mmwave Documentation

Release 0.9.0

Arjun, Dash, Edwin, Jingning, Ruihao

Mar 17, 2020

Contents

1 mmWave Processing Library Reference **3**
- 1.1 mmwave.dataloader . . . 3
- 1.2 mmwave.dsp . . . 6
- 1.3 mmwave.tracking . . . 27
- 1.4 mmwave.clustering . . . 28

2 Indices and tables **31**

Index **33**

CHAPTER 1

mmWave Processing Library Reference

1.1 mmwave.dataloader

1.1.1 mmwave.dataloader.adc

class mmwave.dataloader.adc.**DCA1000**(*static_ip='192.168.33.30'*, *adc_ip='192.168.33.180'*, *data_port=4098*, *config_port=4096*)

Software interface to the DCA1000 EVM board via ethernet.

static_ip

IP to receive data from the FPGA

Type str

adc_ip

IP to send configuration commands to the FPGA

Type str

data_port

Port that the FPGA is using to send data

Type int

config_port

Port that the FPGA is using to read configuration commands from

Type int

General steps are as follows:

1. Power cycle DCA1000 and XWR1xxx sensor
2. Open mmWaveStudio and setup normally until tab SensorConfig or use lua script
3. Make sure to connect mmWaveStudio to the board via ethernet
4. Start streaming data

5. Read in frames using class

Examples

```
>>> dca = DCA1000()
>>> adc_data = dca.read(timeout=.1)
>>> frame = dca.organize(adc_data, 128, 4, 256)
```

close()
Closes the sockets that are used for receiving and sending data

Returns None

configure()
Initializes and connects to the FPGA

Returns None

static organize(*raw_frame*, *num_chirps*, *num_rx*, *num_samples*)
Reorganizes raw ADC data into a full frame

Parameters

- **raw_frame** (*ndarray*) – Data to format
- **num_chirps** – Number of chirps included in the frame
- **num_rx** – Number of receivers used in the frame
- **num_samples** – Number of ADC samples included in each chirp

Returns Reformatted frame of raw data of shape (num_chirps, num_rx, num_samples)

Return type ndarray

read(*timeout=1*)
Read in a single packet via UDP

Parameters **timeout** (*float*) – Time to wait for packet before moving on

Returns Full frame as array if successful, else None

1.1.2 mmwave.dataloader.utils

mmwave.dataloader.utils.**parse_tsw1400**(*path*, *num_chirps_per_frame*, *num_frames*, *num_ants*, *num_adc_samples*, *iq=True*, *num_adc_bits=16*)
Parse the raw ADC data based on xWR16xx/IWR6843 and TSW1400 configuration.

Parse the row-majored binary output from raw ADC data capture to numpy ndarray with the shape of (numFrame, num_chirps_per_frame, num_ants, num_adc_samples). For more details, refer to the original document https://www.ti.com/lit/an/swra581b/swra581b.pdf.

Parameters

- **path** (*str*) – File path of the binary data.
- **num_chirps_per_frame** (*int*) – Total number of chirps from all transmitters in a single frame.
- **num_frames** (*int*) – Number of frames in the recorded binary data.
- **num_ants** (*int*) – Number of physical receivers.

- **num_adc_samples** (*int*) – Number of ADC samples.
- **iq** (*bool*) – True if complex and False if real.
- **num_adc_bits** (*int*) – Number of ADC quantization bits.

Returns Parsed ADC data with the shape of (num_frames, num_chirps_per_frame, num_ants, num_adc_samples)

Return type ndarray

Example

```
>>> # Suppose your binary data is located at "./data/radar_data.bin".
>>> adc_data = parse_tsw1400("./data/radar_data.bin", 128, 200, 4, 256)
>>> # Now your adc_data will be an ndarray with shape (200, 128, 4, 256) and
↪dtype as complex.
```

1.1.3 mmwave.dataloader.radars

class mmwave.dataloader.radars.**TI**(*sdk_version=2.0, cli_loc='COM6', cli_baud=115200, data_loc='COM5', data_baud=921600, num_rx=4, num_tx=2, verbose=False, connect=True, mode=0*)

Software interface to a TI mmWave EVM for reading TLV format. Based on TI's SDKs

sdk_version
Version of the TI SDK the radar is using

cli_port
Serial communication port of the configuration/user port

data_port
Serial communication port of the data port

num_rx_ant
Number of RX (receive) antennas being utilized by the radar

num_tx_ant
Number of TX (transmit) antennas being utilized by the radar

num_virtual_ant
Number of VX (virtual) antennas being utilized by the radar

verbose
Optional output messages while parsing data

connected
Optional attempt to connect to the radar during initialization

mode
Demo mode to read different TLV formats

close()
End connection between radar and machine

Returns None

sample()
Samples byte data from the radar and converts it to decimal

1.1.4 mmwave.dataloader.file_parse

`mmwave.dataloader.file_parse.`**`parse_raw_adc`**(*source_fp*, *dest_fp*)

Reads a binary data file containing raw adc data from a DCA1000, cleans it and saves it for manual processing.

Note: "Raw adc data" in this context refers to the fact that the DCA1000 initially sends packets of data containing meta data and is merged with actual pure adc data. Part of the purpose of this function is to remove this meta data.

Parameters

- **source_fp** (*str*) – Path to raw binary adc data.
- **dest_fp** (*str*) – Path to output cleaned binary adc data.

Returns None

1.2 mmwave.dsp

1.2.1 mmwave.dsp.angle_estimation

`mmwave.dsp.angle_estimation.`**`azimuth_processing`**(*radar_cube*, *det_obj_2d*, *config*, *window_type_2d=None*)

Calculate the X/Y coordinates for all detected objects.

The following procedures will be performed in this function:

1. Filter radarCube based on the range indices from detObj2D and optional clutter removal.
2. Re-do windowing and 2D FFT, select associated doppler indices to form the azimuth input.
3. Doppler compensation on the virtual antennas related to tx2. Save optional copy for near field compensation and velocity disambiguation.
4. Perform azimuth FFT.
5. Optional near field correction and velocity disambiguation. Currently mutual exclusive.
6. Magnitude squared.
7. Calculate X/Y coordinates.

Parameters

- **radar_cube** – (numChirpsPerFrame, numRxAntennas, numRangeBins). Because of the space limitation, TI Demo starts again from the 1D FFT, recalculate the 2D FFT on the selected range bins and pick the associated doppler bins for the azimuth FFT.
- **det_obj_2d** – (numDetObj, 3)
- **config** – [TBD]
- **window_type_2d** – windowing function for the 2D FFT. Default is None.

Returns (numDetObj, 5). Copy of detObj2D but with populated X/Y coordinates.

Return type azimuthOut

`mmwave.dsp.angle_estimation.`**`aoa_bartlett`**(*steering_vec*, *sig_in*, *axis*)

Perform AOA estimation using Bartlett Beamforming on a given input signal (sig_in). Make sure to specify the correct axis in (axis) to ensure correct matrix multiplication. The power spectrum is calculated using the following equation:

$$P_{ca}(\theta) = a^H(\theta) R_{xx}^{-1} a(\theta)$$

This steers the beam using the steering vector as weights:

$$w_{ca}(\theta) = a(\theta)$$

Parameters

- **`steering_vec`** (*`ndarray`*) – A 2D-array of size (numTheta, num_ant) generated from gen_steering_vec
- **`sig_in`** (*`ndarray`*) – Either a 2D-array or 3D-array of size (num_ant, numChirps) or (numChirps, num_vrx, num_adc_samples) respectively, containing ADC sample data sliced as described
- **`axis`** (*`int`*) – Specifies the axis where the Vrx data in contained.

