# Projekt 2: Die diskrete Fourier Transformation

Mokrane Yahiatene

## 1 Einleitung

In diesem Paper möchte ich mich mit der Signal- und Bildanalyse beschäftigen, insbesondere mit der Frequenzanalyse auf Basis der Fourier-Transformation. Wir werden uns hier auf die Optimierung von Bilddaten beschränken, die wir mathematisch als 2 dimensionale Matrix darstellen, welche mit Graustufenwerten befüllt ist. Wie man sehen wird ist diese Transformation sehr nützlich, um diverse Filter anwenden zu können, mit derer man eine Qualitätsverbesserung erreichen möchte. Außerdem werden in den folgenden Abschnitten die Vor- und Nachteile der verschiedenen Filter besprochen, sowie verschiedene Algorithmen und Implementierungen diskreter Fourier Transformationen, um dann deren Laufzeiten zu diskutieren.

## 2 Methode

In diesem Abschnitt werde ich zu aller erst auf die Fourier Transformation im 1-dimensionalen Fall, sowie im 2-dimensionalen Fall eingehen.

`1 dimensionale diskrete Fourier Transformation` :

$$\hat{a}_k = \sum_{n=0}^{N-1} e^{-2\pi i \cdot \frac{nk}{N}} \cdot a_n \ , \ k = 0, ..., M-1$$

Der 1 dimensionale Fall bildet den Vektor $a = (a_0, \ldots, a_{N-1})$ aus dem Ursprungsraum in den Fourierraum ab. Es werden also die Signale auf ein periodisches Frequenzspektrum abgebildet. Hierzu durchläuft er die Summe über die Vektorelemente und bildet diese auf ihre Fourierkoeffizienten ab. Im Frequenzspektrum kann man dann überlagernde Frequenzen die sich zuvor zu einer vermischten, in ihre Einzelnen aufspalten, um diese dann zu bearbeiten.

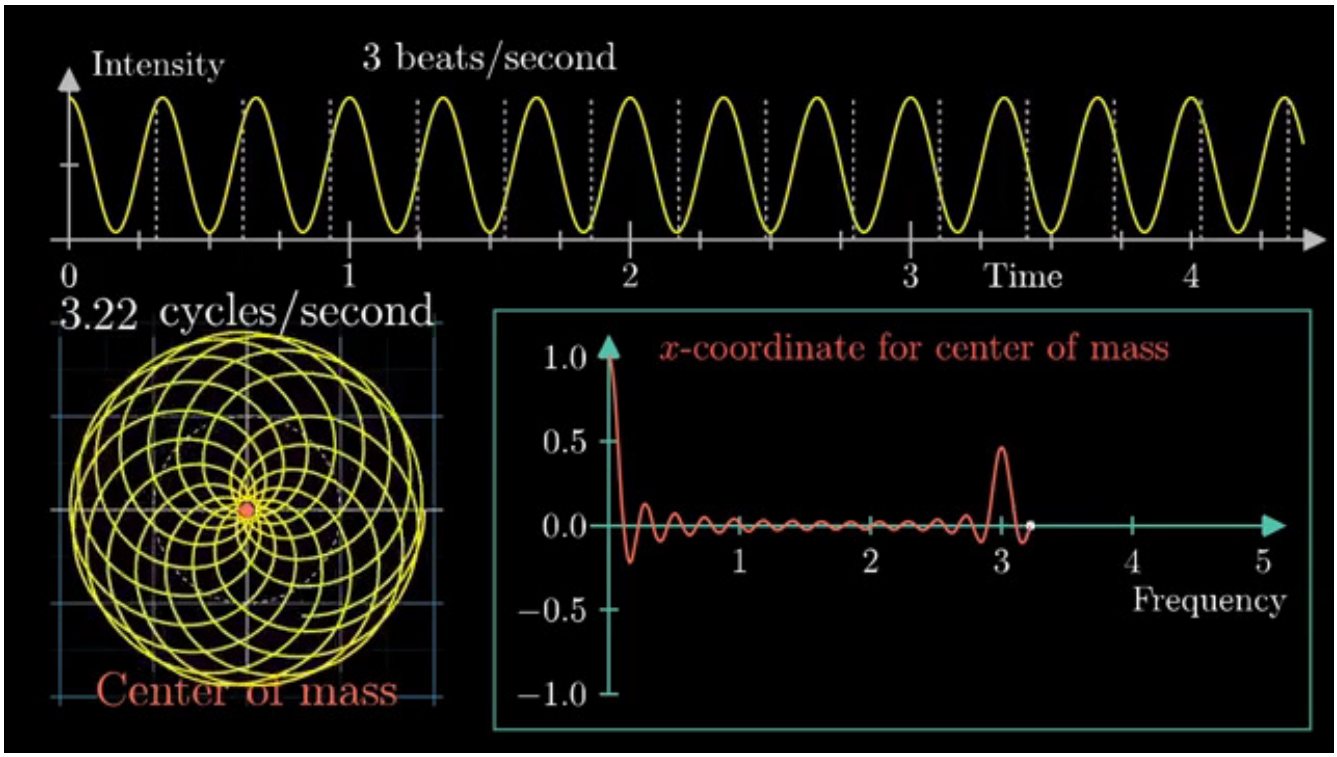



Abbildung 1: Aufwickeln einer Frequenz auf einen Kreis. Das Massenzentrum befindet sich im Ursprung also auf dem Frequenztief.

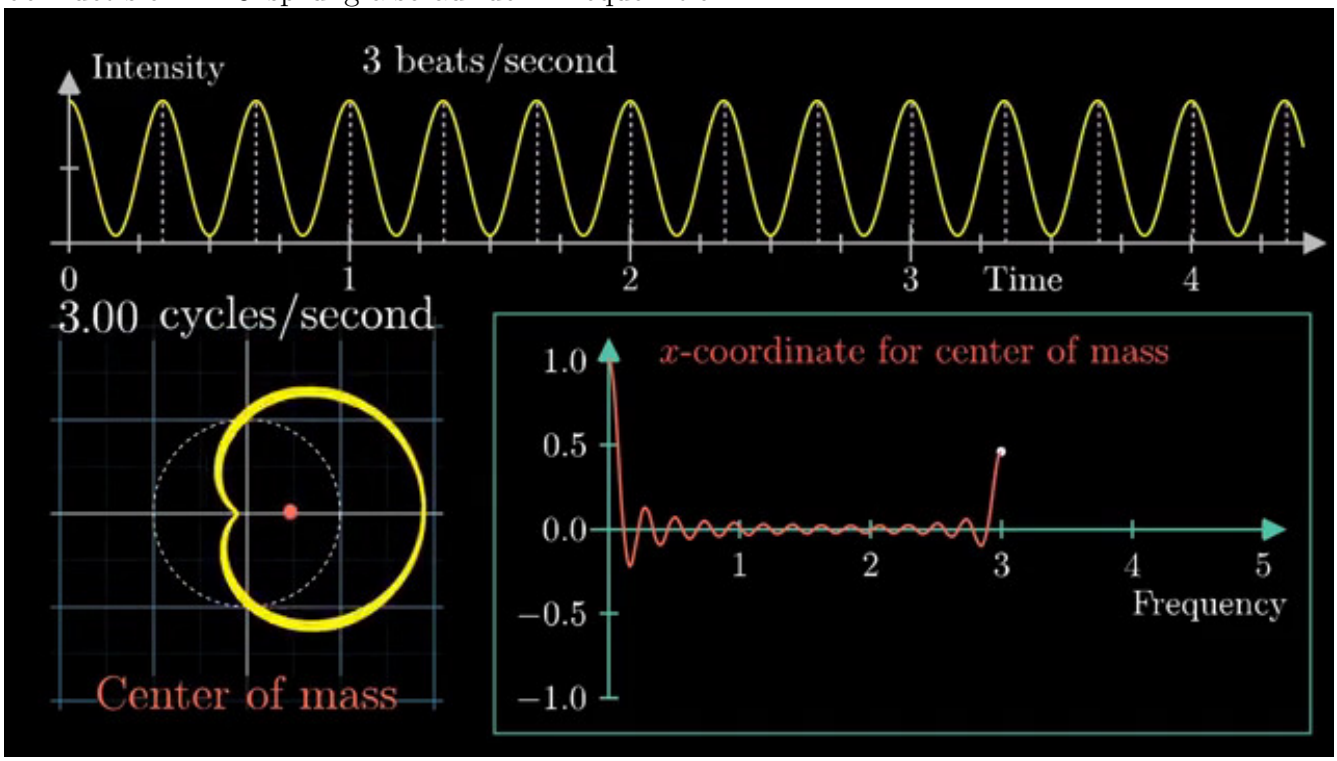



Abbildung 2: Das Massenzentrum befindet sich auf dem Frequenzmaximum bzw. am weitesten vom Ursprung entfernt

Der zwei dimensionale Fall sieht für unsere Bildverarbeitungszwecke wie folgt aus. In diesem Fall bildet man nun die Bildmatrix $a_{m,n}$mit $0 \leq m \leq M-1$ und $0 \leq n \leq N-1$ in den Fourierraum ab. Analog zur obigen Formel bekommt man dann folgende Matrix:

`2 dimensionale diskrete Fourier Transformation` :

$$\hat{a}_{k,l} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} e^{-2\pi i \cdot \frac{mk}{M}} e^{-2\pi i \cdot \frac{nl}{N}} \cdot a_{m,n} \; , k = 0, ..., M-1 \; , l = 0, ..., N-1 \quad (1)$$

Den Ergebnisvektor bzw. Ergebnismatrix nennt man auch Fouriertransformierte. Die inverse Transformation lautet wie folgt.
`2 dimensionale inverse diskrete Fourier Transformation` :

$$\hat{a}_{k,l} = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} e^{2\pi i \cdot \frac{mk}{M}} e^{2\pi i \cdot \frac{nl}{N}} \cdot a_{m,n} \; , k = 0, ..., M-1 \; , l = 0, ..., N-1$$

Man beachte dass, sich die Transformationsformel nur verändert hat und zwar jeweils um den Vorfaktor $\frac{1}{MN}$ und einem Vorzeichenwechsel in der Potenz. Nun stehen uns alle Mittel zur Verfügung um mit der Optimierung mit Hilfe diverser Filter zu beginnen. Das Vorgehen, um die gewünschten Ergebnisse zu erhalten, unterteilt sich in folgende Punkte:

1. Die 2 dimensionale Fourier Transformation auf die Bildmatrix ( Array ) anwenden.
2. Umgruppieren der Werte ( Zentrieren der Nullfrequenz in der Mitte ) bzw. der Frequenzachse .
3. Anwenden der Filter.
4. Zurückshiften der Achse bzw. Zurückgruppierung des Ursprungszustand
5. Anwenden der 2 dimensionalen inversen diskreten Fourier Transformation

Im Folgenden werde ich dem Leser Beispielimplementierungen der oben genannten Verfahren demonstrieren und im nächsten Abschnitte die Resultate erläutern.

Diskrete Fouriertransformation:

```
#Algorithmus der diskreten Fourier Transformation eines 2 dimensionalen Arrays
def ft_own(array):
    arr = np.zeros((M,N),dtype=complex)
    for k in range(0,M,1):
        for l in range(0,N,1):
            for i in range(0,M,1):
                for j in range(0,N,1):
                   arr[k][l] +=array[i][j] * np.exp(-2j*(np.pi*((i*k)/M))) * np.
 ↪exp(-2j*(np.pi*((j*l)/N)))

    return arr
```

Inverse diskrete Fouriertransformation:

```
#Algorithmus der diskreten  inversen Fourier Transformation eines 2␣
 ↪dimensionalen Arrays
def ift_own(array):
    arr = np.zeros((M,N),dtype=complex)
    for k in range(0,M,1):
        for l in range(0,N,1):
            for i in range(0,M,1):
                for j in range(0,N,1):
                   arr[k][l] +=np.divide(1,M*N)*array[i][j] * np.exp(2j*(np.
 ↪pi*((i*k)/M))) * np.exp(2j*(np.pi*((j*l)/N)))

    return arr
```

Wie man erkennen kann sind diese Implementierungen sehr unperformant. Diese Schwäche wird durch den sogenannten Fast Fourier Transformation Algorithmus behoben. In diesem Zuge werden wir uns häufiger der fft2 Methode aus der numpy Library bedienen, die den Fast Fourier Transformation Algorithmus implementiert. Wie genau diese Optimierung erreicht wird und inwiefern die Laufzeit dadurch beeinflusst wird, wird in dem Abschnitt Diskussion näher erläutert.

Das Verschieben der Frequenzachse erriecht man wie folgt in Python: Im folgenden Beispiel bedienen wir uns der numpy roll Methode, die die Spalten bzw. die Zeilen eines Arrays verschiebt:

```
#2 malige Anwenden von numpy roll um Spaltenanzahl/2 und Zeilenanzahl/2 anhand
 ↪der X und Y Achse ermöglicht
#das Verschieben der kleinen Frequenzen(Zeilen/Spalten) in die Mitte des Bildes
plt.imshow(np.abs(np.roll(np.roll(dft_circle,M//2,axis=0),N//
 ↪2,axis=1)),cmap='gray')
```

Um die korrekte Frequenzverschiebung zu erhalten verschieben wir das Array längs der X und Y Achse, um jeweils die Hälfte der Spalten- bzw. Zeilenlänge. Ein analoges Ergebnis erreicht man mit der numpy fftshift Methode, der wir uns auch bedienen werden:

```
plt.imshow(np.abs(np.fft.fftshift(fft_circle)),cmap='gray')
```

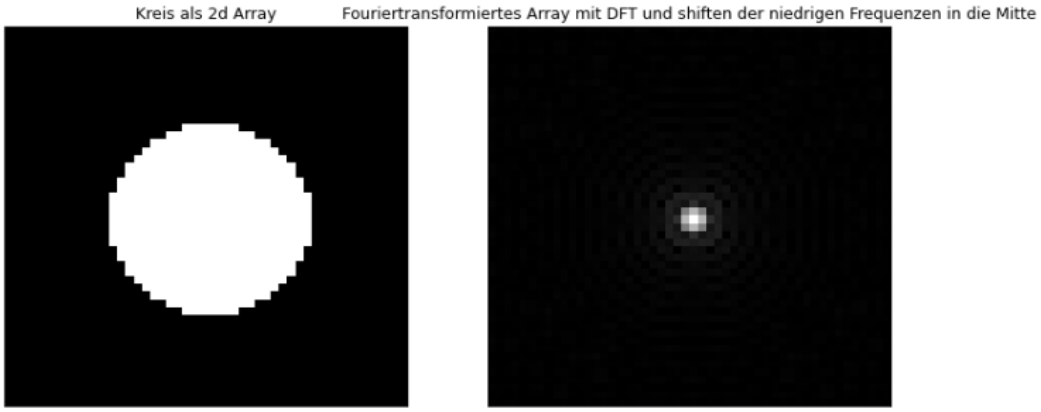
Zu guter Letzt stelle ich noch zwei Filter vor, die sogenannten Tiefpass- bzw. Hochpassfilter.

