsound.py:73, in
_playsoundWin(sound, block)
    71     logger.debug('Starting')
    72     winCommand(u'open {}'.format(sound))
--> 73     winCommand(u'play {}'.format(sound, ' wait' if block else ''))
    74     logger.debug('Returning')
    75 finally:

File ~\AppData\Local\Programs\Python\Python310\lib\site-packages\playsound.py:64, in
_playsoundWin.<locals>.winCommand(*command)
    60     exceptionMessage = ('\n    Error ' + str(errorCode) + ' for command:'
    61                        '\n                ' + command.decode('utf-16') +
    62                        '\n                ' + errorBuffer.raw.decode('utf-16').rstrip('\0'))
    63     logger.error(exceptionMessage)
--> 64     raise PlaysoundException(exceptionMessage)
    65     return buf.value

PlaysoundException:
    Error 259 for command:
        play data/akustik.wav wait
Unbekannter Befehlsparameter.
```

- [IPython.display](#)

Mit diesem Modul wird die Audiodatei mit einem Audioplayer geöffnet, der anschließend per Klick gestartet werden kann:

```
import IPython.display as ipd
```

```
ipd.Audio('data/akustik.wav')
```

- [simpleaudio](#)

Das Modul ermöglicht es, sowohl wav-Dateien als auch NumPy-Arrays abgespielt zu können. Diese Eigenschaft ist für dieses Notebook sehr nützlich, weshalb im Weiteren auf `simpleaudio` zurückgegriffen wird.

```
import simpleaudio as sa
```

```
# Beispiel: wav-Dateien abspielen

# Datei einlesen
wave_obj = sa.WaveObject.from_wave_file('data/akustik.wav')

# abspielen
play_obj = wave_obj.play()
play_obj.wait_done()
```

```
# Beispiel: NumPy-Arrays (Audiosignal) abspielen

# Datei ablesen
fs_Hz, audioSignal = wavfile.read('data/akustik.wav')

# abspielen
play_obj = sa.play_buffer(audioSignal, 1, 2, fs_Hz)
play_obj.wait_done()
```

```
# Beispiel: NumPy-Arrays (Sinussignal) abspielen

# Sinussignal erzeugen
t_s = np.linspace(0, 3, 24000)
sine = np.sin(440 * 2 * np.pi * t_s)

# Werte im 16-Bit-Bereich konvertieren
sound = sine * (2**15 - 1) / np.max(sine)
sound = sound.astype(np.int16)

# abspielen
play_obj = sa.play_buffer(sound, 1, 2, 8000)
play_obj.wait_done()
```

1.3 Signale bearbeiten / Störsignal erstellen

Wenn das Audiosignal als numpy-Array vorliegt, kann das Signal sehr leicht bearbeitet werden. Zuerst soll dafür auf die Audiodatei `akustik.wav` ein Sinus mit der Frequenz von 440 Hz addiert werden. Dadurch erhält die Audiodatei ein Störsignal, welches Sie danach wieder davon zu entfernen versuchen. Da gleich das Signal zudem gefenstert werden soll, wird noch das Modul `signal` importiert, um daraus `get_window()` zu verwenden.

```
# Lösung
# Importieren Sie aus scipy die Teilbibliothek "signal"
from scipy import signal, fft
```

```
# Lösung
'''
Beispiel: Signal addieren, visualisieren und abspielen
'''

# Datei einlesen (originales Audiosignal) und Amplitude normalisieren:
fs_Hz, audioSignal = wavfile.read('data/akustik.wav')
audioSignal = audioSignal / np.max(np.abs(audioSignal))

# Initiale Daten
fsin_Hz = 440 # Sinusfrequenz
signalLength = len(audioSignal) # Länge des Audiosignals
T_s = signalLength / fs_Hz - 1 / fs_Hz # Zeitraum
t_s = np.linspace(0, T_s, signalLength) # Zeitbereich
f_Hz = np.linspace(0, fs_Hz / 2, signalLength / 2) # Frequenzbereich

# Sinussignale erstellen
sine = 1 / 2 * np.sin(fsin_Hz * 2 * np.pi * t_s)
sine_fft = fftpack.fft(sine)

# Signal zur Filterung
corruptedAudioSignal = audioSignal + sine # Additives Signal
corruptedAudioSignal_fft = fftpack.fft(corruptedAudioSignal) # Frequenzgang

# Fensterung (optional)
window = signal.get_window('hamming', signalLength)
corruptedAudioSignal_windowed = corruptedAudioSignal * window
```

```
# plot
plt.subplot(221)
plt.title('Störsignal im Zeitbereich')
plt.xlabel('Zeit [s]')
plt.xlim(0, 0.1)
plt.ylabel('Amplitude')
plt.ylim(-1, 1)
plt.plot(t_s, sine)
plt.subplot(222)
plt.title('Störsignal im Frequenzbereich')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.ylim(0, 500)
plt.plot(f_Hz, np.abs(sine_fft[:signalLength//2]))
plt.subplot(223)
plt.title('Audiosignal mit Störung im Zeitbereich')
plt.xlabel('Zeit [s]')
plt.ylabel('Amplitude')
plt.plot(t_s, corruptedAudioSignal_windowed)
plt.subplot(224)
plt.title('Audiosignal mit Störung im Frequenzbereich')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.ylim(0, 500)
plt.plot(f_Hz, np.abs(corruptedAudioSignal_fft[:signalLength//2]))
plt.gcf().set_size_inches(15, 10)
plt.show()
```

```
# Lösung
# Abspielen
sound = (corruptedAudioSignal * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()
```

```
# Lösung
# Abspielen
sound = (corruptedAudioSignal_windowed * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_windowed))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()
```

Mit `corruptedAudioSignal` beziehungsweise `corruptedAudioSignal_windowed` wurde jetzt ein mit einem Sinuston gestörtes Audiosignal erzeugt. Im Folgenden wird nun versucht, mittels der in Notebook 1.1 und 1.2 kennengelernten Filter diese Störung zu filtern und wieder ein dem Original ähnliches Audiosignal zu rekonstruieren.



2. FIR-Filterung

2.1 FIR-Filter mittels IDFT

Wie in Kapitel 1.1 wird zuerst ein Bandsperrfilter mittels IDFT verwendet und entworfen, um das Sinussignal aus dem Audiosignal zu entfernen. Der Filter soll dabei eine Bandbreite von 60 Hz haben.

```
# Lösung

'''
Beispiel: Entfernen von Signalen mittels selbst definierten FIR-Bandsperrfilters
'''

# Berechnung des FIR Filters:

bw_Hz = 100 # Bandbreite des Sperrbereiches
fcLower_Hz = fsin_Hz - bw_Hz/2
fcUpper_Hz = fsin_Hz + bw_Hz/2
fftLength = 2048
f_Hz = np.linspace(0, fs_Hz, fftLength)
# Definition des idealen Frequenzganges:
H = np.where((f_Hz > fcLower_Hz) & (f_Hz < fcUpper_Hz)
             | (f_Hz > fs_Hz - fcUpper_Hz) & (f_Hz < fs_Hz - fcLower_Hz), 0, 1)
# IFFT und shift:
h = np.fft.ifftshift(fftpack.iff(H, fftLength))

# Filterkern aus der Impulsantwort ausschneiden und Fenster:
filterLength = 1500;
window = signal.get_window('hanning', filterLength+1)
filterKernel = np.real(h[(fftLength - filterLength)//2:(fftLength + filterLength)//2 +
1])*window
k = np.arange(0, filterKernel.size)

# Synthese des ist-Filters:
H_ist = np.abs(np.fft.fft(filterKernel, fftLength))
```

```
# Graphische Darstellung
plt.subplot(131)
plt.title('Idealer Amplitudenfrequenzgang \n des Bandpasses')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$H(e^{j\Omega})$')
plt.xlim([0,1200])
plt.plot(f_Hz, H)
plt.subplot(132)
plt.title('Maskierte und gefensterte \n Impulsantwort des Bandsperrfilters')
plt.xlabel('Sample Index k')
plt.stem(k, filterKernel, use_line_collection=True)
plt.subplot(133)
plt.title('Tatsächlicher Amplitudenfrequenzgang \n des Bandsperrfilters')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$|H(e^{j\Omega})|$')
plt.plot(f_Hz, H_ist)
plt.xlim([0,1200])
plt.subplots_adjust(left=0.1, right=2, top=0.9)
```

```
# Lösung

# Filterung
corruptedAudioSignal_filtered = signal.convolve(corruptedAudioSignal, filterKernel,
'same') # Faltung
corruptedAudioSignal_filtered_fft = fftpack.fft(corruptedAudioSignal_filtered)
```

```
# Plot
fPlot_Hz = np.linspace(0, fs_Hz, signalLength)
plt.subplot(121)
plt.title('Audiosignal nach Bandsperrfilterung')
plt.xlabel('Zeit [s]')
plt.ylabel('Signalamplitude')
plt.plot(t_s, corruptedAudioSignal_filtered)
plt.subplot(122)
plt.title('Frequenzgang nach Filterung')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('Amplitude')
plt.ylim(0, 200)
plt.plot(fPlot_Hz[:signalLength//2],
np.abs(corruptedAudioSignal_filtered_fft[:signalLength//2]))
plt.gcf().set_size_inches(15, 10)
plt.show()

# Abspielen
sound = (corruptedAudioSignal_filtered * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_filtered))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()
```

2.2 FIR-Filter mittels **firwin()**

Für solche Anwendungen können wir aber auch die vorhandenen Filter wie `signal.firwin()` in `scipy.signal` direkt benutzen. Ändern Sie dabei die Ordnung (sie können dafür auch eine for-Schleife mit logarithmisch ansteigender Ordnungszahl implementieren), bis das Ergebnis dem der IDFT ähnelt. Hinweis: die Ordnungszahl n kann gerne größer 1000 sein.

```
# Lösung
'''
Aufgabe: Entfernen von Signalen mittels signal.firwin()
'''
# Parameter
bw_Hz = 60 # Bandbreite
filterOrder = 5001 # Ordnung
wPass_Hz = [(fsin_Hz-bw_Hz/2), (fsin_Hz+bw_Hz/2)]

# Filterung mit Fenster
h_fir = signal.firwin(filterOrder, wPass_Hz, window='hanning', fs=fs_Hz)
H_fir = fftpack.fft(h_fir, signalLength) # Spektral des Filters
corruptedAudioSignal_filtered = signal.convolve(corruptedAudioSignal_windowed, h_fir,
'same') # Faltung
corruptedAudioSignal_filtered_fft = fftpack.fft(corruptedAudioSignal_filtered) # Spektral
des Signals
```

```
# plot
fPlot_Hz
plt.subplot(221)
plt.title('Impulsantwort des Filters')
plt.xlabel('Sample Nummer')
plt.ylabel('Amplitude')
plt.plot(h_fir)
plt.subplot(222)
plt.title('Spektral des Filters')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.plot(fPlot_Hz[:signalLength//2], np.abs(H_fir[:signalLength//2]))
plt.subplot(223)
plt.title('Audiosignal nach Bandsperrfilterung')
plt.xlabel('Zeit [s]')
plt.ylabel('Amplitude')
plt.plot(t_s, corruptedAudioSignal_filtered)
plt.subplot(224)
plt.title('Frequenzgang nach Filterung')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.ylim(0, 200)
plt.plot(fPlot_Hz[:signalLength//2],
np.abs(corruptedAudioSignal_filtered_fft[:signalLength//2]))
plt.gcf().set_size_inches(15, 10)
plt.show()

# Abspielen
sound = (corruptedAudioSignal_filtered * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_filtered))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()
```

Um einen Bandsperrfilter mit hohem Knick und engem Frequenzbereich zu erzeugen, um nur das Sinussignal zu entfernen, benötigt der FIR-Filter eine große Ordnungszahl. Das bedeutet, dass man für die Umsetzung des Filters viel Speicherplatz benötigt und einen großen Rechenaufwand hat, was für eine niedrige Effizienz spricht.

Um zu analysieren, ob IIR-Filter eine bessere Alternative darstellen, wenn es um effizientes Filterdesign geht, soll nun im nächsten Kapitel dasselbe Signal `corruptedAudioSignal` bzw. `corruptedAudioSignal_windowed` mittels der in Notebook 1.2 kennengelernten Filter und Filterdesign bereinigt werden.



3. IIR-Filterung

3.1 Butterworth Filter

Zunächst soll ein Butterworth-Filter mit 10. Ordnung und einer Bandbreite $bw = 60$ Hz entworfen werden. Dazu wird, wie in Kapitel 1.2, das Objekt `signal.butter()` zur Erzeugung der Filterkoeffizienten verwendet. Geben Sie sich nun aber mittels `output=sos` die "second-order sections" aus, da mit diesen diect über das Objekt `signal.sosfilt()` eine Filterung des Signals durchgeführt werden kann.

```
# Lösung
'''
Beispiel: Entfernen von Signalen mittels IIR-Bandsperrfilters (Butterworth)
'''
# Parameter
bw_Hz = 60 # Bandbreite des Filters
filterOrder = 10 # Ordnung des IIR-Filters
wPass_Hz = [fsin_Hz-bw_Hz/2, fsin_Hz+bw_Hz/2] # Frequenzbereich des Sperrbands

# Filterung
sos = signal.butter(filterOrder, wPass_Hz, 'bs', analog=False, fs=fs_Hz, output='sos')
w, H_butt = signal.sosfreqz(sos, int(signalLength/2), fs=fs_Hz)

corruptedAudioSignal_filtered = signal.sosfilt(sos, corruptedAudioSignal)
corruptedAudioSignal_filtered_fft = fftpack.fft(corruptedAudioSignal_filtered)
```

```
# Graphische Darstellung
plt.subplot(311)
plt.title('Frequenzgang des Butterworth Filters mit Ordnung=%d' % filterOrder)
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.plot(fPlot_Hz[:signalLength//2], np.abs(H_butt[:int(signalLength/2)]))

plt.subplot(312)
plt.title('Audiosignal nach Filterung')
plt.xlabel('Zeit [s]')
plt.ylabel('Amplitude')
plt.plot(t_s, corruptedAudioSignal_filtered)

plt.subplot(313)
plt.title('Frequenzgang nach Filterung')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.ylim(0, 200)
plt.plot(fPlot_Hz[:signalLength//2], np.abs(corruptedAudioSignal_filtered_fft[:signalLength//2]))

plt.gcf().set_size_inches(8, 16)
plt.show()

# Abspielen
sound = (corruptedAudioSignal_filtered * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_filtered))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()
```

Um nun den Butterworth-Filter mit Spezifikationen zu erzeugen, soll nun der Filterentwurf mit Ordnungsselektion durchgeführt werden. Dazu wird das Objekt `signal.buttord()` verwendet. Dessen Dämpfung soll im Sperrbereich von $fsin_Hz - bw$ bis $fsin_Hz + bw$ Hz mindestens 60 dB betragen, während die Dämpfung im Durchlassbereich (außerhalb von $[fsin_Hz - bw - 10, fsin_Hz + bw + 10]$ Hz) innerhalb von -10 dB liegen soll. Die Bandbreite soll dabei wieder $bw = 60$ Hz betragen.

