Hochschule München
Studiengang Informatik

Angewandte Mathematik
Übungsblatt 2

Bastian Kersting, Michael Schober, Elena Lilova IF2B

May 28, 2019

Contents

1 Aufgabe: Komplexe Zahlen

Ziel der ersten Aufgabe ist es folgende Summe unter Benutzung der Polarform einer komplexen Zahl zu berechnen:

$$\sum_{k=0}^{30}(1.1 - 02.i)^k$$

Die Bearbeitung der Aufgaben erfolgte in Python. Zunächst wird die komplexe Zahl in Polarform dargestellt.
Für die Umwandlung von der kartesischen in die Polarform, müssen die Länge des Vektors r sowie der zugeörige Winkel φ bestimmt werden.
Gegeben ist die komplexe Zahl $z = (1.1 - 0.2i)$. Um den Vektor r zu bestimmen wird die komplexe Zahl in folgende Formel eingesetzt:

$$r = \sqrt{(1.1^2 - 0.2^2)} \approx 1.118$$

Danach wird der Winkel berechnet:

$$\varphi = arctan(\frac{0.2}{1.1}) \approx -0.180$$

Zuletzt wird die Zahl in Polarform dargestellt:

$$1.118(cos(-0.180) + i * sin(-0.180))$$

Um die Summe zu bestimmen, wurden in Python die Teilergebnisse mit Hilfe einer for-Schleife aufsummiert. Zur Überprüfung des Ergebnisses wird die Summe ein weiteres Mal über die Formel der geometrischen Reihe errechnet, mit k gleich 30:

$$S_{\mathrm{k}} = \frac{1 - (1.1 - 0.2)^{k+1}}{1 - (1.1 - 0.2)}$$

In Python wurde dafür folgender Code implementiert:

```
# die Werte der Komplexen zahl in Normal Form
normalForm = complex( 1.1, -0.2 )

# Betrag
r = (np.sqrt( normalForm.real ** 2 + normalForm.imag ** 2 ))

# Argument
angle = (np.arctan( normalForm.imag / normalForm.real ))

# PolarForm Real und immaginaerteil der Komplexen zahl
polarReal = (r * math.cos( angle ))
polarImag = (r * math.sin( angle ) )
polar = complex(polarReal, polarImag)
```

```
# Die Summe der Komplexen Zahl in Polar Form hoch "k"
# (k in range von 0 bis 30)
sum = 0
for k in range(0, 31):
    sum += polar ** k
print(sum)

# Summe Geometrischer Reihe:
check = ((1 - complex(1.1, -0.2)**31) / (1 - complex(1.1,-0.2)))
print(check)
```

2 Aufgabe: Diskrete Fourier Transformation

2.1 Aufgabenteil a

In dieser Aufgabe sollen für folgende Funktion die Fourier-Koeffizienten berechnet werden:

$$y_n = \sin \frac{\pi * n}{N} * \sin \frac{20\pi * n}{N}$$

Dabei ist N die Anzahl der Messpunkte (in diesem Fall N=1000).

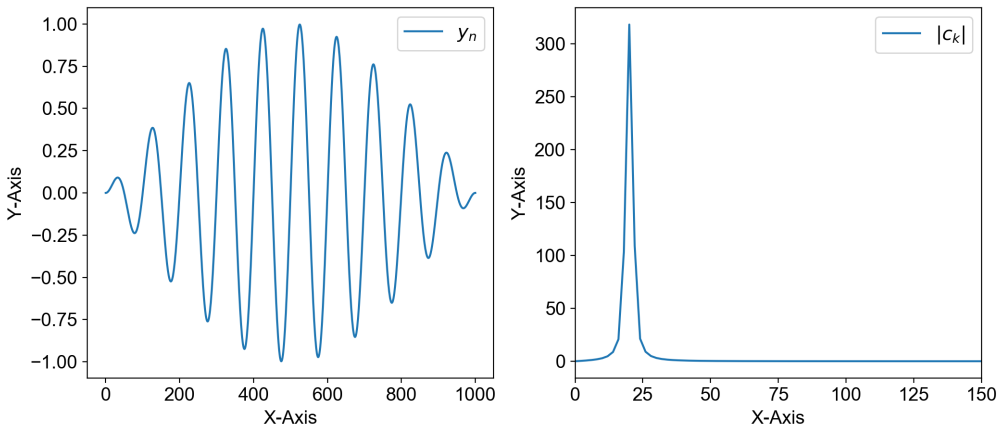

Figure 1: Links: y_k, Rechts: $|c_k|$

Wie in Darstellung 1 zu sehen ist, hat die Funktion der Koeffizienten einen Hochpunkt bei ca. 20. Im folgenden der Quellcode, die Bibliotheksmethode rfft() führt die Fourier-Transformation durch:

```
fig, ax = plt.subplots(1,2,figsize = (12,5))

n = np.linspace(0,1000,1000)
nf = np.linspace(0,1000,501)
N = 1000

yn = np.sin(np.pi * n/N) * np.sin(20 * np.pi* n/N)

fourier = rfft(yn)

plt.xlim(0,150)
ax[0].plot(n,yn, label="$y_n$")
ax[1].plot(nf,abs(fourier), label = "$|c_k|$")

for axis in ax:
    axis.set_xlabel('X-Axis')
    axis.set_ylabel('Y-Axis')
    axis.legend()
```

2.2 Aufgabenteil b

In dieser Aufgabe sollte nun die Fourier-Transformation auf zwei aufgenomene Töne angewendet werden.

2.2.1 Teil I

Die Töne der Trompete bzw. des Pianos lagen jeweils als Datenpunkte (N = 100 000) vor. Sie wurden eingelesen und anschließend die Fourier-Transformation durchgeführt. Doch zuerst werden die Datenpunkte vor der Fourier-Transformation geplottet:

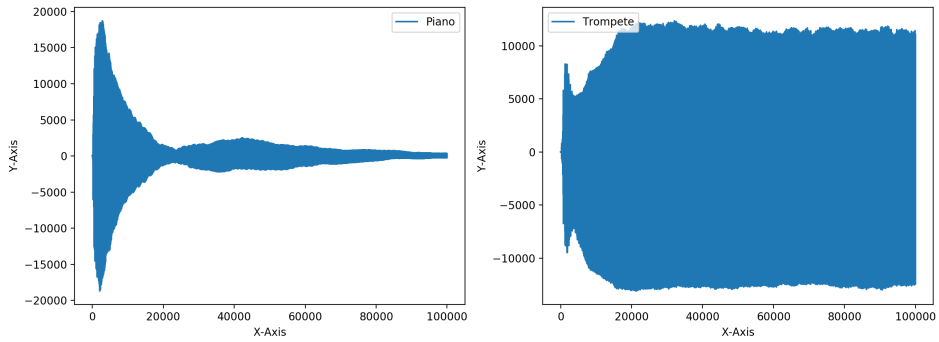

Figure 2: Aufnahme eines Pianos und einer Trompete

Nun wird die Fourier Transformation durchgeführt um mehr über die enthaltenden Töne aussagen zu können. Dazu wird die Methode rfft aus der numpy Bibliothek verwendet:

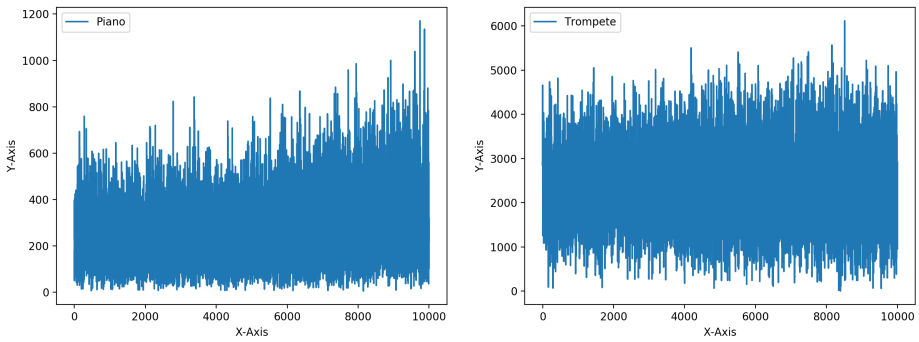

Figure 3: Erste 10 000 Koeffizienten von Piano und Trompete

Betrachtet man die Skaliereung der y-Achsen, fällt auf, dass die Trompete deutlich stärkere Ausschläge hat (Hochpunkt ca. 6000). Daraus lässt sich schließen das die Aufnahme der Trompete lauter war, als die des Pianos.

2.2.2 Teil II

Beide Instrumente spielen den selben Ton. Dieser soll nun herausgefunden werden. Dazu plotten wir alle Koeffizienten und schauen ob beide Instrumente bei einer Frequenz denselben Ausschlag haben.

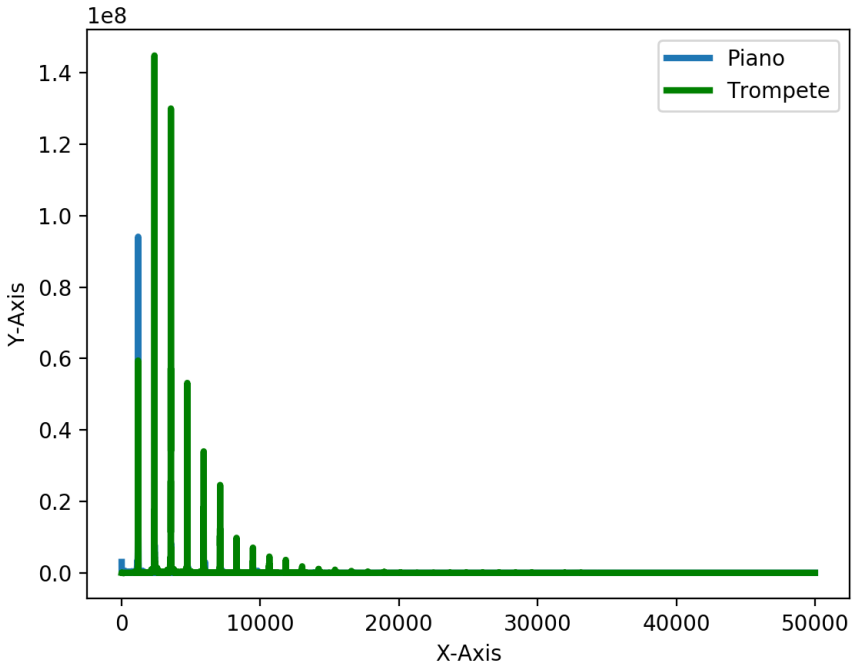

Figure 4: Koeffizienten von Trompete und Piano

Tatsächlich haben beide Töne einen Ausschlag bei einer Frequenz von 1191 Hz. Doch es kann an dieser Stelle noch nicht auf den Ton geschlossen werden. Die Aufnahmmerate

beider Instrumente betrug 44100 pro Sekunde. Da die Anzahl an Datenpunkten N=100000 ist, lässt sich die Zeitdauer der Aufnahme berechnen: $t = \frac{100000}{44100} = 2,268s$. Teilt man die Frequenz von 1191 Hz nun durch die aufgenommene Zeit ergibt sich eine Frequenz von 525,13 Hz. Laut Tontabelle entspricht das einem c". Die anderen Ausschläge der Trompete stammen von mitschwingenden Obertönen und können an dieser Stelle ignoriert werden.
Im folgenden der Quellcode:

```
fig, ax = plt.subplots(2,3,figsize = (16,5))

t = np.linspace(0,100000,100000)
n = np.linspace(0,10000,10000)
u = np.linspace(0,50001,50001)

piano = np.loadtxt("piano.txt",float)
trumpet = np.loadtxt("trumpet.txt",float)

piano_dft = rfft(piano)
trumpet_dft = rfft(trumpet)

piano_second = piano_dft
trumpet_second = trumpet_dft
piano_dft = piano_dft[-10000:]
trumpet_dft = trumpet_dft[-10000:]


print("Gleicher Ausschlag bei (Maximum Piano): ",
            (list(piano_second).index(max(piano_second))))

print("Frequenz des Tones: ",
list(piano_second).index(max(piano_second)) /
            (len(piano) / 44100))


ax[0][0].plot(t,piano)
ax[0][1].plot(t,trumpet)
ax[0][2].plot(n,abs(piano_dft))
ax[1][0].plot(n,abs(trumpet_dft))
ax[1][1].plot(u,abs(piano_second))
ax[1][2].plot(u,abs(trumpet_second))

# Ton c'' mit 523,25
ax[1][1].set_xlim(0,10000)
ax[1][2].set_xlim(0,10000)
```

2.3 Aufgabenteil c

In dieser Aufgabe sind die Daten des Dow Jones zwischen 2006 und 2010 gegeben. Liest man die Daten ein und plottet sie ergibt sich folgendes Bild:

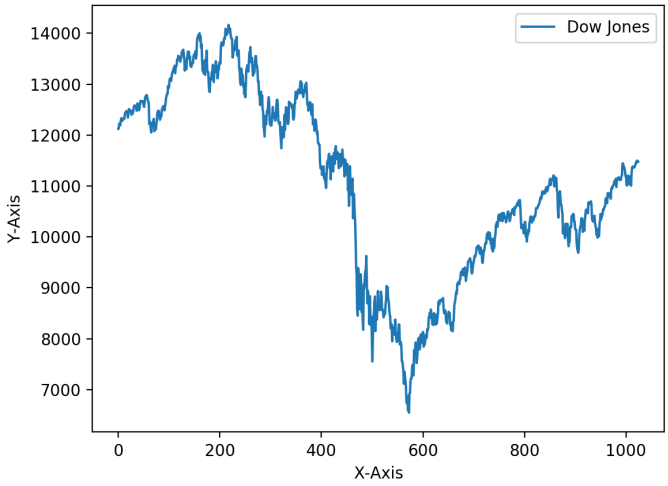

Figure 5: Ungefilterter Dow Jones

Die Kurve ist sehr verrauscht, jeder kleinste Ausschlag des Börsenkurses ist zu sehen. Um dies zu ändern wird die Fourier Transformation durchgeführt und die Daten gefiltert. Die ersten 2 Prozent der Koeffizienten werden behalten, der Rest wird auf null gesetzt. Nun wird die Inverse Fourier Transformation durchgeführt und das Ergebnis geplottet.

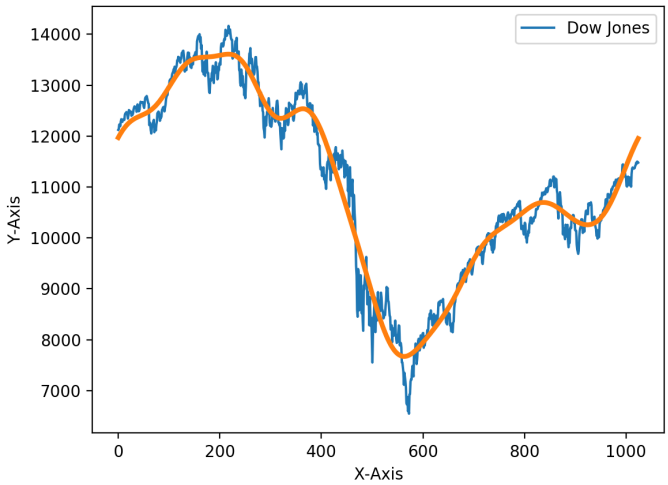

Figure 6: Dow Jones und bereinigte Kurve

Das Ergebnis ist beeindruckend: die orangene Kurve ist eine Trendkurve, durch das Eliminieren der hohen Frequenzen wurde das Rauschen beseitigt. Die orangene Kurve ist bereinigt und zeigt anschaulicher den Kursverlauf.

