Compierchio_Chris_Lab2

February 15, 2022

General tips for computing and plotting discrete fourier transforms (DFT)

- Let g(t) be some time signal that's sampled at dt to get a discrete array/list `g = [ … ]`
- You don't need to code your own DFT, use numpy: `A = numpy.fft.fft(g) * dt`
- You should also use `numpy.fft.fftshift( A )` to shift the fft output such that the 0-frequency component is centered (see why here https://docs.scipy.org/doc/numpy/reference/routines.fft.html#background-information) which you probably want when plotting
- You can use `f_axis = numpy.fft.fftshift( numpy.fft.fftfreq(len(g), dt) )` to create the frequency axis for plotting the shifted spectrum
- Keep in mind the fft output is in general complex, so to compare two fourier transforms (e.g. DFT vs CFT) you should compare either the real and imaginary parts ($z = x + iy$), or the phase and amplitude ($z = re^{i\theta}$). Amplitude plots are most useful for this lab, show all 4 aspects though if you want.
- Note numpy fft assumes the time signal starts at $t = 0$, if yours doesn't you should center it at zero. If you don't then the complex components (x and y, or phase) will be off, but the amplitude should not change (why? analytically, recall that axis shifts in either domain are equivalent to complex exponential scaling, which has amplitude 1, in the other domain).
- If your time signal g is centered at zero, a hack to 'rotate it' to start at zero (and then take the fft and fftshift that) is to do: `fftshift( fft( ifftshift(g) ) )` (you may see mention of this online)

1 Fourier transform of Gaussian Functions (6 pts)

A common function used for the convolution of time series data is the Gaussian function

$$g(t) = \frac{1}{\sqrt{\pi} t_H} e^{-(t/t_H)^2},$$

where t_H is the half duration.

1. Plot $g(t)$ for $t_H = 15$ and $t_H = 45$ sec on the same graph with domain $[-100, 100]$ and $dt = 10^{-3}$.

2. The analytical formula for the Fourier transform of $g(t)$ is

$$G(\omega) = e^{\frac{-\omega^2 t_H^2}{4}}.$$

Compute the discrete Fourier transform (DFT) for both sampled $g(t)$ time series, and compare them to the analytical $G(\omega)$ for both t_H's on the same graph.

Hints:

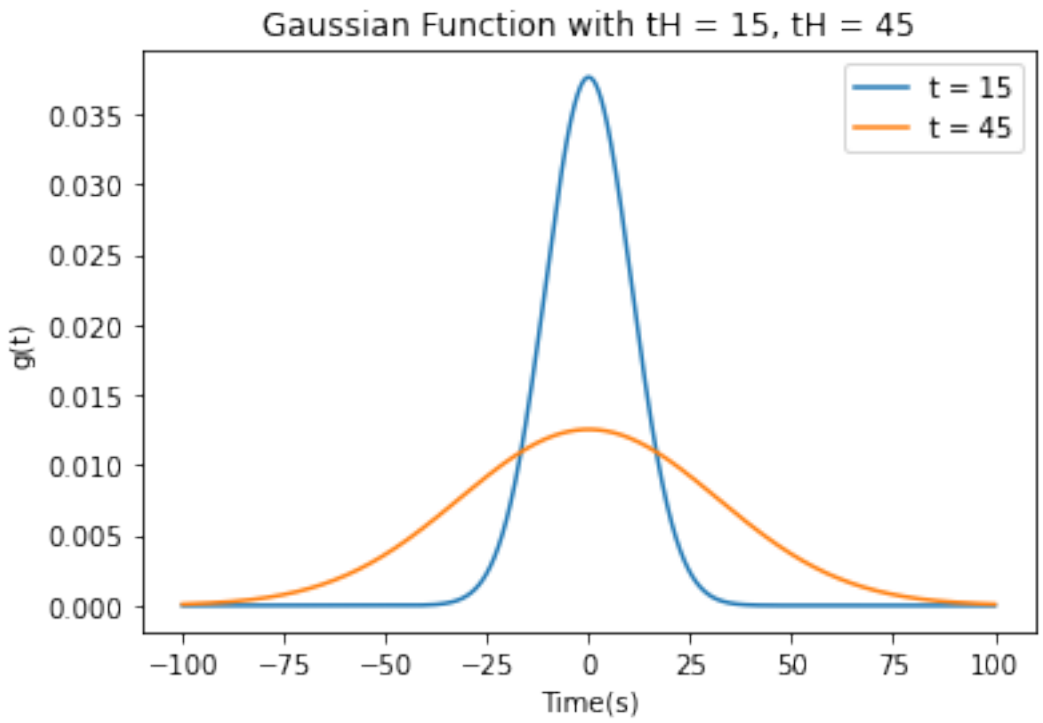
- As numpy fft assumes signal starts from time 0, you can use the shift property of Fourier transform to first shift the $g(t)$ to start from zero, and after `fftshift(fft())` operations, multiply the spectrum by complex exponential sinusoid function.
- You need to sample the theoretical curve $G(w)$ with `w_axis = 2*pi*f_axis`, or else rewrite it as $G(f = \frac{w}{2\pi})$ if you'd rather sample it with `f_axis`
- As a guide (so you can be confident of your fft utilization for the remainder of the lab), we expect that the amplitudes (use `numpy.abs(…)`) of the discrete FT and the continuous FT essentially match. The phase won't necessarily match.

3. Comment on the effect of filtering a general input time function $f(t)$ by $g(t)$ (i.e. convolution of $f(t)$ with $g(t)$), and explain the difference in filtered output after applying Gaussian functions with $t_H = 15$ or 45 secs.

4. Comment on how this is related to the time-frequency uncertainty principle (a signal cannot be infinitesimally sharp both in time and frequency).

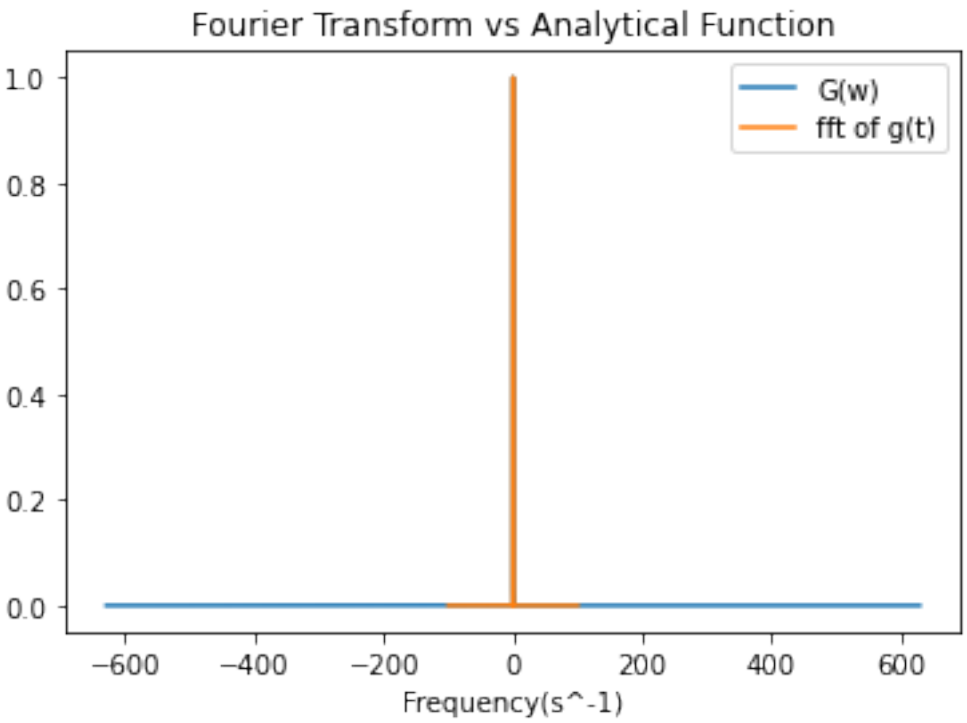
[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import scipy as sp
from scipy import *

def gauss1(t, th):
    return (np.e**(-(t/th)**2)/(np.sqrt(np.pi)*th))

t = np.arange(-100, 100, 0.001)
th = 15

plt.plot(t, gauss1(t, th), label = "t = 15")
plt.title("Gaussian Function with tH = 15, tH = 45")
plt.xlabel("Time(s)")
plt.ylabel("g(t)")

th = 45
plt.plot(t, gauss1(t, th), label = "t = 45")
plt.xlabel("Time(s)")
plt.ylabel("g(t)")
plt.legend(loc = "best")
```

[1]:
```
<matplotlib.legend.Legend at 0x29a2a5a0a90>
```



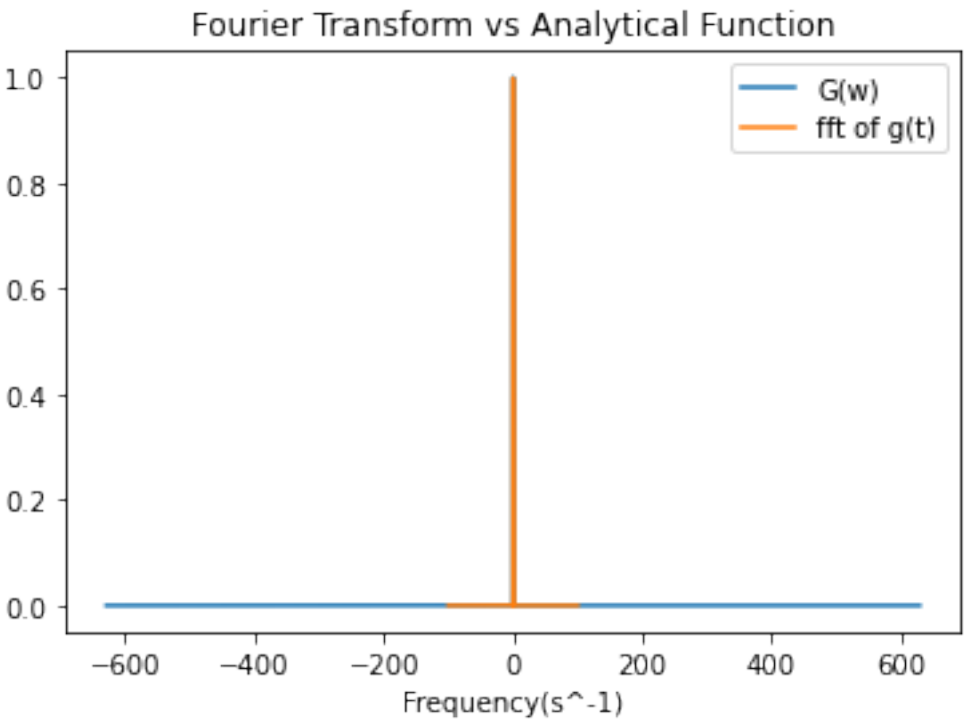
[32]:
```
th = 15

def G(w, th):
    return np.e**((-((w)**2)*(th)**2)/4)

w = 2*np.pi*t

plt.plot(w, G(w, th), label = "G(w)")