```
# Lösung
'''
Aufgabe: Entfernen von Signalen mittels IIR-Bandsperrfilters (Butterworth)
'''
# Ordnungsselektion
bw_Hz = 60 # Bandbreite
optOrder, wPass_rad = signal.buttord([fsin_Hz-bw_Hz/2-10, fsin_Hz+bw_Hz/2+10], [fsin_Hz-bw_Hz/2, fsin_Hz+bw_Hz/2], 10, 60, False, fs_Hz)

# Erzeugen des Bandsperrfilters
sos = signal.butter(optOrder, wPass_rad, 'bs', False, 'sos', fs_Hz)
w, H_butt = signal.sosfreqz(sos, int(signalLength/2), fs=fs_Hz)

# Filterung
corruptedAudioSignal_filtered = signal.sosfilt(sos, corruptedAudioSignal)
corruptedAudioSignal_filtered_fft = fftpack.fft(corruptedAudioSignal_filtered)
```

```

# plot
plt.subplot(311)
plt.title('Frequenzgang des Butterworth Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.plot(fPlot_Hz[:signalLength//2], np.abs(H_butt[:int(signalLength/2)]))

plt.subplot(312)
plt.title('Audiosignal nach Filterung')
plt.xlabel('Zeit [s]')
plt.ylabel('Amplitude')
plt.plot(t_s, corruptedAudioSignal_filtered)

plt.subplot(313)
plt.title('Frequenzgang nach Filterung')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.ylim(0, 200)
plt.plot(fPlot_Hz[:signalLength//2],
np.abs(corruptedAudioSignal_filtered_fft[:signalLength//2]))
plt.gcf().set_size_inches(8, 16)
plt.show()

# Abspielen
sound = (corruptedAudioSignal_filtered * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_filtered))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()

```

Im Titel der ersten Graphik wird die Ordnungszahl ausgegeben. Diese ist um mindestens 2 Zehnerpotenzen geringer als die des FIR-Filters. Das bedeutet, dass man für die Umsetzung des IIR-Filters viel weniger Speicherplatz und Rechenleistung benötigt, was die Anwendung des IIR-Filters in diesem Fall viel effizienter macht.

3.2 Chebyshev Filter

Entwerfen Sie zum Schluss noch einen Chebyshev-Filter vom Typ 2 zur Bereinigung des Audiosignals. Auch hier soll die Ordnungsselektion angewendet werden, wofür das Objekt [signal.cheb2ord\(\)](#) zur Verfügung steht.

Die Bedingungen an den Filter sind wie in 3.1: Die Dämpfung soll im Sperrbereich von $f_{\text{sin_Hz}} - bw$ bis $f_{\text{sin_Hz}} + bw$ Hz mindestens 60 dB betragen, während die Dämpfung im Durchlassbereich (außerhalb von $[f_{\text{sin_Hz}} - bw - 10, f_{\text{sin_Hz}} + bw + 10]$ Hz) innerhalb von -10 dB liegen soll.

Die Bandbreite soll dabei wieder $bw = 60$ Hz betragen.

Für die Berechnung des Filters soll nun aber das Objekt [signal.cheby2\(\)](#) verwendet werden.

```

# Lösung
'''
Aufgabe: Entfernen von Signalen mittels IIR-Bandsperrfilters (Chebyshev II mit
Ordnungsselektion)
'''

# Ordnungsselektion
bw_Hz = 60 # Bandbreite
optOrder, wPass_rad = signal.cheb2ord([fsin_Hz-bw_Hz/2-10, fsin_Hz+bw_Hz/2+10], [fsin_Hz-
bw_Hz/2, fsin_Hz+bw_Hz/2], 10, 60, False, fs_Hz)

# Erzeugen des Bandsperrfilters
sos = signal.cheby2(optOrder, 60, wPass_rad, 'bs', False, 'sos', fs_Hz)
w, H_cheby = signal.sosfreqz(sos, int(signalLength/2), fs=fs_Hz)

# Filterung
corruptedAudioSignal_filtered = signal.sosfilt(sos, corruptedAudioSignal)
corruptedAudioSignal_filtered_fft = fftpack.fft(corruptedAudioSignal_filtered)

```

```

# plot
plt.subplot(311)
plt.title('Frequenzgang des Chebyshev-II Filters mit Ordnung=%d' %optOrder)
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.plot(fPlot_Hz[:signalLength//2], np.abs(H_butt[:int(signalLength/2)]))

plt.subplot(312)
plt.title('Audiosignal nach Filterung')
plt.xlabel('Zeit [s]')
plt.ylabel('Amplitude')
plt.plot(t_s, corruptedAudioSignal_filtered)

plt.subplot(313)
plt.title('Frequenzgang nach Filterung')
plt.xlabel('Frequenz [rad]')
plt.ylabel('Amplitude')
plt.ylim(0, 200)
plt.plot(fPlot_Hz[:signalLength//2],
np.abs(corruptedAudioSignal_filtered_fft[:signalLength//2]))
plt.gcf().set_size_inches(8, 16)
plt.show()

# Abspielen
sound = (corruptedAudioSignal_filtered * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_filtered))).astype(np.int16)
play_obj = sa.play_buffer(sound, 1, 2, fs_Hz)
play_obj.wait_done()

```

Zum Schluss soll nun das bearbeitete Audiosignal gespeichert werden. Dazu kann das Objekt `wavfile.write()` verwendet werden.

```

# Lösung
'''
Beispiel: Daten in eine wav-Datei schreiben
'''
import numpy as np
from scipy.io import wavfile
from playsound import playsound

# Pfad einer neuen wav-Datei bestimmen
file_path = 'data/akustik_filtered.wav'
# Audiodaten in 16-Bit-Format konvertieren
data = (corruptedAudioSignal_filtered * (2**15 - 1) /
np.max(np.abs(corruptedAudioSignal_filtered))).astype(np.int16)
# Audiodaten schreiben
wavfile.write(file_path, fs_Hz, data)

# Testen, ob die Daten erfolgreich gespeichert wurde
playsound('data/akustik_filtered.wav')

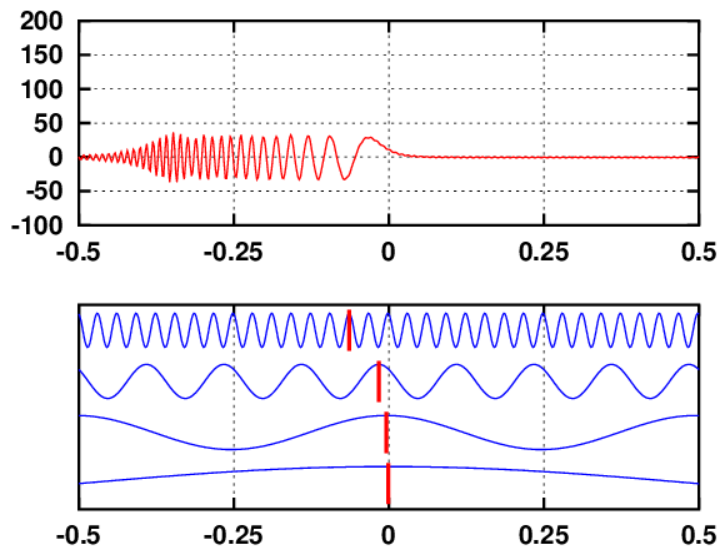
```

References

1. Titelbild von [Encarni Mármol](#)
2. [Play sound in Python](#)

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)

2.1 - Sweep



Sweeps sind eine beliebte Methode zur Messung der Übertragungsfunktion eines linearen Systems, da sie eine Reihe von positiven Eigenschaften aufweisen wie bspw. einen (einstellbaren) glatten Amplitudenfrequenzgang.

Das Thema der Sweeps wird in der Vorlesung in Thema 3: "Messung der Übertragungsfunktion und Impulsantworten" behandelt und das hier präsentierte Beispiel wird in der 6. Übungseinheit per Hand berechnet.

Inhalt

1. Sweep mit konstanter Hüllkurve
2. Sweep mit Dreieck-Hüllkurve
3. Realisierung mit Python-Modulen



1. Sweep mit konstanter Hüllkurve

Ein Sweep $x(t)$ im Zeitbereich ist eine harmonische Schwingung mit zeitabhängigem Phasenargument $\varphi(t)$, dessen Momentanfrequenz $\omega(t)$ mit der Zeit monoton zu- oder abnimmt (weswegen er auch Gleitsinus genannt wird, in Abgrenzung zu Sinusschwingungen mit einer konstanten Phase):
$$x(t) = x_0 \cdot \sin(\varphi(t)); \text{ mit } ; \omega(t) = \frac{d\varphi(t)}{dt}; \rightarrow x(t) = x_0 \cdot \sin\left(\int \omega(t) dt + \varphi_0\right)$$

In der Umgebung eines Zeitpunkts t_0 mit der *Momentanfrequenz* ω_0 nähert sich $x(t)$ einer Sinusfunktion mit der Frequenz ω_0 an.

Zur Erzeugung eines Sweeps mit linearer Hüllkurve und linearer Frequenzänderung erhält man folgende Formel:

$$x(t) = x_0 \cdot \sin\left(\omega_{\text{start}} \cdot t + \frac{\omega_{\text{end}} - \omega_{\text{start}}}{2 \cdot T_s} \cdot t^2\right)$$

Neben den schon bekannten Modulen `matplotlib`, `numpy`, `scipy` und `simpleaudio`, wird in diesem Notebook das Modul `ipywidgets` eingeführt. Damit lassen sich Benutzeroberflächen (user interface / UI) einfügen, wie zum Beispiel Schieberegler, mit denen sich Variablenwerte einstellen lassen. Dieses Modul müssen sie wahrscheinlich noch über `pip install` installieren. Wenn Sie das getan haben, können Sie nun alle für dieses Modul benötigten Module importieren:

```
'''
Import externer Module
'''

# Lösung
import numpy as np
import simpleaudio as sa
import matplotlib.pyplot as plt
from scipy import fftpack, signal
from ipywidgets import interact_manual
```

Die Variablen für die Berechnung des linearen Sweeps sind:

- f_s _Hz: Abtastfrequenz (für diese Aufgabe relativ beliebig),
- A: Amplitude des Sweeps (für diese Aufgabe relativ beliebig),
- f_{Start_Hz} : Startfrequenz des Sweeps,
- f_{End_Hz} : Endfrequenz des Sweeps,
- T_s : Dauer des Sweep.

Erstellen Sie zunächst eine Funktion "sweep_linear", die über die Eingabevariablen [f_s _Hz, $f_{Start_lin_Hz}$, $f_{End_lin_Hz}$, T_s und A] die zwei Ausgabevariablen [t_s (Array mit Abtastzeitpunkten) und sweep (Array mit Werten des linearen Sweeps an den Zeitpunkten von t_s)] erzeugt.

```
'''
Definition der Funktion sweep_linear
'''

# Lösung
def sweep_linear(fs_Hz, fStart_lin_Hz, fEnd_lin_Hz, T_s, A):
    t_s = np.linspace(0, T_s, int(fs_Hz*T_s))
    wstart_rad_per_s = fStart_lin_Hz * 2 * np.pi
    wend_rad_per_s = fEnd_lin_Hz * 2 * np.pi
    sweep = A * np.sin(wstart_rad_per_s * t_s + (wend_rad_per_s - wstart_rad_per_s) * (t_s
** 2) / (2 * T_s))
    return t_s, sweep
```

Nun initialisieren Sie die Variablen für die Berechnung des Sweeps. Die Werte der Variablen sollen dabei wie folgt sein:

- f_s _Hz: 16 kHz,
- A: 1,
- f_{Start_Hz} : 50 Hz,
- f_{End_Hz} : 400 Hz,
- T_s : 4 s.

Führen Sie mit diesen Variablen die Funktion "sweep_linear" aus und lassen Sie sich diese graphisch darstellen:

```
'''
Variableninitialisierung und Erzeugen der Zeitfunktion des Sweeps
'''

# Lösung
# Initialisierung der Variablen
fs_Hz = 16e3
amplitude = 1
fStart_Hz = 50
fEnd_Hz = 400
T_s = 4

# Zeitbereich
time_s, sweep_lin = sweep_linear(fs_Hz, fStart_Hz, fEnd_Hz, T_s, amplitude)
```

```
# Graphische Darstellung des Sweeps
plt.title('Sweep von %d Hz bis %d Hz' % (fStart_Hz, fEnd_Hz))
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(time_s, sweep_lin)
plt.gcf().set_size_inches(20, 5)
plt.show()
```

Nun können Sie sich den erzeugten Sweep auch anhören! Erstellen Sie dafür eine Funktion "play_audio", die mittels `simpleaudio` Arrays ausgibt (wie in Kapitel 1.3 eingeführt). Nutzen Sie dann die neu definierte Funktion, um sich den Sweep auszugeben.
(Achtung: Das Signal ist sehr laut!)

```
'''
Definition der Funktion play_audio
'''
# Lösung
def play_audio(sweep_signal):
    sound = (sweep_signal*(2**15-1)/np.max(np.abs(sweep_signal))).astype(np.int16)
    play_obj = sa.play_buffer(sound, 1, 2, int(fs_Hz))
    play_obj.wait_done()
```


```
'''
Audioausgabe
'''
#Lösung
play_audio(sweep_lin)
```

Es ist zudem sehr interessant, wie das Signal im Frequenzbereich aussieht. Wenden Sie deswegen die in der folgenden Funktion "fft_sweep" definiert Fast-Fourier-Transformation auf den Sweep an:

```
'''
Definition der Funktion fft_sweep
'''
def fft_sweep(sweep, fs_Hz):
    N = sweep.size # Länge von FFT
    f_Hz = np.linspace(0, fs_Hz/2, int(N/2)) # Frequenzbereich
    sweep_fft = fftpack.fft(sweep, N)
    sweep_fft_plot = np.abs(sweep_fft[:len(f_Hz)]) / int(N/2)
    return f_Hz, sweep_fft_plot
```

```
'''
Erzeugen des Frequenzgangs des Sweeps
'''
# Lösung
# Anwendung der FFT
f_Hz, sweep_fft_plot = fft_sweep(sweep_lin, fs_Hz)
```


```
# Graphische Darstellung
plt.title('Amplitudenfrequenzgang')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|X(f)|')
plt.plot(f_Hz, sweep_fft_plot)
plt.axis([fStart_Hz*0.5, fEnd_Hz*1.1, 0, np.max(sweep_fft_plot)*1.1])
plt.gcf().set_size_inches(20, 5)
plt.show()
```


sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.1 - Sweep_solved_15_0.png

Der Frequenzgang zeigt an den Rändern des Frequenzbands eine starke Welligkeit auf. Um diese zu entfernen, kann die Frequenz mittels Ein- und Ausblenden geglättet werden. Dies soll hier mit einem Tukey-Fenster aus dem Objekt `signal.get_window()` umgesetzt werden.

