furs

Shikhar Mittal

Mar 08, 2024

CONTENTS:

CHAPTER ONE

BASICS

Name
Foregrounds due to Unresolved Radio Sources

Author
Shikhar Mittal

Homepage
https://github.com/shikharmittal04/furs

1.1 Why do you need this code?

Use this code to generate the Foregrounds due to Unresolved Radio Sources (FURS).

A cosmological global 21-cm signal hides under foregrounds due to galactic and extragalactic emissions. These foregrounds can easily be 4 to 5 orders of magnitude higher than the signal of interest. For a reliable inference it is important to accurately model these foregrounds. While we have a reasonable understanding of galactic emission (typically fit as log-log polynomial), we do not understand the extragalactic contributions. Based on existing models, this code models the foregrounds due to unresolved extragalactic radio sources.

Read more about it in the paper Mittal et al (2024).

1.2 Installation and requirements

This package can be installed as

```
pip install furs
```

It is recommended to work on a Python version > 3.8. Packages required are

- numpy
- scipy
- matplotlib
- mpi4py
- healpy
- transformcl

1.3 Quick start

The code is run in two main steps:

- Assign the unresolved sources flux densities (at a chosen reference frequency) and spectral indices.
- Then generate the sky maps at desired frequencies of observation.

The following code captures the main functionalities of this package.

```
from furs import furs

#Step-1 initialise the object with default settings
obj = furs.furs()

#Step-2 generate the data at the reference frequency
obj.ref_freq()

#Step-3 generate the sky maps at multiple frequencies as well as their sky average
obj.gen_freq()

#Step-4 finally, generate a sky averaged spectrum vs frequency figure
obj.visual()
```

Save the above code as (say) `eg_script.py` and run it as

```
python eg_script.py
```

Running the code will generate several files. The terminal messages will guide you to these output files. The most important of all files of your interest will be `Tb_nu_map.npy`. However, you may never have to deal with them yourself. To visualise your outputs use the function `visual()`. Read on to see the available features for `visual()`.

The default values have been chosen such that the above script can be run on a PC. Since modern PCs have at least 4 cores, for a better performance one could also run the code as

```
mpirun -np 4 python eg_script.py
```

However, in general and for more realistic flux density ranges and high resolution maps, it is recommended to run the code on HPCs.

1.4 License and citation

The software is free to use on the MIT open source license. If you use the software for academic purposes then we request that you cite the Mittal et al (2024).

CHAPTER TWO

DETAILED EXPLANATION

2.1 Initialisation

`furs.furs()` initialises the class object with default settings. There are total 10 available optional arguments as follows:

1. `nu_o`
   - reference frequency, $nu_0$ (Hz)
   - type *float*
   - default **150e6**
2. `beta_o`
   - mean spectral index, $beta_0$
   - type *float*
   - default **2.681**
3. `sigma_beta`
   - spread in the Gaussian distribution of spectral index, $sigma_beta$
   - type *float*
   - default **0.5**
4. `amp`
   - amplitude of the power-law 2-point angular correlation function (2PACF), $A$. If you want to model a Poissonian distributed sky, set this parameter to 0.
   - type *float*
   - default **7.8e-3**
5. `gam`
   - negative of the exponent of the 2PACF, $gamma$
   - type *float*
   - default **0.821**
6. `logSmin`
   - $log_{10}(S_{mathrm{min}})$, where $S_{mathrm{min}}$ is in Jansky (Jy)
   - type *float*

  - default **-2.0**

7. `logSmax`
   - $log_{10}(S_{mathrm{max}})$, where $S_{mathrm{max}}$ is in Jansky (Jy)
   - type *float*
   - default **-1.0**
8. `dndS_form`
   - sum of 2 double inverse power laws (0), 7th order log-log polynomial (1) or 5th order log-log polynomial (2)
   - type *int*
   - default **0**
9. `path`
   - path where you would like to save all output files
   - type *string*
   - default **‘’**
10. `log2Nside`
    - Number of divisions of each side of the pixel for `HEALPix` maps in units of log_2
    - type *int*
    - default **6**
11. `lbl`
    - This is an additional label that you may want to append to the output files
    - type *string*
    - default **‘’**

Thus, if you want to chose a different set of parameters, you must initialise the object as

```
obj = furs.furs(log2Nside=6, logSmin=-2,logSmax=-1,dndS_form=0, nu_o=150e6, beta_o=2.681,
sigma_beta=0.5, amp=7.8e-3,gam=0.821, path='', lbl='')
```

(Replace the above values by values of your choice.)

2.2 Reference frequency

The function `ref_freq` does 3 tasks:-

- Calcuates the total number of unresolved sources corresponding to your specified `logSmin` and `logSmin`.
- Creates a ‘clustered’ sky density of unresolved radio sources, fluctuation for which follows the 2PACF whose parameters are set by `amp` and `gam`.
- Finally, it visits each pixel on the sky and assigns each source a flux density chosen from a flux distribution function, :math: *\mathrm{d}n/\mathrm{d}S* and a spectral index which is normally distributed. The normal distribution is set by `beta_o` and `sigma_beta`.

Sky pixelisation is set by `log2Nside`. The number of pixels is :math: *N_{\mathrm{pix}} = 12 \times 2^{2k}*, where :math: *k*= `log2Nside`.

The function does not return anything, but produces 4 output files, namely `n_clus.npy`, `Tb_o_individual.npy`, `Tb_o_map.npy`, and `beta.npy` in the path specified by `path` during initialisation. The files are described below.

1. `n_clus.npy` is a 1D array which stores number density of unresolved radio sources as number per pixel. `n_clus[i]` gives the number of sources on the :math: *i^{\mathrm{th}}* pixel, where :math: *i=0,1,\ldots,N_{\mathrm{pix}}-1*. Note that in general `n_clus[i]` will not be a natural number; we simulate for a rounded-off value.
2. Both `Tb_o_individual.npy` and `beta.npy` are array of arrays of unequal sizes and share equal amount of memory. Typically, these files will be huge (for default settings they will of size ~ 17 MB each). Each of `Tb_o_individual[0]`, `Tb_o_individual[1]`, . . . , is an array and they are total $N_{mathrm{pix}}$ in number corresponding to :math: *N_{\mathrm{pix}}* pixels. The last array is `Tb_o_individual[Npix-1]`. The length of array `Tb_o_individual[i]` is equal to the number of sources on the $i^{mathrm{th}}$ pixel, which is `round(n_clus[i])`. The values itself are the brightness temperature contributed by each source in kelvin at reference frequency.
3. The structure of `beta` is same as `Tb_o_individual`. The values itself are the spectral indices assigned to each source.
4. `Tb_o_map` is an array similar in structure to `n_clus`. It is the pixel wise brightness temperature contributed by the extragalactic radio sources at the reference frequency. Thus, `Tb_o_map[i] = numpy.sum(Tb_o_individual[i])`.

2.3 General frequency

The next important task is performed by the function `gen_freq`. It scales the brightness temperature at reference frequency for each source according to a power law to a desired range of frequencies. The desired frequencies should be supplied (in Hz) as a numpy array to this function. For example

```
obj.gen_freq(nu = 1e6*numpy.arange(50,201))
```

The default value is as given in the above command. This function does not return anything but produces 3 files namely `Tb_nu_map.npy`, `Tb_nu_glob.npy`, and `nu_glob.npy` in the path specified by `path` during initialisation. The files are described below.

1. `Tb_nu_map` is a 2D array of shape :math: *N_{\mathrm{pix}}\times N_{\nu}*, so that `Tb_nu_map[i,j]` gives the brightness temperature on the :math: *i^{\mathrm{th}}* pixel at `nu[j]` frequency. :math: *N_{\nu}* is the number of frequencies you gave in the argument of `gen_freq()`.
2. `Tb_nu_glob` is derived directly from `Tb_nu_map`. It is the sky average of the map at each frequency and is thus a 1D array. It is calculated as `Tb_nu_glob = numpy.mean(Tb_nu_map,axis=0)`.
3. `nu_glob.npy` is simply the frequency array you gave else it is the default value.

Note that this function loads `Tb_o_individual.npy` and `beta.npy`. These files can easily be 10s of GB in size for 'realistic' `logSmin` and `logSmax`. Common personal computers have ~ 4 GB RAM. It is thus recommended to run this code on supercomputers. For job submission scipt users are requested to specify `#SBATCH --mem-per-cpu=[size in MB]`, where a recommendation for `size in MB` will be printed by `ref_freq()` function.

2.4 Chromatic distortions

`Tb_nu_map` and hence `Tb_nu_glob` so generated do NOT account for chromatic distortions. They are simply the model outputs for foregrounds due to unresolved radio sources. However, in reality because of the chromatic nature of the antenna beam the actual foregrounds spectrum registered will be different. You can use the function `chromatisize()` to account for the chromaticity.

Since this is experiment specific you will need to provide an external data file: the beam directivity pattern, :math: *D*. This should be a 2D array of shape :math: *N_{\mathrm{pix}}\times N_{\nu}*, such that `D[i,j]` should give the beam directivity at $i^{mathrm{th}}$ pixel at nu[j] frequency. The frequencies at which you generate your data :math: *D* should be the same as the frequencies you gave in `gen_freq()`. (In case you forgot, `gen_freq()` will have saved the frequecy array in your `obj.path` path.) Put this array :math: *D* in your `obj.path` path by the name of `D.npy`.

