Thinking in Waves

September 28, 2023

Thinking in Waves

Learning Outcomes

Following this lecture, students will be able to:

Understand how a function can be represented as the sum of an infinite number of waves, each with different wavelengths/wavenumbers.

Use Python to define a function, transform it into wavenumber space, isolate individual wave components, transform the result back to physical space, and visualize the result.

Introduction

The atmosphere contains many wave types. We're all familiar with Rossby waves, which manifest in the trough-ridge pattern we see in the middle to upper troposphere. We're also likely familiar with other wave types such as gravity waves, which we can see as high-frequency waves eminating from thunderstorms near the tropopause and/or near the surface (to name but one example). There are also other waves, such as sound/acoustic waves, present in the atmosphere as well.

Let's take a step back for a moment, however. We are familiar with cosine and sine functions representing waves; e.g.,

$$A\cos(Bx + C) \tag{1}$$

is a cosine wave with amplitude A, period 2 /B, and phase shift C.

Euler's formula demonstrates how complex-valued exponential functions represent wave-like solutions:

$$e^{i\theta} = \cos(\theta) + i\sin(\theta)e^{-i\theta} = \cos(\theta) - i\sin(\theta) \tag{2}$$

where is any quantity (e.g., kx or t).

The real-valued component of $e^{i\theta}$, i.e., $Re[e^{i\theta}]$, is simply $\cos(\theta)$ - a cosine wave with amplitude 1, period of 2 , and zero phase shift.

Fourier series allow us to express any periodic function - such as global weather data in the zonal direction - in terms of an infinite sum of waves with varying frequencies (time-domain data) or wavenumbers (spatial-domain data; recalling that wavelength is inversely proportional to wavenumber):

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{in\omega_0 t} \tag{3}$$

where n is a wavenumber, c_n are complex-valued coefficients that define the wave amplitudes (with one for each wavenumber), $\omega_0 = \frac{2\pi}{T}$ is the fundamental frequency (where T is the wave's period), and $n\omega_0 = \omega$, defining the wave's frequency.

This Fourier series representation is valid for time-domain data. The analogous version for spatial-domain data is given by:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{ink_0 x} \tag{4}$$

where $k_0 = \frac{2\pi}{L}$ is the fundamental wavenumber (where L is the wave's wavelength) and $nk_0 = k$, defining the wavenumber.

The complex coefficients c_n (assuming time-domain data again) are given by:

$$c_n = \frac{1}{T} \int_T f(t) e^{-i\omega t} dt \equiv F(\omega) \tag{5}$$

where the integration is performed over one wave period. The analogous expression for spatial-domain data involves integrating over one wavelength, with $c_n \equiv F(k)$:

$$c_n = \frac{1}{L} \int_L f(x) e^{-ikx} x \equiv F(k) \tag{6}$$

If our function is continuous rather than discrete, forward and inverse Fourier transforms allow us to transform any function from physical dimensions (time, space) to wave dimensions (frequency, wavenumber) and vice versa.

The forward and inverse Fourier transforms for time- and frequency-domain data are given by:

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} d\omega \tag{7}$$

The first equation represents the forward Fourier transform, transforming a function f from the time domain (t) to a function F in the frequency domain ( ). The second equation represents the inverse Fourier transform, transforming F from the frequency domain back to f in the time domain by integrating it over all frequencies from negative infinity to positive infinity.

The forward and inverse Fourier transforms for spatial- and wavenumber-domain data are given by:

$$F(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-ikx} dx f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(k) e^{ikx} dk \tag{8}$$

The first equation represents the forward Fourier transform, transforming a function from the spatial domain (x) to a function in the wavenumber domain (k; recalling again that wavenumber is inversely proportional to wavelength). The second equation represents the inverse Fourier transform, transforming the function from the wavenumber domain back to the spatial domain by integrating it over all wavenumbers from negative infinity to positive infinity.

Why do we care about this? For one, spectral models such as the ECMWF's Integrated Forecasting System rely on Fourier transforms to represent atmospheric variability in the zonal (east-west) direction. For another, and more relevant to our class (which focuses on grid-based models), thinking in waves allows us to better understand the properties of the numerical methods used in numerical weather prediction models.