plt.plot(t, np.abs(np.fft.fftshift(np.fft.fft(gauss1(t,th))*0.001)), label =␣
↪"fft of g(t)")
plt.title("Fourier Transform vs Analytical Function")
plt.xlabel("Frequency(s^-1)")
plt.legend(loc = "best")
```

[32]:
```
<matplotlib.legend.Legend at 0x2b921420100>
```



[33]:
```
th = 45

plt.plot(w, G(w, th), label = "G(w)")

plt.plot(t, np.abs(np.fft.fftshift(np.fft.fft(gauss1(t,th))*0.001)), label =␣
↪"fft of g(t)")
plt.title("Fourier Transform vs Analytical Function")
plt.xlabel("Frequency(s^-1)")
plt.legend(loc = "best")
```

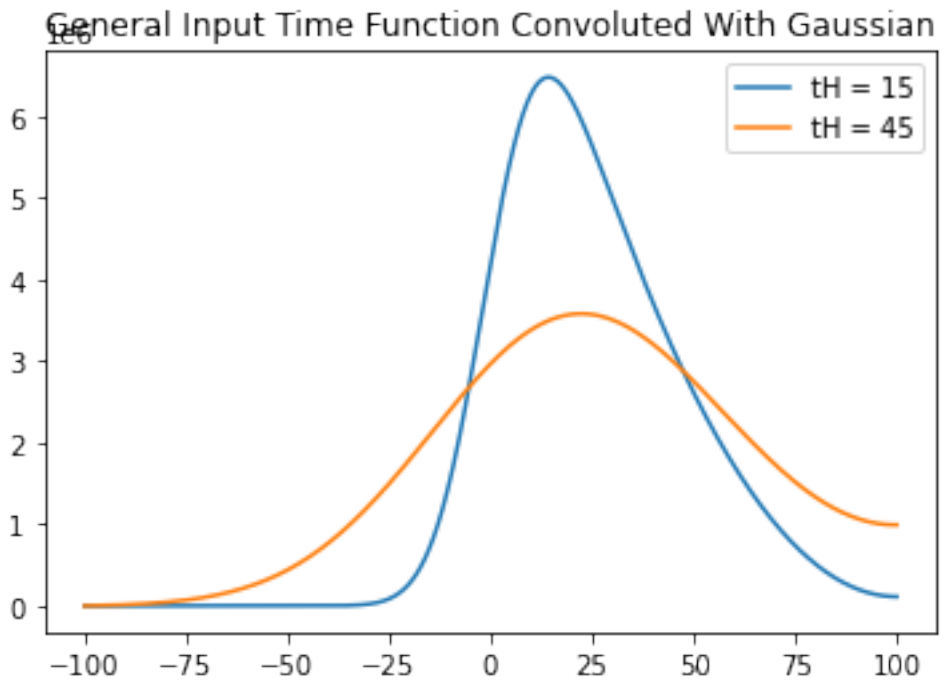
[33]: <matplotlib.legend.Legend at 0x2b921480190>



PART 3

[34]:
```
f = t**2
th = 15
plt.plot(t, np.convolve(f, gauss1(t, th))[:200000], label = "tH = 15")

th = 45
plt.plot(t, np.convolve(f, gauss1(t, th))[:200000], label = "tH = 45")
plt.title("General Input Time Function Convoluted With Gaussian")
plt.legend(loc="best")
```

[34]:
```
<matplotlib.legend.Legend at 0x2b9214f3ca0>
```



3.) Above, I convolved f = t^2 with the Gaussian function and it made the function look more like a Gaussian function. If i extend the axis limits, the function becomes symmetrical (like an M shape). Also, the higher the tH value, the lower the peak of the plot.

4.) This shows an example of the time-frequency uncertainty priciple as the fft plots show an infitesimally sharp peak while the the original plots are smooth Gaussians.

2 Fourier transform of Window Functions (6 pts)

A continuous time signal $f(t)$ can be truncated into a signal of finite length T by window functions $b(t)$:

$$g(t) = f(t)b(t)$$

Typical window functions include:

- Boxcar function
$$b(t) = \begin{cases} 1 & 0 \le t \le T \\ 0 & \text{else} \end{cases}$$
- Hann window
$$b(t) = \begin{cases} \frac{1}{2}\left(1 - \cos\frac{2\pi t}{T}\right) & 0 \le t \le T \\ 0 & \text{else} \end{cases}$$

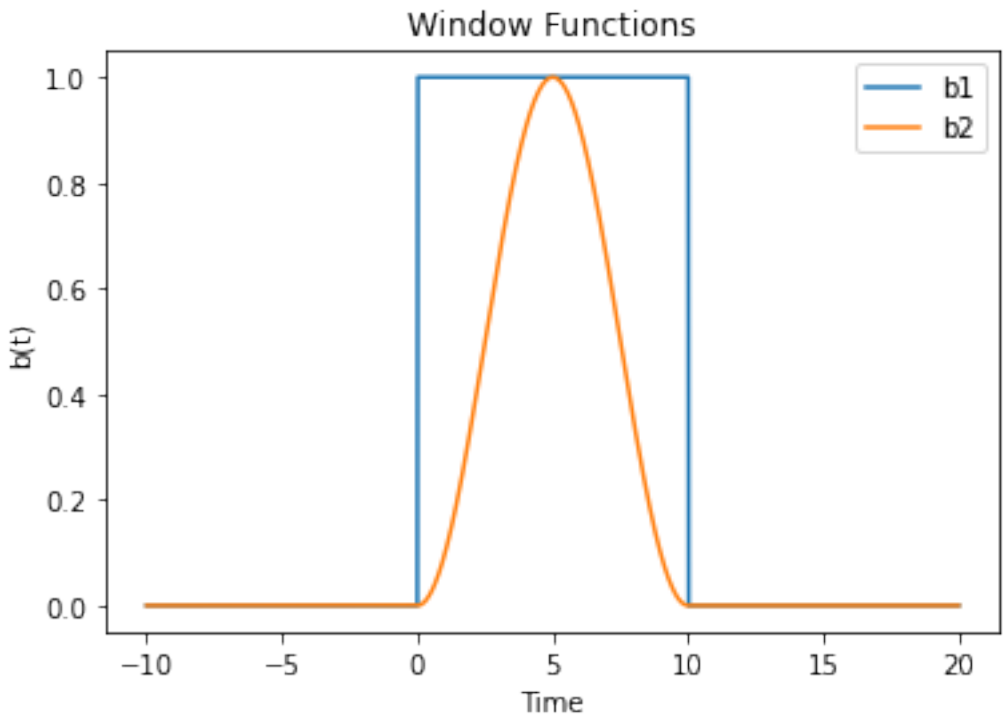
Now let $T = 10$ seconds, and sample both window functions by $\Delta t = 0.01$ seconds:

1. Plot both window functions on the same graph.

2. Calculate the Fourier transform of both functions by numpy `fft()`. Pay extra attention to how you interpret the corresponding frequencies of output results from python. (*Hint*: `fftshift()` may be useful. Also pay attention to the length of the input signal (> 10 sec), as it dictates the frequency resolution for the spectrum.)
3. Plot the Fourier transform of both functions in the appropriate frequency range on the same graph.
4. Based on the FTs, comment on the effect of truncating a continuous time series by either window on its frequency spectrum $G(\omega)$ compared to the original spectrum $F(\omega)$.
5. Speculate on the advantages and disadvantages of boxcar and Hann window functions for truncation.

[45]:
```
def b1(t1, T):
    if 0 <= t1 and t1 <= T:
        return 1
    else:
        return 0