Only after running `ref_freq` and `gen_freq`, run `chromatisize` as

```
obj.chromatisize()
```

No input argument is required. The return value is `None`. This function will generate a file called `T_data.npy` in your path. This will be a 1D array with length of number of frequencies.

2.5 Visualisation

The final part of the code is to visualise the results. Main data for inspection is in the file `Tb_nu_map.npy`. Each of `Tb_nu_map[:,j]` is an array in the standard ring ordered `HEALPix` format and is thus ready for visualisation as a Mollweide projection. You may also be interested in inspecting the global spectrum of extragalactic emission, i.e, temperature as a function of frequency. This is simply the data in the file `Tb_nu_glob.npy` generated by `gen_freq()`.

You may use the function `visual()` for both the above purposes. It is possible to make several other additional figures by simply setting the optional arguments to `True` (see below). This function is again a method of class object `furs` and is thus called as

```
obj = furs.furs()
obj.visual()
```

The following optional arguments are available for this function:-

1. `nu_skymap`
    - the frequency at which you want to produce a Mollweide projection of extragalactic foregrounds
    - type *float*
    - default `nu_o`
2. `t_skymap`
    - Create a sky map of extragalactic foregrounds?
    - type *bool*
    - default `False`
3. `n_skymap`
    - Create a sky map of number density of unresolved radio sources?
    - type *bool*
    - default `False`

4. `dndS_plot`
    - Plot the S distribution function?
    - type *bool*
    - default `False`
5. `aps`
    - Plot the angular power spectrum?
    - type *bool*
    - default `False`
6. `spectrum`
    - Create the foreground spectrum?
    - type *bool*
    - default `True`
7. `chromatic`
    - To the spectrum figure add the sky data curve which accounts for beam chromaticity?
    - type *bool*
    - default `False`
8. `xlog`
    - Set x-axis in log scale? This and the next option are relevant only for the spectrum plot.
    - type *bool*
    - default `False`
9. `ylog`
    - Set y-axis in log scale?
    - type *bool*
    - default `True`
10. `fig_ext`
    - Choose your format of figure file; popular choices include `pdf`, `jpeg`, `png`
    - type *string*
    - default `pdf`

This function will produce figures in the path specficied during initialisation.

CHAPTER

THREE

OTHER FUNCTIONS

There are 5 additional useful methods of the class `furs`. These are:-

1. `acf(chi)`
    - returns the 2PACF, $C(chi)$
    - requires one argument, the angle $chi$ in radians; can be a number or an array
    - output is a dimensionless quantity
2. `dndS(S)`
    - returns flux distribution, $mathrm{d}n/mathrm{d}S$. The functional form will be according to your choice for `dndS_form` you gave during initialisation. Default is 0.
    - requires one argument, the flux density $S$ in Jy; can be a number or an array
    - output is in units of number per unit flux density per unit solid angle, i.e. $mathrm{Jy^{-1}sr^{-1}}$
3. `num_den()`
    - returns the clustered number density as number per pixel, $n_{mathrm{clus}}$
    - no arguments required
    - output is an array of length $N_{mathrm{pix}}$
4. `num_sources()`
    - returns the total number of unresolved extragalactic radio sources for the full sky, $N_{mathrm{s}}$
    - no arguments required
    - output is a pure number
5. `print_input()`
    - If you want to print the all raw parameter values you gave, you may use this function to print them
    - no arguments required
    - no return value

Example usage: to find the number of sources between $10^{-6}$ and $10^{-1}mathrm{Jy}$ do

```
obj = furs(logSmin=-6,logSmax=-1)
Ns = obj.num_sources()
```

CHAPTER

FOUR

GENERAL REMARKS

Users do not have to run `ref_freq()` everytime. If they want to use the same data for source distribution (`n_clus.npy`), flux density (`Tb_o_individual.npy`) and spectral index (`beta.npy`) assignments at reference frequency to generate spectrum and sky maps for a different frequency range, then run only `gen_freq()` for a new choice of `nu`.

Similarly, if you have already run `gen_freq()` and are happy with the specifications of the model then you can directly jump to the `visual()` function.

In case you forgot what data set you generated with what specifications, you can always save your class object using the function `save_furs(class_object,'file_name.pkl')` in the directory where all other outputs are saved and load back using `load_furs`. (Both functions are part of module `furs.py`.)

Thus, after initialising your class object (i.e. `obj = furs([YOUR SPECIFICATIONS])`), you can add to your script

```
furs.save_furs(obj,'myobj.pkl')
```

So when you came back next time you can load it as

```
obj=furs.load_furs('/give/full/path/to/myobj.pkl')
```

You can check that indeed the specfications are correctly loaded by printing them via command `obj.print_input()`.