More Introduction to Python

In the Lab 1 notebook forematter, we saw some basics, like variable assignment, basic `print` statements, and some simple arithmetic and functions (like $\exp$ and $\log_{10}$).

String addition

The `+` operator acts to combine two objects into a new one. In Python this acts the way we expect for integers and floating-point numbers.

This operator is/can be defined for non-numeric types of data. This is particularly useful for strings. For example,

In [208…
```
s1 = 'abc' # define a string
s2 = 'def' # define another string
s = s1 + s2 # add them together
print(s) # display the output
```

```
abcdef
```

In this case we see that using `+` on two strings results in a new string which is the contcatenation of the two inputs.

Sequences (tuples and lists)

So far we have seen objects which are either numberic (integers and floating-point numbers) or strings. There are also more generic (and useful) data types which are sequences of objects.

"Tuples" are sequences of objects. Once you create this sequence, it cannot be changed. They can be made up of a mixture of different object types, as long as those objects don't change. They are defined with `()` symbols. You can access a particular element using the square brackets `[]` and specifying an index, starting from 0. Negative indices are allowed; index -1 corresponds to the last element, -2 to the second-to-last, etc.

In [209…
```
x = (0, 1, 'a', 53.2, 'foo') # define a tuple containing items of different
print(x) # print the whole tuple
print(x[0]) # print the first element (indices start at zero)
print(x[2]) # print the third element
print(x[-1]) # print the last element
```

```
(0, 1, 'a', 53.2, 'foo')
0
a
foo
```

A tuple of length 1 can be defined using a comma before the closing parenthesis. A tuple of length zero is defined with `()`.

We can check the length of a sequence with the `len()` function.

In [210…
```
x = (45.3,) # define a tuple of length 1 (notice the comma)
print(x)
print(x[0])
print('length:', len(x))
x = () # define an empty tuple (length 0)
print(x)
print('length:', len(x))
```

```
(45.3,)
45.3
length: 1
()
length: 0
```

"Lists" are also sequences of objects, except they are more flexible. They themselves can be changed, and they can contain elements that can also be changed (like other lists). This flexibility means they're not suitable for some applications, which is why tuples exist. Lists work like tuples, except they are defined by square brackets `[]`. We can also add to the end of a list using the `append()` method, demonstrated below.

In [211…
```
x = [0, 1, 'a', 53.2, 'foo']
print(x)
print(x[2])
print(x[-1])
x.append('Python is cool')
print(x)
print(x[-1])
print('length:', len(x))
```

```
[0, 1, 'a', 53.2, 'foo']
a
foo
[0, 1, 'a', 53.2, 'foo', 'Python is cool']
Python is cool
length: 6
```

Iterating over sequences

We might want to use sequences as places to hold our input and output.

We can use `for` statements to iterate over a sequence; that is perform a set of operations on each element of the sequence.

As an example, let's define a list of strings, and perform an operation.

In [212…
```
slist = ['cars', 'boats', 'airplanes',
         'bicycles', 'skateboards']

for s in slist:
    print(s, 'are so awesome.')
```

```
cars are so awesome.
boats are so awesome.
airplanes are so awesome.
bicycles are so awesome.
skateboards are so awesome.
```

We can also use empty lists to store the output of our operations.

In [213…
```
outlist = []
for s in slist:
    outlist.append(s+' are so awesome.')
print(outlist)
```

```
['cars are so awesome.', 'boats are so awesome.', 'airplanes are so awesom
e.', 'bicycles are so awesome.', 'skateboards are so awesome.']
```

1d arrays of numbers

Fornumerical work we'll generally not use the Python built-in sequence objects (tuples and lists), but rather a more powerful package called `numpy`. numpy arrays can be arbitrary dimension. They can be created by passing them a tuple or list of objects.

In [214…
```
xtuple = (0., 345., 12.2, 5.7)

import numpy as np # load the numpy package, but call it `np`
xarr = np.array(xtuple) # create a numpy array out of the tuple
print(xarr)
```

```
[  0.  345.   12.2   5.7]
```

The "shape" of a numpy array gives the dimensionality, and the size in each dimension. For a 1d array, the shape is a tuple of length 1 that is the size of the array.

In [215…
```
print('length of tuple:', len(xtuple))
print('length of numpy array:', len(xarr))
print('shape of numpy array:', xarr.shape)
```

```
length of tuple: 4
length of numpy array: 4
shape of numpy array: (4,)
```

Note the difference between the length and shape here; the shape is a tuple of length 1, since there is 1 dimension.

2d arrays of numbers

We can define a 2-dimensional array of numbers. To do it directly, we can create a sequence of sequences (tuple of tuples, list of tuples, or list of lists), and pass that to the `numpy.array()` function.

In [216…
```
xinput = [(23., 3.1, -31.),
          (0., 5., 10.),
          (10, -5., -20.),
          (np.pi, 2.*np.pi, 3.*np.pi),
         ]
xarr = np.array(xinput)
print(xarr)
print('length of numpy array:', len(xarr))
print('shape of numpy array:', xarr.shape)
```

```
[[ 23.          3.1        -31.        ]
 [  0.          5.          10.        ]
 [ 10.         -5.         -20.        ]
 [  3.14159265   6.28318531   9.42477796]]
length of numpy array: 4
shape of numpy array: (4, 3)
```

Notice the shape of our 2d array. The shape is a tuple of length 2 (there are 2 dimensions). We had 4 rows and 3 columns when defining our array, so the shape is `(4,3)`. Also notice that the length of a 2d array only gives the size of the first dimension ("axis" in numpy-speak).

Plotting a 1d array

We'll use the plotting package called `matplotlib` to create and manipulate graphical displays of data (figures). There is a handy interface to it, called the `pylab` interface, which we'll use. (However, we'll be using patterns based on the more powerful object-oriented features.)

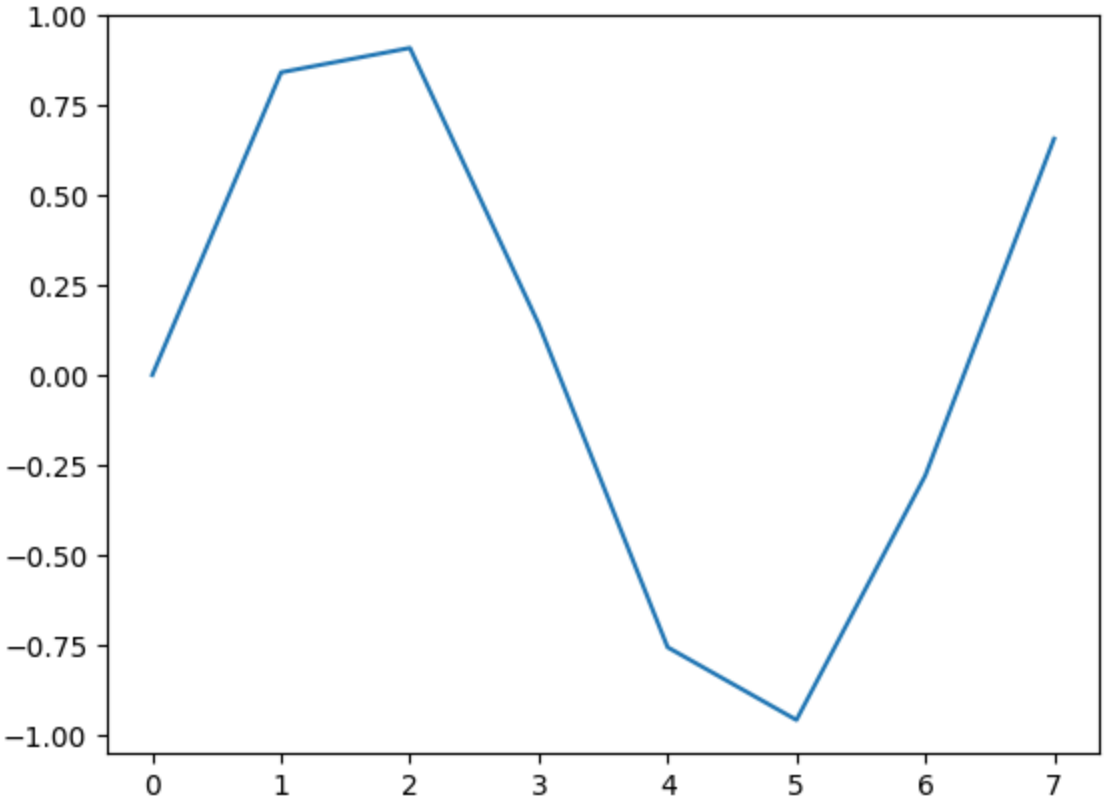
Let's start by plotting a 1d array of data. Here we'll create an index array of x values. Then we'll get y values as a function of x.

