Jan Niclas Ruppenthal
1481198

Heterogenous Computing – Übung 01

Aufgabe 01

Gewählte Implementierungen

Für die erste Implementierung habe ich mich für eine Python-Implementierung der Fourieranalyse entschieden. Die Analyse wurde im Skript **main.py** im Ordner *PythonFourier* implementiert. Das Skript besteht aus drei Bestandteilen:

```
def analyze_wav_file(file_path):
    sample_rate, data = wavfile.read(file_path)

    '''
    Falls zwei Kanaele (Stereo) in der WAV-Datei existieren,
    dann schneiden wir einen Kanal ab, um Rechen- sowie Laufzeit zu sparen.
    '''
    if len(data.shape) > 1:
        data = data[:, 0]

    return data, sample_rate
```

Im ersten Bestandteil wird die übergebene WAV-Datei gelesen. Dabei wird auch die Hälfte der Daten ignoriert, wenn die Audio-Datei aus zwei Kanälen (Stereo) besteht. Dadurch wird Laufzeit sowie Speicherplatz gespart.

```
def analyze(data, block_size, fourier_function):
    num_samples = len(data)
    num_blocks = num_samples - block_size + 1

    aggregated_fft = np.zeros(block_size//2) #Redundanz der Spiegelung
entfernen

    # Fuer jeden Datenblock wird die jeweilige ausgewaehlte Funktion
angewendet.
    for i in range(num_blocks):
        block = data[i:i+block_size]
        fft_result = fourier_function(block)

        # Summiere alle Ergebnisse auf
        aggregated_fft += np.abs(fft_result[:block_size//2])

    # Wir berechnen den Mittelwert, da die Summen sonst zu groß sind
    aggregated_fft /= num_blocks

    return aggregated_fft
```

Hier im Ausschnitt des Quellcodes werden die Daten der WAV-Datei Block für Block analysiert. Dabei werden die Ergebnisse in **aggregated_fft** summiert und nach der Schleife wird der Mittelwert der Ergebnisse bestimmt. Die Methode wird eine DFT- oder FFT-Funktion übergeben. Hierbei kann der Nutzer über Parameter beim Starten des Skripts selbst entscheiden, welche Methode er verwenden möchte.

Der letzte Teil des Programms schreibt die Daten wie **sample_rate**, **block_size** sowie **aggregated_fft** in verschiedenen Dateien, damit die Ergebnisse in **plotting.py** analysiert werden können.

Es wurden vier verschiedene Funktionen implementiert und im folgendem möchte ich alle Implementierungen vorstellen und ihre Laufzeit (ohne Analyse des Speicherbedarfs) vergleichen (n = 1024):

1. **Verwendung von numpy (np.fft.fft)**

   Zum Testen des Skripts habe ich hauptsächlich die **np.fft.fft** Methode verwendet, da sie sehr schnell ist. Das komplette Skript benötigt für die nicht_zu_laut_abspielen.wav ungefähr 1 Minute und 2,96 Sekunden.

2. **Iterative FFT**

   Diese Implementierung habe ich aus einer Webseite[1], die verschiedene Fouriertransformationen miteinander vergleicht. Die iterative Implementierung des Autors ist nach seinen Angaben nur 10-mal langsamer als die **np.fft.fft** Funktion. Für die Datei nicht_zu_laut_abspielen_kurz.wav (Länge: 15 Sekunden) benötigt das Skript ungefähr 1 Minute und 59.02 Sekunden.

3. **Rekursive FFT**

   Diese Implementierung habe ich von ChatGPT generieren lassen. Für die Datei nicht_zu_laut_abspielen_sehr_kurz.wav (Länge: 1 Sekunde) benötigt die komplette Analyse ungefähr 4 Minuten und 43,85 Sekunden.

4. **DFT**

   Auch diese Funktion wurde von der KI generiert. Die DFT-Implementierung ist sehr langsam, da sie für die WAV-Datei nicht_zu_laut_abspielen_sehr_kurz.wav (Länge: 1 Sekunde) länger als 10 Minuten benötigt. Nach 10 Minuten habe ich das Programm abgebrochen.

Ergebnisse

Die Ergebnisse der Fourieranalyse der Datei nicht_zu_laut_abspielen.wav können mithilfe der Datei **plotting.py** analysiert werde. In diesem Skript werden drei verschiedene Diagramme erstellt:

1. Ein Spektrogramm.
2. Ein Diagramm mit den Hauptfrequenzen sowie ihre Amplituden.
3. Beide Daten werden zu einem Diagramm zusammengefasst. Dabei werden sie bis zur letzten gefundenen Hauptfrequenz verkleinert, damit das Diagramm nur die relevanten Informationen enthält.

