

## Data Streaming in MATLAB

This lab will focus on connecting MATLAB to ADI transceiver platforms and streaming data. We will start with the basics of connecting to a device using System objects. Then we will perform some simple exercises with the radio and MATLAB signal processing libraries.

### Checking Radio Connectivity

The first step before we can stream data to any of the ADI SDR devices is to check their connectivity. After connecting a PlutoSDR through USB to your PC, open MATLAB, and move to the folder containing the files necessary for Lab 1:

```
Command Window
>> cd C:\Seminar\lab1\
```

Next, in the command windows type the following and hit enter:

```
Command Window
>> plutoradiosetup
```

This will set up the MATLAB environment for PlutoSDR. Next in the command window type:

```
Command Window
fx>> findPlutoRadio
```

If connected, that command should return something similar to:

```
Command Window
>> findPlutoRadio
ans =
struct with fields:
    RadioID: 'usb:0'
    SerialNum: '104473222a870017fdff070014ced4f484'
fx>> |
```

Each Pluto device will have a unique Serial Number and a unique USB enumeration for a given machine. Multiple Plutos can be plugged into the same machine and their RadioIDs will enumerate automatically.

## MATLAB System Objects for SDR Devices

Each ADI SDR device will have a unique associated System object. System objects are special classes inside MATLAB which share certain procedural methods and APIs, This makes them useful as an interface for hardware, or any data structure that requires state and has specific associated information.

In your MATLAB command line, type the following and hit enter:

Command Window

```
fx>> rx = sdr_rx('Pluto'), tx = sdr_tx('Pluto')|
```

This will initialize both a receive (rx) and transmit (tx) System object for a single PlutoSDR device.

This will output something like:

```
Command Window

>> rx = sdrxx('Pluto'),tx = sdrtx('Pluto')

rx =

comm.SDRRxPluto with properties:

    Main
        DeviceName: 'Pluto'
        RadioID: 'usb:0'
        CenterFrequency: 2.4000e+09
        GainSource: 'AGC Slow Attack'
        ChannelMapping: 1
        BasebandSampleRate: 1000000
        OutputDataType: 'int16'
        SamplesPerFrame: 20000

    Show all_properties

tx =

comm.SDRTxPluto with properties:

    Main
        DeviceName: 'Pluto'
        RadioID: 'usb:0'
        CenterFrequency: 2.4000e+09
        Gain: -10
        ChannelMapping: 1
        BasebandSampleRate: 1000000

    Show all_properties

fx >> |
<
```

Displayed are the parameters of the System objects, which control the radio's sample rate, LO frequency, gain settings, and other configurations. For all SDR devices provided by MathWorks, there will be independent System objects for transmit and receive capabilities.

These parameters can be modified in two ways in MATLAB: A parameter can be set during instantiation:

```
Command Window

fx >> rx = sdrxx('Pluto','CenterFrequency',5.2e9)|
```

or, a parameter can be set through “dot notation” after instantiation:

### Command Window

```
>> rx = sdr_rx('Pluto');  
fx>> rx.CenterFrequency = 5.2e9;
```

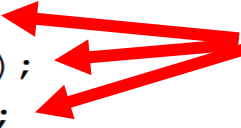
Parameters that require substantial configuration updates, like Sample Rate, may not be tunable in some cases. They can become locked after data is sent to or received by the radio. However, Center Frequency or Gain can be changed at any time.

### Getting Data from SDR Devices

Data is transferred to and from the SDR device in buffers which are also called frames in MATLAB. Using the System object operator, we can send or receive data from the device. Internally, this is called the *step* method and the following calls are equivalent:

### Command Window

```
>> rx = sdr_rx('Pluto');  
>> data = rx();  
>> data = rx.step();  
>> data = step(rx);  
fx>> |
```



Identical

***Run one of these three lines above in your command window and hit enter. Inspecting the output vector “data” and the object attributes as we do in Figure 1.***

```
Command Window
>> data = rx();
>> size(data)

ans =

    20000         1

>> rx.SamplesPerFrame

ans =

    20000

>> data(1:10).'

ans =

1×10 int16 row vector

Columns 1 through 7

    -15 -     50i    35 +     71i     5 +

Columns 8 through 10

     90 -     52i    34 +     49i    -7 +

>> rx.OutputDataType

ans =

'int16'

fx>> |
```

Figure 1

The receiver object can return three different parameters during reception: IQ data, valid out, and an overflow condition.

