Response Tools Data Guide v1

This is a guide for the Github repository: response-tools

Setting up the code

Downloading the code

The first step will likely be installing the Python code. The repository, `response-tools`, is very lightweight and does not require many dependencies. You can either:

- Do the usual to download the repository contents and retain Git version control via
  - **`git clone https://github.com/foxsi/response-tools.git`**
- Or just download the repository contents as a ZIP folder and extract the contents
  - Click the green Code button then select “Download ZIP”

Python virtual environments

It is still recommended to create a Python virtual environment in your preferred manner to house the response code; however, this is completely your own choice.
I will quickly run through an example of creating a virtual environment using conda:

- **`conda create -n response-tools-env python==3.12`**

(recommend using Python 3.12 just because this has been proven to be stable with the software, feel free to test other versions if you like).
Remember to activate your environment with:

- **`conda activate response-tools-env`**

You can check you’re working out of your environment by running:

- **`which python`**
- **`which pip`**

And making sure the environment is in the returned path (I get something like `/Users/kris/miniconda3/envs/response-tools-env/bin/python`). Note the environment name should be in the path.

Installing the code

The Python code can then be installed with:

- **`pip install -e .`**

while in the directory the `response-tools` directory that contains the `setup.py` file.
Any time the code is updated and, say, you pull/download it from Github, make sure to perform the `pip install -e .` line from above again in your environment . Most of the time this is unnecessary but ensures any new changes, updated versions, etc. definitely take effect.

Downloading response files

Where the data lives

Public access to the FOXSI-4 flight data is provided at the site: **https://foxsi.space.umn.edu/data/response/response-components/**.

That link contains several subfolders of response data:

- Attenuation data
- Detector response data
- Effective area data
- Quantum efficiency data

Users can also access the data via FTP at **foxsi.space.umn.edu/FOXSI**. There is no username or password required. Once connected to the FTP server, navigate to **data/response/response-components** to access response files.

Note: users on the UMN eduroam WiFi network will not be able to access the foxsi.space.umn.edu/data site or the FTP server. All other networks tested have been able to access without issue.

Python interface to pull data

The following will download all the response files to your machine:

```python
import response_tools.io.fetch_response_data as fetch
fetch.foxsi4_download_required(verbose=True)
# stop trying to make "fetch" happen
```

The functions in the package are designed to know where these function are and just use them as expected.

As stated in the Even lower level handling section, to find where the files are stored in your filesystem, you can use the global package variable, `responseFilePath`:

```python
import response_tools
print(response_tools.responseFilePath)
```

This allows a user to easily find where the response files are for their own purposes.

The code and how to use it

The code is written in Python and is relatively bare-bones. There are two things that have been made use of that the user might want to know:

1. Values and functions in the code have been made to be *unit aware* where possible.
    - When it comes to response elements, the biggest annoyance is units and keeping track of them. The code makes use of Astropy Units to help the user track what they're working with and to avoid making mistakes.
2. Functions return custom data-class objects, not just tuples of values or arrays.
    - To find out what fields are stored in the data-class, all you have to do is either print the returned `object`, inspect `object.contents`, or run the method `object.print_contents`. (Note for IDL users: Using these objects is similar to using an IDL structure.)
    - The data-class object the functions returned can be accessed by either indexing with the field name à la `["field_name"]` or doing `.field_name` on the returned object.

Remember, with any function you can always run `help(function_name)` and this should let you inspect the documentation written for that function and should explain what the function is, how to use it, and what it returns.

Design and user needs

The Python code is designed to have three API levels:

- **Level 3 (recommended use):** Full telescope loaders for the FOXSI(-4) instrument. These are designed to return the ancillary response function (ARF) and redistribution matrix function (RMF) composed of all the relevant telescope components.
    - These can be found in the `responses` module.
    - E.g., the function `foxsi4_telescope2_arf` will return the ARF data for FOXSI-4 telescope 2.
- **Level 2 (recommended use):** Loader functions that are named after their position in the FOXSI(-4) payload. All of these functions start with `foxsi4_position#` where `#` refers to the telescope position to which the response product belongs.
    - These can be found in the `telescope_parts` module.
    - E.g., the function `foxsi4_position2_optics` will return the optics information relating to position 2 using the predetermined and appropriate function from the `effective_area` module.
- **Level 1:** Loader functions that have as little as possible description to its place in the FOXSI(-4) instrument, relying on its own description.
    - These can be found in modules like `attenuation`, `detector_response`, `effective_area`, and `quantum_efficiency`.

