plots-xrfi

September 10, 2025

1 Examine and make plots of the xRFI algorithm.

Steven Murray

[35]:
```
import datetime
```

[41]:
```
print(f"Last updated: {datetime.datetime.now().strftime('%Y/%m/%d')}")
```

```
Last updated: 2025/09/10
```

This notebook/memo outlines the main source of differences between the xRFI procedure used in the legacy C-code versus the `edges-analysis` Python code, in gritty detail.

The broad arguments made here are outlined in the `edges-analysis` paper. As an example, we use day 2016-250, though the results should be similar for any particular day.

[1]:
```
from pygsdata import GSData
from pathlib import Path
import matplotlib.pyplot as plt
import numpy as np
import sys
import attrs
from edges import filters, modeling as mdl
from edges.averaging import NsamplesStrategy
from functools import cached_property
from edges.filters.xrfi import xrfi_iterative_sliding_window
from edges.modeling.fitting import _c_qrd, _get_a_and_b

sys.path.append("/data7/smurray/edges/projects_with_nive/
 ↪edges-bowman2018-pipeline/src/edges_pipelines/")
import utils
```

1.1 Setup and Data Loading

[2]:
```
alan_output_dir = "/home/smurray/data4/edges/alans-pipeline/scripts/
 ↪H2CaseFieldData/"
```

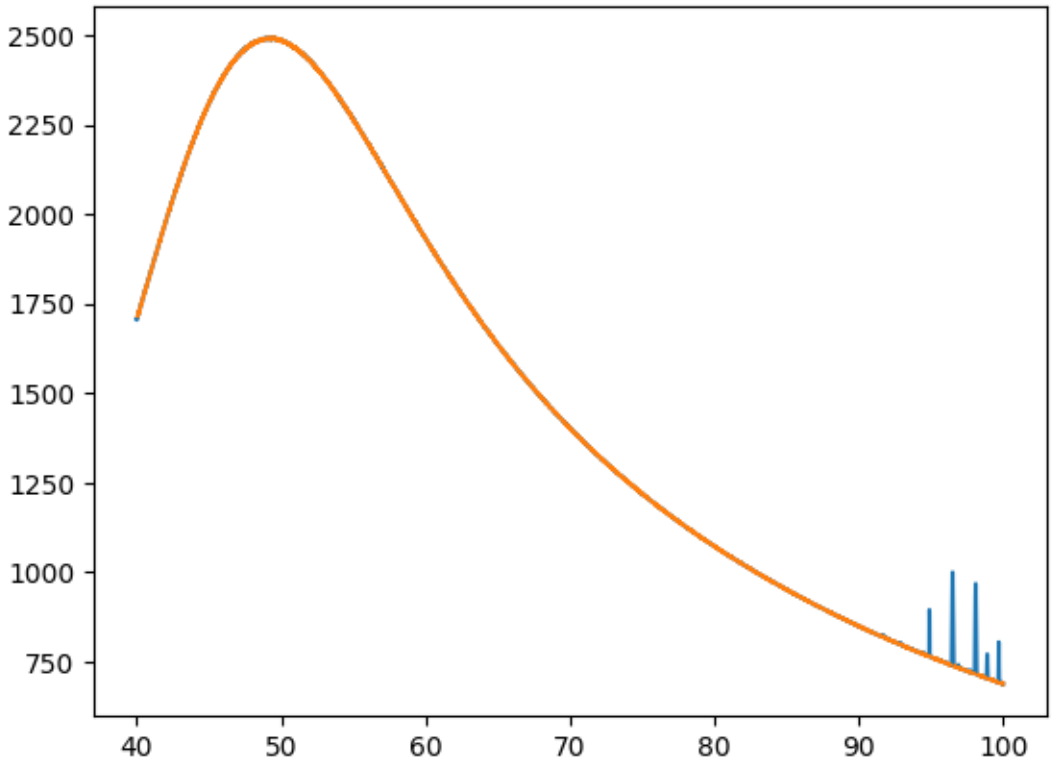
[3]:
```
data = GSData.from_file("/data7/smurray/edges/projects_with_nive/
  ↪edges-bowman2018-pipeline/results/bowman2018repro-injectbeam-doxrfi/
  ↪day-averaged-data/2016-250.unsmoothed.averaged.gsh5")
```

[4]:
```
year, day = map(int,data.name.split("-"))
```

[5]:
```
alan_output_dir = Path(alan_output_dir) / str(utils.yday_to_alanday(year, day))
```

[6]:
```
plt.plot(data.freqs, data.data.squeeze())
plt.plot(data.freqs, np.where(data.complete_flags.squeeze(), np.nan, data.data.
  ↪squeeze()))
```

[6]:
```
[<matplotlib.lines.Line2D at 0x7f681944d850>]
```



The following are high-precision outputs directly from the C-code:

[7]:
```
alan_rfi_flags = ~np.genfromtxt(alan_output_dir/"rfi_flags.txt").astype(bool)

rfi_models = np.loadtxt(alan_output_dir/"rfi_models.txt")
rfi_stds = np.loadtxt(alan_output_dir / "rfi_rmss.txt")
rfi_basis = np.loadtxt(alan_output_dir / "rfi_basisfuncs.txt")
```

1.2 Inspect Initial Models

The xRFI algorithm begins by fitting a linear Fourier model. The exact best-fit model has a large impact on the results, so we explore some different cases here.

First, let's define the actual model that would be used in the RFI algorithm. This model is setup to be as close as possible to the C-code. However, it can have a few differences, due to the fact that cos/sin are evaluated slightly differently and also that π has limited precision in the C-code.

[8]:
```
fmodel = mdl.Fourier(n_terms=37, period=1.5, transform=mdl.ZerotooneTransform(
    range=(
        data.freqs.min().to_value("MHz"),
        data.freqs.max().to_value("MHz"),
    )
)).at(x=data.freqs.to_value("MHz"))
```

Let's also define a model variant where we really try to match the C-code by changing the precision of π:

[9]:
```
alanpi = 3.1415926536

@attrs.define(frozen=True, kw_only=True, slots=False)
class FourierPi(mdl.Model):
    """A Fourier-basis model with limited-precision pi."""

    period: float = attrs.field(default=2 * alanpi, converter=float)

    @cached_property
    def _period_fac(self):
        return 2 * alanpi / self.period

    def get_basis_term(self, indx: int, x: np.ndarray) -> np.ndarray:
        """Define the basis functions of the model."""
        if indx == 0:
            return np.ones_like(x)
        if indx % 2:
            return np.cos(self._period_fac * ((indx + 1) // 2) * x)
        return np.sin(self._period_fac * ((indx + 1) // 2) * x)
```

[10]:
```
fmodel_pi = FourierPi(n_terms=37, period=1.5, transform=mdl.ZerotooneTransform(
    range=(
        data.freqs.min().to_value("MHz"),
        data.freqs.max().to_value("MHz"),
    )
)).at(x=data.freqs.to_value("MHz"))
```

What is the maximum difference compared to the basis functions computed in C?

[11]:
```
def rms(x):
    return np.sqrt(np.mean(np.square(x)))
```

[12]:
```
print(f"RMS difference between default Fourier basis and C code: {rms(fmodel.
  ↪basis - rfi_basis.T):.2e}")
print(f"RMS difference between low-pi-res Fourier basis and C code:␣
  ↪{rms(fmodel_pi.basis - rfi_basis.T):.2e}")
```

```
RMS difference between default Fourier basis and C code: 5.93e-11
RMS difference between low-pi-res Fourier basis and C code: 1.87e-15
```

Now, let's look at the impact on the residuals of the best-fit model to the initial unflagged data. Since this is the starting point of the RFI algorithm, if there are differences here, they will propagate. Here, we can simply fit each model, though we hackily construct one model that uses the output basis functions from the C-code directly. In all the cases here, we use the fitting method closest to the C-code (in fact, it's literally the C-code copied over and compiled):

[13]:
```
fmodel_hacked_basis = attrs.evolve(fmodel_pi, init_basis=rfi_basis.T)
```

[14]:
```
default_fourier_modelfit = fmodel.fit(ydata = data.data[0,0,0], weights=np.
  ↪ones(data.nfreqs), method='qrd-c').evaluate()
lowpi_fourier_modelfit = fmodel_pi.fit(ydata = data.data[0,0,0], weights=np.
  ↪ones(data.nfreqs), method='qrd-c').evaluate()
hacked_fourier_modelfit = fmodel_hacked_basis.fit(ydata = data.data[0,0,0],␣
  ↪weights=np.ones(data.nfreqs), method='qrd-c').evaluate()
```

[15]:
```
print(f"RMS difference between default Fourier model and C code:␣
  ↪{rms(default_fourier_modelfit - rfi_models[0])*1e3:.2e} mK")
print(f"RMS difference between low-pi-res Fourier model and C code:␣
  ↪{rms(lowpi_fourier_modelfit - rfi_models[0])*1e3:.2e} mK")
print(f"RMS difference between hacked Fourier model and C code:␣
  ↪{rms(hacked_fourier_modelfit - rfi_models[0])*1e3:.2e} mK")
```

```
RMS difference between default Fourier model and C code: 2.83e+01 mK
RMS difference between low-pi-res Fourier model and C code: 1.34e+01 mK
RMS difference between hacked Fourier model and C code: 1.90e-03 mK
```

What about the differences due to the fitting method? Here we use the hacked basis in each case, so we know the differences are just from the method itself.

[16]:
```
hacked_fourier_modelfit_qrdpy = fmodel_hacked_basis.fit(ydata = data.
  ↪data[0,0,0], weights=np.ones(data.nfreqs), method='alan-qrd').evaluate()
hacked_fourier_modelfit_qr = fmodel_hacked_basis.fit(ydata = data.data[0,0,0],␣
  ↪weights=np.ones(data.nfreqs), method='qr').evaluate()
hacked_fourier_modelfit_lstsq = fmodel_hacked_basis.fit(ydata = data.
  ↪data[0,0,0], weights=np.ones(data.nfreqs), method='lstsq').evaluate()
```

[17]:
```
print(f"RMS difference between alan-qrd and C code:␣
  ↪{rms(hacked_fourier_modelfit_qrdpy - rfi_models[0])*1e3:.2e} mK")
print(f"RMS difference between qr and C code: {rms(hacked_fourier_modelfit_qr -␣
  ↪rfi_models[0])*1e3:.2e} mK")
print(f"RMS difference between lstsq and C code:␣
  ↪{rms(hacked_fourier_modelfit_lstsq - rfi_models[0])*1e3:.2e} mK")
```

```
RMS difference between alan-qrd and C code: 4.34e+02 mK
RMS difference between qr and C code: 5.11e+02 mK
RMS difference between lstsq and C code: 5.11e+02 mK
```

But how good were the actual fits themselves? They're different, but is the C-code better or worse?

[18]:
```
print(f"RMS residual of default   : {rms(default_fourier_modelfit - data.
  ↪data[0,0,0])*1e3:.10e} mK")
print(f"RMS residual of low-res-pi: {rms(lowpi_fourier_modelfit - data.
  ↪data[0,0,0])*1e3:.10e} mK")
print(f"RMS residual of hacked    : {rms(hacked_fourier_modelfit - data.
  ↪data[0,0,0])*1e3:.10e} mK")

print(f"RMS residual of alan-qrd  : {rms(hacked_fourier_modelfit_qrdpy - data.
  ↪data[0,0,0])*1e3:.10e} mK")
print(f"RMS residual of qr        : {rms(hacked_fourier_modelfit_qr - data.
  ↪data[0,0,0])*1e3:.10e} mK")
print(f"RMS residual of lstsq     : {rms(hacked_fourier_modelfit_lstsq - data.
  ↪data[0,0,0])*1e3:.10e} mK")
```

```
RMS residual of default   : 1.0072786384e+04 mK
RMS residual of low-res-pi: 1.0073802559e+04 mK
RMS residual of hacked    : 1.0073208766e+04 mK
RMS residual of alan-qrd  : 1.0060669585e+04 mK
RMS residual of qr        : 1.0060262886e+04 mK
RMS residual of lstsq     : 1.0060262886e+04 mK
```

The differences between the methods are very small, because they're *dominated* by the residuals in the RFI channels (and by noise etc.). However, at the level o fhte third decimal place, all methods do better than the cumulative-sum C-code QR-decomposition.

1.3 Impact on the RMS model

The differences in the best-fit models amongst the cases investigated above is non-negligible (on the order of ~100's of mK), however with just that we'd expect a very small impact on the final flags, because the RFI that we're trying to detect is hundreds/thousands of K.

However, the metric that we flag on is not directly the residual of the data to the model, but rather the Z-score, which is scaled by the estimated RMS. The RMS is roughly the (smoothed) amplitude of the residuals. Although the fractional error on the residuals is small (where the fraction is against the spectrum amplitude), the residual differences as a ratio to the *residual* amplitude can be much larger, resulting in tens of percent differences in Z.

Let's check that out.

To do this, we run the actual xRFI algorithm, because it tracks the RMS model.

[19]:
```
kw = dict(
    spectrum = data.data.squeeze(),
    freqs=data.freqs.to_value("MHz"),
    weights=np.ones_like(data.data.squeeze()),
    max_iter=100,
    reflag_thresh=0,
    window_frac = 16,
    min_window_size = 10,
    threshold = 2.5,
    #reflag_thresh = 1.0,
    watershed = {
        1.0: 4,
        10.0: 8,
        100.0: 16
    },
)
```

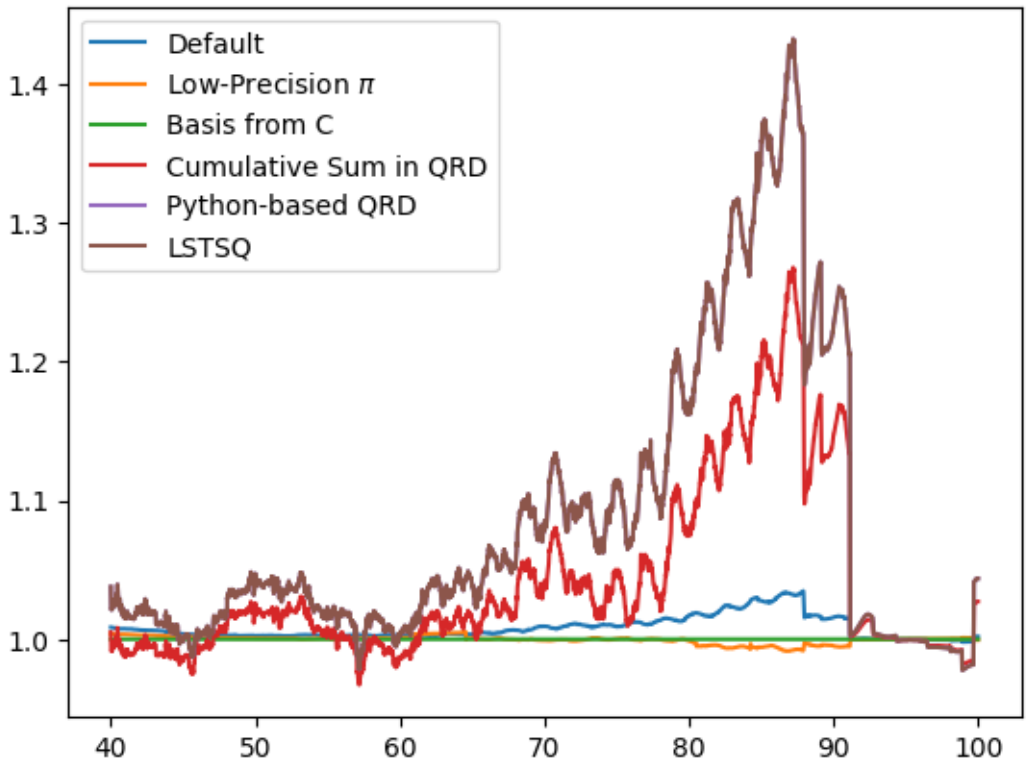
[20]:
```
default_flags, default_info = xrfi_iterative_sliding_window(model=fmodel,␣
 ↪fit_kwargs={"method": "qrd-c"}, **kw)
lowpi_flags, lowpi_info = xrfi_iterative_sliding_window(model=fmodel_pi,␣
 ↪fit_kwargs={"method": "qrd-c"}, **kw)
hacked_flags, hacked_info =␣
 ↪xrfi_iterative_sliding_window(model=fmodel_hacked_basis,␣
 ↪fit_kwargs={"method": "qrd-c"}, **kw)

qrdpy_flags, qrdpy_info =␣
 ↪xrfi_iterative_sliding_window(model=fmodel_hacked_basis,␣
 ↪fit_kwargs={"method": "alan-qrd"}, **kw)
qr_flags, qr_info = xrfi_iterative_sliding_window(model=fmodel_hacked_basis,␣
 ↪fit_kwargs={"method": "qr"}, **kw)
lstsq_flags, lstsq_info =␣
 ↪xrfi_iterative_sliding_window(model=fmodel_hacked_basis,␣
 ↪fit_kwargs={"method": "lstsq"}, **kw)
```

[23]:
```
r = default_info.stds[0]/rfi_stds[0]
print((r.max() - 1)*100)
plt.plot(data.freqs, r, label='Default')

r = lowpi_info.stds[0]/rfi_stds[0]
print((r.max() - 1)*100)
plt.plot(data.freqs, r, label='Low-Precision $\pi$')

r = hacked_info.stds[0]/rfi_stds[0]
print((r.max() - 1)*100)
```

```
plt.plot(data.freqs, r, label='Basis from C')

r = qrdpy_info.stds[0]/rfi_stds[0]
print((r.max() - 1)*100)
plt.plot(data.freqs, r, label='Cumulative Sum in QRD')

r = qr_info.stds[0]/rfi_stds[0]
print((r.max() - 1)*100)
plt.plot(data.freqs, r, label='Python-based QRD')

r = lstsq_info.stds[0]/rfi_stds[0]
print((r.max() - 1)*100)
plt.plot(data.freqs, r, label='LSTSQ')
plt.legend()
```

```
3.4717599629166873
0.5243567286187467
0.00018000511465832147
26.807687603879526
43.29493746821677
43.2949376597761
```

[23]: <matplotlib.legend.Legend at 0x7f6816aad1d0>



1.4 Impact on Flags with xRFI

[28]:
```
def print_flag_diffs(label, info):
    print(f"Flagged in {label}, not in C: {np.sum(info.flags[-1] &␣
 ↪~alan_rfi_flags[-1])}")
    print(f"Flagged in C, not in {label}: {np.sum(~info.flags[-1] &␣
 ↪alan_rfi_flags[-1])}")
```

[30]:
```
print_flag_diffs("Default", default_info)
print_flag_diffs("Low-PI", lowpi_info)
print_flag_diffs("C-Basis", hacked_info)

print_flag_diffs("QRDpy", qrdpy_info)
print_flag_diffs("QR", qr_info)
print_flag_diffs("LSTSq", lstsq_info)
```

```
Flagged in Default, not in C: 3
Flagged in C, not in Default: 92
Flagged in Low-PI, not in C: 24
Flagged in C, not in Low-PI: 66
Flagged in C-Basis, not in C: 0
Flagged in C, not in C-Basis: 0
Flagged in QRDpy, not in C: 64
Flagged in C, not in QRDpy: 46
Flagged in QR, not in C: 47
Flagged in C, not in QR: 183
Flagged in LSTSq, not in C: 47
Flagged in C, not in LSTSq: 183
```

Using the hacked basis directly from C means that we can *exactly* match the C-pipeline in terms of the final flags. That is, using an initial set of basis functions that exactly match, with the exact same C-code used for fitting, produces a best-fit model (initial model) that matches to within 10^{-3} mK, and this does not affect the flags. This means the algorithm itself is working well!

One last question we can ask is whether the small shifts in the basis functions would have affected the other fitting methods as much as they did the C-based QR decomposition.

[31]:
```
lstsq_lowpi_flags, lstsq_lowpi_info =␣
 ↪xrfi_iterative_sliding_window(model=fmodel_pi, fit_kwargs={"method":␣
 ↪"lstsq"}, **kw)
lstsq_default_flags, lstsq_default_info =␣
 ↪xrfi_iterative_sliding_window(model=fmodel, fit_kwargs={"method": "lstsq"},␣
 ↪**kw)
```

[33]:
```
def print_flag_diffs(label, info):
    print(f"Flagged in {label}, not in hacked: {np.sum(info.flags[-1] &␣
 ↪~lstsq_info.flags[-1])}")
    print(f"Flagged in hacked, not in {label}: {np.sum(~info.flags[-1] &␣
 ↪lstsq_info.flags[-1])}")
```

[34]:
```
print_flag_diffs("default", lstsq_default_info)
print_flag_diffs("lowpi", lstsq_lowpi_info)
```

```
Flagged in default, not in hacked: 0
Flagged in hacked, not in default: 0
Flagged in lowpi, not in hacked: 0
Flagged in hacked, not in lowpi: 0
```

The answer is no: using the `lstsq` method is more robust to small changes in the basis set, and is therefore a better method.

1.5 Summary

We find that the `polyfit` and associated `qrd` functions in `acqplot7amoon.c` behave somewhat poorly for badly conditioned linear systems. In particular, for the averaged spectrum on day 2016-250 (the first day in the B18 dataset), using a 37-term Fourier model (the model used in B18 processing to find RFI), we find that very small changes in the basis functions occasion quite significant changes in the best-fit model, and consequently the flags that are determined.

Even small changes like increasing the precision of π from 10 decimal places to full double-precision result in 96 different channels being flagged compared to the base C-code. Furthermore, even using the same algorithm, and *exactly the same basis function*, but using numpy's pairwise-summation method (which is more accurate than simple accumulation) when computing the dot products in preparation for `qrd` results in final differences of more than 100 flagged channels.

This sensitivity to the precise data is not present in numpy's `np.linalg.lstsq` method, which normalizes the basis functions in order to improve the matrix conditioning.