part3_funcs

March 28, 2025

Module containing plotting functionalities and the algorithm to search over the desired DM and width model parameters.

[ ]:
```python
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal
from sigpyproc.readers import FilReader
```

[1]:
```python
def plot_imshow_transients(
        data,
        title,
        xlabel,
        ylabel,
        cbar_label,
        vrange,
        xlim,
        ylim,
        extent,
        figsize,
        fontsize,
        grid=True
    ):
    """
    Plot an imshow waterfall of transient pulsar data.

    Parameters
    ----------
    data : array-like
        2D array containing the data to be visualized.
    title : str
        The title for the plot.
    xlabel : str
        Label for the x-axis.
    ylabel : str
        Label for the y-axis.
    cbar_label : str
        Label for the colorbar.
    vrange : tuple of two floats
```

```
        The minimum and maximum values for the color scale (vmin, vmax).
    xlim : tuple of two floats
        Limits for the x-axis.
    ylim : tuple of two floats
        Limits for the y-axis.
    extent : tuple of four floats
        The bounding box in data coordinates that the image will fill,␣
↪formatted as (left, right, bottom, top).
    figsize : tuple of two floats
        Size of the figure in inches (width, height).
    fontsize : int
        Base font size used for the title and axis labels.
    grid : bool, optional
        Whether to display a grid on the plot (default is True).

    Returns
    -------
    None
    """

    plt.rcParams['text.usetex'] = True
    fig, ax = plt.subplots(figsize = figsize)
    im = ax.imshow(
        data,
        vmin = vrange[0],
        vmax = vrange[1],
        aspect='auto',
        interpolation = 'nearest',
        extent=extent
        )
    ax.set_title(title, fontsize = fontsize)
    ax.set_ylabel(ylabel, fontsize = fontsize-1)
    ax.set_xlabel(xlabel, fontsize = fontsize-1)
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
    ax.grid(grid)
    cb = plt.colorbar(im)
    cb.set_label(cbar_label, fontsize=fontsize-2)
    plt.show()



def read_data(fil_file):
    """
    A thin wrapper around the Filterbank method to read data
    from a Filterbank object
    """
```

```python
    data = fil_file.read_block(0,
                               fil_file.header.nsamples,
                               fil_file.header.fch1,
                               fil_file.header.nchans
                               )
    return data



def prepare_data(Fil_data_file, clean_rfi=True, downsample=True):
    """
    Thin wrapper around both the RFI-flagging and the downsampling
    routines provided by sigpyproc.

    if clean_rfi and downsample are set to True, returns the
    downsampled and rfi-excised Filterbank object.
    """

    Fil_data = FilReader('./'+Fil_data_file+'.fil')

    if clean_rfi:
        _, chan_mask = Fil_data.clean_rfi(method='mad', threshold=3)
        Fil_data_masked = FilReader('./'+Fil_data_file+'_masked.fil')

    if downsample:
        Fil_data_masked.downsample(tfactor=32)
        Fil_data_masked_32 = FilReader('./'+Fil_data_file+'_masked_f1_t32.fil')

    return Fil_data_masked_32



def DM_width_search(signal_data, blank_sky, DM_range, width_range,␣
 ↪cand_threshold, normalize=True):
    """
    Search over a range of dispersion measures (DM) and Gaussian kernel widths␣
 ↪to identify candidate transients.

    This function processes signal and blank sky data by optionally normalizing␣
 ↪them, then dedispersing over a
    given range of DM values. For each DM, it computes the time series by␣
 ↪summing over the frequency axis and then
    convolves these time series with a Gaussian kernel over a range of widths.␣
 ↪For each (DM, width) pair, it evaluates
    a candidate threshold based on the ratio of the maximum convolved signal to␣
 ↪the maximum convolved noise. If the
```

```
    candidate exceeds the specified threshold, the signal-to-noise ratio (SNR) is computed and stored in a 2D array.

    Parameters
    ----------
    signal_data : object
        Data object for the signal, which must have a `normalise` method, a `dedisperse` method, and a `data` attribute
        containing a 2D NumPy array.
    blank_sky : object
        Data object for the blank sky noise, with similar methods and attributes as `signal_data`.
    DM_range : array-like
        1D array of dispersion measure values (e.g., in pc cm ³) over which to perform the search.
    width_range : array-like
        1D array of Gaussian kernel widths to apply during the convolution step.
    cand_threshold : float
        Threshold for candidate selection; candidates are considered only if the ratio of the maximum convolved signal
        to the maximum convolved noise exceeds this value.
    normalize : bool, optional
        If True, both `signal_data` and `blank_sky` are normalized before dedispersion. Default is True.

    Returns
    -------
    DM_width_search : ndarray
        A 2D array of shape (len(DM_range), len(width_range)) containing the computed SNR values for each (DM, width)
        pair that passed the candidate threshold (entries remain zero if the threshold is not met).
    max_DM : float
        The dispersion measure from `DM_range` corresponding to the maximum SNR found.
    max_width : float
        The Gaussian kernel width from `width_range` corresponding to the maximum SNR found.
    """


    DM_width_search = np.zeros((DM_range.shape[0], width_range.shape[0]))
    print(DM_width_search.shape)

    if normalize:
        signal_data = signal_data.normalise()
```

```
        blank_sky = blank_sky.normalise()

    for i, DM in enumerate(DM_range):

        signal_dd = signal_data.dedisperse(DM)
        blank_sky_dd = blank_sky.dedisperse(DM)

        signal_dd_tseries = np.sum(signal_dd.data, axis=0)
        blank_sky_dd_tseries = np.sum(blank_sky_dd.data, axis=0)

        for j, width in enumerate(width_range):
            gauss_kernel = signal.windows.gaussian(400, width)

            signal_convolution = signal.convolve(signal_dd_tseries,␣
↪gauss_kernel)
            blank_sky_convolution = signal.convolve(blank_sky_dd_tseries,␣
↪gauss_kernel)

            max_convolved_signal = np.max(signal_convolution)
            max_convolved_noise = np.max(blank_sky_convolution)

            blank_sky_noise = np.std(blank_sky_convolution)/np.sqrt(width)

            candidate_thresh = (max_convolved_signal/max_convolved_noise)

            if candidate_thresh > cand_threshold:
                SNR = max_convolved_signal / blank_sky_noise
                DM_width_search[i, j] = SNR

    #search DM and width to find max values
    max_DIM_width_inds = np.argmax(DM_width_search)
    max_DM_ind, max_width_ind = np.unravel_index(max_DIM_width_inds,␣
↪DM_width_search.shape)
    max_DM, max_width = DM_range[max_DM_ind], width_range[max_width_ind]

    return DM_width_search, max_DM, max_width



class SignalPlotter:
    def __init__(self, signal_data, blank_sky, DM, width, norm=False):
        """
        Initialize a SignalPlotter instance by processing the input signal and␣
↪blank sky data.

        This constructor optionally normalizes the provided data, then␣
↪dedisperses it using the specified dispersion measure (DM).
```

```
        It computes the original time series by summing over the frequency␣
↪axis, applies a Gaussian convolution with the specified width
        to both the signal and blank sky time series, and calculates the␣
↪signal-to-noise ratio (SNR) time series. Additionally, it
        updates Matplotlib's default font size to 16 for subsequent plots.

        Parameters
        ----------
        signal_data : object
            The signal data object, which must implement a `normalise` method,␣
↪a `dedisperse` method, and contain a `data` attribute
            (a 2D NumPy array).
        blank_sky : object
            The blank sky noise data object, with the same methods and␣
↪attributes as `signal_data`.
        DM : float
            The dispersion measure to be used for dedispersing the data.
        width : float
            The width parameter for the Gaussian kernel used in the convolution.
        norm : bool, optional
            If True, normalize both the signal and blank sky data before␣
↪dedispersion. Default is False.

        Returns
        -------
        None
        """

        self.signal_data = signal_data
        self.blank_sky = blank_sky

        if norm:
            # Normalize the data
            self.signal_data = self.signal_data.normalise()
            self.blank_sky = self.blank_sky.normalise()

        # Dedisperse the data
        self.signal_dd = self.signal_data.dedisperse(DM)
        self.blank_sky_dd = self.blank_sky.dedisperse(DM)