Tiefpassfilter:
Tiefpassfilter unterdrücken im Ortsfrequenzbereich die hohen Frequenzen. Im gefilterten Bild bedeutet das, dass harte Wechsel der Grauwerte(große Grauwertunterschiede) unterdrückt werden.
Hochpassfilter:
Hochpassfilter unterdrücken im Frequenzbereich die niedrigen Frequenzen, das bedeutet , dass starke Kontraste hervorgehoben werden. Schleichende Wechsel der Grauwerte, also geringer Grauwertunterschied, wird hier unterdrückt(Kantenfilter).
Wir werden noch etwas genauer auf die Filter Unterschiede im Diskussionsabschnitt eingehen.

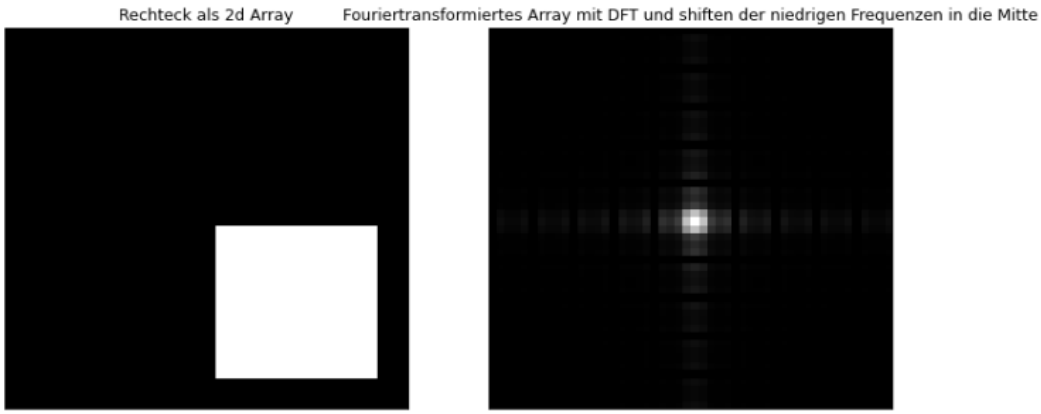
```
def hochpass(img):
    hochpass_arr = np.zeros(img.shape)
    radius = 35  #Radius der Kreisblende
    #generiere Lochblende:
    ax, ay = img.shape
    for i in np.arange(len(img)):
        for j in np.arange(len(img[0])):
            if (i-ax/2)**2 + (j-ay/2)**2 > radius**2:
                hochpass_arr[i][j]=1
            else:
                hochpass_arr[i][j]=0
    return hochpass_arr
def tiefpass(img):
    tiefpass_arr = np.zeros(img.shape)
    radius = 35  #Radius der Kreisblende
    #generiere Lochblende:
    ax, ay = img.shape
    for i in np.arange(len(img)):
        for j in np.arange(len(img[0])):
            if (i-ax/2)**2 + (j-ay/2)**2 < radius**2:
                tiefpass_arr[i][j]=1
            else:
                tiefpass_arr[i][j]=0
    return tiefpass_arr
```

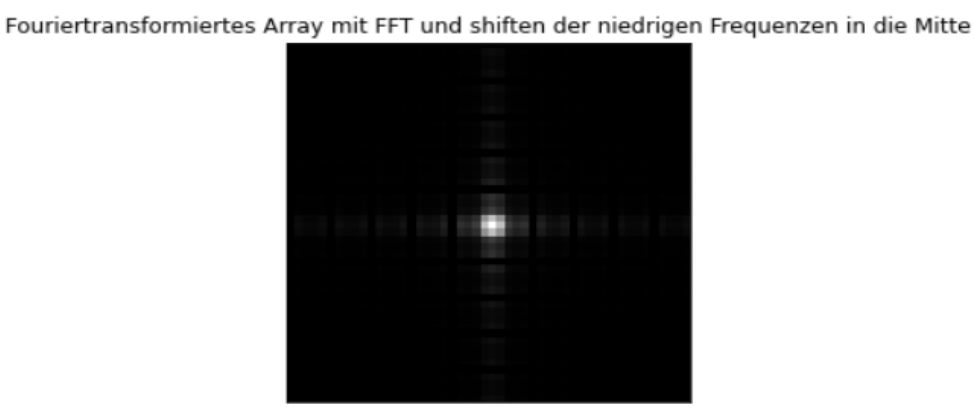
# 3 Resultate

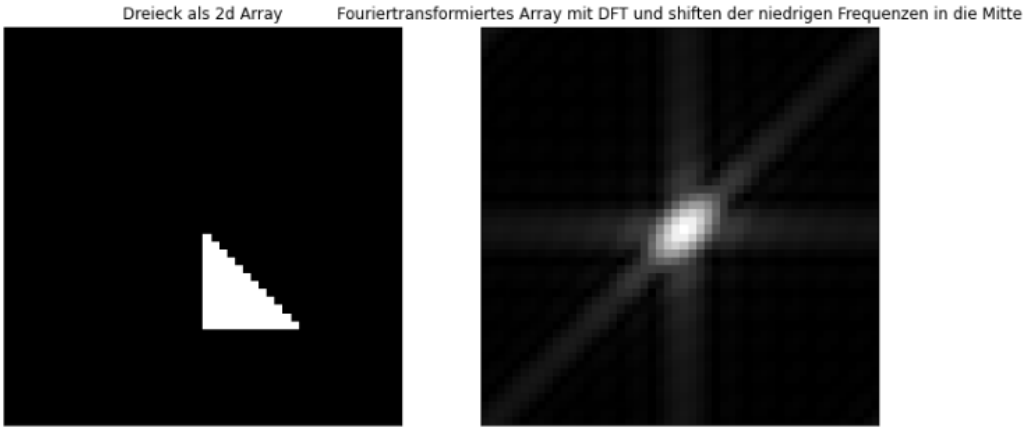
Abbildung 3: Gegenüberstellung des Originalbilds verschiedener Formen (Kreis, Quadrat, Dreieck, Linie) und der Fouriertransformierten(zum Vergleich mit DFT und FFT)