t = np.arange(-10, 20, 0.01)

y = []
for i in range(len(t)):
    y.append(b1(t[i],10))

plt.plot(t, y, label = "b1")

def b2(t, T):
     if 0 <= t and t <= T:
        return 0.5*(1-np.cos((2*np.pi*t)/T))
     else:
        return 0

y = []
for i in range(len(t)):
    y.append(b2(t[i],10))

plt.plot(t, y, label = "b2")
plt.title("Window Functions")
plt.xlabel("Time")
plt.ylabel("b(t)")
plt.legend(loc="best")
```

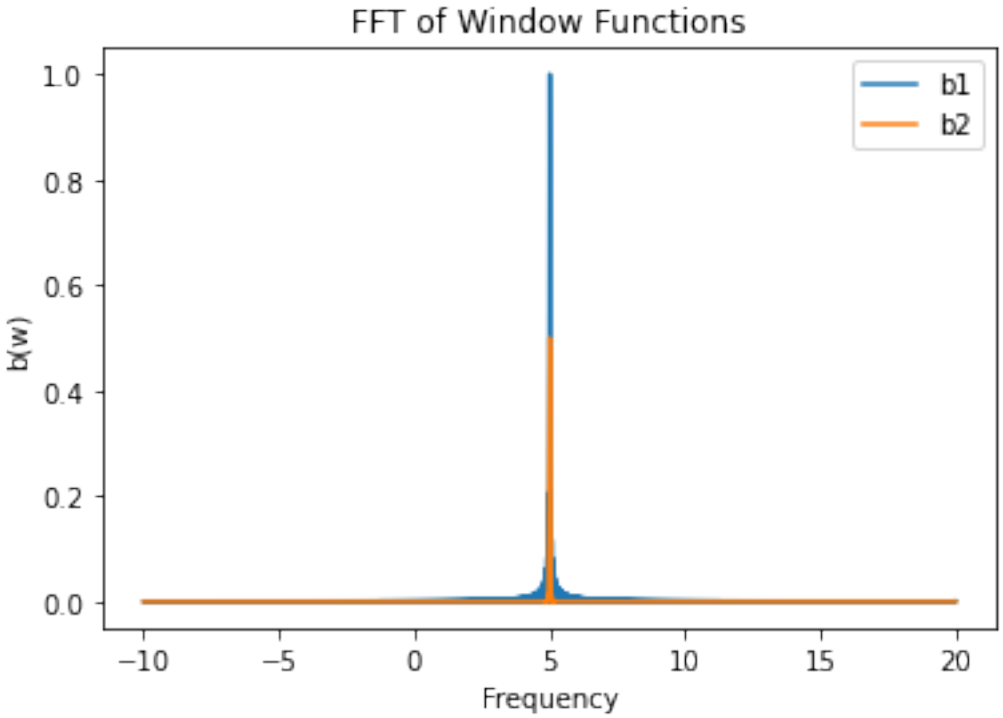
[45]:
```
<matplotlib.legend.Legend at 0x2b9269973a0>
```



[44]:
```
def b1(t1, T):
    if 0 <= t1 and t1 <= T:
        return 1
    else:
        return 0

t = np.arange(-10, 20, 0.01)

y = []
for i in range(len(t)):
    y.append(b1(t[i],10))

plt.plot(t, np.abs(np.fft.fftshift(np.fft.fft(y)*0.001)), label = "b1")

def b2(t, T):
     if 0 <= t and t <= T:
        return 0.5*(1-np.cos((2*np.pi*t)/T))
     else:
        return 0