Specifically, we can use the forward Fourier transform to transform any discrete temporally or spatially varying function into wave space defined by frequency or wavenumber. We can then operate on the transformed function to, for example, retain only certain wavenumbers or frequencies of interest. Finally, we can use the inverse Fourier transform to return to physical space. We demonstrate all of this below using a Gaussian wave.

Getting Started

Let's see this in action. We're first going to load our modules. We only need two: numpy for our numerical operations, including Fourier transforms, and matplotlib's pyplot routine for plotting the results. We also add a configuration directive to tell Python to not use its jedi package for tab-autocompletion in iPython (if we wanted to use that when writing our code).

[1]:
```
%config Completer.use_jedi = False
import numpy as np
import matplotlib.pyplot as plt
```

Function Definition

We're now going to define a few functions for later use. Python requires these be defined before you use them, so that's why you find them here. Our three functions are as follows:

isolate: Performs the forward Fourier transform on an array, zeros out the amplitudes of all wavenumbers except for one specified wavenumber, and returns the inverse Fourier transform.

transform: Performs the forward Fourier transform on an array, zeros out the amplitudes of all wavenumbers at and above the specified wavenumber, and returns the inverse Fourier transform. Thus, transform returns an array containing data over a range of wavenumbers, whereas isolate returns an array containing data only for one wavenumber.

power: Performs the forward Fourier transform on an array and returns the square of its absolute value (which defines the transformed array's power spectrum).

The functions defined here use the real-valued versions of the forward and inverse Fourier transforms. We are dealing exclusively with real-valued data in this example, so using this version reduces the calculation time (over the full complex-valued versions) without impacting the results.

[2]:
```
# ==========================
# FUNCTION: isolate(array, wn)
def isolate(array, wn):
```

```
    fwd = np.fft.rfft(array)
    fwd[0:wn] = 0.0
    fwd[wn+1:] = 0.0
    return np.fft.irfft(fwd)
# ==========================

# ==========================
# FUNCTION: transform(array, wn)
def transform(array, wn):
    fwd = np.fft.rfft(array)
    fwd[wn:] = 0.0
    return np.fft.irfft(fwd)
# ==========================

# ==========================
# FUNCTION: power(array)
def power(array):
    return np.abs(np.fft.rfft(array))**2
# ==========================
```

Initialization

We begin the main part of our program by defining some constants, some of which we will use momentarily in defining our Gaussian wave.

The variables points and dx set up our model grid. The former defines the total number of grid points whereas the latter defines the horizontal spacing between them. The spacing between points is largely irrelevant in this example since we're just looking at a single snapshot in time; it becomes important when we integrate a model forward in time, however.

The variables a, b, and c are parameters to the Gaussian wave: its peak amplitude a, the grid point b at which the peak amplitude is found, and a width parameter c (equal to one standard deviation).

The last statement in this block sets up a 1-D array with points number of points, with values from zero to points. These define the indices for our grid.

[3]:
```
# ==========================
# PROGRAM INITIALIZATION
# Define constants for later use.
points = 100
dx = 10000.
a = 100.0
b = 50
c = 4.0

# Define an array with the indices
# of our model's grid points.
x = np.arange(points)
```

```
# ==========================
```

Wave Definition

A Gaussian wave has the general equation form:

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}} \tag{9}$$

where a, b, and c are as defined before.

Here, we define a zero-valued array of length points to store our wave in, then we define the wave itself using the parameters defined in the previous code block. The commented-out option in the code below would allow us to add some random noise to the wave. If applied, this would perturb the wave's amplitude at each grid point by a random amount ranging from zero to the wave's peak amplitude a (noting that NumPy's random.randint function treats the specified upper bound as exclusive; i.e., a specified value of a+1 will generate data from 0–>a, not including a+1).

[4]:
```
# ==========================
# Define the wave.
# First, define an array of size
# (points) in which to store the
# model solution.
answer1 = np.zeros(points)

# Initialize the model: define the
# Gaussian-wave initial condition.
answer1 = a * np.exp(-1.0*(x-b)**2/(2*c**2))

# OPTIONAL: add some random noise
# to this wave (in the range 0-->a).
#answer1 = answer1 + np.random.randint((a+1), size=100)
# ==========================
```

Example 1: Isolating Wave Components

In this first example, we call our previously defined function isolate to return only the wavenumber 0, 1, 2, 3, and 4 components of the full Gaussian wave. The returned waves depict zero, one, two, three, and four full wavelengths across the model domain, respectively.

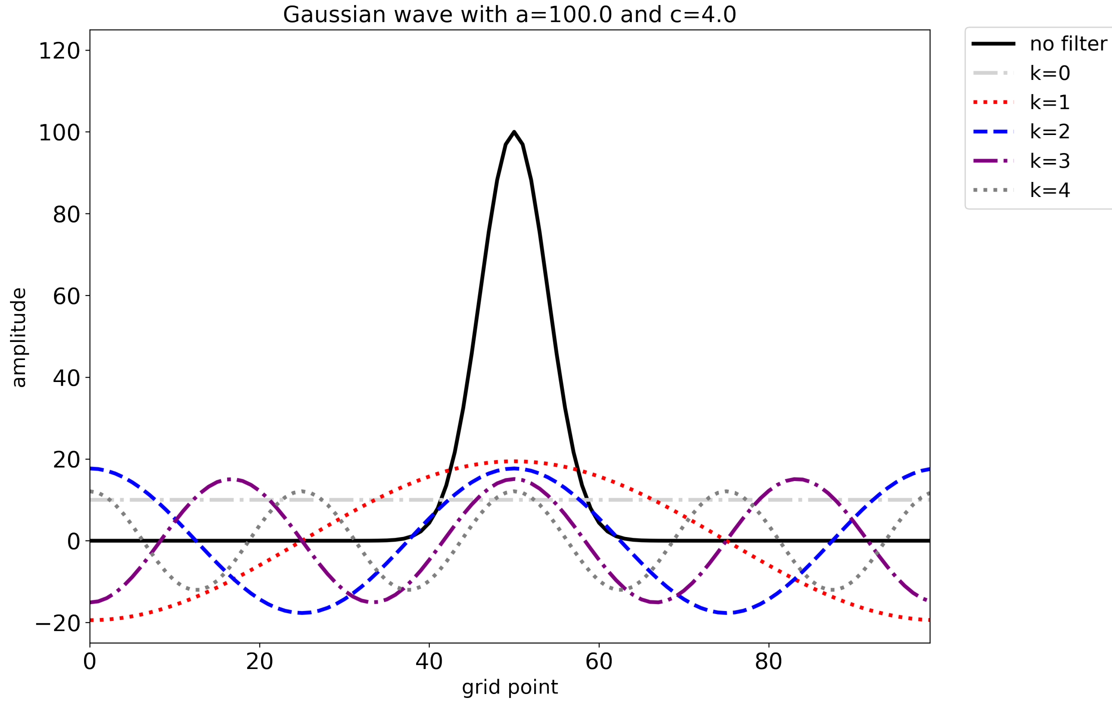
Once these wave components have been obtained, we plot them all on a single plot. To do so, we first set up our figure instance, then we specify an appropriate set of axes (here, 0->99 points in x, amplitudes of -25 to 125 in y). Next, we plot the full wave itself (stored in answer1), then its wavenumber-0, 1, 2, 3, and 4 components.

The remaining lines in this code block tweak the figure: where to have axis tick marks (and what font to give them), what to label the two axes, what to title the plot, how to format the legend, and a statement to show the resulting figure.

[5]:
```
# ==========================
# Call the isolate function.
w0 = isolate(answer1, 0)
w1 = isolate(answer1, 1)
w2 = isolate(answer1, 2)
w3 = isolate(answer1, 3)
w4 = isolate(answer1, 4)
# ==========================



# ==========================
# Plot the five isolated waves.
fig1 = plt.figure(figsize=(12,9), dpi=300)
ax = plt.axes()
ax.axis([0, 99, -25, 125])
ax.plot(x, answer1, linestyle='solid', color='black', linewidth=3, label='no␣
↪filter')
ax.plot(x, w0, linestyle='dashdot', color='lightgray', linewidth=3, label='k=0')
ax.plot(x, w1, linestyle='dotted', color='red', linewidth=3, label='k=1')
ax.plot(x, w2, linestyle='dashed', color='blue', linewidth=3, label='k=2')
ax.plot(x, w3, linestyle='dashdot', color='purple', linewidth=3, label='k=3')
ax.plot(x, w4, linestyle='dotted', color='grey', linewidth=3, label='k=4')
ax.tick_params(axis='both', labelsize=18)
ax.set_xlabel('grid point', fontsize=16)
ax.set_ylabel('amplitude', fontsize=16)
ax.set_title('Gaussian wave with a=' + str(a) + ' and c=' + str(c), fontsize=18)
legend = plt.legend(fontsize=16, bbox_to_anchor=(1.03, 1.02), loc='upper left')
plt.show()
# ==========================
```