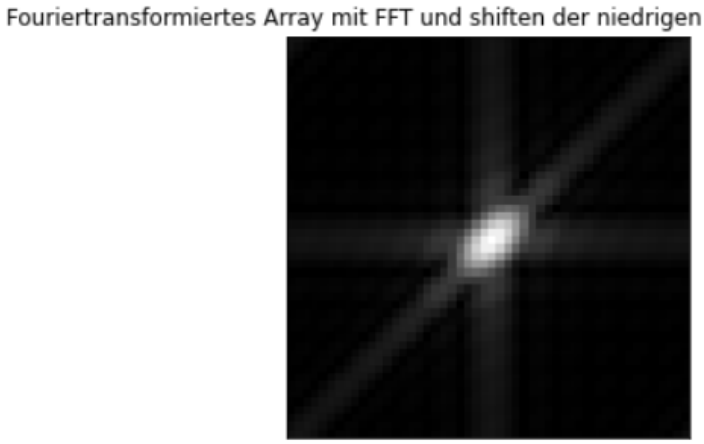


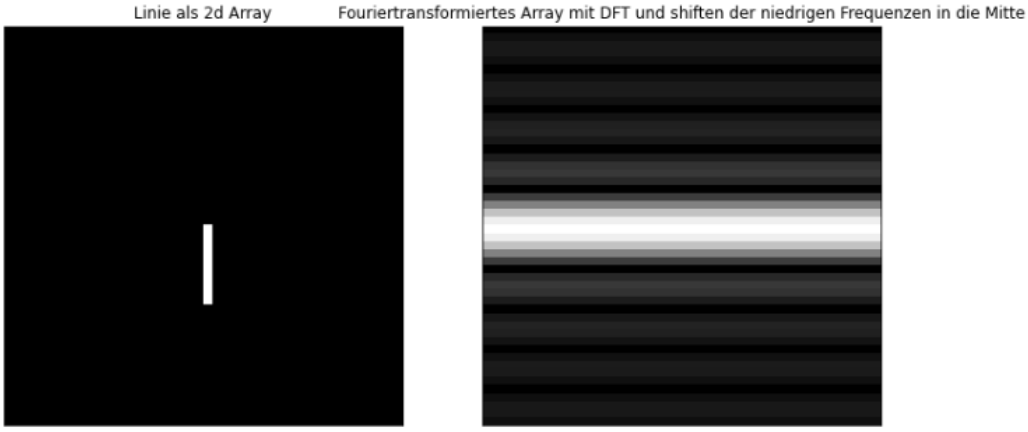






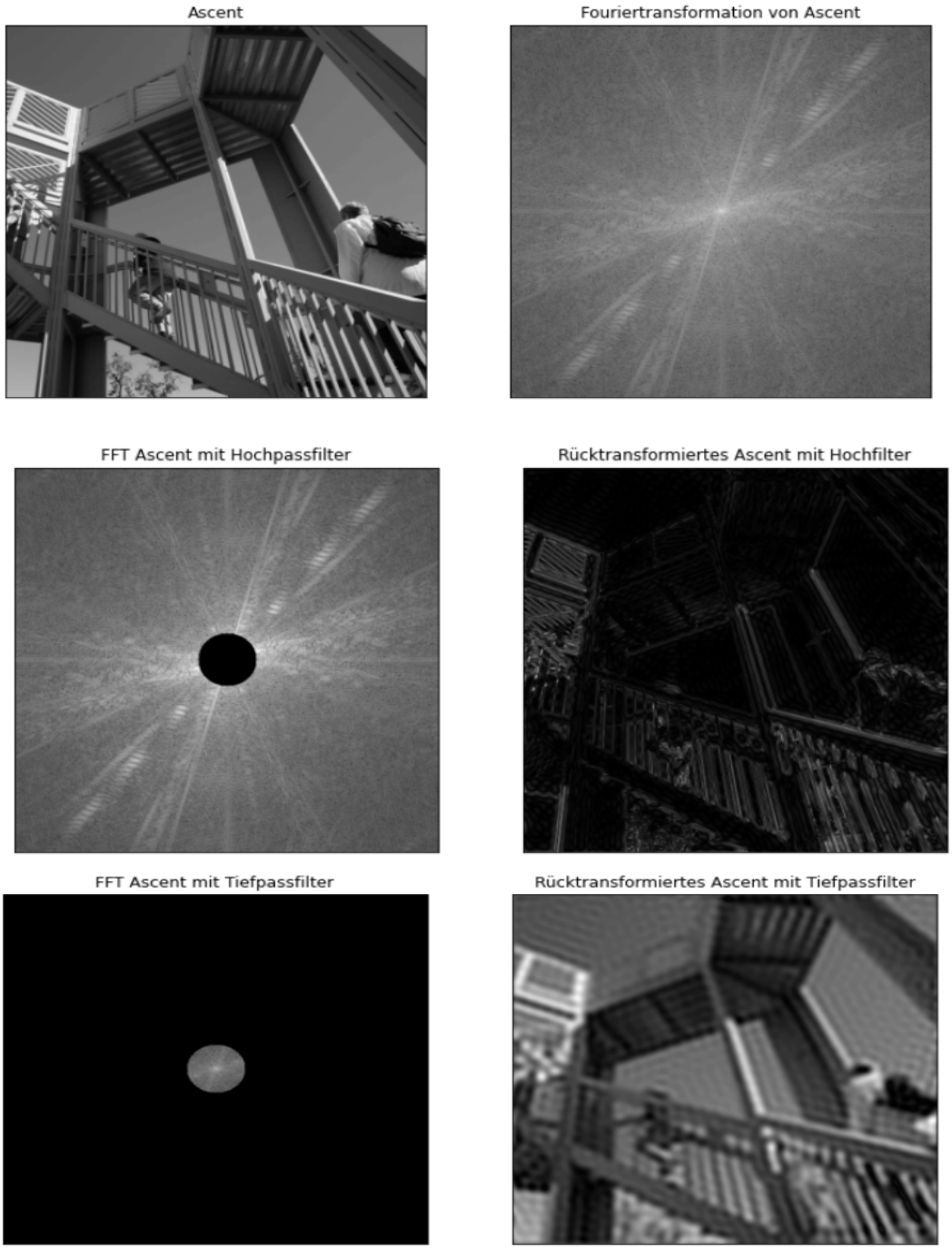

*Zur Überprüfung unseres DFT Algorithmus stehen uns nicht nur visuelle Mittel zur Verfügung(siehe Grafiken oben) sondern auch diverse Numpy Methoden. Ich habe mich für numpy allclose entschieden und führe dies Beispielhaft an meinem Kreis Bild vor. Numpy allclose überprüft ob die Arrays elementweise übereinstimmen mit gegebener Toleranz. Der Default Wert liegt bei 1e^-8 . Außerdem werden wir mit timeit noch kurz die Laufzeit zwischen unserem DFT und dem numpy FFT darstellen und sehen wieviel performanter FFT ist.*

```
#Überprüft Gleichheit von  dft_circle und fft_circle
print( np.allclose(dft_circle,fft_circle))

#Laufzeit
%timeit ft_own(circle)
%timeit np.fft.fft2(circle)
```

