



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

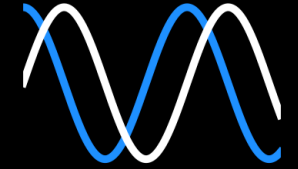
Frank Leonardo, Peraton Labs

Paul Toliver, Peraton Labs

Rob Miller, Peraton Labs

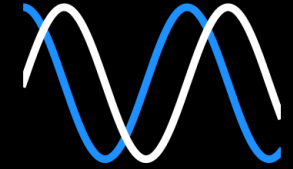


What is TorchSig?



- 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?



The screenshot shows the TorchSig website homepage. At the top, there's a navigation bar with links for Home, About, Downloads, and Documentation. The main header features the TorchSig logo and the tagline "A PyTorch Signal Processing Machine Learning Toolkit". Below this, there are buttons for "Get Started" and "GitHub", and a star count of 227. The "Key Features" section is divided into four columns: "Signals Datasets" (describing the TorchSigNarrowband and TorchSigWideband datasets), "Domain Transforms" (describing complex signal augmentations), "Pretrained Models" (describing the TorchSig API's integration with TorchVision and TorchAudio), and "Research & Development" (describing the open-source code, datasets, and documentation).

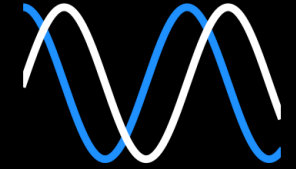
torchsig.com

The screenshot shows the TorchSig GitHub repository page. The repository is owned by TorchDSP and is public. It has 12 issues, 51 forks, and 227 stars. The "Code" tab is selected, showing a list of files and folders. The "About" section on the right describes TorchSig as an open-source signal processing machine learning toolkit based on the PyTorch data handling pipeline. It also shows the latest release, TorchSig v1.1.0, and the number of packages published (16) and contributors (11).

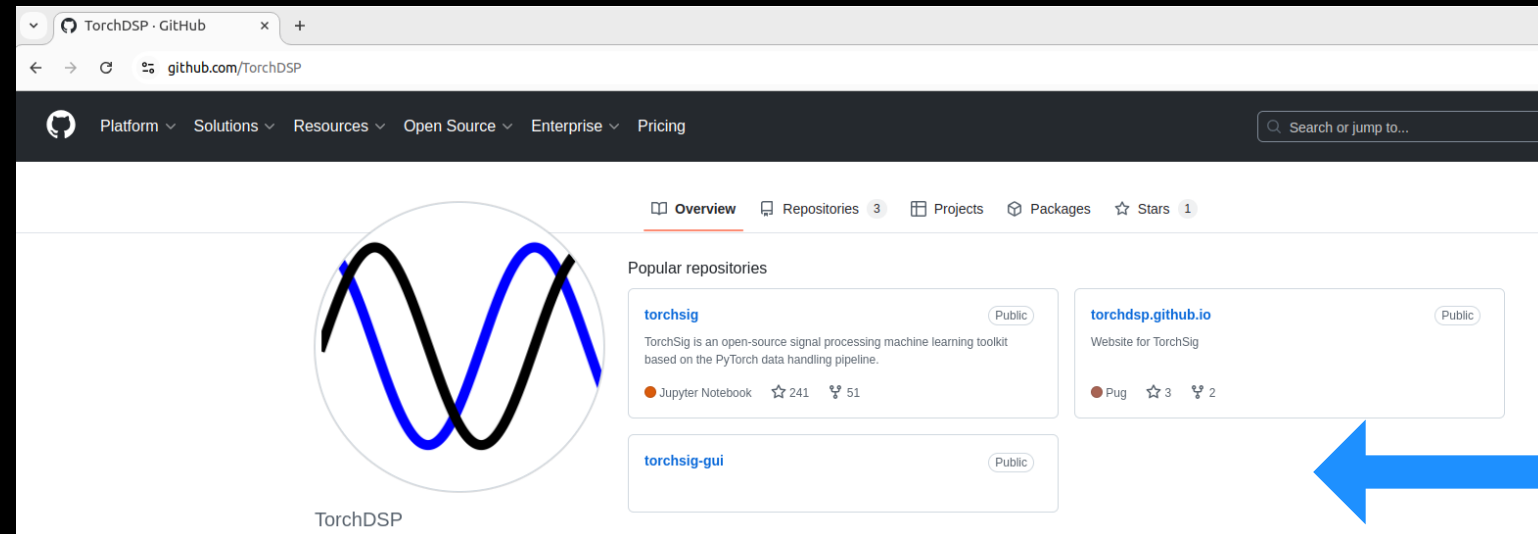
File/Folder	Commit Message	Commit Hash	Time Ago
.github	1.1.0 release (#375)	f8a4189	3 months ago
docs	update version in ReadTheDocs		3 months ago
examples	1.1.0 release (#375)		3 months ago
gr-spectrumdetect	1.1.0 release (#375)		3 months ago
scripts	1.1.0 release (#375)		3 months ago
tests	1.1.0 release (#375)		3 months ago
tools	1.1.0 release (#375)		3 months ago
torchsig	low_pass() works better for this instan...		3 months ago
.gitignore	1.1.0 release (#375)		3 months ago
.pylintrc	v1.0.0-beta release (#273)		5 months ago
.readthedocs.yaml	add pip install .		9 months ago
CONTRIBUTING.md	ready for v0.5.1 release		last year
DISCLAIMER.md	ready for v0.5.1 release		last year

github.com/torchdsp/torchsig

Repository Breakouts



- torchsig
- gr-spectrumdetect
- Others?



Prior Work



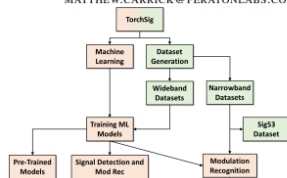
TorchSig: A GNU Radio Block and New Spectrogram Tools for Augmenting ML Training

Phil Vallance
Erebus Oh
Justin Mullins
Manbir Gulati
Jared Hoffman
Matt Carrick

PVALLANCE@LTSNET.NET
EOH@LTSNET.NET
JMULLINS1@LTSNET.NET
MANBIRGULATI@GMAIL.COM
JHOFFMAN@APPLIED-INSIGHT.COM
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

LARGE SCALE RADIO FREQUENCY WIDEBAND SIGNAL DETECTION & RECOGNITION

Luke Boegner¹, Garrett Vanhoy¹, Phillip Vallance², Manbir Gulati³, Dresden Feitzinger¹, Bradley Comar², and Robert D. Miller¹

¹Peraton Labs
²Laboratory for Telecommunication Sciences
³Applied Insight

{luke.boegner,gvanhoy,dresden.feitzinger,miller}@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 Vallance³,
Bradley Comar³, Silvija Kokalj-Filipovic¹, Craig Lennon³, Robert D. Miller¹

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

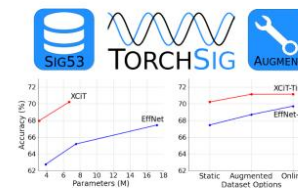


Figure 1. In this work, we introduce the Sig53 modulated signals

GRCCon 2024:
<https://events.gnuradio.org/event/24/contributions/628/>

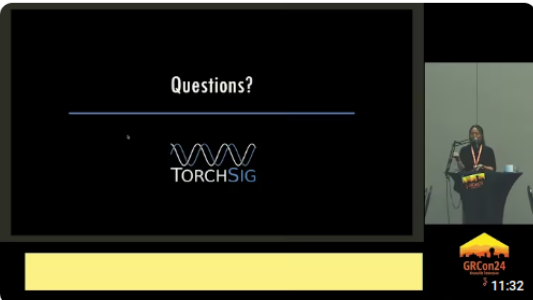
Arxiv 2022:
<https://arxiv.org/abs/2211.10335>

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

Original papers, many changes since publication

Prior Work






M. Carrick. TorchSig: A GNU Radio Block & New Tools for Augmenting ML Training with Real World Data
849 views • 4 months ago

GNU Radio

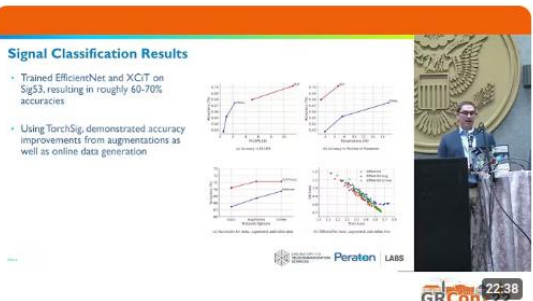
6:47 Added support for HuggingFace and PyTorch, enable broader use and accessibility of TorchSig models ...