Returns A 3D-array of size (numChirps, numThetas, numSamples)

Return type doa_spectrum (ndarray)

Example

```
>>> # In this example, dataIn is the input data organized as numFrames by RDC
>>> frame = 0
>>> dataIn = np.random.rand((num_frames, num_chirps, num_vrx, num_adc_samples))
>>> aoa_bartlett(steering_vec,dataIn[frame],axis=1)
```

`mmwave.dsp.angle_estimation.`**`aoa_capon`**(*x*, *steering_vector*, *magnitude=False*)

Perform AOA estimation using Capon (MVDR) Beamforming on a rx by chirp slice

Calculate the aoa spectrum via capon beamforming method using one full frame as input. This should be performed for each range bin to achieve AOA estimation for a full frame This function will calculate both the angle spectrum and corresponding Capon weights using the equations prescribed below.

$$P_{ca}(\theta) = \frac{1}{a^H(\theta) R_{xx}^{-1} a(\theta)}$$

$$w_{ca}(\theta) = \frac{R_{xx}^{-1} a(\theta)}{a^H(\theta) R_{xx}^{-1} a(\theta)}$$

Parameters

- **`x`** (*`ndarray`*) – Output of the 1d range fft with shape (num_ant, numChirps)
- **`steering_vector`** (*`ndarray`*) – A 2D-array of size (numTheta, num_ant) generated from gen_steering_vec
- **`magnitude`** (*`bool`*) – Azimuth theta bins should return complex data (False) or magnitude data (True). Default=False

Raises `ValueError` – steering_vector and or x are not the correct shape

Returns A list containing numVec and steeringVectors den (ndarray: A 1D-Array of size (numTheta) containing azimuth angle estimations for the given range weights (ndarray): A 1D-Array of size (num_ant) containing the Capon weights for the given input data

Example

```
>>> # In this example, dataIn is the input data organized as numFrames by RDC
>>> Frame = 0
>>> dataIn = np.random.rand((num_frames, num_chirps, num_vrx, num_adc_samples))
>>> for i in range(256):
>>>     scan_aoa_capon[i,:], _ = dss.aoa_capon(dataIn[Frame,:,:,i].T, steering_
↪vector, magnitude=True)
```

mmwave.dsp.angle_estimation.**cov_matrix**(*x*)

Calculates the spatial covariance matrix (Rxx) for a given set of input data (x=inputData). Assumes rows denote Vrx axis.

Parameters **x** (*ndarray*) – A 2D-Array with shape (rx, adc_samples) slice of the output of the 1D range fft

Returns A 2D-Array with shape (rx, rx)

Return type Rxx (ndarray)

mmwave.dsp.angle_estimation.**forward_backward_avg**(*Rxx*)

Performs forward backward averaging on the given input square matrix

Parameters **Rxx** (*ndarray*) – A 2D-Array square matrix containing the covariance matrix for the given input data

Returns The 2D-Array square matrix containing the forward backward averaged covariance matrix

Return type R_fb (ndarray)

mmwave.dsp.angle_estimation.**peak_search**(*doa_spectrum*, *peak_threshold_weight=0.251188643150958*)

Wrapper function to perform scipy.signal's prescribed peak search algorithm Tested Runtime: 45 µs ± 2.61 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Parameters

- **doa_spectrum** (*ndarray*) – A 1D-Array of size (numTheta, 1) containing the theta spectrum at a given range bin
- **peak_threshold_weight** (*float*) – A float specifying the desired peak declaration threshold weight to be applied

Returns The number of max points found by the algorithm peaks (list): List of indexes where peaks are located total_power (float): Total power in the current spectrum slice

Return type num_max (int)

mmwave.dsp.angle_estimation.**peak_search_full**(*doa_spectrum*, *gamma=1.2*, *peak_threshold_weight=0.251188643150958*)

Perform TI prescribed peak search algorithm Tested Runtime: 147 µs ± 4.27 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)

Parameters

- **doa_spectrum** (*ndarray*) – A 1D-Array of size (numTheta, 1) containing the theta spectrum at a given range bin
- **gamma** (*float*) – A float specifying the maximum/minimum wiggle necessary to qualify as a peak

- **peak_threshold_weight** (*float*) – A float specifying the desired peak declaration threshold weight to be applied

Returns The number of max points found by the algorithm ang_est (list): List of indexes where the peaks are located

Return type num_max (int)

mmwave.dsp.angle_estimation.**peak_search_full_variance**(*doa_spectrum*, *steering_vec_size*, *sidelobe_level=0.251188643150958*, *gamma=1.2*)

Performs peak search (TI's full search) will retaining details about each peak including each peak's width, location, and value.

Parameters

- **doa_spectrum** (*ndarray*) – a 1D numpy array containing the power spectrum generated via some aoa method (naive,
- **or capon)** (*bartlett,*) –
- **steering_vec_size** (*int*) – Size of the steering vector in terms of number of theta bins
- **sidelobe_level** (*float*) – A low value threshold used to avoid sidelobe detections as peaks
- **gamma** (*float*) – Weight to determine when a peak will pass as a true peak

Returns A 1D numpy array of custom data types with length numberOfPeaksDetected. Each detected peak is organized as [peak_location, peak_value, peak_width] total_power (float): The total power of the spectrum. Used for variance calculations

Return type peak_data (ndarray)

mmwave.dsp.angle_estimation.**variance_estimation**(*num_max*, *est_resolution*, *peak_data*, *total_power*, *width_adjust_3d_b=2.5*, *input_snr=10000*)

This function will calculate an estimated variance value for each detected peak. This should be run after running peak_search_full_variance

Parameters

- **num_max** (*int*) – The number of detected peaks
- **est_resolution** (*float*) – The desired resolution in terms of theta
- **peak_data** (*ndarray*) – A numpy array of dictionaries, where each dictionary is of the form: {"peakLoc": , "peakVal": , "peakWid": }
- **total_power** (*float*) – The total power of the spectrum
- **width_adjust_3d_b** (*float*) – Constant to adjust the gamma bandwidth to 3dB level
- **input_snr** (*int*) – the linear snr for the input signal samples

Returns A 1D array of variances (of the peaks). The order of the peaks is preserved from peak_data

Return type est_var (ndarray)

mmwave.dsp.angle_estimation.**gen_steering_vec**(*ang_est_range*, *ang_est_resolution*, *num_ant*)

Generate a steering vector for AOA estimation given the theta range, theta resolution, and number of antennas

Defines a method for generating steering vector data input –Python optimized Matrix format The generated steering vector will span from -angEstRange to angEstRange with increments of ang_est_resolution The generated steering vector should be used for all further AOA estimations (bartlett/capon)

Parameters

- **ang_est_range** (*int*) – The desired span of thetas for the angle spectrum.
- **ang_est_resolution** (*float*) – The desired resolution in terms of theta
- **num_ant** (*int*) – The number of Vrx antenna signals captured in the RDC

Returns Number of vectors generated (integer divide angEstRange/ang_est_resolution) steering_vectors (ndarray): The generated 2D-array steering vector of size (num_vec,num_ant)

Return type num_vec (int)

Example

```
>>> #This will generate a numpy array containing the steering vector with
>>> #angular span from -90 to 90 in increments of 1 degree for a 4 Vrx platform
>>> _, steering_vec = gen_steering_vec(90,1,4)
```

mmwave.dsp.angle_estimation.**aoa_estimation_bf_one_point**(*num_ant*, *sig_in*, *steering_vec*)

Calculates the total power of the given spectrum

Parameters

- **num_ant** (*int*) – The number of virtual antennas (Vrx) being used
- **sig_in** (*ndarray*) – A 2D-array of size (num_ant, numChirps) containing ADC sample data sliced for used Vrx
- **steering_vec** (*ndarray*) – A 2D-array of size (numTheta, num_ant) generated from gen_steering_vec

Returns The total power of the given input spectrum

Return type out_value (complex)

mmwave.dsp.angle_estimation.**aoa_est_bf_single_peak_det**(*sig_in*, *steering_vec*)

Beamforming Estimate Angle of Arrival for single peak (single peak should be known a priori) Function call does not include variance calculations Function does not generate a spectrum. Rather, it only returns the array index (theta) to the highest peak

Parameters

- **sig_in** (*ndarray*) – A 2D-array of size (num_ant, numChirps) containing ADC sample data sliced as described
- **steering_vec** (*ndarray*) – A generated 2D-array steering vector of size (numVec,num_ant)

Returns Index of the theta spectrum at a given range bin that contains the max peak

Return type max_index (int)

mmwave.dsp.angle_estimation.**aoa_est_bf_single_peak**(*num_ant*, *noise*, *est_resolution*, *sig_in*, *steering_vec_size*, *steering_vec*)

Beamforming Estimate Angle of Arrival for single peak (single peak should be known a priori) Function call includes variance calculations Function does generate a spectrum.

Parameters

- **num_ant** (*int*) – The number of virtual receivers in the current radar setup
- **noise** (*float*) – Input noise figure
- **est_resolution** (*float*) – Desired theta spectrum resolution used when generating steering_vec
- **sig_in** (*ndarray*) – A 2D-array of size (num_ant, numChirps) containing ADC sample data sliced as described
- **steering_vec_size** (*int*) – Length of the steering vector array
- **steering_vec** (*ndarray*) – A generated 2D-array steering vector of size (numVec,num_ant)

Returns The estimated variance of the doa_spectrum max_index (int): Index of the theta spectrum at a given range bin that contains the max peak doa_spectrum (ndarray): A 1D-Array of size (numTheta, 1) containing the theta spectrum at a given range bin

Return type est_var (float)

mmwave.dsp.angle_estimation.**aoa_est_bf_multi_peak_det**(*gamma*, *sidelobe_level*, *sig_in*, *steering_vec*, *steering_vec_size*, *ang_est*, *search=False*)

Use Bartlett beamforming to estimate AOA for multi peak situation (a priori), no variance calculation

Parameters

- **gamma** (*float*) – Weight to determine when a peak will pass as a true peak
- **sidelobe_level** (*float*) – A low value threshold used to avoid sidelobe detections as peaks
- **sig_in** (*ndarray*) – A 2D-array of size (num_ant, numChirps) containing ADC sample data sliced as described
- **steering_vec** (*ndarray*) – A generated 2D-array steering vector of size (numVec,num_ant)
- **steering_vec_size** (*int*) – Length of the steering vector array
- **ang_est** (*ndarray*) – An empty 1D numpy array that gets populated with max indexes
- **search** (*bool*) – Flag that determines whether search is done to find max points

Returns The number of max points found across the theta bins at this particular range bin doa_spectrum (ndarray): A 1D-Array of size (numTheta, 1) containing the theta spectrum at a given range bin

Return type num_max (int)

mmwave.dsp.angle_estimation.**aoa_est_bf_multi_peak**(*gamma*, *sidelobe_level*, *width_adjust_3d_b*, *input_snr*, *est_resolution*, *sig_in*, *steering_vec*, *steering_vec_size*, *peak_data*, *ang_est*)

This function performs all sections of the angle of arrival process in one function.

1. Performs bartlett beamforming
2. Performs multi-peak search
3. Calculates an estimated variance

Parameters

- **gamma** (*float*) – Weight to determine when a peak will pass as a true peak
- **sidelobe_level** (*float*) – A low value threshold used to avoid sidelobe detections as peaks
- **width_adjust_3d_b** (*float*) – Constant to adjust gamma bandwidth to 3dB bandwidth
- **input_snr** (*float*) – Input data SNR value
- **est_resolution** (*float*) – User defined target resolution
- **sig_in** (*ndarray*) – A 2D-array of size (num_ant, numChirps) containing ADC sample data sliced as described
- **steering_vec** (*ndarray*) – A generated 2D-array steering vector of size (numVec,num_ant)
- **steering_vec_size** (*int*) – Length of the steering vector array
- **peak_data** (*ndarray*) – A 2D ndarray with custom data-type that contains information on each detected point
- **ang_est** (*ndarray*) – An empty 1D numpy array that gets populated with max indexes

Returns

Tuple [ndarray, ndarray]

1. num_max (int): The number of max values detected by search algorithm
2. est_var (ndarray): The estimated variance of this range of thetas at this range bin

`mmwave.dsp.angle_estimation.`**`naive_xyz`**(*virtual_ant*, *num_tx=3*, *num_rx=4*, *fft_size=64*)

Estimate the phase introduced from the elevation of the elevation antennas

Parameters

- **virtual_ant** – Signal received by the rx antennas, shape = [#angleBins, #detectedObjs], zero-pad #virtualAnts to #angleBins
- **num_tx** – Number of transmitter antennas used
- **num_rx** – Number of receiver antennas used
- **fft_size** – Size of the fft performed on the signals

Returns Estimated x axis coordinate in meters (m) y_vector (float): Estimated y axis coordinate in meters (m) z_vector (float): Estimated z axis coordinate in meters (m)

Return type x_vector (float)

`mmwave.dsp.angle_estimation.`**`beamforming_naive_mixed_xyz`**(*azimuth_input*, *input_ranges*, *range_resolution*, *method='Capon'*, *num_vrx=12*, *est_range=90*, *est_resolution=1*)

This function estimates the XYZ location of a series of input detections by performing beamforming on the azimuth axis and naive AOA on the vertical axis.

TI xWR1843 virtual antenna map Row 1 8 9 10 11 Row 2 0 1 2 3 4 5 6 7

phi (ndarray): theta (ndarray): ranges (ndarray): xyz_vec (ndarray):

Parameters

- **azimuth_input** (*ndarray*) – Must be a numpy array of shape (numDetections, numVrx)
- **input_ranges** (*ndarray*) – Numpy array containing the rangeBins that have detections (will determine x, y, z for
- **detection)** (*each*) –
- **range_resolution** (*float*) – The range_resolution in meters per rangeBin for rangeBin->meter conversion
- **method** (*string*) – Determines which beamforming method to use for azimuth aoa estimation.
- **num_vrx** (*int*) – Number of virtual antennas in the radar platform. Default set to 12 for 1843
- **est_range** (*int*) – The desired span of thetas for the angle spectrum. Used for gen_steering_vec
- **est_resolution** (*float*) – The desired angular resolution for gen_steering_vec

Raises

- ValueError – If method is not one of two AOA implementations ('Capon', 'Bartlett')
- ValueError – azimuthInput's second axis should have same shape as the number of Vrx

Returns

1. A numpy array of shape (numDetections,) where each element represents the elevation angle in degrees
2. A numpy array of shape (numDetections,) where each element represents the azimuth in degrees
3. A numpy array of shape (numDetections,) where each element represents the polar range in rangeBins
4. A numpy array of shape (3, numDetections) and format: [x, y, z] where x, y, z are 1D arrays. x, y, z should be in meters

Return type tuple [ndarray, ndarray, ndarray, ndarray, list]

1.2.2 mmwave.dsp.cfar

mmwave.dsp.cfar.**ca** (*x*, **argv*, ***kwargs*)

Detects peaks in signal using Cell-Averaging CFAR (CA-CFAR).

Parameters

- **x** (*ndarray*) – Signal.
- ***argv** – See mmwave.dsp.cfar.ca_
- ****kwargs** – See mmwave.dsp.cfar.ca_

Returns Boolean array of detected peaks in x.

Return type ndarray

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det = mm.dsp.ca(signal, l_bound=20, guard_len=1, noise_len=3)
>>> det
    array([False, False,  True, False, False, False, False,  True, False,
            True])
```

Perform a non-wrapping CFAR

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det =  mm.dsp.ca(signal, l_bound=20, guard_len=1, noise_len=3, mode='constant
↪')
>>> det
    array([False,  True,  True, False, False, False, False,  True,  True,
            True])
```

mmwave.dsp.cfar.**ca_** (*x*, *guard_len=4*, *noise_len=8*, *mode='wrap'*, *l_bound=4000*)

Uses Cell-Averaging CFAR (CA-CFAR) to calculate a threshold that can be used to calculate peaks in a signal.

Parameters

- **x** (*ndarray*) – Signal.
- **guard_len** (*int*) – Number of samples adjacent to the CUT that are ignored.
- **noise_len** (*int*) – Number of samples adjacent to the guard padding that are factored into the calculation.
- **mode** (*str*) – Specify how to deal with edge cells. Examples include 'wrap' and 'constant'.
- **l_bound** (*float or int*) – Additive lower bound while calculating peak threshold.

Returns

Tuple [ndarray, ndarray]

1. (ndarray): Upper bound of noise threshold.
2. (ndarray): Raw noise strength.

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.ca_(signal, l_bound=20, guard_len=1, noise_len=3)
>>> threshold
    (array([70, 76, 64, 79, 81, 91, 74, 71, 70, 79]), array([50, 56, 44, 59, 61,␣
↪71, 54, 51, 50, 59]))
```

Perform a non-wrapping CFAR thresholding

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.ca_(signal, l_bound=20, guard_len=1, noise_len=3, mode=
↪'constant')
>>> threshold
    (array([44, 37, 41, 65, 81, 91, 67, 51, 34, 46]), array([24, 17, 21, 45, 61,␣
↪71, 47, 31, 14, 26]))
```

`mmwave.dsp.cfar.caso(x, *argv, **kwargs)`

Detects peaks in signal using Cell-Averaging Smallest-Of CFAR (CASO-CFAR).

Parameters

- **x** (*ndarray*) – Signal.
- ***argv** – See mmwave.dsp.cfar.caso_
- ****kwargs** – See mmwave.dsp.cfar.caso_

Returns Boolean array of detected peaks in x.

Return type ndarray

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det = mm.dsp.caso(signal, l_bound=20, guard_len=1, noise_len=3)
>>> det
    array([False, False,  True, False, False, False, False,  True,  True,
            True])
```

Perform a non-wrapping CFAR

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det =  mm.dsp.caso(signal, l_bound=20, guard_len=1, noise_len=3, mode=
↪'constant')
>>> det
    array([False,  True,  True, False, False, False, False,  True,  True,
            True])
```

`mmwave.dsp.cfar.`**`caso_`**(*x*, *guard_len=4*, *noise_len=8*, *mode='wrap'*, *l_bound=4000*)

Uses Cell-Averaging Smallest-Of CFAR (CASO-CFAR) to calculate a threshold that can be used to calculate peaks in a signal.

Parameters

- **x** (*ndarray*) – Signal.
- **guard_len** (*int*) – Number of samples adjacent to the CUT that are ignored.
- **noise_len** (*int*) – Number of samples adjacent to the guard padding that are factored into the calculation.
- **mode** (*str*) – Specify how to deal with edge cells.
- **l_bound** (*float or int*) – Additive lower bound while calculating peak threshold.

Returns

Tuple [ndarray, ndarray]

1. (ndarray): Upper bound of noise threshold.
2. (ndarray): Raw noise strength.

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.caso_(signal, l_bound=20, guard_len=1, noise_len=3)
>>> (threshold[0].astype(int), threshold[1].astype(int))
    (array([69, 55, 49, 72, 72, 86, 69, 55, 49, 72]), array([49, 35, 29, 52, 52,␣
↪66, 49, 35, 29, 52]))
```

Perform a non-wrapping CFAR thresholding

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.caso_(signal, l_bound=20, guard_len=1, noise_len=3, mode=
↪'constant')
>>> (threshold[0].astype(int), threshold[1].astype(int))
    (array([69, 55, 49, 72, 72, 86, 69, 55, 49, 72]), array([49, 35, 29, 52, 52,␣
↪66, 49, 35, 29, 52]))
```

`mmwave.dsp.cfar.`**`cago`**(*x*, **argv*, ***kwargs*)

Detects peaks in signal using Cell-Averaging Greatest-Of CFAR (CAGO-CFAR).

Parameters

- **x** (*ndarray*) – Signal.
- ***argv** – See mmwave.dsp.cfar.cago_
- ****kwargs** – See mmwave.dsp.cfar.cago_

Returns Boolean array of detected peaks in x.

Return type ndarray

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det = mm.dsp.cago(signal, l_bound=20, guard_len=1, noise_len=3)
>>> det
    array([False, False,  True, False, False, False, False,  True, False,
            False])
```

Perform a non-wrapping CFAR

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det = mm.dsp.cago(signal, l_bound=20, guard_len=1, noise_len=3, mode='constant
↪')
>>> det
    array([False,  True,  True, False, False, False, False,  True,  True,
            True])
```

mmwave.dsp.cfar.**cago_**(*x*, *guard_len=4*, *noise_len=8*, *mode='wrap'*, *l_bound=4000*)

Uses Cell-Averaging Greatest-Of CFAR (CAGO-CFAR) to calculate a threshold that can be used to calculate peaks in a signal.

Parameters

- **x** (*ndarray*) – Signal.
- **guard_len** (*int*) – Number of samples adjacent to the CUT that are ignored.
- **noise_len** (*int*) – Number of samples adjacent to the guard padding that are factored into the calculation.
- **mode** (*str*) – Specify how to deal with edge cells.
- **l_bound** (*float or int*) – Additive lower bound while calculating peak threshold.

Returns

Tuple [ndarray, ndarray]

1. (ndarray): Upper bound of noise threshold.
2. (ndarray): Raw noise strength.

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.cago_(signal, l_bound=20, guard_len=1, noise_len=3)
>>> (threshold[0].astype(int), threshold[1].astype(int))
    (array([72, 97, 80, 87, 90, 97, 80, 87, 90, 86]), array([52, 77, 60, 67, 70,␣
↪77, 60, 67, 70, 66]))
```

Perform a non-wrapping CFAR thresholding

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.cago_(signal, l_bound=20, guard_len=1, noise_len=3, mode=
↪'constant')
>>> (threshold[0].astype(int), threshold[1].astype(int))
    (array([69, 55, 49, 72, 90, 97, 69, 55, 49, 72]), array([49, 35, 29, 52, 70,␣
↪77, 49, 35, 29, 52]))
```

mmwave.dsp.cfar.**os**(*x*, **argv*, ***kwargs*)

Performs Ordered-Statistic CFAR (OS-CFAR) detection on the input array.

Parameters

- **x** (*ndarray*) – Noisy array to perform cfar on with log values
- ***argv** – See mmwave.dsp.cfar.os_
- ****kwargs** – See mmwave.dsp.cfar.os_

Returns Boolean array of detected peaks in x.

Return type ndarray

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> det = mm.dsp.os(signal, k=3, scale=1.1, guard_len=0, noise_len=3)
>>> det
    array([False,  True,  True, False, False, False, False,  True, False,
            True])
```

mmwave.dsp.cfar.**os_**(*x*, *guard_len=0*, *noise_len=8*, *k=12*, *scale=1.0*)

Performs Ordered-Statistic CFAR (OS-CFAR) detection on the input array.

Parameters

- **x** (*ndarray*) – Noisy array to perform cfar on with log values
- **guard_len** (*int*) – Number of samples adjacent to the CUT that are ignored.
- **noise_len** (*int*) – Number of samples adjacent to the guard padding that are factored into the calculation.
- **k** (*int*) – Ordered statistic rank to sample from.
- **scale** (*float*) – Scaling factor.

Returns

Tuple [ndarray, ndarray]

1. (ndarray): Upper bound of noise threshold.
2. (ndarray): Raw noise strength.

Examples

```
>>> signal = np.random.randint(100, size=10)
>>> signal
    array([41, 76, 95, 28, 25, 53, 10, 93, 54, 85])
>>> threshold = mm.dsp.os_(signal, k=3, scale=1.1, guard_len=0, noise_len=3)
>>> (threshold[0].astype(int), threshold[1].astype(int))
    (array([93, 59, 58, 58, 83, 59, 59, 58, 83, 83]), array([85, 54, 53, 53, 76,␣
↪54, 54, 53, 76, 76]))
```

`mmwave.dsp.cfar.`**`peak_grouping`**(*obj_raw*, *det_matrix*, *num_doppler_bins*, *max_range_idx*, *min_range_idx*, *group_in_doppler_direction*, *group_in_range_direction*)

Performs peak grouping on detection Range/Doppler matrix.

The function groups neighboring peaks into one. The grouping is done according to two input flags: group_in_doppler_direction and group_in_doppler_direction. For each detected peak the function checks if the peak is greater than its neighbors. If this is true, the peak is copied to the output list of detected objects. The neighboring peaks that are used for checking are taken from the detection matrix and copied into 3x3 kernel regardless of whether they are CFAR detected or not. Note: Function always reads 9 samples per detected object from L3 memory into local array tempBuff, but it only needs to read according to input flags. For example if only the group_in_doppler_direction flag is set, it only needs to read middle row of the kernel, i.e. 3 samples per target from detection matrix.

Parameters

- **obj_raw** (*np.ndarray*) – (num_detected_objects, 3). detected objects from CFAR.
- **det_matrix** (*np.ndarray*) – Range-doppler profile. shape is numRangeBins x num_doppler_bins.
- **num_doppler_bins** (*int*) – number of doppler bins.
- **max_range_idx** (*int*) – max range of detected objects.
- **min_range_idx** (*int*) – min range of detected objects
- **group_in_doppler_direction** (*int*) – flag to perform grouping along doppler direction.
- **group_in_range_direction** (*int*) – flag to perform grouping along range direction.

Returns detected object after grouping.

Return type obj_out (np.ndarray)

`mmwave.dsp.cfar.`**`peak_grouping_qualified`**(*obj_raw*, *num_doppler_bins*, *max_range_idx*, *min_range_idx*, *group_in_doppler_direction*, *group_in_range_direction*)

Performs peak grouping on list of CFAR detected objects.

The function groups neighboring peaks into one. The grouping is done according to two input flags: group_in_doppler_direction and group_in_doppler_direction. For each detected peak the function checks if the peak is greater than its neighbors. If this is true, the peak is copied to the output list of detected objects. The neighboring peaks that are used for checking are taken from the list of CFAR detected objects, (not from the detection matrix), and copied into 3x3 kernel that has been initialized to zero for each peak under test. If the neighboring cell has not been detected by CFAR, its peak value is not copied into the kernel. Note: Function always search for 8 peaks in the list, but it only needs to search according to input flags.

Parameters

- **obj_raw** (*np.ndarray*) – (num_detected_objects, 3). detected objects from CFAR.

- **num_doppler_bins** (*int*) – number of doppler bins.
- **max_range_idx** (*int*) – max range of detected objects.
- **min_range_idx** (*int*) – min range of detected objects
- **group_in_doppler_direction** (*int*) – flag to perform grouping along doppler direction.
- **group_in_range_direction** (*int*) – flag to perform grouping along range direction.

Returns detected object after grouping.

Return type obj_out (np.ndarray)

1.2.3 mmwave.dsp.compensation

mmwave.dsp.compensation.**add_doppler_compensation**(*input_data*, *num_tx_antennas*, *doppler_indices=None*, *num_doppler_bins=None*)

Compensation of Doppler phase shift in the virtual antennas.

Compensation of Doppler phase shift on the virtual antennas (corresponding to second or third Tx antenna chirps). Symbols corresponding to virtual antennas, are rotated by half of the Doppler phase shift measured by Doppler FFT for 2 Tx system and 1/3 and 2/3 of the Doppler phase shift for 3 Tx system. The phase shift read from the table using half or 1/3 of the object Doppler index value. If the Doppler index is odd, an extra half of the bin phase shift is added.

The original function is called per detected objects. This functions is modified to directly compensate the azimuth_in matrix (numDetObj, num_angle_bins)

Parameters

- **input_data** (*ndarray*) – (range, num_antennas, doppler) Radar data cube that needs to be compensated. It can be the input of azimuth FFT after CFAR or the intermediate right before beamforming.
- **num_tx_antennas** (*int*) – Number of transmitters.
- **num_doppler_bins** (*int*) – (Optional) Number of doppler bins in the radar data cube. If given, that means the doppler indices are signed and needs to be converted to unsigned.
- **doppler_indices** (*ndarray*) – (Optional) Doppler index of the object with the shape of (num_detected_objects). If given, that means we only compensate on selected doppler bins.

Returns Original input data with the columns related to virtual receivers got compensated.

Return type input_data (ndarray)

Example

```
>>> # If the compensation is done right before naive azimuth FFT and objects is␣
↪detected already. you need to
>>> # feed in the doppler_indices
>>> dataIn = add_doppler_compensation(dataIn, 3, doppler_indices, 128)
```

mmwave.dsp.compensation.**rx_channel_phase_bias_compensation**(*rx_channel_compensations*, *input*, *num_antennas*)

Compensation of rx channel phase bias.

Parameters

- **rx_channel_compensations** – rx channel compensation coefficient.
- **input** – complex number.
- **num_antennas** – number of symbols.

mmwave.dsp.compensation.**near_field_correction**(*idx*, *detected_objects*, *start_range_index*, *end_range_index*, *azimuth_input*, *azimuth_output*, *num_angle_bins*, *num_rx_antennas*, *range_resolution*)

Correct phase error as the far-field plane wave assumption breaks.

Calculates near field correction for input detected index (corresponding to a range position). Referring to top level doxygen @ref nearFieldImplementation, this function performs the Set 1 rotation with the correction and adds to Set 0 in place to produce result in Set 0 of the azimuth_output.

This correction is done per detected objects from CFAR detection

Parameters

- **idx** – index of the detected objects in detected_objects.
- **detected_objects** – detected objects matrix with dimension of 100 x 6, where it holds at most 100 objects and 6 members are rangeIdx, dopplerIdx, peakVal, x, y and z. It is configured as a structured array.
- **start_range_index** – start range index of near field correction.
- **end_range_index** – end range index of near field correction.
- **azimuth_input** – complex array of which length is num_angle_bins+numVirtualAntAzim, where numVirtualAntAzim = 4, 8 or 12 depending on how many Txs are used.

Returns None. azimuth_output is changed in-place.

mmwave.dsp.compensation.**dc_range_signature_removal**(*fft_out1_d*, *positive_bin_idx*, *negative_bin_idx*, *calib_dc_range_sig_cfg*, *num_tx_antennas*, *num_chirps_per_frame*)

Compensation of DC range antenna signature.

Antenna coupling signature dominates the range bins close to the radar. These are the bins in the range FFT output located around DC. This feature is under user control in terms of enable/disable and start/end range bins through a CLI command called calibDcRangeSig. During measurement (when the CLI command is issued with feature enabled), each of the specified range bins for each of the virtual antennas are accumulated over the specified number of chirps and at the end of the period, the average is computed for each bin/antenna combination for removal after the measurement period is over. Note that the number of chirps to average must be power of 2. It is assumed that no objects are present in the vicinity of the radar during this measurement period. After measurement is done, the removal starts for all subsequent frames during which each of the bin/antenna average estimate is subtracted from the corresponding received samples in real-time for subsequent processing.

This function has a measurement phase while calib_dc_range_sig_cfg.counter is less than the preferred value and calibration phase afterwards. The original function is performed per chirp. Here it is modified to be called per frame.

Parameters

- **fft_out1_d** – (num_chirps_per_frame, num_rx_antennas, numRangeBins). Output of 1D FFT.
- **positive_bin_idx** – the first positive_bin_idx range bins (inclusive) to be compensated.
- **negative_bin_idx** – the last -negative_bin_idx range bins to be compensated.
- **calib_dc_range_sig_cfg** – a simple class for calibration configuration's storing purpose.
- **num_tx_antennas** – number of transmitters.
- **num_chirps_per_frame** – number of total chirps per frame.

Returns None. fft_out1_d is modified in-place.

mmwave.dsp.compensation.**clutter_removal**(*input_val*, *axis=0*)

Perform basic static clutter removal by removing the mean from the input_val on the specified doppler axis.

Parameters

- **input_val** (*ndarray*) – Array to perform static clutter removal on. Usually applied before performing doppler FFT. e.g. [num_chirps, num_vx_antennas, num_samples], it is applied along the first axis.
- **axis** (*int*) – Axis to calculate mean of pre-doppler.

Returns Array with static clutter removed.

Return type ndarray

1.2.4 mmwave.dsp.doppler_processing

mmwave.dsp.doppler_processing.**doppler_resolution**(*band_width*, *start_freq_const=77*, *ramp_end_time=62*, *idle_time_const=100*, *num_loops_per_frame=128*, *num_tx_antennas=3*)

Calculate the doppler resolution for the given radar configuration.

Parameters

- **start_freq_const** (*int*) – Frequency chirp starting point.
- **ramp_end_time** (*int*) – Frequency chirp end point.
- **idle_time_const** (*int*) – Idle time between chirps.
- **band_width** (*float*) – Radar config bandwidth.
- **num_loops_per_frame** (*int*) – The number of loops in each frame.
- **num_tx_antennas** (*int*) – The number of transmitting antennas (tx) on the radar.

Returns The doppler resolution for the given radar configuration.

Return type doppler_resolution (float)

mmwave.dsp.doppler_processing.**doppler_processing**(*radar_cube*, *num_tx_antennas=2*, *clutter_removal_enabled=False*, *interleaved=True*, *window_type_2d=None*, *accumulate=True*)

Perform 2D FFT on the radar_cube.

Interleave the radar_cube, perform optional windowing and 2D FFT on the radar_cube. Optional antenna couping signature removal can also be performed right before 2D FFT. In constrast to the original TI codes, CFAR and peak grouping are intentionally separated with 2D FFT for the easiness of debugging.

Parameters

- **radar_cube** (*ndarray*) – Output of the 1D FFT. If not interleaved beforehand, it has the shape of (numChirpsPerFrame, numRxAntennas, numRangeBins). Otherwise, it has the shape of (numRangeBins, numVirtualAntennas, num_doppler_bins). It is assumed that after interleaving the doppler dimension is located at the last axis.
- **num_tx_antennas** (*int*) – Number of transmitter antennas. This affects how interleaving is performed.
- **clutter_removal_enabled** (*boolean*) – Flag to enable naive clutter removal.
- **interleaved** (*boolean*) – If the input radar_cube is interleaved before passing in. The default radar_cube is not interleaved, i.e. has the shape of (numChirpsPerFrame, numRxAntennas, numRangeBins). The interleaving process will transform it such that it becomes (numRangeBins, numVirtualAntennas, num_doppler_bins). Note that this interleaving is only applicable to TDM radar, i.e. each tx emits the chirp sequentially.
- **window_type_2d** (*mmwave.dsp.utils.Window*) – Optional windowing type before doppler FFT.
- **accumulate** (*boolean*) – Flag to reduce the numVirtualAntennas dimension.

Returns

(numRangeBins, num_doppler_bins) complete range-dopper information. Original datatype is uint16_t. Note that azimuthStaticHeatMap can be extracted from zero-doppler index for visualization.

aoa_input (ndarray): (numRangeBins, numVirtualAntennas, num_doppler_bins) ADC data reorganized by vrx physical rx.

Return type detMatrix (ndarray)

`mmwave.dsp.doppler_processing.`**`doppler_estimation`**(*radar_cube*, *beam_weights*, *num_tx_antennas=2*, *clutter_removal_enabled=False*, *interleaved=False*, *window_type_2d=None*)

Perform doppler estimation on the weighted sum of range FFT output across all virtual antennas.

In contrast to directly computing doppler FFT from the output of range FFT, this function combines it across all the virtual receivers first using the weights generated from beamforming. Then FFT is performed and argmax is taken across each doppler axis to return the indices of max doppler values.

Parameters

- **radar_cube** (*ndarray*) – Output of the 1D FFT with only ranges on detected objects. If not interleaved beforehand, it has the shape of (numChirpsPerFrame, numRxAntennas, numDetObjs). Otherwise, it has the shape of (numDetObjs, numVirtualAntennas, num_doppler_bins). It is assumed that after interleaving the doppler dimension is located at the last axis.
- **beam_weights** (*ndarray*) – Weights to sum up the radar_cube across the virtual receivers. It is from the beam-forming and has the shape of (numVirtualAntennas, numDetObjs)
- **num_tx_antennas** (*int*) – Number of transmitter antennas. This affects how interleaving is performed.

- **clutter_removal_enabled** (*boolean*) – Flag to enable naive clutter removal.
- **interleaved** (*boolean*) – If the input radar_cube is interleaved before passing in. The default radar_cube is not interleaved, i.e. has the shape of (numChirpsPerFrame, numRxAntennas, numDetObjs). The interleaveing process will transform it such that it becomes (numDetObjs, numVirtualAntennas, num_doppler_bins). Note that this interleaving is only appliable to TDM radar, i.e. each tx emits the chirp sequentially.
- **window_type_2d** (*string*) – Optional windowing type before doppler FFT.

Returns

(numDetObjs) Doppler index for each detected objects. Positive index means moving away from radar while negative index means moving towards the radar.

Return type doppler_est (ndarray)

1.2.5 mmwave.dsp.noise_removal

mmwave.dsp.noise_removal.**peak_grouping_along_doppler**(*det_obj_2d*, *det_matrix*, *num_doppler_bins*)

Perform peak grouping along the doppler direction only. This is a temporary remedy for the slow and old implementation of peak_grouping_qualified() function residing in dsp.py currently. Will merge this back to there to enable more generic peak grouping.

mmwave.dsp.noise_removal.**range_based_pruning**(*det_obj_2d_raw*, *snr_thresh*, *peak_val_thresh*, *max_range*, *min_range*, *range_resolution*)

Filter out the objects out of the range and not sufficing SNR/peakVal requirement.

Filter out the objects based on the two following conditions: 1. Not within [min_range and max_range]. 2. Does not satisfy SNR/peakVal requirement, where it requires higher standard when closer and lower when further.

mmwave.dsp.noise_removal.**prune_to_peaks**(*det_obj2_d_raw*, *det_matrix*, *num_doppler_bins*, *reserve_neighbor=False*)

Reduce the CFAR detected output to local peaks.

Reduce the detected output to local peaks. If reserveNeighbor is toggled, will also return the larger neighbor. For example, given an array [2, 1, 5, 3, 2], default method will return [2, 5] while reserve neighbor will return [2, 5, 3]. The neighbor has to be a larger neighbor of the two immediate ones and also be part of the peak. the 1st element “1” in the example is not returned because it’s smaller than both sides so that it is not part of the peak.

Parameters

- **det_obj2_d_raw** (*np.ndarray*) – The detected objects structured array which contains the range_idx, doppler_idx, peakVal and SNR, etc.
- **det_matrix** (*np.ndarray*) – Output of doppler FFT with virtual antenna dimensions reduced. It has the shape of (num_range_bins, num_doppler_bins).
- **num_doppler_bins** (*int*) – Number of doppler bins.
- **reserve_neighbor** (*boolean*) – if toggled, will return both peaks and the larger neighbors.

Returns Pruned version of cfar_det_obj_index. cfar_det_obj_SNR_pruned (np.ndarray): Pruned version of cfar_det_obj_SNR.

Return type cfar_det_obj_index_pruned (np.ndarray)

1.2.6 mmwave.dsp.range_processing

mmwave.dsp.range_processing.**range_resolution**(*num_adc_samples*, *dig_out_sample_rate=2500*, *freq_slope_const=60.012*)

Calculate the range resolution for the given radar configuration

Parameters

- **num_adc_samples** (*int*) – The number of given ADC samples in a chirp
- **dig_out_sample_rate** (*int*) – The ADC sample rate
- **freq_slope_const** (*float*) – The slope of the freq increase in each chirp

Returns range_resolution (float): The range resolution for this bin band_width (float): The bandwidth of the radar chirp config

Return type tuple [float, float]

mmwave.dsp.range_processing.**range_processing**(*adc_data*, *window_type_1d=None*, *axis=-1*)

Perform 1D FFT on complex-format ADC data.

Perform optional windowing and 1D FFT on the ADC data.

Parameters

- **adc_data** (*ndarray*) – (num_chirps_per_frame, num_rx_antennas, num_adc_samples). Performed on each frame. adc_data is in complex by default. Complex is float32/float32 by default.
- **window_type_1d** (*mmwave.dsp.utils.Window*) – Optional window type on 1D FFT input. Default is None. Can be selected from Bartlett, Blackman, Hanning and Hamming.

Returns (num_chirps_per_frame, num_rx_antennas, num_range_bins). Also called fft_1d_out

Return type radar_cube (ndarray)

mmwave.dsp.range_processing.**zoom_range_processing**(*adc_data*, *low_freq*, *high_freq*, *fs*, *d*, *resample_number*)

Perform ZoomFFT on complex-format ADC data in a user-defined frequency range.

Parameters

- **adc_data** (*ndarray*) – (num_chirps_per_frame, num_rx_antennas, num_adc_samples). Performed on each frame. adc_data is in complex by default. Complex is float32/float32 by default.
- **low_freq** (*int*) – a user-defined number which specifies the lower bound on the range of frequency spectrum which the user would like to zoom on
- **high_freq** (*int*) – a user-defined number which specifies the higher bound on the range of frequency spectrum which the user would like to zoom on
- **fs** (*int*) – sampling rate of the original signal
- **d** (*int*) – Sample spacing (inverse of the sampling rate)
- **resample_number** (*int*) – The number of samples in the re-sampled signal.

Returns (num_chirps_per_frame, num_rx_antennas, resample_number).

Return type zoom_fft_spectrum (ndarray)

`mmwave.dsp.range_processing.`**`zoom_fft_visualize`**(*zoom_fft_spectrum*, *antenna_idx*, *range_bin_idx*)

to be implemented

1.2.7 mmwave.dsp.utils

`mmwave.dsp.utils.`**`windowing`**(*input*, *window_type*, *axis=0*)

Window the input based on given window type.

Parameters

- **`input`** – input numpy array to be windowed.
- **`window_type`** – enum chosen between Bartlett, Blackman, Hamming, Hanning and Kaiser.
- **`axis`** – the axis along which the windowing will be applied.

Returns:

`mmwave.dsp.utils.`**`XYestimation`**(*azimuthMagSqr*, *numAngleBins*, *detObj2D*)

Given the phase information from 3D FFT, calculate the XY position of the objects and populate the detObj2D array.

Parameters

- **`azimuthMagSqr`** – (numDetObj, numAngleBins) Magnitude square of the 3D FFT output.
- **`numAngelBins`** – hardcoded as 64 in our project.
- **`detObj2D`** – Output yet to be populated with the calculated X, Y and Z information

1.2.8 mmwave.dsp.ZoomFFT

`class` `mmwave.dsp.ZoomFFT.`**`ZoomFFT`**(*low_freq*, *high_freq*, *fs*, *signal=None*)

This class is an implementation of the Zoom Fast Fourier Transform (ZoomFFT).

The zoom FFT (Fast Fourier Transform) is a signal processing technique used to analyse a portion of a spectrum at high resolution. The steps to apply the zoom FFT to this region are as follows:

1. Frequency translate to shift the frequency range of interest down to near 0 Hz (DC)
2. Low pass filter to prevent aliasing when subsequently sampled at a lower sample rate
3. Re-sample at a lower rate
4. FFT the re-sampled data (Multiple blocks of data are needed to have an FFT of the same length)

The resulting spectrum will now have a much smaller resolution bandwidth, compared to an FFT of non-translated data.

`compute_fft`()

Computes the Fast Fourier Transform (FFT) of the signal.

Returns A frequency-shifted, unscaled, FFT of the signal.

Return type X (np.ndarray)

`compute_zoomfft`(*resample_number=None*)

Computes the Zoom Fast Fourier Transform (ZoomFFT) of the signal.

Parameters **`resample_number`** (*`int`*) – The number of samples in the resampled signal.

Returns A frequency-shifted, unscaled, ZoomFFT of the signal. bw_factor (int): Bandwidth factor fftlen (int): Length of the ZoomFFT output Ld (int): for internal use F (int): for internal use

Return type Xd (np.ndarray)

plot_fft (*d=None*)
Plots the Fast Fourier Transform (FFT) of the signal.

Parameters **d** (*int*) – Sample spacing (inverse of the sampling rate)

plot_zoomfft (*resample_number=None*)
Plots the Zoom Fast Fourier Transform (ZoomFFT) of the signal.

Parameters **resample_number** (*int*) – The number of samples in the resampled signal.

set_signal (*signal*)
Sets given signal as a member variable of the class.

e.g. ZoomFFT.create_signal(generate_sinewave(a, b, c) + generate_sinewave(d, e, f))

Parameters **signal** (*np.ndarray*) – Signal to perform the ZoomFFT on

sinewave (*f*, *length*, *amplitude=1*)
Generates a sine wave which could be used as a part of the signal.

Parameters

- **f** (*int*) – Frequency of the sine wave
- **length** (*int*) – Number of data points in the sine wave
- **amplitude** (*int*) – Amplitude of the sine wave

Returns Generated sine wave with the given parameters.

Return type x (np.ndarray)

1.3 mmwave.tracking

1.3.1 mmwave.tracking.ekf

class mmwave.tracking.ekf.**EKF**
Extended Kalman Filter with built in tracking

point_cloud
Array of point objects

Type ndarray

target_desc
Array of detected objects

Type ndarray

t_num
Number of detected objects

Type int

h_track_module
Tracking meta-data

Type object

num_points
Number of detected points

Type int

step()
Step the EKF

Returns

1. list: Information about detected objects
2. int: Number of detected objects

Return type Tuple [list, int]

update_point_cloud(*ranges*, *azimuths*, *dopplers*, *snrs*)
Update step of the EKF

Parameters

- **ranges** – Ordered ranges of the detected points
- **azimuths** – Ordered azimuths of the detected points
- **dopplers** – Ordered dopplers of the detected points
- **snrs** – Ordered snrs of the detected points

Returns None

1.4 mmwave.clustering

1.4.1 mmwave.clustering.clustering

mmwave.clustering.clustering.**associate_clustering**(*new_cluster*, *pre_cluster*, *max_num_clusters*, *epsilon*, *v_factor*, *use_elevation=False*)
Associate pre-existing clusters and the new clusters.

The function performs an association between the pre-existing clusters and the new clusters, with the intent that the cluster sizes are filtered.

Parameters

- **new_cluster** –
- **pre_cluster** –
- **max_num_clusters** –
- **epsilon** –
- **v_factor** –
- **use_elevation** –

mmwave.clustering.clustering.**radar_dbscan**(*det_obj_2d*, *weight*, *doppler_resolution*, *use_elevation=False*)
DBSCAN for point cloud. Directly call the scikit-learn.dbscan with customized distance metric.

DBSCAN algorithm for clustering generated point cloud. It directly calls the dbscan from scikit-learn but with customized distance metric to combine the coordinates and weighted velocity information.

Parameters

- **det_obj_2d** (*ndarray*) – Numpy array containing the rangeIdx, dopplerIdx, peakVal, xyz coordinates of each detected points. Can have extra SNR entry, not necessary and not used.
- **weight** (*float*) – Weight for velocity information in combined distance metric.
- **doppler_resolution** (*float*) – Granularity of the doppler measurements of the radar.
- **use_elevation** (*bool*) – Toggle to use elevation information for DBSCAN and output clusters.

Returns

Numpy array containing the clusters' information including number of points, center and size of the clusters in x,y,z coordinates and average velocity. It is formulated as the structured array for numpy.

Return type clusters (np.ndarray)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

A

adc_ip (*mmwave.dataloader.adc.DCA1000 attribute*), 3
add_doppler_compensation() (*in module mmwave.dsp.compensation*), 20
aoa_bartlett() (*in module mmwave.dsp.angle_estimation*), 6
aoa_capon() (*in module mmwave.dsp.angle_estimation*), 7
aoa_est_bf_multi_peak() (*in module mmwave.dsp.angle_estimation*), 11
aoa_est_bf_multi_peak_det() (*in module mmwave.dsp.angle_estimation*), 11
aoa_est_bf_single_peak() (*in module mmwave.dsp.angle_estimation*), 10
aoa_est_bf_single_peak_det() (*in module mmwave.dsp.angle_estimation*), 10
aoa_estimation_bf_one_point() (*in module mmwave.dsp.angle_estimation*), 10
associate_clustering() (*in module mmwave.clustering.clustering*), 28
azimuth_processing() (*in module mmwave.dsp.angle_estimation*), 6

B

beamforming_naive_mixed_xyz() (*in module mmwave.dsp.angle_estimation*), 12

C

ca() (*in module mmwave.dsp.cfar*), 14
ca_() (*in module mmwave.dsp.cfar*), 14
cago() (*in module mmwave.dsp.cfar*), 16
cago_() (*in module mmwave.dsp.cfar*), 17
caso() (*in module mmwave.dsp.cfar*), 15
caso_() (*in module mmwave.dsp.cfar*), 15
cli_port (*mmwave.dataloader.radars.TI attribute*), 5
close() (*mmwave.dataloader.adc.DCA1000 method*), 4
close() (*mmwave.dataloader.radars.TI method*), 5
clutter_removal() (*in module mmwave.dsp.compensation*), 22
compute_fft() (*mmwave.dsp.ZoomFFT.ZoomFFT method*), 26
compute_zoomfft() (*mmwave.dsp.ZoomFFT.ZoomFFT method*), 26
config_port (*mmwave.dataloader.adc.DCA1000 attribute*), 3
configure() (*mmwave.dataloader.adc.DCA1000 method*), 4
connected (*mmwave.dataloader.radars.TI attribute*), 5
cov_matrix() (*in module mmwave.dsp.angle_estimation*), 8

D

data_port (*mmwave.dataloader.adc.DCA1000 attribute*), 3
data_port (*mmwave.dataloader.radars.TI attribute*), 5
dc_range_signature_removal() (*in module mmwave.dsp.compensation*), 21
DCA1000 (*class in mmwave.dataloader.adc*), 3
doppler_estimation() (*in module mmwave.dsp.doppler_processing*), 23
doppler_processing() (*in module mmwave.dsp.doppler_processing*), 22
doppler_resolution() (*in module mmwave.dsp.doppler_processing*), 22

E

EKF (*class in mmwave.tracking.ekf*), 27

F

forward_backward_avg() (*in module mmwave.dsp.angle_estimation*), 8

G

gen_steering_vec() (*in module mmwave.dsp.angle_estimation*), 9

H

h_track_module (*mmwave.tracking.ekf.EKF attribute*), 27

M

mode (*mmwave.dataloader.radars.TI attribute*), 5

N

naive_xyz() (*in module mmwave.dsp.angle_estimation*), 12
near_field_correction() (*in module mmwave.dsp.compensation*), 21
num_points (*mmwave.tracking.ekf.EKF attribute*), 28
num_rx_ant (*mmwave.dataloader.radars.TI attribute*), 5
num_tx_ant (*mmwave.dataloader.radars.TI attribute*), 5
num_virtual_ant (*mmwave.dataloader.radars.TI attribute*), 5

O

organize() (*mmwave.dataloader.adc.DCA1000 static method*), 4
os() (*in module mmwave.dsp.cfar*), 18
os_() (*in module mmwave.dsp.cfar*), 18

P

parse_raw_adc() (*in module mmwave.dataloader.file_parse*), 6
parse_tsw1400() (*in module mmwave.dataloader.utils*), 4
peak_grouping() (*in module mmwave.dsp.cfar*), 19
peak_grouping_along_doppler() (*in module mmwave.dsp.noise_removal*), 24
peak_grouping_qualified() (*in module mmwave.dsp.cfar*), 19
peak_search() (*in module mmwave.dsp.angle_estimation*), 8
peak_search_full() (*in module mmwave.dsp.angle_estimation*), 8
peak_search_full_variance() (*in module mmwave.dsp.angle_estimation*), 9
plot_fft() (*mmwave.dsp.ZoomFFT.ZoomFFT method*), 27
plot_zoomfft() (*mmwave.dsp.ZoomFFT.ZoomFFT method*), 27
point_cloud (*mmwave.tracking.ekf.EKF attribute*), 27
prune_to_peaks() (*in module mmwave.dsp.noise_removal*), 24

R

radar_dbscan() (*in module mmwave.clustering.clustering*), 28
range_based_pruning() (*in module mmwave.dsp.noise_removal*), 24
range_processing() (*in module mmwave.dsp.range_processing*), 25
range_resolution() (*in module mmwave.dsp.range_processing*), 25
read() (*mmwave.dataloader.adc.DCA1000 method*), 4
rx_channel_phase_bias_compensation() (*in module mmwave.dsp.compensation*), 20

S

sample() (*mmwave.dataloader.radars.TI method*), 5
sdk_version (*mmwave.dataloader.radars.TI attribute*), 5
set_signal() (*mmwave.dsp.ZoomFFT.ZoomFFT method*), 27
sinewave() (*mmwave.dsp.ZoomFFT.ZoomFFT method*), 27
static_ip (*mmwave.dataloader.adc.DCA1000 attribute*), 3
step() (*mmwave.tracking.ekf.EKF method*), 28

T

t_num (*mmwave.tracking.ekf.EKF attribute*), 27
target_desc (*mmwave.tracking.ekf.EKF attribute*), 27
TI (*class in mmwave.dataloader.radars*), 5

U

update_point_cloud() (*mmwave.tracking.ekf.EKF method*), 28

V

variance_estimation() (*in module mmwave.dsp.angle_estimation*), 9
verbose (*mmwave.dataloader.radars.TI attribute*), 5

W

windowing() (*in module mmwave.dsp.utils*), 26

X

XYestimation() (*in module mmwave.dsp.utils*), 26

Z

zoom_fft_visualize() (*in module mmwave.dsp.range_processing*), 25
zoom_range_processing() (*in module mmwave.dsp.range_processing*), 25
ZoomFFT (*class in mmwave.dsp.ZoomFFT*), 26