Abbildung 4: Überprüfung der Richtigkeit unseres DFT Algorithmus mit numpy allclose und dem numpy FFT. Laufzeitanalyse mit timeit unseres DFT und numpy FFT.

```
True
42 s ± 879 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
50.6 µs ± 301 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

Abbildung 5: Fouriertransformierte vom Ascent Bild und Anwendung des Tiefpass- und Hochpassfilters.



# 4 Diskussion

Im ersten Abschnitt dieses Kapitels, wollen wir auf die Unterschiede zwischen der "herkömmlichen"DFT-Implementierung und der FFT-Implementierung eingehen. Um eine DFT in einer Dimension zu ermitteln müssen wir jeden Wert des Vektors mit der Exponentialfunktion multiplizieren, die wiederum die Laufvariable enthält. Daraus folgt, dass wir N Multiplikationen mal N Additionen erhalten , was zu der Laufzeit von

$$\mathbb{O}(N^2) \tag{2}$$

Bei einem 2 dimensionalen Matrix kommen noch die jeweilige Anzahl der Spalten hinzu und wir erhalten somit eine Laufzeit von:

$$\mathbb{O}((MN)^2) \tag{3}$$

Wie wir im Abschnitt Resultate gesehen haben, stoßen viele Maschinen bei dem Algorithmus an die Grenzen der Berechenbarkeit. In unseren Beispielen habe ich aus diesem Grund Bilder erzeugt mit einer Pixelgröße, die nicht höher als 50x50 Pixel waren. Mit dem FFT Algorithmus von Cooley-Tukey wird die Laufzeit nun erheblich verbessert. Dieser benutzt den Ansatz des divide and conquer, das bedeutet das wiederholte Teilen des Problems in viele kleinere Teilprobleme. Auf diese kleinere Teilprobleme lassen sich dann die DFTs anwenden. Dies erreicht man, in dem man die Summen in gerade und ungerade Summanden aufteilt und darauf den DFT Algorithmus anwendet. Der wichtigste Aspekt des FFT Algorithmus ist jedoch das Ausnutzen von Symmetrien. Hierzu fragt man sich inwieweit man $\hat{a}_{N+k}$ vereinfachen kann(1 dimensionaler Fall). Man nutzt die Identität $e^{2\pi in} = 1$ aus:

$$\hat{a}_{N+k} = \sum_{n=0}^{N-1} a_n \cdot e^{-2i\pi(N+k)\frac{n}{N}} = \sum_{n=0}^{N-1} a_n \cdot e^{-2i\pi n} \cdot e^{\frac{-2i\pi kn}{N}} = \sum_{n=0}^{N-1} a_n \cdot e^{\frac{-2i\pi kn}{N}}$$

Damit wird schnell klar dass:

$$\hat{a}_{N+k} = \hat{a}_k$$

$$\hat{a}_{i\cdot N+k} = \hat{a}_k \text{ für alle } i \in \mathbb{Z}$$

Dieses Aufsplitten in kleinere Teilprobleme, kann man nun beliebig oft durchführen. Je nachdem welchen FFT Algorithmus man wählt, gibt es jedoch Limitierungen. Wie z.B. bei dem 2 Radix FFT, der nur so lange weiterläuft und aufsplittet bis die Größe des Teilarrays keine Potenz von 2 mehr ist. Somit erhält man im 1 dimensionalen Fall eine Laufzeit von:

$$\mathbb{O}(N(log_2(N)))$$

Analog haben wir dann natürlich im 2 dimensionalen Fall eine Laufzeit von:

$$\mathbb{O}(M \cdot N \cdot (log_2(MN))))$$

Im ein dimensionalen Fall erhalten wir für verschiedene Vektorgrößen:

| N | $10^3$ | $10^6$ | $10^9$ |
|---|---|---|---|
| $N^2$ | $10^6$ | $10^12$ | $10^18$ |
| $Nlog_2N$ | $10^4$ | $20 \cdot 10^6$ | $30 \cdot 10^9$ |

Wie oben schon einmal erwähnt möchte ich nun auf die Unterschiede der jeweiligen Filter eingehen. Filter in der Bildverarbeitung haben folgende Ziele:

1. Verminderung des Signalrauschens
2. Glättung
3. Kantendetektion
4. Beseitigung von Bildstörungen

Man unterscheidet i.A. zwischen linearen und nichtlinearen Filtern, wobei wir uns hier nur mit dem linearen Fall beschäftigen:

$$\tilde{a} = H \cdot a \tag{4}$$

Durch das Multiplizieren des Filters H auf das Bild a erhalten wir ein neues Bild. Wie oben schon angedeutet beschäftigen wir uns mit Tiefpassfiltern und Hochpassfiltern. Auswirkungen der Tiefpassfilter:

1. Das Bild wird "weicher"
2. Grauwertkanten werden verwischt

3. Rauschunterdrückung

4. Keine Auswirkungen im nicht-hochfrequenten Bereich des Bildes

Aus diesem Grund nenn man die Tiefpassfilter auch Glättungsfilter. Dazu gehören z.B.:

1. Rechteckfilter

2. Binomialfilter

3. Gaußfilter

Auswirkungen des Hochpassfilters:

1. Bild wird "härter"

2. feine Strukturen werden hervorgehoben

3. Grauwertübergänge bzw. Kanten werden verstärkt

Beispiele für diese Art von Filtern sind:

1. Scharfzeichnungsfilter

2. Laplace-Filter

3. Prewitt-Filter

4. Sobel-Filter

Wir gehen nun noch auf ein paar Details des Gauß - Filters ein. Der Gauß-Filter eines Bildes der Größe $n \cdot m$ und der Varianz $\sigma^2$ wird durch folgende Formel beschrieben:

$$g_{nm} = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{n^2+m^2}{2\sigma^2}} \tag{5}$$

Die Pixel des Bildes werden mit Hilfe der gaußschen Glockenkurve gewichtet. So werden die Pixel in der Umgebung der Mitte des Bildes stärker gewichtet. Der Gauß-Filter wirkt gleichmäßig auf hochfrequente Bildanteile wie z.B. Kanten oder isolierte Störungen. Diese werden weniger verwischt wie z.B. dem Rechteckfilter.

# 5 Zusammenfassung

Dieses Paper gab uns einen kurzen Einblick in die Methoden der Bild- und Signalverarbeitung. Mit Hilfe der diskreten Fourier Transformation waren wir in der Lage, diverse Bilder umzuwandeln und mit der inversen diskreten Fouriertransformation wieder zurück zu übersetzen. Wir haben über die Effizienz der DFT diskutiert und auch die schnellere Variante, dem sogenannten FFT eingeführt und angewendet. Dies ermöglichte es uns Filter anzuwenden, um z.B. das optische Erscheinungsbild zu verbessern. Außerdem haben wir über die Effekte, der oben genannten Filter gesprochen.

# Literatur


[1] Stefan Gerlach, Computerphysik, Springer Spektrum, 2. Auflage 2019, Kapitel 18.3

[2] Hans Rudolf Schwarz, Norbert Köckler, Numerische Mathematik, Vieweg+Teubner Verlag / Springer Fachmedien, 8., aktualisierte Auflage 2011, Kapitel 3.7.2

[3] https://www.physi.uni-heidelberg.de/Einrichtungen/AP/python/FFT_lena.html

[4] https://en.wikipedia.org/wiki/Discrete_Fourier_transform

[5] https://en.wikipedia.org/wiki/Fast_Fourier_transform

# Projekt2_Notebook

October 9, 2020

***Die diskrete Fourier Analyse***

*Author: Mokrane Yahiatene*

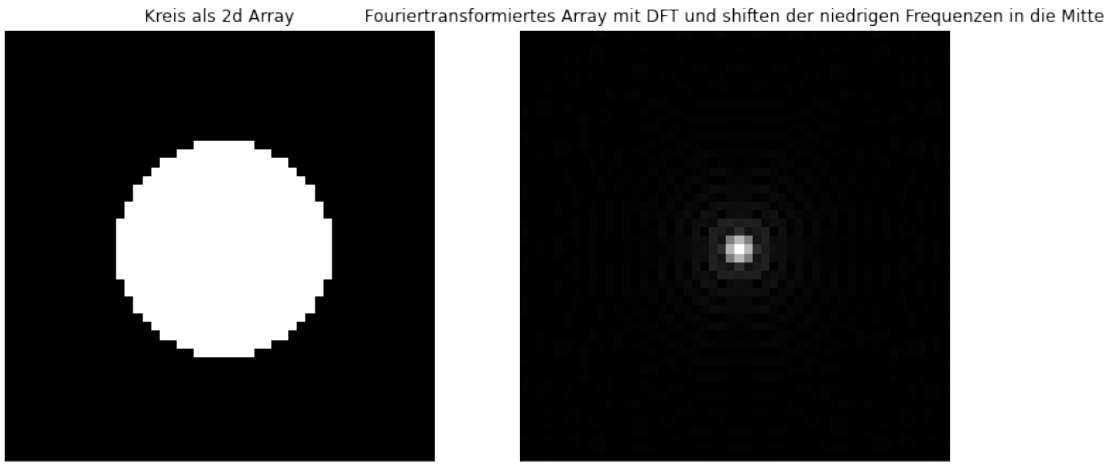
[52]:
```
import numpy as np
import matplotlib.pyplot as plt
from skimage import draw
from scipy import misc as ms
%matplotlib inline
# Hier wird die Größe des Bildarrays festgelegt
M=50
N=50
#Algorithmus der diskreten Fourier Transformation eines 2 dimensionalen Arrays
def ft_own(array):
    arr = np.zeros((M,N),dtype=complex)
    for k in range(0,M,1):
        for l in range(0,N,1):
            for i in range(0,M,1):
                for j in range(0,N,1):
                   arr[k][l] +=array[i][j] * np.exp(-2j*(np.pi*((i*k)/M))) * np.
 ↪exp(-2j*(np.pi*((j*l)/N)))