[1] https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/

Jan Niclas Ruppenthal
1481198

Um die Hauptfrequenzen zu finden, habe ich die lokalen Maxima mithilfe von **find_peaks** von der Bibliothek **scipy.signal** in **aggregated_fft** gesucht und danach habe ich die dazugehörigen Frequenzen bestimmt:

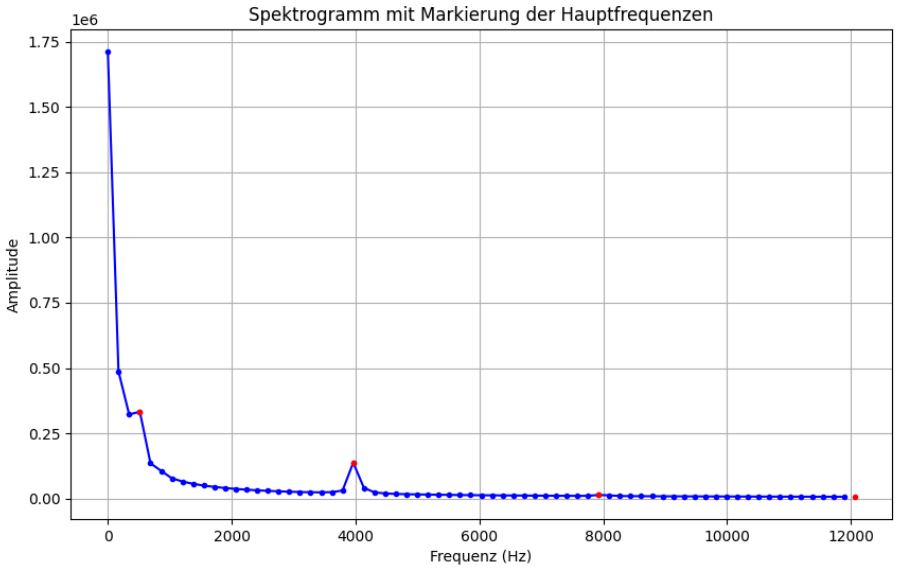
```
def get_main_frequencies_with_amplitude(aggregated_fft, sample_rate,
block_size):
    peaks, _ = find_peaks(aggregated_fft)

    main_frequencies = []
    for peakIndex in peaks:
        main_frequencies.append(peakIndex * sample_rate / block_size)

    return main_frequencies, peaks
```

Im Folgenden gebe ich die zusammengefassten Diagramme für verschiedene Blockgrößen an:

Diagramme mit n = 256 (Laufzeit: 26,01 Sekunden):

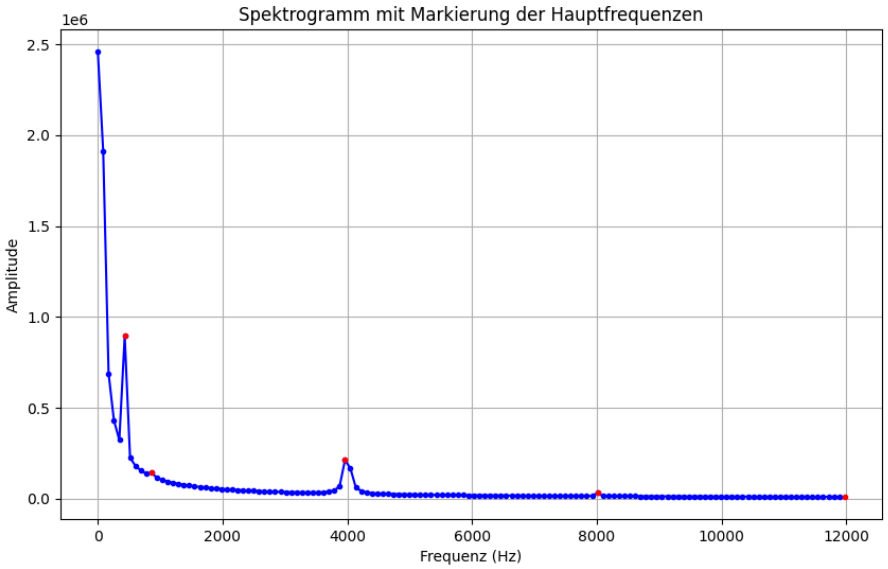


	Hauptfrequenz	Amplitude
0	516.796875	333175.05291205505
1	3962.109375	137711.3895842521
2	7924.21875	14272.137659725775
3	12058.59375	7244.413181194034

Problem: Das Maximum wird von **find_peaks** nicht gefunden.

Jan Niclas Ruppenthal
1481198

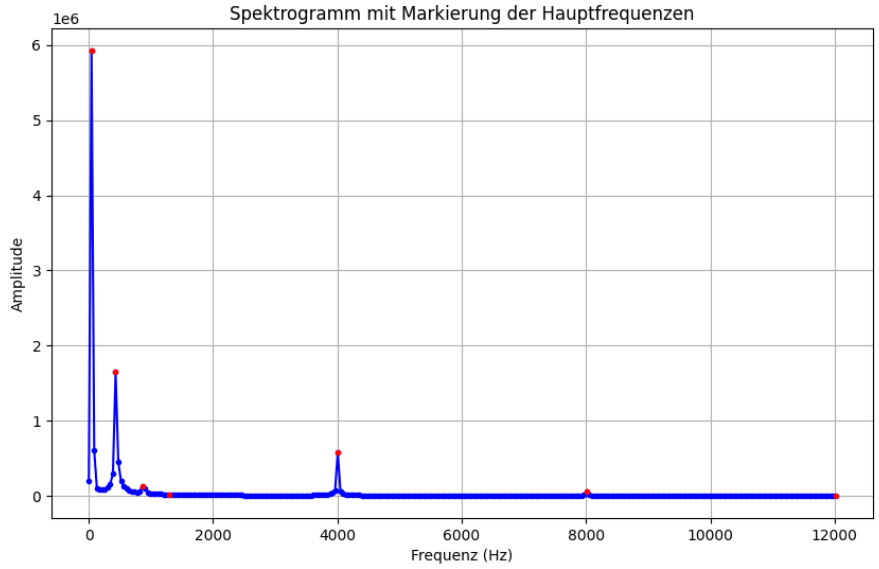
Diagramm mit n = 512 (Laufzeit: 33,01 Sekunden):