GRCon23 - Updates to TorchSig An Open Source Signal Processing ML Toolkit - by Garrett Vanhoy
880 views • 1 year ago

GNU Radio

TorchSig, a toolkit for applying deep-learning applications to wireless signals, was released last year and presented at GRCon.



GRCon22 - Open-Source Large Scale RFML Dataset, Toolkit, Models - by Luke Boegner
654 views • 2 years ago

GNU Radio

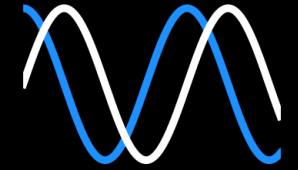
9:58 Using TorchSig, demonstrated accuracy improvements from augmentations as well as online data generation ...

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

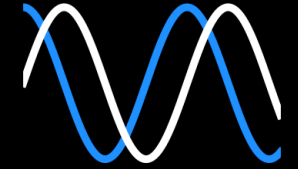
Chirp-based:

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

Unmodulated Carrier/Tone:

Tone

Dataset Generation



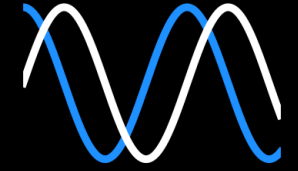
- Datasets are collections of IQ and their associated metadata
- IQ is stored in a NumPy array
- Metadata includes sample rate, signal bandwidths, signal center frequencies and others
- Datasets can be generated synthetically or with externally generated data (v2.0 only)

Transforms



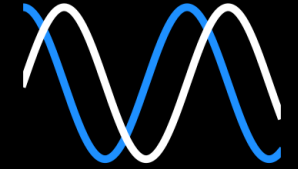
- Two categories of transforms:
 - Analog RF impairments
 - ML Transforms
- Analog RF Impairments:
 - Applies simulated real-world RF hardware effects
 - Examples: nonlinear amplifier compression, phase noise, IQ imbalance
- ML transforms:
 - Prepare and condition data for improved ML performance
 -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

ML Models

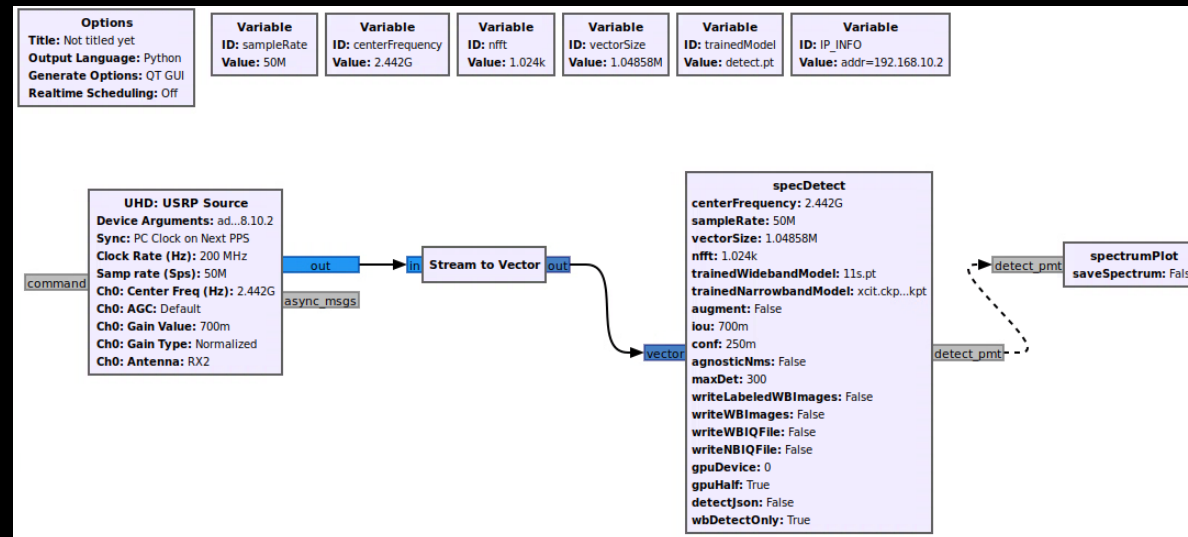


- Pre-trained Models:
 - `gr-spectrumdetect/examples/trained_model_download.sh`
 - `11s.pt`: YOLOv11 for energy detection on wideband spectrograms
 - `detect.pt`: YOLOv8 for energy detection on wideband spectrograms
 - `xcit.ckpt`: XCiT model trained for narrowband mod-rec
- Ability to train custom models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