```
'''
Erzeugen des Tukey-Fensters
'''
# Lösung
window = signal.get_window(('tukey', 0.2), len(sweep_lin))
```

```
plt.title('Tukey-Fenster')
plt.xlabel('Zeit [s]')
plt.ylabel('window(t)')
plt.plot(time_s, window)
plt.gcf().set_size_inches(20, 5)
plt.show()
```


sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.1 - Sweep_solved_18_0.png

```
'''
Fensterung des Sweep-Signals
'''


# Lösung
# Zeitbereich
sweep_win = sweep_lin * window

# Frequenzbereich
f_Hz, sweep_win_fft_plot = fft_sweep(sweep_win, fs_Hz)
```

```
# Graphische Darstellung
plt.subplot(211)
plt.title('Sweep von %d Hz bis %d Hz' %(fStart_Hz, fEnd_Hz))
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(time_s, sweep_win)

plt.subplot(212)
plt.title('Amplitudenfrequenzgang')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|X(f)|')
plt.plot(f_Hz, sweep_win_fft_plot)
plt.axis([fStart_Hz*0.5, fEnd_Hz*1.1, 0, np.max(sweep_fft_plot)*1.1])

plt.gcf().set_size_inches(20, 10)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.1 - Sweep_solved_20_0.png

```
'''
Audioausgabe
'''

# Lösung
play_audio(sweep_win)
```

Durch die Fensterung sollte die Welligkeit im Frequenzbereich verringert worden sein. Nur verringert sich dadurch auch die Amplitude der äußeren Frequenzen.

Um sich die Veränderungen von Fenstern oder Frequenzen interaktiv anschauen zu können, wird nun das Modul [ipywidgets](#) genutzt. Durch dieses können Variablen verändert werden und mittels Drop-Down-Menüs oder Häkchen-Kästen die Ausgabe verändert werden. Probieren Sie es einfach aus. Sie müssen dafür noch in dem folgenden Code Ihre definierten Funktionen einfügen:

```
'''
Initialisierung fester Variablen
'''

fs_Hz = 16e3
amplitude = 1
```

```
'''
Interaktive Sweepdarstellung
'''

# Lösung
@interact_manual(T_s_i=(0.1, 5, 0.1), fStart_Hz_i=(10, 5000, 10), fEnd_Hz_i=(10, 5000,
10), windowing_i = False, window_type_i=[('tukey', 1), ('triang', 2)], win_alpha_i=(0, 1,
0.01), output_sound_i = True)
def interactive_linear_sweep(T_s_i=2, fStart_Hz_i=10, fEnd_Hz_i=100, windowing_i=False,
window_type_i='tukey', win_alpha_i=0.2, output_sound_i=True):
    # Erzeugung des Sweeps
    time_s, sweep = sweep_linear(fs_Hz, fStart_Hz_i, fEnd_Hz_i, T_s_i, amplitude)

    # Optionales Fenstern des Sweeps
    if windowing_i == True:
        if window_type_i == 1:
            window = signal.get_window(('tukey', win_alpha_i), len(sweep))
        elif window_type_i == 2:
            window = signal.get_window('triang', len(sweep))
        else:
            window = signal.get_window('boxcar', len(sweep))
        sweep = sweep * window

    # Graphische Darstellung des Zeitbereichs
    plt.title('Sweep von %d Hz zu %d Hz' %(fStart_Hz_i, fEnd_Hz_i))
    plt.xlabel('Zeit [s]')
    plt.ylabel('x(t)')
    plt.plot(time_s, sweep)
    plt.gcf().set_size_inches(20, 5)
    plt.show()

    # FFT des Sweeps
    f_Hz, sweep_fft_plot = fft_sweep(sweep, fs_Hz)

    # Graphische Darstellung des Frequenzbereichs
    plt.title('Amplitudenfrequenzgang')
    plt.xlabel('Frequenz [Hz]')
    plt.ylabel('|X(f)|')
    plt.plot(f_Hz, sweep_fft_plot)
    if fStart_Hz_i <= fEnd_Hz_i:
        plt.axis([fStart_Hz_i*0.5, fEnd_Hz_i*1.2, 0, np.max(sweep_fft_plot)*1.1])
    else:
        plt.axis([fEnd_Hz_i*0.5, fStart_Hz_i*1.2, 0, np.max(sweep_fft_plot)*1.1])

    plt.gcf().set_size_inches(20, 5)
    plt.show()

    # Optionale Audioausgabe
    if output_sound_i == True:
        play_audio(sweep)
```



2. Sweep mit Dreieck-Hüllkurve

Im Nachfolgenden soll der in Übung 6.1 händisch berechnete Sweep implementiert und visualisiert werden. Dabei soll der Sweep im Frequenzbereich eine konstante Amplitude in seinem Band erhalten. Um das Frequenzverhalten eines linearen Sweeps zu betrachten, welches mit einem Dreieck gefenstert wurde, kann in der vorherigen Zelle betrachtet werden, wenn man das Fenster 'triang' wählt.

Zunächst werden die Grundparameter definiert. Die Start- und Endfrequenz $f_{\text{start}} = 200 \text{ Hz}$ und $f_{\text{end}} = 1000 \text{ Hz}$ bzw. ω_{start} und ω_{end} sowie die notwendige Abtastfrequenz $f_s = 16 \text{ kHz}$ festgelegt:

```
'''
Grundparameter definieren:
'''

# Lösung
fs_Hz = 16e3      # Abtastfrequenz (für diese Aufgabe relativ beliebig)
dt_s = 1/fs_Hz    # Zeitintervall
fStart_Hz = 200    # Startfrequenz [Hz]
fEnd_Hz = 1000    # Endfrequenz [Hz]
wStart_rad_per_s = fStart_Hz * 2 * np.pi # Startfrequenz [rad/s]
wEnd_rad_per_s = fEnd_Hz * 2 * np.pi    # Endfrequenz [rad/s]
wm_rad_per_s = (wStart_rad_per_s + wEnd_rad_per_s) / 2 # Mittenfrequenz [rad/s]
```

Anschließend werden die Funktionen für die Phasenwinkel $\varphi_1(t)$ und $\varphi_2(t)$ definiert. Nutzen Sie dafür die berechneten Formeln aus 6.1 c):

```
'''
Definition der Funktion calculatePhi1
'''
# Lösung
def calculatePhi1(t_s, wStart_rad_per_s, k, T_s):
    phi = k*t_s**4/(3*T_s**2) + wStart_rad_per_s*t_s
    return phi
```

```
'''
Definition der Funktion calculatePhi2
'''
# Lösung
def calculatePhi2(t_s, wStart_rad_per_s, k, T_s):
    phi = k * t_s**4 / (3 * T_s**2) - 4*k / (3*T_s) * t_s**3 \
    + 2*k * t_s**2 - k*T_s*t_s + wStart_rad_per_s*t_s \
    + 7*k/48 * T_s**2 - wStart_rad_per_s*T_s/2
    return phi
```

Durch die Diskontinuität in der Einhüllenden ($a(t)$) müssen in diesem Spezialfall zusätzliche Bedingungen erfüllt werden, damit die Phase an der Übergangsstelle kontinuierlich verläuft. Zum einen muss der Sweep-Phasenwinkel $\varphi(t)$ an der Stelle $T/2$ gleich sein:
$$\sin\left(\varphi_1\left(\frac{T}{2}\right)\right) = \sin\left(\varphi_2\left(\frac{T}{2}\right)\right)$$

Substituiert man hier die berechneten Gleichungen aus der Übung für $\varphi_1(T/2)$ und $\varphi_2(T/2)$ ergibt sich die erste Bedingung
$$\sin\left(\frac{T \cdot (7\omega_1 + \omega_2)}{16}\right) = \sin(0) = 0.$$

Zum anderen muss auch die Ableitung des Sweep-Phasenwinkels, $\frac{d}{dt}\varphi(t)$ gleich sein, d.h. aus der ersten Bedingung folgt (innere Ableitung nicht vergessen):

$$\cos\left(\frac{T \cdot (7\omega_1 + \omega_2)}{16}\right) \cdot \left(\frac{\omega_1 + \omega_2}{2}\right) = \cos(0) \cdot \left(\frac{\omega_1 + \omega_2}{2}\right) = 1$$

$$\Leftrightarrow \cos\left(\frac{T \cdot (7\omega_1 + \omega_2)}{16}\right) = 1.$$

Beide Gleichungen sind erfüllt, wenn das Argument ein Vielfaches von 2π ist. Damit kann die Sweepdauer T explizit durch

$$\frac{T}{16} \cdot (\omega_1 + \omega_2) = 2\pi \cdot n, \quad n=0,1,2,3,\dots$$

berechnet werden und ist durch $n \in \mathbb{N}$ nicht komplett frei wählbar. Dieser Zusammenhang für die Periode T soll nun in der nachfolgenden Funktion definiert:

```
'''
Definition der Funktion calculatePhi2
'''
# Lösung
def calculatePeriod(wStart_rad_per_s, wEnd_rad_per_s, n_periods):
    T_s = 2*n_periods*np.pi*16 / (7*wStart_rad_per_s + wEnd_rad_per_s)
    return T_s
```

Zudem muss der Faktor k aus 6.1.b) berechnet werden. Dafür wird nun auch eine Funktion definiert:

```
'''
Definition der Funktion calculatek
'''
# Lösung
def calculatek(wStart_rad_per_s, wEnd_rad_per_s, T_s):
    k = 3/T_s * (wEnd_rad_per_s - wStart_rad_per_s)
    return k
```

Zu guter Letzt werden noch die beiden Funktionen für die Berechnung der Amplitudenmodulaton $a(t)$ für beide Hälften definiert. Nutzen Sie dafür die Formeln aus 6.1 a):

```
'''
Definition der Amplituden-Funktionen
'''
# Lösung
def calculateAmplitude1(t_s, T_s):
    a = 2/T_s*t_s
    return a

def calculateAmplitude2(t_s, T_s):
    a = -2/T_s*(t_s-T_s)
    return a
```

Nun berechnen wir zuerst die Sweepdauer, indem wir die Funktion "calculatePeriod" verwenden und dafür die Variable n_periods auf 300 festsetzen (der Wert ist hier egal). Daraus lässt sich zudem die Konstante k aus 6.1. b) bestimmen:

```
'''
Berechnung von T_s und k
'''
# Lösung
n_periods = 300;
# Sweepdauer T berechnen in Abhängigkeit von ganzen Perioden 2*pi*n:
T_s = calculatePeriod(wStart_rad_per_s, wEnd_rad_per_s, n_periods) # die Dauer des Sweeps
print("Sweepdauer: {} Sekunden.\n".format(T_s))

# Konstante k:
k = calculatek(wStart_rad_per_s, wEnd_rad_per_s, T_s)
```

Sweepdauer: 2.0 Sekunden.

Jetzt können wir den Sweep damit berechnen und plotten:


```
'''
Implementierung des Sweeps nach Übung 6.1
'''
# Lösung
# Laufvariable t definieren:
t1_s = np.linspace(0, T_s/2, int(fs_Hz*T_s/2))
t2_s = np.linspace(T_s/2+dt_s, T_s, int(fs_Hz*T_s/2-1))

# Phasenwinkel für beide Abschnitte:
phi1_rad = calculatePhi1(t1_s, wStart_rad_per_s, k, T_s)
phi2_rad = calculatePhi2(t2_s, wStart_rad_per_s, k, T_s)

# Einhüllende für beide Abschnitte:
a1 = calculateAmplitude1(t1_s, T_s)
a2 = calculateAmplitude2(t2_s, T_s)

# Vollständigen Sweep zusammensetzen:
sweep1 = a1 * np.sin(phi1_rad)
sweep2 = a2 * np.sin(phi2_rad)
t_s = np.append(t1_s, t2_s)
sweep = np.append(sweep1, sweep2)
```


```
# Plot
plt.title('Sweep von %d Hz bis %d Hz' %(fStart_Hz, fEnd_Hz))
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_s, sweep)
plt.gcf().set_size_inches(20, 5)
plt.show()
```

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.1 - Sweep_solved_40_0.png

Um sicherzustellen, dass der Sweep tatsächlich das (annähernd) glatte Spektrum über den definierten Frequenzbereich von f_{start} bis f_{end} besitzt, wird der Amplitudenfrequenzgang $|X(k)|$ über die FFT berechnet. Dazu kann die im ersten Teil definierte Funktion "fft_sweep" verwendet werden:

```
'''
Aufgabe: Spektrum des Sweeps berechnen
'''
# Lösung
f_Hz, sweep_fft_plot = fft_sweep(sweep, fs_Hz)
```

```
# Plot
plt.title('Amplitudenfrequenzgang')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('|X(f)|')
plt.plot(f_Hz, sweep_fft_plot)
plt.axis([fStart_Hz*0.5, fEnd_Hz*1.2, 0, np.max(sweep_fft_plot)*1.1])
plt.gca().set_size_inches(10, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.1 - Sweep_solved_43_0.png

Abspielen des Sweepssignal $x(t)$ ergibt das charakteristische Sweepgeräusch. Hierzu können Sie auch die aus dem ersten Part definierte Funktion "playaudio()" verwenden:

```
'''
Aufgabe: Sweep abspielen
'''
# Lösung
play_audio(sweep)
```

Zum Schluss wollen wir uns das Ergebnis noch einmal als interaktives Modul anschauen. Dazu wird wieder die Funktion `interact` aus dem Modul `ipywidgets` verwendet:

```
'''
Initialisierung fester Variablen
'''
fs_Hz = 16e3
dt_s = 1/fs_Hz
```



```

'''
Interaktive Sweepdarstellung
'''
# Lösung
@interact_manual(n_periods_i=(1, 1000, 1), fStart_Hz_i=(10, 5000, 10), fEnd_Hz_i=(10,
5000, 10), output_sound_i = True)
def interactive_rect_sweep(n_periods_i=300, fStart_Hz_i=200, fEnd_Hz_i=1000,
output_sound_i=True):

    # Berechnung der Koeffizienten
    wStart_rad_per_s_i = fStart_Hz_i * 2 * np.pi # Startfrequenz [rad/s]
    wEnd_rad_per_s_i = fEnd_Hz_i * 2 * np.pi # Endfrequenz [rad/s]
    T_s_i = calculatePeriod(wStart_rad_per_s_i, wEnd_rad_per_s_i, n_periods_i)
    k_i = calculatek(wStart_rad_per_s_i, wEnd_rad_per_s_i, T_s_i)

    # Laufvariable t definieren:
    t1_s_i = np.linspace(0, T_s_i/2, int(fs_Hz*T_s_i/2))
    t2_s_i = np.linspace(T_s_i/2+dt_s, T_s_i, int(fs_Hz*T_s_i/2-1))

    # Phasenwinkel für beide Abschnitte:
    phi1_rad_i = calculatePhi1(t1_s_i, wStart_rad_per_s_i, k_i, T_s_i)
    phi2_rad_i = calculatePhi2(t2_s_i, wStart_rad_per_s_i, k_i, T_s_i)

    # Einhüllende für beide Abschnitte:
    a1_i = calculateAmplitude1(t1_s_i, T_s_i)
    a2_i = calculateAmplitude2(t2_s_i, T_s_i)

    # Vollständigen Sweep zusammensetzen:
    sweep1_i = a1_i * np.sin(phi1_rad_i)
    sweep2_i = a2_i * np.sin(phi2_rad_i)
    t_s_i = np.append(t1_s_i, t2_s_i)
    sweep = np.append(sweep1_i, sweep2_i)

    # Graphische Darstellung des Zeitbereichs
    plt.title('Sweep von %d Hz zu %d Hz' %(fStart_Hz_i, fEnd_Hz_i))
    plt.xlabel('Zeit [s]')
    plt.ylabel('x(t)')
    plt.plot(t_s_i, sweep)
    plt.gcf().set_size_inches(20, 5)
    plt.show()

    # FFT des Sweeps
    f_Hz, sweep_fft_plot = fft_sweep(sweep, fs_Hz)

    # Graphische Darstellung des Frequenzbereichs
    plt.title('Amplitudenfrequenzgang')
    plt.xlabel('Frequenz [Hz]')
    plt.ylabel('|X(f)|')
    plt.plot(f_Hz, sweep_fft_plot)
    if fStart_Hz_i <= fEnd_Hz_i:
        plt.axis([fStart_Hz_i*0.5, fEnd_Hz_i*1.2, 0, np.max(sweep_fft_plot)*1.1])
    else:
        plt.axis([fEnd_Hz_i*0.5, fStart_Hz_i*1.2, 0, np.max(sweep_fft_plot)*1.1])

    plt.gcf().set_size_inches(20, 5)
    plt.show()

    # Optionale Audioausgabe
    if output_sound_i == True:
        play_audio(sweep)

```



3. Realisierung mit Python-Modulen


Im Modul `scipy` ist die Funktion `signal.chirp()` als Frequenzgesteuerter Kosinusgenerator vorhanden, dadurch können Sweeps deutlich einfacher erzeugt werden. Zudem kann mit dem Modul `signal.spectrogram()` das Signal in einem Spektrogramm betrachtet werden.

```
'''
Erzeugen des Sweep mittels signal.chirp()
'''
from scipy.signal import chirp, spectrogram

# Sweep-Parameter definieren:
fsChirp_Hz = 16000
TChirp_s = 3
tChirp_s = np.linspace(0, TChirp_s, TChirp_s*fsChirp_Hz, endpoint=False)
chirpStartFreq_Hz = 200
chirpEndFreq_Hz = 1000

# Sweep und dessen Amplitudenfrequenzgang berechnen:
xSweep = chirp(tChirp_s, chirpStartFreq_Hz, TChirp_s, chirpEndFreq_Hz, 'linear')
fChirp_Hz, tChirpSg_s, XSweep_f = spectrogram(xSweep, fsChirp_Hz, nperseg=250)
```

```
# Plot
plt.subplot(121)
plt.title('Linearer Chirp, f(0)=%d Hz, f(%d)=%d Hz' %(chirpStartFreq_Hz, TChirp_s,
chirpEndFreq_Hz))
plt.xlabel('Zeit [s]')
plt.ylabel('$x_{sweep}(t)$')
plt.plot(tChirp_s, xSweep)
plt.subplot(122)
plt.title('Spektrogramm')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$X_{sweep}(f)$')
plt.pcolormesh(tChirpSg_s, fChirp_Hz, XSweep_f, shading='gouraud')
plt.gca().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.1 - Sweep_solved_51_0.png

```
'''
Aufgabe: Sweep abspielen
'''
# Werte in 16-Bit-Daten konvertieren
sound = (xSweep*(2**15-1)/np.max(np.abs(xSweep))).astype(np.int16)

# Abspielen
play_obj = sa.play_buffer(sound, 1, 2, int(fsChirp_Hz))
play_obj.wait_done()
```

Die Verwendung eines Spektrogramms statt eines einfachen Spektrums zur Darstellung des Frequenzbereiches (in diesem Falle in zusätzlicher Abhängigkeit von der Zeit) wird im nächsten Notebook noch ausführlich behandelt.

References

1. Titelbild von [wikimedia](#)
2. [Sweep \(Signalverarbeitung\)](#)
3. [Sinusoidal Sweep Signals](#)
4. [Sine Sweep](#)

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)



2.2 - Linear Predictive Coding

Das "Linear Predictive Coding" 

(LPC) ist ein Verfahren, das hauptsächlich in der Audiosignalverarbeitung und Sprachverarbeitung verwendet wird, um die Spektralhüllkurve eines digitalen Sprachsignals in komprimierter Form unter Verwendung der Informationen eines linearen Vorhersagemodells darzustellen. Es ist eine der leistungstärksten Sprachanalysetechniken und eine der nützlichsten Methoden zum Codieren von Sprache in guter Qualität mit einer niedrigen Bitrate und liefert hochgenaue Schätzungen von Sprachparametern. LPC ist die am weitesten verbreitete Methode in der Sprachcodierung und Sprachsynthese.

Thematisch wird LPC erst in Thema5-Hüllkurven behandelt, da sich mit LPC aber im Zuge der Bestimmung der Einhüllenden auch die Systemfunktion des z.B. Vokaltraktes (d.h. des dazugehörige *Synthesfilters*) schätzen lassen, wird sie in den Notebooks an dieser Stelle eingeführt. LPC unterscheidet sich aber grundlegend von dem Vorgehen der Messung der Übertragungsfunktion, wie es am Beispiel des Sweeps im Notebook 2.1 behandelt wurde.

Inhalt

	1. Grundlagen
	2. Anwendungsbeispiel

1. Grundlagen



Die artikulatorische Sprachsynthese (anders als z.B. die datengestützte Sprachsynthese, wie sie in sämtlichen Sprachassistenten angewendet wird) basiert auf der sog. Quelle-Filter-Theorie, welche davon ausgeht, dass die Sprachproduktion in zwei Prozesse unterteilt werden kann, die unabhängig voneinander sind: die Erzeugung eines Anregungssignales und die anschließende Filterung dieses Anregungssignales. Unabhängig bedeutet in diesem Kontext, dass der nachgeschaltete Filterprozess das Anregungssignal nicht verändert, also keine Rückkopplung entsteht.

Vereinfacht dargestellt erzeugen bei der Sprachproduktion die Glottis und die Stimmlippen unter Anregung eines Luftstromes das Anregungssignal, welches vom darauffolgenden Vokaltrakt gefiltert wird. Das Anregungssignal kann dabei stimmhaft sein (d.h. es besitzt eine Grundperiode), stimmlos (Rauschsignal) oder eine Mischung aus Beidem. Wenn sich die Stimmlippen in einem vokalisierten Zustand befinden (Vibration der Stimmlippen), werden stimmhafte Laute (zum Beispiel Vokale) erzeugt. Wenn sich die Stimmbänder in einem "stillen" Zustand befinden, werden stimmlose Laute (zum Beispiel Konsonanten) erzeugt.

LPC bietet die Möglichkeit, die Systemfunktion des Vokaltraktes zu schätzen, ohne das Anregungssignal zu kennen.

Grundidee:

Die Grundidee der LPC ist zunächst generell, dass die einzelnen Abtastwerte eines linear gefilterten Ausgangssignals $y(k)$ (in Falle der Sprachproduktion das abgestrahlte Sprachschallsignal) nicht unabhängig voneinander sind. Stattdessen lässt sich jeder Abtastwert $y(k)$ aus einer Linearkombination endlich vielen vorangegangenen Abtastwerten annähern:
$$\hat{y}(k) = \sum_{i=1}^N a_i y(k-i)$$
 Die Gewichtungsfaktoren a_i heißen *Prädiktorkoeffizienten*.

Für jeden Abtastwert wird damit ein gewisser Prädiktionsfehler $e(k)$ gemacht:
$$e(k) = y(k) - \hat{y}(k) = y(k) - \sum_{i=1}^N a_i y(k-i)$$

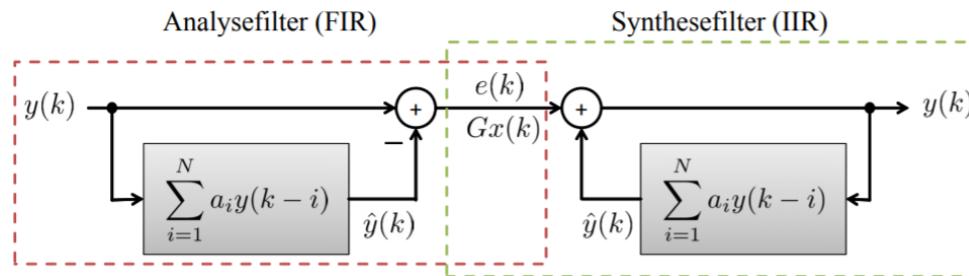
Die a_i werden geschätzt, indem der mittlere quadratische Fehler (der sog. MSE, *mean squared error*) im betrachteten Signalabschnitt minimiert wird.

Die Gleichung für $y(k)$ entspricht formal der Rekursionsgleichung für einen IIR-Allpol-Filter $H(z)$, wobei das Fehlersignal als (mit der Verstärkung G skaliertes) Eingangssignal $x(k)$ betrachtet werden kann:
$$y(k) = e(k) + \sum_{i=1}^N a_i y(k-i) = Gx(k) + \sum_{i=1}^N a_i y(k-i)$$

Die Systemfunktion ist somit
$$H(z) = \frac{Y(z)}{E(z)} = \frac{1}{1 - \sum_{i=1}^N a_i z^{-i}}$$

Dieser Filter ist der sogenannte **LPC-Synthesefilter**. Der Amplitudengang des Synthesefilters entspricht bei einem weißen Anregungsspektrum der Hüllkurve im Frequenzbereich.

Umgekehrt lässt sich das Fehlersignal $e(k)$ durch die Filterung von $y(k)$ mit dem inversen Filter $A(z) = 1/H(z)$ bestimmen. Dieser Filter wird als **Analysefilter** bezeichnet.



2. Anwendungsbeispiel

LPC findet, wie bereits erwähnt, breite Anwendung in der Spracherkennung, Sprachsynthese und Sprachkompression. Als Beispiel soll hier die lineare Prädiktion zur Formanterkennung verwendet werden, d.h., um die Formantfrequenzen (Resonanzfrequenzen) des Vokaltrakts zu schätzen. Im Quellfilter Sprachsignalmodell entsprechen die Formantfrequenzen den komplexen Polpaaren der Übertragungsfunktion.

Ausgehend von einem kurzen Abschnitt des Signals werden die Prädiktorkoeffizienten geschätzt, welche wiederum die Filterkoeffizienten a_i des Nennerpolynoms darstellen. Jede komplexe Wurzel der diskreten Systemfunktion des Synthesefilters lässt sich bekanntermaßen auch über
$$p_i = r_i e^{j\Omega_i}, \quad r_i = \sqrt{\text{real}(p_i)^2 + \text{imag}(p_i)^2}, \quad \Omega_i = \text{atan}\left(\frac{\text{imag}(p_i)}{\text{real}(p_i)}\right)$$
 darstellen, d.h. die geschätzten Formantfrequenzen f_i sind
$$f_i = \frac{\Omega_i \cdot f_s}{2\pi}$$
 Wobei nur die Frequenzen mit positivem Vorzeichen in Frage kommen. Außerdem müssen die Formantfrequenzen überhalb der Grundfrequenz liegen, d.h. eine zusätzliche Bedingung ist $f_i > \approx 90\text{-}100\text{ Hz}$.

Importieren Sie zuerst die verwendeten externen Module:

```
'''
Import externer Module
'''
# Lösung

import numpy as np
import simpleaudio as sa
import matplotlib.pyplot as plt
import scipy
from scipy.io import wavfile
from scipy import signal
import librosa
from ipywidgets import interact_manual
```

Nun laden Sie das Audiosignal `akustik.wav` in das Projekt und bestimmen Sie dessen Eigenschaften:

```
'''
Audiosignal Laden
'''

# Lösung
fs_Hz, audioSignal = wavfile.read('data/akustik.wav') # Sample Rate, Audiosignal im
Array-Form
audioSignal = audioSignal/np.max(np.abs(audioSignal)) # Normalisierung
signalLength = len(audioSignal) # Länge des Audiosignals
T_s = signalLength/fs_Hz - 1/fs_Hz # Zeit
t_s = np.linspace(0, T_s, signalLength) # Zeitbereich
f_Hz = np.linspace(0, fs_Hz/2, signalLength//2) # Frequenzbereich

print("Abtastrate : %d Hz, Signaldauer : %.2f s" %(fs_Hz, T_s))
```

```
Abtastrate : 16000 Hz, Signaldauer : 3.76 s
```

Hören Sie sich nun das Array `audioSignal` an. Erstellen Sie dafür eine Funktion "play_audio", die mittels `simpleaudio` Zahlenarrays als Signal über die Lautsprecher ausgibt.

```
'''
Definition der Funktion play_audio
'''

# Lösung
def play_audio(acoustic_signal_i):
    sound = (acoustic_signal_i*(2**15-
1)/np.max(np.abs(acoustic_signal_i))).astype(np.int16)
    play_obj = sa.play_buffer(sound, 1, 2, int(fs_Hz))
    play_obj.wait_done()
```

```
'''
Audioausgabe
'''

# Lösung
play_audio(audioSignal)
```

Zur Bestimmung der Segmentlänge ist ein Kompromiss zwischen zwei gewünschten Eigenschaften zu finden:

1. Das Segment soll so klein wie möglich sein, um das Spektrum an einem Zeitpunkt zu erhalten,
2. Das Segment soll so groß wie möglich sein, um das Spektrum invariant gegenüber der Lage der Grundperiode zu machen.

Die Grundperiode des Signals liegt zwischen 8-12 ms. Als guten Kompromiss zwischen den beiden Eigenschaften hat sich die Segmentlänge von 32 ms erwiesen. Das hat auch den Vorteil, dass diese Länge bei einer Abtastrate von 16 kHz insgesamt 512 Elemente umfasst, welches eine optimale Größe bei der Anwendung der FFT bedeutet.