	Hauptfrequenz	Amplitude
0	430.6640625	894103.3127653046
1	861.328125	144166.81759392956
2	3962.109375	211856.65926018986
3	8010.3515625	31070.373745312892
4	11972.4609375	10190.26021941069

Problem: Das Maximum wird von **find_peaks** nicht gefunden.

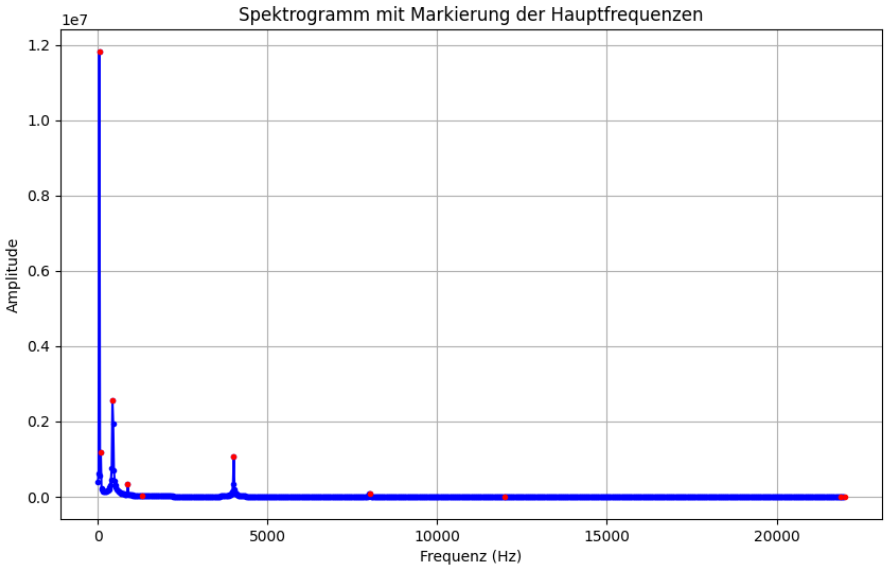
Diagramm mit n = 1024 (Laufzeit: 1 Minute und 1,63 Sekunden):



Jan Niclas Ruppenthal
1481198

	Hauptfrequenz	Amplitude
0	43.06640625	5926643.14962656
1	430.6640625	1645417.3411351012
2	861.328125	131049.86352009393
3	1291.9921875	22309.71339958239
4	4005.17578125	579259.092924025
5	8010.3515625	53893.82868761788
6	12015.52734375	5031.239973424879

Diagramm mit n = 2048 (Laufzeit: 1 Minute und 48,60 Sekunden)



	Hauptfrequenz	Amplitude
0	43.06640625	11817352.386685152
1	86.1328125	1189682.7379957668
2	430.6640625	2556781.7773357825
3	882.861328125	348324.8519683373
4	1313.525390625	39594.5089813851
5	4005.17578125	1076912.1577983745
6	8010.3515625	78507.41775864325
7	11993.994140625	10745.343528356061
8	21899.267578125	2364.5411533128213
9	22006.93359375	2364.397336842568

Problem: Die letzten beiden Frequenzen sind zu nah beieinander.

Analyse der Frequenzen

Wenn man sich die Ergebnisse für n = 1024 betrachtet, dann fällt auf, dass die WAV-Datei aus drei Grundfrequenzen besteht. Dabei haben die letzten beiden Grundfrequenzen jeweils zwei Harmonische:

1. **43.06640625 Hz**
2. **430.6640625 Hz**
    a. 861.328125 Hz (1. Harmonische)
    b. 1291.9921875 Hz (2. Harmonische)
3. **4005.17578125 Hz**
    a. 8010.3515625 Hz (1. Harmonische)
    b. 12015.52734375 Hz (2. Harmonische)

Wenn man zusätzlich noch die Amplituden dieser Frequenzen betrachtet, dann fällt auf, dass die Amplituden der Harmonischen mindestens um einen Faktor von 10 verringert werden.