y = []
for i in range(len(t)):
    y.append(b2(t[i],10))
```

```
plt.plot(t, np.abs(np.fft.fftshift(np.fft.fft(y)*0.001)), label = "b2")
plt.title("FFT of Window Functions")
plt.xlabel("Frequency")
plt.ylabel("b(w)")
plt.legend(loc="best")
```

[44]: <matplotlib.legend.Legend at 0x2b924a04460>



4.) This is similar to convolving the time series function with the boxcar function.

5.) Based on the FFTs above, it seems truncating with boxcar functions creates noise around the spike, whereas the window function produces little to no noise around this region. Zooming in on the functions at around x = 5, we can see the boxcar truncation oscillates a lot more than the window function. Perhaps these results are because the window function is smoother like a Gaussian.

3 Radial Distribution Function (12 pts)

Background

Liquids have no fixed internal structure. Yet they do have some short range order in the sense that they have preferred intermolecular spacings, which are determined by the locations of minima in the intermolecular potentials. The microscopic structure of liquids is often characterized by a quantity

known as the Radial Distribution Function $g(r)$, which is essentially the probability (Relative to the average probability, which means that $g(r)$ tends to 1 at large r, where the neighbour is too far away to feel any interaction.) that a molecule has a neighbouring molecule at distance r. Typically $g(r)$ shows a value that approaches zero at small r since molecules cannot occupy the same space; it also shows a peak at the preferred distance of nearest neighbours, and secondary peaks at preferred distances of more distant neighbours. If a suitable collimated beam of particles (e.g. X-rays or neutrons) is sent through a sample of the liquid, some of the particles are scattered. The number of particles scattered through a given angle is related to the Fourier Transform of $g(r)$ evaluated at the wavenumber k corresponding to the transverse momentum transfer associated with that scattering angle. Kittel derives this relationship in Chapter 17 of Introduction to Solid State Physics.

If this all sounds complicated, all you need to know here is that something called the Structure Factor $S(k)$ is effectively measured by looking at the scattered intensity as a function of scattering transverse wavenumber k (proportional to scattering angle), and that the Radial Distribution Function is related to it by

$$g(r) = 1 + \frac{1}{2\pi^2 \rho r} \int_0^\infty k\,(S(k) - 1)\sin(kr)dk$$

where ρ is liquid number density (number of atoms per unit volume, computable from the three constants mentioned in the introduction), k is wavenumber, and r is radius.

1. You may have noticed some resemblance between expression (6) and the Fourier transform. First show that the integration part $\int_0^\infty k(S(k) - 1)\sin(kr)dk$ can be rewritten as

$$p(r) = \int_{-\infty}^\infty \frac{1}{2i} k\,(S(k) - 1)\,e^{ikr}dk.$$

   Hint: The structure factor S(k) is even, since there should be no reason why scattering intensity would be different for one direction (+k) compared to its opposite (−k). Using the fact that S(k) is even may be useful.

2. Now we can make some connections between the Radial Transfer Function and the Fourier Transform, if we substitute $r \to t$ and $k \to \omega$. What is the Fourier transform $P(k)$ of $p(r)$? Is $P(k)$ a real, imaginary or general complex function? Is it even or odd? How will these affect $p(r)$? Is that what you expect? Plot $P(k)$ as a function of k ranging from -15A^{-1} to 15A^{-1} based on `argon.py` (i.e. import and use the variables defined there).

   Hint: In constructing $S(k)$ from `argon.py`, you should make an "even" array twice the length (minus 1) of YanData. YanData represents the structure factor (i.e. $S(k)$) for argon sampled at the dk defined in the `argon.py` file. It's specifically $S(k)$ sampled from `k = 0` to `k = len(YanData) * dk`, so create an even function out to the same length in the negative direction (i.e. the "k-axis" it's sampled on would be `-len(YanData)*dk, ... 0, ..., + len(YanData)*dk`).

3. Write a Python function `[gn, rn] = RDFcalc(S, dk, rho)` to calculate Radial Distribution Function $g(r)$ from Structure Factor $S(k)$ data, sampled at dk, and density ρ, and output the RDF vector g_n and its corresponding radial distance vector r_n.

   Hint: for Python `fft()` and `ifft()` functions, realize that the values of the Fourier Transform corresponding to negative frequencies are stored in the second half of the arrays given to

(`ifft`) or obtained from it (`fft`). You also have to study the difference between the DFT and FT to multiply the right factors.

4. With the data provided in `argon.py`, compute the corresponding Radial Distribution Function $g(r)$. Plot your results for r from 0 to 15A;. Over what range of radius can you trust your result?

   Hint: To check if your results make sense, recall that $g(r)$ is related to the probability that a molecule has a neighbouring molecule at distance r, therefore, should be close to 0 when $r \to 0$, i.e. two molecules can not occupy the same space, and you can set $g(r = 0) = 0$. Recall $\lim_{r\to\infty} g(r) = 1$. Also note the unit ρ used in $g(r)$ formula (6).

5. From the $g(r)$ you computed, estimate the average molecular radius R_a of liquid argon. Give your reasoning and state what accuracy you can justify for your estimation.

6. Now we explore the effect of sampling range. Yan sampled in wavenumber k out to $k_{max} = 15.24\text{A}^{-1}$, and he could have saved himself work by not collecting as much data, i.e., reducing k_{max}. But how much could he have reduced the sampling length k_{max}, while still seeing distinct peaks in the Radial Distribution Function? Also explain theoretically what you observe.

   Hint: Plot on top of the $g(r)$ obtained in Part 4, the $g(r)$'s you compute for a series of k_{max} values. You can try half k_{max} each time to look for changes. For the theoretical explanations for part 6 and 7, realize the interchangability of $t \leftrightarrow \omega$ ($r \leftrightarrow k$).

7. To explore the effect of data sampling, let's assume Yan decided to save his work by sampling less often (i.e. increasing dk). How large a dk can he use to be able to still recover the first two peaks clearly? State your answers and a theoretical justification for what you expect to see if you increase dk too much.

   Hint: Plot on top of the $g(r)$ obtained from `argon.py` data, the $g(r)$'s you obtain when you subsample the same dataset. Try doubling dk each time to observe the effect of coarser sampling.

4 PART 1

Using Euler's sine formula:

$$sin(kr) = \frac{e^{ikr} - e^{-ikr}}{2i}$$

we can sub this into the integral to obtain:

$$p(r) = \int_0^\infty k\,(S(k) - 1)\,\frac{e^{ikr} - e^{-ikr}}{2i} dk$$

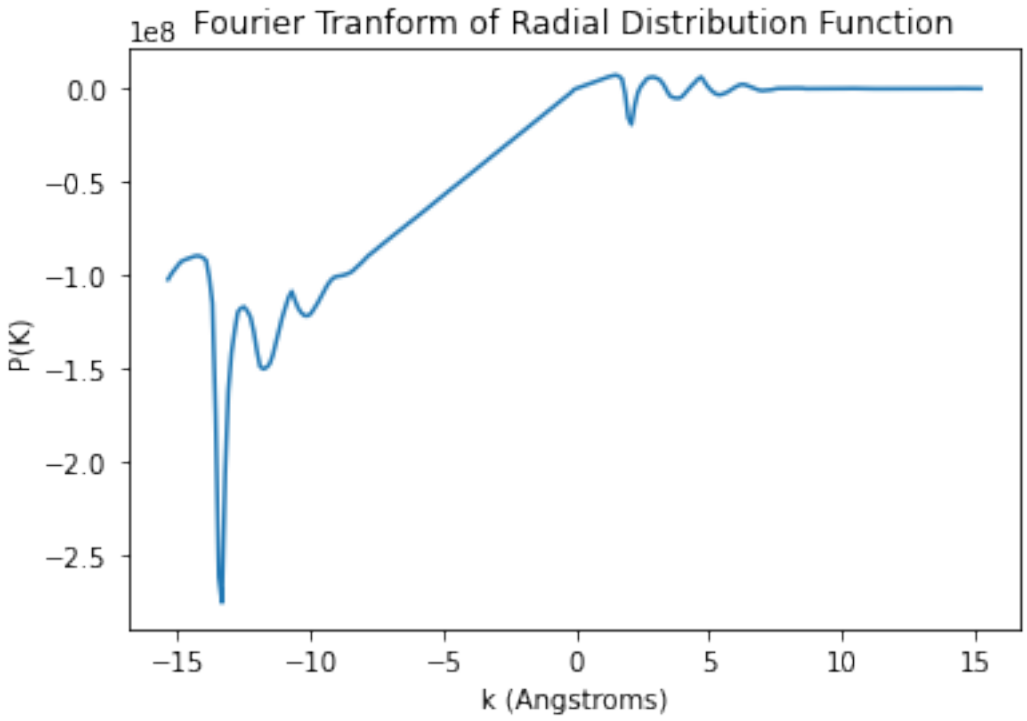
but since S(k) is even, we can ignore the negative portion and replace the integral bounds with $-\infty$ and ∞:

$$p(r) = \int_{-\infty}^\infty \frac{k}{2i}(S(k) - 1)e^{ikr} dk$$

5 PART 2

[20]:
```
from argon import *

