FMCW Radar System

Mostafa Alizadeh

Jul 09, 2019

PRELIMINARY ON USING MMWAVE RADARS

The radar application is in the suppport of *sensor* community to expedite developement stages of sensing applications. The application is developed based on TI radars for **real-time** processing and demonstration.

CHAPTER

ONE

BRIEF INTRODUCTION TO TI RADAR

Here you should explain the main features of the TI radar . . .

CHAPTER

TWO

FMCW RADAR INTRODUCTION

TI mmwave radars are *frequency modulated continuous wave* (FMCW) radars such that the frequency is swept linearly. FMCW radars have unique advantages, which cannot be presented in other radars at once. Those are:

- Being a mm-wave radar: the high attenuation in mmwave frequencies provides a high isolation between the co-located operating radars even if they are separated in a few meters. Indeed, tiny displacements in mm are comparable to the wavelength thus they can be detected. This high sensitivity is required to detect the chest wall movement, which is in mm order.
- Discriminating range or localizing: because the radar can distinguish the reflections from different ranges, potentially it can be used for multi-subject vital signs detection. This feature is recognized as the main advantage of an FMCW radar in[1]. Indeed, high propagation attenuation reduces the possibility of having an echo signal, which is bounced off multiple reflectors. Most probably, the echo signal is reflected off a single object if the environment is not rich scattering. In that area, the received signal at particular range experienced a line of sight wireless channel. In contrast, CW radars suffer from multipath fading because they collect all reflections from all objects at all visible ranges in a one sinusoid signal.
- Being robust against thermal noise: FM signals are more robust against noise in comparison to AM signals. Also, in FMCW radars the vital sign information is encoded in the received phase similar to FM signals. Thus, FMCW radar is less affected by the noise in comparison to impulse radars.

2.1 FMCW radar basics

In any radar, the electromagnetic wave is sent into the environment containing various objects. Then the echo of the wave is captured at a receiver. A simplified block diagram of such a system is shown in *Fig. 1* in which both the transmitter and the receiver are at the same location. Each chirp at the output of the FMCW generator is a sinusoid signal whose frequency is swept from fmin to fmax (*Fig. 2*). Here the frequency is swept linearly with a positive slope of K and a duration of T_r implying that the sweeping bandwidth is $f_{max}f_{min} = KT_r$. The received signal at the output port of the receiver antenna is amplified and correlated with the transmit signal, which results in a signal called beat signal. The beat signal contains information about the objects in the scene. Particularly, the delay in the reflected signal is translated to an instantaneous frequency difference between the transmitted and the received chirps.

[1] S. Wang, A. Pohl, T. Jaeschke, M. Czaplik, M. Köny, S. Leonhardt, and N. Pohl, "A novel ultra-wideband 80 ghz fmcw radar system for contactless monitoring of vital signs," in 2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC), Aug 2015, pp. 4978–4981.

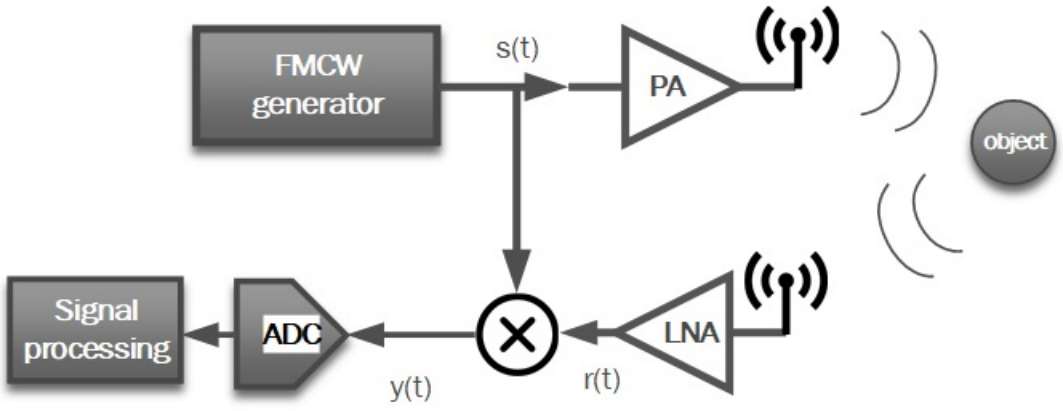

Fig. 1

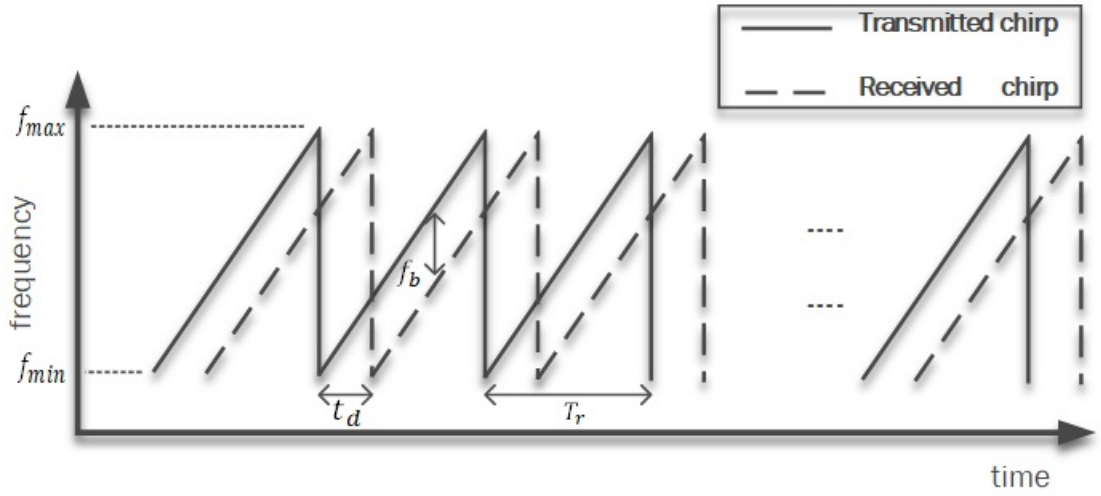

Fig. 2

Assume that the complex chirp signal is:

$$s(t) = A_t \exp\left(j(2\pi f_{min} t + \pi K t^2)\right), 0 < t < T_r \qquad (1)$$

f_{min} is the start frequency (and $_{max}$ is the corresponding wavelength) and At is the magnitude related to the transmit power. Suppose that there is only a single small object situated at the distance of R0 to the radar but it is moving around R0, which results in a time-varying distance to the radar. Let us denote this time-varying distance by $R(t) = R0 + x(t)$ and x(t) is a function represents the distance variations around R0. Furthermore, the reflected wave off the object at the receiver is the delayed version of s(t) with a delay of $t_d = 2R(t)/c$, which is the round-trip time of the wave. c is the light speed throughout the whole paper. Consequently, the IF signal for only a single chirp duration will be:

$$\begin{aligned} y(t) &= s(t)s^*(t - t_d) \\ &= A_t A_r \exp\left(j(\phi(t) - \phi(t - t_d))\right), \quad t_d < t < T_r, \end{aligned} \qquad (2)$$

The thermal noise and other channel considerations are ignored for simplifications, but A_r has a relationship to A_t by

the radar equation[2]. The beat signal, $y(t)$, can be expressed as follows:

$$\begin{aligned} y(t) &= A_t A_r \exp\left(j(2\pi f_{min} t_d + 2\pi K t_d t - \pi K t_d^2)\right) \\ &\approx A_t A_r \exp\left(j(2\pi f_{min} t_d + 2\pi K t_d t\right) \\ &= A_t A_r \exp\left(j(\psi(t) + \omega_b t)\right), \quad t_d < t < T_r \end{aligned}$$

$$y(t) \approx A_t A_r \exp\left(j(\psi(t) + \omega_b t)\right), \quad t_d < t < T_r \tag{3}$$

$$\psi(t) = 4\pi \frac{R_0 + x(t)}{\lambda_{max}}, \qquad \omega_b = 4\pi \frac{K R_0}{c}, \tag{4}$$

the second approximate equality in *eq3* is obtained by ignoring the third term in the phase, which is very small. The third term is negligible because K is in $10^{12} Hz/s$ order while t_d is in 1ns thus the term is in the order of 10^{-6}. Equation (*eq4*) obtained after replacing t_d to (*eq3*) and ignoring the $x(t)t$ term because t is in $1\mu s$ and x(t) is almost constant for one chirp as we will see later. Furthermore, $\psi(t)$ varies with x(t) relative to λ_{max}. So, the phase variations in the scale of the maximum wavelength can greatly change the beat signal phase. For example, a radar operating at 6 GHz is 10 times less sensitive in comparison to a 60 GHz radar. In addition, x(t) is almost constant within one chirp because subjects are not moving more than 1 mm per chirp equivalent to $1mm/1\mu s = 10^3 m/s$.

2.1.1 Radar equation

If the P_t is nominal transmit power and the target is at a distance of R, then the received power is related to the transmit power of a radar by the following:

$$P_r = \frac{P_t G_t G_r \sigma \lambda^2}{(4\pi)^3 R^4}, \tag{5}$$

where G_t, and G_r are the transmit and the receive antenna gains, respectively. σ is the RCS of the target. Also, λ is the wavelength of the travelling wave. For physiological motion, the area under which the chest moves determines the RCS. Equation *eq5* is known as the **radar equation**. We assume that the room temperature is 300 Kelvin (or about 25 Celsius) and the transmit chirp has a sweeping bandwidth of 4 GHz, then the noise power at the output terminal of the receiver antenna is $P_n = 10\, log_{10}(KTb_s) \approx -103\, dB$ in which K is *Boltzmann's constant**[0]. In addition, if the receiver NF is NF dB and the minimum SNR required at the based band is denoted by SNR_{min}, then the minimum received signal power at the output port of Rx antenna will be $P_{r,min} = NF + P_n + SNR_{min}$ in which NF is the noise figure and all the variables are in dB. By using *eq5*, which relates P_r to the target range, the maximum range versus minimum required SNR at the based band can be obtained:

$$R_{max} = \sqrt[4]{\frac{P_t G_t G_r \sigma \lambda^2}{P_{r,min}(4\pi)^3}}, \quad P_{r,min} = NF + P_n + SNR_{min}, \tag{6}$$

[2] C. A. Balanis, Modern Antenna Handbook. John Wiley & Sons, Incorporated, google-Books-ID: Q9OgkQEACAAJ.

[0] $K = 4.138 \times 10^{-23}\ J/K$. Do not confuse this K with the chirp slope.

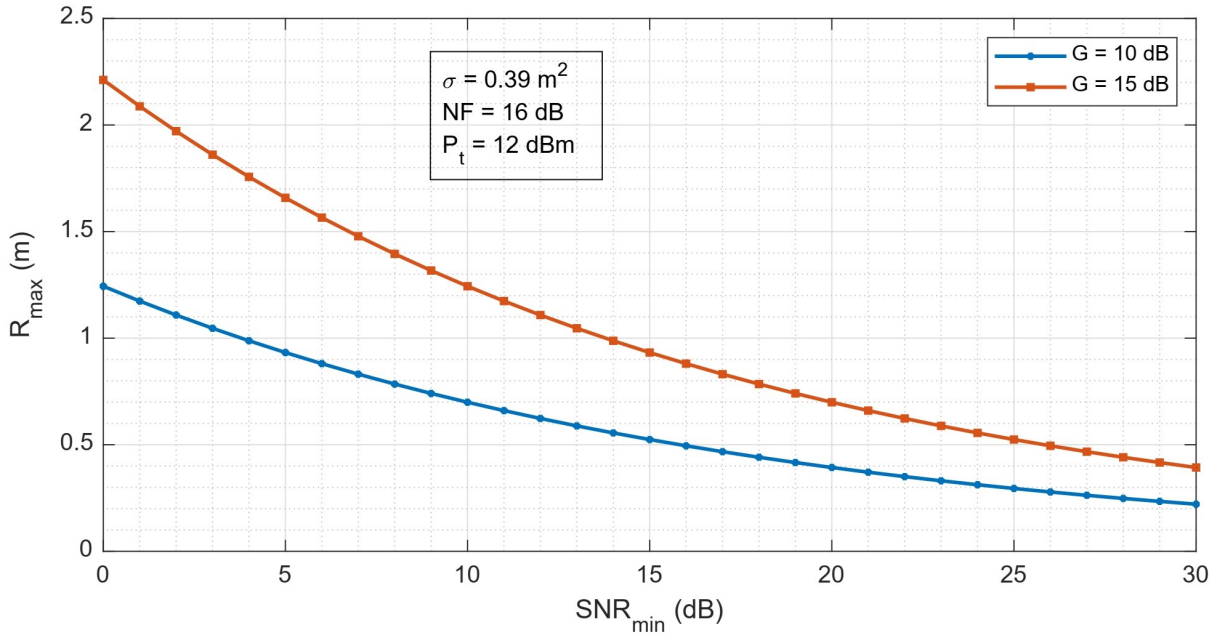

Fig. 3: Maximum range of a target versus minimum required SNR at the based band. The gain of Tx/Rx antennas are the same as G.

In figure *Fig. 3*, R_{max} is plotted versus minimum SNR with values annotated. In fact, $\sigma = 0.39m^2$ is an approximate value for a human as mentioned in[3]. The transmit power of 12 dBm is the output power of AWR1642/AWR1443 chips (see table ref{TIParams}).

2.2 Range detection

From *eq3* and *eq4*, $\psi(t)$ can be approximated by sampling *x(t)*:

$$\psi(t) = 4\pi\frac{R_0 + x(t_0)}{\lambda_{max}}, \qquad \omega_b = 4\pi\frac{KR_0}{c}, \qquad (7)$$

where t_0 is any time in $[t_d, T_r]$. This equation is used to detect the range of a subject, R_0. To this end, an FFT is applied over samples of a chirp to obtain the spectrum of the beat signal, which has peaks corresponding to the subjects at different ranges. This FFT reveals range information so it is called **range FFT**. Each range FFT bin represents a particular distance with an associated phase similar to $\psi(t)$. Furthermore, as we mentioned before, there can be a very small shift in ω_b due to residual delays incurred by the PA and the LNA. Although the little frequency shift exists, it diminishes after the radar warms up.

2.3 Doppler or speed detection

From *eq3*, *eq4*, x(t) can be any function of time depending on the moving trajectory of the target with respect to the radar. Assuming a radial movement†[0], so $x(t) = vt$ where *v* is the radial velocity. By substituting it to *eq4* then to *eq3*, we have:

[3] Amy Diane Droitcour. Non-Contact Measurement of Heart and Respiration Rates with a Single-Chip Microwave Doppler Radar. 2006.

[0] *Radial movement* is a movement along the radial axis of a spherical coordinates with the radar at the centre.

$$y(t) \approx A_t A_r \exp\left(j\left(4\pi\frac{R_0}{\lambda_{max}} + 4\pi K\frac{R_0}{c}t + 4\pi K\frac{v}{c}t^2 + 4\pi\frac{v}{\lambda_{max}}t\right)\right) \qquad (8)$$

the first term in the exponent is constant, the second term is for the range, and the third term is very small due to the order of t. The last term in the exponent is desired since it is linear in time and is the function of *v*. Though, if we take the **range FFT** and look to specific range, we observe a signal like *eq8* with only the last term in the exponent. Thus, by taking the second FFT on a range bin across a sequence of chirps, we obtain the spectrum of the range containing peaks corresponding to the target velocities‡[0].

2.4 Angle of arrival detection

In a typical FMCW radar, the received signal from l'th receiver antenna and the k'th target can be expressed as:

$$\boldsymbol{x}_{kl}(t_f, t_s) = b_{kl}\exp\left(-j\left(2\pi f_b t_f + \tau_{kl} + \xi_{kl} + \Delta\psi_{kl}(t_f, t_s)\right)\right) \qquad (9)$$

where b_{kl} relates to the received power from the l'th virtual receiver and the k'th target, which depends on the target's RCS and range, and the antenna gain in the direction from the wave is scattered and received at the antenna. f_b is beat frequency corresponding to the range of the target. τ_{kl} is phase shift due to angle of arrivals and is a function of θ_k and ϕ_k corresponding to the azimuth and elevation angles (see *Fig. 4*). ξ_{kl} is the constant phase representing the phase difference between the virtual receivers for k'th target due to manufacturing differences of the receiver channels and the angle of arrivals (or antenna pattern in the direction of the received wave) as well. Note that ξ_{kl} is different than τ_{kl} as it is not a particular function of the arrival angles, but we can say something about τ_{kl} in terms of θ_k and ϕ_k. Also, $\psi_{kl}(t_f, t_s)$ is the residual phase noise[4]. Also, we did not included the thermal additive noise in *eq9*. The previous equation can be written in a matrix-vector form containing all the signal received by all virtual receivers if we ignore the residual phase noise. So,

$$x_k(t_f, t_s) = \Gamma_k a_k(\theta_k, \phi_k) s_k(t_f) + e_k(t_f, t_s) \qquad (10)$$

in which $a_k(\theta_k, \phi_k)$ is known as steering vector and Γ_k is a diagonal matrix containing the complex coefficients of $b_{kl}exp(-j\xi_{kl})$. $s_k(t_f)$ has only range information if the targets are stationary. In fact, if they are moving, $s_k(t_f)$ will be depending on the target's range and the velocity . If we assume that Γ_k is the same for all k, then the received vector from all virtual antennas and all targets are:

$$\boldsymbol{x}(t_f, t_s) = \sum_{k=1}^{K}(\Gamma\boldsymbol{a}_k(\theta_k, \phi_k)s_k(t_f) + e_k(t_f, t_s)) = \Gamma A\boldsymbol{s}(t_f) + e(t_f, t_s) \qquad (11)$$

where matrix A is:

$$A = [\boldsymbol{a}(\theta_1, \phi_1), \boldsymbol{a}(\theta_2, \phi_2), \cdots, \boldsymbol{a}(\theta_K, \phi_K)] \qquad (12)$$

and $s(t_f)$ is:

[0] The spectrum of the range bin should be scaled by $\lambda_{max}/2$ to convert the frequency to the actual *m/s* (see *eq8*).

[4] M. C. Budge and M. P. Burt, "Range correlation effects in radars," in The Record of the 1993 IEEE National Radar Conference, 1993, pp. 212–216.

$$\boldsymbol{x}(t_f) = [s_1(t_f), s_K(t_f), \cdots, s_K(t_f)]^T \qquad (13)$$

and K is the number of targets and the noise vector has elements which are the accumulation the noises from each antennas. If there is coupling between the antenna elements, then a matrix C should be multiplied to the left of Γ to account the coupling.

As previously mentioned, the steering vector has a specific relationship to the angle of arrivals. *Fig. 4* helps to derive the relationship. The antennas are shown by the solid black dots. They are separated by dx and dy along x-axis and z-axis, respectively. The red arrows are along the direction at which the wave is received by the radar. It makes an angle of ϕ with xy plane and θ with y-axis in xy plane. The relative phase difference between the signal received from l'th receiver to the first receiver on the origin can be expressed as below if the antenna is on the x-axis:

$$-\frac{2\pi}{\lambda_{max}} l dx \sin(\theta)\cos(\phi) \qquad (14)$$

and this is the following if the antenna is shifted up in z direction:

$$-\frac{2\pi}{\lambda_{max}} l_x dx \sin(\theta)\cos(\phi) + \frac{2\pi}{\lambda_{max}} l_z dy \sin(\phi) \qquad (15)$$

where l_x, l_z are the shifts to the negative x and positive z, respectively. By knowing the antenna array configuration, we can obtain the steering vector $a_k(\theta_k, \phi_k)$.

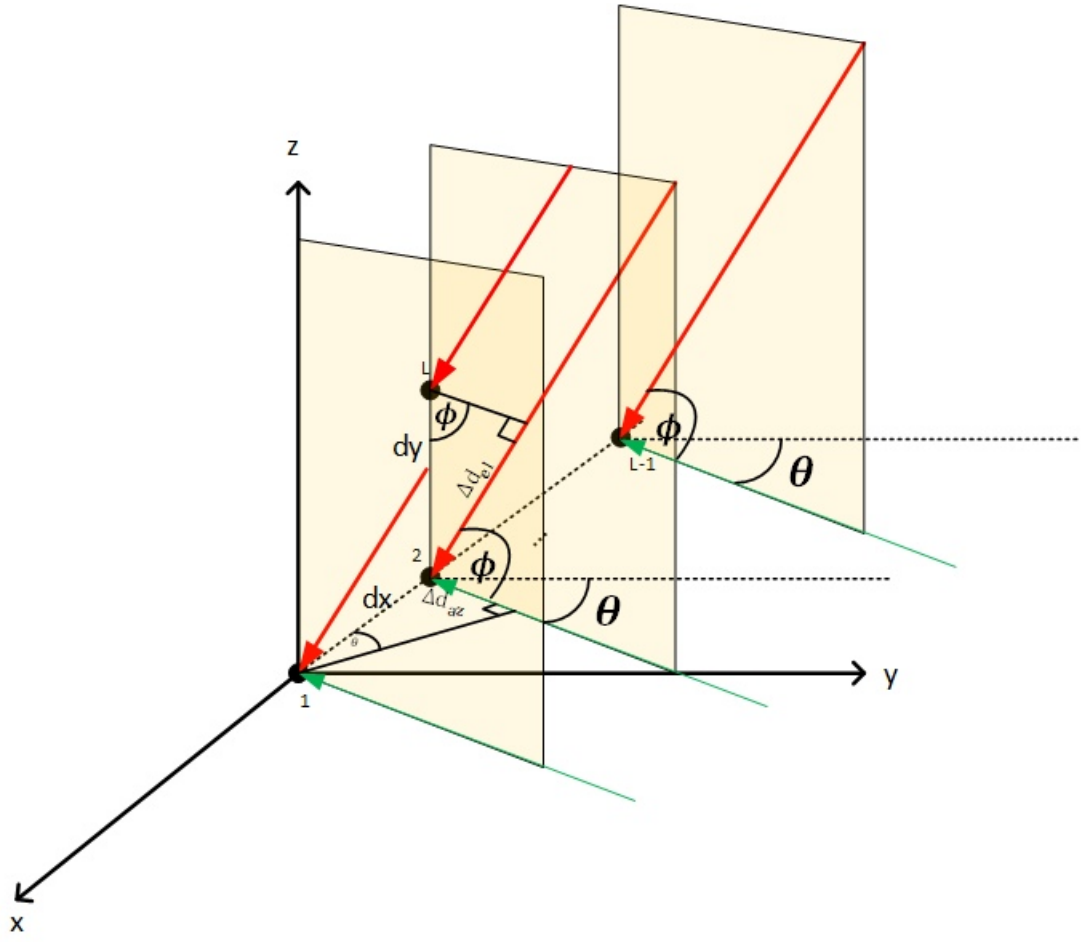

Fig. 4: Steering vector construction based on the array configuration

2.5 High-resolution spectral estimation

2.6 Object detection with CFAR

2.7 References

CHAPTER

THREE

RADAR CALIBRATION

3.1 References

CHAPTER

FOUR

NOTES ON PARALLEL PROCESSING

Gilles Kahn provided clear semantics of **process interaction** which facilitates well-structured programming of dynamically evolving networks of processes. - At any given point, a process can be either enabled or blocked waiting for data on only one of its input channels: **it cannot wait for data from more than one channel**. - A proper schecduling property should :

- infinite running loop
- avoid deadlock[1]
- control the buffer sizes to be limited
- achieve maximum throughput by managing the memories

[1] In *concurrent computing*, a deadlock is a state in which each member of a group is waiting for another member, including itself, to take action, such as sending a message or **more commonly releasing a lock**.

CHAPTER

FIVE

HOW TO USE RADAR APPLICATION

First of all we need to import necessary libraries:

```
import socket
```

5.1 Dependencies

- *numpy*
- *pyqtgraph*
- *PyQt5*
- ...

5.2 Range detection

The following is an examplef for plotting range profiles in **real-time**:

```
import socket

UDP_IP = '192.168.33.30'
UDP_port = 4098
UDP_buff_size = 65000

DCASock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_IP)
DCASock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# setting the socket buffer size
DCASock.setsockopt(socket.SOL_SOCKET, socket.SO_RCVBUF, UDP_buff_size)
DCASock.bind((UDP_IP,UDP_port))
```

CHAPTER

SIX

RADAR PACKAGE

6.1 RadarFcnLib module

This module is intended for defining all radar functions from configuring, controling, running the radar, and reading DCA1000 packets down to obtaining different maps and higher layer of signal processing. Each function is compared against the Matlab outputs or it is tested within Python by a sample data.

6.1.1 Tracking UPDATES:

Date	Description
March-6-2019	Function annotations are added.
March-6-2019	Unnecessary initial variables are removed in *unpackDCApayload*.
March-6-2019	Timers are commented in *readADCbits*. These are for tracking the execution time.
March-10-2019	packetReorderZeroFilling considers the cases where zero filling process fills the buffer more than one round.
March-10-2019	*readADCbits* assumes that the received byte stream is flattened in contrast to what was assumed before.
March-12-2019	*readADCbits* should do np.reshape by *F* order otherwise the samples of each channel will be interleaved.
March-12-2019	*FormingVirtualChans* should do np.reshape by *F* order otherwise the samples of each channel will be interleaved.
May-02-2019	The table is added.
May-04-2019	*DCA1000_CMD_RESPONSE_SEQ* added.
Jun-01-2019	*build_uis* function added.
Jun-07-2019	*SockBuf* input is removed from *packetReorderZeroFilling*.
Jun-22-2019	*DataExport* input is removed from *packetReorderZeroFilling* and instead the function returns the export buffer. This is not mmeory efficient since it requires twice memory as before: one is for export buffer and the other for Q buffer outside the function.

Author Mostafa Alizadeh

radar.RadarFcnLib.**AveRemove**(*a: numpy.array*, *retAve: bool*, *retAveRm: bool*)

This function takes in a 3 dimensional array [x,y,z] as input and takes the average along the second dimension so the average array size would be [x,1,z]. When the average is taken, it will only create a matrix with dimension [x,z] as dimension y has been shrunk into one row. This matrix will be re-sized to return the dimensions [x,1,z] which would then be subtracted from the initial Array A to return the removed average array. The new averaged removed matrix will be returned.

Parameters

- **a** (*np.array*) – 3D input array
- **retAve** (*bool*) – whether or not return the average value
- **retAveRm** (*bool*) – whether or not return the removed average array

Returns based on *retAve* and *retAve* returns are determined

radar.RadarFcnLib.**CFAR2D**()

radar.RadarFcnLib.**DCA1000_CMD_RESPONSE_SEQ**(*CMD_SEQ: list*, *DCA1000Sock: socket.socket*, *DestAddr: ()*)

Sends and receives a sequence of message-reponses to and from the DCA1000 board. All the DCA1000 commands are supported except `RESET_AR_DEV_CMD_CODE`, `PLAYBACK_START_CMD_CODE`, `PLAYBACK_STOP_CMD_CODE`, `SYSTEM_ERROR_CMD_CODE`, `CONFIG_DATA_MODE_AR_DEV_CMD_CODE`, `INIT_FPGA_PLAYBACK_CMD_CODE`. The function sends a set of predefined commands which set default parameters for the DCA1000. The followings are the the default values for different parameters:

- **CONFIG_FPGA_GEN_CMD_CODE:**

Parameter	Default value
Raw data mode	1
Device	AWR1243
Capture	LVDS
Stream over	Ethernet
ADC bits	16 bits
Timer info.	in seconds

- **CONFIG_EEPROM_CMD_CODE:**

Parameter	Default value
FPGA MAC Addr.	12:90:78:56:34:12
Config. Port #	4098

- **CONFIG_PACKET_DATA_CMD_CODE:**

Parameter	Default value
Eth. packet size	1472
Eth. Packet delay	25 ns

To start recording from the board, you should send the following sequence of commands:

1. `SYSTEM_CONNECT_CMD_CODE`
2. `CONFIG_FPGA_GEN_CMD_CODE`
3. `CONFIG_PACKET_DATA_CMD_CODE`
4. `RECORD_START_CMD_CODE`

To stop recording send RECORD_STOP_CMD_CODE command. For the input arguments and the outputs see *parameters*.

Parameters

- **CMD_SEQ** (*list[int]*) – The list of commands to be sent in order.
- **DCA1000Sock** (*socket.socket*) – The socket which is bound to the (System IP, configuration port). It is used for both send and receive of the commands and responses.
- **DestAddr** (*tuple[int,int]*) – FPGA (IP, port) address.

Returns A list of true and false indicating whether a message is submitted correctly or not. :rtype: list[int]

radar.RadarFcnLib.**FFTrmNegativeFreqs**(*InMat: numpy.array*, *Nfft: int = None*)

Takes FFT over columns of the input array (3D, 2D, or 1D array) and removes the negative frequencies. FFT is applied on the first axis.

TODO: Add windowing as an option.

Parameters

- **InMat** (*np.array*) – Input array.
- **Nfft** (*int*) – FFT size. If it is None then the FFT size is equal to the size of the columns, otherwise it is Nfft. However, the size along axis=0 should be an integer multiple of 2.

Returns A NumPy array containing the FFT of the input array with removed negative frequencies.

radar.RadarFcnLib.**FormingVirtualChans**(*Inmat: numpy.array*, *NumTxAnt: int*, *NumChirpSmps: int*)

The *Inmat* is reshaped to a 3D array whose the 3rd dimension index is the virtual channel index. The input should have columns corresponding to each actual Rx channel samples. The 2D array made of the first two dimensions of the output 3D array contains chirp samples of a virtual channel.

Parameters **Inmat** (*np.array*) – A matrix corresponding to the received data from actual Rx channels. Columns of the matrix should contain Rx channel data. It is assumed that the *Inmat* is a NumPy 2D array. Note that the order of Rx channels are from the first to the last column.

Returns A 3D array with *NumVirtualChan = NumTxAnt * Inmat.shape[1]*.

radar.RadarFcnLib.**build_uis**(*SRC: str = '.'*, *DEST: str = '.'*)

Building all '.ui' files in the *SRC* directory and put the results in *DEST* directory.

Parameters

- **SRC** (*str*) – source directory.
- **DEST** (*str*) – destination directory.

radar.RadarFcnLib.**packetReorderZeroFilling**(*seqNum_: int*, *SeqNum: int*, *ByteCount: int*, *ByteCnt_: int*, *PayBits: bytearray*, *BufPt: int*, *RxBufSize: int*, *ExportLen: int*, *ByteBuf: bytearray*, *ZeroBytes: bytearray*)

The function does packet reordering and zero filling for the received packets in *PayBits*. In fact, it checks whether the sequence numbers in *PayBits* are in order or not. If it is not, it will add appropriate amount of zero bytes to the buffer i.e. *ByteBuf*. After filling the buffer or when it is "almost" full, it puts the first *ExportLen* bytes to *DataExport* queue. Consider that buffers should be mutable object to make changes on them such that they will be available outside the function.

Parameters

- **seqNum** (*int*) – The previous received sequence number.
- **SeqNum** (*int*) – The current received sequence number.
- **ByteCount** (*int*) – The current byte count from DCA1000.
- **ByteCnt** (*int*) – The previous bytecount from DCA1000.
- **PayBits** (*bytearray*) – Payload bits received from DCA1000.
- **BufPt** (*int*) – A pointer to the current position of ByteBuf to fill the buffer.

- **RxBufSize** (*int*) – The size of the buffer.
- **ExportLen** (*int*) – The size of the output buffer
- **ByteBuf** (*bytearray*) – The input buffer.
- **ZeroBytes** (*bytearray*) – The zero-bytes array is a pre-allocated memory to reduce the memory usage by avoiding re/allocating of the zero-byte memory.

Returns The updated position of the buffer pointer and the *DataExport* buffer.

radar.RadarFcnLib.**readADCbits**(*ADCbits: bytearray*, *NumADCBits: int*, *isReal: bool*)

Reads ADC bits from input and output *uint* samples i.e. 2-byte samples with little-endian order. This is a duplication of Matlab function provided by TI. It returns a matrix having columns with the received samples of each Rx channel.

Parameters

- **ADCbits** (*bytearray*) – Input list of bits read from the *DCA1000* packets
- **NumADCBits** (*int*) – Number of ADC bits within each real/imaginary sample (# bits of ADCs).
- **isReal** (*bool*) – An indication whether or not ADC has real or complex samples.

Returns Returns the output matrix.

radar.RadarFcnLib.**readStoredDCA1000**(*FileNames: []*, *RxBufSize: int*, *DataExport: queue.Queue*)

Reads the binary files stored by *mmave studio* before packer reordering and zero filling. This function is good for offline analysis. The input *FileNames* is a list of binary file names to do packet reordering and zero filling on them in order.

Parameters

- **FileNames** – a list of file names in a correct sequence to read the binary DCA packets saved by *mmwave sutdio*.
- **RxBufSize** (*int*) – Buffer size for *packetReorderZeroFilling*.
- **DataExport** (*queue.Queue*) – The thread *queue* for exporting reordered data.

Returns True if it was successful, otherwise False.

radar.RadarFcnLib.**rmAntCoupling**(*VirChans: numpy.array*, *MAxCalibRngIdx: int*, *AveLen: int*)

Calculates the antenna coupling for the close ranges determined by *MAxCalibRngIdx*. Computes the average for the first bunch of samples with length *AveLen*. Also note that, it takes the average on the complex samples not on magnitude or phase separately.

Parameters

- **VirChans** (*np.array*) – Input 3D array which comes normally from the output of *FFTrmNegativeFreqs()*
- **MAxCalibRngIdx** (*int*) – Zero indexing of the maximum range index such that the antenna coupling effect will be calculated for the ranges up to that range index.
- **AveLen** (*int*) – Averaging length.

Returns The antenna coupling compensation array, which can be used later for removing the antenna coupling signature.

radar.RadarFcnLib.**tst_dll**()

radar.RadarFcnLib.**unpackDCApayload**(*fHand*)

Takes a file handler and returns the ADC bits. The file is a binary file stored by mmwave studio in the format mentioned in the TI user guide of "DCA1000EVM Data Capture Card" section 5.3.

Parameters **fHand** – File handler

return Sequence numbers in a list.

Returns Payload sizes in a list

Returns ADC bits in a binary sequence

6.2 params module

All the radar constant values.

CHAPTER

SEVEN

APPLICATION PACKAGE

class app.mainForm1.**Ui_plots**

Bases: object

The main UI form.

retranslateUi(*plots*)

setupUi(*plots*)

7.1 Module contents

CHAPTER

EIGHT

ERROR HANDLING

The radar application has three layers: 1. *mmwavelink layer*: the mmwavelink is the interface to the radar chip for programming, configuring, and monitoring the radar. It has four different errors:

- mmwave app: if an exception raised at this stage, the error raised due to the following reasons:
  - The radar and the **SPI board** (see Glossaries) are not connected
  - The radar and the **SPI board** are connected but they are not recognized by the host
- Profile config: profile configuration parameters are wrong
- Chirp config: chirp configuration parameters are wrong
- Frame config: FMCW frame configuration parameters are wrong

2. *baseband layer*: the baseband signal processing includes deriving *range-slowtime* map, *range-Doppler* map, *range-azimuth* map, and *3D point cloud*. This layer has the following errors: *TODO*

- Reading packets: ...

3. *application layer*:

exception Errors.**DopplerErr**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**FormingVirChanErr**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**IOError**
Bases: *Errors.basebandErr*

exception Errors.**PortConnection**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**RangeErr**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**WorkerReturn**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**appErr**(*msg*)
Bases: Exception

exception Errors.**basebandErr**(*msg*)
Bases: Exception

exception Errors.**chirpConfErr**(*msg*)
Bases: *Errors.mmwavelinkErr*

exception Errors.**frameConfigErr**(*msg*)
Bases: *Errors.mmwavelinkErr*

exception Errors.**mmwAppErr**(*msg*)
Bases: *Errors.mmwavelinkErr*

exception Errors.**mmwavelinkErr**(*msg*)
Bases: Exception

exception Errors.**packetorderingErr**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**profConfErr**(*msg*)
Bases: *Errors.mmwavelinkErr*

exception Errors.**readingADCbitsErr**(*msg*)
Bases: *Errors.basebandErr*

exception Errors.**readingDCA1000Err**(*msg*)
Bases: *Errors.basebandErr*

CHAPTER

NINE

PROCESSING MODULE

9.1 Processing module

This module contians all processing classes derived from the base class. The base class has mechanisms to handle the data passing between each processing block.

class processing.**base_processing**
Bases: object

The abstract class for all signal processing ecosystem.

class processing.**p_source_block**(*BlockName: str*, *NPorts: int*)
Bases: *processing.process_blocks*

The process block source which is passing generated data down to the process

run()
Method to be run in sub-process; can be overridden in sub-class

worker()

class processing.**process_blocks**(*name*)
Bases: *processing.base_processing*, multiprocessing.context.Process

This is to derive all multiprocessed blocks to achieve full parallelism. TODO: setting daemon returns error:

assert self._popen is None, 'process has already started' AttributeError: 'DCAReader' object has no attribute '_popen'

delay_calc()

run()
Method to be run in sub-process; can be overridden in sub-class

worker()

class processing.**t_general_block**(*BlockName: str*, *NInPorts: int*, *NOutPorts: int*)
Bases: *processing.thread_blocks*

All threaded general blocks should be derived from this class. Initializing parameters are:

Parameters

- **BlockName** (*str*) – name of the processing block
- **NInPorts** (*int*) – number of output queues
- **NOutPorts** (*int*) – number of output queues

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

worker()

class processing.**t_sink_block**(*BlockName: str*, *NInPorts: int*)
Bases: *processing.thread_blocks*

All threaded sink blocks should be derived from this class. Initializing parameters are:

Parameters

- **BlockName** (*str*) – name of the processing block
- **NInPorts** (*int*) – number of output queues

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

worker()

class processing.**t_source_block**(*BlockName: str*, *NPorts: int*)
Bases: *processing.thread_blocks*

All threaded source blocks should be derived from this class. Initializing parameters are:

Parameters

- **BlockName** (*str*) – Name of the processing block
- **NQs** (*int*) – Number of output queues

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

worker()

class processing.**thread_blocks**(*name*)
Bases: *processing.base_processing*, threading.Thread

This is to derive multithreaded blocks.

delay_calc()

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

worker()

class processing.**threaded_process_blocks**(*ProcName: str*)
Bases: *processing.process_blocks*, *processing.thread_blocks*

Multithreaded processes ...

get_threads(*ListThreads*)

run()
Method to be run in sub-process; can be overridden in sub-class

worker()

class processing.**top_block**
Bases: *processing.top_flow*

All signal processing blocks in the *flowgraph* should be connected by this class. It manages for the correct flow connection of the graph from the source set to the sink set. It manages the IOs between the blocks.

add_Qs(*blck1: processing.base_processing*, *port1*, *blck2: processing.base_processing*, *port2*)

connect(*block1*, *port1: int*, *block2*, *port2: int*)

cpu_trace()

print_arcs()

print_nodes()

throughput_calc()

update_A(*A_*)

class processing.**top_flow**
Bases: *processing.base_processing*

The signal processing *flowgraph* parent class which handles the signal flow in the graph.

connect()

flow_start()

flow_stop()

get_A()

get_cpu_usage()

handle_A()

mem_available()

mem_used()

num_cores()

set_A(*A_*)

class processing.**tp_source_block**(*ProcName: str*, *NPortsIn: int*, *NPortsOut: int*)
Bases: *processing.thread_blocks*

A source for threaded process which takes a list of input *multiprocess.Queue()* to collect data from other processes and pass them down to the threads in the process by output *queue.Queue()*.

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

worker()

CHAPTER

TEN

EXAMPLE OF USING RADAR API

The following video shows the example:

CHAPTER

ELEVEN

GLOSSARIES

SPI board	a board with a SPI connection to a radar chipset.
RCS	is a hypothetical area required to intercept the power density at the target such that if the total reflected power is scattered isotropically, the power density at the Rx is achieved.
NF	Noise figure is defined as the ratio between the input SNR to the output SNR of a circuit component.

CHAPTER

TWELVE

COPYRIGHT

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX