mobile communications series

SOFTWARE-DEFINED RADIO for ENGINEERS

TRAVIS F. COLLINS
ROBIN GETZ
DI PU
ALEXANDER M. WYGLINSKI

Software-Defined Radio for Engineers

For a listing of recent titles in the *Artech House Mobile Communications*, turn to the back of this book.

Software-Defined Radio for Engineers

Travis F. Collins
Robin Getz
Di Pu
Alexander M. Wyglinski

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the U.S. Library of Congress.

British Library Cataloguing in Publication Data
A catalog record for this book is available from the British Library.

ISBN-13: 978-1-63081-457-1

Cover design by John Gomes

10 9 8 7 6 5 4 3 2 1

Dedication

To my wife Lauren
—Travis Collins

To my wonderful children, Matthew, Lauren, and Isaac, and my patient wife, Michelle—sorry I have been hiding in the basement working on this book. To all my fantastic colleagues at Analog Devices: Dave, Michael, Lars-Peter, Andrei, Mihai, Travis, Wyatt and many more, without whom Pluto SDR and IIO would not exist.
—Robin Getz

To my lovely son Aidi, my husband Di, and my parents Lingzhen and Xuexun
—Di Pu

To my wife Jen
—Alexander Wyglinski

Contents

CHAPTER 7

Carrier Synchronization

This chapter will introduce the concept of carrier frequency offset between transmitting and receiving nodes. Specifically, a simplified error model will be discussed along with two recovery methods that can operate jointly or independently based on their implementation. Carrier recovery complements timing recovery, which was implemented in the previous Chapter 6, and is necessary for maintaining wireless links between radios with independent oscillators.

Throughout this chapter we will assume that timing mismatches between the transmitting and receiving radios have already been corrected. However, this is not a requirement in all cases, specifically in the initial implementation provided here, but will become a necessary condition for optimal performance of the final implementation provided. For the sake of simplicity we will also ignore timing effects in our simulations except when discussing Pluto SDR itself, since obviously timing correction cannot be overlooked in that case. With regard to our full receiver diagram outline in Figure 7.1, we are now considering the carrier recovery and CFO blocks.

7.1 Carrier Offsets

The receiving and transmitting nodes are generally two distinct and spatially separate units. Therefore, relative frequency offsets will exist between their LOs due to natural effects such as impurities, electrical noise, and temperature differences, among others. Since these differences can also be relatively dynamic the LOs will drift with respect to one another. These offsets can contain random phase noise, frequency offset, frequency drift, and initial phase mismatches. However, for simplicity we will only model this offset as a fixed value. This is a reasonable assumption at the time scale of RF communications.

When considering commercial oscillators, the frequency offset is provided in parts per million (PPM), which we can translate into a maximum carrier offset for a given frequency. In the case of the Pluto SDR the internal LO is rated at 25 PPM [1] (2 PPM when calibrated) and we can use (7.1) to relate maximum carrier offset Δf to our operating carrier frequency f_c.

$$f_{o,max} = \frac{f_c \times PPM}{10^6} \tag{7.1}$$

The determination of $f_{o,max}$ is important because it provides our carrier recovery design criteria. There is no point wasting resources on a capability to correct for a frequencies beyond our operational range. However, scanning techniques can be used in such cases but are beyond the scope of this book.

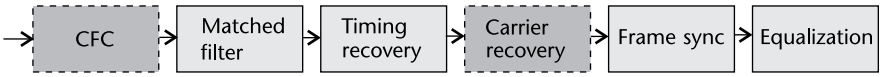

Figure 7.1 Receiver block diagram.

Mathematically we can model a corrupted source signal at baseband $s(k)$ with a carrier frequency offset of f_o (or ω_o) as

$$r(k) = s(k)e^{j(2\pi f_o kT+\theta)} + n(k) = s(k)e^{j(\omega_o kT+\theta)} + n(k). \tag{7.2}$$

where $n(k)$ is a zero-mean Gaussian random process, T is the symbol period, θ is the carrier phase, and ω_o the angular frequency.

In the literature, carrier recovery is sometimes defined as carrier phase recovery or carrier frequency recovery. These generally all have the same goal of providing a stable constellation at the output of the synchronizer. However, it is important to understand the relation of frequency and phase, which will make these naming conventions clear. An angular frequency ω, or equivalently in frequency $2\pi f$, is purely a measure of a changing phase θ over time:

$$\omega = \frac{d\theta}{dt} = 2\pi f. \tag{7.3}$$

Hence, recovering the phase of the signal is essentially recovering that signal's frequency. Through this relation is the common method for estimating frequency of a signal since it cannot be measured directly unlike phase. We can demonstrate this technique with a simple MATLAB script shown in Code 7.1. There we generate a simple continuous wave (CW) tone at a given frequency, measure the instantaneous phase of the signal, and then take the difference of those measurements as our frequency estimate. The instantaneous phase θ of any complex signal $x(k)$ can be measured as

$$\theta = tan^{-1}\left(\frac{Im(x(k))}{Re(x(k))}\right), \tag{7.4}$$

where *Re* and *Im* capture the real and imaginary components of the signal respectively. In Code 7.1 we also provide a complex sinusoid generation through a Hilbert transform with the function `hilbert` from our real signal. Hilbert transforms are very useful for generating analytic or complex representations of real signals. If you wish to learn more about Hilbert transforms, Oppenheim [2] is a suggested reading based in signal processing theory.

In Figure 7.2 we provide the outputs from Code 7.1. In Figure 7.2(a) it first can be observed that the Hilbert transform's output is equal to the CW tone generated from our sine (imag) and cosine (real) signal. In Figure 7.2(b) we can clearly see that the estimation technique based on phase difference correctly estimates the frequency of the signal in question. In this script we also utilized the function `unwrap` to prevent our phase estimates from becoming bounded between $\pm\pi$. This estimation is a straightforward application of the relation from (7.3). Alternatively, it can be useful to examine a frequency offset, but usually only large offsets, in the frequency domain itself. This is useful since time domain signals alone, especially when containing modulated data and noise, can be difficult to interpret for such an offset. In Figure 7.3, PSDs of an original and offset signal are shown, which

Code 7.1 `freqEstimate.m`

```
% Sinusoid parameters
fs = 1e3; fc = 30; N = 1e3;
t = 0:1/fs:(N-1)/fs;
% Create CW Tone
r = cos(2*pi*fc*t); i = sin(2*pi*fc*t);
% Alternatively we can use a hilbert transform from our real signal
y = hilbert(r);
% Estimate frequency from phase
phaseEstHib = unwrap(angle(y))*fs/(2*pi); freqEstHib = diff(phaseEstHib);
phaseEstCW = unwrap(atan2(i,r))*fs/(2*pi); freqEstCW = diff(phaseEstCW);
tDiff = t(1:end-1);
```

From the MATLAB Code 7.1 examine the frequency range of this estimation technique with respect to the sampling rate f_s and the frequency of the tone f_c. (Ignore the output of the Hilbert transform for this exercise.) What is roughly the maximum frequency that can be correctly estimated and what happens when the frequency offset exceeds this point?

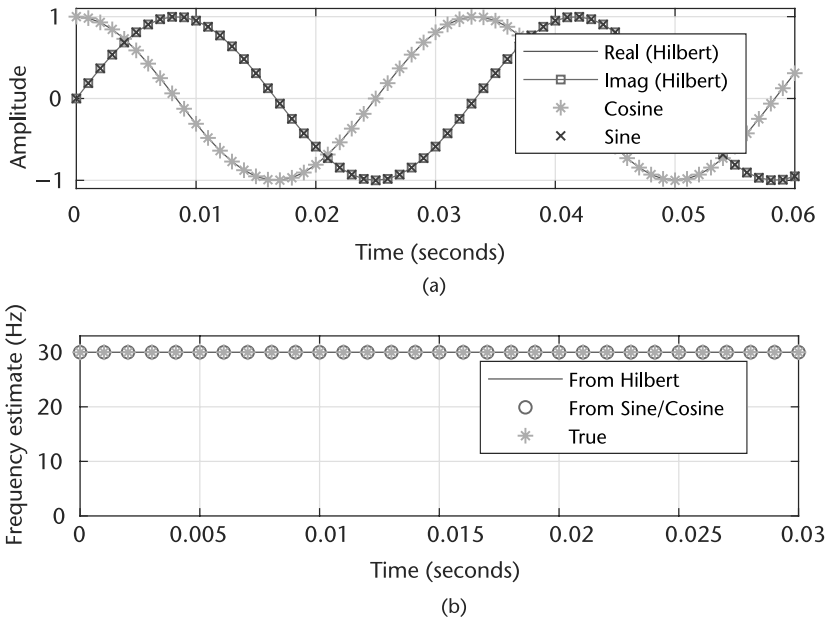

Figure 7.2 Outputs of MATLAB scripts for a simple frequency estimation technique compared with the true offset. (a) CW tones generated from sine/cosine and Hilbert transform, and (b) frequency estimates of CW tones.

clearly demonstrates this perspective. Here the signal maintains a 10-kHz offset with respect to the original signal, which is well within the 25-PPM specification of communicating Pluto SDR above 200 MHz.

Moving complex signals in frequency is a simple application of (7.2), which was how Figure 7.3(b) was generated. The example MATLAB script in Code 7.2

demonstrated how to shift a complex signal using an exponential function. Alternatively, sinusoids can be used directly if desired. In the script provided it is important to upsample or oversample the signal first, as performed by the SRRC filter in Code 7.2. This makes the frequency shift obvious since the main signal energy is limited to a fraction of the bandwidth.

Code 7.2 `freqShiftFFT.m`

```
% General system details
fs = 1e6; samplesPerSymbol = 1; frameSize = 2^8;
modulationOrder = 2; filterOversample = 4; filterSymbolSpan = 8;
% Impairments
frequencyOffsetHz = 1e5;
% Generate symbols
data = randi([0 samplesPerSymbol], frameSize, 1);
mod = comm.BPSKModulator(); modulatedData = mod(data);
% Add TX Filter
TxFlt = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',...
    filterOversample, 'FilterSpanInSymbols', filterSymbolSpan);
filteredData = TxFlt(modulatedData);
% Shift signal in frequency
t = 0:1/fs:(frameSize*filterOversample-1)/fs;
freqShift = exp(1i.*2*pi*frequencyOffsetHz*t.');
offsetData = filteredData.*freqShift;
```

7.2 Frequency Offset Compensation

There are many different ways to design a wireless receiver, using many different recovery techniques and arrangement of algorithms. In this section we will consider

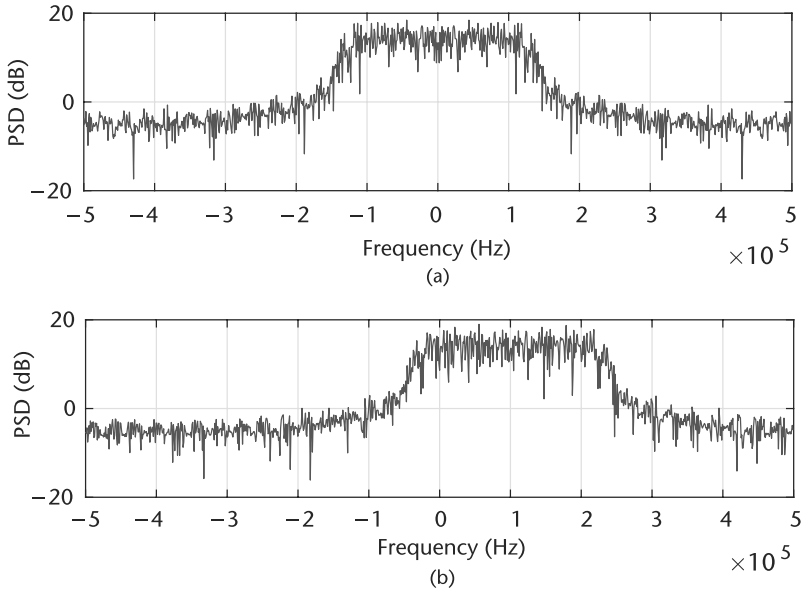

Figure 7.3 Comparison of frequency domain signals with and without frequency offsets. (a) PSD of BPSK signal without frequency offset, and (b) PSD of BPSK signal with 10-kHz offset.

Change `filterOversample` in Code 7.2 above and observe the spectrum. Explain what you observe. Next with the original script increase the frequency offset in units of $0.1F_s$, where F_s is the sample rate, from $0.1F_s$ to $1.0F_s$. Explain the observed effect.

frequency offset first and then proceed to manage the remaining synchronization tasks. As discussed in Section 10.3, the oscillator of Pluto SDR is rated at 25 PPM. Transmitting signals in an unlicensed band, such as 2.4 GHz, can produce a maximum offset of 120 kHz between the radios. Since this is quite a large range we will develop a two-stage frequency compensation technique separated into coarse and fine frequency correction. This design is favorable, since it can reduce convergence or locking time for estimation of the relative carrier.

7.2.1 Coarse Frequency Correction

There are two primary categories of coarse frequency correction in the literature: data-aided (DA) and blind correction. DA techniques utilize correlation type structures that use knowledge of the received signal, usually in the form of a preamble, to estimate the carrier offset f_o. Although DA methods can provide accurate estimates, their performance is generally limited by the length of the preambles [3], and as the preamble length is increased this decreases system throughput.

Alternatively, blind or nondata-aided (NDA) methods can operate over the entire duration of the signal. Therefore, it can be argued in a realistic system NDA can outperform DA algorithms. These coarse techniques are typically implemented in an open-loop methodology, for ease of use. Here we will both outline and implement a NDA FFT-based technique for coarse compensation. The concept applied here is straightforward, and based on our initial inspection provided in Figure 7.3, we can provide a rough estimate on the symbols offsets. However, directly taking the peak from the FFT will not be very accurate, especially if the signal is not symmetrical in frequency. To compensate for this fact, we will remove the modulation components of the signal itself by raising the signal to its modulation order M. From our model in (7.2), ignoring noise, we can observe the following:

$$r^M(k) = s^M(k)e^{j(2\pi f_o kT+\theta)M}. \tag{7.5}$$

This will shift the offset to M times its original location and make $s(t)$ purely real or purely complex. Therefore, the $s^M(t)$ term can be ignored and only the remaining exponential or tone will remain. To estimate the position of this tone we will take the FFT of $r^M(t)$ and relate the bin with the most energy to the location of this tone. Figure 7.4 is an example frequency plot of $r^M(t)$ for a BPSK signal generated from the MATLAB Code 7.2 offset by 10 kHz. The peak is clearly visible at twice this frequency as expected. Formally this frequency estimation can be written in a single equation as [4]

$$\hat{f}_o = \frac{1}{2TK}\arg\Big|\sum_{k=0}^{K-1} r^M(k)e^{-j2\pi kT/K}\Big| \tag{7.6}$$

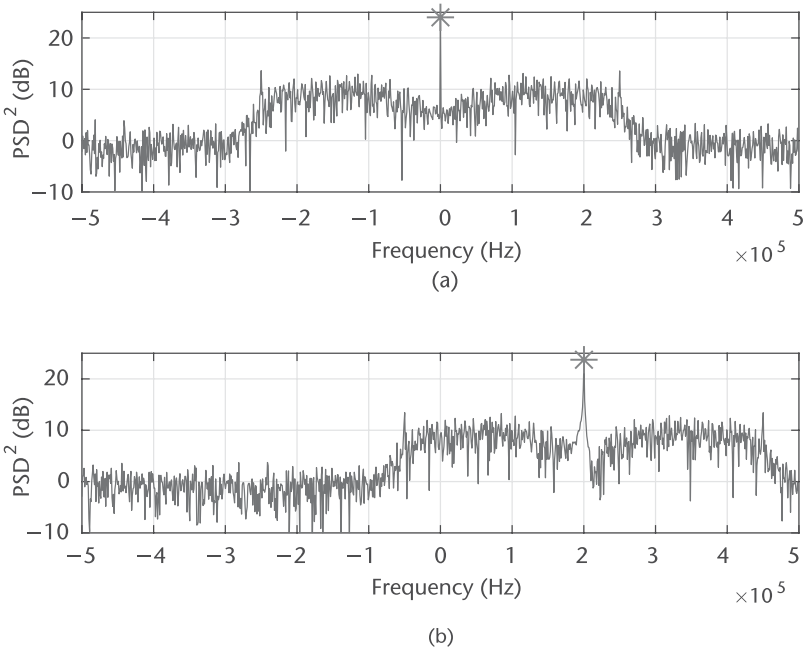

Figure 7.4 Comparison of frequency domain signals with and without frequency offsets. (a) *PSD* of squared BPSK signal without frequency offset, and (b) *PSD* of squared BPSK signal with 10-kHz offset.

where K is the *FFT* length. The estimation in (7.6) is defined as coarse since the resulting $\hat{f}_o$ can only be one of K values produced by the FFT. However, we can extend this accuracy by interpolating across a fixed set of FFT bins over multiple estimates if desired. The frequency resolution of each FFT bin for the signal is simply

$$f_r = \frac{1}{MTK}. \tag{7.7}$$

Therefore, we can increase the performance of our estimator by increasing the FFT size or by decreasing the sample rate of the system. However, do not reduce the sample below the bandwidth of your signal of interest.

What is the limitation of this method? (What happens when the M becomes larger?) Finally, add AWGN to the receive signal at different SNR value and examine when the peak become difficult to determine. Provide a plot of peak estimated MSE versus SNR for this analysis.

Implementing this method in MATLAB is straightforward and for efficiency K should alway be the base two number for efficiency of the FFT. In Code 7.3 we produce an estimate for each K samples of data, and compensate for the arrangement of frequencies from the `fft` function. When using this technique we should also consider other aspects of the system or impacts this operation can have. From the perspective of downstream algorithms, they will observe a frequency

Code 7.3 `fftFreqEst.m`

```
%% Estimation of error
fftOrder = 2^10; k = 1;
frequencyRange = linspace(-sampleRateHz/2,sampleRateHz/2,fftOrder);
% Precalculate constants
offsetEstimates = zeros(floor(length(noisyData)/fftOrder),1);
indexToHz = sampleRateHz/(modulationOrder*fftOrder);
for est=1:length(offsetEstimates)
    % Increment indexes
    timeIndex = (k:k+fftOrder-1).';
    k = k + fftOrder;
    % Remove modulation effects
    sigNoMod = offsetData(timeIndex).^modulationOrder;
    % Take FFT and ABS
    freqHist = abs(fft(sigNoMod));
    % Determine most likely offset
    [~,maxInd] = max(freqHist);
    offsetInd = maxInd - 1;
    if maxInd>=fftOrder/2 % Compensate for spectrum shift
       offsetInd = offsetInd - fftOrder;
    end
    % Convert to Hz from normalized frequency index
    offsetEstimates(est) = offsetInd * indexToHz;
end
```

correction every K samples. Ideally $\hat{f}_o$ remains constant, but this is unlikely if the offset is close to an FFT bin boundary. Resulting is frequency jumps in the signal $\pm f_r$ from previous signals. Unfortunately these abrupt changes can disrupt feedback algorithm downstream, which are ill-equipped to deal to sudden shift in frequency or phase of the signal they are estimating/correcting. To combat this we have two main strategies. First, the estimates can be averaged over time with a filter, smoothing out the changes over time. The second option would be to only apply this correction at the start of a frame. Since the offset should be relatively stationary across a reasonably sized frame, a single measurement should be accurate over that duration of time. This correction is also considered coarse since it can only be accurate to within f_r, which only enforces this type of correction interval.

With that said a weakness of this FFT-based technique is that it requires a significant amount of data for a reasonable estimate. This technique will also produce unpure tones when oversampled at the transmitter with transmit filters. However, other techniques such as from Luise [5] are designed for burst-type applications where less data relative to the FFT method above is required. Unfortunately, the Luise method is a biased estimator unlike the FFT method.

7.2.2 Fine Frequency Correction

After coarse frequency correction (CFC) there will still be offset based on the configured resolution chosen f_r. Fine frequency correction (FFC), also called carrier phase correction, should produce a stable constellation for eventual demodulation. Essentially this will drive the remaining frequency offset of the received signal to zero. We can describe this correction as producing a stable constellation due to how fine frequency offset effects are typically examined with a constellation diagram. If a

Using loopback with Pluto SDR and Code 7.3, measure the frequency estimate's mean squared error as a function of the difference between the center frequencies ($f_{o,max}$) of the transmitter and receiver. Use BSPK signal here and examine f_Δ from $0 - 100$ kHz at 1-MHz baseband sampling rate.
Repeat this, but fix the transmitter to a gain of -30 and take estimates with the receiver in manual gain mode at 10, 30, and 50.

discrete digitally modulated signal exhibits frequency offset, this will cause rotation over time as examined in a constellation diagram. In Figure 7.5 we demonstrate this effect where each number relates a sample's relative occurrence in time, which provides this perspective of rotation. The signal itself is BPSK, causing it to jump across the origin with different source symbols. If a positive frequency offset is applied the rotation will be counterclockwise and clockwise with a negative offset. The rate of the rotation is equal to the frequency offset, which is where our notion of ω (angular frequency) comes from, as previously defined in (7.3).

This offset can also be observed with Pluto SDR in a similar way. In Figure 7.6 we transmitted a BPSK signal in loopback with 1-kHz difference between transmit and receive LOs. We observe a similar rotation as in Figure 7.5 in Figure 7.6(b). In order to correctly visualize this effect we needed to perform timing correction, which was borrowed from Chapter 6. Without timing correction the signal is difficult to interpret from the constellation plot as observed in Figure 7.6(a). Since timing correction was performed in this case before the frequency was corrected, this required use of the Gardner technique as detailed in Section 6.4.1. Unlike CFC,

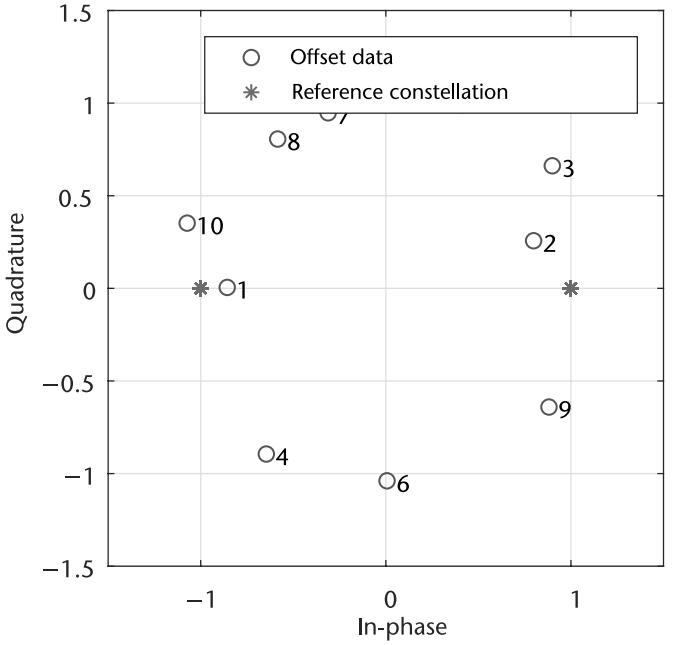

Figure 7.5 Rotating constellation of BPSK source signal with frequency offset.

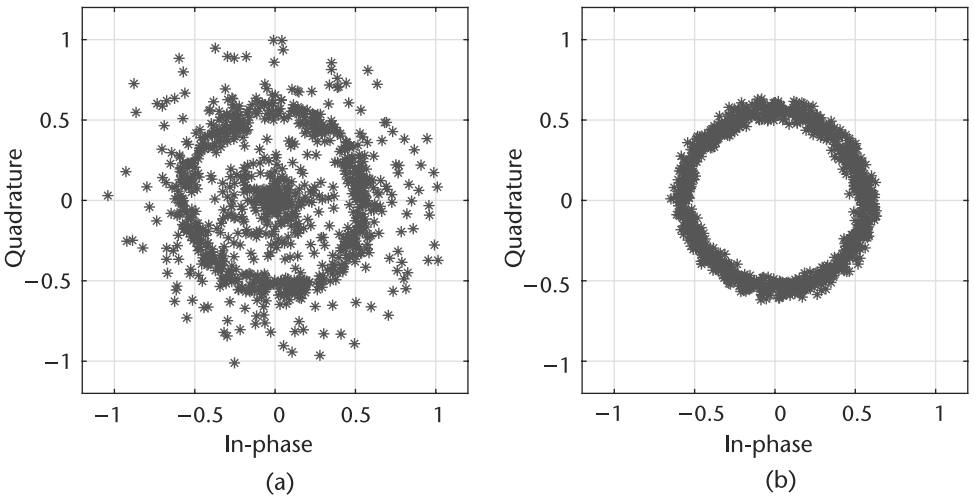

Figure 7.6 BPSK signal transmit through Pluto SDR in loopback with 1-kHz offset at 1 MHz. (a) BPSK signal before timing correction, and (b) BPSK signal after timing correction.

which uses a feedforward technique, for FFC we will utilize a feedback or closed-loop method based PLL theory as examined in Chapter 4. The structure of this algorithm is provided in Figure 6.11 derived from [6, Chapter 7], which relates back our original outline in Figure 6.11.

This all-digital PLL-based algorithm works by first measuring the phase offset of a received sample in the phase error detector (PED), which we call the error signal $e(n)$. The PED is designed based on the structure of the desired receive constellation/symbols. Next, the loop filter helps govern the dynamics of the overall PLL. The loop filter can determine operational frequency (sometimes called pull-in range), lock time, and responsiveness of the PLL, as well as smoothing out the error signal. Finally, we have the direct digital synthesizer (DDS), whose name is a remnant of analog PLL designs with voltage-controlled oscillators (VCOs). The DDS is responsible for generation of the correction signal for the input, which again will be fed back into the system. In the case of the FFC design, this PLL should eventually produce an output signal with desired offset equal to zero.

Starting with the PED, the goal of this block is simply to measure the phase or radial offset of the input complex data from a desired reference constellation. By reference and by extension $e(n)$, we are actually referring to the distance from the constellation bases. In the case of QAM, PSK, and PAM these will always be the real and imaginary axes. However, you may need to extend this perspective with regard to FSK or other modulation schemes. The primary reasoning behind this idea is that it will remove the scaling aspect in a specific dimension, and instead consider the ratio of energy or amplitude in a specific basis. To better understand this concept let us consider QPSK, which has the following PED equation:

$$e(n) = sign(Re(y(n))) \times Im(y(n)) - sign(Im(y(n))) \times Re(y(n)). \tag{7.8}$$

In (7.8) $e(n)$ is essentially measuring the difference between the real and imaginary portions of $y(n)$, and will only be zero when $Re(y(n)) = Im(y(n))$. You will notice that this will force the output constellation only to a specific orientation,

but not a specific norm value. However, if $y(n)$ requires a different orientation this can be accomplished after passing through the synchronizer with a simple multiply with the desired phase shift of ϕ_{POST} as

$$y_{SHIFT}(n) = y(n)e^{j*\phi_{POST}}. \tag{7.9}$$

Note that $y_{SHIFT}(n)$ should not be fed into the PED.

In the case of other modulation schemes the PED error estimation will change based on the desired signal arrangement. BPSK or PAM for example will have the following error estimation:

$$e(n) = sign(Re(y(n))) \times Im(y(n)). \tag{7.10}$$

This PED error function again has the same goal of providing the error signal only for the orientation of $y(n)$. For (7.10) $e(n)$ will only be zero when $y(n)$ is purely real.

The reasoning behind (7.8) and (7.10) is straightforward. On the other hand, the loop filter in all PLL designs is the most challenging aspect, but it provides the most control over the adaptation of the system. Again here we will use a PI filter as our loop filter, which was detailed in Section 6.3.1. The last piece to this FFC synchronizer is the DDS, which is just an integrator. Since the loop filter produces a control signal, which is equivalent to the frequency of the input signal, it becomes necessary to extract the phase of this signal instead. The transfer functions used for the integrator here are

$$D(s) = G_3\frac{1}{s} \quad \rightarrow \quad D(z) = G_3\frac{z^{-1}}{1-z^{-1}}. \tag{7.11}$$

Note that we have added an additional delay of a single sample in the discrete domain, and since we are producing a correction signal $G_3 = -1$. Again this integrator can be implemented with a biquad filter.

In this arrangement of the PLL shown in Figure 7.7, the system should produce an output $y(n)$, which has minimal phase and frequency offsets. Going around the loop again in Figure 7.7, the PED will first produce an error equal to the phase offset associated with the observed corrected[1] symbol $y(n)$, then the loop filter will relate this observed error and weight it against all previous errors. Finally, the DDS will convert the weighted error/control signal $f(n)$ to a phase $\phi(n)$, which we use to correct the next input sample $x(n+1)$. In the case of frequency offsets, ϕ will continuously change since is it a phase value, not a frequency value. However, if the input signal is too dynamic or the gains of the filters are not set appropriately, the PLL will not be able to keep up with the changing phase (frequency) of x.

For the calculation of the gain values (G_1, G_2) of the loop filter, utilize the following equations based on a preferred damping factor ζ and loop bandwidth B_{Loop}:

$$\theta = \frac{B_{Loop}}{M(\zeta + 0.25/\zeta)} \qquad \Delta = 1 + 2\zeta\theta + \theta^2 \tag{7.12}$$

1. We define this as a corrected symbol since it has passed through the rotator and we will not apply additional phase shifts to this sample. This is also the output of the PLL.

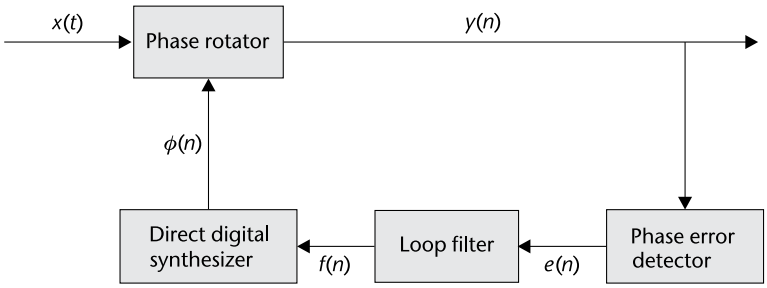

Figure 7.7 FFC structure based on PLL design for feedback corrections.

$$G_1 = \frac{4\zeta\theta/\Delta}{MK} \qquad G_2 = \frac{(4/M)\theta^2/\Delta}{MK} \tag{7.13}$$

where M is the number of sample per symbol and K is the detector gain. For QPSK and rectangular QAM $K = 2$, but for PAM and PSK $K = 1$. Note that B_{Loop} is a normalized frequency. If you are interested in how these are derived, consult [6, Appendix C] for a full detailed analysis. For the selection of ζ refer back to Section 6.3.1, which has the same definition here. The selection of B_{Loop} should be related to the maximum estimated normalized frequency locking range $\Delta_{f,lock}$ range desired:

$$\Delta_{f,pull} \sim 2\pi\sqrt{2}\zeta B_{Loop}. \tag{7.14}$$

Note that this value is an estimate based off a linearized model of the PLL. Therefore inconsistencies may exist in the simulated versions. However, this PLL design should perform well even under strong noise conditions when configured correctly. Unlike the CFC correction this FFC will generally not have the same operational range. In your designs, it may be useful to start with a damping factor of $\zeta = 1$ and a loop bandwidth of $B_{Loop} = 0.01$. From experience, using an overdamped system here is preferable since it directly increases the pull-in range. However, it will take the loop longer to converge.

Starting from Code 7.4 implement a carrier recovery algorithm for BPSK. Tune this implementation for a normalized frequency offset of 0.001 and 0.004. Evaluate these implementations over a range of SNR for the MSE of their frequency estimates.

We now have all the necessary pieces to implement the FFC synchronizer, for which we provide a full reference implementation in Code 7.4. However, it is important to discuss some of the design considerations. First, we have stated the output of the FFC synchronizer can have a target of a specific orientation of the output constellation, which is solely determined by the PED. However, the synchronizer may not always be able to achieve this target constellation orientation, meaning the constellation may appear slightly rotated or appear at multiples of the expected position. This will result from signals with larger carrier offsets than the FFC was configured to handle or most notably when the system is configured in an underdamped way. Alternatively, if the received signal has poor

SNR this will also degrade the effective pull-in-range of the synchronize of cause a nondesirable lock position. This is illustrated in Code 7.4, where two different ζ configurations are used. The system is also driven close to the estimated maximum offset for the configuration. In general these estimates will be conservative and will require empirical testing for a specific modulation scheme, SNR, and loop filter configuration. However, in this case if we examine the converged signals in Figure 7.8 we notice an interesting set of outcomes. In Figure 7.8(b) the constellation actually converges to a false minimum. This is a result of the dynamics of the PLL, which is in an underdamped state. Forcing the system to be more rigid will provide the correct result as in Figure 7.8(a). However, if ζ is too large the synchronize will not converge or can take a very long time to do so.

Introduce timing offset into the model for Code 7.4. For the recovery process take your implementation from Chapter 4 for timing recovery and place this into the system. Evaluate these implementations over a range of SNR for the MSE of their frequency estimates.

When implementing and testing your own system it can be useful to actually measure the frequency estimation of the synchronizer itself. Since we know that the output of the DDS ϕ is the instantaneous phase correction needed for the next symbol, we can simply apply (7.3) with similar computations as in Code 7.1. From the angular frequency estimates we can translate this to a more tangible frequency estimate in hertz as in (7.3). From inspecting the derivative of `Phase` (ϕ) for Code 7.4 we can examine the convergence of the estimate for an offset of 20 Hz with $f_s = 1000$ Hz. In Figure 7.9 we plot f_{est} where there is an obvious convergence around the correct value. However, since the signal contains noise and there is inherent noise to the PLL, the estimate will not be static. This is useful in a real system since the offsets between transmitter and receiver LOs will always be dynamic with respect to one another.

7.2.3 Performance Analysis

To evaluate the synchronization performance a number of variables can be considered. These include but are not limited to lock time, effective pull-in range, and converged error vector magnitude (EVM). These metrics should be balanced in a way that meets the needs for a specific design since they will clash with one another. For example, it can be simple to design a system with a fast lock time, but it will probably have limited pull-in range. This is a direct relation to (7.14) and a secondary measurement from [6, Appendix C], which defines the normalized frequency lock delay:

$$t_{\Delta,Max} \sim \frac{32\zeta^2}{B_{Loop}}. \tag{7.15}$$

We can demonstrate this trade-off between ζ and B_{Loop} if we focus on the error signal directly from the PED. Modifying the code from 7.4 by fixing the normalized carrier offset to 0.01, $\zeta = 1.3$, and selecting two different values for B_{Loop} we

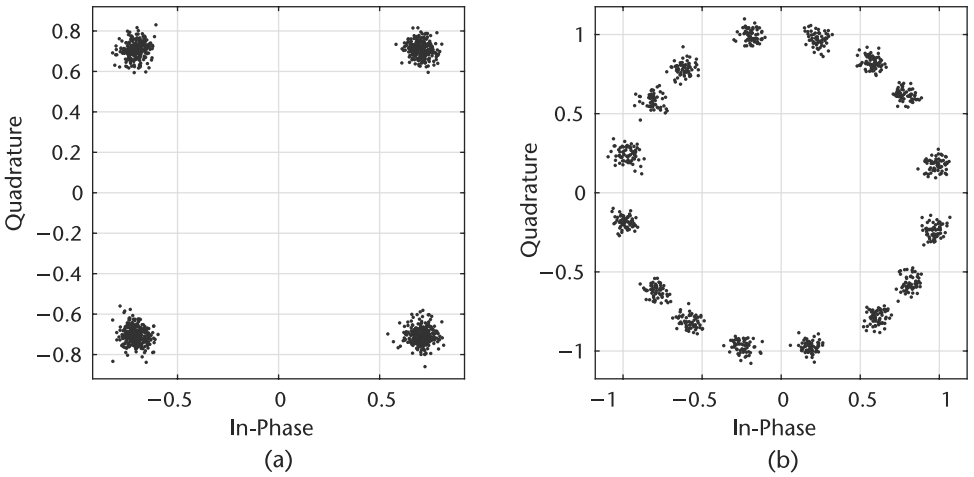

Figure 7.8 Converged QPSK signals after carrier recovery with different damping factors, both (a) overdamped ($\zeta = 1.3$), and (b) underdamped ($\zeta = 0.9$).

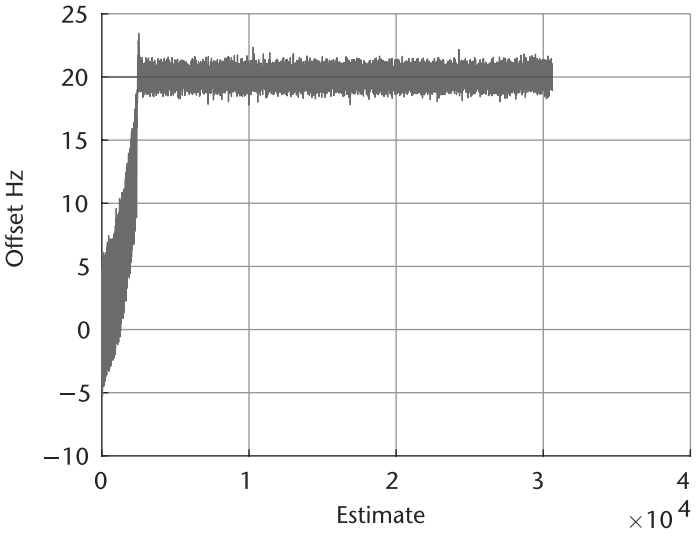

Figure 7.9 Estimations over time and eventual convergence of implemented FFC for 20-Hz offset.

Based off your existing simulation solutions that recover signals with both timing and carrier offset, introduce Pluto SDR as the channel mechanism. It may be useful to start with Code 6.1 and 7.4. Evaluate your implementation in loopback with increasing frequency difference between transmit and receive LOs.

can observe $e(n)$ in Figure 7.10. In both configurations of B_{Loop} the normalized offset is less than $\Delta_{f,pull}$. In the case for $B_{Loop} = 0.24$, the system converges to a solution within a few tens of samples, while the $B_{Loop} = 0.03$ case is an order of magnitude slower. However, the variance of the converged error signal σ_e^2 is three times smaller for the case when $B_{Loop} = 0.03$. This error will appear as phase noise on $y(n)$, which will affect the demodulation correctness of the signal.

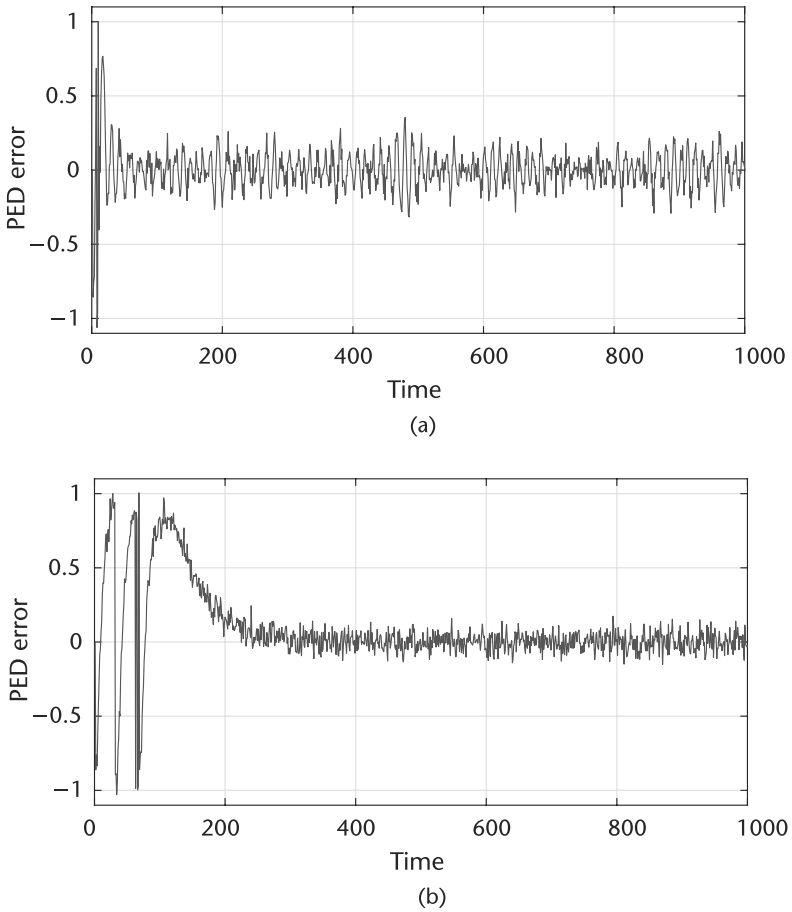

Figure 7.10 Error signal from QPSK PED for different loop bandwidth for time. (a) $B_{Loop} = 0.24$ with $\sigma_e^2 = 0.0103$ after convergence, and (b) $B_{Loop} = 0.03$ with $\sigma_e^2 = 0.0031$ after convergence.

7.2.4 Error Vector Magnitude Measurements

Evaluating the EVM for $y(n)$ will provide a measure of this phase noise in the recovered signal. EVM is a very useful measurement to understand the algorithmic performance in the system. EVM measures the residual error of the constellation with respect to a reference position. To calculate EVM in percent RMS we can use the following equation:

$$EVM_{RMS} = 100 \times \sqrt{\sum_{k=0}^{N-1} e_{const}(k) \sum_{k=0}^{N-1} (Re(\bar{y}(k))^2 + Im(\bar{y}(k))^2)}, \tag{7.16}$$

where

$$e_{const}(k) = (Re(y(k)) - Re(\bar{y}(k)))^2 + (Im(y(k)) - Im(\bar{y}(k)))^2 \tag{7.17}$$

and $\bar{y}(k)$ is the reference symbol for $y(k)$. EVM is a measure on the dispersiveness of the received signal. Therefore, the lower the EVM values the better. In some situations it can be useful to calculate EVM in decibels, which can be converted

Code 7.4 badlock.m

```
%% General system details
sampleRateHz = 1e6; samplesPerSymbol = 1; frameSize = 2^10;
numFrames = 10; nSamples = numFrames*frameSize;
DampingFactors = [0.9,1.3]; NormalizedLoopBandwidth = 0.09;
%% Generate symbols
order = 4; data = pskmod(randi([0 order-1], nSamples, 1),order,0); % QPSK
%% Configure LF and PI
LoopFilter = dsp.IIRFilter('Structure', 'Direct form II transposed', ...
    'Numerator', [1 0], 'Denominator', [1 -1]);
Integrator = dsp.IIRFilter('Structure', 'Direct form II transposed', ...
    'Numerator', [0 1], 'Denominator', [1 -1]);
for DampingFactor = DampingFactors
    %% Calculate range estimates
    NormalizedPullInRange = min(1, 2*pi*sqrt(2)*DampingFactor*...
        NormalizedLoopBandwidth);
    MaxFrequencyLockDelay = (4*NormalizedPullInRange^2)/...
        (NormalizedLoopBandwidth)^3;
    MaxPhaseLockDelay = 1.3/(NormalizedLoopBandwidth);
    %% Impairments
    frequencyOffsetHz = sampleRateHz*(NormalizedPullInRange);
    snr = 25; noisyData = awgn(data,snr);% Add noise
    % Add frequency offset to baseband signal
    freqShift=exp(1i.*2*pi*frequencyOffsetHz./sampleRateHz*(1:nSamples)).';
    offsetData = noisyData.*freqShift;
    %% Calculate coefficients for FFC
    PhaseRecoveryLoopBandwidth = NormalizedLoopBandwidth*samplesPerSymbol;
    PhaseRecoveryGain = samplesPerSymbol;
    PhaseErrorDetectorGain = log2(order); DigitalSynthesizerGain = -1;
    theta = PhaseRecoveryLoopBandwidth/...
        ((DampingFactor + 0.25/DampingFactor)*samplesPerSymbol);
    delta = 1 + 2*DampingFactor*theta + theta*theta;
    % G1
    ProportionalGain = (4*DampingFactor*theta/delta)/...
        (PhaseErrorDetectorGain*PhaseRecoveryGain);
    % G3
    IntegratorGain = (4/samplesPerSymbol*theta*theta/delta)/...
        (PhaseErrorDetectorGain*PhaseRecoveryGain);
    %% Correct carrier offset
    output = zeros(size(offsetData));
    Phase = 0; previousSample = complex(0);
    LoopFilter.release();Integrator.release();
    for k = 1:length(offsetData)-1
        % Complex phase shift
        output(k) = offsetData(k+1)*exp(1i*Phase);
        % PED
        phErr = sign(real(previousSample)).*imag(previousSample)...
            - sign(imag(previousSample)).*real(previousSample);
        % Loop Filter
        loopFiltOut = step(LoopFilter,phErr*IntegratorGain);
        % Direct Digital Synthesizer
        DDSOut = step(Integrator,phErr*ProportionalGain + loopFiltOut);
        Phase =  DigitalSynthesizerGain * DDSOut;
        previousSample = output(k);
    end
    scatterplot(output(end-1024:end-10));title('');
end
```

from (7.16) as

$$EVM_{dB} = 20\,log_{10}\left(\frac{EVM_{RMS}}{100}\right). \tag{7.18}$$

Calculating EVM in decibels is very common in OFDM standards due to high-order constellations that can be transmitting, which require a significant EVM margin to recover. For convience, the Communications System Toolbox include a system object called `comm.EVM` to provide these calculations for us.

Starting with Code 7.4, evaluate the EVM of converged signals with regard to ζ and B_{Loop}. Select values of ζ in underdamped, overdamped, and critically damped configurations.

7.3 Phase Ambiguity

The last topic to consider for carrier synchronization is phase ambiguity. Phase ambiguity arises from the fact that the FFC synchronizer outlined here is blind to the true orientation of the transmitted signal. For a given symmetrical modulation scheme there can be a number of convergent orientations, which can be related to the modulation order. For example, PAM will have two possible orientations, QPSK and rectangular QAM will have four, while MPSK with have M possible orientations. However, there are a number of solutions to compensate for this problem, which includes code words, use of an equalizer with training data, and differential encoding. There are different use cases for each implementation.

7.3.1 Code Words

The use of code words is a common practice for resolution of phase ambiguity, which relies on a known sequence in the received data. This is typically just the preamble itself, which will exist in each frame and is known at the receiver. This strategy can be used before or after demodulation if desired. If performed post demodulation, the output bits must be remapped onto their true positions. This process is best explained through an example. Consider the source words w and associated QPSK symbols s:

$$w = [1,\ 0,\ 3] \qquad s = [(-1, 1i)\,(1, 1i)\,(-1, -1i)]. \tag{7.19}$$

The possible received symbols would be

$$\begin{aligned} s_1 &= [(-1, 1i)\,(1, 1i)\,(-1, -1i)] \\ s_2 &= [(-1, -1i)\,(-1, 1i)\,(1, -1i)] \\ s_3 &= [(1, -1i)\,(-1, -1i)\,(1, 1i)] \\ s_4 &= [(1, 1i)\,(1, -1i)\,(-1, 1i)]. \end{aligned} \tag{7.20}$$

Demodulating each code word symbol and comparing with the expected result would provide the necessary mapping to correctly demodulate the remaining data symbols. For an implementation it would be useful to demodulate all the preamble

symbols and take the most common orientation mapping, since relying on a single symbol can be error-prone.

Alternatively, the phase offset θ_p from the correct orientation can be measured directly where p is the received preamble symbols, p_r is the reference or true preamble symbols, and the correction required is simply

$$\theta_p = tan^{-1}\left(\sum_n \frac{Im(p(n)^* \times p_r(n))}{Re(p(n)^* \times p_r(n))}\right), \tag{7.21}$$

assuming $p(n)$ has a desirable orientation. Then the remaining signal y would be corrected as

$$y_c = y\, e^{-j\theta_p}. \tag{7.22}$$

7.3.2 Differential Encoding

The second option to deal with phase ambiquity is to differentially encode the source bits themselves. The goal here is to make the true data dependent on the difference between successive bits, not on the received bits themselves. To encode the source data we apply the following at the transmitter:

$$b_t(n) = b_t(n-1) \oplus b(n), \tag{7.23}$$

where b_t are the transmitted encoded bits, b are the uncoded bits, and $\oplus$ is a modulo two addition. To decode the signal we basically perform (7.23) in reverse as

$$b(n) = b_t(n) \oplus b_t(n-1). \tag{7.24}$$

Usually in this scheme the first bit is ignored since it is only based on itself, not the difference between two consecutive bits. Engineers may point to this as wasteful, but this reduces any complex mathematics associated with measuring offsets with symbols and only requires bit-level operations. This can also reduces the bit error rate of a received signal due to propagation of bit errors.

7.3.3 Equalizers

The third popular option is to rely on an equalizer to correct this ambiguity for the system. Using training data the equalizer can learn and correct for this phase shift, which in essence is just a complex multiplication. Equalizers will be discussed in detail in Chapter 9. However, this is a small task for an equalizer implementation if channel correct or synchronization are not performed by the equalizer as well.

7.4 Chapter Summary

In this chapter we have discussed and provided a model of carrier offset and how it relates to receiver operations. From this model we have provided two schemes for compensating for carrier offset including coarse and fine recovery algorithms. However, other implementations do exist that can jointly perform timing and carrier recovery [6] if desired. We have examined how these algorithms can be used at the system level, as well as how individual performance can be evaluated. These include characterization of their parameterization as well as metric on the recovered data.

In summary, carrier offset compensation is a necessary synchronization technique when transmitting data between two disjoint nodes with independent LOs.

References

[1] Analog Devices, Inc., ADALM-PLUTO SDR Active Learning Module, http://www.analog.com/media/en/news-marketing-collateral/product-highlight/ADALM-PLUTO-Product-Highlight.pdf.

[2] Oppenheim, A.V., and R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall, 1989.

[3] Morelli, M., and U. Mengali, "Feedforward Frequency Estimation for PSK: A Tutorial Review," *European Transactions on Telecommunications*, Vol. 9, No. 2, 1998, pp. 103–116.

[4] Wang, Y., K. Shi, and E. Serpedin,"Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach," *EURASIP Journal on Advances in Signal Processing*, (2004) 2004: 856139, https://doi.org/10.1155/S1110865704403175.

[5] Luise, M. and R. Reggiannini, "Carrier Frequency Recovery in All-Digital Modems for Burst-Mode Transmissions," *IEEE Transactions on Communications*, Vol. 43, No. 2, 1995, pp. 1169–1178.

[6] Rice, M., *Digital Communications: A Discrete-Time Approach*, Third Edition, Pearson/Prentice Hall, 2009.