Schneiden Sie das Signal nun in ein 32 ms bzw. 512 Elemente langes Segment, in dem der Vokal /a/ gesagt wird (Startpunkt zum Beispiel bei 1,18 s), und berechnen Sie das Spektrum des /a/-Vokaltrakt:

```
'''
Spektrum eines 32 ms langen /a/-Segments des Wortes „Akustik“
'''

# Lösung
# Zeitintervall bestimmen
n_start = int(1.18 / T_s * signalLength) # Startsample-Nummer
n_end = n_start + 512 # EndsampLe-Nummer
t_aSegment_s = t_s[n_start:n_end]


# Audiosignal segmentieren
a_segment = audioSignal[n_start:n_end]

# Spektrum berechnen
n_fft = 512; # Länge von FFT
f_aSegment_Hz = np.linspace(0, fs_Hz-fs_Hz/2-fs_Hz/n_fft, int(n_fft/2)) #
Frequenzbereich
aSegment_fft = scipy.fftpack.fft(a_segment, n_fft)
aSegment_fft_plot = np.abs(aSegment_fft[:len(f_aSegment_Hz)]) / int(n_fft/2)
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Audiosignal eines /a/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_aSegment_s, a_segment)

plt.subplot(122)
plt.title('Amplitudenfrequenzgang des /a/-Segments')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('log|X(f)|')
plt.plot(f_aSegment_Hz, np.log(aSegment_fft_plot))

plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_13_0.png

Die FFT geht von einer periodischen Fortführung des Signals aus. Beim Segmentieren des Signals entstehen dadurch Sprünge zwischen dem ersten und letzten Wert, die hochfrequente Anteile erzeugen, die im Signal eigentlich nicht vorhanden sind. Deshalb sollte das Signal vor der Verarbeitung gefenstert werden. Verwenden Sie dafür ein "Von-Hann-Fenster":

```
'''
Anwendung des von-Hann-Fensters auf das /a/-Segment
'''


# Lösung
# Audiosegment fenstern
a_segment_win = signal.hann(a_segment.size)*a_segment

# Spektrum des von-Hann-gefensterten Signals
aSegment_fft_win = scipy.fftpack.fft(a_segment_win, n_fft)
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Audiosignal des gefensterten /a/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_aSegment_s, a_segment_win)

plt.subplot(122)
plt.title('Amplitudenfrequenzgang des gefensterten /a/-Segments')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('log|X(f)|')
plt.plot(f_aSegment_Hz, np.log(np.abs(aSegment_fft_win[:len(f_aSegment_Hz)])))

plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_16_0.png

Geben Sie sich nun das segmentierte Audiosignal `a_segment` aus. Da das Signal nur 32 ms lang ist, muss das Signal wiederholt werden. Definieren Sie deshalb dafür eine Funktion "repeat_audio", die das eingegebene Signal mehrfach hintereinander hängt (erwarten sie dabei keine schön klingendes Signal. Es soll lediglich demonstriert werden, dass eine Tendenz hörbar ist, um welchen Vokal es sich handelt, selbst wenn eine sehr simple Verkettung der Einzelsegmente vorgenommen wird).

```
'''
Definition der Funktion repeat_audio
'''

# Lösung
def repeat_audio(acoustic_signal_i, signal_output_i=False):
    sound = acoustic_signal_i
    for i in range(0, 7):
        sound = np.concatenate((sound,sound), axis = None)

    # Ausgabe
    if signal_output_i == True:
        sound = (sound*(2**15-1)/np.max(np.abs(sound))).astype(np.int16)
        play_obj = sa.play_buffer(sound, 1, 2, int(fs_Hz))
        play_obj.wait_done()
    return sound
```

Hören Sie sich sowohl das unbearbeitete als auch das gefensternte Segment an:

```
'''
Audioausgabe
'''

# Lösung
# a_segment_rep = repeat_audio(a_segment, signal_output_i = True)
a_segment_win_rep = repeat_audio(a_segment_win, True)

# play_audio(a_segment_rep)
# play_audio(a_segment_win_rep)
```

Das bloße wiederholte Abspielen des Segments sorgt für eine schlechte Audioausgabe. Um dies etwas zu verbessern, kann das Signal überlappt hintereinander abgespielt werden (dies wird nochmal wichtiger beim Abspielen des synthetisierten Signals). Die folgende Funktion "overlap_audio" tut genau das. Als Eingabe wird das Segment und eine Wiederholzeit angegeben. Das Signal wird ausgegeben und zudem die berechnete Überlappung der Segmente zurückgegeben. Wenden Sie die Funktion auf das /a/-Segment an mit einer Wiederholdauer von 10 ms. Hören Sie sich das Signal an und schauen Sie sich einen Ausschnitt der entstandenen Audiospur an.

```
'''
Definition der Funktion repeat_audio
'''

def overlap_audio(signal_segment_i, t_overlap_ms_i, signal_output = False):
    n_repetition = int(0.001 * t_overlap_ms_i * 16000) # Berechnung der Wiederholdauer
    in Abtastpunkte

    playable_audio = np.zeros(512) # Initialisierung
    for i in range(0, 200): # Überlappung von 200 Segmenten
        playable_audio[i*n_repetition:] = playable_audio[i*n_repetition:] +
        signal_segment_i
    playable_audio = np.concatenate((playable_audio, np.zeros(n_repetition)))
    playable_audio = playable_audio[190:-512] # Schneiden der Unüberlappten
    Stellen (vorne und hinten)

    # Ausgabe
    if signal_output == True:
        sound = (playable_audio*(2**15-1)/np.max(np.abs(playable_audio))).astype(np.int16)
        play_obj = sa.play_buffer(sound, 1, 2, int(fs_Hz))
        play_obj.wait_done()
    return playable_audio # Rückgabe des berechneten Arrays
```

```
'''
Audioausgabe und graphische Darstellung der
'''

# Lösung
T_overlap_ms = 10

playable_audio_a = overlap_audio(a_segment, T_overlap_ms, True)

playable_audio_a_win = overlap_audio(a_segment_win, T_overlap_ms, False)

t_range_s = np.linspace(0, (1024-1)/fs_Hz, 1024)
```


```

# Graphische Darstellung
plt.subplot(121)
plt.title('addiertes Audiosignal des /a/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_range_s, playable_audio_a[:1024])

plt.subplot(122)
plt.title('addiertes Audiosignal des gefensterten /a/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_range_s, playable_audio_a_win[:1024])

plt.gcf().set_size_inches(15, 5)
plt.show()

```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_24_0.png

Ähnlich können Sie das Spektrum eines 32 ms langen /u/-Segments des Wortes „Akustik“ berechnen (Anfangspunkt zum Beispiel: 1,42 s). Berechnen Sie das Spektrum auch für das Von-Hann-gefensterte Segment:

```

'''
Spektrum eines 32 ms langen /u/-Segments des Wortes „Akustik“ berechnen
'''
# Lösung
# Zeitintervall bestimmen
n_start = int(1.42 / T_s * signalLength) # Startsample-Nummer
n_end = n_start + 512                  # Endsample-Nummer

# Audiosignal segmentieren
t_uSegment_s = t_s[n_start:n_end]
u_segment = audioSignal[n_start:n_end]

# Fenstern
u_segment_win = signal.hann(u_segment.size)*u_segment

# Spektrum berechnen
n_fft = 512 # Länge von FFT
f_uSegment_Hz = np.linspace(0, fs_Hz-fs_Hz/2-fs_Hz/n_fft, int(n_fft/2)) # Frequenzbereich

uSegment_fft = scipy.fftpack.fft(u_segment, n_fft)
uSegment_win_fft = scipy.fftpack.fft(u_segment_win, n_fft)
uSegment_fft_plot = np.abs(uSegment_fft[:len(f_uSegment_Hz)]) / int(n_fft/2)
uSegment_win_fft_plot = np.abs(uSegment_win_fft[:len(f_uSegment_Hz)]) / int(n_fft/2)

```

```

# Graphische Darstellung
plt.subplot(221)
plt.title('Audiosignal des /u/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_uSegment_s, u_segment)


plt.subplot(222)
plt.title('Amplitudenfrequenzgang des /u/-Segments')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('log|X(f)|')
plt.plot(f_uSegment_Hz, np.log(uSegment_fft_plot))

plt.subplot(223)
plt.title('Audiosignal des gefensterten /u/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_uSegment_s, u_segment_win)

plt.subplot(224)
plt.title('Amplitudenfrequenzgang des gefensterten /u/-Segments')
plt.xlabel('Frequenz [Hz]')
plt.ylabel('log|X(f)|')
plt.plot(f_uSegment_Hz, np.log(uSegment_win_fft_plot))

plt.gcf().set_size_inches(15, 10)
plt.show()

```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_27_0.png

Geben Sie sich nun auch das segmentierte Signal `u_segment` und das gefenstertere Signal `u_segment_win` mit den Funktionen "repeat_audio" und "overlap_audio" (10 ms-Wiederholdauer) aus:

```
'''
Audioausgabe
'''
# Lösung
T_overlap_ms = 10

playable_audio_u = overlap_audio(u_segment, T_overlap_ms, True)


playable_audio_u_win = overlap_audio(u_segment_win, T_overlap_ms, False)

t_range_s = np.linspace(0, (1024-1)/fs_Hz, 1024)
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('addiertes Audiosignal des /u/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_range_s, playable_audio_u[:1024])

plt.subplot(122)
plt.title('addiertes Audiosignal des gefensterten /u/-Segments')
plt.xlabel('Zeit [s]')
plt.ylabel('x(t)')
plt.plot(t_range_s, playable_audio_u_win[:1024])

plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_30_0.png

Jetzt können wir LP-Koeffizienten berechnen, um die Pole der Systemfunktion des Synthesefilters zu bestimmen. Dafür verwenden wir nun direkt eine Funktion des Moduls `librosa`, der LP-Koeffizienten sucht: [librosa.lpc\(y, order\)](#).

Dieser soll zunächst für das ungefensterte /a/-Segment "a_segment" mit einer Filterordnung der Größe $N = 12$ betrachtet werden.

```
'''
Die LP-Koeffizienten des /a/-Segments berechnen und die Frequenzgang bzw. Polverteilung
des Synthesefilters aufzeichnen
'''

# Lösung
N = 12
b, a = [1], librosa.lpc(a_segment, N)
fH_Hz, H = scipy.signal.freqz(b, a, fs=fs_Hz)
z, p, k = scipy.signal.tf2zpk(b, a)


# Modellordnung
# LP-Koeffizienten bestimmen
# Amplitudenfrequenzgang
# Null-Pol Verteilung
```

```
C:\Users\bernh\AppData\Local\Temp\ipykernel_31504\1322339488.py:7: FutureWarning: Pass
order=12 as keyword args. From version 0.10 passing these as positional arguments will
result in an error
    b, a = [1], librosa.lpc(a_segment, N)
                                # LP-Koeffizienten bestimmen
```

```
# Graphische Darstellung
plt.subplot(121)
plt.title('Amplitudenfrequenzgang des Synthesefilters für /a/ mit Ordnung=%d' %N)
plt.xlabel('Frequenz [Hz]')
plt.ylabel('$\log|H(e^{j\Omega})|$')
plt.plot(fH_Hz, np.log(np.abs(H)))

plt.subplot(122)
plt.title('Polverteilung des Synthesefilters mit Ordnung=%d' %N)
plt.xlabel('Real')
plt.ylabel('Image')
theta = np.arange(0, 2*np.pi, 0.01)
plt.plot(np.cos(theta), np.sin(theta), c='r', lw=0.2)
plt.plot(np.real(p), np.imag(p), 'x')
plt.axis("equal")

plt.gcf().set_size_inches(15, 4)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_33_0.png


Kritisch ist dabei die Wahl der „richtigen“ Modellordnung N . Die Unterschiede im Frequenzgang bei der Wahl unterschiedlicher Modellgrößen, soll nun für das /u/-Vokal-Segment genauer betrachtet werden. Erzeugen Sie dafür mittels for-Schleife die Amplitudenfrequenzgänge bei Wahl der Synthesefilterordnung auf 4., 8., 12., 20. und 24. Ordnung:

```
'''
Aufgabe: Die LP-Koeffizienten des /u/-Segments mit verschiedenen Ordnungen berechnen und
die Frequenzgang bzw. Null-Pol Verteilung des Synthesefilters aufzeichnen
'''


# Lösung
for N in (4, 8, 12, 20, 24):
    b, a = [1], librosa.lpc(u_segment, N)          # LP-Koeffizienten bestimmen
    fH_Hz, H = scipy.signal.freqz(b, a, fs=fs_Hz)  # Amplitudenfrequenzgang
    z, p, k = scipy.signal.tf2zpk(b, a)           # Null-Pol Verteilung

    # Graphische Darstellung
    plt.subplot(121)
    plt.title('Amplitudenfrequenzgang des Synthesefilters für /u/ mit Ordnung=%d' %N)
    plt.xlabel('Frequenz [Hz]')
    plt.ylabel('$\log|H(e^{j\Omega})|$')
    plt.plot(fH_Hz, np.log(np.abs(H)))
    plt.subplot(122)
    plt.title('Polverteilung des Synthesefilters mit Ordnung=%d' %N)
    plt.xlabel('Real')
    plt.ylabel('Image')
    theta = np.arange(0, 2*np.pi, 0.01)
    plt.plot(np.cos(theta), np.sin(theta), c='r', lw=0.2)
    plt.plot(np.real(p), np.imag(p), 'x')
    plt.axis("equal")
    plt.gcf().set_size_inches(15, 4)
    plt.show()
```


C:\Users\bernh\AppData\Local\Temp\ipykernel_31504\1368372881.py:8: FutureWarning: Pass order=4 as keyword args. From version 0.10 passing these as positional arguments will result in an error
b, a = [1], librosa.lpc(u_segment, N) # LP-Koeffizienten bestimmen

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_35_1.png


C:\Users\bernh\AppData\Local\Temp\ipykernel_31504\1368372881.py:8: FutureWarning: Pass order=8 as keyword args. From version 0.10 passing these as positional arguments will result in an error
b, a = [1], librosa.lpc(u_segment, N) # LP-Koeffizienten bestimmen

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_35_3.png


C:\Users\bernh\AppData\Local\Temp\ipykernel_31504\1368372881.py:8: FutureWarning: Pass order=12 as keyword args. From version 0.10 passing these as positional arguments will result in an error
b, a = [1], librosa.lpc(u_segment, N) # LP-Koeffizienten bestimmen

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_35_5.png

C:\Users\bernh\AppData\Local\Temp\ipykernel_31504\1368372881.py:8: FutureWarning: Pass order=20 as keyword args. From version 0.10 passing these as positional arguments will result in an error
b, a = [1], librosa.lpc(u_segment, N) # LP-Koeffizienten bestimmen

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_35_7.png

C:\Users\bernh\AppData\Local\Temp\ipykernel_31504\1368372881.py:8: FutureWarning: Pass order=24 as keyword args. From version 0.10 passing these as positional arguments will result in an error
b, a = [1], librosa.lpc(u_segment, N) # LP-Koeffizienten bestimmen

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_35_9.png


Um sich nun den Synthesefilter anzuhören, kann dieser invers Fouriertransformiert werden, um daraus die Impulsantwort zu erhalten:

```
'''
Impulsantwort des modellierten /u/-Vokalfilters mit der Ordnung N = 24
'''

# Lösung
# Impulsantwort h des Modellfilters H
h = scipy.fftpack.iffth(H, len(H))
t_syn_s = np.linspace(0, (len(H)-1)/(fs_Hz), len(H))
```

```
# Graphische Darstellung
plt.title('Impulsantwort des Synthesefilters für /u/ mit Ordnung=%d' %N)
plt.xlabel('Zeit [s]')
plt.ylabel('h [t]')

plt.plot(t_syn_s, np.real(h))
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/2.2 - LPC_solved_38_0.png

Erzeugen Sie nun ein Audiosignal, indem Sie die ersten 10 ms bzw. ~170 Samples (genauer [512:512+170]) mittels der "repeat_audio" abspielen.