The full wave has a peak amplitude of 100 (a) at grid point 50 (b). It is about 20 grid points wide; smaller values of c would result in even narrower waves whereas larger values of c would result in wider waves.

As expected, wavenumber 0 (i.e., no trough or ridge) is a horizontal line; in this example, it has an amplitude of about 10. Wavenumber 1 is a single trough-ridge over the domain. It has a peak amplitude of about 20, with the ridge crest at grid point 50. The ridge crest being located at grid point 50 means that this wavenumber has the same phase as does the full Gaussian wave; a ridge crest at a different grid point would indicate that the wave component has a shifted phase from that of the full Gaussian wave). Wavenumbers 2, 3, and 4 contain two, three, and four troughs and ridges over the domain, respectively, with a ridge crest centered at grid point 50 for each wave. Starting after wavenumber 1, each wavenumber's peak amplitude is somewhat smaller than that of the next-smallest wavenumber (i.e., wavenumber 2's peak amplitude is smaller than wavenumber 1's peak amplitude, etc.).

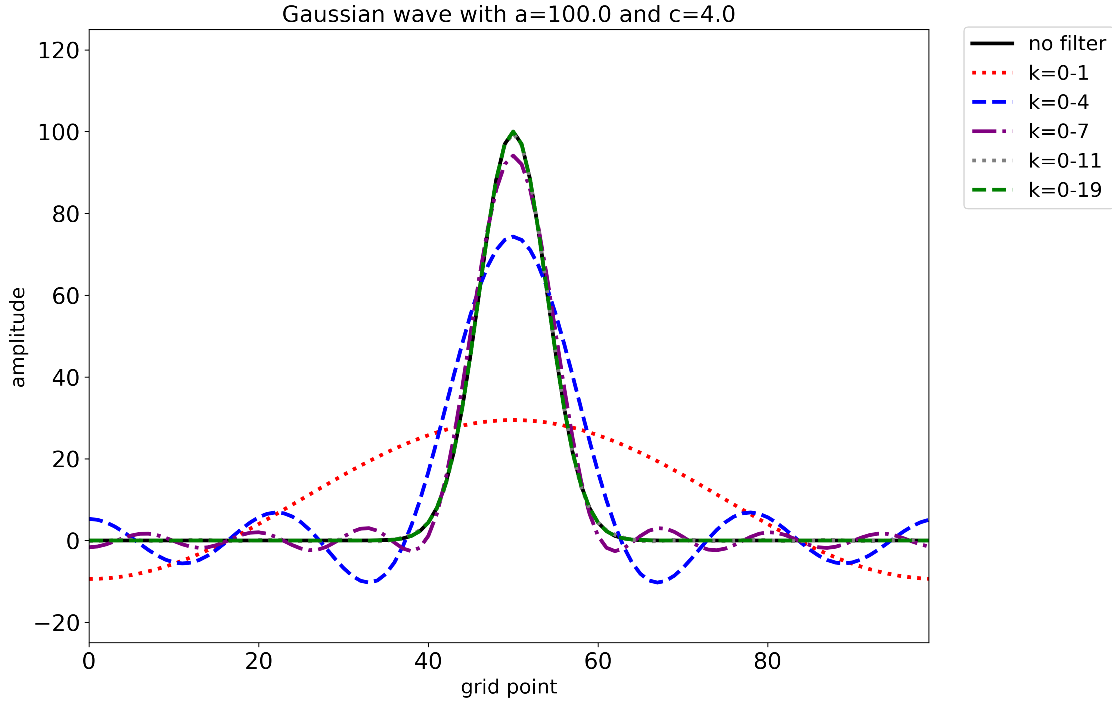
Example 2: Wavenumber Filtering

Now that we've seen what each individual wavenumber (at least over the range of wavenumbers from 0 to 4) looks like, we can move to determining what combinations of them look like. Below, we call our previously defined transform function to return versions of the wave that retain only a limited number of wavenumbers: 0-1, 0-4, 0-7, 0-11, and 0-19. (Note: the function calls specify 2, 5, 8, 12, and 20 rather than 1, 4, 7, 11, and 19 as 2, 5, 8, 12, and 20 are the smallest wavenumbers at which we wish to zero out the wave amplitudes.)

The plotting code is structured very similarly to that in Example 1.

[6]:
```
# ==========================
# Filter the wave at wavenumbers
# at and larger than those specified.
a2 = transform(answer1, 2)
a5 = transform(answer1, 5)
a8 = transform(answer1, 8)
a12 = transform(answer1, 12)
a20 = transform(answer1, 20)
# ==========================

# ==========================
# Plot the four wave approximations.
fig2 = plt.figure(figsize=(12,9), dpi=300)
ax = plt.axes()
ax.axis([0, 99, -25, 125])
ax.plot(x, answer1, linestyle='solid', color='black', linewidth=3, label='no␣
↪filter')
ax.plot(x, a2, linestyle='dotted', color='red', linewidth=3, label='k=0-1')
ax.plot(x, a5, linestyle='dashed', color='blue', linewidth=3, label='k=0-4')
ax.plot(x, a8, linestyle='dashdot', color='purple', linewidth=3, label='k=0-7')
ax.plot(x, a12, linestyle='dotted', color='grey', linewidth=3, label='k=0-11')
ax.plot(x, a20, linestyle='dashed', color='green', linewidth=3, label='k=0-19')
ax.tick_params(axis='both', labelsize=18)
ax.set_xlabel('grid point', fontsize=16)
ax.set_ylabel('amplitude', fontsize=16)
ax.set_title('Gaussian wave with a=' + str(a) + ' and c=' + str(c), fontsize=18)
legend = plt.legend(fontsize=16, bbox_to_anchor=(1.03, 1.02), loc='upper left')
plt.show()
# ==========================
```



Retaining more wavenumbers provides us with an approximation that more closely resembles the full Gaussian wave. The red-dotted line is the combination of the curves for wavenumbers 0 and 1 from the first plot; it looks like wavenumber 1 alone, but with a higher amplitude given by the addition of the wavenumber 0 curve (which had a constant amplitude of about 10). The blue-dashed line is the combination of the curves for wavenumbers 0, 1, 2, 3, and 4 from the first plot. The purple-dashed line, representing the combination of the curves for wavenumbers 0-7, reasonably approximates the full wave. The grey-dotted line, representing the combination of the curves for wavenumbers 0-11, very closely resembles the full wave, and the green-dashed line, representing the combination of the curves for wavenumbers 0-19, is practically identical to the full wave.

Example 3: Power Spectrum Analysis

This example computes and plots the power spectra of each of the truncated waves from Example 2. The power spectrum is defined as the transformed function's squared amplitudes. It provides a measure of the power, or energy, at each wavelength (or inverse wavenumber).

In the code below, we first compute the power spectra for the truncated waves from Example 2 by calling our previously defined power function. We also define the frequencies - or, more precisely, wavelengths since we are dealing with spatially rather than temporally varying data - associated with our wave. This allows us to plot the power spectra as a function of wavelength, which is a bit more intuitive than is wavenumber.

A brief discussion of the np.fft.rfftfreq helper function that is used to define the wave's wavelengths: Let N = answer1.size, the number of model grid points. This function creates an array of N/2 + 1 (if N is even) or (N-1)/2 + 1 (if N is odd) points with equally spaced values between 0 and 0.5. These values are then scaled by the grid spacing (dx) to give an array freqs with units of m^{-1}. In this one-dimensional example, these values are our wavenumbers. The result of 1/freqs - taken later

in the code - provides wavelength with units of m (noting that if we were dealing with spherical data, we would need to compute 2 /freqs rather than 1/freqs). Dividing this result by the grid spacing dx returns the number of wavelengths; i.e., 20 = 20Δx. The resulting values range from 2Δx, the shortest-resolvable wave on the grid, to 100Δx, the longest-resolvable wave on the grid. The wavenumber-0 wave, which has a wavelength of ∞Δx, is not depicted.