Sk = np.flip(YanData)
Sk = np.append(YanData, 0)
Sk = -Sk
Sk = np.append(Sk,YanData)

r = ((molWeight)/(2*np.pi*Navogadro*massRho))**(1/3)
k = np.arange(-len(YanData)*dk, len(YanData)*dk, 0.1195330739)

P = (1/(2*1j*np.pi*massRho*r))*k[:257]*(Sk-1)
plt.plot(k[:257], np.imag(P))
plt.title("Fourier Tranform of Radial Distribution Function")
plt.xlabel("k (Angstroms)")
plt.ylabel("P(K)")
```

[20]:
```
Text(0, 0.5, 'P(K)')
```



The Fourier Transform of p(r) is:

$$P(k) = \frac{k(S(k) - 1)}{2i\pi\rho r}$$

It is an imaginary function and is odd as a result. These will affect p(r) because if the Fourier

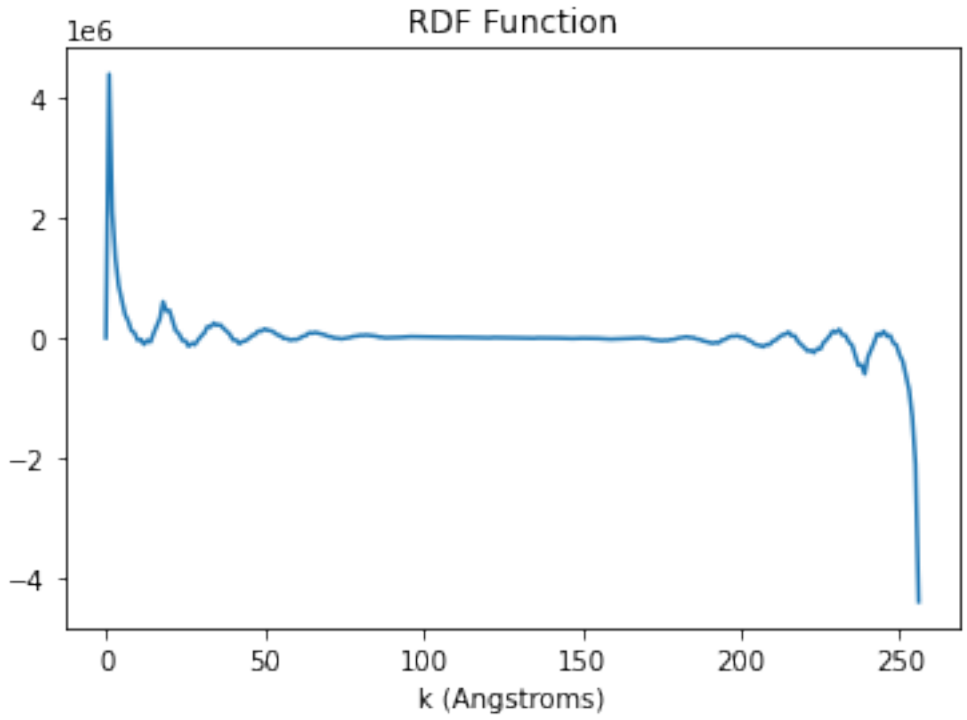
transform is imaginary and odd, so will p(r). This is what I expect.

6 PART 3

[3]:
```
def RDFcalc(S, dk, rho):
    k = np.arange(-len(YanData)*dk, len(YanData)*dk, 0.1195330739)
    p = np.fft.ifft((1/(2*1j*np.pi*massRho*r))*k[:257]*(Sk-1))
    gn = (1 + (1/(2*np.pi))*p)
    rn = (k[:257]*(S-1))/((np.fft.fft(p))*(2*1j*np.pi*rho))
    return gn
    return rn
plt.plot(RDFcalc(Sk, dk, massRho))
plt.title("RDF Function")
plt.xlabel("k (Angstroms)")
```