```
'''
Audioausgabe
'''

# Lösung
# repeat_audio(np.real(h[Len(h)//2:]))
# repeat_audio(playable_audio_u)
# play_audio(playable_audio)
```

```
'\nAudioausgabe\n'
```

Zum Schluss soll nun mittels `interact_manual` Änderungen in der Modellordnung betrachtet werden:

```

'''
Interaktive LPC-Darstellung
'''
# Lösung

@interact_manual(N_i=(2, 50, 2), segment_type_i=[('a-Segment', 1), ('u-Segment', 2)],
output_sound_i = True, output_type_i=[('real', 1), ('real+win', 2), ('real+win+overlap',
3), ('syn+uncut', 4), ('syn+overlap', 5)], t_overlap_ms_i = (1, 15, 1))
def interactive_linear_sweep(N_i=10, segment_type_i = 'a-Segment', output_sound_i=True,
output_type_i = 'win+added', t_overlap_ms_i = 10):
    if segment_type_i == 1:
        segment = a_segment
    elif segment_type_i == 2:
        segment = u_segment

    # LP-Koeffizienten
    b, a = [1], librosa.lpc(segment, N_i) # LP-Koeffizienten bestimmen
    fH_Hz, H = scipy.signal.freqz(b, a, fs=fs_Hz) # Amplitudenfrequenzgang
    z, p, k = scipy.signal.tf2zpk(b, a) # Null-Pol Verteilung

    # Impulsantwort h des Modellfilters H
    h = scipy.fftpack.ifft(H, 512)
    t_s = t_s = np.linspace(0, len(H)/fs_Hz, len(H))

    # Audioauswahl
    if output_type_i == 1: # real + uncut
        chosen_audio = repeat_audio(segment, False)
    elif output_type_i == 2: # real + windowed
        chosen_audio = repeat_audio(segment*signal.hann(segment.size), False)
    elif output_type_i == 3: # real + windowed + overlap
        chosen_audio = overlap_audio(segment*signal.hann(segment.size), t_overlap_ms_i,
False)
    elif output_type_i == 4: # syn + uncut
        chosen_audio = repeat_audio(np.real(h), False)
    elif output_type_i == 5: # syn + overlap
        chosen_audio = overlap_audio(np.real(h), t_overlap_ms_i, False)

    # Graphische Darstellung
    plt.subplot(221)
    plt.title('Amplitudenfrequenzgang des Synthesefilters mit Ordnung=%d' %N_i)
    plt.xlabel('Frequenz [Hz]')
    plt.ylabel('$\log|H(e^{j\Omega})|$')
    plt.plot(fH_Hz, np.log(np.abs(H)))

    plt.subplot(222)
    plt.title('Polverteilung des Synthesefilters mit Ordnung=%d' %N_i)
    plt.xlabel('Real')
    plt.ylabel('Image')
    theta = np.arange(0, 2*np.pi, 0.01)
    plt.plot(np.cos(theta), np.sin(theta), c='r', lw=0.2)
    plt.plot(np.real(p), np.imag(p), 'x')
    plt.axis("equal")
    plt.gcf().set_size_inches(15, 4)

    plt.subplot(223)
    plt.title('Impulsantwort des Synthesefilters mit Ordnung=%d' %N_i)
    plt.xlabel('Zeit [s]')
    plt.ylabel('h [t]')
    plt.plot(t_s, np.real(h))

    plt.subplot(224)
    plt.title('Ausschnitt des Audiosignals')
    plt.xlabel('Zeit [s]')
    plt.ylabel('x [t]')
    plt.plot(t_s, chosen_audio[:512])
    plt.gcf().set_size_inches(20, 10)
    plt.show()

    # Audioausgabe
    if output_sound_i == True:
        play_audio(chosen_audio)

```

Dies ist nur eine rudimentäre Umsetzung eines synthetisierten Vokaltrakts, für die es diverse Verbesserungsmethoden gibt. Außerdem fällt direkt auf, dass sich die Grundfrequenz des synthetisierten Vokals (bzw. der "Pitch") ändert, wenn sich die Fensterlänge der überlappten Fenster ändert. Eine Alternative dazu bietet zum Beispiel der sog. PSOLA Algorithmus ("Pitch Synchronous Overlap Add"), mit dem das Audiosignal so manipuliert werden kann, dass die Aussprache schneller ist, aber die Grundfrequenz gleich bleibt. Für Interessierte wird auf diese Methoden in der Vorlesung "Sprachsynthese" im 9. Semester näher eingegangen.

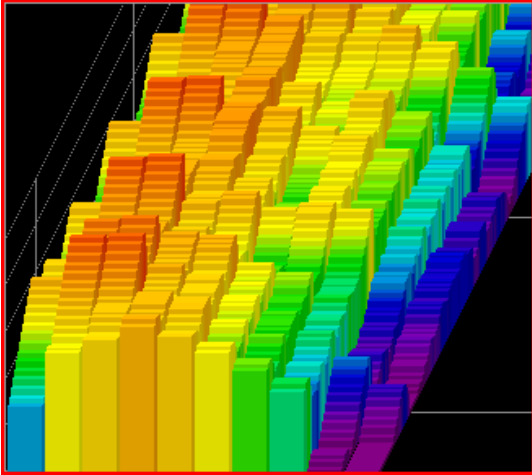
```
sa.stop_all()
```

References

1. [Emflazie, adaptivedigital](#)
2. [Linear predictive coding](#)
3. [Introduction - Linear predictive coding](#)

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)

3.1 - Short Time Fourier Transformation



Die Kurzzeit-Fourier-Transformation (Englisch: "short-time-fourier-tranform", kurz: STFT) ist eine Fourier-bezogene Transformation, mit der die Frequenzanteile und deren Phasenlage einzelner Zeitabschnitte eines Signals bestimmt werden. In der Praxis besteht das Verfahren zum Berechnen von STFTs darin, ein längeres Zeitsignal in kürzere Segmente gleicher Länge zu unterteilen und dann die Fourier-Transformation für jedes Segment separat zu berechnen. Die erzeugten Einzelspektren werden dann zur Analyse der spektralen Zusammensetzung über die Zeit in einem sog. **Spektrogramm** aneinandergesetzt.

Dieses Notebook soll Ihnen die Theorie hinter der STFT näher bringen und mit Anwendungsbeispielen zeigen, wann diese Analysetechnik genutzt werden kann.

Die Gliederung des Notebooks ist wie folgt aufgebaut:

- In *Abschnitt 1* werden einige Limitationen einer normalen Fouriertransformation bei instationären Signalen gezeigt.
- In *Abschnitt 2* wird die Fensterung und dadurch Segmentierung von instationären Signalen durchgeführt und dessen Frequenzgänge berechnet.
- In *Abschnitt 3* werden diese einzelnen Frequenzgänge grafisch zusammengefügt und somit das besagte Spektrogramm erzeugt.
- Im letzten *Abschnitt 4* wird die STFT auf verschiedene Audio-Beispiele angewendet.

Inhalt

	1. Nichtstationäre Signale
	2. Fensterung
	3. Kurzzeitspektrum
	4. Anwendungsbeispiele



1. Nichtstationäre Signale

Signale, die sich mit der Zeit ändern, werden als nichtstationäre Signale bezeichnet.

Die Fourier-Transformation ist v.a. für stationäre Signale geeignet ist, deren spektrale Zusammensetzung (Frequenz und Phasenlage) sich zeitlich nicht ändert. In der Praxis sind die Signale jedoch häufig nichtstationäre Signale, wodurch diese Bedingung nicht mehr erfüllt ist. Wenn das Spektrum über eine einzige Fourier-Transformation bestimmt wird, gehen nicht nur spektrale Charakteristika verloren, sondern u.U. auch ihre zeitliche Zuordnung (d.h. *wann* die Frequenzkomponente im Zeitsignal auftritt). Daher müssen bei nicht stationären Signalen die lokalen Eigenschaften berücksichtigt werden. Das nichtperiodische Signal wird deswegen in zeitlich stationär angenommene Signalabschnitte unterteilt.

Als Beispiel sollen zunächst einige Zeitsignale erzeugt und deren Frequenzgänge analysiert werden. Importieren Sie dafür zunächst alle für dieses Notebook notwendigen Module und installieren Sie gegebenenfalls fehlende Module via `pip install ...`

```
'''
Import externer Module
'''
# Lösung

import numpy as np
import cmath

from scipy import fftpack, signal

import matplotlib.pyplot as plt
from matplotlib import cm

from mpl_toolkits.mplot3d import axes3d

import librosa
import librosa.display

from scipy.io import wavfile

import IPython.display as ipd
from ipywidgets import interact, FloatSlider
```

Falls verhindert werden soll, dass große Cell-Outputs (Plots, Interaktive Grafiken etc.) mit einem Scrollbar versehen werden, kann diese Zeile ausgeführt werden:

```
%%javascript
IPython.OutputArea.prototype._should_scroll = function(lines) {
    return false;
}
```

In diesem Notebook soll zudem das Konzept des [Dictionary](#) zur übersichtlichen Speicherung von Daten eingeführt werden. Ein Dictionary wird (vergleichbar mit Listen und Tupeln) mit geschweiften Klammern `{ ... }` erzeugt. Ein Dictionary speichert alle möglichen Datentypen, die man mit einem Schlüsselwort wieder aus diesem aufrufen kann. Die Syntax ist dabei wie folgt:

```
{ Key_1: Value_1,
  Key_2: Value_2,
  ... }
```

Der Schlüssel muss dabei ein String (also `".."` oder `'..'`) sein.

Als Beispiel sollen alle Informationen einer Audiodatei gespeichert werden. Diese kann beinhalten:

- Strings: Speicherort, Titel, Autor*in
- Zahlen (float, int,...): Dauer, Abtastfrequenz
- Listen bzw. Arrays... (auch Numpy-Array): Audiospur(en)
- ...

Speichern Sie im Folgenden nun Ihr (vorgegebenes) Lieblingslied in ein Dictionary:

```
'''
Erzeugen und Speichern von Dictionaries
'''

# Lösung
author = "Wolfgang Petry"           # Speichern Sie 'author' unter dem Schlüssel
'Autor_in' in dict_hit_no1
title = 'Wahnsinn'                  # Speichern Sie 'title' unter dem Schlüssel 'Titel'
in dict_hit_no1
duration_s = 319.25                 # Speichern Sie 'duration_s' unter dem Schlüssel
'Spieldauer' in dict_hit_no1
fs_Hz = 44100                       # Speichern Sie 'fs_Hz' unter dem Schlüssel
'Abtastfreq Links' in dict_hit_no1
track_left = [1, 0.458, 0.23, -1]   # Speichern Sie 'track_left' unter dem Schlüssel
'Spur Links' in dict_hit_no1
track_right = [1, 0.5, 0.02, 1]      # Speichern Sie 'track_right' unter dem Schlüssel
'Spur rechts' in dict_hit_no1

dict_hit_no1 = {'Autor_in': author, 'Titel': title, 'Spieldauer': duration_s,
'Abtastfreq': fs_Hz, 'Spur links': track_left, 'Spur rechts': track_right}
```

```
print("Einträge im Dictionary: {}".format(len(dict_hit_no1)))
```

Einträge im Dictionary: 6

Um wieder an die in dem Dictionary gespeicherten Daten zu kommen, kann man diese auf zwei Weisen mit dem Schlüssel-bzw. Key-String aufrufen:

`dict_hit_no1['key_string']` oder `dict_hit_no1.get('key_string')`

Testen Sie beide Verfahren nun aus:

```
'''
Laden von Daten aus Dictionaries
'''

# Lösung

print("Interpet: {}, Dauer: {} Sekunden".format(dict_hit_no1['Autor_in'],
dict_hit_no1.get('Spieldauer')))
```

Interpet: Wolfgang Petry, Dauer: 319.25 Sekunden

Um nun das angefangene Dictionary zu erweitern, kann dies mit dem Befehl `dict_XY.update(...)` erfolgen. Dabei kann man auch ein anderes Dictionary mit einfügen. Wenn sie einen Schlüssel nutzen, den es schon im Dictionary gab, werden die Daten überschrieben (nutzen Sie dies zur Aktualisierung der Daten).

Aktualisieren Sie nun das Dictionary Ihres Lieblingsliedes mit einem Genre und einer besseren Audiospur:

```
'''
Erweiterung und Aktualisierung von Dictionaries
'''

# Lösung
genre = "1a Kuscheelrock"           # Speichern Sie 'genre' unter dem Schlüssel
'Genre' in dict_mus_1
track_left_np = np.array([1, 0.458, 0.23, -1]) # Überschreiben Sie 'Spur links' in
dict_mus_1
track_right_np = np.array([1, 0.5, 0.02, 1])    # Überschreiben Sie 'Spur rechts' in
dict_mus_1
dict_better_version = {'Spur links': track_left_np, 'Spur rechts': track_right_np}

# Speichern Sie 'Genre' ins Dictionary
dict_hit_no1.update({'Genre': genre})

# Updaten Sie das Unterdictionary 'dict_better_version' in 'dict_hit_no1'
dict_hit_no1.update(dict_better_version)
```

```
print("Einträge im Dictionary: {}".format(len(dict_hit_no1)))
```

Dictionaries können genutzt werden, um den Code übersichtlicher zu gestalten. Im Folgenden sollen jetzt mehrere instationäre Testsignale erzeugt werden, welche aus verschiedenen Sinusschwingungen zusammengesetzt sind.