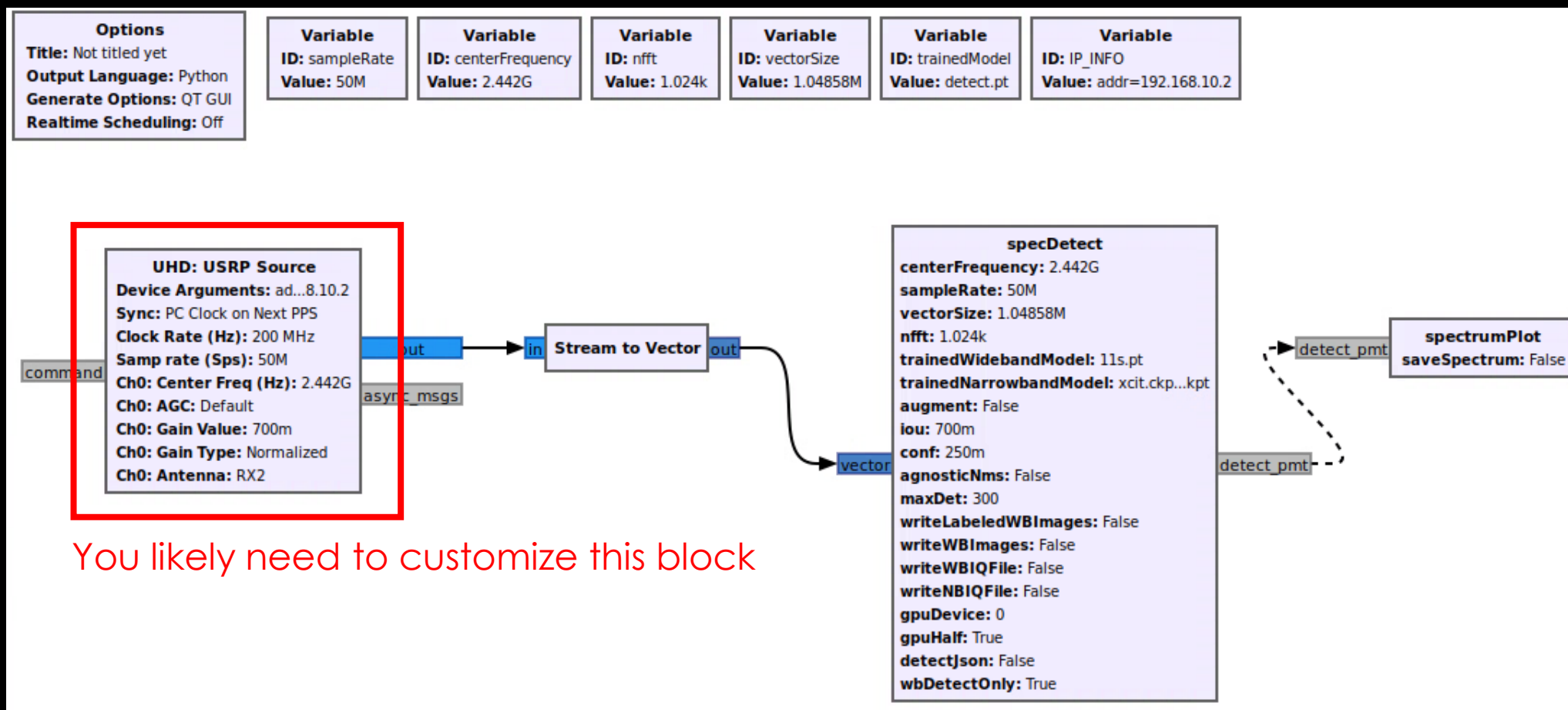
gr-spectrumdetect



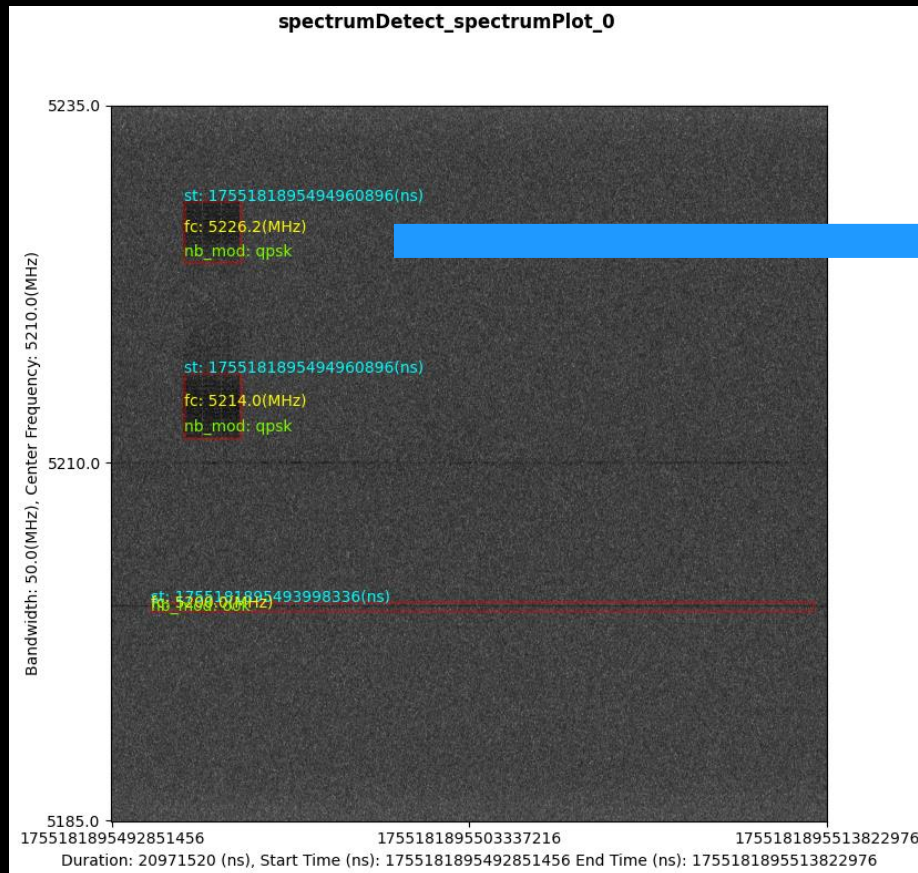
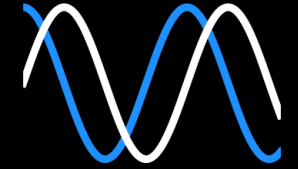
- GNU Radio Flowgraph and block
- Energy detection in spectrogram
- Modulation recognition (improvements under development)



gr-spectrumdetect



gr-spectrumdetect



st: 1755181895494960896(ns)
fc: 5226.2(MHz)
nb_mod: qpsk

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

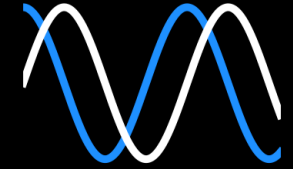
Tutorial Notebooks



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`

```
Wideband Dataset

This notebook showcases the Wideband dataset.

# Define Variables
num_iq_samples_dataset = 4096 # 64^2
fft_size = 64
impairment_level = 0 # clean
num_signals_max = 5
num_signals_min = 0

# Dataset Metadata
In order to create a NewTorchSigDataset, you must define parameters in DatasetMetadata. This can be done either in code or inside a YAML file. Below we show how to do both. Look at create_dataset_example.yaml for a sample YAML file.

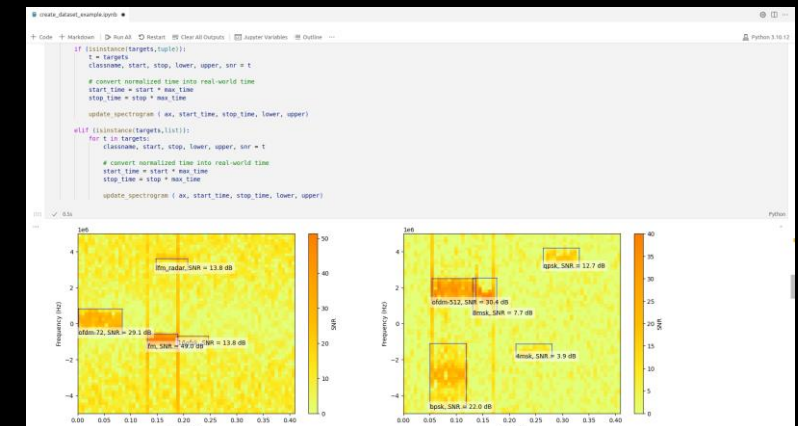
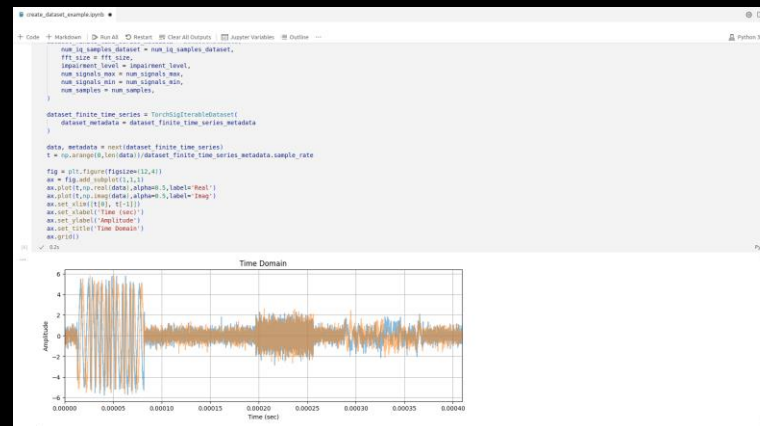
There are four required parameters:
1. num_iq_samples_dataset - how much IQ data per sample
2. fft_size - size of FFT (number of bins) to be used in spectrogram.
3. impairment_level - what environment impairment to simulate
4. num_signals_max - maximum number of signals per sample.

Additionally, there are several optional parameters that can be overridden.

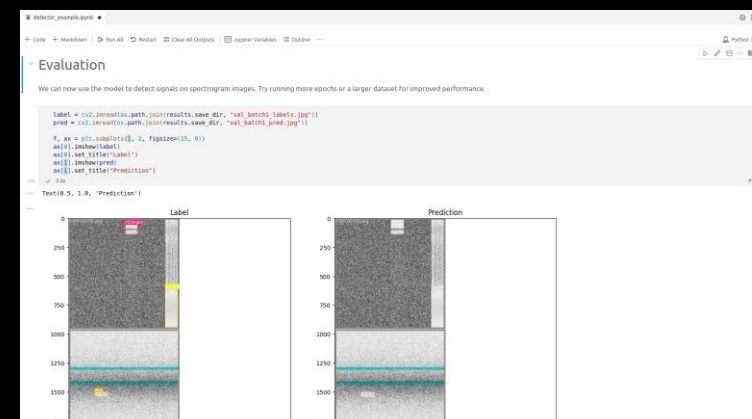
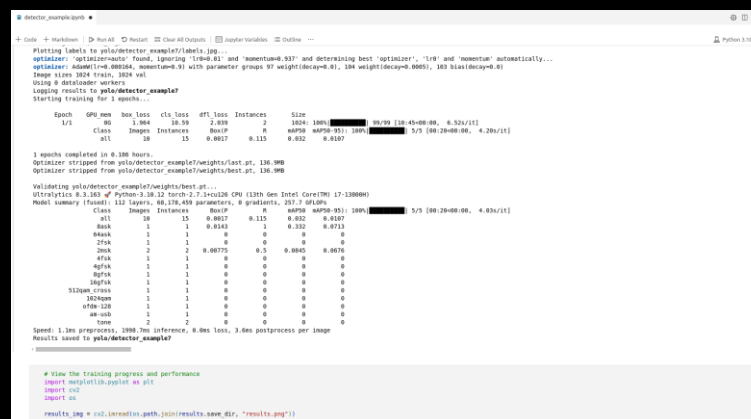
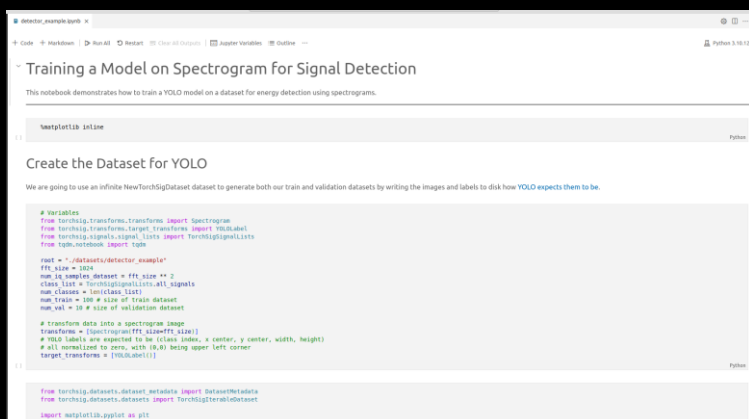
# Option 1: Instantiate DatasetMetadata object
from torchsig.datasets.dataset_metadata import DatasetMetadata

dataset_metadata_1 = DatasetMetadata(
    num_iq_samples_dataset = num_iq_samples_dataset,
    fft_size = fft_size,
    impairment_level = impairment_level,
    num_signals_max = num_signals_max,
    num_signals_min = num_signals_min
)

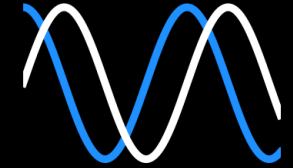
print(dataset_metadata_1)
```



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



Tutorial Notebooks



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

```
Training a Model on IQ Samples for Classification

This notebook demonstrates how to train a PyTorch model on IQ Samples for modulation recognition.

# Variables
from torchsig.signals import TorchSigSignalList
from torchsig.transforms import TorchSigTransform
from torchsig.transforms.target_transforms import TorchSigTargetTransform

root = "data/classifier_example"
fft_size = 256
num_iq_samples_dataset = fft_size ** 2
class_list = TorchSigSignalList.all_signals
family_list = TorchSigSignalList.family_list
num_classes = len(class_list)
num_samples_train = len(class_list) * 10 # roughly 10 samples per class
num_samples_val = len(class_list) * 2
impairment_level = 0
seed = 123456789
# 10-based mod-rec only operates on 1 signal
num_signals_max = 1
num_signals_min = 1

# complex2D turns a 1D array of complex values into a 2D array, with one channel for the real component, while the other is for the imaginary component
transforms = [complex2D]
# classnames form our target labels into the index of the class according to class_list
target_transform = TorchSigTargetTransform()

# Create the Dataset
from torchsig.datasets.dataset_metadata import DatasetMetadata
from torchsig.datasets.dataloader import TorchSigDataModule

train_metadata = DatasetMetadata(
    num_iq_samples_dataset = num_iq_samples_dataset,
    fft_size = fft_size,
    impairment_level = impairment_level,
    class_list = class_list,
    target_transform = target_transform,
)
```

```
Create the Model

We use our own KCTF model code and utils, but this can be replaced with your own model architecture in PyTorch, Ultralytics, timm, etc.

from torchsig.models import KCTFClassifier
from torchsig import summary

model = KCTFClassifier(
    input_channels=2,
    num_classes=num_classes,
)

summary(model)

# 1/30
Layer (type-depth-idx)      Param #
-----
KCTFClassifier             --
  KCTFNet: 1-1              --
    -Res1: 2-1              380
    -Conv2dLayer: 3-1        12,864
    -MaxpoolingLayer: 3-2    12,480
    -Res2: 3-3              5,486,824
    -ModelList: 3-4          896,400
    -ModelList: 3-5          384
    -LayerNorm: 3-6          --
    -Dropout: 3-7            --
    -Identity: 3-8           11,864
    -Conv2d: 3-2            --
  -PoolLayer: 3-2           --

Total params: 6,398,064
Trainable params: 6,398,064
Non-trainable params: 0

Train the Model

Using the PyTorch Lightning Trainer, we can train our model for modulation recognition on IQ dataset.
```

```
# We can do this over the whole test dataset to check the accuracy of our model
predictions = []
true_classes = []
num_correct = 0
device = torch.device("cpu" if torch.cuda.is_available() else "cpu")

for sample in test_dataset:
    data, actual_class = sample
    model.to(device)
    with torch.no_grad():
        data = torch.from_numpy(data).to(device).unsqueeze(0)
        pred = model(data)
        predicted_class = torch.argmax(pred).numpy()
        predictions.append(predicted_class)
        true_classes.append(actual_class)
        if predicted_class == actual_class:
            num_correct += 1

# We try increasing num_epochs or train dataset size to increase accuracy
print(f"Correct Predictions = {num_correct}")
print(f"Percent Correct = {num_correct / len(test_dataset)}")

# We can also plot a confusion matrix using sklearn's confusion matrix tool
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

matrix = confusion_matrix(true_classes, predictions, labels=list(range(len(family_list))))
disp = ConfusionMatrixDisplay(matrix, display_labels=family_list)
disp.plot()

sklearn.metrics.plot_confusion_matrix(ConfusionMatrixDisplay, 8x75276163836)

cm = ConfusionMatrixDisplay(matrix, display_labels=family_list)
cm.plot()
```

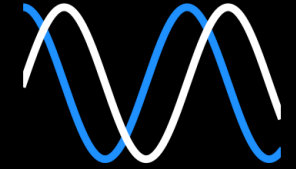
TorchSig 2.0



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

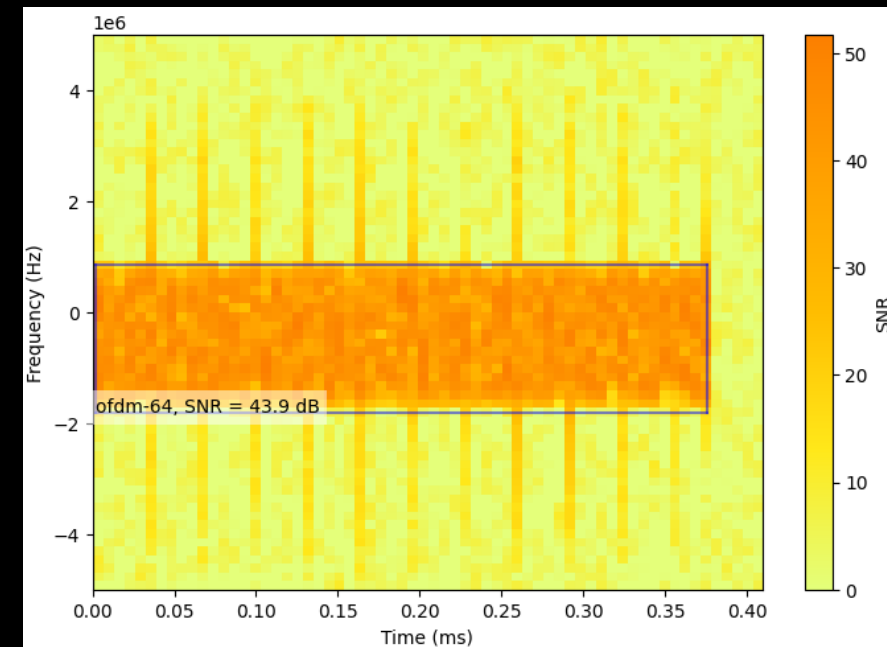


Previously:

```
narrowband_metadata = NarrowbandMetadata(...)
```

```
narrowband_dataset = NewNarrowband(narrowband_metadata)
```

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



WB/NB Unification

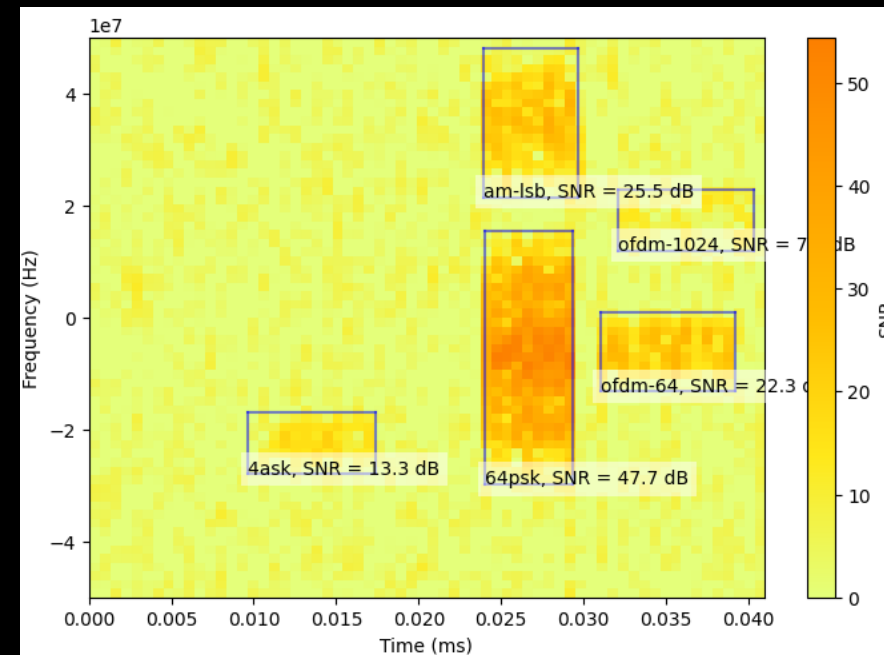


Previously:

```
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:

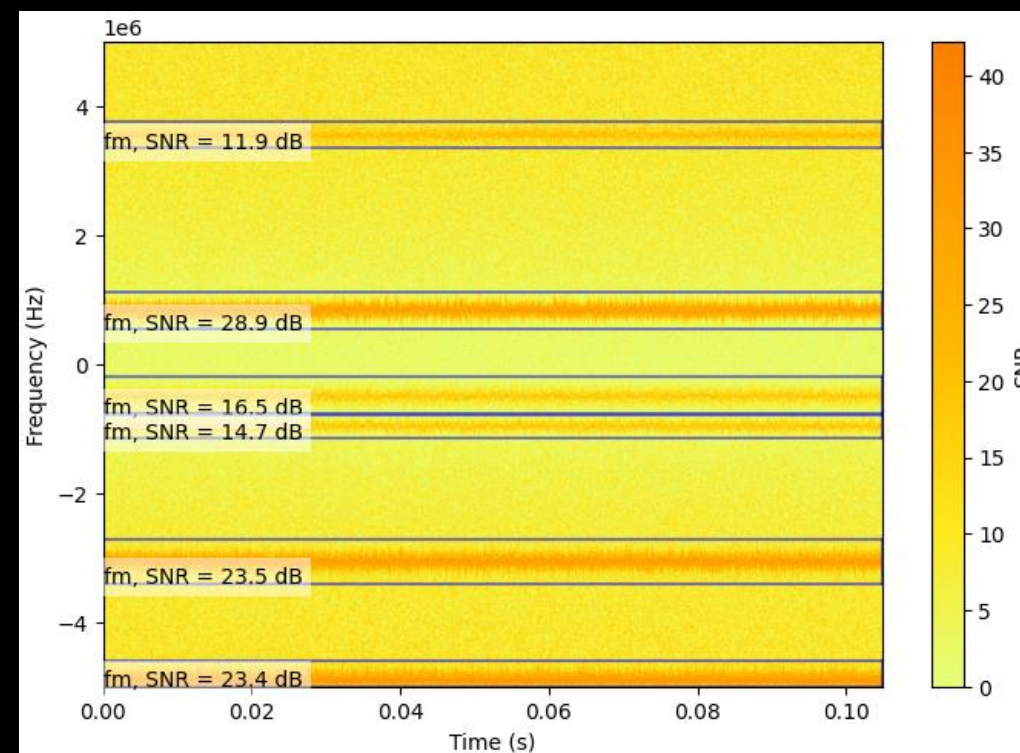
- signal_center_freq_min
- signal_center_freq_max

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

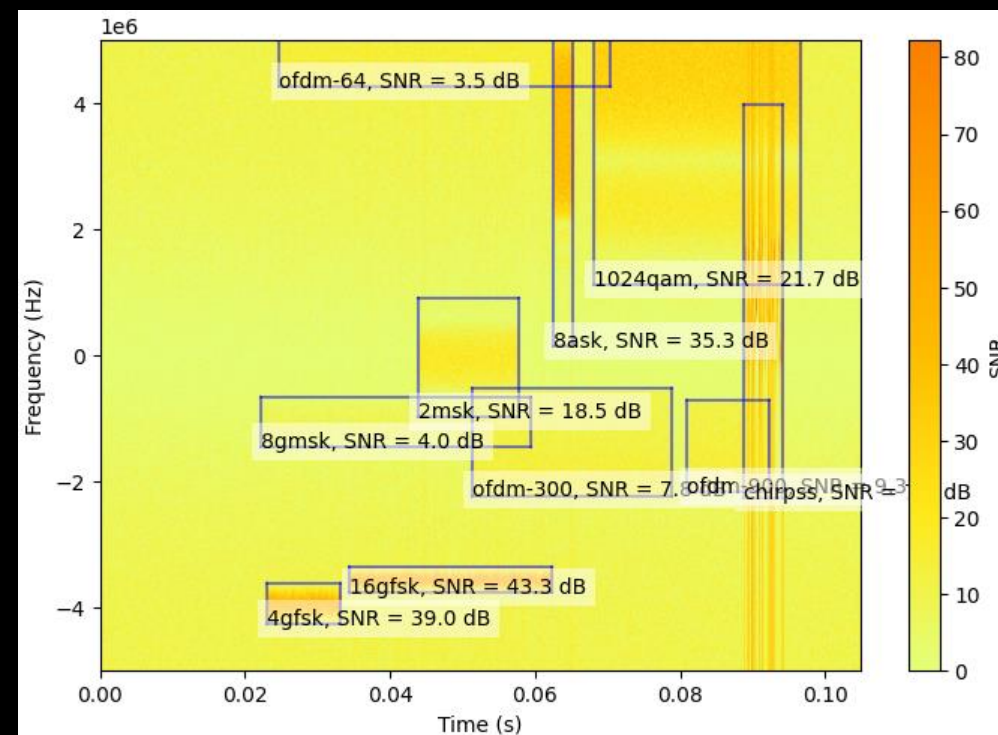


Dataset Customization



```
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

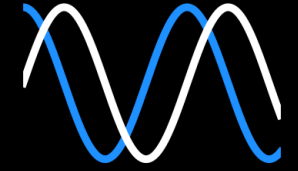
- `num_signals_min = 0`
- `num_signals_max = 3`
- By default, uniform distribution:
- `num_signals_distribution = [0.25, 0.25, 0.25, 0.25]`

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



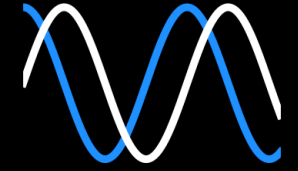
- By default all 57 signals are created
- Signals in `class_list` are created with uniform probability
- Can be overwritten with `class_list_distribution`
- Determines likelihood that a specific modulation is created
- Example:
 - `class_list = ['bpsk', 'qpsk', '16qam']`
 - `class_list_distribution = [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



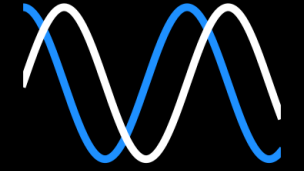
- Previously;

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

- Now using an iterable dataset:

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

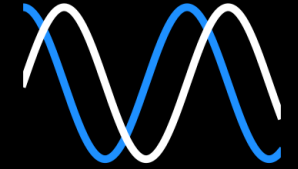
HDF5



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

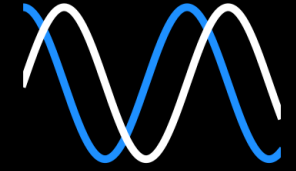


Co-channel



- `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

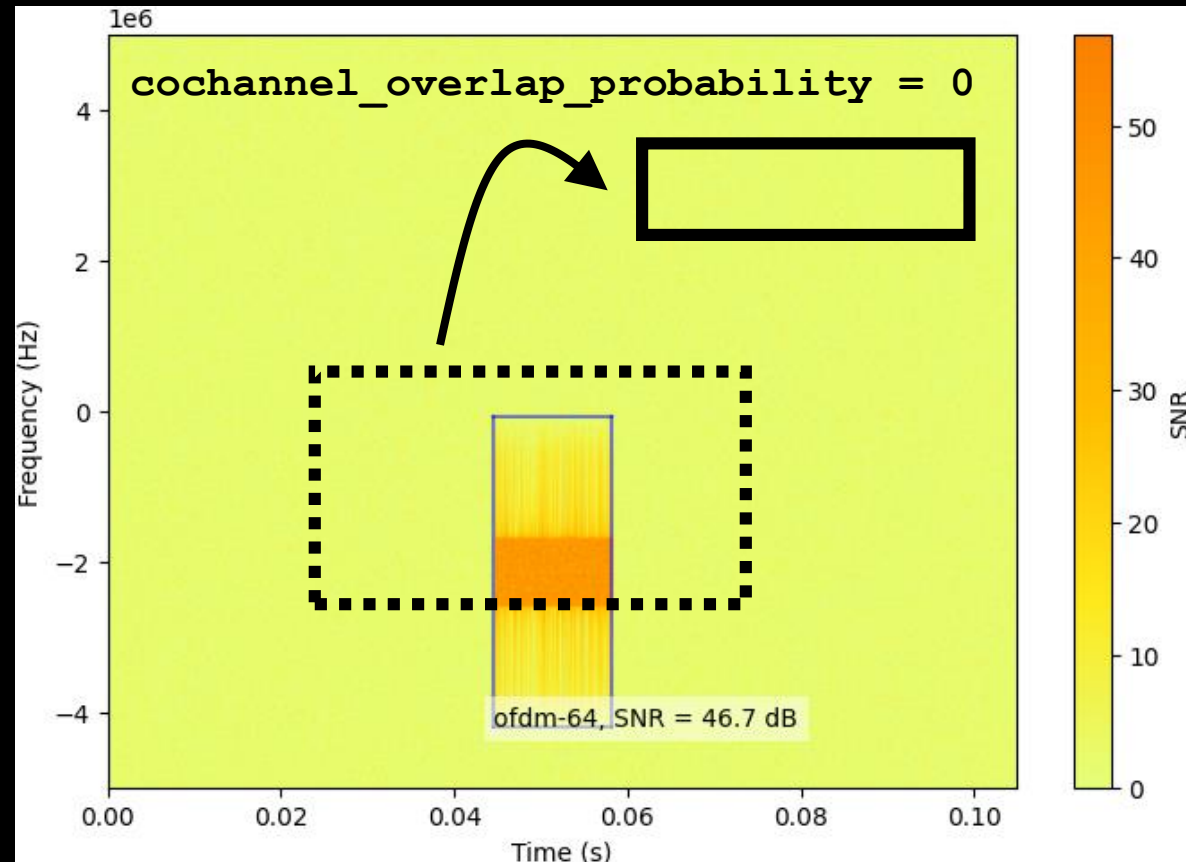


Attempting to place a second signal which would cause co-channel interference

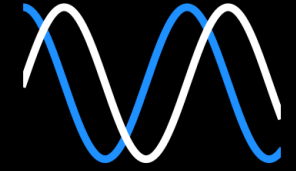
Algorithm uses RNG with `cochannel_overlap_probability` to determine if signal stays

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



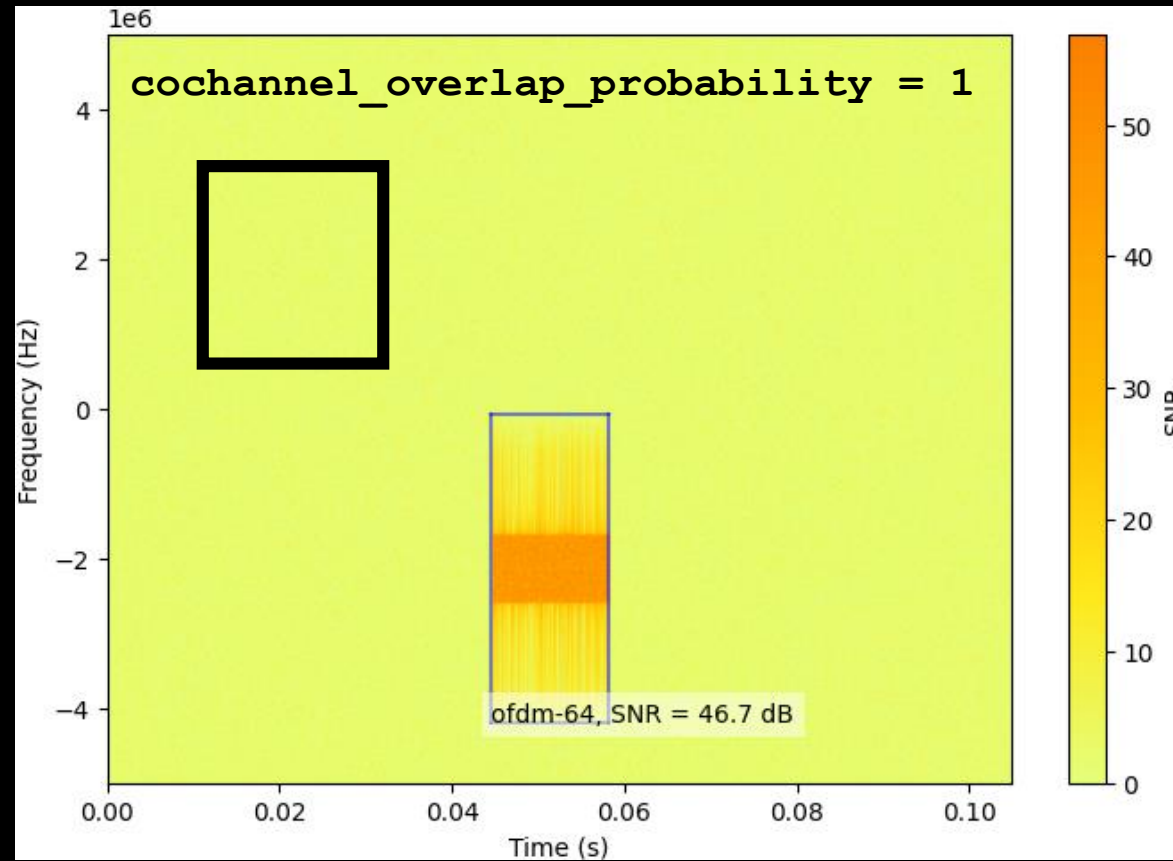
Co-channel



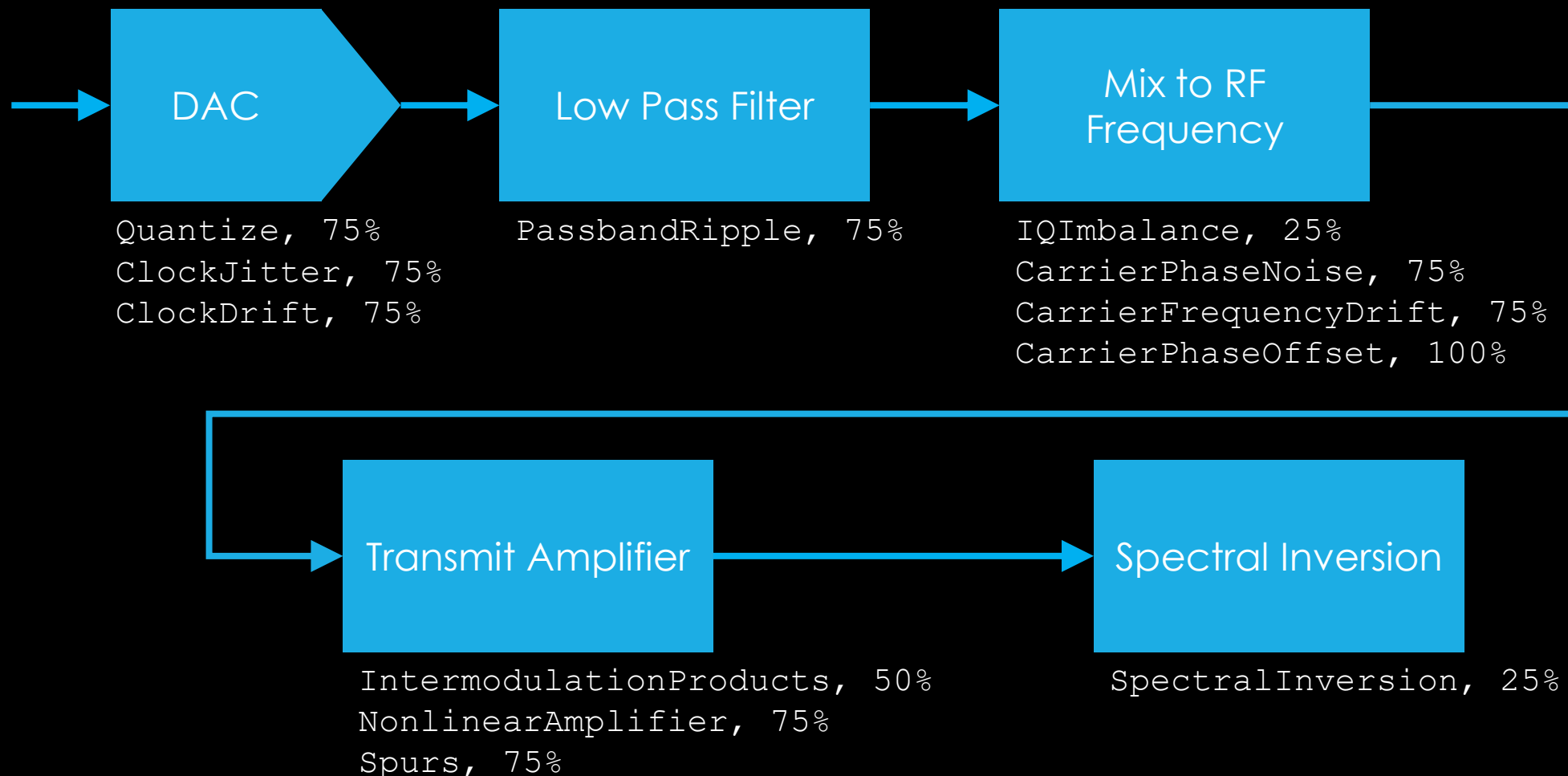
Does not enforce co-channel interference

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

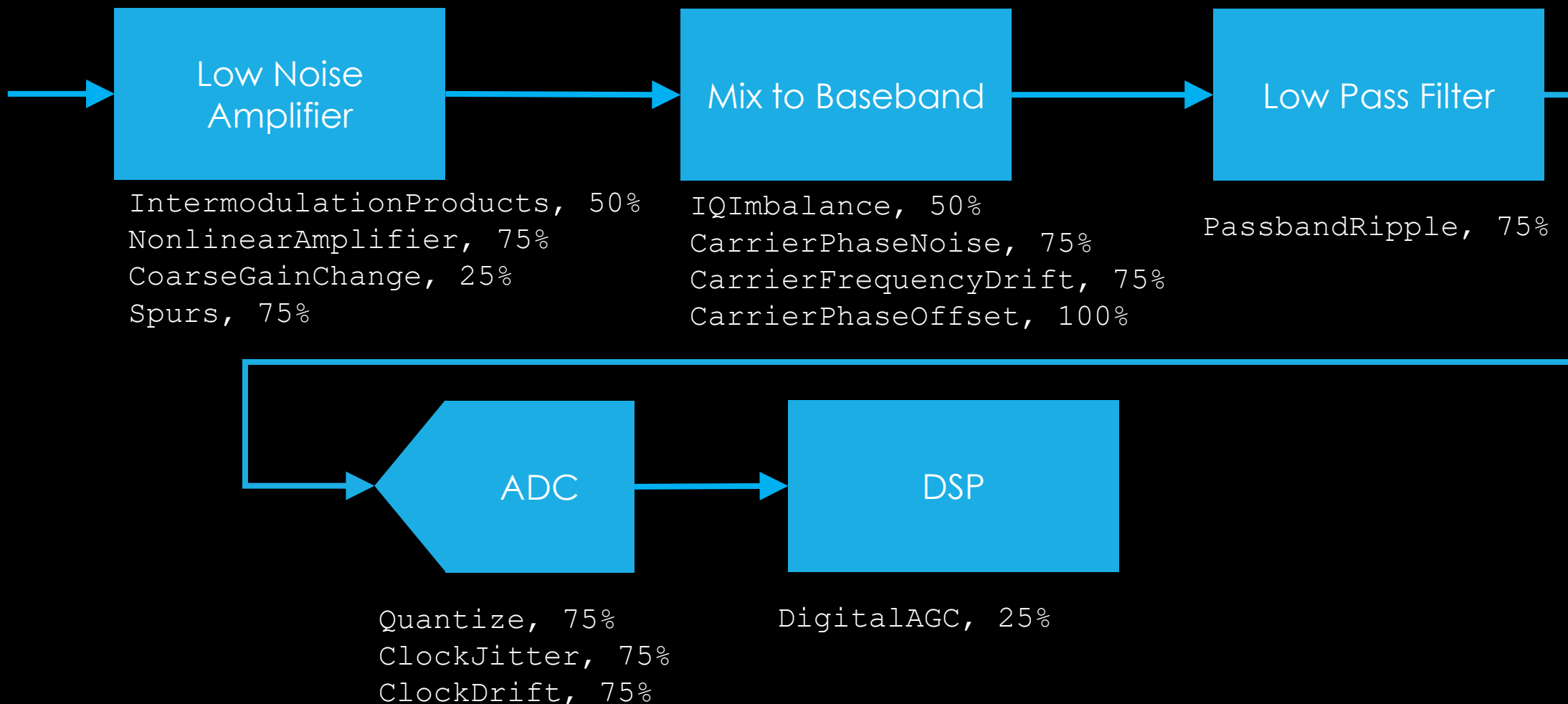
`cochannel_overlap_probability = 1`
always allows for co-channel, but it does not force the location of the signal to cause it



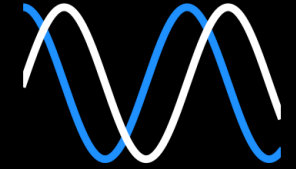
Analog RF Impairments



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

```
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=[])

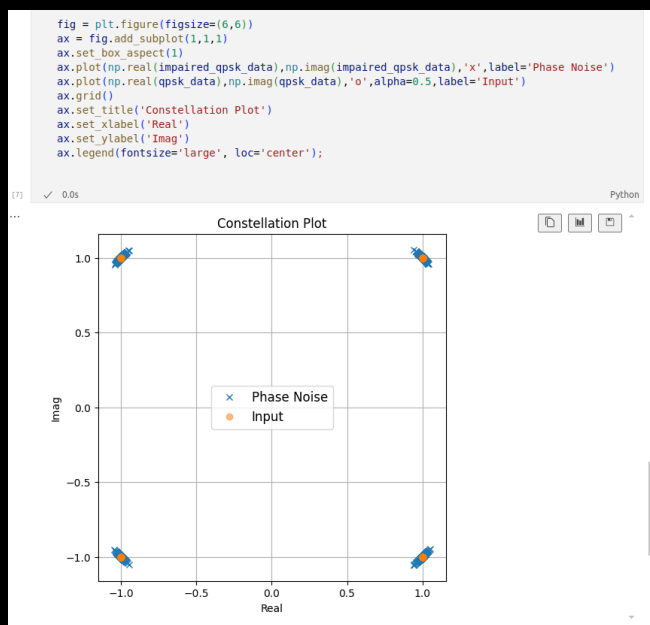
dataset_unimpaired = TorchSigIterableDataset(
    dataset_metadata=dataset_finite_metadata,
    transforms=[Spectrogram(fft_size=fft_size)],
    component_transforms=[],
    target_labels=[])

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

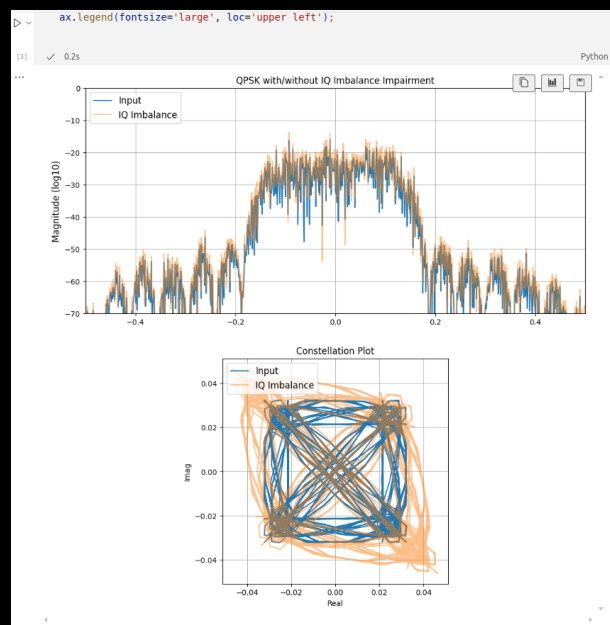
Impairment Notebooks



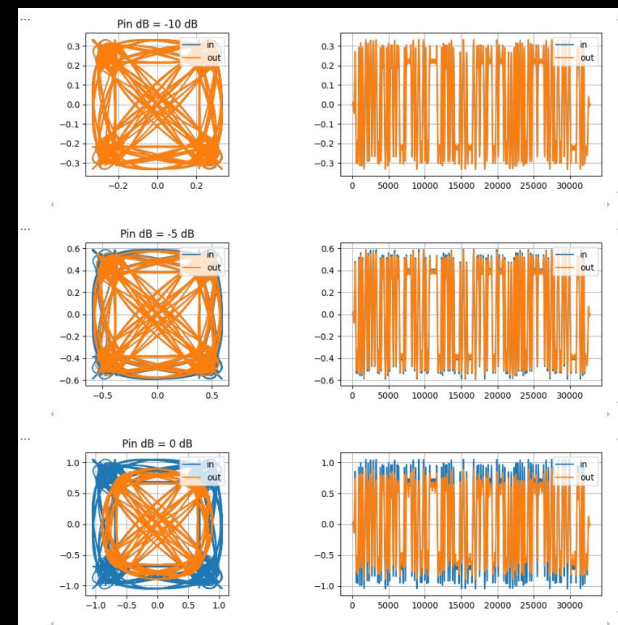
- Explanations and examples for analog RF impairments
- examples/transforms directory



Carrier Phase Noise

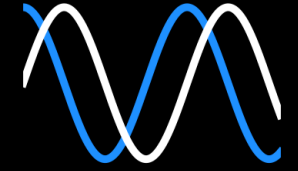


IQ Imbalance



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
<code>size</code>	N/A	int	Number of data samples, dataset size
<code>load_dataset_metadata</code>	N/A	<code>ExternalDatasetMetadata</code>	Dataset information, see <code>datasets/dataset_metadata.py</code> for more information.
<code>load</code>	<code>idx: int</code>	<code>(np.ndarray, List[Any])</code>	Load sample <code>idx</code> , which includes data as <code>np.ndarray</code> and targets 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.

```
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
<code>size</code>	N/A	int	Number of data samples, dataset size
<code>load_dataset_metadata</code>	N/A	<code>ExternalDatasetMetadata</code>	Dataset information, see <code>datasets/dataset_metadata.py</code> for more information.
<code>load</code>	<code>idx: int</code>	<code>(np.ndarray, List[Any])</code>	Load sample <code>idx</code> , which includes data as <code>np.ndarray</code> and targets 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.

```
import numpy as np
import datetime as dt
import os
from sigmf import SigMFFile, sigmf_file
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