```
C:\Users\chris\anaconda3\lib\site-packages\numpy\core\_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[3]:
```
Text(0.5, 0, 'k (Angstroms)')
```

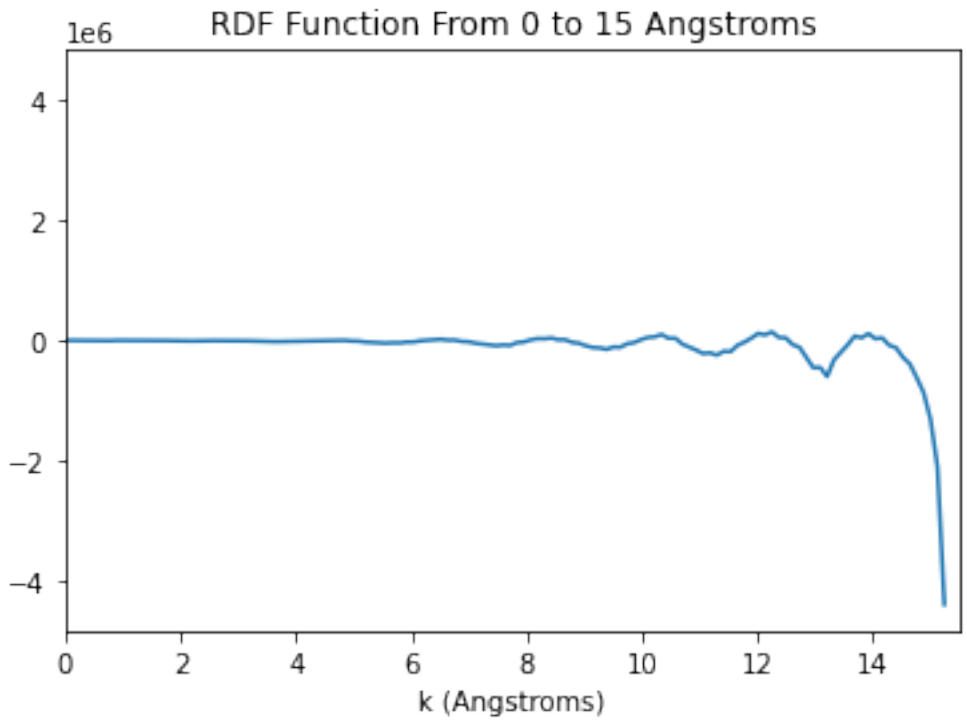


7 PART 4

[4]:
```
plt.plot(k[:257],RDFcalc(Sk, dk, massRho))
plt.xlim(0, 15.5)
plt.title("RDF Function From 0 to 15 Angstroms")
plt.xlabel("k (Angstroms)")
```

```
C:\Users\chris\anaconda3\lib\site-packages\numpy\core\_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[4]:
```
Text(0.5, 0, 'k (Angstroms)')
```



We can only trust it up until around an x value of 10 since that satisfies the limit that g(r) goes to 1.

8 PART 5

[6]:
```
p = np.fft.ifft((1/(2*1j*np.pi*massRho*r))*k[:257]*(Sk-1))
R = np.real(np.mean((k[:257]*(Sk-1))/((np.fft.fft(p))*(2*1j*np.pi*massRho))))
R
```

[6]:
```
1.7816313174028957e-08
```

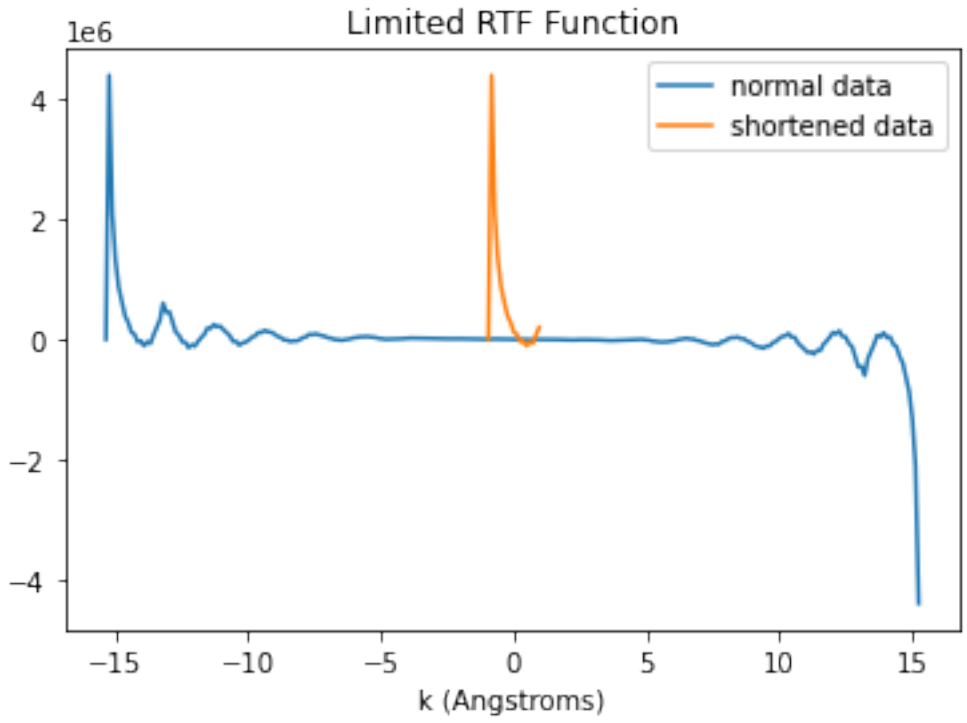
We can rearrange our expression for p(r) to find the average R. The average R is only accurate assuming there is minimal spacing between molecules.

9 PART 6

[44]:
```
plt.plot(k[:257],RDFcalc(Sk, dk, massRho), label = "normal data")

k2 = np.arange(-len(YanData)*dk/16, len(YanData)*dk/16, 0.1195330739)

plt.plot(k2[:257], RDFcalc(Sk, dk, massRho)[:17], label = "shortened data")
plt.title("Limited RTF Function")
plt.xlabel("k (Angstroms)")
plt.legend()
```

```
C:\Users\chris\anaconda3\lib\site-packages\numpy\core\_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
C:\Users\chris\anaconda3\lib\site-packages\numpy\core\_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[44]:
```
<matplotlib.legend.Legend at 0x29a2d644910>
```



It seems that using a k size of kmax/16 will be the point at which the last peak disappears.

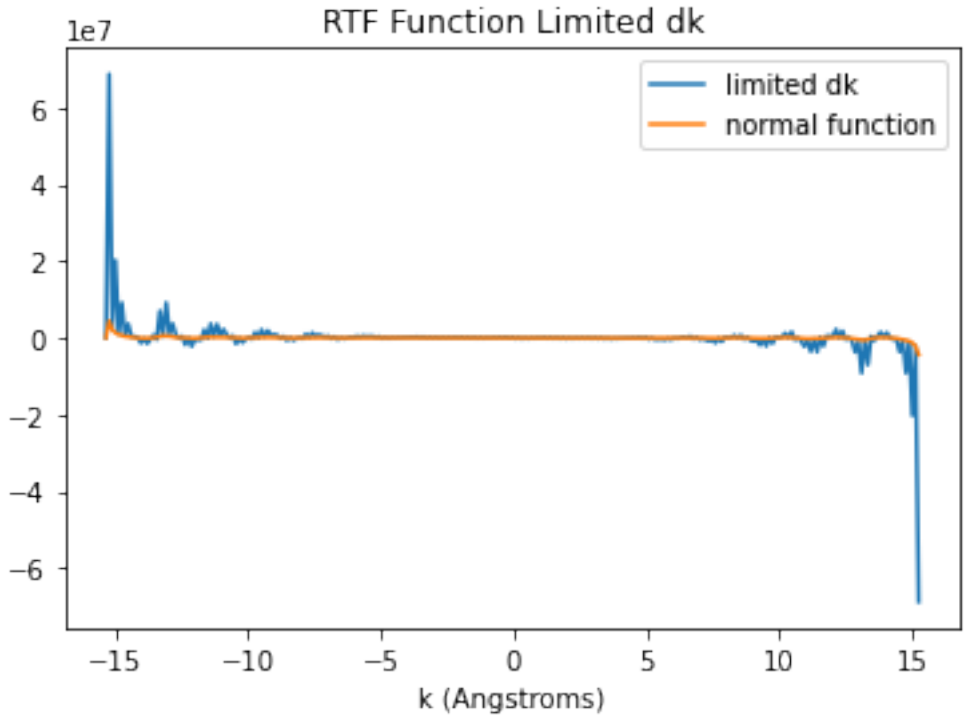
Shortening this range allows for particles to not be detected properly as the wavenumber is too small to detect other molecules within larger ranges.

10 PART 7

[13]:
```
plt.plot(k[:257],RDFcalc(Sk, 1, massRho), label = "limited dk")
plt.plot(k[:257],RDFcalc(Sk, dk, massRho), label = "normal function")
plt.title("RTF Function Limited dk")
plt.xlabel("k (Angstroms)")
plt.legend()
```

```
C:\Users\chris\anaconda3\lib\site-packages\numpy\core\_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
C:\Users\chris\anaconda3\lib\site-packages\numpy\core\_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[13]: `<matplotlib.legend.Legend at 0x29a2bec9d90>`



The first two peaks disappear around dk = 1. This is because the sampling data is too large to capture anything important to the plots (less data points gives more of a chance to miss peaks).

[ ]: