TorchSig

TorchSig 2.0: New Transforms, Custom Datasets and Future Plans

Erebus Oh, LTS

Justin Mullins, LTS

Matt Carrick, Peraton Labs

Matt Vondal, Peraton Labs

Jared Hoffman, Applied Insight

Frank Leonardo, Peraton Labs

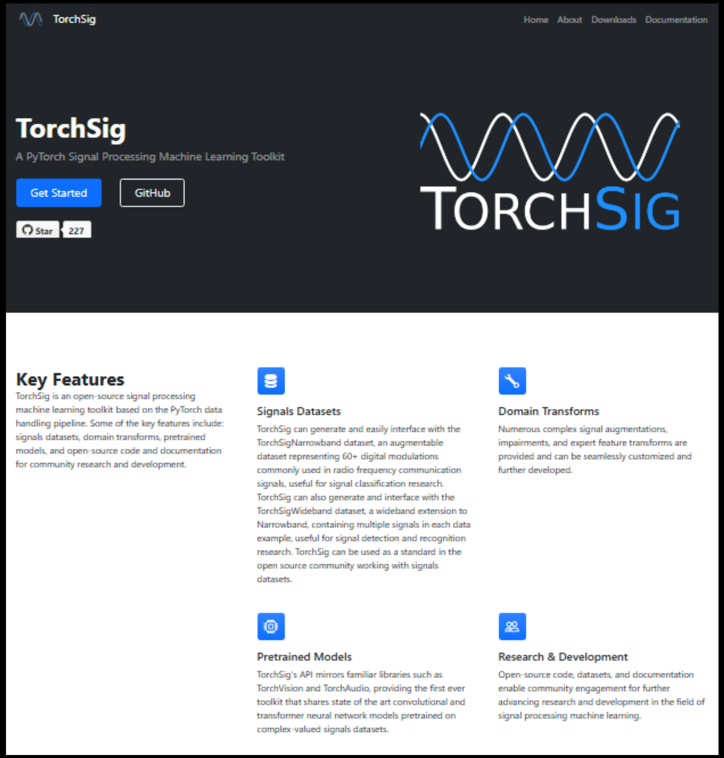
Paul Toliver, Peraton Labs

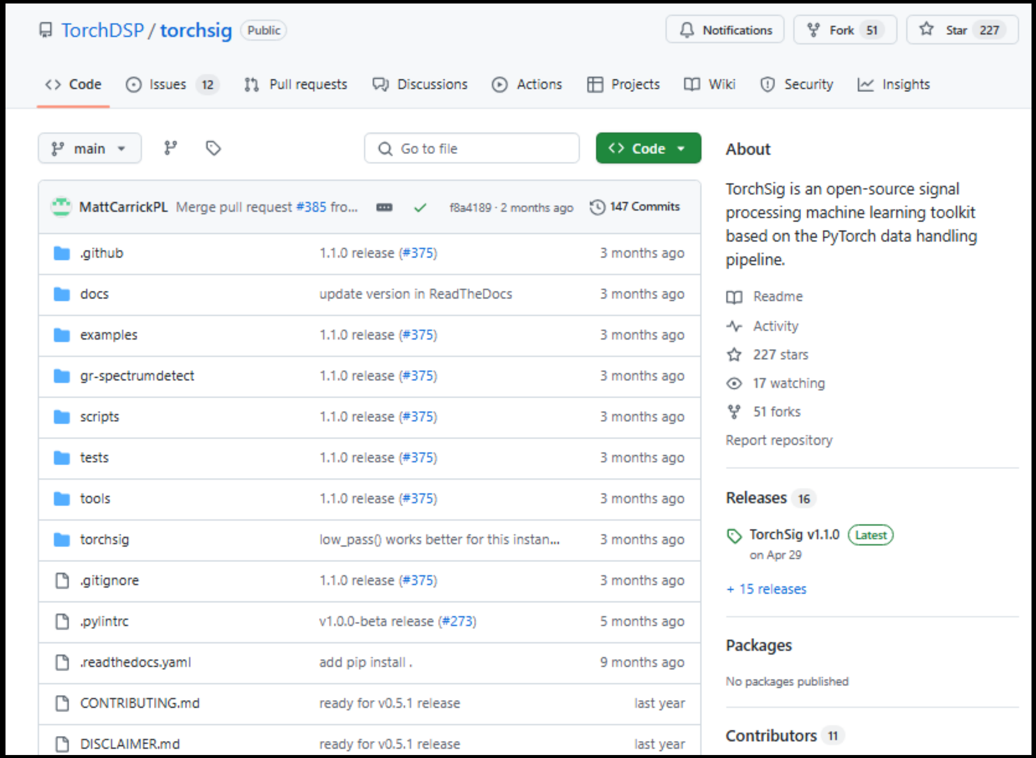
Rob Miller, Peraton Labs

What is TorchSig?

- Dataset Generation
  - Signal modulators
  - Analog RF impairments and ML transforms
- ML Models
  - Pre-trained
  - Make your own
- Tutorial notebooks
- gr-spectrumdetect: GR block for spectrogram energy detection

What is TorchSig?



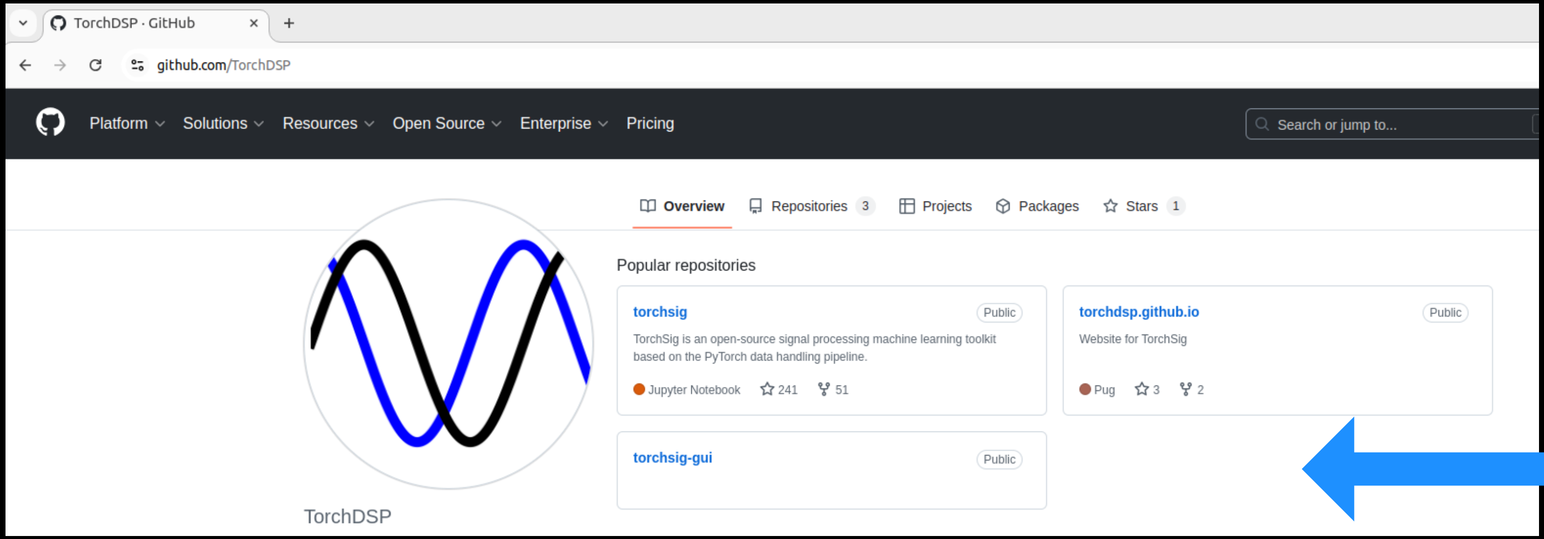


torchsig.com

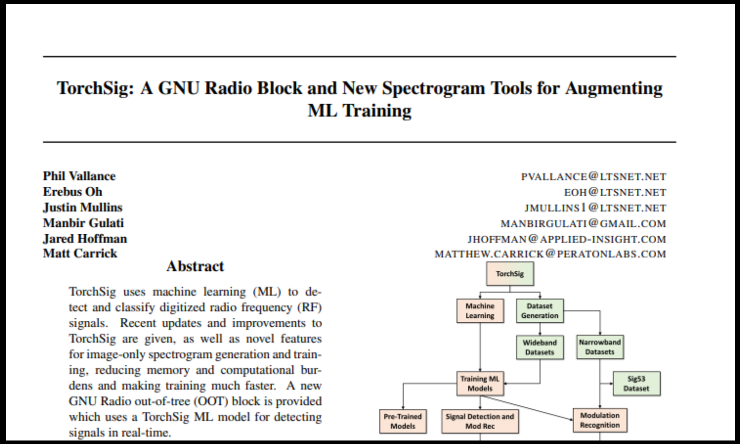
github.com/torchdsp/torchsig

Repository Breakouts

- torchsig
- gr-spectrumdetect
- Others?



Prior Work

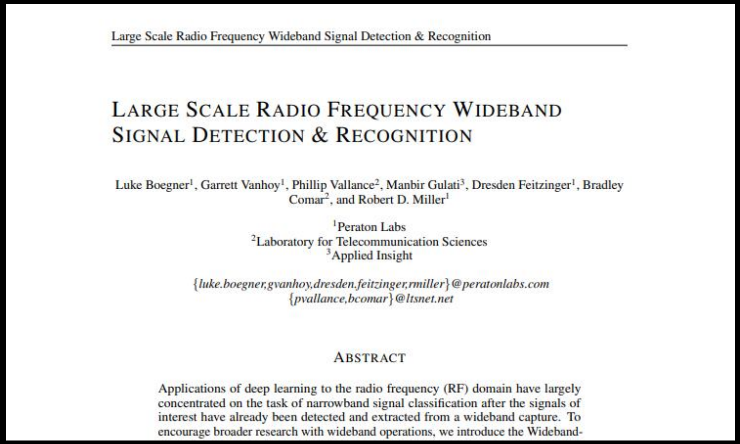


TorchSig: A GNU Radio Block and New Spectrogram Tools for Augmenting ML Training

Phil Vallance PVALLANCE@LTSNET.NET
Erebus Oh EOH@LTSNET.NET
Justin Mullins JMULLINS1@LTSNET.NET
Manbir Gulati MANBIRGULATI@GMAIL.COM
Jared Hoffman JHOFFMAN@APPLIED-INSIGHT.COM
Matt Carrick MATTHEW.CARRICK@PERATONLABS.COM

Abstract

TorchSig uses machine learning (ML) to detect and classify digitized radio frequency (RF) signals. Recent updates and improvements to TorchSig are given, as well as novel features for image-only spectrogram generation and training, reducing memory and computational burdens and making training much faster. A new GNU Radio out-of-tree (OOT) block is provided which uses a TorchSig ML model for detecting signals in real-time.



LARGE SCALE RADIO FREQUENCY WIDEBAND SIGNAL DETECTION & RECOGNITION

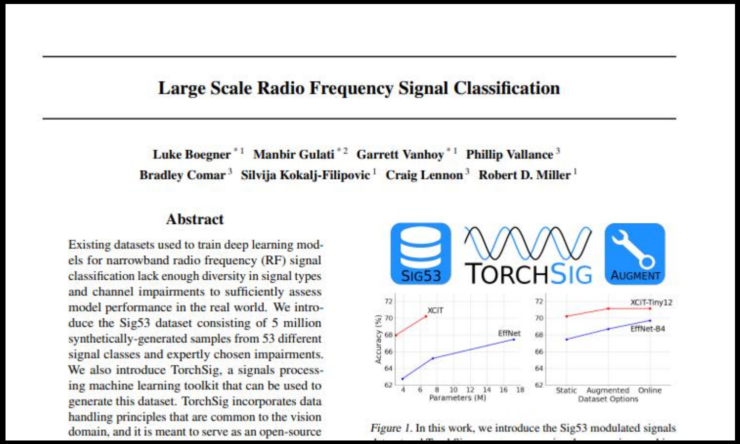
Luke Boegner[1], Garrett Vanhoy[1], Phillip Vallance[2], Manbir Gulati[3], Dresden Feitzinger[1], Bradley Comar[2], and Robert D. Miller[1]

[1]Peraton Labs
[2]Laboratory for Telecommunication Sciences
[3]Applied Insight

{luke.boegner,gvanhoy,dresden.feitzinger,rmiller}@peratonlabs.com
{pvallance,bcomar}@ltsnet.net

ABSTRACT

Applications of deep learning to the radio frequency (RF) domain have largely concentrated on the task of narrowband signal classification after the signals of interest have already been detected and extracted from a wideband capture. To encourage broader research with wideband operations, we introduce the Wideband-



Large Scale Radio Frequency Signal Classification

Luke Boegner [*1] Manbir Gulati [*2] Garrett Vanhoy [*1] Phillip Vallance [3]
Bradley Comar [3] Silvija Kokalj-Filipovic [1] Craig Lennon [3] Robert D. Miller [1]

Abstract

Existing datasets used to train deep learning models for narrowband radio frequency (RF) signal classification lack enough diversity in signal types and channel impairments to sufficiently assess model performance in the real world. We introduce the Sig53 dataset consisting of 5 million synthetically-generated samples from 53 different signal classes and expertly chosen impairments. We also introduce TorchSig, a signals processing machine learning toolkit that can be used to generate this dataset. TorchSig incorporates data handling principles that are common to the vision domain, and it is meant to serve as an open-source

Figure 1. In this work, we introduce the Sig53 modulated signals

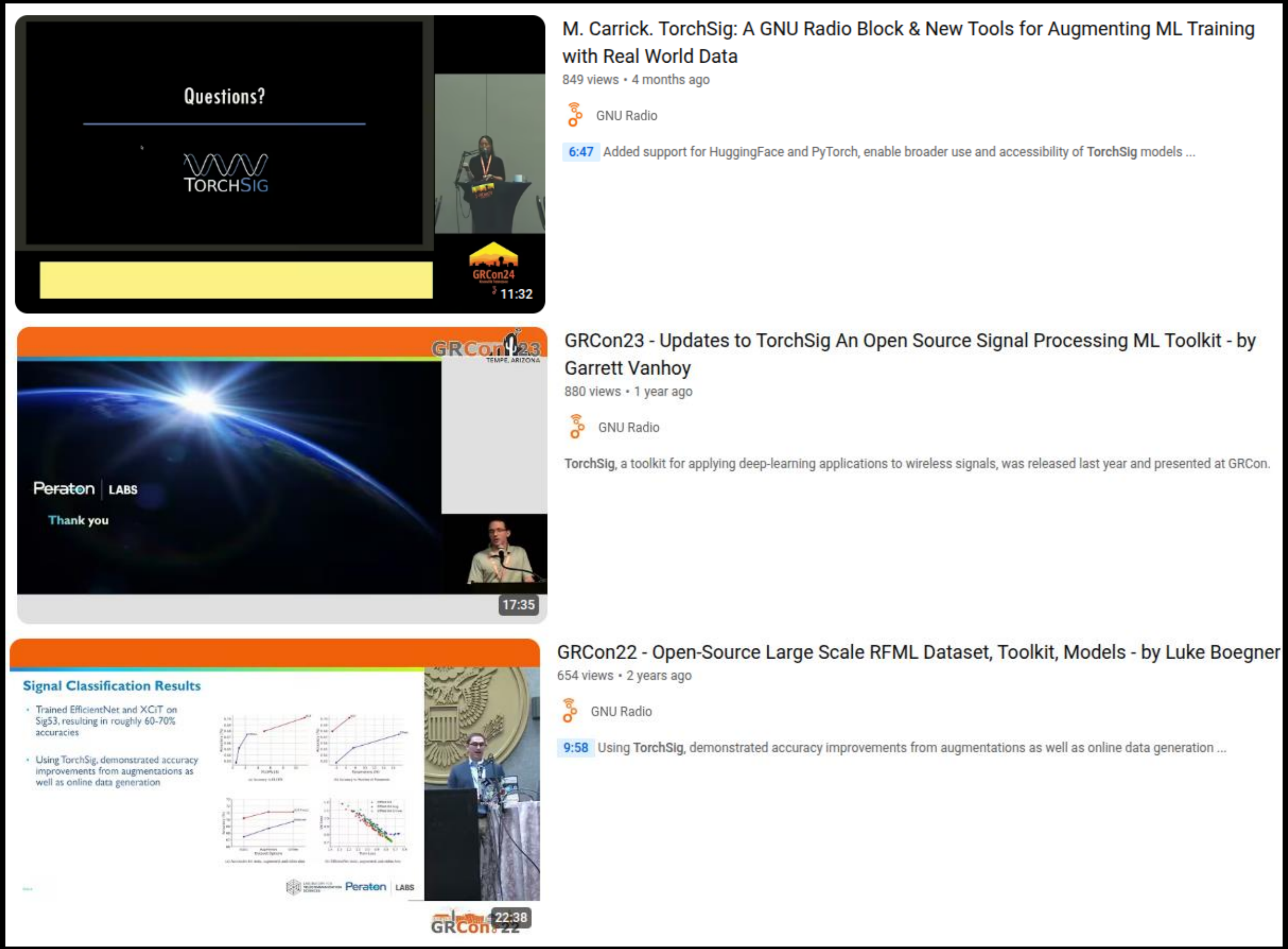
GRCon 2024: https://events.gnuradio.org/event/24/contributions/628/

Arxiv 2022: https://arxiv.org/abs/2211.10335

Arxiv 2022: https://arxiv.org/abs/2207.09918

Original papers, many changes since publication

Prior Work



GRCon 2024: https://www.youtube.com/watch?v=pPrHAWjqk6M

GRCon 2023: https://www.youtube.com/watch?v=2OBGBa6Oq2c

GRCon 2022: https://www.youtube.com/watch?v=uDLjl7QuWio

Recent Releases

v0.6 (Oct 24): AM, FM, Chirp SS, LFM signals

v0.6.1 (Jan 24): Improved notebook tutorials, bug fixes

v1.0 (Mar 25): A near "from scratch" rewrite

v1.1 (Apr 25): Speed improvements, bug fixes

v2.0 (Sep 25): Analog RF impairments, Custom Datasets, NB/WB dataset unification

Signal Modulators

Constellations:
- OOK, BPSK, QPSK
- 4-ASK, 8-ASK, 16-ASK, 32-ASK, 64-ASK
- 4-PSK, 8-PSK, 16-PSK, 32-PSK, 64-PSK
- 16-QAM, 32-QAM, 32-QAM Cross, 64-QAM
- 128-QAM, 256-QAM, 512-QAM, 1024-QAM

FSK:
- 2-FSK, 4-FSK, 8-FSK, 16-FSK
- 2-GFSK, 4-GFSK, 8-GFSK, 16-GFSK
- 2-MSK, 4-MSK, 8-MSK, 16-MSK
- 2-GMSK, 4-GMSK, 8-GMSK, 16-GMSK

OFDM:
- 64-OFDM, 72-OFDM, 128-OFDM, 180-OFDM, 256-OFDM 300-OFDM, 512-OFDM, 600-OFDM, 900-OFDM, 1024-OFDM, 1200-OFDM, 2048-OFDM

Analog AM/FM:
- AM-DSB-SC, AM-DSB, AM-LSB, AM-USB
- FM

Chirp-based:
- Linear Frequency Modulated (LFM) Radar, LFM Data
- Chirp Spread Spectrum

Unmodulated Carrier/Tone:
- Tone

Dataset Generation

- Datasets are collections of IQ and their associated metadata
- IQ is stored in a NumPy array
- Metadata includes sample rate, signal bandwidths, signal center frequencies and others
- Datasets can be generated synthetically or with externally generated data (v2.0 only)

Transforms

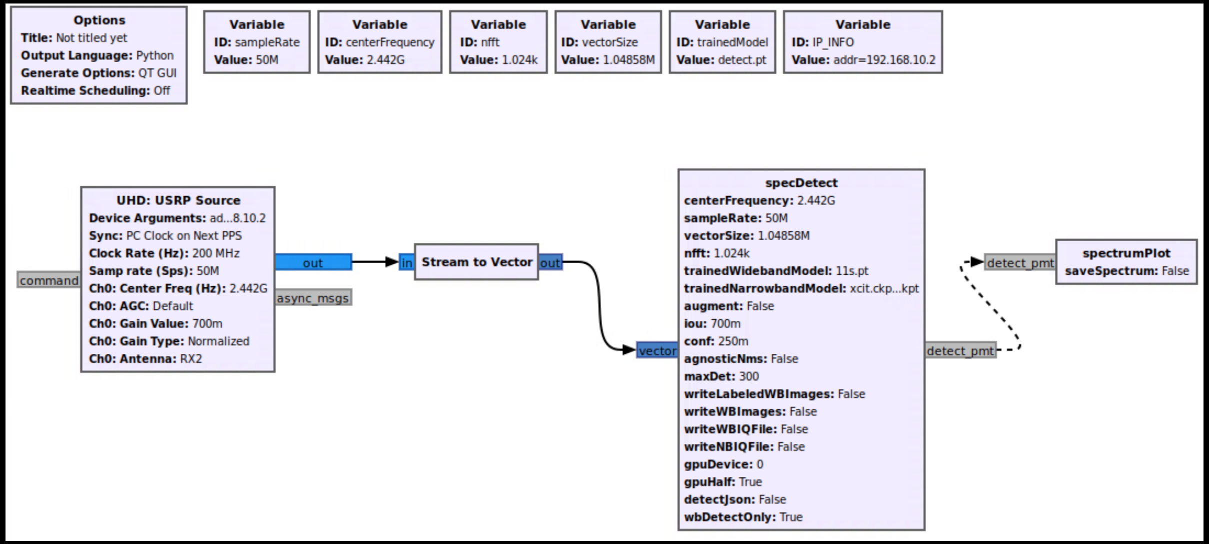
- Two categories of transforms:
  - Analog RF impairments
  - ML Transforms
- Analog RF Impairments:
  - Applies simulated real-world RF hardware effects
  - Examples: nonlinear amplifier compression, phase noise, IQ imbalance
- ML transforms:
  - Prepare and condition data for improved ML performance
  - Examples: I &Q swap, time reversal, drop samples

ML Models

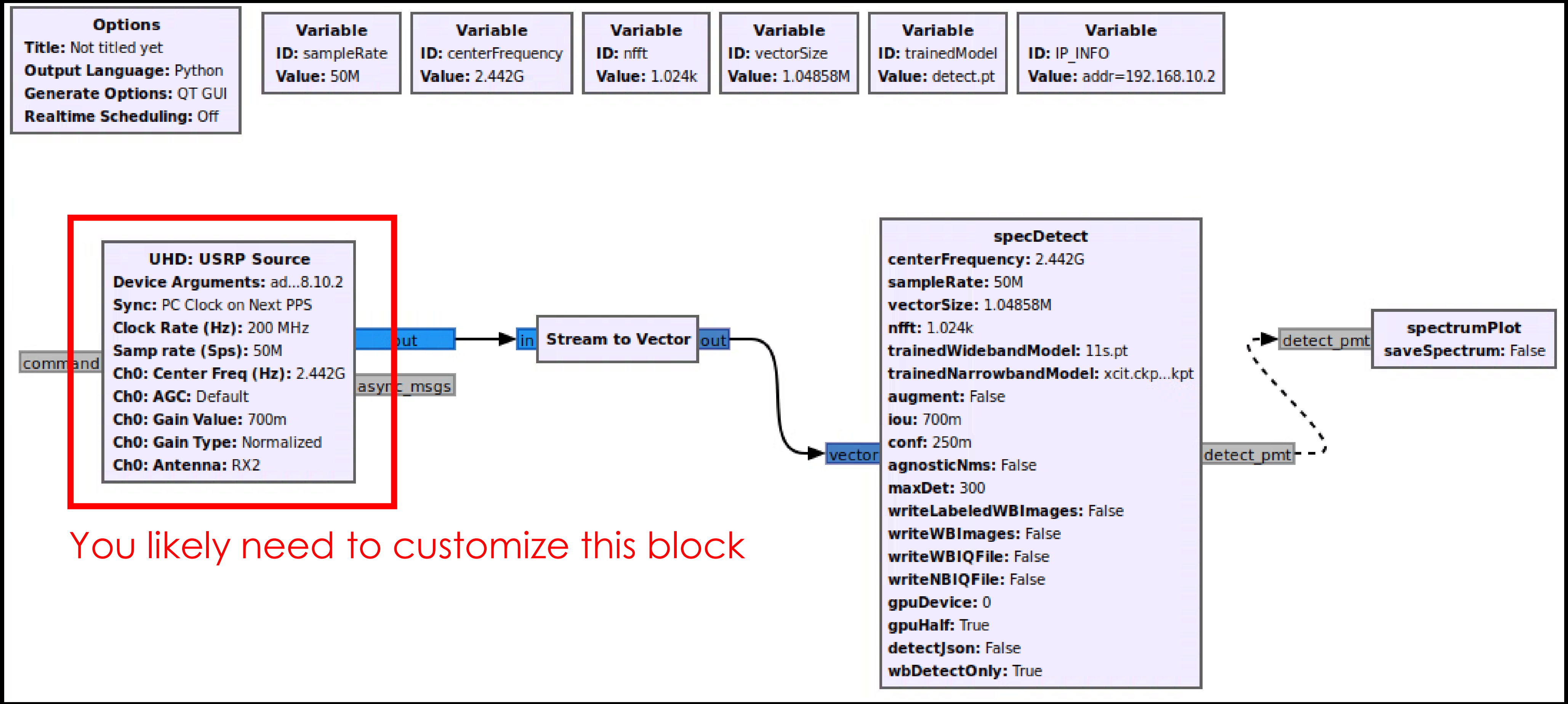
- Pre-trained Models:
  - `gr-spectrumdetect/examples/trained_model_download.sh`
  - `11s.pt`: YOLOv11 for energy detection on wideband spectrograms
  - `detect.pt`: YOLOv8 for energy detection on wideband spectrograms
  - `xcit.ckpt`: XCiT model trained for narrowband mod-rec
- Ability to train custom models on synthetic or custom datasets (2.0 only)

gr-spectrumdetect

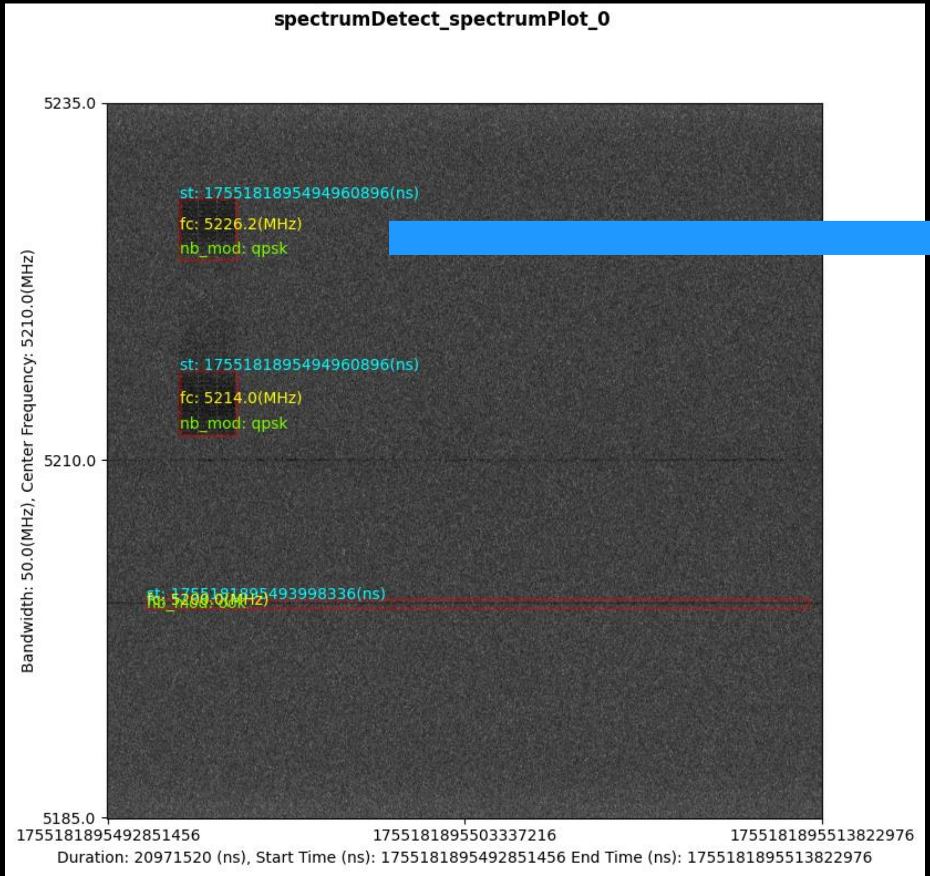
- GNU Radio Flowgraph and block
- Energy detection in spectrogram
- Modulation recognition (improvements under development)

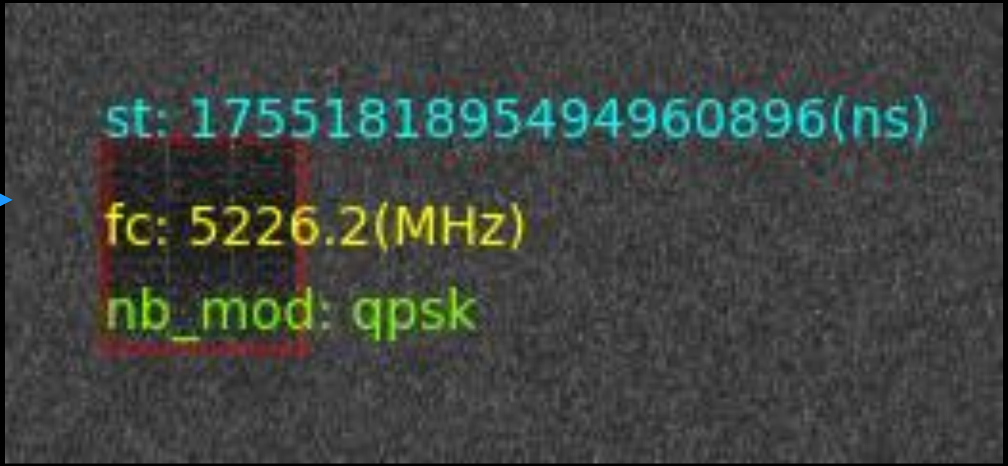


gr-spectrumdetect



gr-spectrumdetect





- Timestamp
- Center Frequency
- Modulation
- More fields available in JSON (bandwidth, start/stop time, etc.)

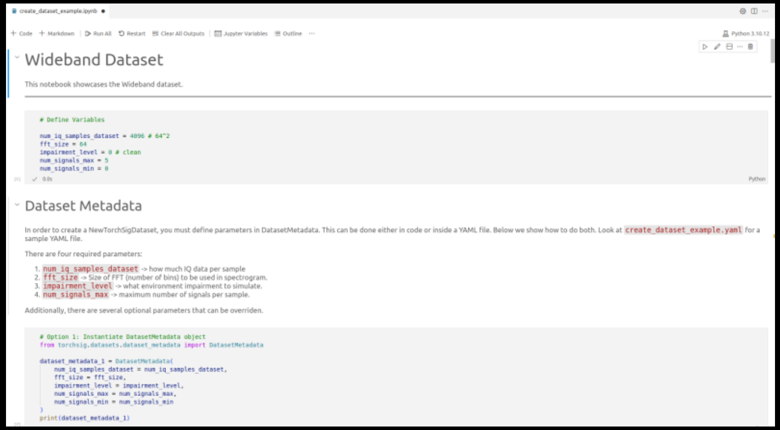
Tutorial Notebooks

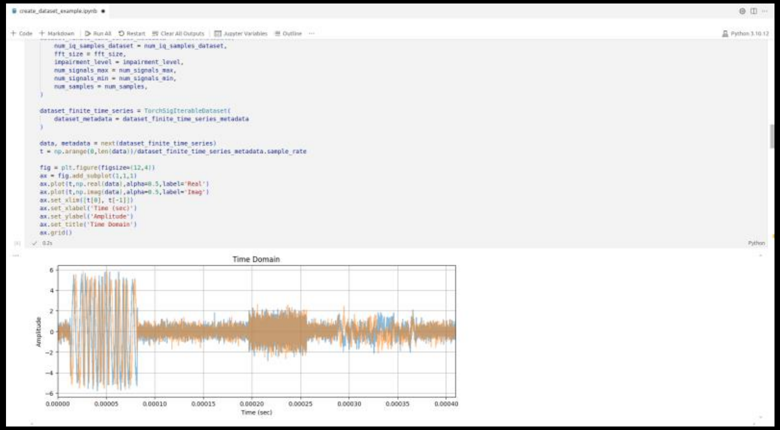
Notebooks serve as tutorials:

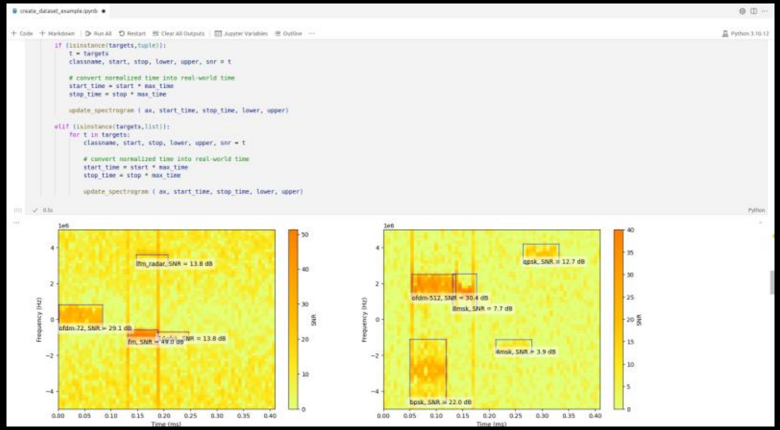
- Dataset creation & customization
- Training ML model for mod-rec on IQ samples
- Training ML model for detection in spectrogram
- RNG Seeding & Dataset repeatability
- Saving and loading datasets with YAML config files
- Link to Google Colab notebooks at the end of talk

Tutorial Notebooks

- Create and customize datasets
- `examples/create_dataset_example.ipynb`

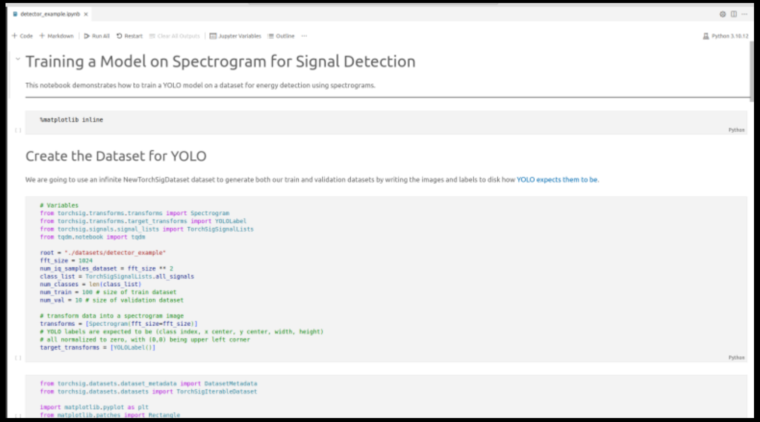


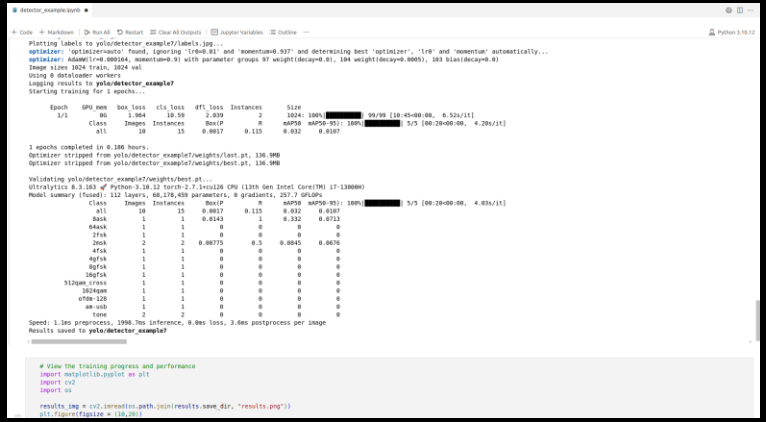


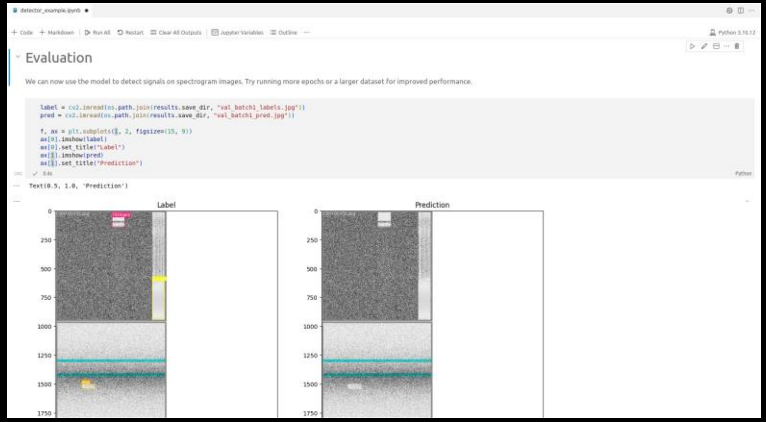


Tutorial Notebooks

- Train a spectrogram energy detector model
- `examples/detector_example.ipynb`

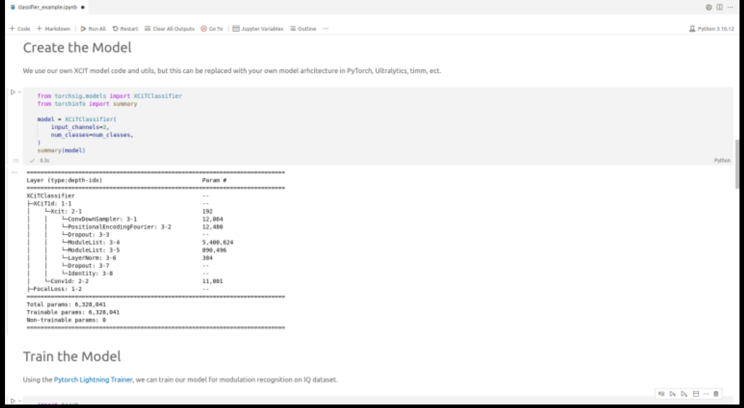


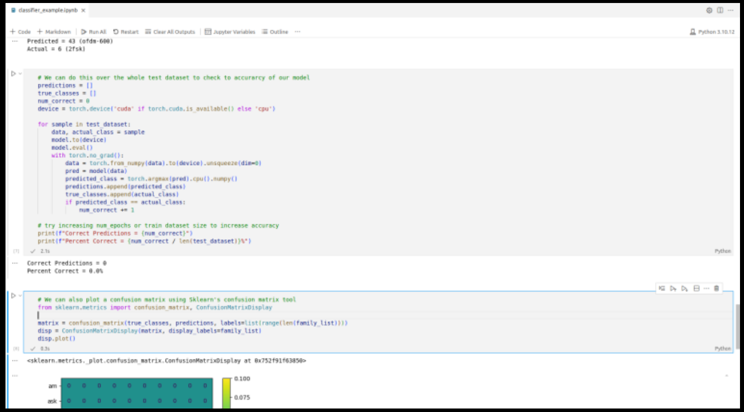




Tutorial Notebooks

- Train an IQ-based modulation recognition model
- `examples/classifier_example.ipynb`





TorchSig 2.0

New Features to v2.0:

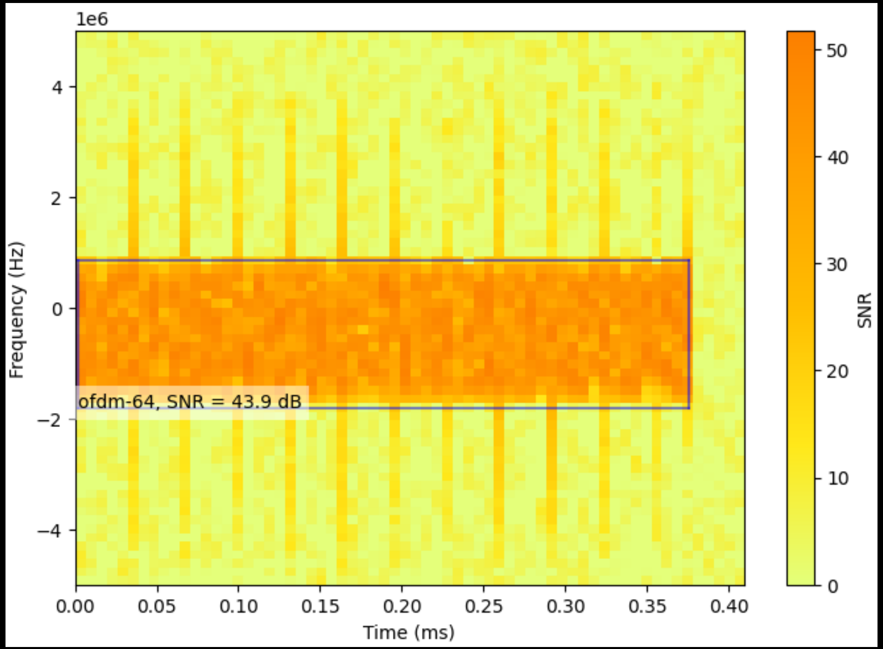
- Narrowband and Wideband “unification”
- More Dataset Customization
- Controlling co-channel interference in dataset generation
- RF Analog Impairments
- Custom & External Datasets

WB/NB Unification

Previously:

```
narrowband_metadata = NarrowbandMetadata(…)
narrowband_dataset = NewNarrowband(narrowband_metadata)
```

- 1 Signal
- Large relative burst duration
- Large relative bandwidth

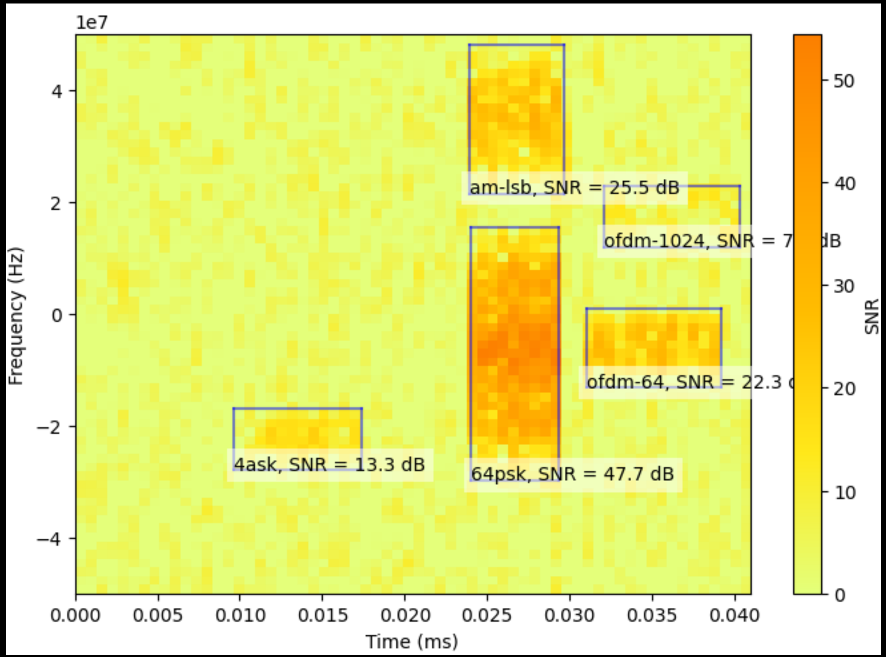


WB/NB Unification

Previously:

```
wideband_metadata = WidebandMetadata(…)
wideband_dataset = NewWideband(wideband_metadata)
```

- Many signals
- Small relative burst duration
- Small relative bandwidth



WB/NB Unification

Now in 2.0:

```
dataset_metadata = DatasetMetadata(...)
iterable_dataset = TorchSigIterableDataset(dataset_metadata)
```

- Customizable number of signals, burst duration, bandwidth, etc.
- Narrowband and Wideband-like datasets can be reproduced with YAML configs:
  - `torchsig/datasets/default_configs/narrowband_defaults.yaml`
  - `torchsig/datasets/default_configs/wideband_defaults.yaml`

Dataset Customization

Range for number of signals in spectrogram:

- num_signals_min
- num_signals_max

Range for SNR:

- snr_db_min
- snr_db_max

Range for the burst length:

- signal_duration_min
- signal_duration_max

Range for the bandwidth:

- signal_bandwidth_min
- signal_bandwidth_max

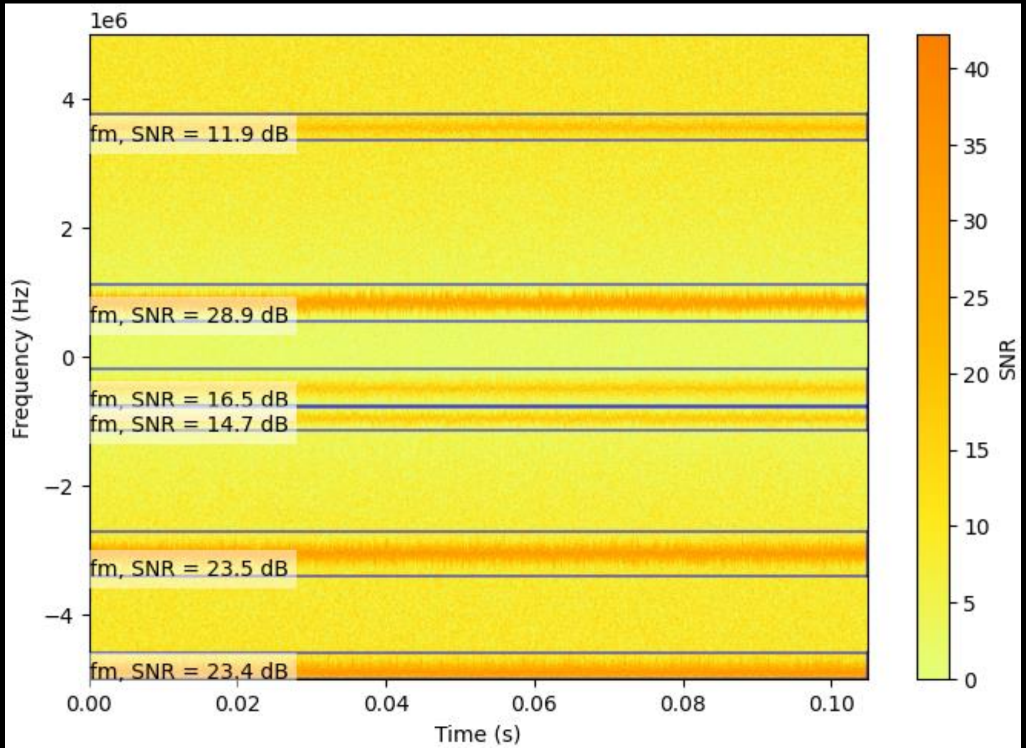
Range for the center frequency:

- signal_center_freq_min
- signal_center_freq_max

Dataset Customization

```
from torchsig.signals.signal_lists import TorchSigSignalLists
sample_rate = 10e6
dataset_metadata = DatasetMetadata(
    sample_rate = sample_rate,
    num_iq_samples_dataset = 1024**2,
    fft_size = 1024,
    num_signals_min = 6,
    num_signals_max = 6,
    snr_db_min=10,
    snr_db_max=30,
    signal_duration_min=1.0*num_iq_samples_dataset/sample_rate,
    signal_duration_max=1.0*num_iq_samples_dataset/sample_rate,
    signal_bandwidth_min=sample_rate/10,
    signal_bandwidth_max=sample_rate/10,
    cochannel_overlap_probability=0,
    class_list=TorchSigSignalLists.fm_signals
)
```

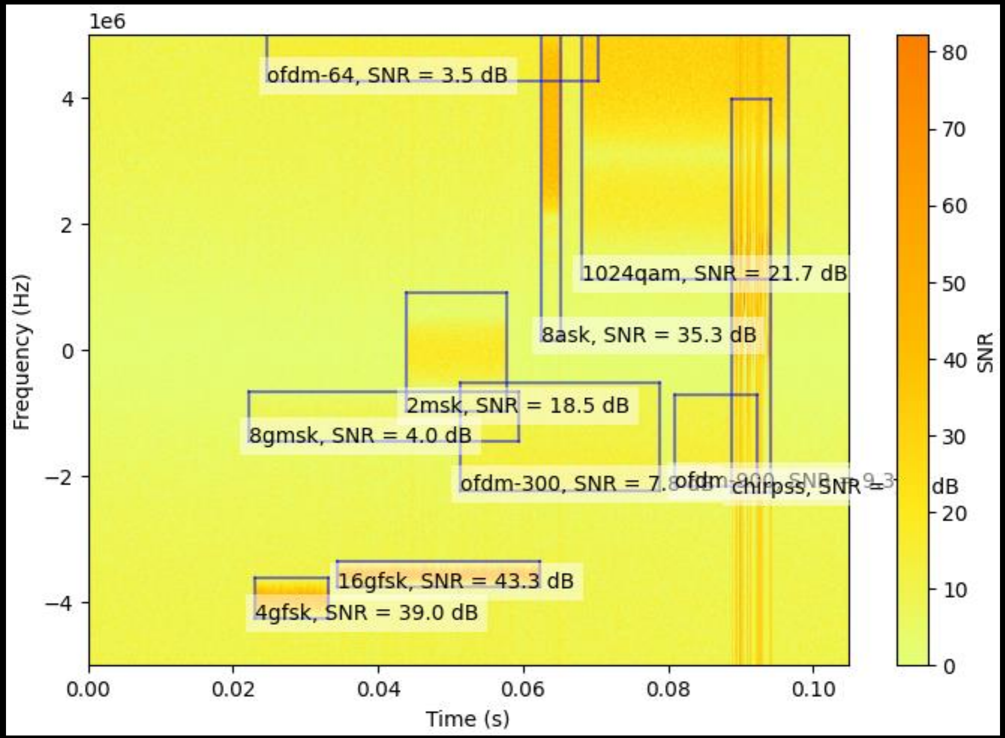
Simulated Broadcast FM Channels



Dataset Customization

```
sample_rate = 10e6
dataset_metadata = DatasetMetadata(
    sample_rate = sample_rate,
    num_iq_samples_dataset = 1024**2,
    fft_size = 1024,
    num_signals_min = 10,
    num_signals_max = 10,
    snr_db_min=0,
    snr_db_max=50,
    signal_duration_min=0.01*num_iq_samples_dataset/sample_rate,
    signal_duration_max=0.5*num_iq_samples_dataset/sample_rate,
    signal_bandwidth_min=sample_rate/10,
    signal_bandwidth_max=sample_rate/5,
    cochannel_overlap_probability=1,
}
```

2.4 GHz-like Congested Spectrum



More Dataset Customization

`num_signal_distribution` enables weighting for different numbers of signals

- `num_signals_min = 0`
- `num_signals_max = 3`
- By default, uniform distribution:
- `num_signals_distribution = [0.25, 0.25, 0.25, 0.25]`

But can change weighting as desired;

- `num_signals_distribution = [0.1, 0.25, 0.25, 0.4]`
- 10% chance for 0 signals
- 25% chance for 1 signal
- 25% chance for 2 signal
- 40% chance for 3 signals

Does not have to sum to 1 perfectly, internal algorithm will balance it

More Dataset Customization

- By default all 57 signals are created
- Signals in `class_list` are created with uniform probability
- Can be overwritten with `class_list_distribution`
- Determines likelihood that a specific modulation is created

- Example:
- `class_list = ['bpsk', 'qpsk', '16qam']`
- `class_list_distribution = [0.1, 0.2, 0.7]`
- 10% chance for BPSK, 20% chance for QPSK, 70% chance for 16-QAM

More Dataset Customization

- `class_list` parameter can be used to limit signals
- `from torchsig.signals.signal_lists import TorchSigSignalLists`
  - `.am_signals`
  - `.chirpss_signals`
  - `.constellation_signals`
  - `.fm_signals`
  - `.fsk_signals`
  - `.lfm_signals`
  - `.ofdm_signals`
  - `.tone_signals`
- `class_list` can then be the aggregate of multiple lists
  - `class_list = TorchSigSIgnalLists.am_signals + TorchSigSIgnalLists.fm_signals`
- Create a custom list by hand:
  - `class_list = [‘2gmsk’, ‘32qam’, ‘ofdm-1024’, ‘4fsk’]`

Iterable Dataset

- Previously;

```
for index in range(10):
    data, metadata = dataset[index]
```

- Now using an iterable dataset:

```
for index in range(10):
    data, metadata = next(dataset)
```

HDF5

- Datasets on disk now use HDF5
- Replacement for Zarr
- *Should* be faster for large datasets

Co-channel

- `cochannel_overlap_probability` does not ensure co-channel, but acts to prevent it
- Range is [0,1]; 0% to 100%
- Enabled when:
    - More than 1 signal is generated
    - When a signal is placed and has potential co-channel interference
    - Then determines the probability that the co-channel can remain, or if new placement is required

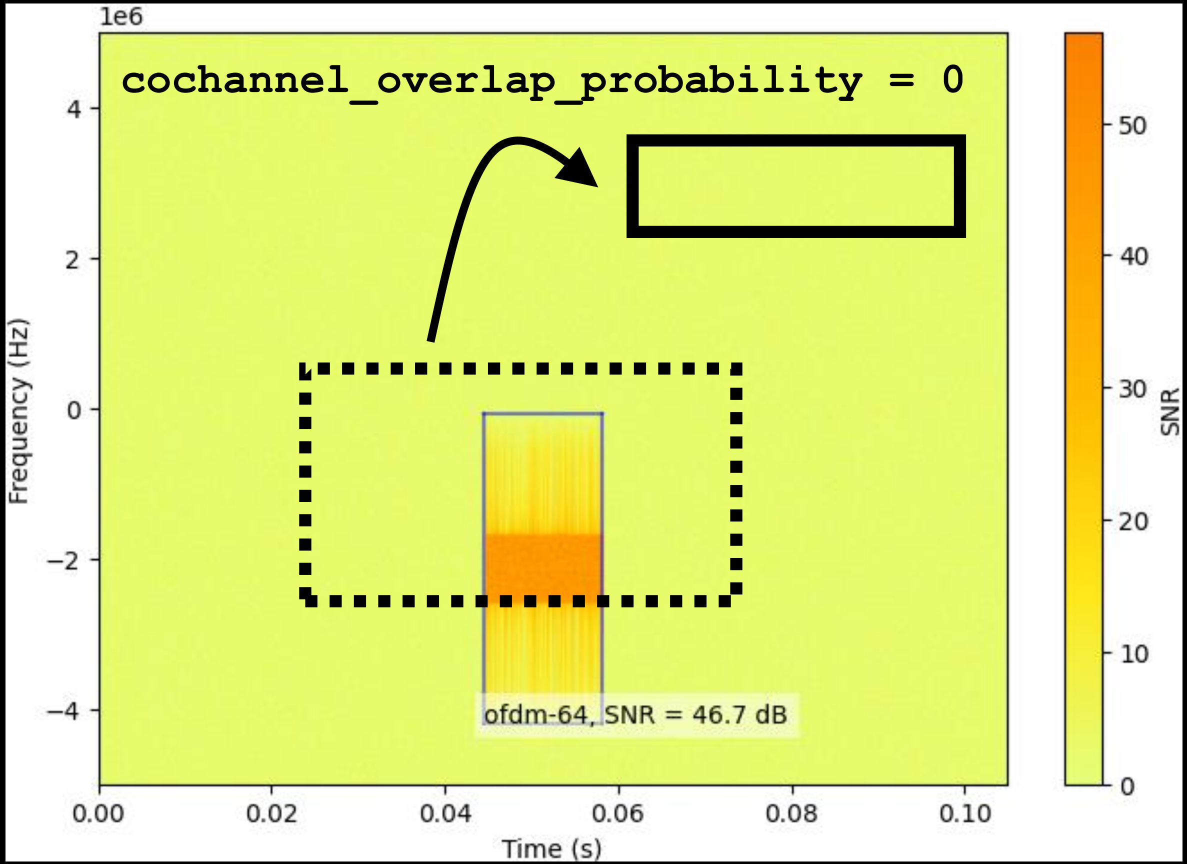
Co-channel

Attempting to place a second signal which would cause co-channel interference

Algorithm uses RNG with `cochannel_overlap_probability` to determine if signal stays

If not, then a new signal is placed elsewhere

`cochannel_overlap_probability = 0` does not allow for co-channel, so a second location must be found

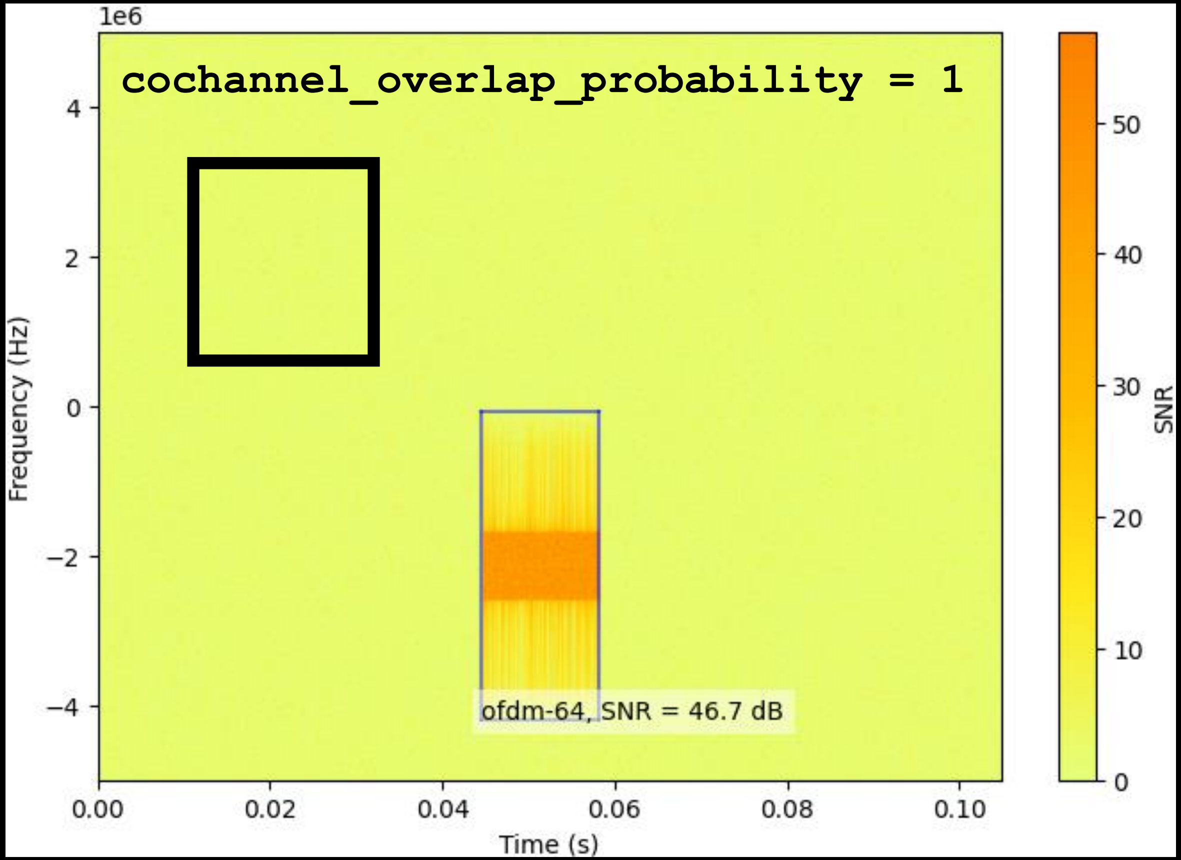


Co-channel

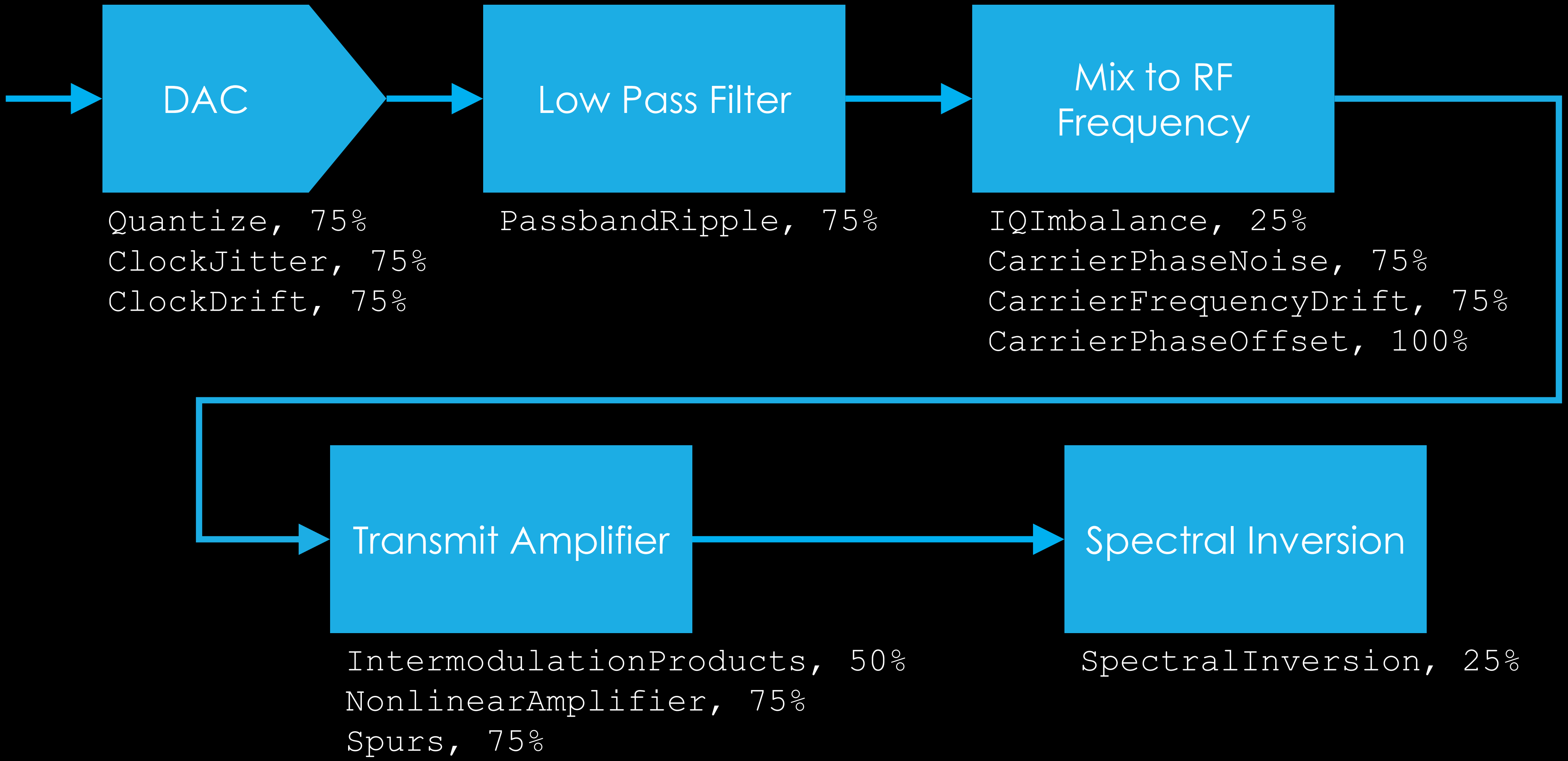
Does not enforce co-channel interference

Signal is placed, and if there is no co-channel, then it always stays

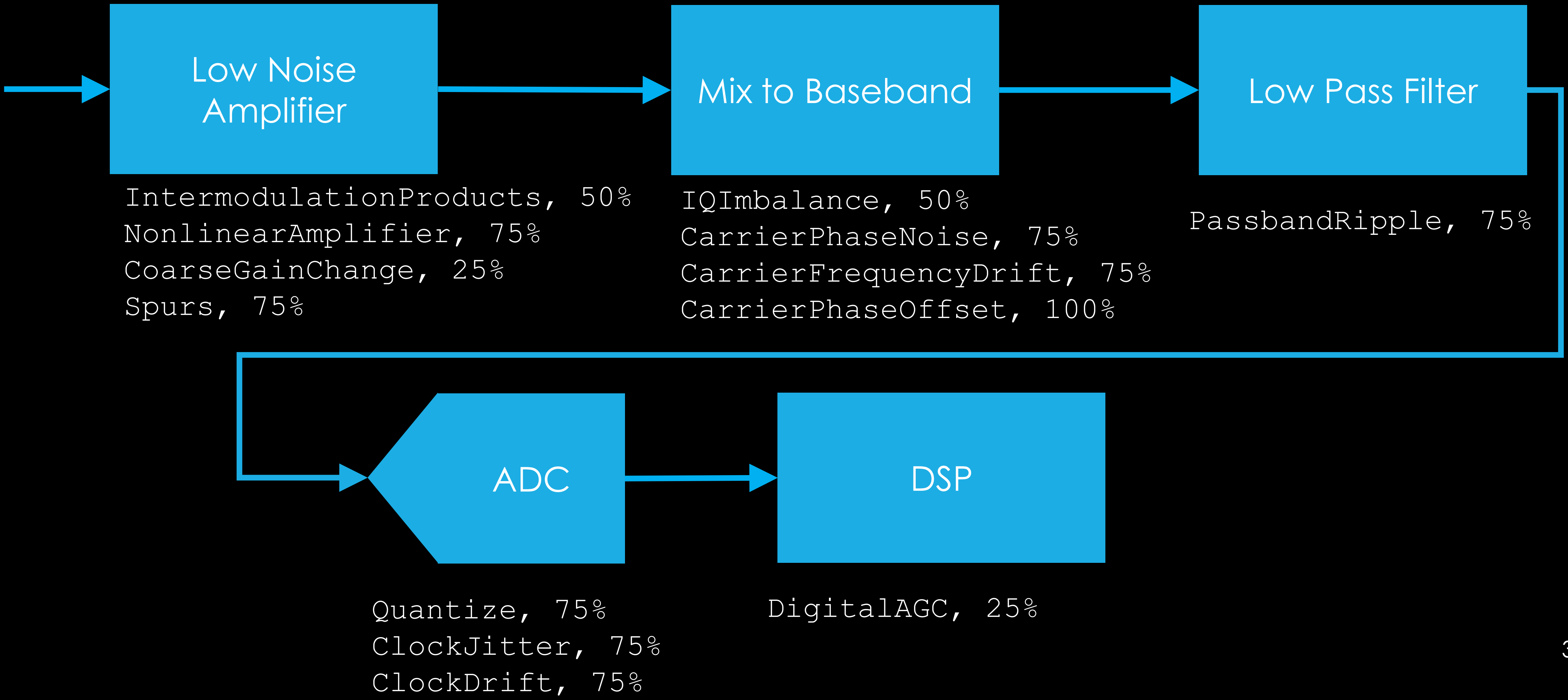
`cochannel_overlap_probability = 1` always allows for co-channel, but it does not force the location of the signal to cause it



Analog RF Impairments



Analog RF Impairments



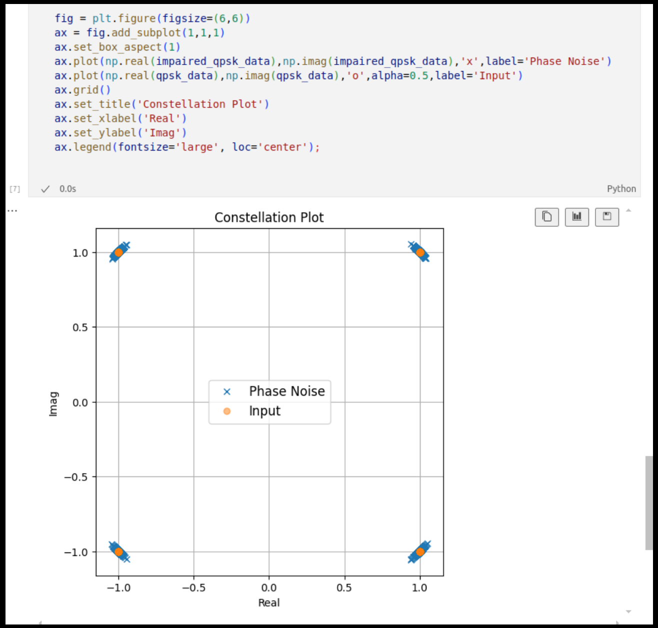
Analog RF Impairments

- level = 0
  - ML transforms
- level = 1
  - Analog RF Transmit impairments
  - Analog RF Receive impairments
  - ML transforms
- level = 2
  - Analog RF Transmit impairments
  - Transmit channel models
  - Analog RF Receive impairments
  - ML transforms

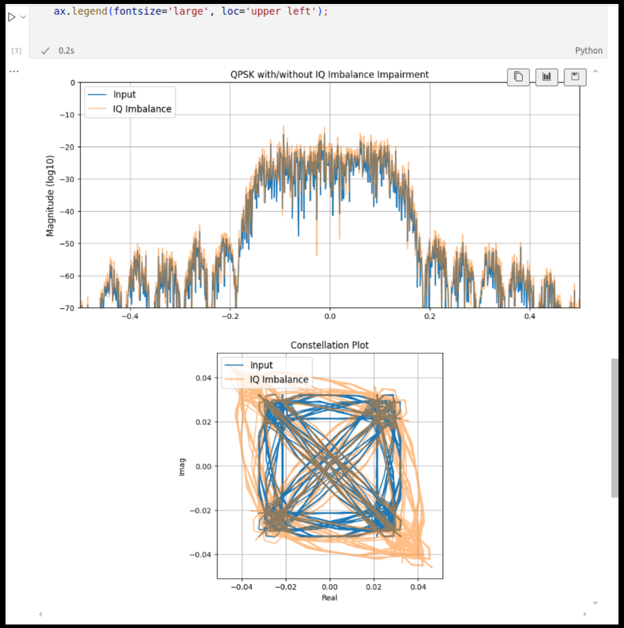
```python
from torchsig.transforms.impairments import Impairments

impairments = Impairments(level=1)
burst_impairments = impairments.signal_transforms
whole_signal_impairments = impairments.dataset_transforms

burst_impairments, whole_signal_impairments

dataset_impaired = TorchSigIterableDataset(
    dataset_metadata=dataset_finite_metadata,
    transforms=[whole_signal_impairments, Spectrogram(fft_size=fft_size)],
    component_transforms=[burst_impairments],
    target_labels=[])

dataset_unimpaired = TorchSigIterableDataset(
    dataset_metadata=dataset_finite_metadata,
    transforms=[Spectrogram(fft_size=fft_size)],
    component_transforms=[],
    target_labels=[])

dataset_impaired.seed(seed)
dataset_unimpaired.seed(seed)
```

Impairment Notebooks

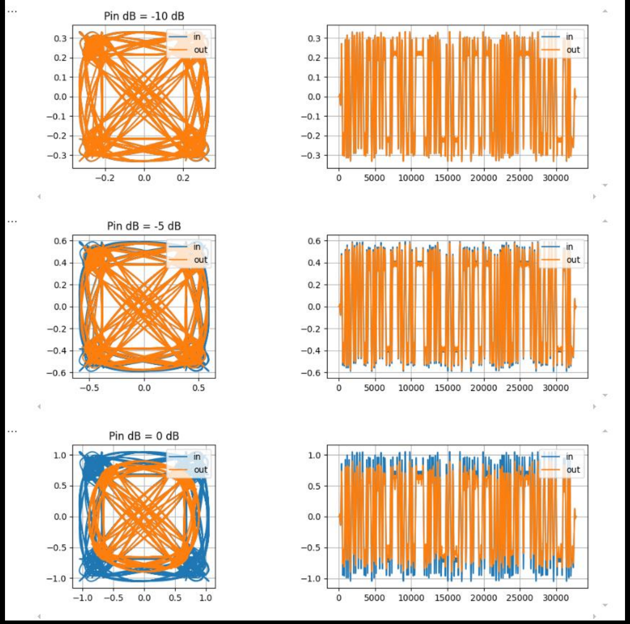
- Explanations and examples for analog RF impairments
- `examples/transforms` directory



Carrier Phase Noise



IQ Imbalance



Nonlinear Amplifier

Custom Datasets

- v2.0 adds an initial capability to read in externally generated datasets
- Can include synthetic, lab or real-world captures
- Requires defining your own metadata:
  - Sample rate?
  - Signal center frequency?
  - Signal bandwidth?
  - Signal start time?
  - Signal stop time?

Custom Datasets: NPY and SigMF

- Example Notebooks:
  - torchsig/examples/bring_your_own_data_npy_example.ipynb
  - torchsig/examples/bring_your_own_data_sigmf_example.ipynb

Importing External Data into TorchSig: Bring Your Own Data (BYOD) NumPy

This notebook shows how to import externally created data into TorchSig using a basic NumPy data plus JSON metadata example file format.

The main code that the user must write is a subclass of `ExternalFileHandler`, which will be passed into a `ExternalTorchSigDataset`. The `ExternalFileHandler` class must implement 3 methods:

Method	Arguments	Return	Description
`size`	N/A	int	Number of data samples, dataset size
`load_dataset_metadata`	N/A	`ExternalDatasetMetadata`	Dataset information, see `datasets/dataset_metadata.py` for more information.
`load`	idx: int	(np.ndarray, List[Any])	Load sample `idx`, which includes data as np.ndarray and taregts as a list.

If you want to apply TorchSig's transforms and impairments to your data, note that `load` must return targets that are in `List[Dict]` format, where each dict describes a signal. Additionally, the dict must have the fields required by each transform, e.g., `FamilyName` target transform requires the signal to have `class_name` in its metadata. It is up to the user to figure out what metadata is needed for what transforms/target transforms they wish to use.

```python
import numpy as np
import os
import csv
import json
from typing import Tuple, Dict, List, Any
import itertools
import pprint

# TorchSig
from torchsig.datasets.datasets import ExternalTorchSigDataset
from torchsig.datasets.dataset_metadata import ExternalDatasetMetadata
from torchsig.utils.file_handlers.base_handler import ExternalFileHandler
from torchsig.transforms.transforms import ComplexTo2D
```

Importing External Data into TorchSig: Bring Your Own Data (BYOD) SigMF

This notebook shows how to import externally created data into TorchSig using a basic SigMF example file format.

The main code that the user must write is a subclass of `ExternalFileHandler`, which will be passed into a `ExternalTorchSigDataset`. The `ExternalFileHandler` class must implement 3 methods:

Method	Arguments	Return	Description
`size`	N/A	int	Number of data samples, dataset size
`load_dataset_metadata`	N/A	`ExternalDatasetMetadata`	Dataset information, see `datasets/dataset_metadata.py` for more information.
`load`	idx: int	(np.ndarray, List[Any])	Load sample `idx`, which includes data as np.ndarray and taregts as a list.

If you want to apply TorchSig's transforms and impairments to your data, note that `load` must return targets that are in `List[Dict]` format, where each dict describes a signal. Additionally, the dict must have the fields required by each transform, e.g., `FamilyName` target transform requires the signal to have `class_name` in its metadata. It is up to the user to figure out what metadata is needed for what transforms/target transforms they wish to use.

```python
import numpy as np
import datetime as dt
import os
from sigmf import SigMFFile, sigmffile
from typing import Tuple, Dict, List, Any

# TorchSig
from torchsig.datasets.datasets import ExternalTorchSigDataset
from torchsig.datasets.dataset_metadata import ExternalDatasetMetadata
from torchsig.utils.file_handlers.base_handler import ExternalFileHandler
from torchsig.transforms.transforms import ComplexTo2D
```

Python

Custom Datasets: NPY and SigMF

- Two notebooks for reading NPY and SigMF
- How to:
  - Format when writing to disk
  - Write a custom file handler
  - Convert into dataset with metadata
  - Interact with custom-data dataset
- Requires some user coding to operate
- Does not yet integrate with TorchSig synthetic dataset generation
- Other data formats possible when using notebooks as guidelines

What's Next?

- Planned for v2.1+
  - Joining synthetic and external datasets for joint ML training
  - Updated pre-trained models
- What does the community want?

Questions?

bit.ly/torchsig-grcon-2025

TorchSig