Then we'll create a figure, set up some graphing axes, and plot onto those axes. Then we will display our plot, and save it to a PDF.

In [217…
```
x = np.arange(8) # this gives us an array of indices
print(x)

y = np.sin(x) # this is the value of y at every index

# Let's set up some plotting ability
import pylab # This is an interface to the matplotlib plotting library. We c
fig = pylab.figure() # creates a Figure object, onto which we can put differ
ax = fig.add_subplot(111) # we'll create an Axes object, just one instance f
```

```
# Now let's generate a plot of our y data onto the Axes object
ax.plot(y)

# outputting plots
pylab.show() # This will show the figure on the screen.
fig.savefig('test-1d.pdf') # This will save the figure to the filename we sp
```

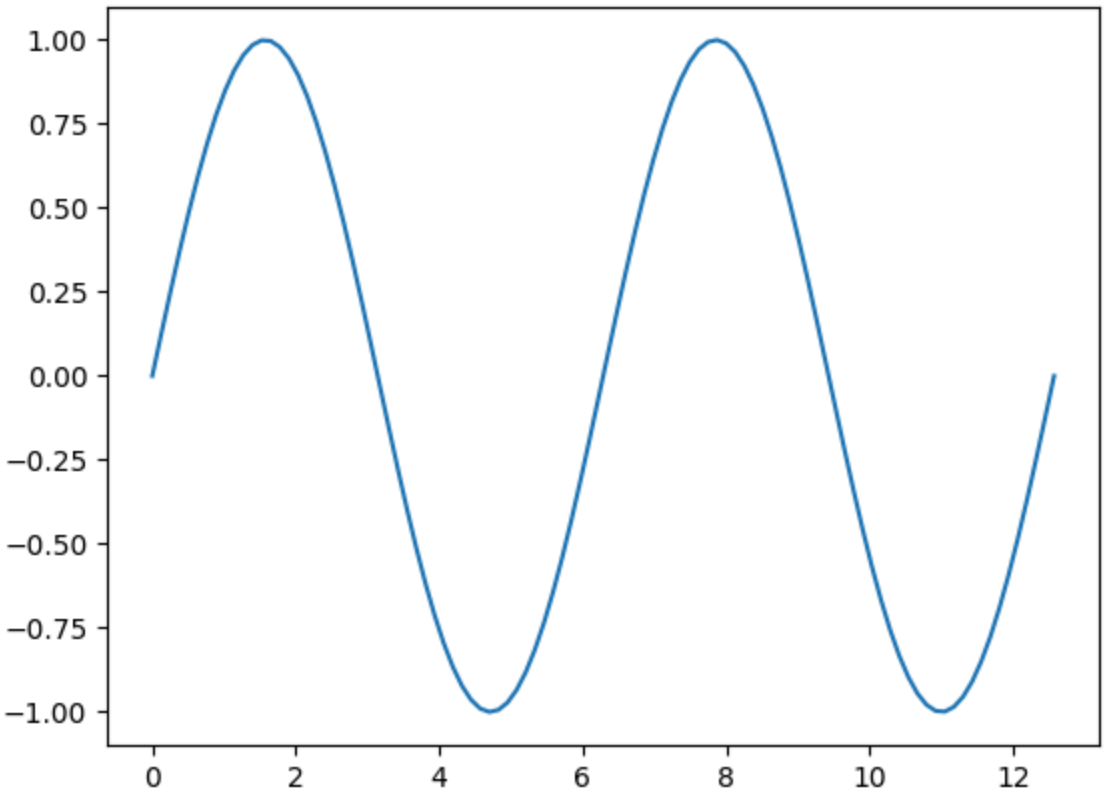
```
[0 1 2 3 4 5 6 7]
```



We could also plot y as a function of x. Let's get a bit more finely sampled in our x values with numpy's `linspace()` function.

In [218…

```
x = np.linspace(0., 4.*np.pi, 100) # linearly spaced points from 0 to 4pi, v
y = np.sin(x)
fig = pylab.figure() # creates a Figure object, onto which we can put differ
ax = fig.add_subplot(111) # we'll create an Axes object, just one instance f
ax.plot(x, y) # plot the y vs. x
pylab.show() # This will show the figure on the screen.
```



Solar Spectroscopy

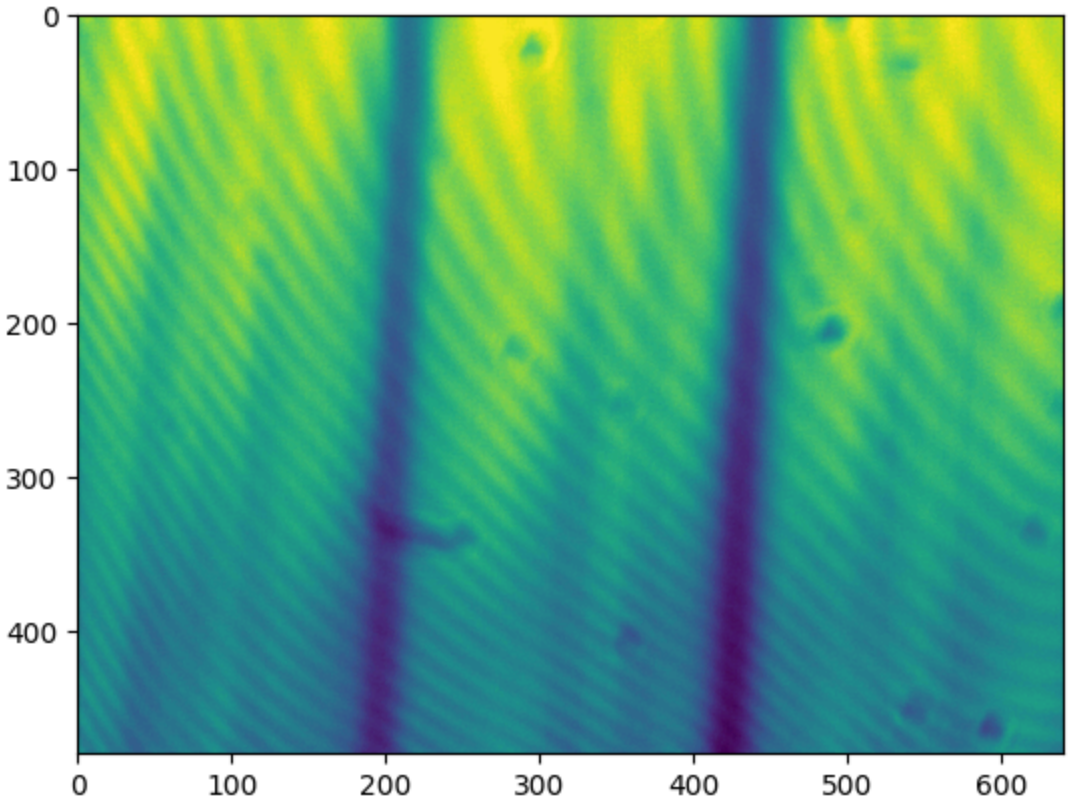
Loading and Displaying the 2d Image Data

