fourier_antennas_1

September 21, 2018

1 IOR 1: Fourier transforms and antennas 1: the basics

In the lectures we looked at the power pattern of n (individually isotropic) antennas, equally spaced along a line and connected to a summing point with equal lengths of cable. For a point source overhead (i.e., at the 'zenith'), the sum of the signals from these antennas will be greater than from any single antenna (which is usually the reason for going to the bother!). However, for a source an angle $\theta > 0$ from the zenith the signal will be reduced. This is because there will be a small delay between the signal reaching each antenna, so the contributions to the sum will no longer be in phase.

If the antennas are a distance a apart, and the signal has a wavelength λ, the phase difference between signals from adjacent antennas is

$$\phi = \frac{2\pi}{\lambda} a \sin\theta \simeq kas \qquad (\text{where } s = \sin\theta).$$

$k = 2\pi/\lambda$ as usual. When we sum n signals, each of amplitude E_0, we will get a resultant that depends on the direction to the source, θ, because of this phase lag, ϕ:

$$E_{\text{tot}} = \sum_{q=1}^{n} E_0 \exp\left[i(q-1)\phi\right].$$

The power response of the array is $P \propto |E_{\text{tot}}|^2$, and in the lectures we showed that the expression for the power pattern reduces to

$$P(s) \propto \left(\frac{\sin knas/2}{\sin kas/2}\right)^2.$$

(To a mathematician this is the square of a *Dirichlet kernel*.) Power patterns are normalised so that their maximum value is unity.

1.0.1 Visualising the power pattern

Let's plot that for a wavelength of 1 m, and antenna separation of 3 m and $n = 5$:

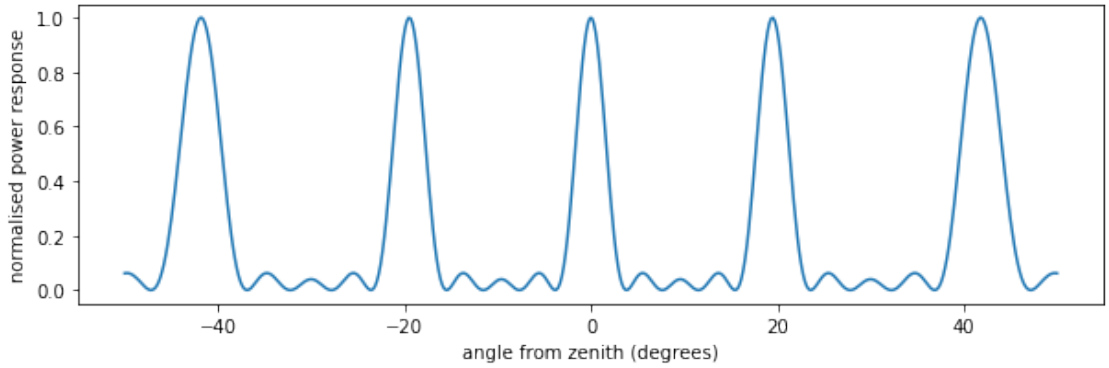
```
In [1]: %matplotlib inline

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

```
        k = 2 * np.pi / 1 # wavenumber: 2 pi / lambda
        a = 3 # antenna separation
        n = 5 # number of antennas
```

```
In [2]: def P(s): # compute the power pattern
            np.seterr(divide='ignore') # a technicality: we'll need this to treat the division
            P = ((np.sin(n * k * a * s / 2) )/np.sin( k * a * s / 2) )**2
            P[np.isnan(P)] = n**2  # set the correct limiting form when s = 0
            return P / n**2 #return the power pattern, normalised to 1 by diving by the max va
```

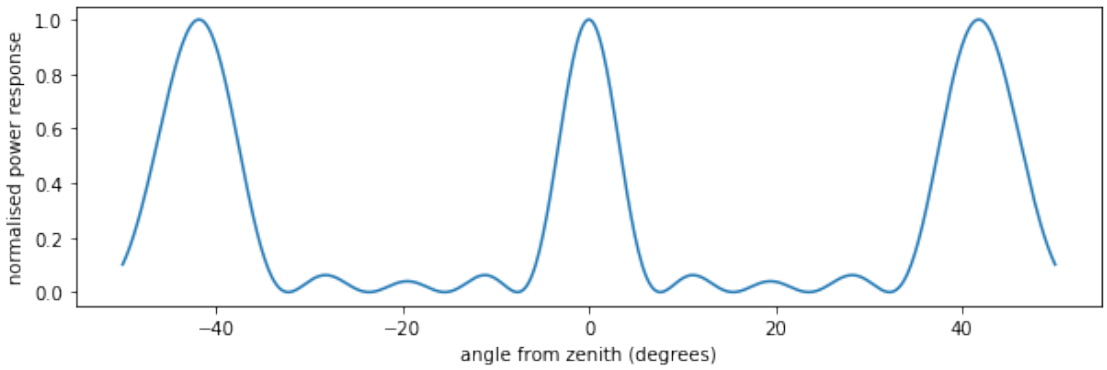
```
In [3]: theta = np.linspace(-50,50,500) # generate an array of angle values +/- 50 degrees fro
        s = np.sin(theta / 180 * np.pi)
```

```
In [4]: plt.rcParams['figure.figsize'] = [10,3]
        plt.plot(theta,P(s))
        plt.xlabel('angle from zenith (degrees)')
        plt.ylabel('normalised power response');
```



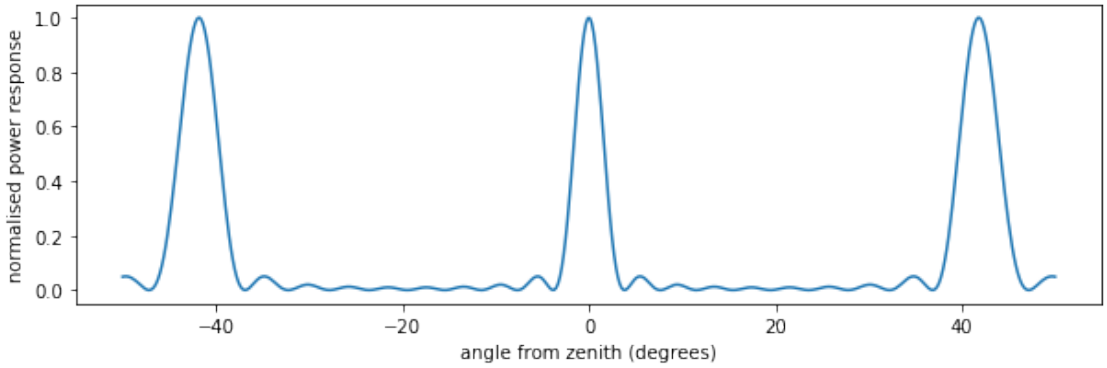
We see that the antenna array has a maximum response at the zenith (good!) but that it also has this maximum response at other angles. These are 'grating responses' which occur when the path difference between antennas is an integer number of wavelengths. If we put the antennas closer, they become fewer:

```
In [5]: a = 1.5 # twice as close
        plt.plot(theta,P(s))
        plt.xlabel('angle from zenith (degrees)')
        plt.ylabel('normalised power response');
```



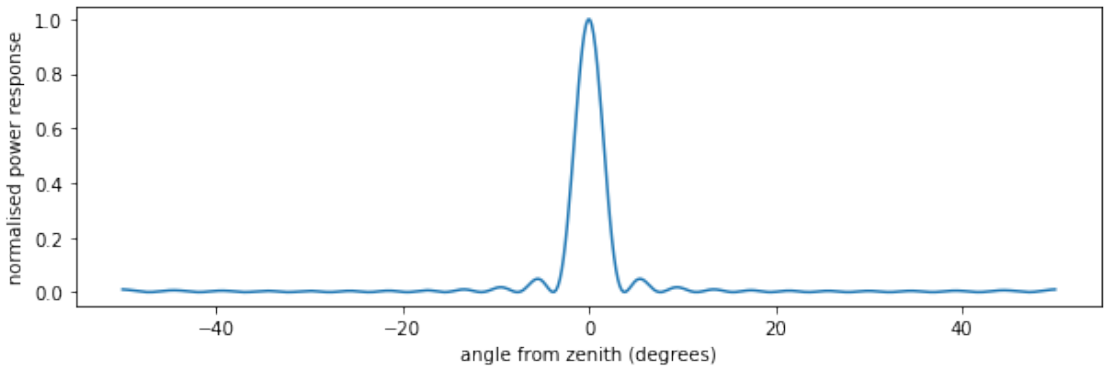
but note that the central maximum is now wider. Let's increase the number of antennas to 10, restoring the original overall length of the array:

```
In [6]: a = 1.5 # twice as close
        n = 10 # but twice as many
        plt.plot(theta,P(s))
        plt.xlabel('angle from zenith (degrees)')
        plt.ylabel('normalised power response');
```



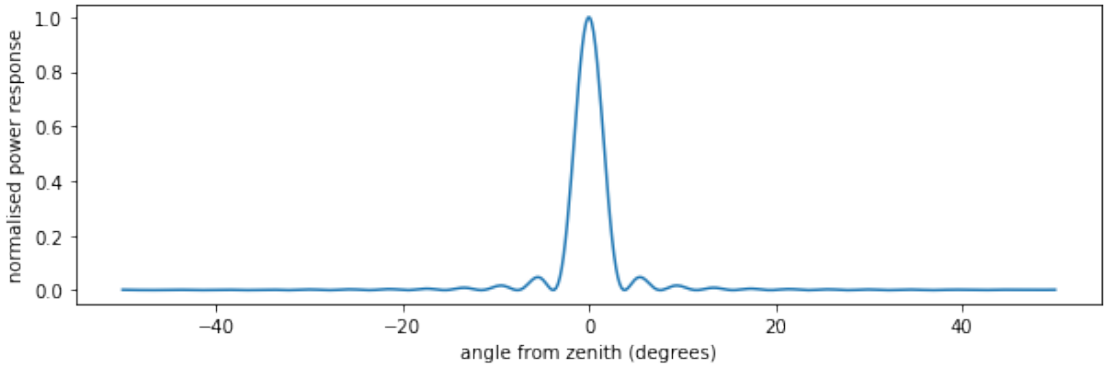
and so we are back to the original central beamwidth. Now separate them by just one wavelength, but have 15 antennas. This suppresses the grating responses and maintains the original beamwidth:

```
In [7]: a = 1 # three times as close
        n = 15 # but three times as many
        plt.plot(theta,P(s))
        plt.xlabel('angle from zenith (degrees)')
        plt.ylabel('normalised power response');
```



It's interesting to see what happens if we continue this: put the antennas 0.5 wavelengths apart and have 30 of them:

```
In [8]: a = 0.5
        n = 30
        plt.plot(theta,P(s))
        plt.xlabel('angle from zenith (degrees)')
        plt.ylabel('normalised power response');
```



There's basically no change in the pattern! We are now sampling the field so finely that there are no grating responses and the resolution is determined by the overall extent of the array. We have reached the continuum limit, and the array is now acting as a single antenna with an aperture width of na = constant.

1.0.2 Using Fourier transforms

In this limit we can replace the sum with an integral. Our original expression for our total received signal

$$E_{\text{tot}}(s) \propto \sum_{q=1}^{n} \exp\left[i(q-1)kas\right]$$

becomes

$$E_{\text{tot}}(s) \propto \int_{-\infty}^{\infty} A(x) \exp(ikxs)\, \mathrm{d}x,$$

where $A(x)$ is the 'aperture function' determining the relative contribution of the antenna at x. So the total signal , as a function of direction on the sky s, is simply proportional to the Fourier transform of this aperture function, and our power pattern is

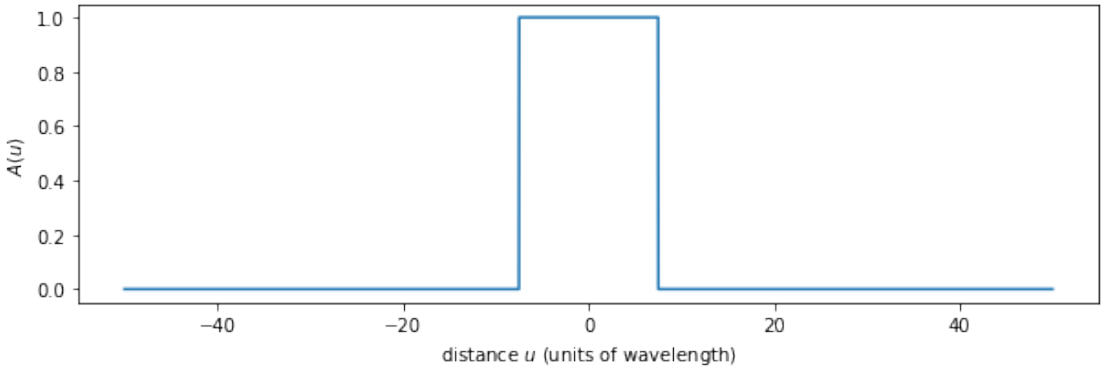
$$P(s) \propto |E_{\text{tot}}(s)|^2.$$

Note there are several ways to define and normalise Fourier transforms. If we measure distances on the ground in units of the wavelength, so that $u = x/\lambda$, our definition corresponds to

$$P(s) \propto \left| \int_{-\infty}^{\infty} A(u) \exp(2\pi i u s)\, \mathrm{d}u \right|^2.$$

So let's show this works for our example. We chose a total aperture that was 15 wavelengths across, and uniform within that. The corresponding aperture function looks like this:

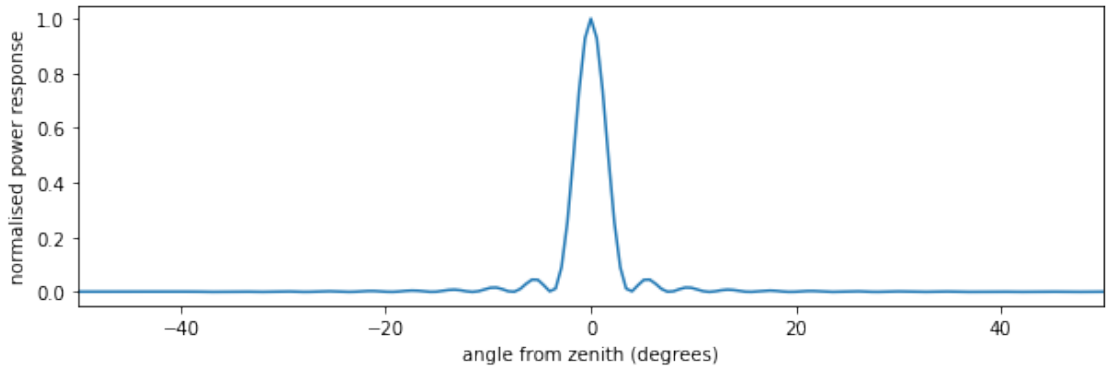
```
In [9]: u = np.arange(-50,50,0.01) # choose a range of u values (start, stop, step)

        A = np.piecewise(u, [np.abs(u) < 15/2, np.abs(u) >= 15/2], [1, 0]) # a useful function
        plt.plot(u,A)
        plt.xlabel('distance $u$ (units of wavelength)')
        plt.ylabel('$A(u)$');
```



We can now compute the power pattern using Fourier transforms. This is straightforward in NumPy:

```
In [10]: E = np.fft.fftshift( np.fft.fft(A) ) # do the fft of A and 'shift' the result so that
         P = np.abs(E)**2
         P = P / P.max() # the normalised power pattern

         s = np.fft.fftshift(np.fft.fftfreq(E.size, 0.01)) # generate the s axis values
         np.seterr(invalid = 'ignore') # our result is only valid for abs(s)<1 so ignore any e
         theta = np.arcsin(s)/np.pi*180 # turn into degrees for comparison with the results ab
```