```
'''
Erzeugen von nichtstationären Signalen
'''

# Lösung
# Initialisierung der Variablen
fs_Hz = 400 # Abtastfrequenz
L_periods = 10 # Länge des Signals in Perioden

# Vier Sinus-Signale:
f1_Hz = 9;
f2_Hz = 10;
f3_Hz = 15;
f4_Hz = 20;
t1_s = np.arange(0, 1/f1_Hz*L_periods, 1/fs_Hz);
t2_s = np.arange(0, 1/f2_Hz*L_periods, 1/fs_Hz);
t3_s = np.arange(0, 1/f3_Hz*L_periods, 1/fs_Hz);
t4_s = np.arange(0, 1/f4_Hz*L_periods, 1/fs_Hz);

seg1 = 2*np.sin(2*np.pi*f1_Hz*t1_s)
seg2 = 5*np.sin(2*np.pi*f2_Hz*t2_s)
seg3 = 7*np.sin(2*np.pi*f3_Hz*t3_s)
seg4 = 4*np.sin(2*np.pi*f4_Hz*t4_s)

# Drei verschiedene Signale mit unterschiedlichen Segmentzusammensetzungen
s1 = np.concatenate((seg1, seg2, seg3, seg4))
s2 = np.concatenate((seg3, seg4, seg2, seg1))
s3 = np.concatenate((seg2, seg1, seg4, seg3))
t_s = np.linspace(0, len(s1)/fs_Hz, len(s1))

signalLength = len(s1);
print("Länge des Signals: {}".format(signalLength))
Nfft = int(2**(np.floor(np.log(signalLength)/np.log(2))+1)) # Länge der Fft
print("Länge der FFT: {}".format(Nfft))
f_Hz = np.arange(0, fs_Hz, fs_Hz/Nfft) # Frequenzachse

# Initialisierung und füllen des Dictionaries:
dict_s = {'s1': s1, 's2': s2, 's3': s3}
dictIndex = 0;
for s in (s1, s2, s3):
    dictIndex += 1

# Transformation in den Frequenzbereich:
S = fftpack.fft(s, Nfft) # Frequenzgänge
dict_s.update({'S{0}'.format(dictIndex) : S})

# Graphische Darstellung
plt.subplot(131)
plt.title('Signal {0} {1}'.format(dictIndex, dictIndex))
plt.xlabel('Zeit [s]')
plt.ylabel('s(t)')
plt.plot(t_s, s)
plt.subplot(132)
plt.title('Frequenzgang des Signals {0} {1}'.format(dictIndex, dictIndex))
plt.xlabel('Frequenz [Hz]')
plt.xlim(0, 40)
plt.ylabel('|S(f)|')
Sabs = np.abs(S)
plt.plot(f_Hz, Sabs)
line1, = plt.plot([f1_Hz, f1_Hz], [0, max(Sabs)], '-r', label='f=' + str(f1_Hz) + ' Hz')
line2, = plt.plot([f2_Hz, f2_Hz], [0, max(Sabs)], '-g', label='f=' + str(f2_Hz) + ' Hz')
line3, = plt.plot([f3_Hz, f3_Hz], [0, max(Sabs)], '-b', label='f=' + str(f3_Hz) + ' Hz')
line4, = plt.plot([f4_Hz, f4_Hz], [0, max(Sabs)], '-m', label='f=' + str(f4_Hz) + ' Hz')
plt.gcf().set_size_inches(20, 5)
plt.legend(handles=[line1, line2, line3, line4])
plt.subplot(133)
plt.title('Phasengang des Signals {0} {1}'.format(dictIndex, dictIndex))
plt.xlabel('Frequenz [Hz]')
plt.xlim(0, fs_Hz/5)
plt.ylabel('$\phi(f)$ [rad]')
SPhase = np.angle(S)
plt.plot(f_Hz, SPhase)
plt.show()
```


Länge des Signals: 1312
Länge der FFT: 2048

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_15_1.png
C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_15_2.png
C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_15_3.png

Die absoluten Frequenzgänge $|S(f)|$ sind nahezu identisch und enthalten keine erkennbare Information mehr darüber, in welchem Segment welche harmonische Schwingung lag, während man die jeweiligen Amplitudenhöhen immer ungenauer werden, je kleiner die Signalabschnitte werden (durch die Wahl von L_{periods}).

Um alle Ergebnisse besser zu inspizieren, werden nun alle Ergebnisse über die `interact`-Funktion interaktiv dargestellt:

```
'''
Interaktive Grafik
'''
@interact( domain = [('1. Zeitsignal', 's{ }'), ('2. Frequenzsignal', 'S{ }')], \
            signal1 = True, signal2 = True, signal3 = True, type= [('Realteil', 1), \
            ('Imaginärteil', 2), ('Betrag', 3), ('Phase', 4)], \
            lower_bound = FloatSlider(min=0,max=Nfft, step=1, continuous_update=False), \
            upper_bound = FloatSlider(min=1,max=Nfft, step=1, continuous_update=False) )
def interactive_signals(domain = 'Zeitsignal', signal1 = True, signal2 = True, signal3 =
True, \
                        type = 'abs', lower_bound = 0, upper_bound = Nfft):
    s = []
    col = []
    legend_label = []

    if signal1:
        s.append(dict_s.get(domain.format(1)))
        col.append('y')
        legend_label.append('Segment 1')
    if signal2:
        s.append(dict_s.get(domain.format(2)))
        col.append('g')
        legend_label.append('Segment 2')
    if signal3:
        s.append(dict_s.get(domain.format(3)))
        col.append('r')
        legend_label.append('Segment 3')

    m = []
    for n in s:
        if type == 1:
            m.append(n.real)
        elif type == 2:
            m.append(n.imag)
        elif type == 3:
            m.append(abs(n))
        elif type == 4:
            if domain == 's{ }':
                print('Phase ist im Zeitbereich nicht möglich. Es wird der Betrag
angezeigt.')
            m.append(abs(n))
        else:
            m.append(np.angle(n))
    s = m

    # Grafische Darstellung
    i = 0
    for n in s:
        plt.plot(n, color = col[i], label=legend_label[i])
        plt.xlim(lower_bound, upper_bound)
        i+=1
    plt.legend()
    plt.gcf().set_size_inches(20, 5)
    plt.show()
```



2. Fensterung

Im folgenden Abschnitt soll die Fensterung näher betrachtet werden, mit der das nichtstationäre Signal in quasi-stationäre Signalabschnitte segmentiert wird.

Ein bloßes Ausschneiden des Signals würde nicht nur potentiell Sprünge im periodisch forgesetzten Zeitsignal erzeugen, was bereits in Notebook 2.2 thematisiert wurde, sondern ist auch gleichbedeutend mit einer Rechteckfensterung. Anders formuliert ist das einfache Ausschneiden eines Signalabschnittes gleichbedeutend mit der Multiplikation mit einem Rechteckfenster im Zeitbereich, welches einer Faltung mit einer Spaltfunktion im Frequenzbereich und dadurch ein Verwischen der Spektrallinien zur Folge hat. Deshalb ist es wichtig, eine geeignetere Fensterfunktion $h(t)$ zu wählen (z.B. ein von-Hann-Fenster). Die Fensterlänge T wird entsprechend der Länge des quasi-periodischen Signalsegments gewählt. Ein weiterer Parameter ist der Grad der Überlappung aufeinanderfolgender Signalabschnitte (siehe Vorlesung Thema4, Folie 15 ff.).

Nutzen Sie die vorgegebene Funktion `live_plot()`, um sich einen Durchlauf der Segmentierung, Fensterung und Fouriertransformation auf das Signal `s1` anschauen zu können. Durch den Aufruf von `ipd.clear_output()` wird der vorherige Plot gelöscht, auf dessen Stelle ein neuer Plot eingefügt wird. (Anmerkung: für richtige Animationen von Plots bietet sich auch die [animation](#) API von Matplotlib an. Da dies aber nicht der Inhalt dieses Notebooks ist, soll die Animation simpel gehalten werden). Die Funktionsargumente sind

- die Zeitachse `t_s`
- das Audiosignal `audioSignal`
- die Zeitachse eines gefensterten Signalausschnittes `tSegment_s`
- der gefensterte Signalausschnitt `winAudioSignalSegment`
- die Frequenzachse des Spektrums `fSegment_Hz`
- das Spektrum selbst `Ssegment`
- der Schritt, an dem sich das Spektrogramm gerade befindet `stepIndex`
- Limits für die Achsen `xLim_max` & `yLim_max`

```
'''
Funktion definieren: Dynamische Darstellung
'''

def live_plot(t_s, audioSignal, tSegment_s, winAudioSignalSegment, fSegment_Hz, Ssegment,\
              stepIndex, xLim_max= 50, yLim_max= 80):

    plt.ion()
    ipd.clear_output(wait=True)
    fig, (ax1, ax2) = plt.subplots(1, 2)

    ax1.set_title('Fensterung des Signals')
    ax1.set_xlabel('Zeit [s]')
    ax1.set_ylabel('s(t)')
    ax1.plot(t_s, audioSignal)
    ax1.plot(tSegment_s, winAudioSignalSegment)

    ax2.set_title('Frequenzgang des gefensterten Signalsegments %d' %stepIndex)
    ax2.set_xlabel('Frequenz [Hz]')
    ax2.set_ylabel('|S(f)|')
    ax2.plot(fSegment_Hz, Ssegment)
    plt.xlim(0, xLim_max)
    plt.ylim(0, yLim_max)

    fig.set_size_inches(20, 5)
    plt.show()
```

1. Normalisieren Sie zunächst das Signal `s1`, dass dessen Maximalwert den Wert 1 beträgt.
2. Erzeugen Sie ein [Tukey-Fenster](#) mit einem alpha-Parameter von 0.7, dessen Fensterlänge `T_samples` 2^x Samples beträgt (z.B. $x=7$ für 128 samples). Dies soll auch die Länge der FFT sein.
3. Stellen Sie eine Überlappung des Zeitfensters `overlap_samples` auf die Hälfte oder zwei Drittel der Fensterlänge `Twin_samples` ein. Die Überlappung und der Vorschub zweier Fenster sind voneinander abhängig: Anzahl an Segmenten = (Fensterlänge - Overlap)/Vorschub (Vorschub ist

hier als `slide_samples` bezeichnet).

4. Erzeugen Sie eine for-Schleife, in welcher jedes Segment gefenstert und Fourier-transformiert wird. Achten Sie dabei darauf, dass die Schleife bei Erreichen des Signalendes korrekt beendet wird und nicht auf Indices außerhalb des Arrays zugegriffen wird. Berechnen Sie daher vorher die Anzahl an möglichen Segmenten (`numSegments`).
5. Fügen Sie zum Schluss die Funktion `live_plot` in die for-Schleife ein und übergeben Sie diesem die benötigten Parameter und Werte.

```
'''
Dynamische Fensterung des Signals 's1'
'''

# Lösung
%matplotlib inline

# Signal s1 aus dem dict() auslesen und normlisieren:
s = dict_s.get('s1')/np.max(dict_s.get('s1'))
T_samples = np.size(s)
T_s = T_samples/fs_Hz
t_s = np.arange(0, T_s, 1/fs_Hz)


# Erzeugung des Fensters
Nfft = 2**7;                                # FFT Länge
Twin_samples = Nfft                          # Fensterlänge in samples
overlap_samples = Twin_samples//4           # Überlappung zwischen zwei Segmenten
slide_sample = Twin_samples - overlap_samples # Vorschub eines Abschnittes pro Segment
window = signal.get_window('tukey', 0.7), Nfft # Fenster
f_Hz = np.arange(0, fs_Hz, fs_Hz/Nfft)

# Erzeugung des Segments, Fouriertransformation dessen und grafische Anzeige:
numSegments = (T_samples - overlap_samples)//slide_sample

# Überprüfung der Parameter:
print("Länge des Signals: {} samples".format(T_samples))
print("Fenster/FFT Länge: {} samples".format(Twin_samples))
print("Vorschub: {} samples".format(overlap_samples))
print("Anzahl an Segmenten: {}".format(numSegments))

for seqIndex in range(numSegments):
    print("Segment {}".format(seqIndex))
    tSegment_s = t_s[seqIndex*slide_sample: seqIndex*slide_sample + Twin_samples]
    sSegment = s[seqIndex*slide_sample: seqIndex*slide_sample + Twin_samples]
    winsSegment = sSegment * window
    winS = np.abs(fftpack.fft(winsSegment, Nfft))

    # Graphische Darstellung
    live_plot(t_s, s, tSegment_s, winsSegment, f_Hz, winS, seqIndex, xLim_max=50,
              yLim_max=90)
```

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_21_0.png

Die Wahl der Fensterfunktion $h(t)$ und der Zeitfensterlänge `Twin_samples` muss problemspezifisch erfolgen. Im Allgemeinen kann ein kurzes Zeitfenster eine höhere Auflösung des zeitlichen Zustands und ein langes Fenster eine höhere Auflösung im Frequenzbereich bieten. Zwischen beidem muss der beste Kompromiss gesucht werden.



3. Kurzzeitspektrum


Setzt man den Signalabschnitt als zu transformierende Funktion in das Fourier-Integral ein, erhält man die Hintransformationsgleichung der STFT wie folgt:
$$X(\omega, t) = \int_{-\infty}^{\infty} x(\tau) h(t - \tau) e^{-j\omega \tau} d\tau$$
 Das dabei entstehende Spektrum hat außer der Kreisfrequenz ω nun auch noch den Analysezeitpunkt t als Variable. Da gewöhnlich vorausgesetzt wird, dass die Zeitfensterlänge T endlich ist, können die Integrationsgrenzen entsprechend eingeeengt werden.

Zur Berechnung der STFT können direkt Funktionen aus installierten Bibliotheken genutzt werden. Hier sollen die Funktionen `signal.stft()` und `librosa.stft()` genutzt werden.

```
'''
Spektrogramm mit scipy.signal.stft()
'''
for i in range(1,4):
    f, t, Zxx = signal.stft(dict_s.get('s{}'.format(i)), fs=fs_Hz, window=('tukey',
0.7), nperseg=200) # Frequenzgänge
    dict_s.update({'s{}_scipy_stft'.format(i): (f, t, Zxx)})

    # Grafische Darstellung:
    plt.subplot(1,3,i)
    plt.pcolormesh(t, f, np.abs(Zxx), shading = 'auto')
    plt.title('STFT Magnitude des Signals %d' %i)
    plt.xlabel('Zeit [s]')
    plt.ylabel('Frequenz [Hz]')
    plt.ylim(0, 50)
    plt.colorbar()

plt.gcf().set_size_inches(20, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_24_0.png

Durch die Darstellung der Signale im Spektrogramm wird nun aus dem Amplitudenfrequenzgang deutlich klarer, welche Frequenzelemente sich in welchen Zeitabschnitten befinden. Über die Fensterlänge `nperseg` lässt sich die Frequenzauflösung auf Kosten der Zeitauflösung verbessern. Wird `nperseg` klein genug gewählt (z.B. = 50), wird sogar die Periodizität des Zeitsignals sichtbar. Die Frequenzauflösung ist in diesem Falle dann aber sehr grob.