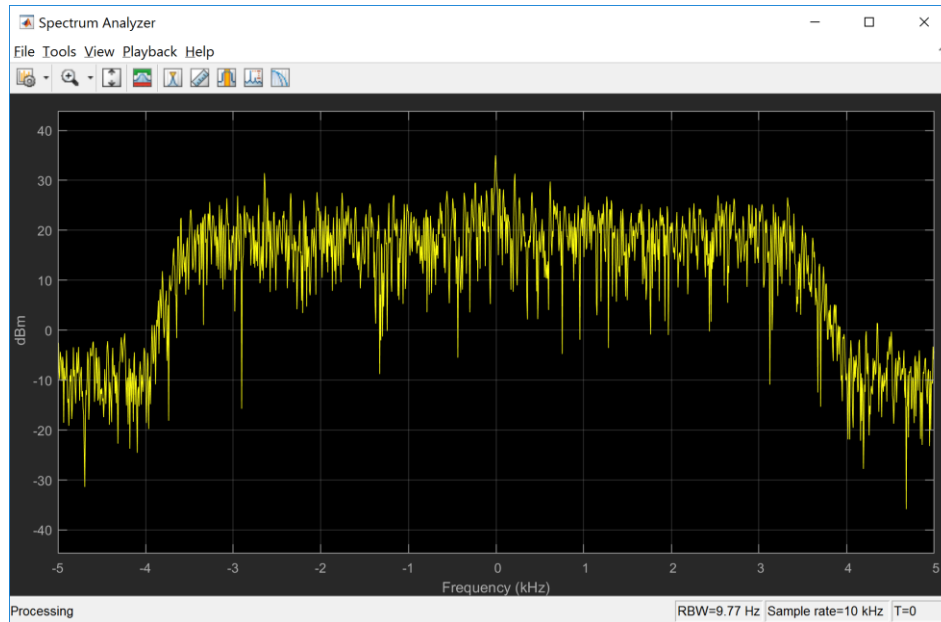
```
Command Window
fx>> [IQData, Valid, Overflow] = rx() |
```

Let us first look at the IQ data from the SDR, which by default will be an int16 data type, but can be cast to double. **Run the following commands which will collect data from the receiver and display it in a spectrum scope:**

### Command Window

```
>> sa = dsp.SpectrumAnalyzer();  
>> rx = sdrx('Pluto', 'SamplesPerFrame', 2^14);  
fx >> sa( rx() );
```

You should be able to view a similar plot to the one shown below:



It can be useful to scale the axes using the button on the Spectrum Analyzer scope.


Now we can repeatedly pull data (stream) from the radio and view it in the Spectrum Analyzer **by running the following command which uses the objects we just created:**

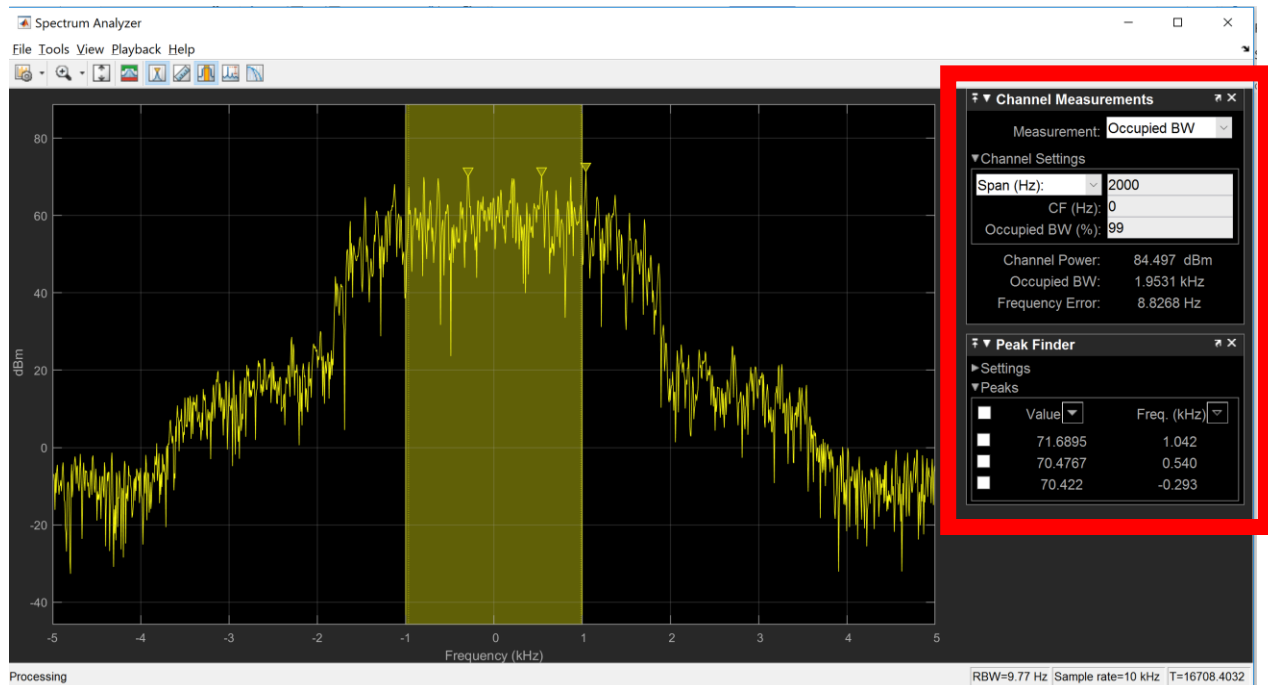
### Command Window

```
fx >> for k=1:1e4, sa( rx() ); end
```

While we are streaming data, we can enable measurements on the Spectrum Analyzer. **Push the Peak**



**Finder button** and the **Channel Measurements button**  on the Spectrum Analyzer. These will open measurement panels in the spectrum analyzer:



*If you know a nearby FM station (89.9 MHz is used below for example), update the Center Frequency of the receiver to that frequency and inspect the receive power of that station. The center frequency can be updated on the System object with the MATLAB command:*

#### Command Window

```
>> rx.CenterFrequency = 89.9e6;
fx>> for k=1:1e4, sa(rx());end
```

### Transmitter Functionality and AGC Operation

MATLAB is a sequential language and cannot perform multiple tasks simultaneously. For example, triggering transmit and receive events simultaneously cannot be done in a single MATLAB script. To enable simultaneous transmit and receive, a method exists in the transmitter System Object called "transmitRepeat" which can be used to continuously transmit a signal. Below is the API to do so but is provided as just an example. **Do not run the code below**

```
% Set up TX
Tx = sdrTx('Pluto');
Tx.transmitRepeat(<waveform>);
% Now the transmitter will continuously repeat
% the vector "waveform" without gaps
```

Another feature of AD936x family of transceivers is that all contain internal Automatic Gain Control functionality (AGC). By default, they can be configured in the following modes:

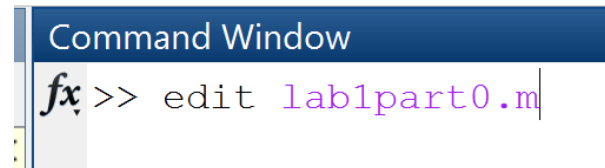
- Manual

- Fast Attack
- Slow Attack

These modes are configurable, and their operation can be explored in the AD9361 simulation model.

We will use the transmit repeat functionality to see how the AGC reacts over time in each of its modes.

**Start by opening script *lab1part0.m* from the command line as:**



In the second block (starting at line 7) of this script, we generate a complex sinusoid for testing, as shown below:

```
%% Generate test sinewave
amplitude = 1; frequency = 1e4;
swv1 = dsp.SineWave(amplitude, frequency);
swv1.SampleRate = SampleRate;
swv1.SamplesPerFrame = SamplesPerFrame;
swv1.ComplexOutput = true;
y = swv1();
```

Next a single radio is configured, with the receiver in “Slow Attack Mode”, in which is set in line 17:

```
15 %% Show AGC changing
16 rx = sdrx('Pluto', 'SamplesPerFrame', SamplesPerFrame);
17 rx.GainSource = 'AGC Slow Attack';
18 tx = sdrx('Pluto', 'SamplesPerFrame', SamplesPerFrame);
19 tx.transmitRepeat(y);
```

When the script is run, the transmitter is initialized with an attenuation of -20dB. Then, after half the frames are received, the transmitter is reconfigured with a gain of 0dB

```
for k=1:Frames
    z(k,:) = rx();
    % Update gain halfway through
    if k==floor(Frames/2)
        tx.Gain = 0;
        tx.transmitRepeat(y);
        disp('Gain Change');
    end
end
```



**Run this script to by typing:**

Command Window

```
fx>> lab1part0
```

The plot observed should be similar to Figure 2. The AGC initially converges after the first 1,000 samples. At  $\sim 1.75 \times 10^5$  samples, the transmitter is reconfigured with a new gain. This new signal begins at  $\sim 2.2 \times 10^5$  samples, where it saturates the receiver and slowly ramps down to the desired output level.

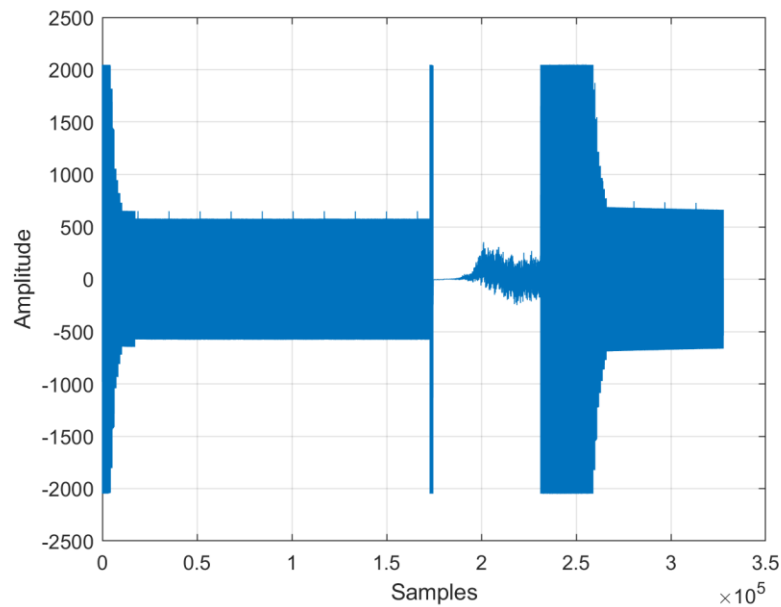


Figure 2 Received amplitude of sinusoid from lab1part0.m script

**Now we can observe the “Fast Attack” mode by modifying line 17 to change the AGC mode:**

```
15 %% Show AGC changing
16 rx = sdrx('Pluto','SamplesPerFrame', SamplesPerFrame);
17 rx.GainSource = 'AGC Fast Attack';
18 tx = sdrtx('Pluto','Gain',-20);
19 tx.transmitRepeat(v);
```

**Now rerun the script with command**

Command Window

```
fx>> lab1part0
```

**and the resulting plot should look similar to Figure 3.** In this case, the signal converges to the target power level much faster.

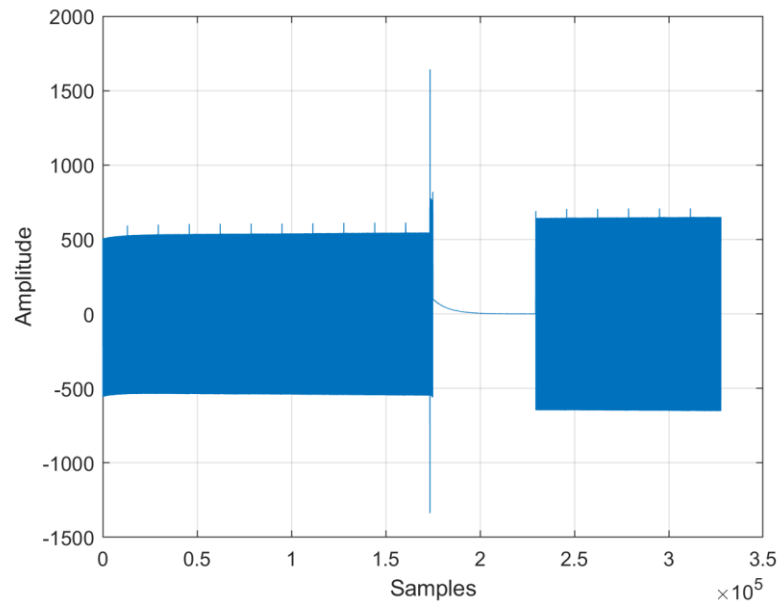


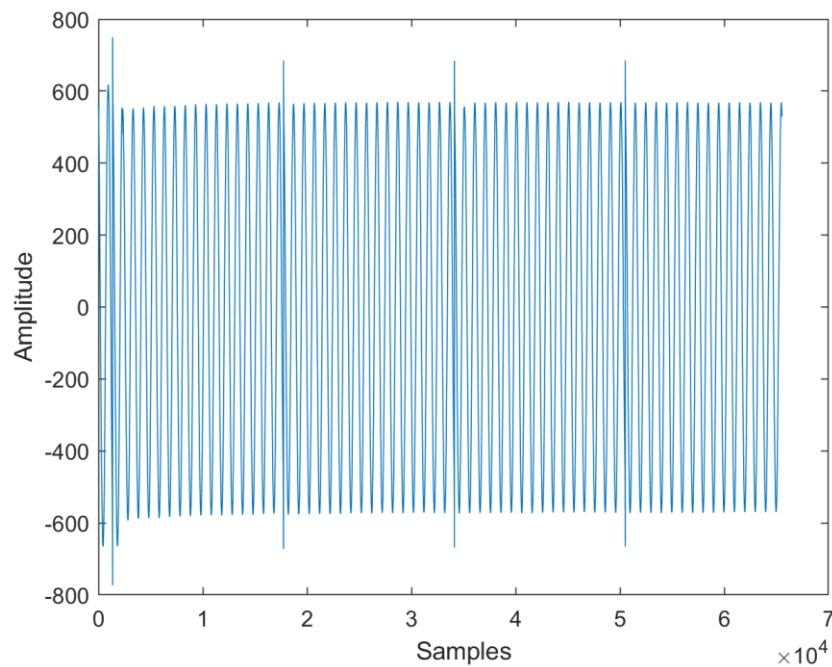
Figure 3 Received amplitude of sinusoid from lab1part0.m script in Fast-Attack mode

### Understanding Buffers and Real-Time Data Collection

Open the script “lab1part1.m” with the command:

```
Command Window
fx >> edit lab1part1.m
```

Run the script and notice the periodic spikes that occur in the received data. These should look similar to:



Question 1: What is the source of these spikes?

Question 2: Suggest a solution to remove these spikes in the data

### Overflows

The AD9363 transceiver in the PlutoSDR is capable of operating at 61.4 MHz of complex bandwidth, which is suitable for many modern radio standards. However, downstream bottlenecks will usually limit data transfer speed. For example, USB2.0 will limit data transfer to 25 MB/s, or ~6.5 MS/s of data rate (6.5 MHz of complex bandwidth). A detailed outline of the data path from the transceiver to MATLAB is outlined in Figure 4. You should keep this in mind when streaming data from the transceiver as there will be latency introduced in the USB transmission due to the added overhead. Once data gets to the PC it must be processed, and even high-end PCs, performing calculations (especially double precision ones) will be difficult at this speed.

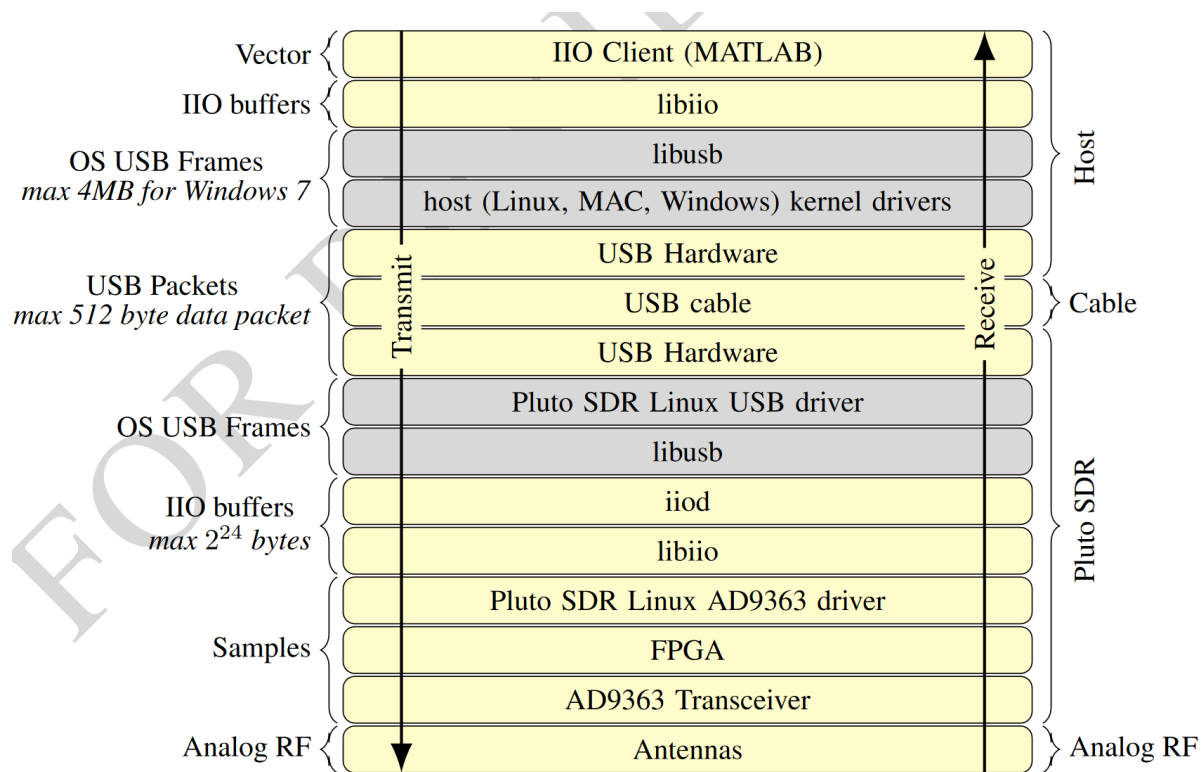


Figure 4 Data path from AD9363 to MATLAB

For applications that are relatively low in bandwidth, or those that require less complex processing, it may be possible to operate in “real-time” in MATLAB and Simulink, and the Host PC can process data faster than it can be produced by the device. Our goal here, however, is to develop a deployable system, so optimizing MATLAB code for simulation speed is not a priority. Therefore, when performing data “streaming”, we are actually operating on bursts of data, and not a continuous stream, and need to be mindful of data overflows and ensure that sufficient data is captured in each burst to provide a meaningful assessment of our algorithm.

Data overflow is a condition where data is lost because the step method of the radio object is not called fast enough. Checking the output of the step-call to the radio will tell us if this condition occurred. To reduce this possibility, we utilize coding styles, or templates, that avoid upfront processing, capture a large chunk of contiguous data, and then perform processing on that data. This ensures that enough contiguous data is captured by the host PC to test the algorithms.

**We will start by opening the script `lab1part2.m` from the command line:**

```

Command Window
fx >> edit lab1part2.m

```

This script has two main parameters to configure on lines 12 and 13:

```
%% CHANGE ME HERE

% Overflow only in SA
SampleRate = 4.5e6;
SamplesPerRXFrame = 2^16;
```

- **SampleRate** is the rate the device is configured to collect data
- **SamplesPerRXFrame** is the size of the buffer transfer from PlutoSDR to MATLAB.

The remaining part of the script will check for overflow conditions while data is processed in the capture loop, which drives a Spectrum Analyzer scope visualization. We also check for overflow conditions when this visualization is done after all the data is collected.

**Run the script *lab1part2* and view the information displayed on the console:**

```
>> lab1part2
(OUTLOOP) Overflow events: 0 of 100
(INLOOP) Overflow events: 59 of 100
```

The displayed numbers relate to the number of overflows that occurred. This is a great example of why we might not want to do processing within the capture loop. We can try to reduce these overflow events by either increasing the **SamplesPerRXFrame** value, which reduces the overhead per sample of data pull from PlutoSDR, or we can reduce the **SampleRate** if our application allows.

**Edit line 13 to reduce the *SamplesPerRXFrame* to  $2^{15}$  as follows:**

```
11 % Overflow only in SA
12 SampleRate = 4.5e6;
13 SamplesPerRXFrame = 2^15;
14 FramesToCollect = 1e2;
15
```

**Now rerun the script and observe overflow conditions that occur in both cases:**

```
>> lab1part2
(OUTLOOP) Overflow events: 78 of 100
(INLOOP) Overflow events: 99 of 100
fx >>
```

Now try reducing the sample rate. **Edit line 12 to reduce the *SampleRate* as follows:**

```
11 % Overflow only in SA
12- SampleRate = 1e6;
13- SamplesPerRXFrame = 2^15;
14- FramesToCollect = 1e2;
```

**Rerun the script and observe overflow conditions that occur in both cases:**

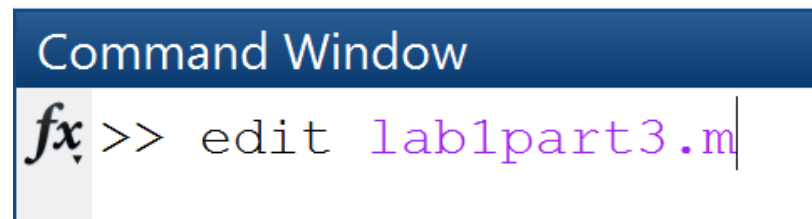
```
>> lab1part2
(OUTLOOP) Overflow events: 0 of 100
(INLOOP) Overflow events: 0 of 100
```

**Question:** The maximum SamplePerFrame possible is  $2^{20}$ . At this size, what is the maximum rate you can collect data without overflow?

### Signal Processing Is Easy (By Example)

In this exercise we will explore some of the communication signal processing pieces available in MATLAB and how they can be used with PlutoSDR.

Starting with script lab1part3.m:

A screenshot of the MATLAB Command Window. The title bar is dark blue with the text "Command Window" in white. The main area has a light gray background. On the left, there is a small icon of a notepad with the letters "fx" in a stylized font. To the right of the icon, the text ">> edit lab1part3.m" is displayed in a purple monospace font. A vertical cursor line is positioned at the end of the text.

```
fx >> edit lab1part3.m
```

This script sets up a loopback-based system which utilizes a QPSK signal to transmit data through PlutoSDR. Since we control both the transmitter and receiver, and since they share the same clock, we can perform some useful testing with our algorithms.

The default script generates a signal, passes it through PlutoSDR, and performs Timing Recovery with the “comm.SymbolSynchronizer” System Object. Timing recovery is necessary since there is a random delay between transmitter and receiver and the transmitter and receiver LO’s on Pluto have random phase differences.

**Run the unedited script and inspect the constellation diagrams produced.**

## Command Window

```
>> lab1part3
Overflow events: 0 of 10
Performing Timing Synchronization
fx>> |
```

**Running the script multiple times, you will observe different constellations each time.** This is due to the impairments already discussed. An example constellation set is shown in Figure 5.

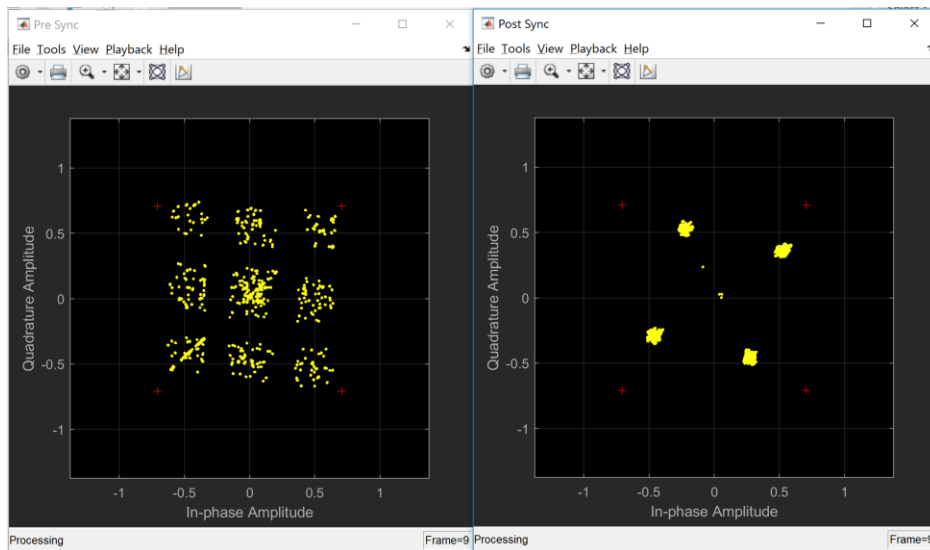


Figure 5 Constellation of QPSK signal before and after timing recovery

**Next modify line 10 in the provided script to change the frequency offset between radios to 100Hz:**

```
6      %% CHANGE ME HERE
7      SampleRate = 1e6;
8      SamplesPerRXFrame = 2^16;
9      FramesToCollect = 10;
10     FrequencyOffset = 100;
```

**Now run the script again with this change.**

## Command Window

```
>> lab1part3
Overflow events: 0 of 10
Performing Timing Synchronization
fx>> |
```

Looking at the constellations in Figure 6 we can notice a rotation, which is expected due to the frequency difference between the receiver and transmitter. Since PlutoSDR has independent LO's for the transmitter and receiver, we can configure the radio in this mismatched configuration.

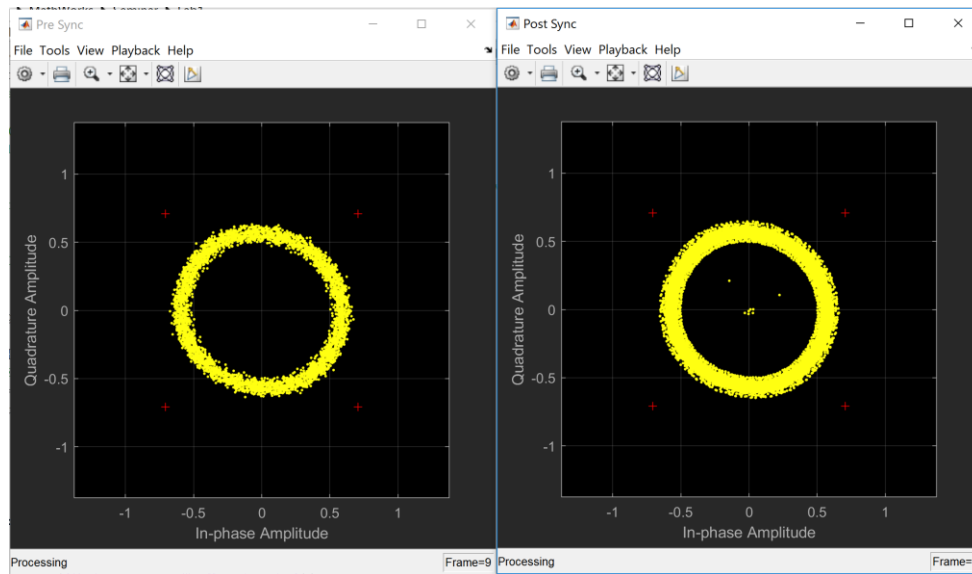


Figure 6 Constellation of signal before and after timing recovery with remaining carrier offset.

When transmitting between two different radios, we will always have to compensate for frequency offset since they will have different LO's. We are emulating this effect in a single radio by selecting different center frequencies for receive and transmit.

We can compensate for this carrier frequency offset by performing carrier synchronization. For this we will use the `comm.CarrierSynchronizer` System object, which implements the necessary algorithms to perform carrier recovery on a timing synchronized signal.

**To enable this synchronization, uncomment lines 47-49 so they appear as:**

```
46      %% Insert Carrier Synchronization here
47      fprintf('Performing Carrier Synchronization\n');
48      cs = comm.CarrierSynchronizer('SamplesPerSymbol',1);
49      savedPost = cs(savedPost);
```

**Rerun the script with these modifications to visualize the timing- and frequency-synchronized signal.** This should look similar to Figure 7 below, which has the correct phase orientation to the constellation for QPSK.



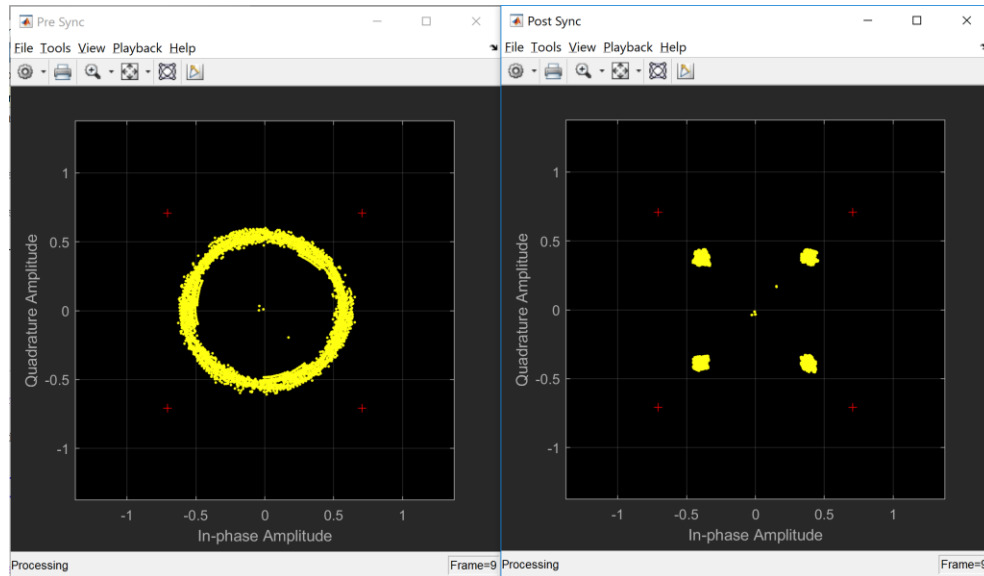


Figure 7 Constellation of QPSK recovery signal before and after carrier and timing synchronization

**Now modify line 10 to increase the offset to 4Khz and rerun the script.**

We now observe there is still rotation even after synchronization, producing a constellation similar to Figure 6 as in Figure 8.

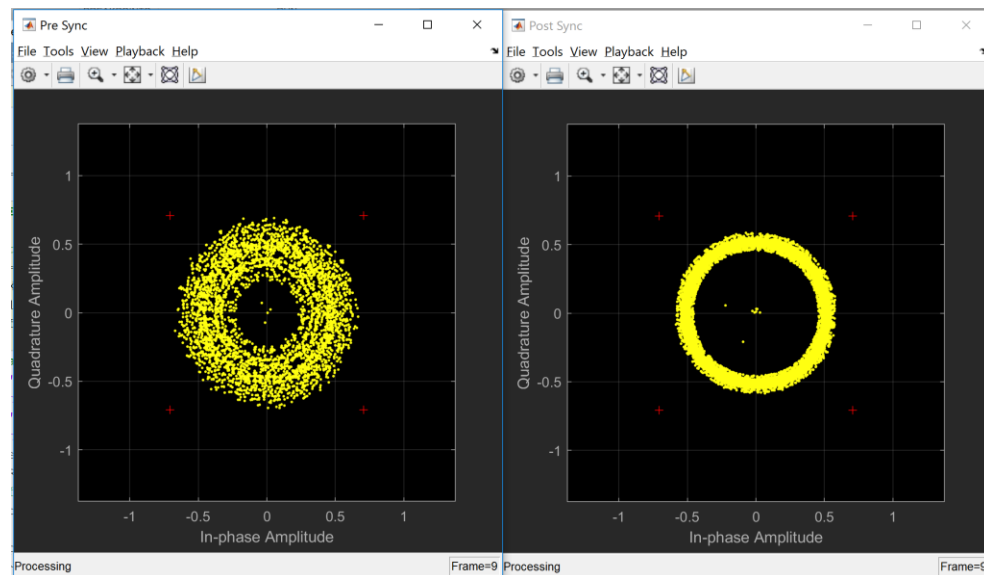


Figure 8 Constellation of QPSK recovery signal before and after carrier and timing synchronization with remaining offset

**It is now your task to update the `comm.CarrierSynchronization System` object's parameters to produce a stable (non-rotating) constellation. If you can accomplish this task, increase the offset to 5kHz and repeat this process if you are feeling adventurous.**