```
np.seterr(all = 'print') #restore error reporting
plt.plot(theta,P)
plt.xlabel('angle from zenith (degrees)')
plt.ylabel('normalised power response')
plt.xlim(-50, 50);
```

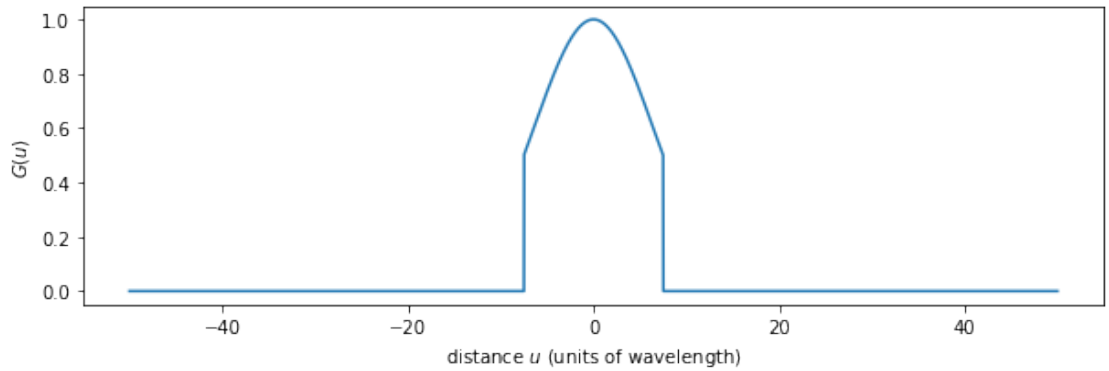


This appears identical to the power pattern we computed from the discrete antenna array, demonstrating that the Fourier transform approach is equivalent. With more complicated aperture functions it is almost always easier to use Fourier transforms to compute the power pattern.

Notice those little wiggly sidelobes on each side of the main beam. These are generated by the two sharp edges in the aperture function. We can make them smaller by smoothing those corners.

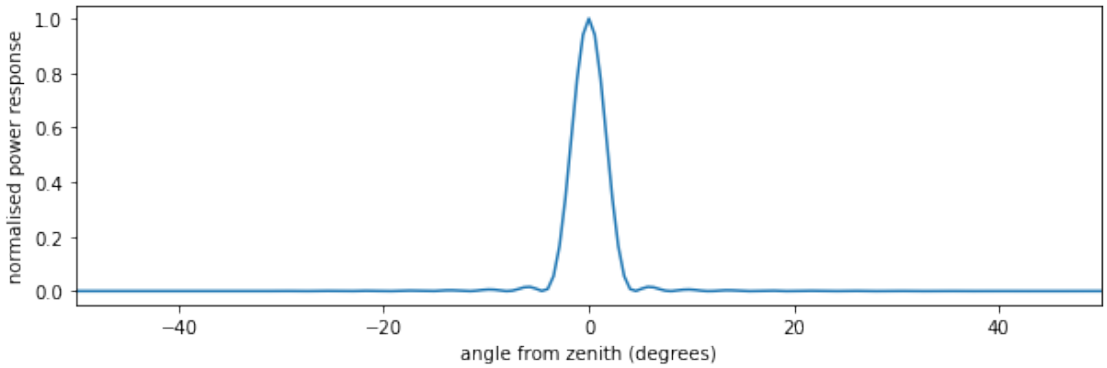
Let's define a gaussian aperture function with the same width at 'half height', and truncate it at that point:

```
In [11]: width = 15 / 2 / (np.log(2))**0.5
         G = np.exp(-(u/width)**2)
         G = G*A # truncate to the width of A
         plt.plot(u,G)
         plt.xlabel('distance $u$ (units of wavelength)')
         plt.ylabel('$G(u)$');
```



.... and plot the corresponding beam pattern:

```
In [12]: E = np.fft.fftshift( np.fft.fft(G) ) # do the fft of A and 'shift' the result so that
         P = np.abs(E)**2
         P = P / P.max() # the normalised power pattern

         s = np.fft.fftshift(np.fft.fftfreq(E.size, 0.01)) # generate the s axis values
         np.seterr(invalid = 'ignore') # our result is only valis for abs(s)<1 so ignore error
         theta = np.arcsin(s) / np.pi*180 # turn into degrees
         np.seterr(all = 'print')
         plt.plot(theta,P)
         plt.xlabel('angle from zenith (degrees)')
         plt.ylabel('normalised power response')
         plt.xlim(-50, 50);
```



The sidelobes are lower, but the beam is a little wider -- it's a trade-off. Try narrowing the gaussian in the above example, so that the sharp edge at the base of the aperture function is less pronounced. You will find that the beam pattern is wider, but the sidelobes are greatly reduced. Remember the Fourier transform of a gaussian is another gaussian, so without any truncation to the gaussian aperture function the beam would be gaussian.

Now we know the basics we can proceed to episode 2: tricks and tips.

Graham Woan 2018