Aufgabe 02

Um den Speicherbedarf des Python-Skripts **main.py** zu analysieren, habe ich die Bibliothek *tracemalloc*[2] verwendet. Für die Analyse mit *tracemalloc* habe ich den Quellcode so verändert, dass vor und nach der Fourieranalyse den momentanen und höchsten (**current** und **peak**) Speicherbedarf in Bytes gelesen wird. Innerhalb der Methode **analyze** lese ich nun die beiden Werte in jedem Schritt und speichere sie in einer Liste. Nachdem alle Blöcke der WAV-Datei analysiert wurden, schreibe ich alle Werte zum Speicher in die Dateien **currentMB.txt** und **peakMB.txt**. Mithilfe dieser Dateien kann man zwei Diagramme (siehe **plotting_memory.py**) erstellen, die den Speicherbedarf des Programms darstellen soll.

Modifizierter Programmcode (ohne Kommentare und Ausgabe):

```
current, peak = tracemalloc.get_traced_memory()
aggregated_fft = analyze(wav_data, block_size, fourier_function)

current, peak = tracemalloc.get_traced_memory()
tracemalloc.stop()
```

In der Hauptmethode (**main** in **main.py**, Zeile 174 - 180) startet das Programm die Analyse des Speicherbedarfs. Bevor die Fourieranalyse gestartet wird, gibt das Programm den momentanen und höchsten Speicherbedarf aus. Nach dieser Analyse der Datei wird nochmal der Speicherbedarf ausgegeben und die Speicheranalyse über *tracemalloc* wird gestoppt.

[2] https://docs.python.org/3/library/tracemalloc.html

```
current, peak = tracemalloc.get_traced_memory()
currentDataMB = [current/10**6]
peakDataMB = [peak/10**6]

for i in range(num_blocks):
    tracemalloc.reset_peak()
    block = data[i:i+block_size]
    fft_result = fourier_function(block)
    aggregated_fft += np.abs(fft_result[:block_size//2])

    current, peak = tracemalloc.get_traced_memory()
    currentDataMB.append(current/10**6)
    peakDataMB.append(peak/10**6)


# Wir berechnen den Mittelwert, da die Summen sonst zu groß sind
aggregated_fft /= num_blocks

current, peak = tracemalloc.get_traced_memory()
currentDataMB.append(current/10**6)
peakDataMB.append(peak/10**6)
write_data_to_file(currentDataMB, "currentMB.txt")
write_data_to_file(peakDataMB, "peakMB.txt")
```

In der Methode **analyze** in **main.py** (Zeile 113 - 142) wird der Speicherbedarf gelesen und in zwei separaten Listen gespeichert. Zudem wird in jeder Iteration der Schleife den höchsten Stand (**peak**) zurückgesetzt und erneut wird der Bedarf in die beiden Listen hinzugefügt. Nach der Schleife wird erneut der Speicher gelesen und gespeichert. Danach werden alle Werte aus den Listen in zwei Dateien geschrieben.

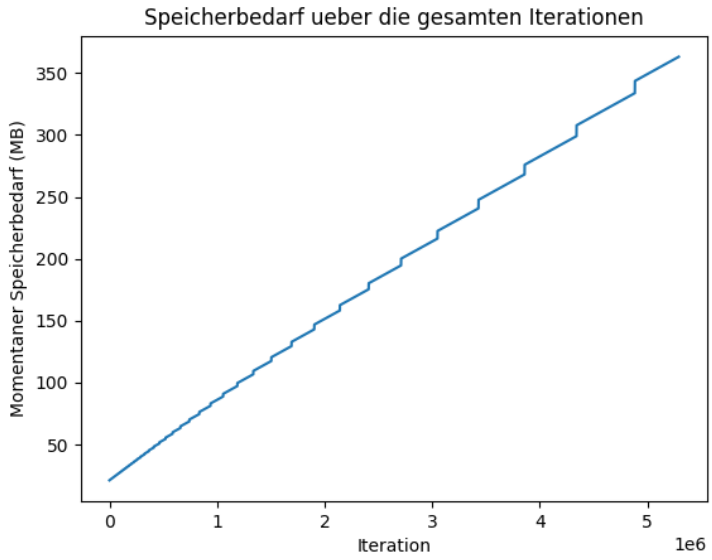
Die Speicheranalyse für die zweite Aufgabe wurde auf dem folgenden Computer ausgeführt:

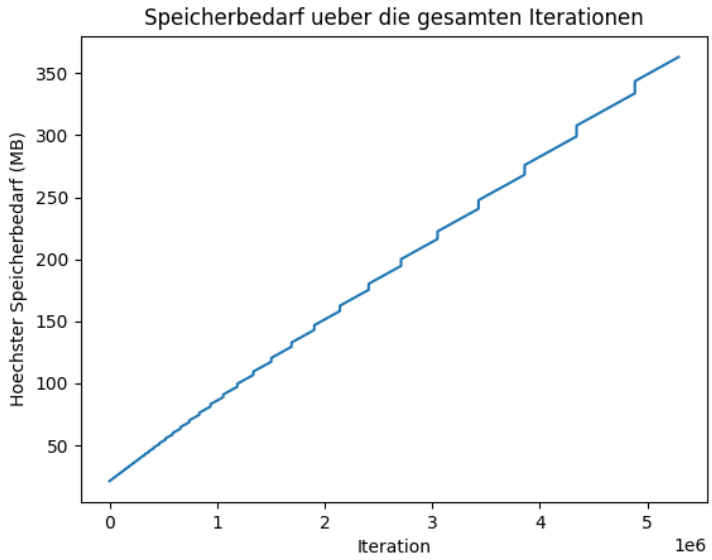
- Prozessor: Intel(R) Core(TM) Ultra 7 155H 1.40 GHz
- Installierter RAM: 32,0 GB
- Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor
- Edition: Windows 11 Home
- Version: 23H2
- Betriebssystembuild: 22631.3593

Des Weiteren habe ich nicht den Speicherbedarf des restlichen Programms analysiert, da wir nur an der Fourieranalyse interessiert sind.