Alternativ kann das Spektrogramm mit den Funktionen [signal.spectrogram\(\)](#) oder [librosa.display.specshow\(\)](#) berechnet werden. Beide Funktionen sollten identische Ergebnisse liefern, lediglich standard colormap ist unterschiedlich.


```
'''
Spektrogramm mit Librosa.stft()
'''

windowLength = 200;
for i in range(1,4):
    S = np.abs(librosa.stft(dict_s.get("s{}".format(i)), n_fft=windowLength, \
                                win_length=windowLength, window=('tukey', 0.7)))

    # (Optional) in dB konvertieren:
    # S_db = librosa.amplitude_to_db(S)

    # Graphische Darstellung
    plt.subplot(1,3,i)
    librosa.display.specshow(S, sr=fs_Hz, x_axis='s', y_axis='hz')
    plt.title('Spektrogramm des Signals %d' %i)
    plt.xlabel('Zeit [s]')
    plt.ylabel('Frequenz [Hz]')
    plt.ylim(0, 50)
    plt.colorbar()

plt.gcf().set_size_inches(20, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_26_0.png

Das Spektrogramm kann auch dreidimensional geplottet werden:


```
# 3D-Visualisierung

fig = plt.figure()

for plotIndex in range(1,4):
    f_Hz, t_s, magnitude = dict_s.get('s{}_scipy_stft'.format(plotIndex)) # STFT
    fLimits_Hz = (f>0) & (f<50) # Frequenzgrenze
    f_Hz = f[fLimits_Hz] # Frequenzbereich
    magnitude = magnitude[fLimits_Hz]

    # Graphische Darstellung
    ax = fig.add_subplot(1,3,plotIndex, projection='3d')
    ax.set_title('3D-Visualisierung des Signals %d' %plotIndex)
    ax.set_xlabel('Zeit [s]')
    ax.set_ylabel('Frequenz [Hz]')
    ax.set_zlabel('Amplitude')
    ax.plot_surface(t[None, :], f_Hz[:, None], abs(magnitude), cmap=cm.cool)

plt.gcf().set_size_inches(15, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_28_0.png

4. Anwendungsbeispiele

Für die STFT gibt es viele Anwendungsmöglichkeiten, wie zum Beispiel für die Analyse von Sprache. In diesem Abschnitt soll die STFT zur Analyse von Musiksignalen verwendet werden. Dazu soll zunächst eine Funktion definiert werden, mit der Informationen aus einer .wav-Datei importiert und als Dictionary abgespeichert werden können.

```
'''
Funktion definieren: wav-Datei Laden
'''

# Lösung
def wav_data(name, file):

    fs_Hz, s = wavfile.read(file) # Abstrastfrequenz, Signal
    if len(np.shape(s))>1:
        s = s[:,0] # Linken Kanal speichern
    s = s/np.max(np.abs(s)) # Signal normalisieren
    L = len(s) # Signal-Länge
    N = int(2**(np.floor(np.log2(L))+1)) # FFT-Länge
    T_s = L/fs_Hz - 1/fs_Hz # Zeitdauer
    t_s = np.linspace(0, T_s, L) # Zeitbereich

    # alle benötigte Daten in Dictionary speichern
    data = {'name':name, 'signal':s, 'fs_Hz':fs_Hz, 'signalLength':L, 't_s':t_s}

    return data
```

- **Beispiel 1: missing fundamental / Die fehlende Grundfrequenz**

Unsere Wahrnehmung von Akustik wird nicht nur von den einzelnen Frequenzkomponenten innerhalb eines Tons bestimmt, sondern u.a. auch von den Abständen der Obertöne/Harmonischen zueinander. Dies führt zu einem interessanten Phänomen des sog. "Residualtons" (im Englischen etwas treffender "missing fundamental"), bei dem zwei Töne, von denen einer keine (oder nur eine sehr gering ausgeprägte) Grundfrequenz aufweist, trotzdem sehr ähnlich klingend wahrgenommen werden.

Laden sie dazu die beiden Audiodateien `toneF.wav` und `toneF_noFund.wav` ein, anhand dieser Effekt demonstriert werden soll.

```
'''
toneF.wav Laden und abspielen
'''

# Lösung
file = 'data/toneF.wav'
# Akustische Ausgabe
toneF = wav_data('Ton F', file)
# Laden von toneF
ipd.Audio(file)
```

0:00 / 0:00

```
'''
toneF_noFund.wav Laden und abspielen
'''

# Lösung
file = 'data/toneF_noFund.wav'
# Laden von toneF_noFund mit dem Namen 'Ton F ohne Fundament'
toneF_noFund = wav_data('Ton F ohne Fundament', file)
# Akustische Ausgabe
ipd.Audio(file)
```

0:00 / 0:01

Beide Töne sollten relativ ähnlich klingen, vor allem von der wahrgenommenen Tonhöhe. Ein Blick auf das Spektrogramm verrät aber, dass im zweiten Falle die Grundfrequenz fehlt. Erstellen Sie dazu das Spektrogramm beider Audiosignale mit `librosa`. Skalieren Sie die Frequenzachse logarithmisch ([librosa.amplitude_to_db\(\)](#)) Um die Frequenzkomponenten klarer darzustellen.


```
'''
toneF und toneF_NoFund über Spektrogramm darstellen
'''

# Lösung
i = 0
for tone in (toneF, toneF_noFund):

    S = abs(librosa.stft(tone['signal']))
    S_db = librosa.amplitude_to_db(S) # in dB konvertieren

    # Graphische Darstellung
    i += 1
    plt.subplot(1, 2, i)
    librosa.display.specshow(S_db, sr=tone['fs_Hz'], x_axis='s', y_axis='hz',
cmap=cm.gist_heat)
    plt.title('Spektrogramm von %s' %tone['name'])
    plt.xlabel('Zeit [s]')
    plt.xlim(0, 0.5)
    plt.ylabel('Frequenz [Hz]')
    plt.ylim(0, 1500)
    plt.colorbar()

plt.gcf().set_size_inches(20, 5)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_35_0.png

- **Beispiel 2: Melodie aus verschiedenen Instrumenten**

Als ein weiteres Beispiel für die Nützlichkeit von Spektrogrammen sollen die beiden Audiodateien `melodie_klavier.wav`, `melodie_geige.wav` sowie `melodie_stimme.wav` untersucht werden. Laden Sie diese ein. Die Signale stellen dieselbe Melodie dar, aber mit unterschiedlichen Instrumenten bzw. gesungen (Vocoder).

```
'''
melodie_klavier.wav laden und abspielen
'''

file = 'data/melodie_klavier.wav'
melodyPiano = wav_data('Klaviermelodie', file)
ipd.Audio(file)
```

0:00 / 0:09

```
'''
melodie_geige.wav laden und abspielen
'''

file = 'data/melodie_geige.wav'
melodyViolin = wav_data('Geigemelodie', file)
ipd.Audio(file)
```

0:00 / 0:10

```
'''
melodie_stimme.wav laden und abspielen
'''

file = 'data/melodie_stimme.wav'
melodyVoice = wav_data('Stimme', file)
ipd.Audio(file)
```

0:00 / 0:10

Stellen Sie nur das Spektrogramm zur Analyse der drei Signale dar. Variieren Sie FFT Länge, Fenstertyp und Überlappung (Achtung, der Grad der Überlappung wird in Anzahl an Abtastwerten abgegeben, nach welchen sich die Fensterung wiederholt, nicht relativ zur FFT Länge).

```
'''
Kurzzeitspektrum
'''


numSamplesPerSegment = 512
numSamplesOverlap = numSamplesPerSegment/4

#Lösung
plotIndex = 0
for melody in (melodyPiano, melodyViolin, melodyVoice):

    plotIndex += 1
    f_Hz, t_s, magnitude = signal.stft(melody['signal'], melody['fs_Hz'], window='hann', \
                                       nperseg=numSamplesPerSegment, noverlap=numSamplesOverlap)

    # Graphische Darstellung
    plt.subplot(3, 1, plotIndex)
    plt.pcolormesh(t_s, f_Hz, np.abs(magnitude), shading = 'auto')
    plt.title('STFT Magnitude der %s' %melody['name'])
    plt.xlabel('Zeit [s]')
    plt.xlim(0, 8)
    plt.ylabel('Frequenz [Hz]')
    plt.ylim(0, 3000)
    plt.colorbar()

plt.gcf().set_size_inches(20, 20)
plt.show()
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_41_0.png

Wie im Spektrogramm gut zu sehen ist, sind die Töne der einzelnen Instrumente aus einer Grundfrequenz und mehreren Harmonischen aufgebaut, die immer Vielfache der Grundfrequenz sind. Die höherfrequenten Beiträge dieser "Obertöne" lassen verschiedene Musikinstrumente unterschiedlich klingen.

• Beispiel 3: Spektrogramm vs. Musiknoten

Nun analysieren Sie bitte anhand der nun kennengelernten Verfahren das Audiosignal von `musik.wav`[9]:

```
'''
musik.wav laden und abspielen
'''

# Lösung
file = 'data/musik.wav'
music = wav_data('Musiksignal', file)
ipd.Audio(file)
```

0:00 / 0:12

Führen Sie zunächst eine dynamische Fensterung (wie in Abschnitt 2) mit folgenden Eigenschaften durch,

- Zeitfenstertyp: Blackman,
- Zeitfensterlänge: 2^{14} Samples,
- Vorschub: 50 % der Fensterlänge,

um noch einmal den Prozess der Erstellung des Spektrogramms zu visualisieren (die Fensterlänge von $2^{14} = 16384$ Samples ist relativ groß und wird nur für diese Visualisierung gewählt, damit das Plotten nicht zu lange dauert). Gehen sie dabei wie in Abschnitt 2 vor. Ggf. können Sie die Routine zur Berechnung der Zeitversetzung, Fensterung und Fouriertransformation der Signalabschnitte auch als eigene Funktion kapseln.

```
'''
Aufgabe: Dynamische Fensterung darstellen
'''

# Lösung
%matplotlib inline

# Signal importieren und normalisieren
audioSignal = music['signal'] / np.max(music['signal'])
fs_Hz = music['fs_Hz']
T_samples = np.size(audioSignal)
T_s = T_samples/fs_Hz
t_s = np.arange(0, T_s, 1/fs_Hz)

# Erzeugung des Fensters
Nfft = 2**14; # FFT Länge
Twin_samples = Nfft # Fensterlänge in samples
overlap_samples = Twin_samples//4 # Überlappung zwischen zwei Segmenten
slide_samples = Twin_samples - overlap_samples # Vorschub eines Abschnittes pro Segment
window = signal.get_window(('tukey', 0.7), Nfft) # Fenster
f_Hz = np.arange(0, fs_Hz, fs_Hz/Nfft)

# Erzeugung des Segments, Fouriertransformation dessen und grafische Anzeige:
numSegments = (T_samples - overlap_samples)//slide_samples

for seqIndex in range(numSegments):
    print("Segment {}".format(seqIndex))
    tSegment_s = t_s[seqIndex*slide_samples: seqIndex*slide_samples + Twin_samples]
    sSegment = audioSignal[seqIndex*slide_samples: seqIndex*slide_samples + Twin_samples]
    winsSegment = sSegment * window
    winS = np.abs(fftpack.fft(winsSegment, Nfft))/Nfft

    # Grafische Darstellung:
    live_plot(t_s, audioSignal, tSegment_s, winsSegment, f_Hz, winS, seqIndex,
              xLim_max=fs_Hz/10, yLim_max=0.1)
```

 C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-

sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_45_0.png


Auch hier lohnt die Darstellung des kurzen Musikabschnittes als Spektrogramm. Variieren Sie die Parameter wie Fenstertyp, Überlappung, Fensterlänge usw.


```
'''
Aufgabe: Spektrogramm darstellen
'''

# Lösung
S = abs(librosa.stft(music['signal'], win_length = 1024))
S_db = librosa.amplitude_to_db(S)
T_samples = np.size(S_db)
# Einige interessante Frequenzen:
f1_Hz = 700
f2_Hz = 950
f3_Hz = 1050

# Grafische Darstellung:
librosa.display.specshow(S_db, sr=music['fs_Hz'], x_axis='s', y_axis='hz', cmap=cm.bwr)
freqLine1, = plt.plot([0, T_samples],[f1_Hz, f1_Hz], '-k', label='f1_Hz=' + str(f1_Hz) +
'Hz')
freqLine2, = plt.plot([0, T_samples],[f2_Hz, f2_Hz], '-k', label='f2_Hz=' + str(f2_Hz) +
'Hz')
freqLine3, = plt.plot([0, T_samples],[f3_Hz, f3_Hz], '-k', label='f3_Hz=' + str(f3_Hz) +
'Hz')
plt.title('Spektrogramm des Musiksignals')
plt.xlabel('Zeit [s]')
plt.ylabel('Frequenz [Hz]')
plt.ylim(0, 1500)
plt.legend(handles=[freqLine1, freqLine2, freqLine3])
plt.colorbar()

plt.gcf().set_size_inches(20, 5)
plt.show()
```

C:/Users/bernh/Desktop/Uni/Uni 8. SS/SHK_AIS/tud-
sv/sv_book_solved/_build/jupyter_execute/Einführung in Python/3.1 - STFT_solved_48_0.png

Im Spektrogramm sind z.B. die Frequenzanteile um 700 Hz, 950 Hz, 1050 Hz deutlich zu sehen, die zu den exakten Tonhöhen $F_5 = 698.46$ Hz, $B^b_5 = 932.33$ Hz und $C_6 = 1046.50$ Hz korrespondieren. Weitere Beziehungen zwischen Tonhöhen und Frequenzen sind in der Tabelle unter Referenz [7] zu finden.



References

1. Titelbild von [Ethan Weil](#)
2. [Short-time Fourier transform](#)
3. Lehrbuch: [Intelligente Signalverarbeitung 1 - Signalanalyse](#)
4. Video: [The Short Time Fourier Transform | Digital Signal Processing](#)
5. [The Missing Fundamental](#)
6. [Music Feature Extraction in Python](#)
7. [Frequencies for equal-tempered scale](#)
8. Referenz von Colormap: [matplotlib.cm](#)
9. [Wiz Khalifa - See You Again ft. Charlie Puth](#)

Notebook erstellt von Arne-Lukas Fietkau, Yifei Li und [Christoph Wagner](#)