    return arr

#Algorithmus der diskreten  inversen Fourier Transformation eines 2␣
 ↪dimensionalen Arrays
def ift_own(array):
    arr = np.zeros((M,N),dtype=complex)
    for k in range(0,M,1):
        for l in range(0,N,1):
            for i in range(0,M,1):
                for j in range(0,N,1):
                   arr[k][l] +=np.divide(1,M*N)*array[i][j] * np.exp(2j*(np.
 ↪pi*((i*k)/M))) * np.exp(2j*(np.pi*((j*l)/N)))

    return arr
```

*Erstellen verschiedener Formen mit Hilfe von 2- dimensionalen Arrays befüllt mit Nullen und Einsen. Daraufhin wird auf jede Form die diskrete Fourier Transformation und die Fast Fourier*

*Transformation angewendet.*

[66]:
```
plt.figure(figsize=(12,12))
plt.subplot(1,2,1)
plt.xticks([])
plt.yticks([])
circle = np.zeros((M, N))
rows_circle, columns_circle = draw.disk((N/2,M/2),N/4)
circle[rows_circle, columns_circle] = 1
plt.imshow(circle.real,cmap='gray')
plt.title('Kreis als 2d Array')
plt.subplot(1,2,2)
plt.xticks([])
plt.yticks([])
plt.title('Fouriertransformiertes Array mit DFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
dft_circle = ft_own(circle)
#2 malige Anwenden von numpy roll um Spaltenanzahl/2 und Zeilenanzahl/2 anhand␣
 ↪der X und Y Achse ermöglicht
#das Verschieben der kleinen Frequenzen(Zeilen/Spalten) in die Mitte des Bildes
plt.imshow(np.abs(np.roll(np.roll(dft_circle,M//2,axis=0),N//
 ↪2,axis=1)),cmap='gray')
```

[66]:
```
<matplotlib.image.AxesImage at 0x7fc8fba15970>
```

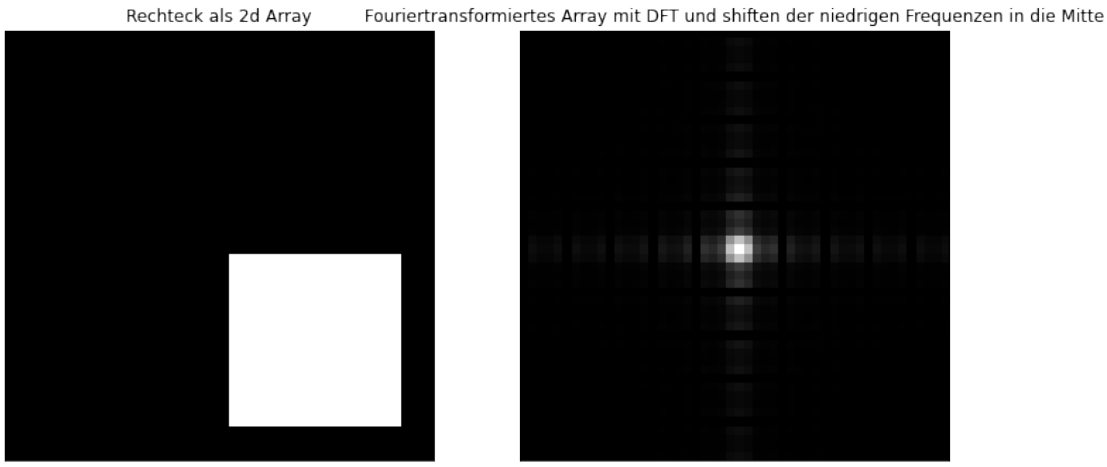



[54]:
```
plt.title('Fouriertransformiertes Array mit FFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
fft_circle = np.fft.fft2(circle)
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(np.fft.fftshift(fft_circle)),cmap='gray')
```

[54]: <matplotlib.image.AxesImage at 0x7fc900190910>

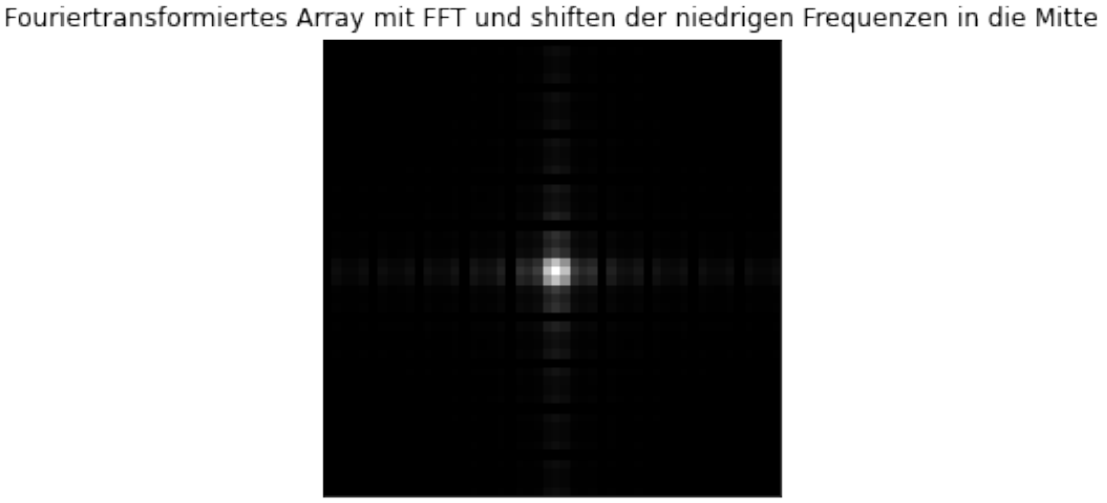

[55]:
```
plt.figure(figsize=(12,12))
plt.subplot(1,2,1)
plt.xticks([])
plt.yticks([])
plt.title('Rechteck als 2d Array')
rectangle = np.zeros((M, N))
rows_rectangle, columns_rectangle = draw.rectangle((N//3+10,M//3+10),(N//2+20,M/
 ↪/2+20))
rectangle[rows_rectangle, columns_rectangle] = 1
plt.imshow(rectangle,cmap='gray')
#2 malige Anwenden von numpy roll um Spaltenanzahl/2 und Zeilenanzahl/2 anhand␣
 ↪der X und Y Achse ermöglicht
#das Verschieben der kleinen Frequenzen(Zeilen/Spalten) in die Mitte des Bildes
plt.subplot(1,2,2)
plt.title('Fouriertransformiertes Array mit DFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
dft_rectangle = ft_own(rectangle)
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(np.roll(np.roll(dft_rectangle,M//2,axis=0),N//
 ↪2,axis=1)),cmap='gray')
```

[55]: <matplotlib.image.AxesImage at 0x7fc8fba0e850>




[56]:
```
plt.title('Fouriertransformiertes Array mit FFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
fft_rectangle = np.fft.fft2(rectangle)
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(np.fft.fftshift(fft_rectangle)),cmap='gray')
```

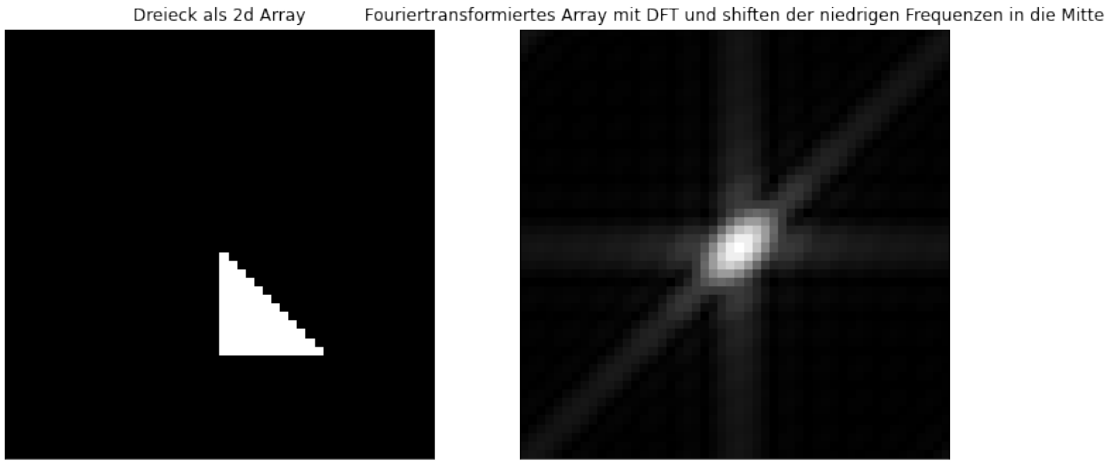
[56]: <matplotlib.image.AxesImage at 0x7fc9001d0220>




[57]:
```
plt.figure(figsize=(12,12))
plt.subplot(1,2,1)
plt.title('Dreieck als 2d Array')
triangle = np.zeros((M, N))
```