Wenn man nun das Skript mit der Datei nicht_zu_laut_abspielen.wav mit einer Blockgröße von 1024 und mit der FFT-Implementierung von *numpy* laufen lässt, dann werden die folgenden Diagramme erstellt:

Jan Niclas Ruppenthal
1481198





Man sieht, dass der Speicherbedarf der Analyse mit jeder Iteration steigt. Ich hatte jedoch nicht erwartet, dass der verwendete Speicher kontinuierlich zunimmt. Ich ging davon aus, dass der Speicherverbrauch in einem bestimmten Bereich schwanken würde. Ich nahm an, dass Python während der Iterationen auch Speicher freigeben würde. Zuerst habe ich gedacht, dass ich an der falschen Stelle den Speicherverbrauch messe, weshalb ich zu Testzwecken vor dem Aufruf der fourier_function (Zeile 121) den Speicherbedarf auslese. Dennoch erhielt ich ein ähnliches Diagramm.

Jan Niclas Ruppenthal
1481198

Aufgabe 03

Für die dritte Aufgabe des ersten Übungsblatts habe ich die folgenden verschiedenen Varianten zur Analyse des Speicherbedarfs verwendet:

1. Java-Programm auf meinem Windows-Laptop (Spezifikationen wurden oben beschrieben)
2. Python-Programm auf einem Raspberry Pi Model B Revision 2
3. Java-Programm auf einem Raspberry Pi Model B Revision 2
4. Python-Programm auf einem Laptop mit Linux Mint
5. Java-Programm auf einem Laptop mit Linux Mint

Das Java-Programm befindet sich im Ordner *JavaFourier*. Die Hauptklasse **Main** in **Main.java** ist ähnlich aufgebaut wie die Datei **main.py**. Zuerst wird die übergebene WAV-Datei gelesen und danach wird die Fourieranalyse auf eine übergebene Datei angewendet. Man kann auch die Funktion zur Fourieranalyse über Parameter beim Starten der JAR-Datei wie im Python-Skript auswählen.

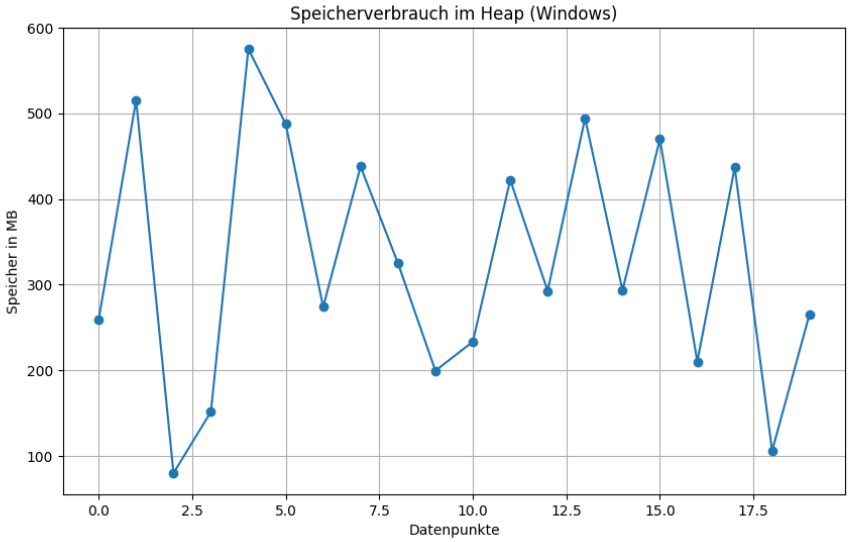
Zur Analyse des Speicherbedarfs des Java-Programms habe ich *JConsole* verwendet, da dies in der OpenJDK enthalten ist[3]. Ich habe *JConsole* verwendet, weil ich dafür kein weiteres Programm auf dem Raspberry Pi installieren muss. In diesem Tool habe ich hauptsächlich den Reiter „Arbeitsspeicher“ mit dem Diagramm „Heap-Nutzung“ betrachtet. Dabei sieht man in diesem Diagramm die Verwendung des Heap in MB.

Für die Analyse des Speichers habe ich die Datei nicht_zu_laut_abspielen.wav und eine Blockgröße von 1024 gewählt (wie auch in Aufgabe 2). Des Weiteren habe ich zur Analyse eine implementierte FFT (org.apache.commons.math3.transform.FastFourierTransformer[4]) verwendet.