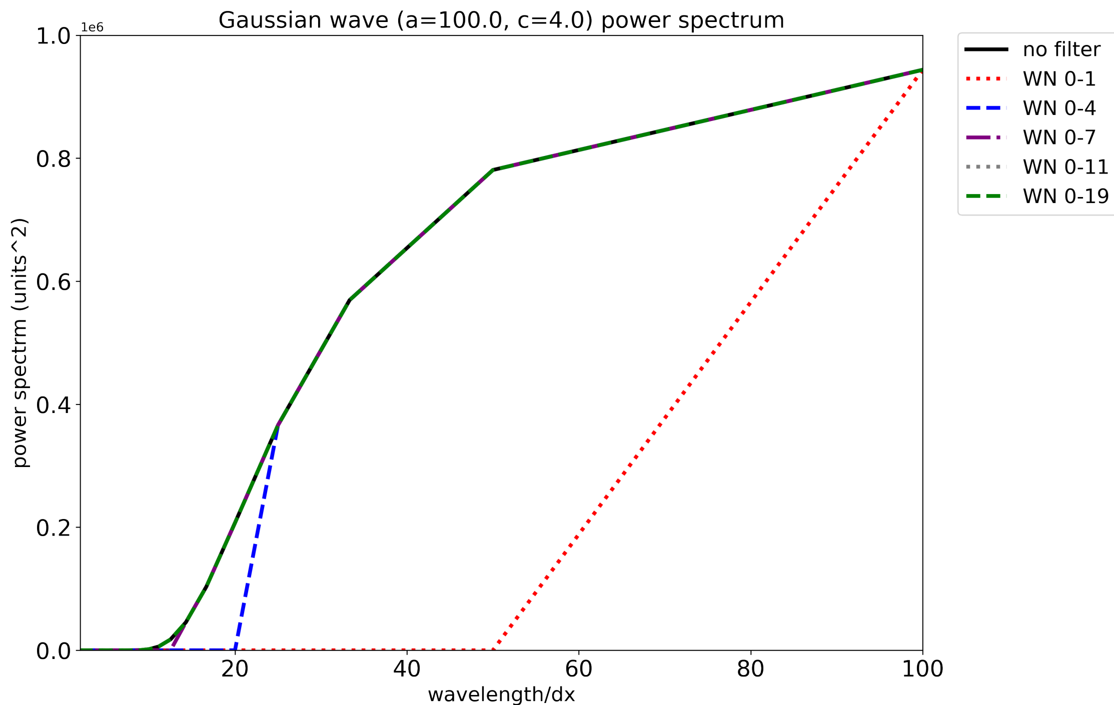
The plotting code is similarly structured to that in Examples 1 and 2. Note that the x-axis ranges from 2 to 100, representing 2Δx (the smallest-resolvable wavelength) to 100Δx (the longest-resolvable wavelength, representing a wave with a single wavelength over our full model domain with 100 points).

[7]:
```
# ==========================
# Compute the power spectra for the
# truncated waves from Example 2.
ps = power(answer1)
ps2 = power(a2)
ps5 = power(a5)
ps8 = power(a8)
ps12 = power(a12)
ps20 = power(a20)
freqs = np.fft.rfftfreq(answer1.size, dx)
# ==========================

# ==========================
# Plot the resulting power spectra.
fig3 = plt.figure(figsize=(12,9), dpi=300)
ax = plt.axes()
ax.axis([2, 100, 0, 1e6])
ax.plot(1/freqs/dx, ps, linestyle='solid', color='black', linewidth=3,␣
 ↪label='no filter')
ax.plot(1/freqs/dx, ps2, linestyle='dotted', color='red', linewidth=3,␣
 ↪label='WN 0-1')
ax.plot(1/freqs/dx, ps5, linestyle='dashed', color='blue', linewidth=3,␣
 ↪label='WN 0-4')
ax.plot(1/freqs/dx, ps8, linestyle='dashdot', color='purple', linewidth=3,␣
 ↪label='WN 0-7')
ax.plot(1/freqs/dx, ps12, linestyle='dotted', color='grey', linewidth=3,␣
 ↪label='WN 0-11')
ax.plot(1/freqs/dx, ps20, linestyle='dashed', color='green', linewidth=3,␣
 ↪label='WN 0-19')
ax.tick_params(axis='both', labelsize=18)
ax.set_xlabel('wavelength/dx', fontsize=16)
ax.set_ylabel('power spectrm (units^2)', fontsize=16)
ax.set_title('Gaussian wave (a=' + str(a) + ', c=' + str(c) + ') power␣
 ↪spectrum', fontsize=18)
legend = plt.legend(fontsize=16, bbox_to_anchor=(1.03, 1.02), loc='upper left')
plt.show()
```

```
# ==========================
```

```
/opt/tljh/user/lib/python3.7/site-packages/ipykernel_launcher.py:18:
RuntimeWarning: divide by zero encountered in true_divide
/opt/tljh/user/lib/python3.7/site-packages/ipykernel_launcher.py:19:
RuntimeWarning: divide by zero encountered in true_divide
/opt/tljh/user/lib/python3.7/site-packages/ipykernel_launcher.py:20:
RuntimeWarning: divide by zero encountered in true_divide
/opt/tljh/user/lib/python3.7/site-packages/ipykernel_launcher.py:21:
RuntimeWarning: divide by zero encountered in true_divide
/opt/tljh/user/lib/python3.7/site-packages/ipykernel_launcher.py:22:
RuntimeWarning: divide by zero encountered in true_divide
/opt/tljh/user/lib/python3.7/site-packages/ipykernel_launcher.py:23:
RuntimeWarning: divide by zero encountered in true_divide
```



Recall from Example 1 that the wavenumber-1 wave had the largest amplitude, with subsequent wavenumbers having smaller amplitudes. That results in what we see here: the greatest power (representing squared amplitude) in the longest wavelength/smallest wavenumber with progressively less power in progressively smaller wavelengths/larger wavenumbers.

The wavenumbers 0-1 curve has a single value at wavelength 100Δx; its next value, at 50Δx (corresponding to a wave with two wavelengths, aka the wavenumber-2 wave), is zero. The wavenumbers 0-4 curve has a similar dropoff between 25Δx (wave with four wavelengths) and 20Δx (wave with five wavelengths), and the wavenumbers 0-7 curve has a similar dropoff between 14.29Δx (wave with seven wavelengths) and 12.5Δx (wave with eight wavelengths).

We'll see later this semester that the properties of the numerical methods used in modern numerical weather prediction models are often different between wavelengths. Plots such as the one above will be helpful for demonstrating this, as they allow us to visualize the power at different wavelengths without having to plot each wave individually.

Try it Yourself

In the meantime, I encourage you to experiment with different initial wave structures - especially different widths - to see how they are associated with different individual wave structures and power spectra. I also encourage you to experiment with adding random noise to the Gaussian wave, whether using the structure given above or something with higher or lower potential amplitudes, to see how this influences the individual wave structures (especially at higher wavenumbers/shorter wavelengths) and associated power spectra. You can do these right from this notebook - just choose Cell -> Run All after making any changes to run the code again. Note that you will likely need to change the y-axis plotting ranges when performing these tests!

You will need to make a copy of this notebook if you wish to experiment with the code here. Unfortunately, the "Duplicate" function on our JupyterHub only has the ability to duplicate the notebook in this shared folder, in which you don't have the ability to create files. There are two workarounds:

Download and Upload the Notebook: From the File menu of this notebook, choose Download -> Notebook (.ipynb). Then, exit the notebook back to the shared_notebooks folder. From there, click on the blue folder icon at upper left, which will take you back to your main directory. Next, click the Upload button in the upper right, then direct the prompt to upload the notebook you just downloaded. You'll then be able to edit this copy of the notebook to your heart's content.

Use the Terminal to Copy the Notebook: Exit this notebook back to the shared_notebooks folder. From there, click on the New button at upper right, then choose Terminal. This will open a terminal in your main JupyterHub folder. From there, use cp to copy the notebook from the shared_notebooks folder to your folder: cp shared_notebooks/Thinking in Waves.ipynb . (keeping the spaces and the trailing period). You can then exit the terminal by typing exit, pressing the Enter key on your keyboard, and then closing the terminal window.