        # Compute original time series by summing over frequency axis
        self.signal_dd_tseries = np.sum(self.signal_dd.data, axis=0)
        self.blank_sky_dd_tseries = np.sum(self.blank_sky_dd.data, axis=0)

        # Create a Gaussian kernel and convolve with the time series (using␣
↪mode='same' for matching lengths)
```

```
        gauss_kernel = signal.windows.gaussian(400, width)
        self.signal_convolution = signal.convolve(self.signal_dd_tseries,␣
↪gauss_kernel, mode='same')
        self.blank_sky_convolution = signal.convolve(self.blank_sky_dd_tseries,␣
↪gauss_kernel, mode='same')

        # Compute the SNR time series:
        self.SNR = self.signal_convolution / (np.std(self.
↪blank_sky_convolution) / np.sqrt(width))
        plt.rcParams.update({'font.size': 16})  # Set default font size to 14

    def plot_original_time_series(self):
        """
        Plot the original (dedispersed) time series for both the signal and␣
↪blank sky.
        """
        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        axes[0].plot(self.signal_dd_tseries)
        axes[0].set_title("Signal")
        axes[0].set_xlabel("Time (samples)")
        axes[0].set_ylabel("Intensity (counts)")
        axes[0].grid(True)

        axes[1].plot(self.blank_sky_dd_tseries)
        axes[1].set_title("Blank Sky")
        axes[1].set_xlabel("Time (samples)")
        axes[1].set_ylabel("Intensity (counts)")
        axes[1].grid(True)

        plt.tight_layout()
        plt.show()

    def plot_convolved_time_series(self):
        """
        Plot the convolved time series for both the signal and blank sky.
        """
        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        axes[0].plot(self.signal_convolution)
        axes[0].set_title("Signal Convolved with Gaussian Kernel")
        axes[0].set_xlabel("Time (samples)")
        axes[0].set_ylabel("Convolved Intensity (counts)")
        axes[0].grid(True)

        axes[1].plot(self.blank_sky_convolution)
        axes[1].set_title("Blank Sky Convolved with Gaussian Kernel")
```

```
        axes[1].set_xlabel("Time (samples)")
        axes[1].set_ylabel("Convolved Intensity (counts)")
        axes[1].grid(True)

        plt.tight_layout()
        plt.show()

    def plot_snr(self, use_log_scale=False):
        """
        Plot the SNR time series and its histogram.

        Parameters:
            use_log_scale (bool): If True, sets a logarithmic scale for the␣
↪histogram's y-axis.
        """
        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        # SNR time series plot
        axes[0].plot(self.SNR)
        axes[0].set_title("SNR Time Series (Optimal DM \\& width)")
        axes[0].set_xlabel("Time (samples)")
        axes[0].set_ylabel("SNR")
        axes[0].grid(True)

        # SNR histogram
        axes[1].hist(self.SNR, bins=100)
        if use_log_scale:
            axes[1].set_yscale('log')
            axes[1].set_ylabel("Counts (log scale)")
        else:
            axes[1].set_ylabel("Counts")
        axes[1].set_title("SNR Histogram (Optimal DM \\& width)")
        axes[1].grid(True)
        axes[1].set_xlabel("SNR")

        plt.tight_layout()
        plt.show()

    def plot_calibrated_pulse(self, transfer_function, title):
        """
        Use the transfer function to covert to Janskys and make
        a plot of the dedispersed pulse at the DM  that maximizes
        the SNR.

        Parameters
        ----------
        transfer_function (array-like):
```

```
            Array allowing the conversion from counts per frequency to Jansky's␣
↪per frequency.
        title (string):
            The title of the plot.

        Returns
        -------
        None
        """

        #fix the nans in the transfer function
        transfer_function = np.nan_to_num(transfer_function, nan=0.0, posinf=0.
↪0, neginf=0.0)

        #use the transfer function to convert to Janskys
        calibrated_signal = self.signal_dd.data*transfer_function[:, None]

        calibrated_signal = np.mean(calibrated_signal, axis=0)

        #plot the dedispersed pulse
        fig, ax = plt.subplots(figsize = (8, 6))
        ax.plot(calibrated_signal)
        ax.set_title(title, fontsize = 16)
        ax.set_xlabel('Freq (samples)', fontsize = 15)
        ax.set_ylabel('Flux Density(Jy)', fontsize = 15)
        ax.grid(True)
        plt.show()
```

[ ]:

```
```