NIUS Project Report

February 28, 2024

PulseJitterAnalysis

PranavKalsi

IndianInstituteofScience, Bangalore

1 Introduction

Pulsars are fast spinning Neutron Stars with high magnetic fields. As a result charged particles accelerate to relativistic speeds in the spining magnetic field leading to a beamed syncrotron emission. And when this beam happens to be in the line of sight of the Earth, we precieve them as periodic Pulsed signals.

Pulsars spin with amazing stability, which allows us to use them as highly-accurate clocks, distributed all over the galaxy. Pulsar timing is a powerful technique which allows pulsar observations to be used in tests of General Relativity (Detect gravitational waves), Plasma Physics (Understand the properties of interviening plasma) and Nuclear physics.

1.1 High precision Timing

High precision timing of the pulsar signals is very important for numerous applications listed above. We often have to develop methods to distinguish pulsar signals from the radio noise from the background . Individual Pulse signals are usually dominated by the noise , so we average many pulses (in time and frequency) over the course of an observation to increase the signal to noise ratio. If the noise is Gaussian, it should be cancelled out, leaving the signal from the pulsar. This results in an average 'pulse profile'. This technique is known as 'pulse folding'. The Pulse Profile is like a unique fingerprint of the pulsar.

High Precision timing can be summarized in a few steps: 1. Generate a template profile of the pulsar by integrating the pulse over long time. 2. Generate average folded profiles called "subints". 3. Find time of arrival of the folded pulses by comparing it with standard Template profile using suitable algorithms.

We will demonstrate the above steps by taking a sample data of B1642-03 Pulsar. We will use PSRchive (both Python interface and CLI) for analysing the pulsar data.

2 Data Analysis

Lets first look at some parameters of the datafile by running the following command in the shell

psredit B1642-03_59064.714066_500.norfix.fits

Commands	Specifications	Value
file	Name of the file	B1642-03_59064.714066_500.norfix.fits
nbin	Number of pulse phase bins	512
nchan	Number of frequency channels	4096
npol	Number of polarizations	1
nsubint	Number of sub-integrations	2404
type	Observation type	Pulsar
site	Telescope name	GMRT
name	Source name	B1642-03
coord	Source coordinates	16:45:02.041-03:17:57.82
freq	Centre frequency (MHz)	400.0244140625
bw	Bandwidth (MHz)	-200
dm	Dispersion measure (pc/cm^3)	35.755500793457
rm	Rotation measure (rad/m^2)	0
dmc	Dispersion corrected	0
rmc	Faraday Rotation corrected	0
polc	Polarization calibrated	0
scale	Data units	FluxDensity
state	Data state	Intensity
length	Observation duration (s)	2403.16841984003
int*:@	int:help for attribute list	
ext:obs_mode	Observation Mode	PSR
ext:obsfreq	Centre frequency	400.0244140625
ext:obsbw	Bandwidth	-200
ext:obsnchan	Number of channels	4096
ext:hdrver	Header Version	6.2
ext:date	File Creation Date	2023-12-28T22:22:04
ext:coord_md	Coordinate mode	J2000
ext:equinox	Coordinate equinox	2000
ext:trk_mode	Tracking mode	UNSET
ext:bpa	Beam position angle	0
ext:bmaj	Beam major axis	0
ext:bmin	Beam minor axis	0
ext:stt_date	Start UT date	UNSETTUNSE
ext:stt_time	Start UT	
ext:stt_imjd	Start MJD	59064
ext:stt_smjd	Start second	61695
ext:stt_offs	Start fractional second	0.444644598341256
ext:stt_lst	Start LST	0
ext:stt_crd1	Start coord 1	16:45:02.041
ext:stt_crd2	Start coord 2	-03:17:57.819

Commands	Specifications	Value
ext:stp_crd1	Stop coord 1	UNSET
ext:stp_crd2	Stop coord 2	UNSET
ext:ra	Right ascension	16:45:02.041
ext:dec	Declination	-03:17:57.819
obs:observer	Observer name(s)	
obs:projid	Project name	
rcvr:name	Receiver name	unknown
rcvr:basis	Basis of receptors	lin
rcvr:hand	Hand of receptor basis	+1
rcvr:sa	Symmetry angle of receptor basis	45deg
rcvr:rph	Reference source phase	0deg
rcvr:fdc	Receptor basis corrected	0
rcvr:prc	Receptor projection corrected	0
rcvr:ta	Tracking angle of feed	0deg
be:name	Name of the backend instrument	GWB
be:phase	Phase convention of backend	+1
be:hand	Handedness of backend	+1
be:dcc	Downconversion conjugation corrected	0
be:phc	Phase convention corrected	0
be:delay	Backend propn delay from digi. input.	0
be:config	Configuration filename	
be:nrcvr	Number of receiver channels	0
be:tcycle	Correlator cycle time	0
hist:nrow	Number of rows in history	1
hist:nbin_prd	Nr of bins per period	512
hist:tbin	Time per bin or sample	0.000757272593098122
hist:chan_bw	Channel bandwidth	-0.048828125
hist:cal_file	Calibrator filename	NONE
aux:dm_model	Auxiliary dispersion model	NONE
aux:dmc	Auxiliary dispersion corrected	0
aux:rm_model	Auxiliary birefringence model	NONE
aux:rmc	Auxiliary birefringence corrected	0
sub:int_type	Time axis (TIME, BINPHSPERI, BINLNGASC, etc)	TIME
sub:int_unit	Unit of time axis (SEC, PHS (0-1), DEG)	SEC
sub:tsamp	[s] Sample interval for SEARCH-mode data	0
sub:nbits	Nr of bits/datum (SEARCH mode 'X' data, else 1)	-1
sub:nch_strt	Start channel/sub-band number (0 to NCHAN-1)	-1
sub:nsblk	Samples/row (SEARCH mode, else 1)	-1
sub:nrows	Nr of rows in subint table (search mode)	2404
sub:zero_off	Zero offset for SEARCH-mode data	0
sub:signint	1 for signed ints in SEARCH-mode data, else 0	0

The following data analysis demonstrates the use of python interface of PSRchive for pulsar Data analysis. We will include the following examples: 1. Importing and reading PSRfits file ,data visualization and manipulation and saving it to a new file. 2. Generating time of Arrivals and visualizing the residuals.