The first steps are to load in our data. We'll have to specify where our data are located.

For a first step, let's locate the data directory, specify a sample file and display its image.

In [219…
```
import pylab
```

In [188…
```
data_dir = '/data/2025-Fall/Lab2/Group4/' # EDIT this to reflect correct pat
im_fn = 'll_1.fits'

from astropy.io import fits  # this library provides interaction to FITS fil
im = fits.getdata(data_dir+im_fn) # this loads the image data from our FITS

# Set up a figure and axes
fig = pylab.figure() # creates a Figure object, onto which we can put differ
ax = fig.add_subplot(111) # we'll create an Axes object, just one instance f

# Now let's generate a display of our image data onto the Axes object
ax.imshow(im) # the imshow() command displays 2d arrays of data

pylab.show() # This will show the figure on the screen.
```

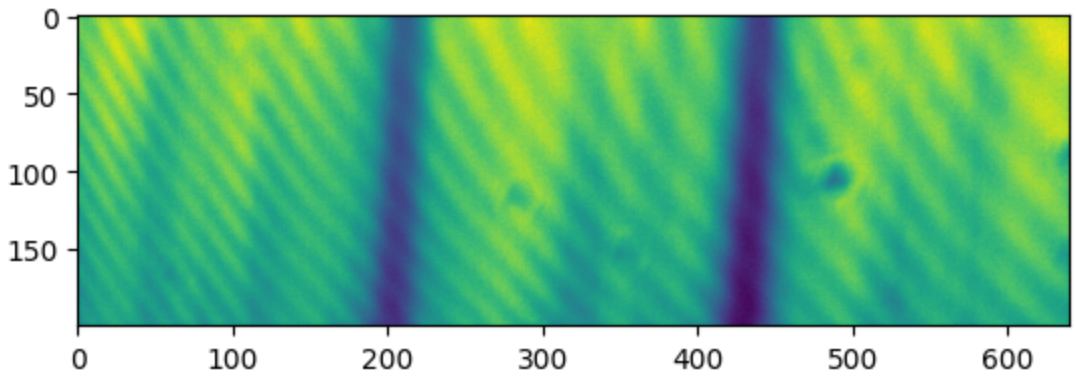


Let's look at the above to see which rows have good data. Since the Sun might not cover the whole slit, or the slit might be blocked on one side, or the image is off the detector, we might not get good data on all rows. (That said, maybe you do!)

Suppose rows 100 to 300 were good. We could create a new array that had only those rows through the process of indexing, `im[100:300, :]`. This says give us rows 100 to 300, and all columns.

In [190…

```
# FIXME  change these rows to match your own data.
im = im[100:300, :]  # use indexing to get a subset of the original array

fig = pylab.figure() # creates a new Figure object
ax = fig.add_subplot(111) # create a new Axes object
ax.imshow(im) # display our new subarray
pylab.show()
```



Reducing the Image to a 1d Spectrum

Now we'll create a 1-dimensional array by collapsing the valid rows. We will use the `sum()` method which adds together the specified data along the specified dimension (a.k.a. "axis" in numpy-speak).

In [191…
```
spectrum = im.sum(axis=0) # sum over all the rows to get the spectrum
print(spectrum.shape) # check the dimensionality of the spectrum
```

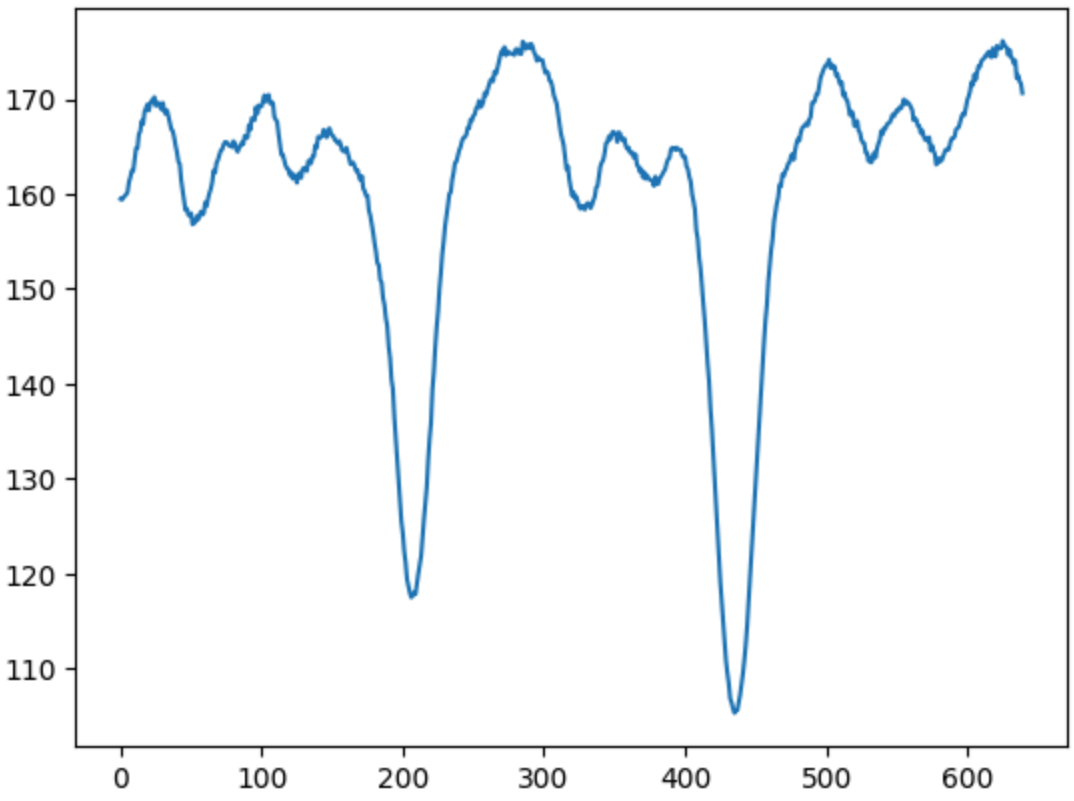
```
(640,)
```

Note that the spectrum is one dimensional, with 640 elements (the x size of the detector).