Im folgenden der Quellcode, die Methode irfft führt die Inverse Fourier Transformation durch, also das berechnen konkreter Datenpunkte aus den Koeffizienten der Fouriertransformation:

```
fig, ax = plt.subplots()

dow_unfiltered = np.loadtxt("dow.txt")

def filterData(data):
    tp = int(0.02 * len(data))
    for i in range(tp,len(data)):
        data[i] = 0
    return data

n = np.linspace(0,1024,1024)

dow_dft = rfft(dow_unfiltered)
dow_filtered = filterData(dow_dft)
dow_idft = irfft(dow_filtered)

ax.plot(n,dow_unfiltered)
ax.plot(n,dow_idft, lw =3)
```

3 Aufgabe: Die Klasse Komplex

Gegeben ist die Klasse "Komplex.py". Deren Methoden erhalten den realen und imaginären Teil einer komplexen Zahl und können damit diverse Rechenoperationen ausführen.

3.1 Aufgabenteil a

In dieser Aufgabe soll die Klasse erweitert werden. Hierfür sollen Methoden für Multiplikation und Division implementiert werden.
Gegeben sind zwei komplexe Zahlen:

$$z_1 = x_1 + iy_1$$

$$z_2 = x_2 + iy_2$$

Für die Multiplikation der Zahlen gilt folgende Formel:

$$z_1 * z_2 = (x_1 + iy_1)(x_2 + iy_2)$$

Dabei ist es wichtig zu beachten, dass $i^2 = -1$ gilt.
Die Methode wird wie folgt im Python implementiert:

```
def __mul__(self, other):
    re = (self._re * other._re) - (self._im * other._im)
    im = (self._im * other._re) + (self._re * other._im)

    return Komplex(re, im)
```

Die Division zweier komplexer Zahlen ist umfangreicher. Ausgangspunkt sind wieder z_1 und z_2:

$$z_1 = x_1 + iy_1$$

$$z_2 = x_2 + iy_2$$

Die Division erfolgt durch folgende Formel:

$$\frac{z_1}{z_2} = \frac{x_1 + iy_1}{x_2 + iy_2} * \frac{x_2 - iy_2}{x_2 - iy_2}$$

Die Methode wird wie folgt im Python implementiert:

```
def __div__(self, other):
    nominator_real = float((self._re * other._re) -
    (self._im * (-other._im)))
    nominator_imag = float((self._im * other._re) +
    (self._re * (-other._im)))

    denominator = float((other._re * other._re) - (other._im * (-other._im))
    + (other._im * other._re) + (other._re * (-other._im)))

    re = nominator_real / denominator
    im = nominator_imag / denominator
    return Komplex(re, im)
```

3.2 Aufgabenteil b

Um die Ergebnisse richtig ausgeben zu können, wurde die String Methode folgendermaßen angepasst:

```
def __str__(self):
    if self._im < 0:
        return str(self._re) + '+' + '(' + str(self._im) + ' i' + ')'
    else:
        return str(self._re) + ' + ' + str(self._im) + ' i'
```

3.3 Aufgabenteil c

Im letzten Teil der Aufgabe wurde der Klasse eine weitere Methode hinzugefügt. Diese gibt eine komplexe Zahl in exponentieller Form aus.
Gegeben ist die komplexe Zahl: $z = x + iy$. Um die Zahl in seiner exponentiellen Form darstellen zu können, wird wie in Aufgabe 1 die Länge des Vektors und der zugehörige Winkel bestimmt:

$$r = \sqrt{(x^2 + y^2)}$$

und

$$\varphi = arctan(\frac{y}{x})$$

Die Exponentialform folgt durch Einsetzen in die Formel:

$$z = re^{i\varphi}$$

In Python implementiert:

```
def __EulerFormel__(self):
    r = math.sqrt( (self._re) ** 2 + (self._im) ** 2 )
    angle = math.atan( self._im / self._re )

    return str (r.__round__(3)) + "exp(" + str(angle.__round__(3))
                                + ')' + 'i'
```