Nachdem das Programm beendet wurde, habe ich in der *JConsole* die Daten zur Verwendung des Heap in eine .csv-Datei exportiert, sodass ich mit *matplotlib* ein Diagramm generieren konnte (siehe Ordner *PlottingJavaMemory*).

[3] https://openjdk.org/tools/svc/jconsole/
[4] https://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/org/apache/commons/math3/transform/FastFourierTransformer.html

Java-Programm auf meinem Windows-Laptop



Hier in dem obigen Diagramm ist das Muster, welches ich auch schon für Python erwartet habe. Es wird Speicher belegt, der jedoch vom Garbage Collector irgendwann wieder freigegeben wird. Der höchste Punkt war bei fast 600 MB. Es ist dennoch erstaunlich, dass Java fast doppelt so viel Speicher benötigt als Python (~300 MB). Für dieses Programm konnte ich nicht viele Datenpunkte sammeln, da das Programm nach 1 Minute und 24 Sekunden fertig war.

Die nächsten beiden Speicheranalysen wurden auf einem Raspberry Pi Model B Revision 2 mit den folgenden Spezifikationen[5] durchgeführt:

- Prozessor: BCM2835 (ARM11 Prozessor)
- Installierter RAM: 512 MB
- Betriebssystem: Raspberry Pi OS (32 Bit)
- Kernel-Version[6]: 6.6
- Debian Version[6]: 12 (bookworm)

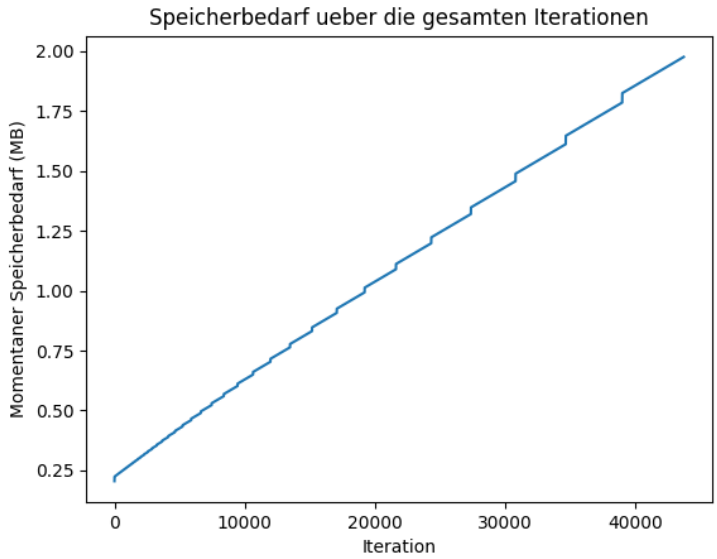
Damit man die Ergebnisse des Speicherbedarfs besser vergleichen kann, habe ich die gleiche Blockgröße und den gleichen FFT-Algorithmus für beide Programme verwendet.

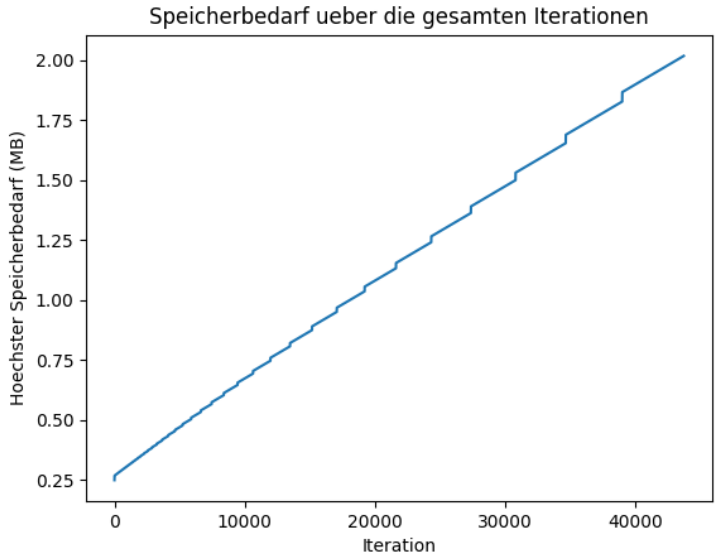
[5] https://www.pololu.com/product/2750
[6] https://www.raspberrypi.com/software/operating-systems/

Jan Niclas Ruppenthal
1481198

Python-Programm auf dem Raspberry Pi Model B Revision 2

Mithilfe der Bibliothek *tracemalloc* und der Datei **plotting_memory.py** wurden die folgenden Diagramme erstellt:

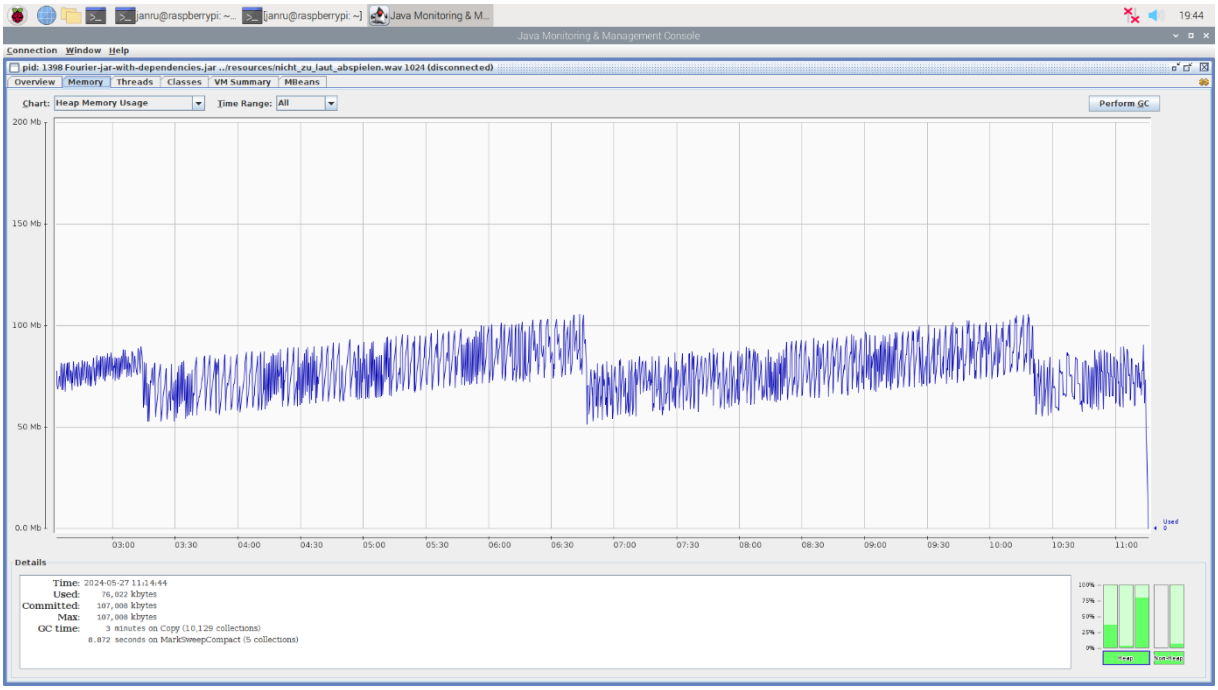




Leider konnte ich nicht die zweiminütige Datei auf dem Raspberry Pi mit dem Python-Skript analysieren lassen, da der Kleincomputer nach 48 Stunden eingefroren ist. Da ich dennoch das Python-Programm auf dem Pi starten wollte, habe ich die Datei nicht_zu_laut_abspielen_kurz.wav (Länge: 15 Sekunden) verwendet. Dies hat auch reibungslos funktioniert.

Die Diagramme sind strukturell sehr ähnlich zu den Diagrammen aus Aufgabe 2. Man erkennt, dass der Speicherbedarf und die Anzahl der Iterationen der for-Schleife geringer waren. Da ich eine kürzere Datei verwendet habe, kann man die beiden Diagramme nicht vergleichen.

Java-Programm auf dem Raspberry Pi Model B Revision 2



In diesem Diagramm zur Heap Nutzung erkennt man auch das schwankende Muster. Dennoch gibt es zwei interessante Aspekte, die man beobachten kann: Es gibt drei Stellen, an denen vom Garbage Collector sehr viel Speicher freigegeben wird. Ich bin mir zwar nicht sicher, aber ich denke, dass eine Full Garbage Collection ausgeführt wird. Dies passiert, wenn die Applikation nicht genug Speicher zur Verfügung hat. Hierbei werden die Objekte der jungen und alten Generation gelöscht[7].

Des Weiteren gehe ich davon aus, dass bei einer Full Garbage Collection der maximale Heap vergrößert wird. Im linken Teil des Diagramms war das Maximum bei ungefähr 80 MB. Aber im zweiten Teil ist das Maximum auf ungefähr 107 MB. Da der Heap wieder das Maximum erreicht hat, wird erneut ein Full Garbage Collection durchgeführt. Dies geschieht so lange, das Programm beendet ist.

Der zweite interessante Aspekt ist, dass das Diagramm zwischen den Full Garbage Collection monoton wachsend ist. Dies weist darauf hin, dass das Programm mehr Speicher benötigt.

Die nächsten Speicheranalysen wurden auf dem folgenden Gerät durchgeführt:

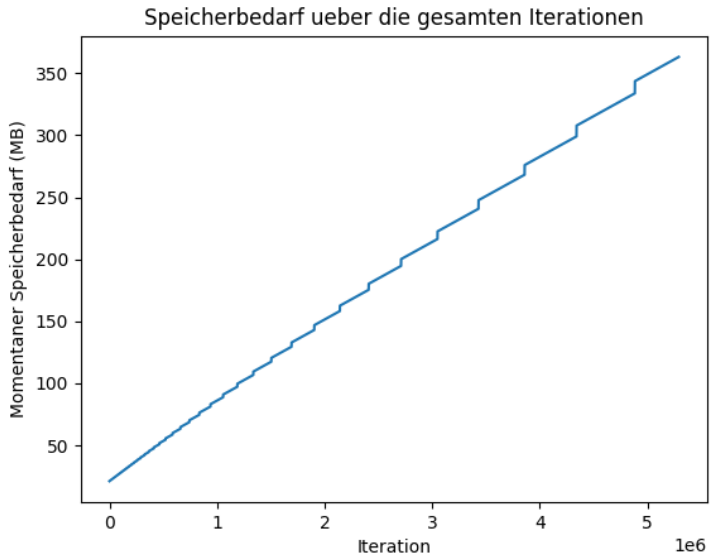
- Prozessor: Intel(R) Core(TM) i5-8265U CP
- Installierter RAM: 8,0 GB
- Betriebssystem: Linux Mint
- Version: 21.1 (Vera)
- Linux Kernel Version: 5.19.0-46-generic

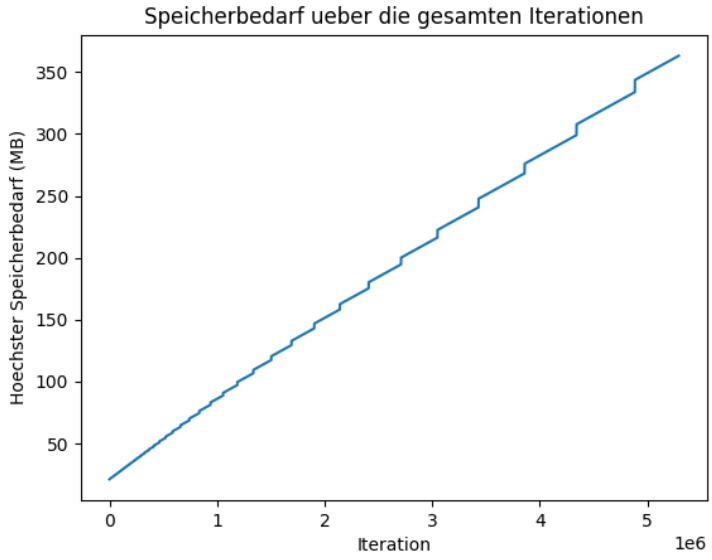
[7] https://backstage.forgerock.com/knowledge/kb/article/a75965340

Jan Niclas Ruppenthal
1481198

Python-Programm auf einem Laptop mit Linux Mint

Mithilfe der Bibliothek *tracemalloc* und der Datei **plotting_memory.py** wurden die folgenden Diagramme erstellt:



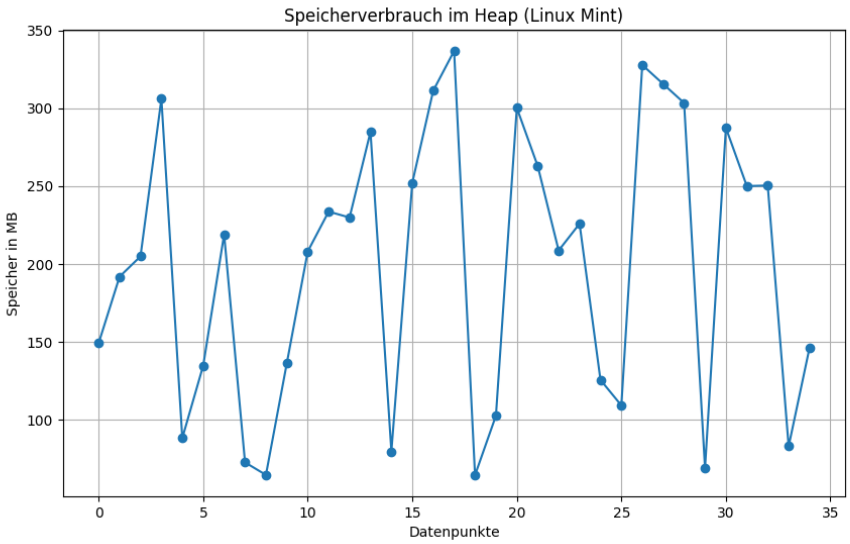


Im Vergleich zu den Diagrammen aus Aufgabe 2, steigt auch in dieser Variante der Speicherbedarf in jeder Iteration. Dabei sehen die Diagramme auch fast identisch aus.

Dabei haben alle Diagramme, die den Speicher des Python-Skripts in veranschaulicht, in allen Varianten die gleiche Struktur: Der Speicherbedarf steigt pro Iteration an.

Jan Niclas Ruppenthal
1481198

Java-Programm auf einem Laptop mit Linux Mint



Wie auch in Windows ergibt sich durch die Garbage Collection mein erwartetes Muster. Jedoch war der höchste Speicherverbrauch fast bei 350 MB im Gegensatz zu fast 600 MB auf dem Windows-Laptop.

Abschließender Vergleich

In allen Diagrammen zum Speicherbedarf des Python-Skripts zur Fourieranalyse erkennt man, dass der verwendete Speicher kontinuierlich wächst. Dabei haben auch alle Diagramme die gleiche Struktur.

Die Diagramme zum Speicher des Java-Programms weisen das gleiche Muster auf. Hierbei zeigt der Raspberry Pi Eigenschaften der Full Garbage Collection und dabei wird der Heap vergrößert.