Dynamic Spectra Plotting

Kuunal Mahtani

Supervisor: Dr. Jing Luo

May 25th, 2020

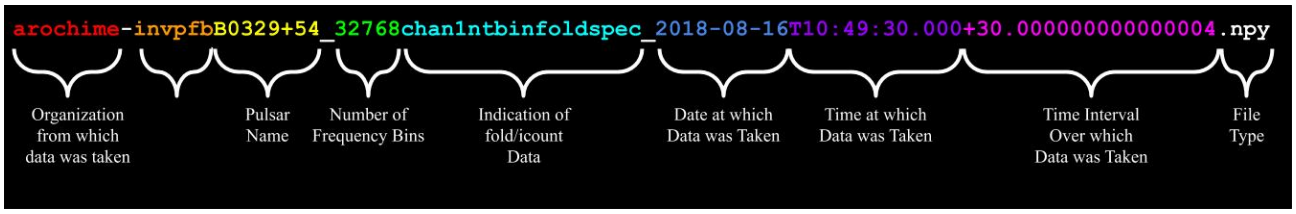
Contents

1 Introduction

This document will explain how to create Dynamic Spectra plots for radio-frequency data measured from as specific pulsar. This document will refer to already folded data.

2 Raw Data Files

The raw data files should be labelled to include information about the data stored in each file. An example of the file name is given as follows: `arochime-invpfbB0329+54_32768chan1ntbinfoldspec_2018-08-16T10:49:30.000+30.000000000000004.npy` and follows the naming convention as follows:



In order to create a dynamic spectra, there should be n number of folded data files and n number of icount data files. Note that for each folded data file conducted at a specific date and time, there exists a corresponding icount data file conducted at the same date and time. The data in each folded data file comprises of a 4D numpy array, in which the first axis is time, second is frequency, third is phase, and fourth is polarization, as follows: `[time,frequency,phase,polarization]`. The data in each icount data file comprises of a 3D array with axes identical to the folded data file, except without the polarization axis. Of particular note in the polarization axis are the XX and YY polarization which are in the `0` and `3` axis, respectively.

3 Analysis

Given that there are $2n$ files to process to produce a single Dynamic Spectra plot, it is advised that the raw data files should be loaded into the notebook in a large array of shape `[n,times,frequencies,phases,polarizations]` wherein each file is stored in the `i`th index of the first axis, and later called upon when necessary.

The raw data needs to be normalized for each file. To do this, divide the folded data file by the corresponding icount data file taken at the same date and time. Note that since the icount data file is a 3D array, it will need to be reshaped to a 4D array.

```
normalized_data = folded_data/icount_data.reshape(icount_data.shape+(1,))
```

It is of note to keep the polarization data, hence the XX and YY polarizations should be summed.

```
summed_polarization_data = normalized_data[... ,0] + normalized_data[... ,3]
```

Using this data, it is possible to create a phase vs. frequency plot quite easily. One can see that the pulse occurs between certain phase bounds more or less consistently across all frequencies. One way of determining these bounds would be to take the mean of the normalized data along the time and frequency axis in order to obtain a phase vs weight data, then plot the resulting data set and determine the bounds by eye.

```
phase_only_data = np.mean(summed_polarization_data,axis=0,keepdims=True)
## averaging over time values, keeping array same dimension

phase_only_data = np.mean(phase_only_data[0,:,:],axis=0)
##averaging over frequency values, result is an array the length of the phase values

matplotlib.plot(phase_only_data)
```

However, this method may lead to misidentification of the bounds, if the data being analyzed is not entirely consistent with the rest of the data files. Therefore, there is a second, albeit more computationally heavy method to determining the boundaries.

To do this, we want to find the bounds of the pulse for each time axis for each data file. Firstly, initialize a 2D array with shape `(n*number_of_time_bins_per_file,2)`. The reasoning will become clear later.

```
##In example, for the data conducted on August 16th for Pulsar B0329+54 (3 time bins per data file):

boundary_values = np.zeros((n*3,2))
```

Let us focus on one data file first. For each time axis `i` in the normalized data array, divide the normalized data by its' mean calculated along the frequency axis.

```
mean_divided_data = summed_polarization_data[i,:,:] /
                                summed_polarization_data[i,:,:].mean(1,keepdims=True)
```

Following, the baseline should be removed from this data. We sum the data along the frequency axis to obtain a 1D array with only the phase data. We then calculate the baseline and standard deviation from this frequency-summed data, and subtract the baseline from the frequency-summed data. we now have a 1D array with phase data where the baseline has been subtracted, the baseline, and the standard deviation. We can use the latter two in order to determine the indices of the pulse boundaries.

```
frequency_summed_data = np.sum(mean_divided_data,axis=0)

baseline = np.mean(frequency_summed_data)

standard_dev = np.std(frequency_summed_data)

no_baseline_data = frequency_summed_data - baseline
```

Once this is done, it is straightforward to set up a conditional in order to determine which values of the data without baseline are greater than one standard deviation of the data, and from these values which ones occur first and last in indices.

```
lower_bound = int(len(no_baseline_data))
upper_bound = 0

for j in range(len(no_baseline_data)):
    if (no_baseline_data[j] > standard_dev):
        if lower_bound>j:
            lower_bound = j
        elif upper_bound < j:
            upper_bound = j
```

Now, if we have done this over one data file we should have found the upper and lower bounds of the distinct time values for a single data file. Recall that the determination of `mean_divided_data`, `frequency_summed_data`, `base`, `standard_dev`, `upper_bound` and `lower_bound` are done in a loop over the number of time bins in the time axis. Hence, over each loop we should store this data in one slot of our previously defined array of shape `(n*number_of_time_bins_per_file,2)`.

```
boundary_values[k] = (lower_bound,upper_bound)
k = k+1
```

Once this has been iterated over all data files, we should now have our `boundary_values` array filled with lower bounds for each time bin in the first index and upper bounds for each time bin in the second index. As a result, `np.mean(bounds_2[:,0])` gives us the average of the lower bound values and `np.mean(bounds_2[:,1])` gives us the average of the upper bound values. These values will be floats, and hence should be rounded (up for the upper bound, down for the lower bound) to the nearest integer. We shall going forward use these values as constant upper and lower bounds.

```
constant_lower_bound = int(np.floor(np.mean(bounds_2[:,0])))
constant_upper_bound = int(np.ceil(np.mean(bounds_2[:,1])))
```

Now that we have our bounds, we need to clean up our data by taking a weighted average. Let us once more focus on one individual file first. To do this, we shall first reduce our data in the time and frequency axis to obtain a 1D array of phase values. We average our data in the time axis, then sum the resulting array in the frequency axis.

```
phase_only_data = np.mean(summed_polarization_data, axis=0, keepdims=True)
phase_only_data = np.sum(phase_only_data, axis=0)
```

We now need to create "on" and "off" pulse profiles. The "on" pulse profile will be "on" over the phase bins of the pulse and =0 or "off" over all other phase bins. We create a 1D array of zeros the length of the number of phase bins in our data, then assign the values of our `phase_only_data` to this new array, but only within the boundary limits we determined previously. We want to create the "off" pulse profile such that it has a value of 1 over a relatively large range of phase bins which **DO NOT** cover the pulse.

```
pulse_profile_on = np.zeros_like(phase_only_data)
pulse_profile_on[constant_lower_bound:constant_upper_bound] =
                                    phase_only_data[constant_lower_bound:constant_upper_bound]

pulse_profile_off = np.zeros_like(phase_only_data)
pulse_profile_off[100:200] = 1

## for the aforementioned data, the pulse occurs between phase bins 425 and 475 (there are a
## total of 512 phase bins). Hence, the range of [100:200] was chosen to illustrate a section
## of the phase bins far away from the pulse which do not contain the pulse.
```

The weighted average of the data is then calculated as the sum along the phase axis of the product between the `summed_polarization_data` and the pulse profiles, divided by the sum of the pulse profiles. We can further clean up the data by taking the ratio of the weighted on pulse and off pulse profiles minus 1.

```
weighted_avg_on_pulse = np.sum(summed_polarization_data*pulse_profile_on,axis=2)
                                                                    /np.sum(pulse_profile_on)

weighted_avg_off_pulse = np.sum(summed_polarization_data*pulse_profile_off,axis=2)
                                                                    /np.sum(pulse_profile_off)

cleaned_data = weighted_avg_on_pulse/weighted_avg_off_pulse - 1
```

We now have our cleaned data. Recall from the phase vs. frequency plot that some frequency bands need to be weighted to = 0, since they only contain noise. This is easy to do and can be done once the data has been cleaned. Now we have our data analyzed for a single file. We loop this process over all files, and in each iteration of the loop concatenate our cleaned data along the time axis in order to obtain a 2D array containing entries in the time and frequency axis. Using this, we may then plot the Dynamic Spectra plot.

```
##(in a loop iterating over all files over "j")

if (j == 0):
    Dynamic_Spectra_Data = masked_cleaned_data
else:
    Dynamic_Spectra_Data = np.concatenate((Dynamic_Spectra_Data,masked_cleaned_data))

plt.figure(figsize=(14,14))
plt.imshow(Dynamic_Spectra_Data.T,aspect='auto')
plt.xlabel('Time', size=20)
plt.ylabel('Frequency', size=20)
plt.colorbar()
plt.savefig('../Dynamic/Dynamic_Spectra.png')
```

The result of this process for the data collected on August 16th, 2018 on Pulsar B0329+54 is as follows:

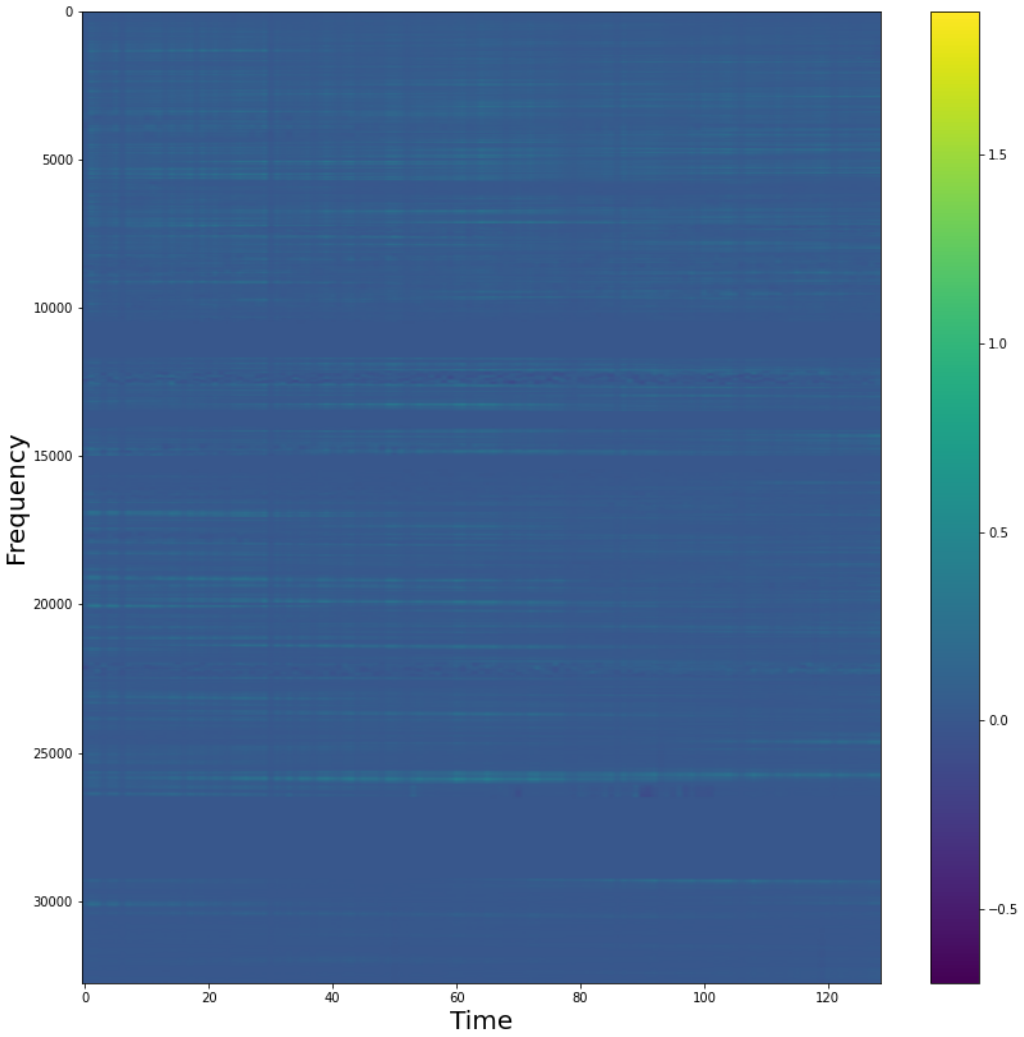


Figure 1: Dynamic Spectra for B0329+54 for data collected on August 16th, 2018

4 Citations & References

-