```
rows_triangle, columns_triangle = draw.polygon((N-N//2,M-M//4,M-M//4),(N-N//
 ↪2,M-M//4,M//2),shape=triangle.shape)
triangle[rows_triangle,columns_triangle]=1
plt.xticks([])
plt.yticks([])
plt.imshow(triangle,cmap='gray')
plt.subplot(1,2,2)
plt.title('Fouriertransformiertes Array mit DFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
dft_triangle = ft_own(triangle)
#2 malige Anwenden von numpy roll um Spaltenanzahl/2 und Zeilenanzahl/2 anhand␣
 ↪der X und Y Achse ermöglicht
#das Verschieben der kleinen Frequenzen(Zeilen/Spalten) in die Mitte des Bildes
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(np.roll(np.roll(dft_triangle,M//2,axis=0),N//
 ↪2,axis=1)),cmap='gray')
```

[57]: <matplotlib.image.AxesImage at 0x7fc8fb24a3d0>




[58]:
```
plt.title('Fouriertransformiertes Array mit FFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
fft_triangle = np.fft.fft2(triangle)
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(np.fft.fftshift(fft_triangle)),cmap='gray')
```

[58]: <matplotlib.image.AxesImage at 0x7fc8fb25c8e0>

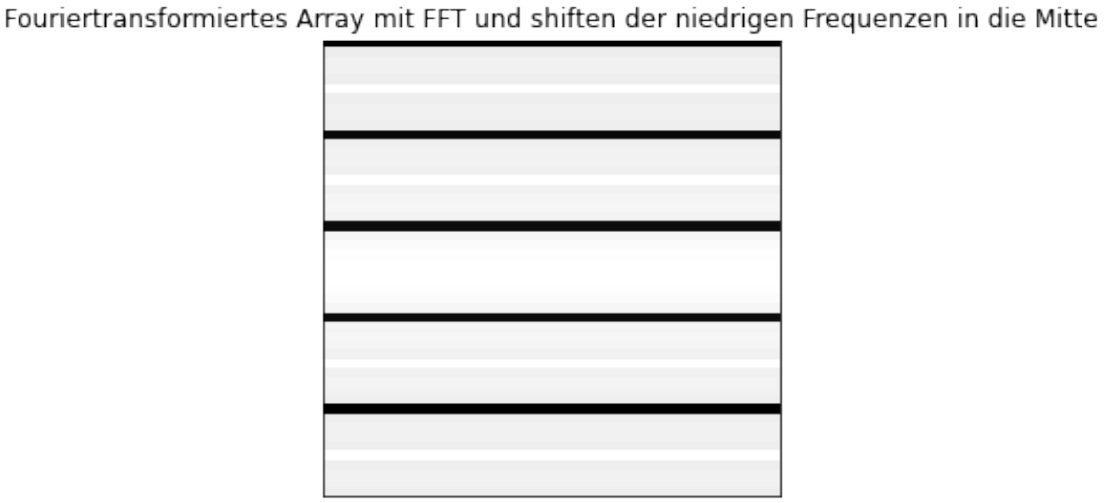

[59]:
```
plt.figure(figsize=(12,12))
plt.subplot(1,2,1)
plt.title('Linie als 2d Array')
line = np.zeros((M, N))
rows_line, columns_line = draw.line(N-N//3,N//2,M//2,M//2)
line[rows_line,columns_line]=1
plt.xticks([])
plt.yticks([])
plt.imshow(line,cmap='gray')
plt.subplot(1,2,2)
plt.title('Fouriertransformiertes Array mit DFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
dft_line = ft_own(line)
plt.xticks([])
plt.yticks([])
#2 malige Anwenden von numpy roll um Spaltenanzahl/2 und Zeilenanzahl/2 anhand␣
 ↪der X und Y Achse ermöglicht
#das Verschieben der kleinen Frequenzen(Zeilen/Spalten) in die Mitte des Bildes
plt.imshow(np.abs(np.roll(np.roll(dft_line,M//2,axis=0),N//
 ↪2,axis=1)),cmap='gray')
```

[59]:
```
<matplotlib.image.AxesImage at 0x7fc8fb410a30>
```


[60]:
```
plt.title('Fouriertransformiertes Array mit FFT und shiften der niedrigen␣
 ↪Frequenzen in die Mitte')
fft_line = np.fft.fft2(line)
plt.xticks([])
plt.yticks([])
plt.imshow(np.log(np.abs(np.fft.fftshift(fft_line))),cmap='gray')
```