Even lower level handling

A user can decide in which way they would like to interact with the response products. A potential **Level 0** would be the user interacting with the files themselves directly. Depending

on how the package is installed the files may be stored in different locations on your machine.
To find where the files are stored in the package, and indeed your local machine, you can use the global package variable, `responseFilePath`:

```python
import response_tools
print(response_tools.responseFilePath)
```

The functions in the package use this path to automatically find the correct files.

Code example

There is a place for Python code examples in the codebase itself here. For a list of possible functions to use, see the What exists section.
The example here will just show briefly how to use the code and follows the specific examples already in the repository (`functions_and_outputs.py` and `plot_arf_rmf_drm.py`).
We'll stick with **Level 3** and use FOXSI-4's Telescope 2 as an example. To start, let's import the correct module and give ourselves the use of Numpy and Astropy Units:

```python
import astropy.units as u
import numpy as np

import response_tools.responses as responses
```

We can then choose to, say, obtain the ARF of a given FOXSI-4 telescope by first specifying the energies (`mid_energies`) we want the ARF to be evaluated at and the off-axis angle for any optic component (`off_axis_angle`):

```python
# energies from 4 to 20 keV in increments of 0.5 keV
mid_energies = np.arange(4, 20, 0.5) << u.keV
off_axis_angle = 0 << u.arcmin

tel2_arf = responses.foxsi4_telescope2_arf(mid_energies=mid_energies,
                                           off_axis_angle=off_axis_angle)
```

Inspecting the data-class object

Let's see what the output, `tel2_arf`, has in it. We can make use of a number of methods here:

Method/Field Example	Description
`tel2_arf.fields`	Returns a list of all the fields contained in the data-class.

`tel2_arf.contents`	Returns the data-class contents in a dictionary format.
`tel2_arf.print_contents`	This should *nicely* print the output of `tel2_arf.contents` and make it easier to read.
`tel2_arf.function_path`	Tracks the functions used to produce the output.
`tel2_arf.filename`	The file name+path source of the output.

So, using any of the above will give an idea of what's in `tel2_arf`. To access specific values/arrays/information, we can enquire on the data-class object directly.
For example, we consider the field **"filename"** that should be present in *all* data-classes used in this package and we'll give the fake value of **"really-cool-file.fits"**. We can access the fields in three different ways: 2 are normal and recommended, 1 way is not recommended but still fine:

```python
fn = tel2_arf.filename # recommended
fn = tel2_arf["filename"] # recommended
fn = tel2_arf.contents["filename"] # not recommended
```

All three of the examples will set `fn` equal to **"really-cool-file.fits"**.
A useful field for tracking where your result came from can be found in the `function_path` field. This will explain the paths the data-class has taken as it's run through different functions. E.g., if the file is loaded by `function1` which is then used in `function2` the `function_path` field will be `"function1\n->function2"`.
There are multiple different specialised data-class objects but they all start with the `filename` and `function_path` fields. However, the other fields are well named and should be fairly easy to understand.

Data-class fields

Of course, other fields exist in more specific data-classes (e.g., for effective areas or transmissions) that do not exist in *every* data-class output. These fields will only appear in the relevant data-classes:

Field Example	Description
`mid_energies`	A unit aware array (likely in keV) of the energies at which the data was evaluated.
`off_axis_angle`	A unit aware value (likely in arc-minutes) of the off axis angle at which the data was evaluated.
`times`	A unit aware list (likely in seconds) of the times at which the data was evaluated.

`model`	A boolean indicating whether the returned data is from a model (`True`) or measurement (`False`).
`response`	A unit aware array, likely the combination of several telescope components to make part or the whole of a telescope's response.
`input_energy_edges` or `output_energy_edges`	A unit aware array of the bin edges of a matrix response. The "input" edges referring to the incoming photon energies and the "output" to the exiting detector channels.
`transmissions`	A unit aware array (likely dimensionless) of the attenuator's transmission.
`attenuation_type`	A descriptive label for the attenuator.
`detector_response`	A unit aware matrix (e.g., counts/photon or DN/photon) describing a detector's photon-to-observable probability.
`detector`	A descriptive label for the detector.
`effective_areas`	A unit aware array (likely cm^2) of the optic's effective area.
`optic_id`	A descriptive label for the optic.
`quantum_efficiency`	A unit aware array (likely dimensionless) of the detector's quantum efficiency.
`response_type`	Describes whether the response in the data-class is an ARF, RMF, or DRM.
`telescope`	Details the telescope to which the data, or components of the data, belongs.
`elements`	A tuple of the data-classes used to create the one containing this field.

These fields should make it extra clear to a user exactly what they are handling when it comes to the output of the function.
If you're ever unsure as to which fields exist in your returned object, you can check using the fields field as described in Inspecting the data-class object.

Plotting fields from the object

After you have your object(s), you might want to plot the contents for visual inspection. We can use the information in the data-class object to avoid a lot of manual bookkeeping jobs like tracking units.

```python
import matplotlib.pyplot as plt

plt.figure()
plt.plot(tel2_arf.mid_energies,
         tel2_arf.response)
plt.xlabel(f"Energy [{tel2_arf.mid_energies.unit:latex}]")
plt.ylabel(f"Response [{tel2_arf.response.unit:latex}]")
plt.title(f"{tel2_arf.telescope}:{tel2_arf.response_type}")
plt.show()
```

Real working example

For a real working example, let's assume you're after the full response for the telescope 2 (we're really walking through the plot_arf_rmf_drm.py example). For this, we'll want the ARF and the RMF before we combine them into one product.
The ARF can be evaluated at any number of energies but the ones we're interested in are the energies specific to the detector's RMF.
Therefore, it makes sense to work out the telescope 2 RMF first, extract the energies we're interested in (mid-points of the RMF energy bins), then evaluate the ARF before combining them together:

```python
import astropy.units as u
import numpy as np

import response_tools.responses as responses

# set up the ARF with the RMF information then make the DRM
off_axis_angle = 0 << u.arcmin
pos_rmf = responses.foxsi4_telescope2_rmf(region=0)
mid_energies = (pos_rmf.input_energy_edges[:-1]\
                +pos_rmf.input_energy_edges[1:])/2
pos_arf = responses.foxsi4_telescope2_arf(mid_energies=mid_energies,
                                          off_axis_angle=off_axis_angle)
pos_drm = responses.foxsi4_telescope_response(pos_arf, pos_rmf)
```

We can then plot the results as we normally would:

```python
from matplotlib.colors import LogNorm
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt

# define the plot
fig = plt.figure(figsize=(18, 5))
gs = gridspec.GridSpec(1, 3)
```

```
# the ARF info
gs_ax0 = fig.add_subplot(gs[0, 0])
gs_ax0.plot(pos_arf.mid_energies, pos_arf.response)
gs_ax0.set_xlabel(f"Photon Energy [{pos_arf.mid_energies.unit:latex}]")
gs_ax0.set_ylabel(f"Response [{pos_arf.response.unit:latex}]")
gs_ax0.set_title(f"Telescope 2: ARF")

# the RMF info
gs_ax1 = fig.add_subplot(gs[0, 1])
r = gs_ax1.imshow(pos_rmf.response.value,
                  origin="lower",
                  norm=LogNorm(vmin=0.001),
                  extent=[np.min(pos_rmf.output_energy_edges.value),
                          np.max(pos_rmf.output_energy_edges.value),
                          np.min(pos_rmf.input_energy_edges.value),
                          np.max(pos_rmf.input_energy_edges.value)]
                  )
cbar = plt.colorbar(r)
cbar.ax.set_ylabel(f"Response [{pos_rmf.response.unit:latex}]")
gs_ax1.set_xlabel(f"Count Energy
[{pos_rmf.output_energy_edges.unit:latex}]")
gs_ax1.set_ylabel(f"Photon Energy
[{pos_rmf.input_energy_edges.unit:latex}]")
gs_ax1.set_title(f"Telescope 2: RMF")

# the DRM info
gs_ax2 = fig.add_subplot(gs[0, 2])
r = gs_ax2.imshow(pos_drm.response.value,
                  origin="lower",
                  norm=LogNorm(vmin=0.001),
                  extent=[np.min(pos_drm.output_energy_edges.value),
                          np.max(pos_drm.output_energy_edges.value),
                          np.min(pos_drm.input_energy_edges.value),
                          np.max(pos_drm.input_energy_edges.value)]
                  )
cbar = plt.colorbar(r)
cbar.ax.set_ylabel(f"Response [{pos_drm.response.unit:latex}]")
gs_ax2.set_xlabel(f"Count Energy
[{pos_drm.output_energy_edges.unit:latex}]")
gs_ax2.set_ylabel(f"Photon Energy
[{pos_drm.input_energy_edges.unit:latex}]")
gs_ax2.set_title(f"Telescope 2: DRM")

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

The above will produce a figure showing the ARF, RMF, and the DRM all together in the one plot.

What exists

To access any of the functions listed below, simply import the response-tools code and then you can use the _dot_ notation to work down to the function you desire. E.g., if we wanted to access the `foxsi4_telescope_response` function in the responses.py module to find out what it does:

```python
import response_tools.responses as resp
help(resp.foxsi4_telescope_response)
```

The above should let you inspect the doc-string for the `foxsi4_telescope_response` function, detailing what the function is, how the function can be used, and what it returns.

responses.py **(level 3, recommended use)**

- foxsi4_telescope_response
- foxsi4_telescope2_arf
- foxsi4_telescope2_flight_arf
- foxsi4_telescope2_rmf
- foxsi4_telescope3_arf
- foxsi4_telescope3_flight_arf
- foxsi4_telescope3_rmf
- foxsi4_telescope4_arf
- foxsi4_telescope4_flight_arf
- foxsi4_telescope4_rmf
- foxsi4_telescope5_arf
- foxsi4_telescope5_flight_arf
- foxsi4_telescope5_rmf

telescope_parts.py **(level 2, recommended use)**

- foxsi4_position2_thermal_blanket
- foxsi4_position2_optics
- foxsi4_position2_uniform_al
- foxsi4_position2_detector_response
- foxsi4_position3_thermal_blanket
- foxsi4_position3_optics
- foxsi4_position3_al_mylar
- foxsi4_position3_pixelated_attenuator
- foxsi4_position3_detector_response
- foxsi4_position4_thermal_blanket
- foxsi4_position4_optics
- foxsi4_position4_uniform_al
- foxsi4_position4_detector_response

- foxsi4_position5_thermal_blanket
- foxsi4_position5_optics
- foxsi4_position5_al_mylar
- foxsi4_position5_pixelated_attenuator
- foxsi4_position5_detector_response

attenuation.py **(level 1)**

- att_thermal_blanket
- att_uniform_al_cdte
- att_pixelated
- att_al_mylar
- att_sigmoid
- att_cmos_filter
- att_cmos_obfilter
- att_cmos_collimator_ratio
- att_foxsi4_atmosphere

detector_response.py **(level 1)**

- cdte_det_resp
- cmos_det_resp

effective_area.py **(level 1)**

- eff_area_msfc_10shell
- eff_area_msfc_hi_res
- eff_area_nagoya_hxt
- eff_area_nagoya_sxt
- eff_area_cmos
- eff_area_cmos_telescope

quantum_efficiency.py **(level 1)**

- qe_cmos

FOXSI-4 structure

When using the above code and combining responses, it might be useful to know the composition of each telescope (synonymous with position).

Telescope	Front of optic	Optic	Behind optic	Detector
0	OBF+Collimator	MSFC 2-shell X10	Pre-filter+QE	CMOS
1	OBF+Collimator	Nagoya 1-shell SXT	Pre-filter+QE	CMOS
2	Thermal blanket	MSFC 10-shell X-7	Al (0.015”)	CdTe
3	Thermal blanket	MSFC 2-shell X09	Al Mylar+Pix. Att.	CdTe
4	Thermal blanket	Nagoya 1-shell HXT	Al (0.005”)	CdTe
5	Thermal blanket	MSFC 10-shell X-8	Al Mylar+Pix. Att.	CdTe
6	Thermal blanket	MSFC 2-shell X11	Al Mylar	Timepix

Each component should be easily mappable onto the functions provided in the response-tools codebase with the exception of the Timepix response.