It is helpful for plotting and fitting to define a one-dimensional array containing the array index. First we find the size of the spectrum image and then make a 1-d index array. We can then construct a graph of the two absorption features.

In [192…
```
x = np.arange(len(spectrum)) # this gives us an array of indices
```

In [193…
```
fig = pylab.figure()
ax = fig.add_subplot(111)

# let's plot the entire spectrum
ax.plot(x, spectrum)

# and show it
pylab.draw()
```

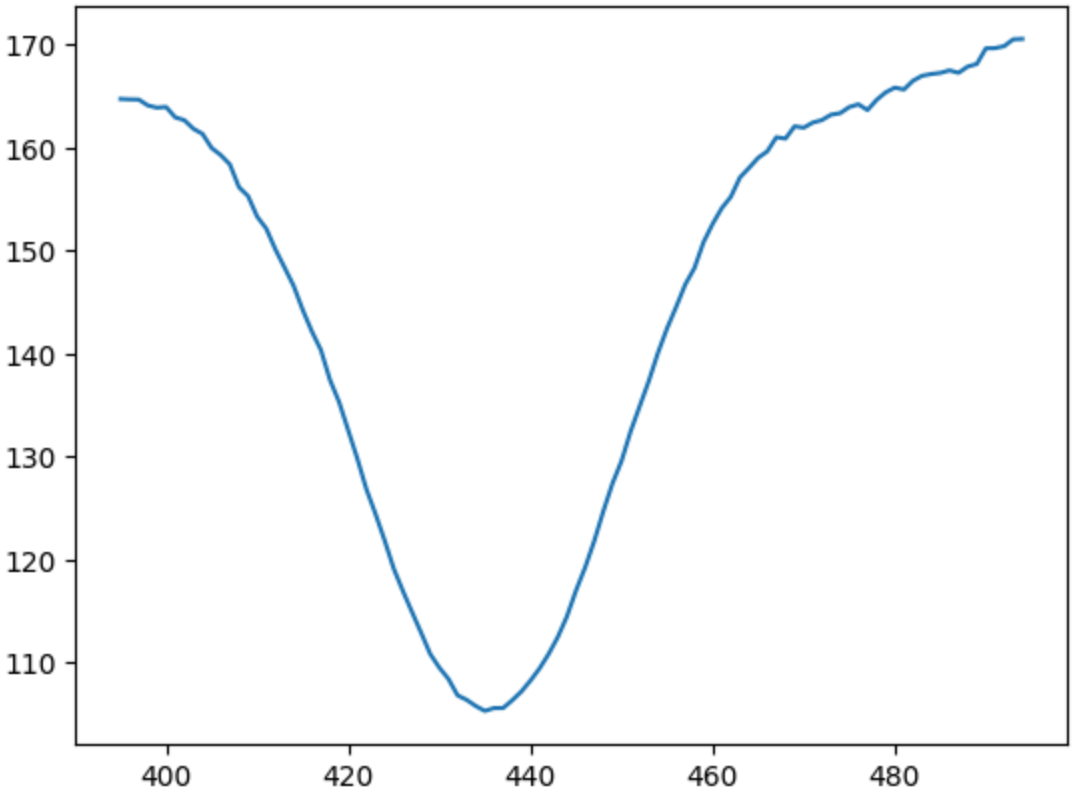


Hopefully we can see two absorption lines!

Fitting for an individual line

What we need is to fit a Gaussian to each line individually in order to measure the center of each line's position. To do this use the plot of the complete spectrum to roughly identify the center pixel. Let's pick the line on the left, and say it's roughly at $x=170$. We can zoom in on that line by plotting only a subrange of the spectrum. The `x` variable works like an index to keep the x axis correct.

In [194…
```
fig = pylab.figure()
ax = fig.add_subplot(111)

# let's plot the subrange of the spectrum
ax.plot(x[395:495], spectrum[395:495])

# and show it
pylab.draw()
```



Next we want to use a routine being provided that fits the position of the absorption line. It is called `fit_abs_line()`. Again, the `x` variable is used to index the array so the values returned are for pixel locations in the original spectrum.

In [195…
```
import sys; sys.path.append('/home/a180i/lib/python/') # this is a temporary
import a180 # import the module of routines for this class
```

```
# let's perform the fit of the absorption line
fy, params = a180.fit_abs_line(x[395:495], spectrum[395:495])

# Use the help feature to read more on what fit_abs_line() does.
help(a180.fit_abs_line)
```

```
Help on function fit_abs_line in module a180:

fit_abs_line(x, y, sp=None)
    Fit an absorption line to a 1-d spectrum.

    Syntax:
      fit_y, parms = fit_abs_line(x, y, sp=None)

    fit_y    The model profile.
    parms    An array of parameters:  [x0, sigma, A, B, C, D]
    sp       An optional array of starting parameters

    The fit is performed such that
       fit_y = B + Cx + Dx^2 - A/sqrt(2 pi)/sigma * exp(-(x-x0)^2 / 2 / sigm
a^2)
```

In this usage we are fitting the data from elements 140 to 200 to a quadratic background plus a Gaussian. If we print the `params` array, it contains the coefficients of the fit.

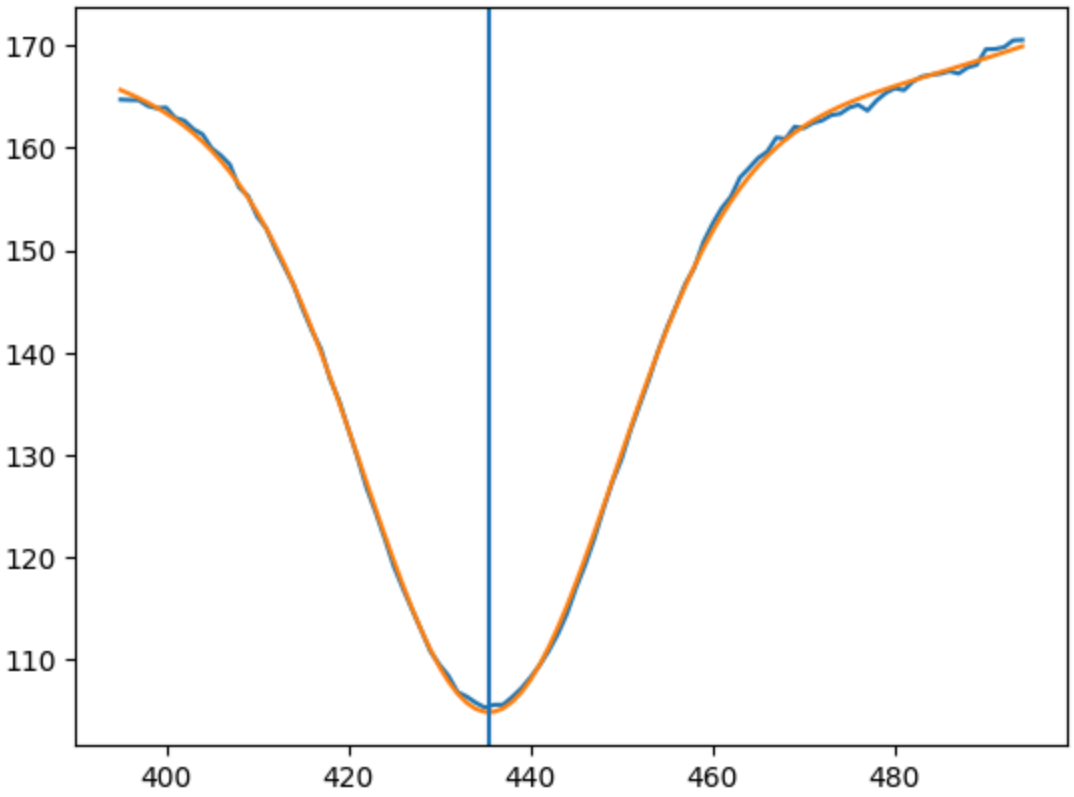
In [196…
```
print(params)
```

```
[ 4.35404747e+02  1.35885666e+01  1.93083818e+03  6.64901869e+02
 -2.30044735e+00  2.62844820e-03]
```

The `params[0]` element (which is the first one listed) is the center of the Gaussian. The routine also produces an array `fy` which is the best fit function evaluated at the x locations in `x[140:200]`. You can overplot this fit on top of the data. You'll again need to use the `x` variable to keep the pixel locations straight:

In [198…
```
fig = pylab.figure()
ax = fig.add_subplot(111)

ax.plot(x[395:495], spectrum[395:495])  # let's plot the subrange of the spe
ax.plot(x[395:495], fy) # and also our best-fit curve
ax.axvline(params[0]) # mark a vertical line at the best-fit position

# and show it
pylab.draw()
print(params[0])
```

```
435.4047472729716
```



Check that the `fy` model is a good fit to the data. If so, we should record the `params[0]` element as the position of the line you we're examining.

We need to repeat this procedure for the right-side spectral line. And we need to do it for all images in our dataset.

Application to several exposures

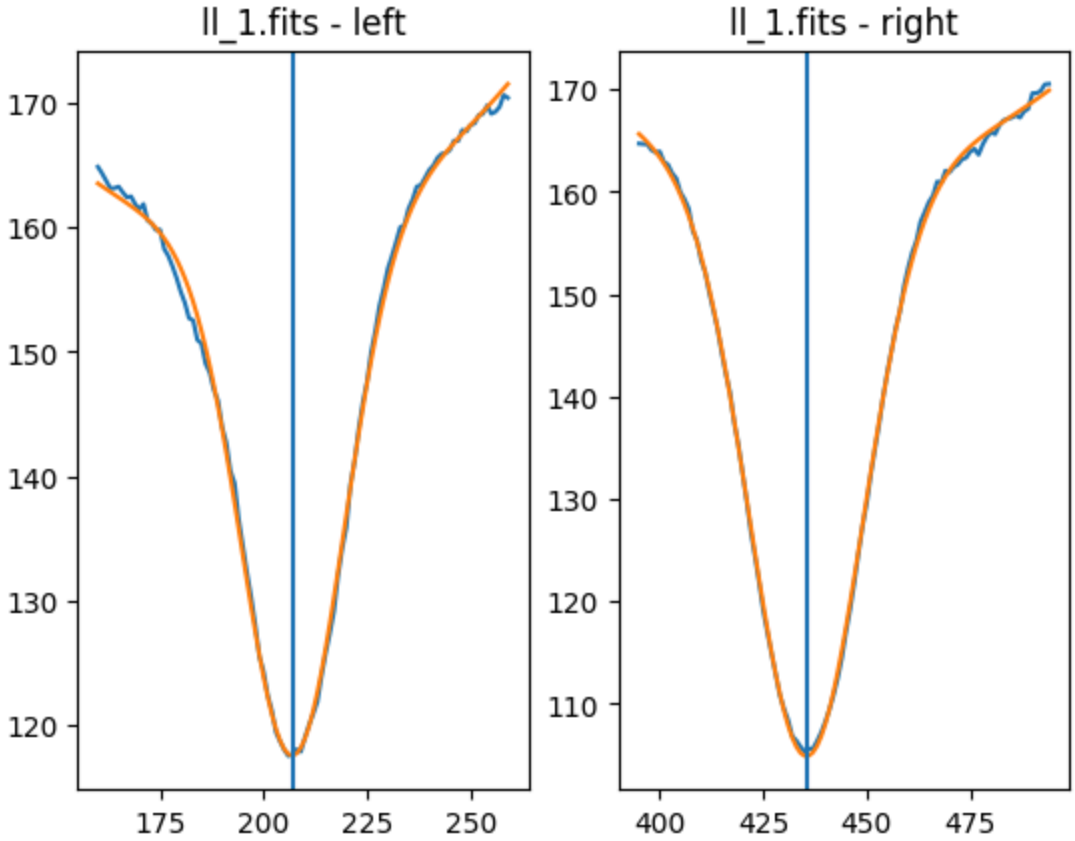
Let's look at a structure for doing the left-side lines on an entire list of files.

In [362…

```
# let's define a list of filenames.  This should be a list of exposures on c
fns = ['ll_1.fits',
       'll_2.fits',
       'll_3.fits',
       'll_4.fits',
       'll_5.fits',
      ]

# let's run a loop over the filenames
leftl_pos = [] # an empty list to store our best-fit results
rightl_pos = []

for i, fn in enumerate(fns):
    # load the data
    print(data_dir+fn)
    im = fits.getdata(data_dir+fn) # load from disk
    im = im[100:300, :] # get the good rows
    spectrum = im.sum(axis=0) # compute the 1d spectrum
```

```
    # get the line fit for the left side
    fyl, paramsl = a180.fit_abs_line(x[160:260], spectrum[160:260])
    leftl_pos.append(paramsl[0]) # store the best-fit position

    # get the line fit for the right side
    fyr, paramsr = a180.fit_abs_line(x[395:495], spectrum[395:495])
    rightl_pos.append(paramsr[0]) # store the best-fit position

    # only plot for the first image
    if i == 0:
        fig = pylab.figure()
        ax1 = fig.add_subplot(121)
        ax2 = fig.add_subplot(122)

        # left
        ax1.plot(x[160:260], spectrum[160:260])
        ax1.plot(x[160:260], fyl)
        ax1.axvline(paramsl[0])
        ax1.set_title(fn + ' - left')

        # right
        ax2.plot(x[395:495], spectrum[395:495])
        ax2.plot(x[395:495], fyr)
        ax2.axvline(paramsr[0])
        ax2.set_title(fn + ' - right')

        pylab.show()

leftl_pos = np.array(leftl_pos) # convert the list to a 1-d numpy array
rightl_pos = np.array(rightl_pos)
```

```
/data/2025-Fall/Lab2/Group4/ll_1.fits
```




```
/data/2025-Fall/Lab2/Group4/ll_2.fits
/data/2025-Fall/Lab2/Group4/ll_3.fits
/data/2025-Fall/Lab2/Group4/ll_4.fits
/data/2025-Fall/Lab2/Group4/ll_5.fits
```

Now lets examine the mean and standard deviation of our measured positions.

In [363…
```
print('left:', round(leftl_pos.mean(), 3), round(leftl_pos.std(),3))
print('right:', round(rightl_pos.mean(), 3), round(rightl_pos.std(), 3))
```

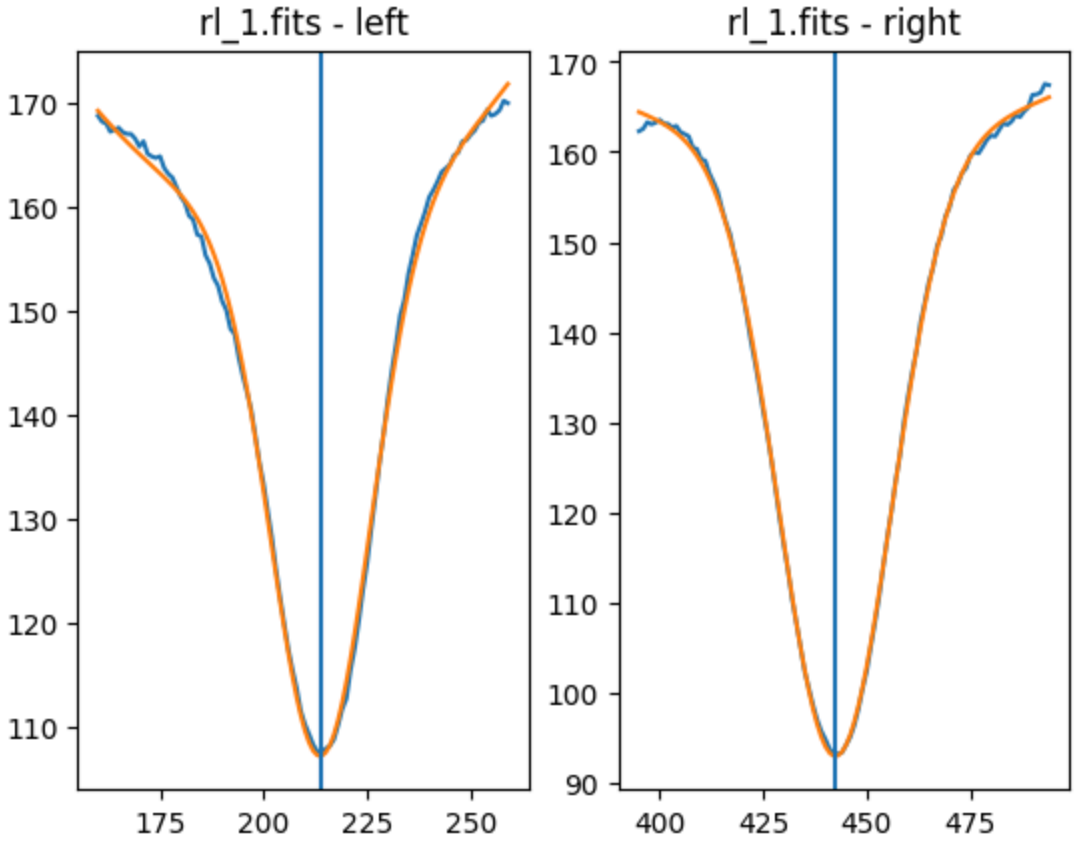
```
left: 207.228 0.049
right: 435.493 0.051
```

Note we have measurements for one limb of the Sun. We need measurements from the other limb of the Sun as well.

In [359…
```
# let's define a list of filenames. This should be a list of exposures on or
fns = ['rl_1.fits',
       'rl_2.fits',
       'rl_3.fits',
       'rl_4.fits',
       'rl_5.fits']

leftr_pos = []
rightr_pos = []

for i, fn in enumerate(fns):
    print(data_dir + fn)
    im = fits.getdata(data_dir + fn)
    im = im[100:300, :]  # crop rows
```

```
    spectrum = im.sum(axis=0)  # 1D spectrum

    # left-side fit
    fyl, paramsl = a180.fit_abs_line(x[160:260], spectrum[160:260])
    leftr_pos.append(paramsl[0])

    # right-side fit
    fyr, paramsr = a180.fit_abs_line(x[395:495], spectrum[395:495])
    rightr_pos.append(paramsr[0])

    # only plot for the first image
    if i == 0:
        fig = pylab.figure()
        ax1 = fig.add_subplot(121)
        ax2 = fig.add_subplot(122)

        # left
        ax1.plot(x[160:260], spectrum[160:260])
        ax1.plot(x[160:260], fyl)
        ax1.axvline(paramsl[0])
        ax1.set_title(fn + ' - left')

        # right
        ax2.plot(x[395:495], spectrum[395:495])
        ax2.plot(x[395:495], fyr)
        ax2.axvline(paramsr[0])
        ax2.set_title(fn + ' - right')

        pylab.show()

leftr_pos = np.array(leftr_pos)
rightr_pos = np.array(rightr_pos)
```

```
/data/2025-Fall/Lab2/Group4/rl_1.fits
```




```
/data/2025-Fall/Lab2/Group4/rl_2.fits
/data/2025-Fall/Lab2/Group4/rl_3.fits
/data/2025-Fall/Lab2/Group4/rl_4.fits
/data/2025-Fall/Lab2/Group4/rl_5.fits
```

In [202…

```
print('left:', round(leftr_pos.mean(), 3), round(leftr_pos.std(),3))
print('right:', round(rightr_pos.mean(), 3), round(rightr_pos.std(), 3))
```

```
left: 213.742 0.071
right: 442.369 0.073
```

Computing the rotation rate

These steps follow the steps in your lab handout. Don't forget to follow your uncertainties!

In [356…

```
# input / known values
lambda1 = 588.99504  # nm (line 1)
lambda2 = 589.59236  # nm (line 2)
error_lam = 0.00001  # nm (1σ lab uncertainty for each λ)
sig_lam_lab = math.sqrt(2 * (error_lam)**2)  # uncertainty on Δλ_lab = sqrt(

delta_lambda_lab = lambda2 - lambda1
lambda_center = 0.5 * (lambda1 + lambda2)

# constants
c_km_s = 299792.458

left_pix1, left_sig1 = leftl_pos.mean(), leftl_pos.std()
```

```
left_pix2, left_sig2 = rightl_pos.mean(), rightl_pos.std()
right_pix1, right_sig1 = leftr_pos.mean(), leftr_pos.std()
right_pix2, right_sig2 = rightr_pos.mean(), rightr_pos.std()

# geometry / corrections
B0_deg = 5.81
error_B0 = 0.01
lat_image_deg = 26.13
error_lat_deg = 0.01
r_frac = 0.9
error_r = 0.3

# step 1: pixel differences and nm/pix scale with uncertainties
delta_pix_left = (left_pix2 - left_pix1)
delta_pix_right = (right_pix2 - right_pix1)
sigma_delta_pix_left  = math.sqrt(left_sig1**2  + left_sig2**2)
sigma_delta_pix_right = math.sqrt(right_sig1**2 + right_sig2**2)

# nm/pix for each side
nm_per_pix_left  = delta_lambda_lab / delta_pix_left
nm_per_pix_right = delta_lambda_lab / delta_pix_right
nm_per_pix_mean  = 0.5 * (nm_per_pix_left + nm_per_pix_right)

# propagate uncertainty in nm/pix including both Δλ_lab and Δpix uncertainti
# nm_per_pix = Δλ_lab / Δpix
# ∂(nm/pix)/∂(Δλ) = 1/Δpix
# ∂(nm/pix)/∂(Δpix) = -Δλ / Δpix^2
sigma_nm_per_pix_left  = math.sqrt((1.0 / delta_pix_left)**2 * (sig_lam_lab*
                                   (delta_lambda_lab / (delta_pix_left**2))*
sigma_nm_per_pix_right = math.sqrt((1.0 / delta_pix_right)**2 * (sig_lam_lab
                                   (delta_lambda_lab / (delta_pix_right**2))

# mean uncertainty (uncorrelated left/right)
sigma_nm_per_pix_mean = 0.5 * math.sqrt(sigma_nm_per_pix_left**2 + sigma_nm_

print("1) Wavelength scale (nm/pixel):")
print(f"   left  : Δpix = {delta_pix_left:.3f}  --> {nm_per_pix_left:.6f} ±
print(f"   right : Δpix = {delta_pix_right:.3f} --> {nm_per_pix_right:.6f} ±
print(f"   mean  : {nm_per_pix_mean:.6f} ± {sigma_nm_per_pix_mean:.6e} nm/pi

# step 2a: observed Doppler shift in nm
sep_pix_line1 = (right_pix1 - left_pix1)
sep_pix_line2 = (right_pix2 - left_pix2)
sigma_sep_pix_line1 = math.sqrt(right_sig1**2 + left_sig1**2)
sigma_sep_pix_line2 = math.sqrt(right_sig2**2 + left_sig2**2)

dlam_line1 = nm_per_pix_mean * sep_pix_line1
dlam_line2 = nm_per_pix_mean * sep_pix_line2
dlam_observed_nm = 0.5 * (dlam_line1 + dlam_line2)

# propagate Δλ (product of nm_per_pix_mean and sep_pix)
# σ(dlam) = sqrt((sep_pix * σ_nm_per_pix_mean)^2 + (nm_per_pix_mean * σ_sep_
sigma_dlam_line1 = math.sqrt((sep_pix_line1 * sigma_nm_per_pix_mean)**2 + (r
sigma_dlam_line2 = math.sqrt((sep_pix_line2 * sigma_nm_per_pix_mean)**2 + (r
sigma_dlam_observed_nm = 0.5 * math.sqrt(sigma_dlam_line1**2 + sigma_dlam_li
```

```
print("2a) Observed Doppler shift (Δλ):")
print(f"   Δλ (line 1) = {dlam_line1:.6f} ± {sigma_dlam_line1:.6e} nm")
print(f"   Δλ (line 2) = {dlam_line2:.6f} ± {sigma_dlam_line2:.6e} nm")
print(f"   Average Δλ  = {dlam_observed_nm:.6f} ± {sigma_dlam_observed_nm:.6

# step 2b: convert to velocity and propagate
# v = c * (Δλ / λ0)
v_line1_km_s = c_km_s * (dlam_line1 / lambda1)
v_line2_km_s = c_km_s * (dlam_line2 / lambda2)
v_observed_km_s = 0.5 * (v_line1_km_s + v_line2_km_s)

# include uncertainty from Δλ and from λ0 itself
# σ_v = sqrt( (c/λ)^2 σ_dλ^2 + (c * dλ / λ^2)^2 σ_λ^2 )
sigma_v_line1 = math.sqrt((c_km_s / lambda1)**2 * (sigma_dlam_line1**2) +
                          (c_km_s * dlam_line1 / (lambda1**2))**2 * (error_l
sigma_v_line2 = math.sqrt((c_km_s / lambda2)**2 * (sigma_dlam_line2**2) +
                          (c_km_s * dlam_line2 / (lambda2**2))**2 * (error_l

sigma_v_observed_km_s = 0.5 * math.sqrt(sigma_v_line1**2 + sigma_v_line2**2)

print("2b) Observed Doppler shift (velocity):")
print(f"   v (line 1) = {v_line1_km_s:.6f} ± {sigma_v_line1:.6e} km/s")
print(f"   v (line 2) = {v_line2_km_s:.6f} ± {sigma_v_line2:.6e} km/s")
print(f"   Average Δv_observed = {v_observed_km_s:.6f} ± {sigma_v_observed_k

# step 3: tilt correction (B0 non-exact)
B0_rad = math.radians(B0_deg)
sigma_B0_rad = math.radians(error_B0)
tilt_factor = 1.0 / math.cos(B0_rad)  # sec(B0)
v_tilt = v_observed_km_s * tilt_factor

# derivative of sec is sec*tan, so σ(sec) = |sec* tan| * σ(B0)
sigma_tilt_factor = abs(tilt_factor * math.tan(B0_rad)) * sigma_B0_rad
sigma_v_tilt = math.sqrt((tilt_factor * sigma_v_observed_km_s)**2 + (v_obser

print(f"3) Tilt correction (B0={B0_deg:.2f}°): v = {v_tilt:.6f} ± {sigma_v_t

# step 4: slit position correction with propagated r_frac uncertainty
v_slit = v_tilt / r_frac
# propagate: σ(v_slit)^2 = (1/r)^2 σ(v_tilt)^2 + (v_tilt / r^2)^2 σ_r^2
sigma_v_slit = math.sqrt((sigma_v_tilt / r_frac)**2 + ((v_tilt * error_r) /

print(f"4) Slit position correction (r_frac={r_frac:.3f} ± {error_r:.3f}): v

# step 5: latitude correction (non-exact)
lat_rad = math.radians(lat_image_deg)
sigma_lat_rad = math.radians(error_lat_deg)
lat_factor = 1.0 / math.cos(lat_rad)
v_lat = v_slit * lat_factor
sigma_lat_factor = abs(lat_factor * math.tan(lat_rad)) * sigma_lat_rad
sigma_v_lat = math.sqrt((lat_factor * sigma_v_slit)**2 + (v_slit * sigma_lat

print(f"5) Latitude correction (lat={lat_image_deg:.2f}°): v = {v_lat:.6f} ±

# step 6: equatorial rotation speed
v_equator_km_s = 0.5 * v_lat
```

```
sigma_v_equator_km_s = 0.5 * sigma_v_lat

print(f"6) Final equatorial rotation velocity = {v_equator_km_s:.2f} ± {sigm
```

```
1) Wavelength scale (nm/pixel):
   left  : Δpix = 228.264  --> 0.002617 ± 8.161157e-07 nm/pix
   right : Δpix = 228.627 --> 0.002613 ± 1.163790e-06 nm/pix
   mean  : 0.002615 ± 7.107129e-07 nm/pix

2a) Observed Doppler shift (Δλ):
   Δλ (line 1) = 0.017032 ± 2.256198e-04 nm
   Δλ (line 2) = 0.017979 ± 2.330182e-04 nm
   Average Δλ  = 0.017505 ± 1.621741e-04 nm

2b) Observed Doppler shift (velocity):
   v (line 1) = 8.669005 ± 1.148382e-01 km/s
   v (line 2) = 9.141950 ± 1.184837e-01 km/s
   Average Δv_observed = 8.905477 ± 8.250181e-02 km/s

3) Tilt correction (B0=5.81°): v = 8.951461 ± 8.292796e-02 km/s

4) Slit position correction (r_frac=0.900 ± 0.300): v = 9.946067 ± 3.316636e
+00 km/s

5) Latitude correction (lat=26.13°): v = 11.078302 ± 3.694193e+00 km/s

6) Final equatorial rotation velocity = 5.54 ± 1.85 km/s
```

Physical size of the Sun

Calculating rotational period

In [357…

```
# Hard-coded measured values (means and 1σ uncertainties)
rho, sigma_rho = 20.5, 2   # same units as r* (e.g. pixels or cm)
r1_spot1, sigma_r1s1 = 10.8, 0.5
r2_spot1, sigma_r2s1 = 4.2, 0.5
r1_spot2, sigma_r1s2 = 8.4, 0.5
r2_spot2, sigma_r2s2 = 6.1, 0.5
dt, sigma_dt = 3 * 24 * 3600, 300        # seconds (3 days ± 300 s)

def theta_and_sigma(r, sigma_r, rho, sigma_rho):
    # Return θ = arcsin(r/ρ) [rad] and propagated σθ
    ratio = r / rho
    theta = math.asin(ratio)
    dtheta_dr = 1.0 / (rho * math.sqrt(1.0 - ratio**2))
    dtheta_drho = -r / (rho**2 * math.sqrt(1.0 - ratio**2))
    sigma_theta = math.sqrt((dtheta_dr * sigma_r)**2 +
                            (dtheta_drho * sigma_rho)**2)
    return theta, sigma_theta

# Compute θ values
theta1_s1, sigma_t1s1 = theta_and_sigma(r1_spot1, sigma_r1s1, rho, sigma_rho
theta2_s1, sigma_t2s1 = theta_and_sigma(r2_spot1, sigma_r2s1, rho, sigma_rho
theta1_s2, sigma_t1s2 = theta_and_sigma(r1_spot2, sigma_r1s2, rho, sigma_rho
```

```
theta2_s2, sigma_t2s2 = theta_and_sigma(r2_spot2, sigma_r2s2, rho, sigma_rho

# Full angles per spot (sum of θ1+θ2)
A1 = theta1_s1 + theta2_s1
sigma_A1 = math.sqrt(sigma_t1s1**2 + sigma_t2s1**2)
A2 = theta1_s2 + theta2_s2
sigma_A2 = math.sqrt(sigma_t1s2**2 + sigma_t2s2**2)

# Average full angle
A_avg = 0.5 * (A1 + A2)
sigma_Aavg = 0.5 * math.sqrt(sigma_A1**2 + sigma_A2**2)

# Rotation rate (ω = A_avg / dt)
omega = A_avg / dt
sigma_omega = math.sqrt((sigma_Aavg / dt)**2 + ((A_avg * sigma_dt) / (dt**2)

# Results
print("\nResults (radians, seconds)")
print(f"θ1_spot1 = {theta1_s1:.8f} ± {sigma_t1s1:.8f}")
print(f"θ2_spot1 = {theta2_s1:.8f} ± {sigma_t2s1:.8f}")
print(f"A1 = {A1:.8f} ± {sigma_A1:.8f}")
print()
print(f"θ1_spot2 = {theta1_s2:.8f} ± {sigma_t1s2:.8f}")
print(f"θ2_spot2 = {theta2_s2:.8f} ± {sigma_t2s2:.8f}")
print(f"A2 = {A2:.8f} ± {sigma_A2:.8f}")
print()
print(f"A_avg = {A_avg:.8f} ± {sigma_Aavg:.8f}")
print(f"ω = {omega:.2e} ± {sigma_omega:.2e} rad/s")
print(f"ω = {omega*86400:.2e} ± {sigma_omega*86400:.2e} rad/day")
```

```
Results (radians, seconds)
θ1_spot1 = 0.55486583 ± 0.06693338
θ2_spot1 = 0.20633911 ± 0.03221766
A1 = 0.76120495 ± 0.07428361

θ1_spot2 = 0.42218667 ± 0.05133691
θ2_spot2 = 0.30213689 ± 0.03971527
A2 = 0.72432355 ± 0.06490594

A_avg = 0.74276425 ± 0.04932250
ω = 2.87e-06 ± 1.90e-07 rad/s
ω = 2.48e-01 ± 1.64e-02 rad/day
```

In [337…

```
# Convert v_equator to m/s
v_equator_m_s = v_equator_km_s * 1e3
sigma_v_m_s = sigma_v_equator_km_s * 1e3

# Solar radius: R = v / ω
R_m = v_equator_m_s / omega
# propagate uncertainty: σ_R = R * sqrt((σ_v/v)^2 + (σ_ω/ω)^2)
sigma_R_m = R_m * math.sqrt((sigma_v_m_s / v_equator_m_s)**2 + (sigma_omega

# Convert to km and solar radii (1 R_sun = 6.957e5 km)
R_km = R_m / 1e3
sigma_R_km = sigma_R_m / 1e3
R_sun_units = R_km / 6.957e5
```

```
sigma_R_sun = sigma_R_km / 6.957e5

print("\nSolar radius")
print(f"R = {R_m:.2e} ± {sigma_R_m:.2e} m")
print(f"R = {R_km:.2e} ± {sigma_R_km:.2e} km")
print(f"R = {R_sun_units:.3f} ± {sigma_R_sun:.3f} R_sun")
```

```
Solar radius
R = 1.93e+09 ± 6.57e+08 m
R = 1.93e+06 ± 6.57e+05 km
R = 2.778 ± 0.945 R_sun
```

Mass of the Sun

In [341…

```
# Constants
G = 6.67430e-11  # m^3 kg^-1 s^-2
# values with uncertainties
T_days = 365.24
sigma_T_days = 0.01  # uncertainty in orbital period
T_sec = T_days * 24 * 3600
sigma_T_sec = sigma_T_days * 24 * 3600

theta_sun_rad = 0.00465  # angular radius of Sun in radians (~0.53°/2)
sigma_theta_rad = 0.00005  # uncertainty in angular size

# Estimate Sun-Earth distance from R and angular size
d = R_m / theta_sun_rad

sigma_d_m = d * math.sqrt((sigma_R_m / R_m)**2 + (sigma_theta_rad / theta_su


M_sun_kg = 4.0 * math.pi**2 * d**3 / (G * T_sec**2)
# Propagate uncertainty:
sigma_M_sun_kg = M_sun_kg * math.sqrt((3 * sigma_d_m / d)**2 + (2 * sigma_T_

# Convert to AU and solar masses
AU_m = 1.495978707e11
d_AU = d / AU_m
sigma_d_AU = sigma_d_m / AU_m

M_sun_units = M_sun_kg / 1.9885e30
sigma_M_sun_units = sigma_M_sun_kg / 1.9885e30

print("\nDistance to Sun")
print(f"d = {d:.2e} ± {sigma_d_m:.2e} m")
print(f"d = {d_AU:.2f} ± {sigma_d_AU:.2f} AU")

print("\nSun's mass from orbital mechanics")
print(f"M_sun = {M_sun_kg:.2e} ± {sigma_M_sun_kg:.2e} kg")
print(f"M_sun = {M_sun_units:.2f} ± {sigma_M_sun_units:.2f} M_sun")
```

```
Distance to Sun
d = 4.16e+11 ± 1.41e+11 m
d = 2.78 ± 0.95 AU

Sun's mass from orbital mechanics
M_sun = 4.27e+31 ± 4.35e+31 kg
M_sun = 21.46 ± 21.90 M_sun
```

This overestimation is primarily due to the dependence of mass on the cube of the orbital distance, which itself carried substantial propagated uncertainty from earlier measurements of the Sun’s angular size and distance.