Now we will use Python interface of PSRchive for furthur Data analysis.

2.1 Playing with PSRfits file

[466]:
```
# importing important modules
import matplotlib.pyplot as plt
import numpy as np
import psrchive
import pylab
import scipy
from IPython.display import Math
Math(r" \newcommand{\centering}{\begin{centre}}")
Math(r" \newcommand{\endcentering}{\end{centre}}")
%matplotlib ipympl
```

[2]:
```
Raw_archive=psrchive.Archive_load('/home/common_user/Projects/N_J_P/Data/
↪B1642-03_59064.714066_500.norfix.fits')
```

```
Unrecognized telescope code (GMRT)
```

[3]:
```
Data=Raw_archive.get_data()
```

[4]:
```
Data.shape
```

[4]:
```
(2404, 1, 4096, 512)
```

The following can be inffered from above: 1. number of subints = 2404 2. number of polarization channels = 1 3. number of Frequency channels = 4096 4. number of Phase bins = 512

One can crunch the data in different dimensions . for example:=

[5]:
```
Raw_archive.fscrunch_to_nchan(1024)
```

Above command takes 4096 frequeny channels and converts them into 1024 by taking average in pair of 4 consecutive channels.

Similarily one can run the following commands to crunch in different dimensions:

1. tscrunch_to_nsub(n)# will crunch the existing number of subints to 'n' by appropriately taking average over consecutive subints.
2. tscrunch(n) # will combine n consecutive subints into one and thus reduce the number of subints by n . For example if we had 2404 subints tscrunch(n) will crunch it to 2404/n subints.
3. pscrunch(n) # For crunching polarization
4. fscrunch(n) # For crunching frequency channels

5. bscrunch(n) # For crunching phase bins
6. bscrunch_to_nbin(n) # For crunching phase bins to n

For more such commands, refer https://psrchive.sourceforge.net/manuals/python/

2.2 Visualization

We have already created dedispersed fits file by the following commands:

[3]:
```
Raw_archive.dedisperse() # to dedisperse the data
Raw_archive.remove_bfrom scipy import statsaseline()  #to remove the baseline␣
↪data
Raw_archive.unload('Dedispersed_raw_data_B1642-03') # to save it to a new file
```

[3]:
```
Raw_archive=psrchive.Archive_load('Dedispersed_raw_data_B1642-03')
```

```
Unrecognized telescope code (GMRT)
```