```
<ipython-input-60-a10cc7e4c082>:5: RuntimeWarning: divide by zero encountered in
log
  plt.imshow(np.log(np.abs(np.fft.fftshift(fft_line))),cmap='gray')
```

[60]: `<matplotlib.image.AxesImage at 0x7fc8fb52b310>`




*Zur Überprüfung unseres DFT Algorithmus stehen uns nicht nur visuelle Mittel zur Verfügung(siehe*

*Grafiken oben) sondern auch diverse Numpy Methoden. Ich habe mich für numpy allclose entschieden und führe dies Beispielhaft an meinem Kreis Bild vor. Numpy allclose überprüft ob die Arrays elementweise übereinstimmen mit gegebener Toleranz. Der Default Wert liegt bei 1e^-8 . Außerdem werden wir mit timeit noch kurz die Laufzeit zwischen unserem DFT und dem numpy FFT darstellen und sehen wieviel performanter FFT ist.*

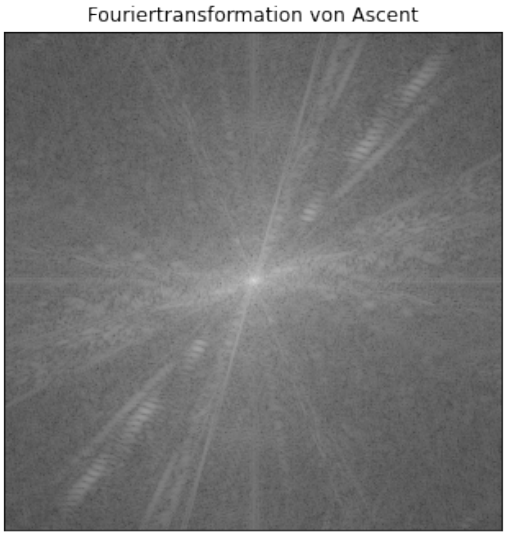
[61]:
```
#Überprüft Gleichheit von  dft_circle und fft_circle
print( np.allclose(dft_circle,fft_circle))

#Laufzeit
%timeit ft_own(circle)
%timeit np.fft.fft2(circle)
```

```
True
42 s ± 879 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
50.6 µs ± 301 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

*Im Folgenden importieren wir aus der scipy misc library das ascent Image und wenden die Fourier Transformation an, umd dann diverse Fourier Filter anzuwenden. Wir werden auch nach der Transformation die log Funktion auf das Bild anwenden, damit wir den Kontrast erhöhen, um mehr zu erkennen. Daraufhin rucktransformieren wir das Bild wieder um die verschiedenen Auswirkungen der Filter zu diskutieren.*

[62]:
```
plt.figure(figsize=(12,12))
plt.subplot(1,2,1)
plt.title('Ascent')
ascent = ms.ascent()
plt.xticks([])
plt.yticks([])
plt.imshow(ascent,cmap='gray')
plt.subplot(1,2,2)
plt.title('Fouriertransformation von Ascent')
fft_ascent= np.fft.fft2(ascent)
plt.xticks([])
plt.yticks([])
plt.imshow(np.log(np.abs(np.fft.fftshift(fft_ascent))),cmap='gray')
```

[62]:
```
<matplotlib.image.AxesImage at 0x7fc8fb172670>
```

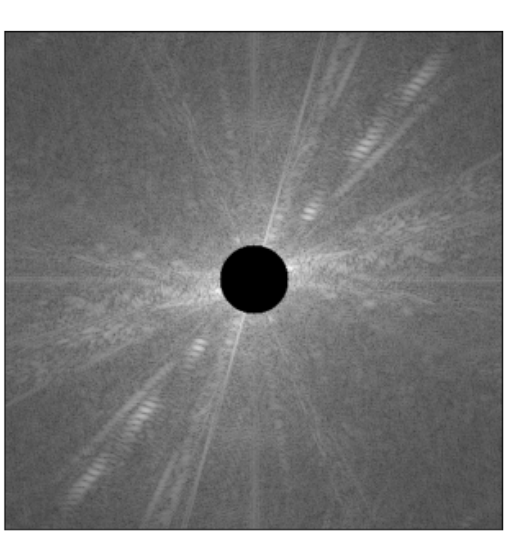




[63]:
```python
def hochpass(img):
    hochpass_arr = np.zeros(img.shape)
    radius = 35  #Radius der Kreisblende
    #generiere Lochblende:
    ax, ay = img.shape
    for i in np.arange(len(img)):
        for j in np.arange(len(img[0])):
            if (i-ax/2)**2 + (j-ay/2)**2 > radius**2:
                hochpass_arr[i][j]=1
            else:
                hochpass_arr[i][j]=0
    return hochpass_arr
def tiefpass(img):
    tiefpass_arr = np.zeros(img.shape)
    radius = 35  #Radius der Kreisblende
    #generiere Lochblende:
    ax, ay = img.shape
    for i in np.arange(len(img)):
        for j in np.arange(len(img[0])):
            if (i-ax/2)**2 + (j-ay/2)**2 < radius**2:
                tiefpass_arr[i][j]=1
            else:
                tiefpass_arr[i][j]=0
    return tiefpass_arr
```

[64]:
```python
#Shiften der Werte des Images um danach den Filter anzu wenden
shift_fft_ascent = np.fft.fftshift(fft_ascent)
#Anwenden des Hochpassfilter auf transformiertes Image
hochpass_ascent = shift_fft_ascent * hochpass(ascent)
```

```
plt.figure(figsize=(12,12))
plt.title('FFT Ascent mit Hochpassfilter')
plt.subplot(1,2,1)
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(hochpass_ascent)**0.2, cmap = 'gray') #Potenz für die␣
 ↪Kontrastanpassung
#Rückshiften und Rücktransformation des Ascent Image
plt.subplot(1,2,2)
plt.title('Rücktransformiertes Ascent mit Hochfilter')
ascent_filtered = np.abs(np.fft.ifft2(np.fft.ifftshift(hochpass_ascent)))
plt.xticks([])
plt.yticks([])
plt.imshow(ascent_filtered,cmap='gray')
```

[64]: <matplotlib.image.AxesImage at 0x7fc8fb0f7e80>

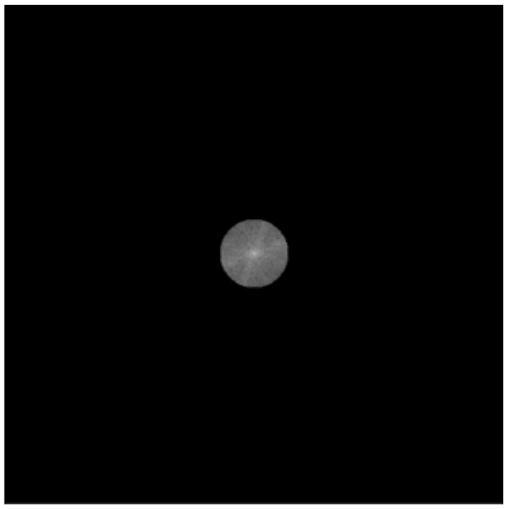



[65]:
```
tiefpass_ascent = shift_fft_ascent * tiefpass(ascent)
plt.figure(figsize=(12,12))
plt.subplot(1,2,1)
plt.title('FFT Ascent mit Tiefpassfilter')
plt.xticks([])
plt.yticks([])
plt.imshow(np.abs(tiefpass_ascent)**0.15, cmap = 'gray') #Potenz für die␣
 ↪Kontrastanpassung
#Rückshiften und Rücktransformation des Ascent Image
plt.subplot(1,2,2)
plt.title('Rücktransformiertes Ascent mit Tiefpassfilter')
```

```
ascent_filtered = np.abs(np.fft.ifft2(np.fft.ifftshift(tiefpass_ascent)))
plt.xticks([])
plt.yticks([])
plt.imshow(ascent_filtered,cmap='gray')
```

[65]: <matplotlib.image.AxesImage at 0x7fc8fb052970>

FFT Ascent mit Tiefpassfilter



Rücktransformiertes Ascent mit Tiefpassfilter