[4]:
```
Dd_Data=Raw_archive.get_data()
```

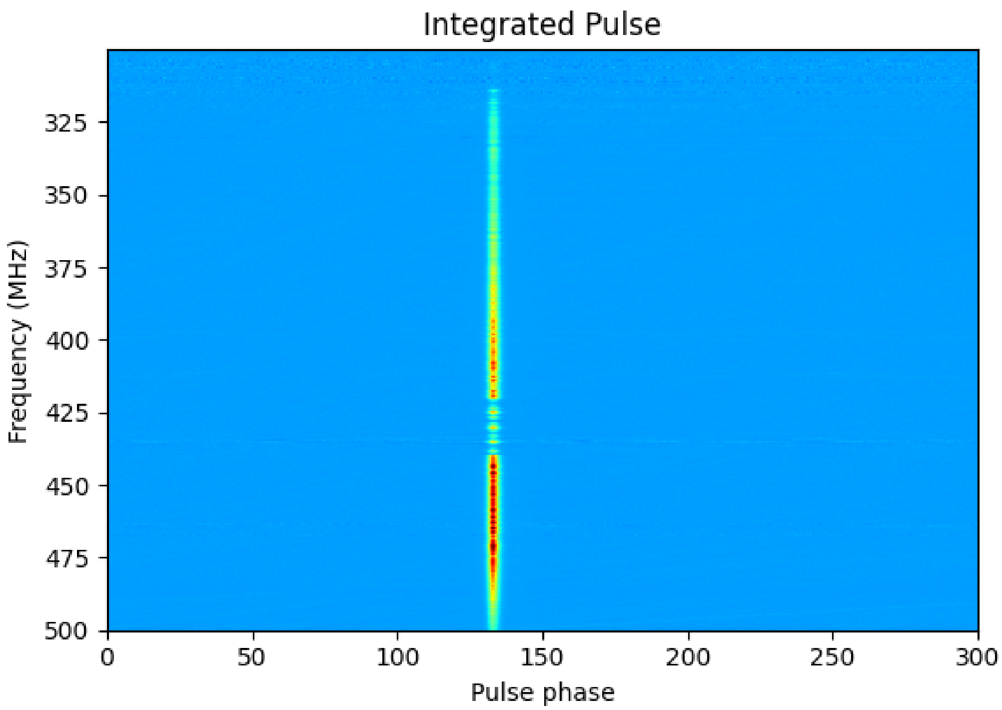
[5]:
```
Dd_Data.shape
```

[5]:
```
(2404, 1, 4096, 512)
```

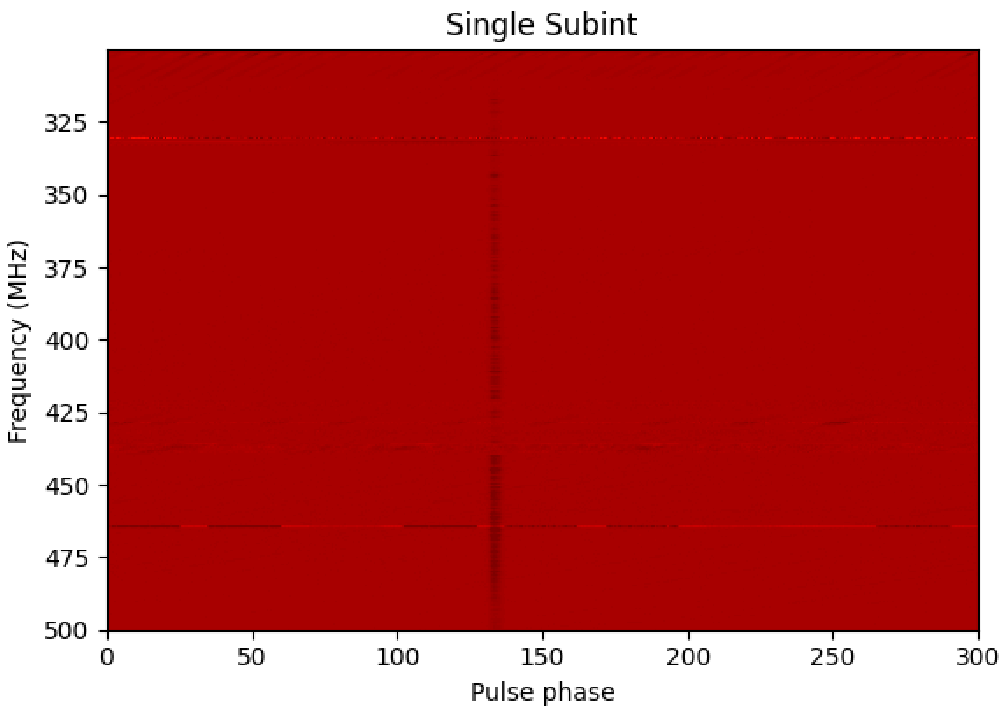
2.2.1 Pulse Phase vs frequency plot

Here we will see the magic of averaging:

[22]:
```
fig1, ax1 = plt.subplots()
freq_lo = Raw_archive.get_centre_frequency() - Raw_archive.get_bandwidth()/2.0
freq_hi = Raw_archive.get_centre_frequency() + Raw_archive.get_bandwidth()/2.0
ax1.imshow(Dd_Data[:,0,:,:].
↪mean(0),extent=(0,300,freq_lo,freq_hi),cmap='jet',vmax=1000) # the pulse␣
↪phase is streached by 300 for proper display
ax1.set_title('Integrated Pulse')
ax1.set_xlabel('Pulse phase')
ax1.set_ylabel('Frequency (MHz)')
plt.show()
```



[23]:
```
fig1, ax1 = plt.subplots()
freq_lo = Raw_archive.get_centre_frequency() - Raw_archive.get_bandwidth()/2.0
freq_hi = Raw_archive.get_centre_frequency() + Raw_archive.get_bandwidth()/2.0
ax1.imshow(Dd_Data[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',vmax=1000) # the pulse phase is␣
 ↪streached by 300 for proper display
ax1.set_title('Single Subint')
ax1.set_xlabel('Pulse phase')
ax1.set_ylabel('Frequency (MHz)')
plt.show()
```



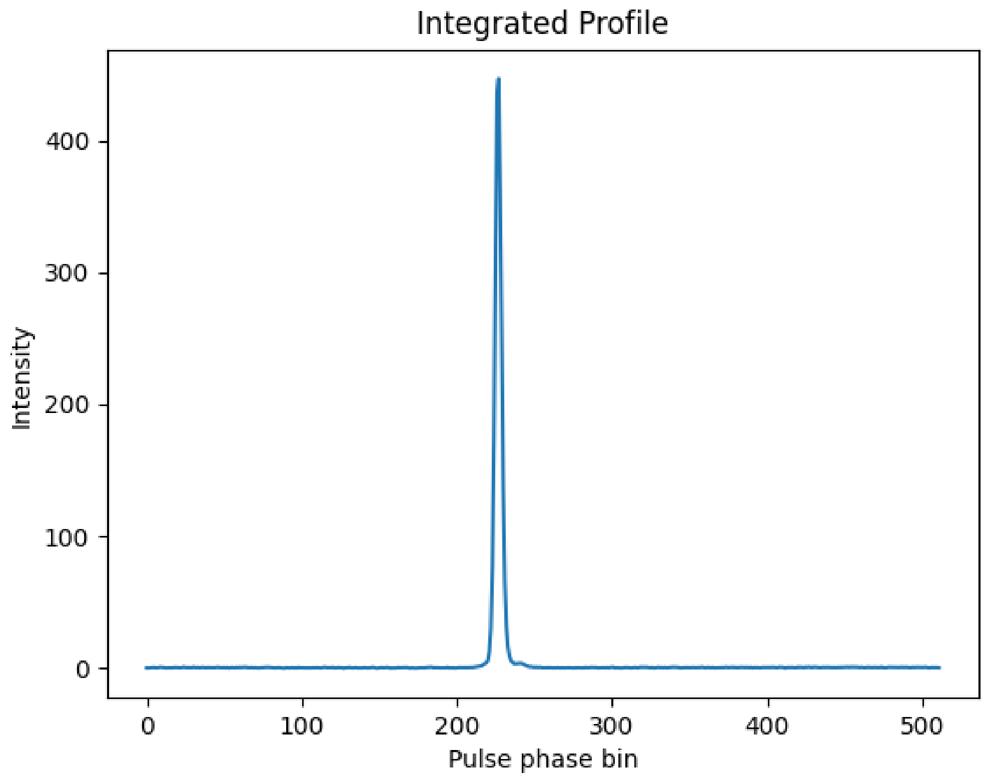
We can clarly compare the signal and noise levels in the above graphs

2.2.2 Pulse Phase vs Intensity plot after crunching in frequency

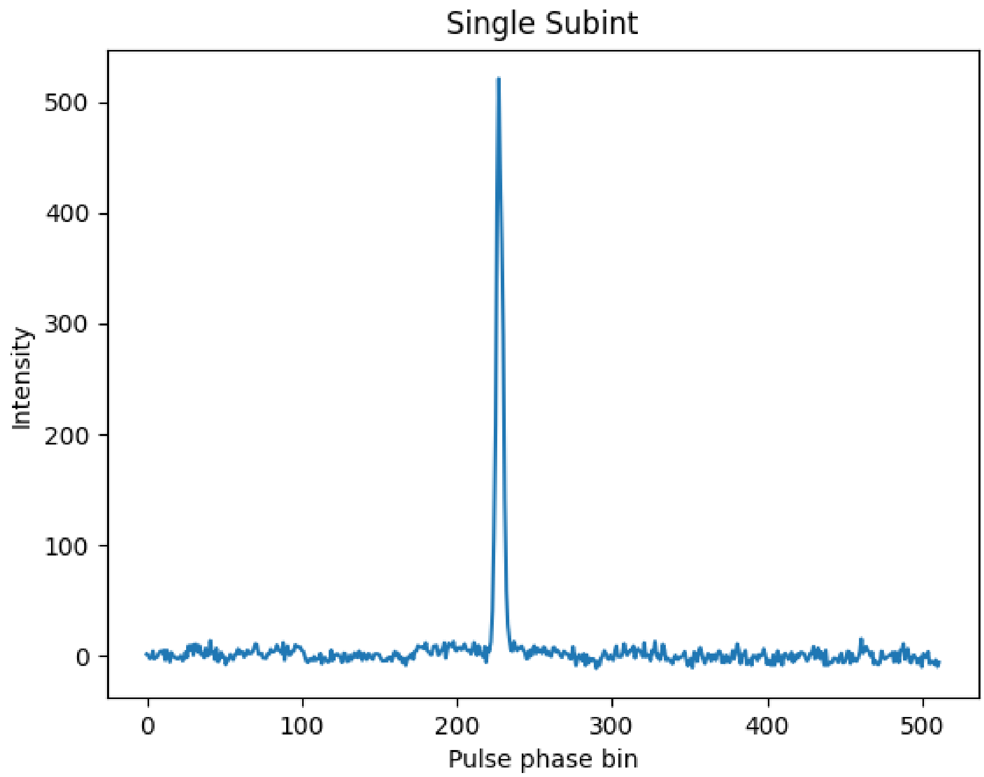
[29]:
```
Raw_archive.fscrunch_to_nchan(1) # crunching frequency data to 1 channel
Raw_archive.unload('single_chan_B1642-03')
```

[30]:
```
Data=Raw_archive.get_data()
```

[40]:
```
fig2, ax2=plt.subplots()
ax2.plot(Data[:,0,0,:].mean(0))
ax2.set_title('Integrated Profile')
ax2.set_xlabel('Pulse phase bin')
ax2.set_ylabel('Intensity')
plt.show()
```




```
[41]: fig2, ax2=plt.subplots()
      ax2.plot(Data[1,0,0,:])
      ax2.set_title('Single Subint')
      ax2.set_xlabel('Pulse phase bin')
      ax2.set_ylabel('Intensity')
      plt.show()
```



Again we compare the noise .

2.2.3 Preparing Template profile and other fits files

In this section we will prepare the following files and analyse them later: we will crunch the original file to 64 channels for ease of computing and then preapre fits files with average of subints as 4 , 16 ,64 , 256 ,1024.

[4]:
```
# note: we have imported the dedispersed file again as Raw_archive, by running␣
↪cell #[3]
Raw_archive.fscrunch_to_nchan(64) # crunching frequency channels from 4096 to 64
```

[5]:
```
Raw_archive.unload('64_chan_B1642-03')
```

[8]:
```
Raw_archive.tscrunch(4) # collapsing 4 adjacent subints _ so the number of␣
↪subints become 2404/4 = 601
Raw_archive.unload('601_subints_B1642-03')
```

[12]:
```
Raw_archive.tscrunch(4) # Furthur collapsing 4 adjacent subints _ so the number␣
↪of subints become 2404/(4*4) = 151 (greatest integer)
Raw_archive.unload('151_subints_B1642-03')
```

```
Raw_archive.tscrunch(4) # Furthur collapsing 4 adjacent subints _ so the number␣
 ↪of subints become 2404/(4*4*4) = 38 (greatest integer)
Raw_archive.unload('38_subints_B1642-03')
```

[18]:
```
Raw_archive.tscrunch(4) # Furthur collapsing 4 adjacent subints _ so the number␣
 ↪of subints become 2404/(4*4*4*4) = 10 (greatest integer)
Raw_archive.unload('10_subints_B1642-03')
```

[21]:
```
Raw_archive.tscrunch(4) # Furthur collapsing 4 adjacent subints _ so the number␣
 ↪of subints become 2404/(4*4*4*4*4) = 3 (greatest integer)
Raw_archive.unload('3_subints_B1642-03')
```

[38]:
```
Raw_archive.tscrunch(3) # Furthur collapsing 4 adjacent subints _ so the number␣
 ↪of subints become 2404/(4*4*4*4*4*3) = 1 (greatest integer)
Raw_archive.unload('1_subint_B1642-03')
```

[52]:
```
# we have already created a template file (average of all 2404 subints) with 64␣
 ↪channels
```

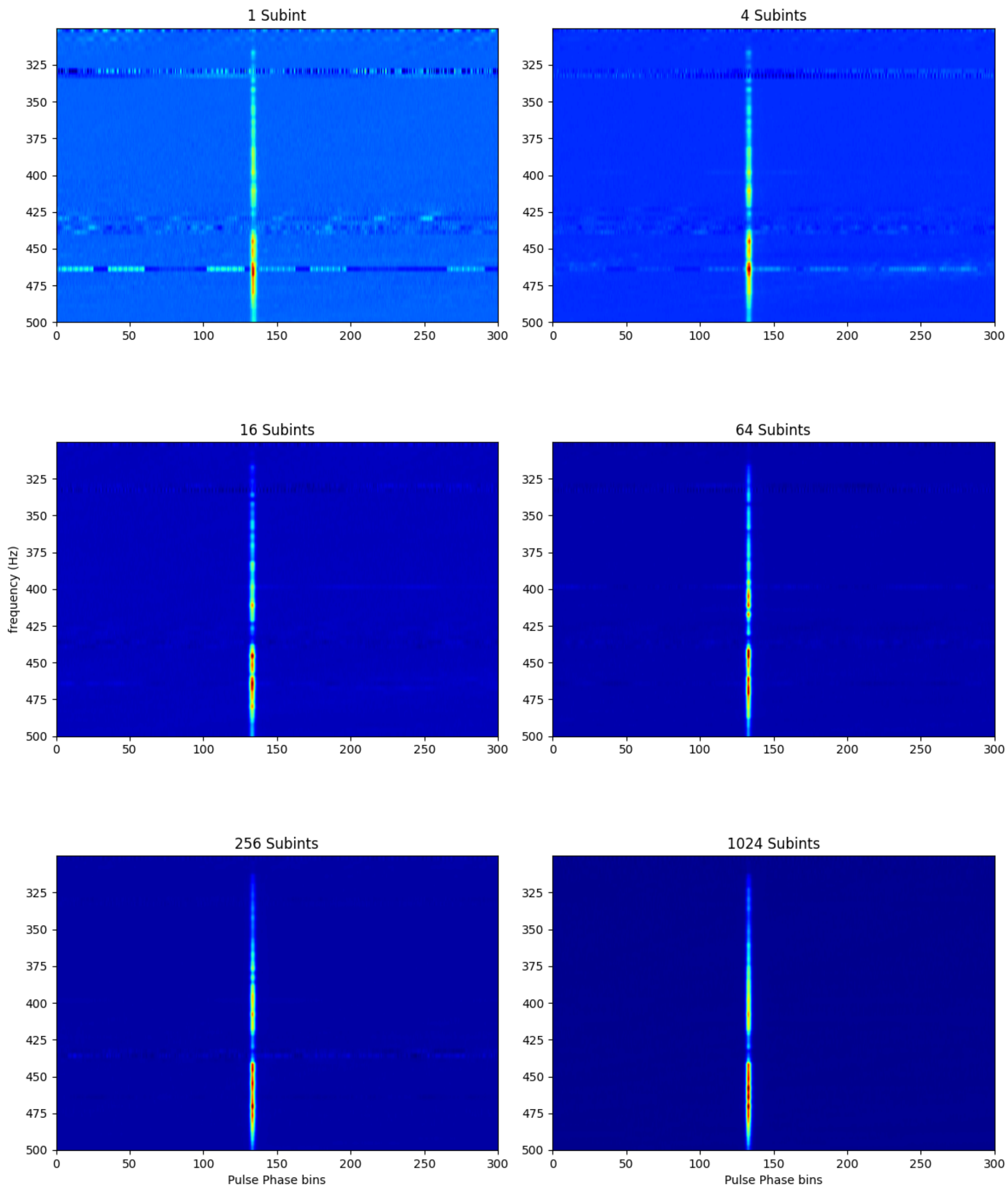
[437]:
```
Arch1=psrchive.Archive_load('64_chan_B1642-03')
Arch4=psrchive.Archive_load('601_subints_B1642-03')
Arch16=psrchive.Archive_load('151_subints_B1642-03')
Arch64=psrchive.Archive_load('38_subints_B1642-03')
Arch256=psrchive.Archive_load('10_subints_B1642-03')
Arch1024=psrchive.Archive_load('3_subints_B1642-03')
#L=[Arch1,Arch4,Arch16,Arch64,Arch256,Arch1024]
```

[55]:
```
Data1=Arch1.get_data()
Data4=Arch4.get_data()
Data16=Arch16.get_data()
Data64=Arch64.get_data()
Data256=Arch256.get_data()
Data1024=Arch1024.get_data()
```

2.2.4 Plotting different averaged subints

[95]:
```
plt.clf()
fig,axes = plt.subplots(3,2,figsize=(12,15))
freq_lo = Arch1.get_centre_frequency() - Arch1.get_bandwidth()/2.0
freq_hi = Arch1.get_centre_frequency() + Arch1.get_bandwidth()/2.0

axes[0,0].imshow(Data1[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',label='4 Subints')
axes[0,0].set_title('1 Subint')
```

```
axes[0,1].imshow(Data4[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',label='4 Subints')
axes[0,1].set_title('4 Subints')

axes[1,0].imshow(Data16[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',label='4 Subints')
axes[1,0].set_title('16 Subints')

axes[1,1].imshow(Data64[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',label='4 Subints')
axes[1,1].set_title('64 Subints')

axes[2,0].imshow(Data256[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',label='4 Subints')
axes[2,0].set_title('256 Subints')

axes[2,1].imshow(Data1024[1,0,:,:
 ↪],extent=(0,300,freq_lo,freq_hi),cmap='jet',label='4 Subints')
axes[2,1].set_title('1024 Subints')

axes[1,0].set_ylabel('frequency (Hz)')

axes[2,0].set_xlabel('Pulse Phase bins')
axes[2,1].set_xlabel('Pulse Phase bins')

plt.tight_layout()
plt.show()
```



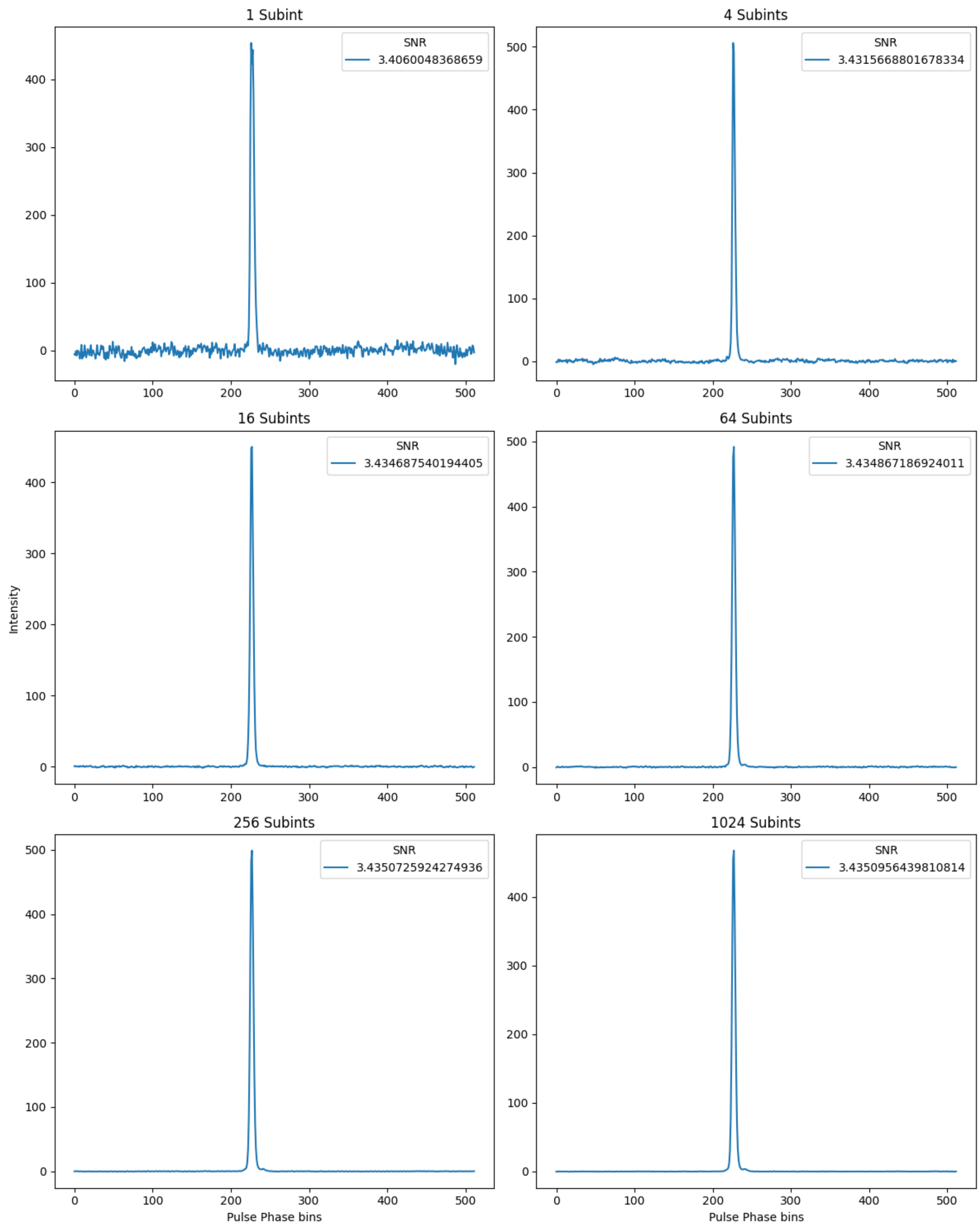
One can clearly see the differnce in noise level

2.2.5 Finding SNR after collapsing in frequency

The SNR is calculated as follows: 1. Compute on Pulse RMS value 2. Compute off Pulse RMS Value 3. Retrun 1. / 2.

[467]:
```
def SNR(Dataset,Pulse_start_bin,Pulse_end_bin):
    ssr=(1/(512-(Pulse_end_bin-Pulse_start_bin))) *  (np.sum((Dataset[0:
 ↪Pulse_start_bin])**2) + np.sum( (Dataset[Pulse_start_bin:-1])**2 ) )
    off_pstd = np.sqrt( ssr )
    on_pstd = np.sqrt((1/( Pulse_end_bin-Pulse_start_bin) )*np.
 ↪sum((Dataset[Pulse_start_bin:Pulse_end_bin])**2) )
    pstd=np.std(Dataset)
    return (on_pstd/off_pstd)
```

[209]:
```
plt.clf()
fig3,axes3 = plt.subplots(3,2,figsize=(12,15))
axes3[0,0].plot(Data1[10][0].mean(0),label=SNR(Data1[10][0].mean(0),210,250))
axes3[0,0].set_title('1 Subint')
axes3[0,0].legend(title='SNR')

axes3[0,1].plot(Data4[10][0].mean(0),label=SNR(Data4[10][0].mean(0),210,250))
axes3[0,1].set_title('4 Subints')
axes3[0,1].legend(title='SNR')

axes3[1,0].plot(Data16[10][0].mean(0),label=SNR(Data16[10][0].mean(0),210,250))
axes3[1,0].set_title('16 Subints')
axes3[1,0].legend(title='SNR')

axes3[1,1].plot(Data64[10][0].mean(0),label=SNR(Data64[10][0].mean(0),210,250))
axes3[1,1].set_title('64 Subints')
axes3[1,1].legend(title='SNR')

axes3[2,0].plot(Data256[4][0].mean(0),label=SNR(Data256[4][0].mean(0),210,250))
axes3[2,0].set_title('256 Subints')
axes3[2,0].legend(title='SNR')

axes3[2,1].plot(Data1024[1][0].mean(0),label=SNR(Data1024[1][0].
 ↪mean(0),210,250))
axes3[2,1].set_title('1024 Subints')
axes3[2,1].legend(title='SNR')

axes3[1,0].set_ylabel('Intensity')

axes3[2,0].set_xlabel('Pulse Phase bins')
axes3[2,1].set_xlabel('Pulse Phase bins')

plt.tight_layout()
plt.show()
```



Oh!! We don't see much difference in the SNR as we increase the number of staked subints. why? This is because our signal profile is extrmely prominent. The effect of stacking (averaging) is visible when the pulse signal is compareable to the noise level.

2.3 Timing Analysis

Here we will demonstrate time of arrivals of the pulses and plot the residuals:

2.3.1 Timing Algorithms :

At the present time, pat can use any one of five algorithms to determine the phase shift between the standard template and the observed Profile.

1. Fourier Phase Gradient (PGS):
   It takes advantage of a property of the Fourier transform known as the "shift theorem", which states that the Fourier transform of a function and a shifted copy of the function differ only by a linear phase gradient. Fitting for this gradient in the Fourier domain can determine the shift between two similar Profiles. This algorithm is very precise when the S/N of the Profile is high.
2. Gaussian Interpolation Shift (GIS):
   Pat calculates the discrete cross correlation function of the Profile with the template and a Gaussian curve is then fitted to the resulting points to allow interpolation between each phase bin. In this manner, TOAs can be determined to within approximately 1/10 of the width of an individual phase bin. This algorithm is less susceptible to noise contamination than the PGS method, but is less precise when the S/N is high.
3. Parabolic Interpolation Shift (PIS):
   This is the oldest method available. It is very similar to the GIS method, but uses only the peak bin of the cross correlation function and one bin on either side to define a parabola that is used for interpolation.
4. Zero Pad Fitting (ZPF):
   This method interpolates the cross correlation function by Fourier transforming, padding the result with zeroes and transforming back to the time domain. It is somewhat experimental and the error estimate it returns is not reliable.
5. Sinc Interpolation Shift (SIS):
   This algorithm is similar in effect to the ZPF method.

All of the above methods fit only the Stokes I profile. The user can also choose to fit the full polarimetric profile in the Fourier domain using an algorithm described by van Straten in the Astrophysical Journal (in press). Normally, the standard template profile is loaded from a Pulsar::Archive, but it is also possible to use an analytic standard template constructed from Gaussian components. pat is compatible with multiple TOA output formats, including parkes, itoa, princeton and the more modern tempo2 format. Additional flags can be added to the default output when using the tempo2 format, allowing the user to carry extra information (like the name of the instrument used to record the data) along with the TOAs.

2.3.2 Generating TOAs

The psrchive.ArrivalTime().get_toas() generates time of arrival in form of strings , so we use custom functions to extract toas. P.S. This part is not well documented on official website.

[450]:
```
def get_toas_string_tuple(std,obs):
    #std is the standard pulse profile , and obs is the
    arrtim = psrchive.ArrivalTime()
    arrtim.set_shift_estimator('PGS')        # Set algorithm (see 'pat -A' help)
```

```python
    arrtim.set_format('Tempo2')

    # Load template profile
    std.pscrunch()
    #std.fscrunch_to_nchan(1)
    arrtim.set_standard(std)

    # Load observation profiles
    obs.pscrunch()
    obs.fscrunch_to_nchan(1)
    arrtim.set_observation(obs)

    # Result is a tuple of TOA strings:
    toas = arrtim.get_toas()
    return toas

def get_toas(std,obs):
    toas_string_tuple=get_toas_string_tuple(std,obs)
    toas=np.array([np.zeros(len(toas_string_tuple)),np.
 ↪zeros(len(toas_string_tuple)) ])
    for a in range(len(toas[0])):
        toas[0][a]=float(toas_string_tuple[a].split()[2]) # Time of arrival
        toas[1][a]=float(toas_string_tuple[a].split()[2]) # Uncertainity
    return toas

def get_residuals(toa_templ,toas):
    residuals=toas
    for a in range(len(toas[0])):
        residuals[0][a]=(toas[0][a]) - ((toa_templ)+(a*TP/86400))
        residuals[1][a]=(residuals[1][a])/86400
    return residuals

def get_differences(std,obs):
    toas=get_toas(std,obs)
    diff_toas=np.zeros(len(toas[0])-1)
    for a in range(len(toas[0])-1):
        diff_toas[a]=toas[0][a+1] - toas[0][a]
    return diff_toas
```

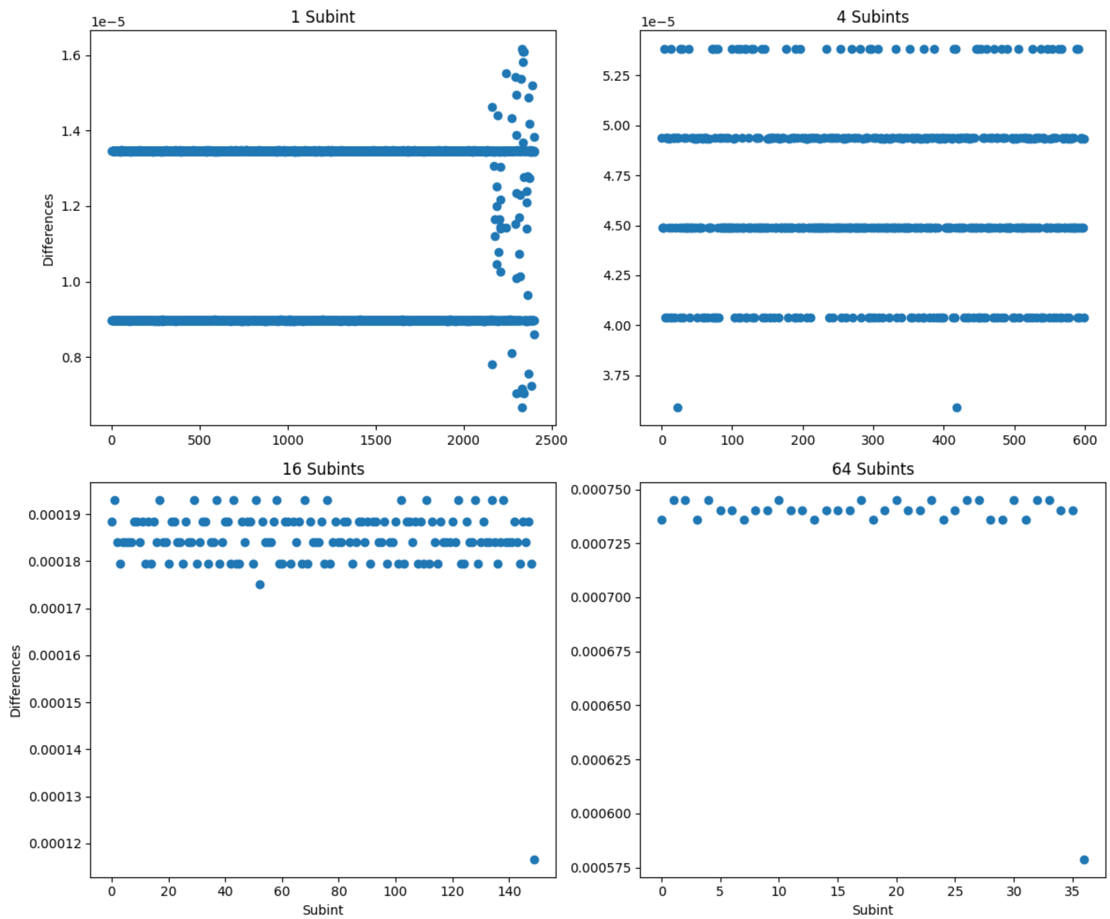
Instead of generating residuals we look at the difference between TOA of pulse in two adjacent subints , because PSRchive Python interface lacks function description for generating residuals.

[457]:
```python
d1=get_differences(Template,Arch1)
d4=get_differences(Template,Arch4)
d16=get_differences(Template,Arch16)
d64=get_differences(Template,Arch64)
```

```
d256=get_differences(Template,Arch256)
d1024=get_differences(Template,Arch1024)
```

[463]:

```
plt.clf()
fig3,axes3 = plt.subplots(2,2,figsize=(12,10))
axes3[0,0].scatter(np.arange(len(d1)),d1)
axes3[0,0].set_title('1 Subint')


axes3[0,1].scatter(np.arange(len(d4)),d4)
axes3[0,1].set_title('4 Subints')


axes3[1,0].scatter(np.arange(len(d16)),d16)
axes3[1,0].set_title('16 Subints')


axes3[1,1].scatter(np.arange(len(d64)),d64)
axes3[1,1].set_title('64 Subints')

#axes3[2,0].scatter(np.arange(len(d256)),d256)
#axes3[2,0].set_title('256 Subints')


#axes3[2,1].scatter(np.arange(len(d1024)),d1024)
#axes3[2,1].set_title('1024 Subints')


axes3[1,0].set_ylabel('Differences')
axes3[0,0].set_ylabel('Differences')
axes3[1,0].set_xlabel('Subint')
axes3[1,1].set_xlabel('Subint')

plt.tight_layout()
plt.show()
```



We see that subint pulse period oscillation between some descrete values. This issue has to be looked into.

3 References :

1. https://psrchive.sourceforge.net/index.shtml
2. https://ui.adsabs.harvard.edu/abs/2004hpa..book.....L
3. http://ipta.phys.wvu.edu/files/student-week-2018/pulsar_timing_lecture.pdf
4. https://youtu.be/1z6asuKtIvw