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 6

Timing Synchronization

In the next series of chapters we will be introducing receiver synchronization and signal recovery concepts, which are necessary for wireless communications between multiple devices. In this chapter will introduce the concept of timing recovery and approach this topic first for two primary reasons. First the downstream recovery methods used in this book for coherent modulations are sensitive to timing offset and must be compensated for first. The second reason is for prototyping with the Pluto SDR. We will be relying heavily on the loopback features of the radio, which will allow for control of nonidealities to some degree. However, since signals must travel a distance between the transmitting DAC and receiving ADC there will be a fixed but random time offset between the chains. This is where timing recovery is used to correct for this offset. With that said, a receiver can be designed in many different ways but the specific ordering of chapters here relates to the successive application of algorithms to be used: First timing recovery, then carrier phase correction, and finally frame synchronization. Once these three major pieces are handled we will then move on to more advanced topics including equalization and coding. Blocks in Figure 6.1 will be highlighted at the start of each relevant chapter to outline the progress of the overall receiver design and show how they fit with one another. In this chapter, matched filtering and timing recovery are highlighted.

In this chapter, the concept of timing recovery will be broken down into five primary sections. A brief overview of transmit filters will be first discussed, which is necessary to understand how we algorithmically perform timing recovery. Then we will move on to a simple model to demonstrate timing error, which will include Pluto SDR as well for realistic data. Finally, several recovery techniques will be discussed that adaptively handle correction of timing problems. Debugging methodology will be provided to understand how to tune these techniques for your own datasets. In regard to the algorithms discussed, an analytic analysis or derivations will not be provided explicitly. However, these algorithms will instead be treated as tools used to build a receiver component by component, where only a top-level understanding is necessary. Alternative sources will be referenced for deeper analysis, but in this work we will focus on implementations and specifically implementations with SDRs. Our goal here is to motivate the construction of a receiver initially from existing works, and readers can explore further literature if they wish to extract more performance from their implementation.

6.1 Matched Filtering

In digital communications theory when matched filtering is discussed it is typically called pulse-shaping at the transmitter and matched filtering at the receiver for

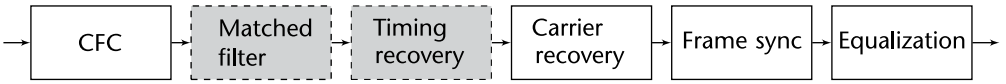

Figure 6.1 Receiver block diagram.

reference. The goal of these techniques is threefold: first, to make the signal suitable to be transmitted through the communication channel by limiting its effective bandwidth, second, increase the SNR of the received waveform, and third, to reduce intersymbol interference (ISI) from multipath channels and nonlinearities. Pulse-shaping was discussed in Section 2.7.5, but we will revisit the topic here from a slightly different approach that focuses on more practical aspects of these filters.

By filtering a symbol, sharp phase and frequency transitions are reduced resulting in a more compact and spectrally efficient signal. Figure 6.2 provides a simple example of a DBPSK signal's frequency representation before and after filtering with a transmit filter. As we can see the effective bandwidth of the signal is reduced, primarily from the upsampling/interpolation that is applied at the transmitter. Since time and frequency are inversely related we get this reduction in bandwidth. These filter stage implementations will typically upsample and downsample signals, which reduce their effective bandwidth. However, upsampling inherently increases the so-called surface area of a symbol, making it easier to determine, since we will have multiple copies of it at the receiver. Therefore, we are trading recoverability for bandwidth since data will be produced at the same rate from the transmitter but will not utilize the entire bandwidth available. These operations of rate transitions (upsampling/downsampling) are performed during the matched filtering stages since it is efficient to utilize a single filter to perform both operations.

The filter used to generate this figure was a square-root raised cosine (SRRC) filter, which is a common filter used in communication systems. We provided the raised cosine (RC) filter in Section 2.7.5, but the more common SRRC has the impulse response:

$$h(t) = \begin{cases} \dfrac{1}{\sqrt{T_s}}\left(1-\beta+4\dfrac{\beta}{\pi}\right), & t=0 \\ \dfrac{\beta}{\sqrt{2T_s}}\left[\left(1+\dfrac{2}{\pi}\right)\sin\left(\dfrac{\pi}{4\beta}\right)+\left(1-\dfrac{2}{\pi}\right)\cos\left(\dfrac{\pi}{4\beta}\right)\right], & t=\pm\dfrac{T_s}{4\beta} \\ \dfrac{1}{\sqrt{T_s}}\dfrac{\sin\left[\pi\dfrac{t}{T_s}(1-\beta)\right]+4\beta\dfrac{t}{T_s}\cos\left[\pi\dfrac{t}{T_s}(1+\beta)\right]}{\pi\dfrac{t}{T_s}\left[1-\left(4\beta\dfrac{t}{T_s}\right)^2\right]}, & \text{otherwise} \end{cases} \tag{6.1}$$

where T_s is the symbol period and $\beta \in [0,1]$ is the roll-off factor. In practice these filters will be arranged in two ways as outlined in Figure 6.3. First we can place a single RC filter at the transmitter or place a SRRC at both the transmitter and receiver. Both options produce Nyquist filter arrangements of signals to reduce or eliminate ISI. Details on how to select β and T_s will be discussed later in Section 8.2. However, we examine β through use of an eye diagram in Figure 6.4, and we can easily see the time domain effects for very different roll-offs of $\beta = [0.3, 0.99]$. For

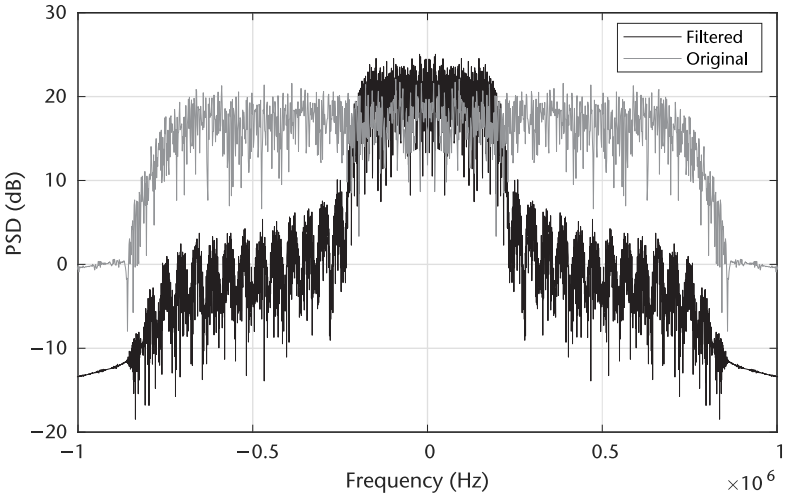

Figure 6.2 Frequency spectrum of PSK signal before and after pulse-shaping filter.

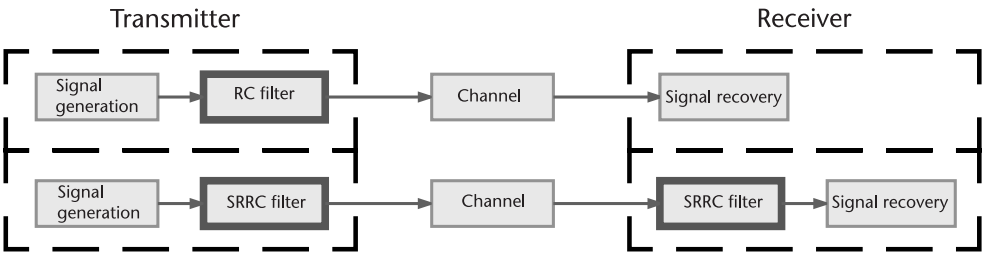

Figure 6.3 Arrangements of transmit filters with respect to the transmitter and receiver nodes for raised cosine and root-raised cosine filters.

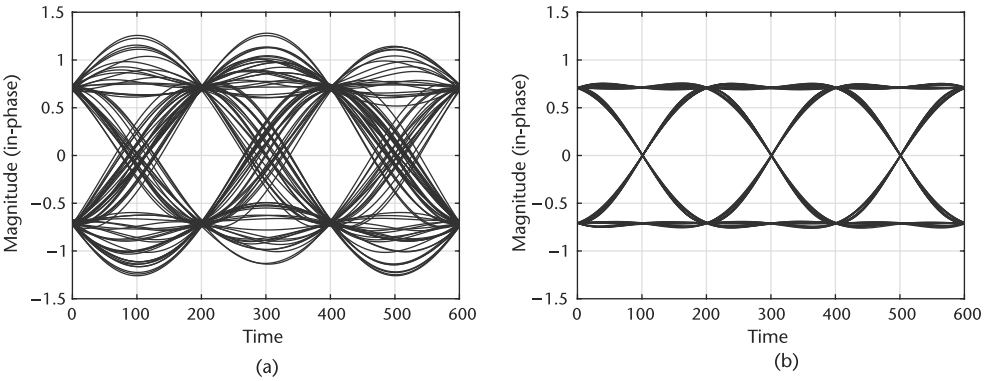

Figure 6.4 Eye diagrams of in-phase portion of QPSK signal after being passed through SRRC filters with different β values. (a) $\beta = 0.3$, and (b) $\beta = 0.99$.

these results the signal was passed through a SRRC filter at the transmitter, AWGN channel with an SNR of 27 dB, and SRRC filter at the receiver. A high SNR was used to keep the eyes open and remove noise as the limiting factor for eventual decisions. At the decisions times 200 and 400 the symbols are well defined in both cases but with $\beta = 0.99$ the transitions are less noisy. However, with a $\beta = 0.99$ the frequency domain outer bands contain more energy as seen in Figure 2.52, which may be undesirable.

Alternatively, we can examine the impulse response with respect to β in Figure 6.5, which compares the RC and SRRC responses. You will notice that unlike the RC filter, the impulse response is not zero at the intervals of T_s. However, the composite response of the SRRC filters at the transmitter and receiver will have zeros at intervals of T_s like the RC filter. The SRRC is used since it is a Nyquist type filter, which produces zero ISI when sampled correctly [1] as discussed in Chapter 2. We can demonstrate the effect of ISI by introducing a simple nonlinearity into the channel and consult the resulting eye diagrams that were introduced in Section 2.4.1. Nonlinearities cause amplitude and phase distortions, which can happen when we clip or operate at the limits of our transmit amplifiers. For more details on the model used, consult [2], but other models exists such as in [3]. In Figure 6.6 we observe the effects of ISI as the eye becomes compressed and correct sampling becomes difficult to determine. We will revisit ISI effects again when equalization is discussed in Chapter 9.

As mentioned previously, rate conversion will typically occur in these transmit or receive filters. Therefore, a polyphase filter can be used where the taps of the SRRC filter are used within the individual arms of the polyphase filter. This is a very efficient implementation since the taps will be applied at the lower rates,

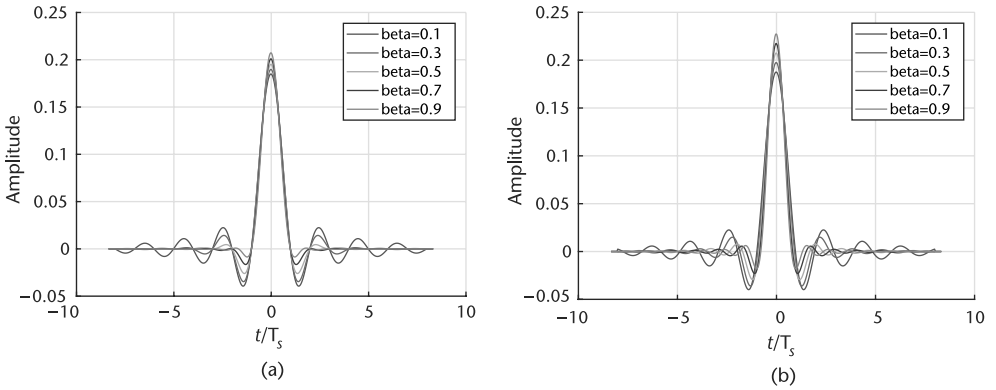

Figure 6.5 Impulse response comparison between raised-cosine and square-root raised-cosine filters. (a) RC impulse response, and (b) SRRC impulse response.

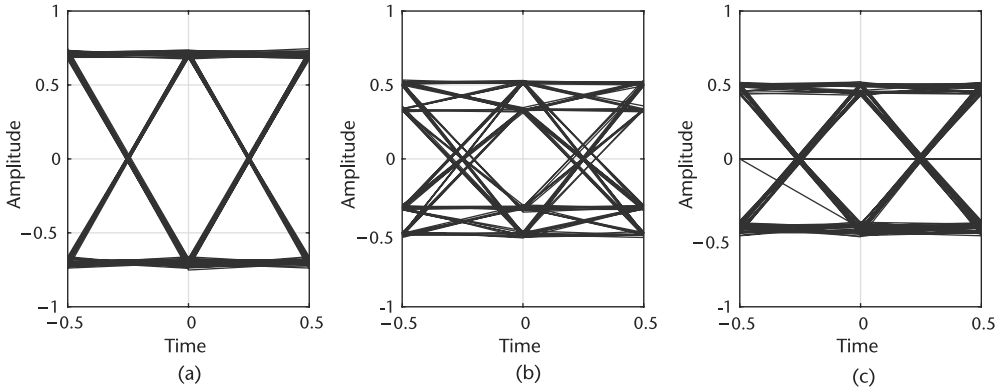

Figure 6.6 Eye diagrams of QPSK signal affected by nonlinearity causing ISI, which is reduced by SRRC matched filtering. (a) Original signal at transmitter, (b) passed through nonlinearity without pulse-shaping, and (c) SRRC filters used at transmitter and receiver with nonlinearity.

before interpolation or after decimation within the polyphase design, reducing the number of multiplies required.

The final aspect of the matched filters we want to discuss and provide insight into is SNR maximization. This argument logically comes out of the concept of correlation. Since the pulsed-shaped/filtered signal is correlated with the pulse-shaped filter and not the noise, matched filtering will have the effect of SNR maximizing the signal, creating peaks at central positions of receive pulses. We demonstrate this effect in Figure 6.7, where we present data transmitted with and without pulse-shaping under AWGN. In the middle plot of Figure 6.7(b) we observe a signal closely related to the originally transmitted sequence, even under high noise. However, without pulse-shaping even visually the evaluation of the transmitted pulse becomes difficult. We even observe demodulation errors in this third plot of Figure 6.7(c) without any timing offset introduced.

6.2 Timing Error

Timing error between transmitter and receiver is a simple concept to understand, but can be difficult to visualize and debug in a communication system. In the most basic sense the purpose of symbol timing synchronization is to align the clocking signals or sampling instances of two disjointed communicating devices. In Figure 6.8(a) we consider a simple example where we overlap the clocking signals of the transmit and receiver nodes and the input signal to be sampled. The sampling occurs at the rising clock edges, and the optimal timing is the transmitter's clock. A small delay τ, less than a single clock cycle, is introduced at the receiver. This is known as a fractional delay since it is less than a sample. Due to this delay the signal is sampled at the wrong positions and the eventual demodulated signal is incorrect. Figure 6.8(b) shows a comparison of the correct and receiver's demodulated symbols with an obvious error occurring at the second transition.

Mathematically we can model this offset received signal r as

$$r(t) = \sum_n x(n)h(t - \tau(t) - nT_s) + v(t), \tag{6.2}$$

where x is the transmitted symbol, h is the pulse shape of the transmit filter, τ is the fractional offset, T_s is the sampling period, n is the sample index, and v is the additive channel noise. After reception the r is passed through the receive matched filter and the relation of the source symbols will be

$$y(t) = \sum_n x(n)h_A(t - \tau(t) - nT_s) + v_h(t), \tag{6.3}$$

where $h_A = h(t) * \bar{h}(-t)$ is the autocorrelation of the transmit filter and its conjugate used on the source data x, v_h is the shaped noise, and y is the output symbols. This demonstrated our notion of matched filtering and correlation. You will notice that the delay τ is indexed as well, since this delay will change since the oscillator at the transmitter and receiver will not have identical frequencies. Therefore, over time this timing difference will drift. However, changes in τ from symbol to symbol should be small relative to the sample rate of the device in a practical RF communication system.

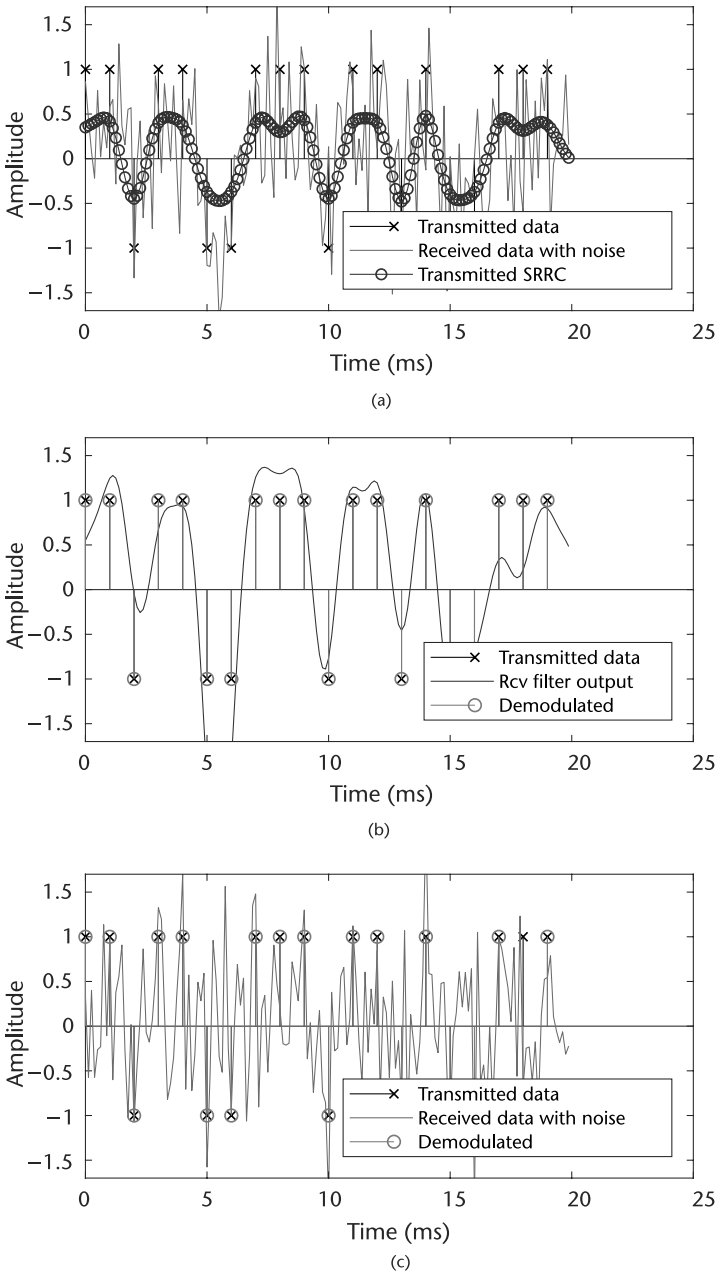

Figure 6.7 Comparison of pulse-shaped and nonpulse-shaped received signals after an AWGN channel. An obvious advantage is visible in the receiver signal when using matched filtering. (a) Transmitted SRRC filtered signal, (b) received SRRC filtered signal, and (c) received signal without SRRC filtering at the receiver.

As discussed in Section 6.1 we will interpolate the signal to be transmitted at the transmit filter stage before actually sending the signal. Although this reduces the throughput of our system it provides the receiver more data to perform decisions without having to oversample the signal itself. In MATLAB we will use `comm.RaisedCosineTransmitFilter`, which first uses a polyphase interpolator to upsample the signal and applies the necessary RC or SRRC taps.

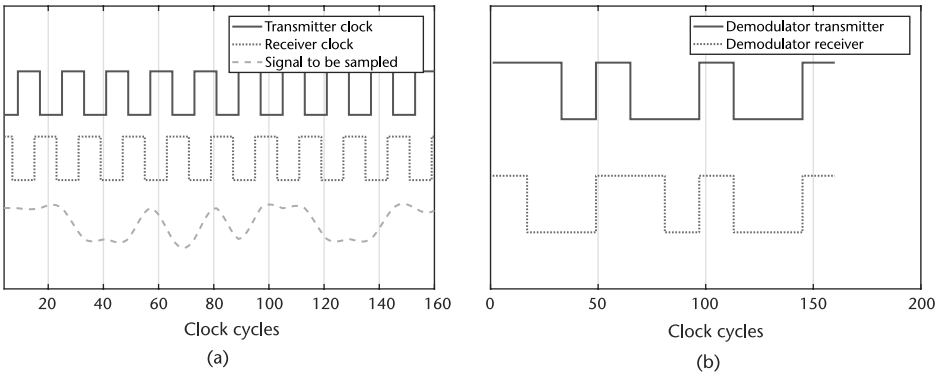

Figure 6.8 Comparison of different clock timings associated with a analog waveform and the resulting sampled and demodulated output. (a) Transmitter and receiver clocking signals with analog waveform to be sampled, and (b) demodulator outputs of receiver and transmitter according to their sampling times.

The upsampling factor N, also known as sample per symbol, will be chosen based on the recovery algorithms used and the desired data rate of the system. In general increasing N can improve the recovery process at the receiver to a point, but again this will reduce our useful bandwidth, forcing hardware to run at higher rates to achieve the same throughput.

Next if we consider timing error from the perspective of the constellation diagram we will observe clustering or scattering of the symbols. In Figure 6.9(a), we provide a simulated timing offsets (τ) of $0.2N$ and $0.5N$, where N is the samples per symbol. An offset of $0.5N$ is the worst case because we are exactly between two symbols. In Figure 6.9(b) we provide a QPSK single transmitted through loopback of a single Pluto SDR. We can clearly observe a similar clustering and some rotation from the transmit and receive chains lack of phase synchronization. This clustering happens because the receiver is actually sampling the transitions between samples. For example, if symbols $y(n)$ and $y(n+1)$ are $[1+i]$ and $[-1-i]$, respectively. Then if they are sampled at time $n+0.5$, the resulting point will be close to zero.

In the case of the Pluto SDR constellation plot in Figure 6.10(b) why does the constellation appear rotated? It may be helpful to refer to Chapter 5.

At the receiver the unknown delay τ must be estimated to provide correct demodulation downstream. A crude but simple way we can illustrate a correction for the offset is to fractionally resample the signal with use of a polyphase filter. We will utilize the `dsp.VariableFractionalDelay` in the script below, which implements a polyphase filter for a given delay provided. We can use this with Pluto SDR to demonstrate different delays we should provide to correct for the offset. At the correct value of $\hat{\tau}$, where $\hat{\tau}+\tau = kT_s$ and $k = \mathbb{Z}_{\geq 0}$, the constellation will have four distinct points.

In Figure 6.10, four example delays are used as estimates to correct for the timing missmatch during loopback on a single Pluto SDR. These represent four instances from the above MATLAB script.

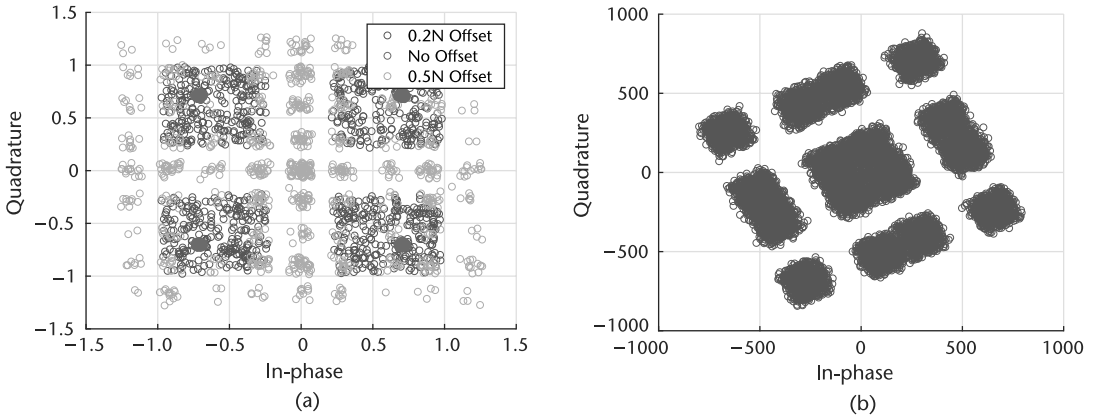

Figure 6.9 Comparison of simulation versus hardware timing offset. (a) Simulation-only example of several timing offsets with a QPSK received signal, and (b) hardware example created with a QPSK signal transmitted through Pluto SDR using a loopback cable.

Code 6.1 Loopback Pluto Example: **`plutoLoopback.m`**

```
% User tunable (samplesPerSymbol>=decimation)
samplesPerSymbol = 12; decimation = 4;
% Set up radio
tx = sdrtx('Pluto','Gain',-20);
rx = sdrrx('Pluto','SamplesPerFrame',1e6,'OutputDataType','double');
% Create binary data
data = randi([0 1],2^15,1);
% Create a QPSK modulator System object and modulate data
qpskMod = comm.QPSKModulator('BitInput',true); modData = qpskMod(data);
% Set up filters
rctFilt = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol', samplesPerSymbol);
rcrFilt = comm.RaisedCosineReceiveFilter( ...
    'InputSamplesPerSymbol',  samplesPerSymbol, ...
    'DecimationFactor',       decimation);
% Pass data through radio
tx.transmitRepeat(rctFilt(modData)); data = rcrFilt(rx());
% Set up visualization and delay objects
VFD = dsp.VariableFractionalDelay; cd = comm.ConstellationDiagram;
% Process received data for timing offset
remainingSPS = samplesPerSymbol/decimation;
% Grab end of data where AGC has converged
data = data(end-remainingSPS*1000+1:end);
for index = 0:300
    % Delay signal
    tau_hat = index/50;delayedsig = VFD(data, tau_hat);
    % Linear interpolation
    o = sum(reshape(delayedsig,remainingSPS,...
      length(delayedsig)/remainingSPS).',2)./remainingSPS;
    % Visualize constellation
    cd(o); pause(0.1);
end
```

6.3 Symbol Timing Compensation

There are many ways to perform correction for symbol timing mismatches between transmitters and receivers. However, in this chapter we will examine three digital

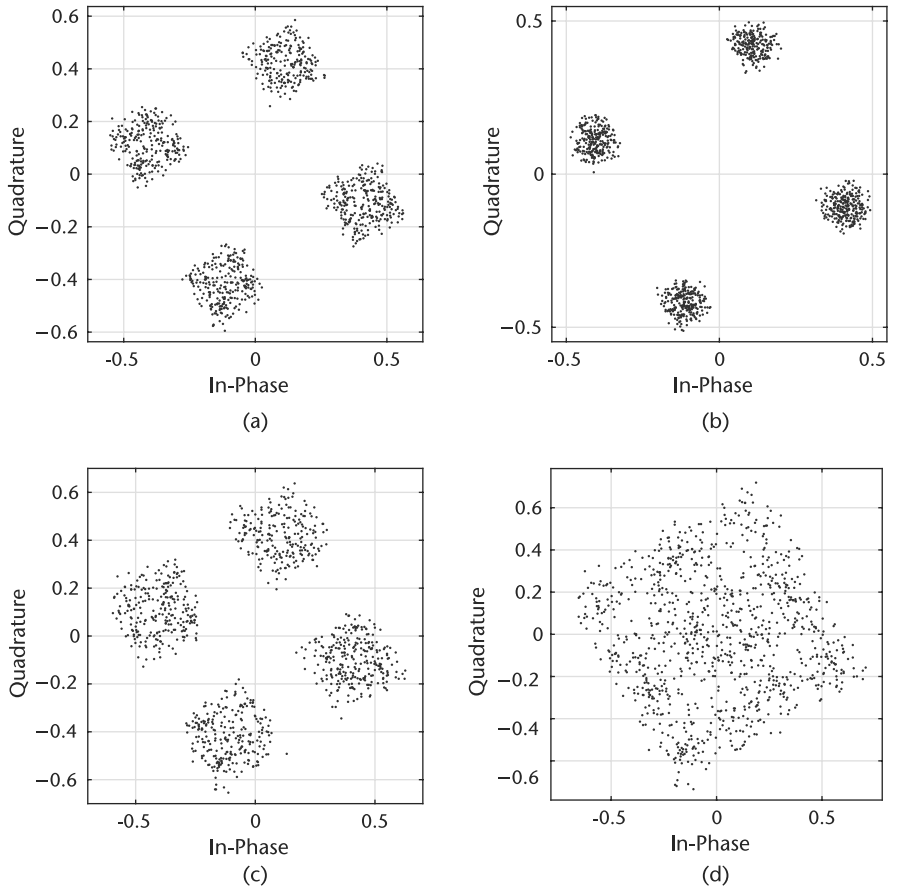

Figure 6.10 Resulting constellations of Pluto SDR loopback data after different fractional delays $\hat{\tau}$. (a) $\hat{\tau} = 0.1$, (b) $\hat{\tau} = 0.5$, (c) $\hat{\tau} = 1$, and (d) $\hat{\tau} = 1.5$.

Using the above MATLAB code verify the timing offset observed. Is this a fixed offset? Change the frequency of both transmitter and receiver to 900 MHz, then explain the observation. Change she sampling rate of both the transmitter and receiver to 2 MHz, then explain your observation.

PLL strategies that will also share the same methodology as in Chapter 7 for our carrier recovery implementations. This type of timing recovery was chosen because it can be integrated with our future recovery solutions, can be robust, and is not overly complex algorithmicly. A basic PLL structure will be introduced first, which will be used to derive our feedback timing correction design. This will frame the discussion around Figure 6.11, where we will replace the individual blocks leading to our eventual design in Figure 6.12. During this process we will provide an overview conceptually how timing is synchronized and then move into each individual block, explaining their design. The specific detectors discussed will be Zero-Crossing, Müller/Mueller, and Gardner. However, more can be found in the literature from Mengali [4] and Oerder [5] among others. Rice [6] provides a more indepth analysis

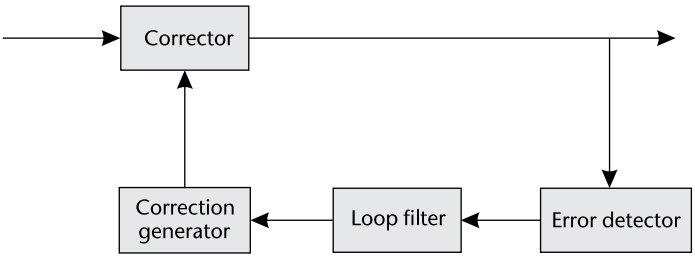

Figure 6.11 Basic PLL structure with four main component blocks.

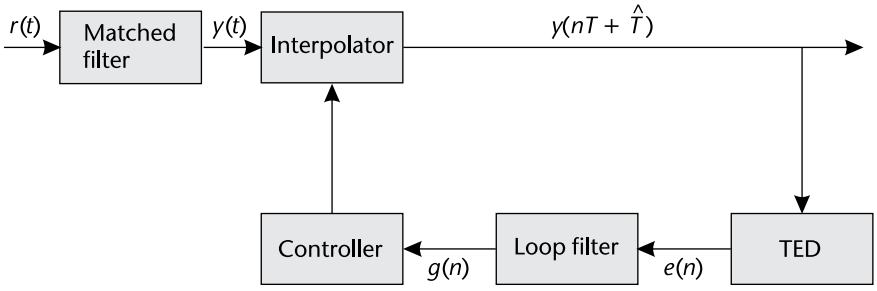

Figure 6.12 Basic structure of PLL for timing recovery for both decision direction and blind timing recovery. There are five major blocks that measure, control, and correct for the timing error in the received signal y.

of these techniques and covers purely analog designs as well as hybrid analog and digital designs. Here we will focus on MATLAB implementations and algorithmic structural features.

6.3.1 Phase-Locked Loops

The timing correction we will be using is a feedback or closed-loop method based on PLL theory. The structure of this algorithm is provided in Figure 6.11 derived from [6, Chapter 7], which essentially locks when an error signal is driven to zero. There are many different designs for PLLs and implementation strategies, but here we will outline four basic components that we will interchange here for timing correction and in the following chapter on carrier recovery. This all-digital PLL-based algorithm shown here works by first measuring some offset, such as timing error or phase error, of the received sample in the error detector (ED), which we call the error signal *e*. The ED is designed based on the structure of the desired receive constellation/symbols or the nature of the sequence itself. Next, the loop filter helps govern the dynamics of the overall PLL. The loop filter can determine operational ranges, lock time, and dampness/responsiveness of the PLL. Next, we have the correction generator. The correction generator is responsible for generation of the correction signal for the input, which again will be fed back into the system. Finally is the corrector itself, which modifies the input signal based on input from the correction generator. Working together, these blocks should eventually minimize *e* over time and contually adapt to future changes from the environment or the system itself.

The correction generator, error detector, and corrector are specific to the purpose of the PLL structure, such as timing recovery or carrier recovery. However,

the loop filter can be shared among the designs with modification to its numerical configuration. The loop filter in all PLL designs is the most challenging aspect, but provides the most control over the adaption of the system. Here we will use a proportional-plus-integrator (PI) filter as our loop filter, which maintains a simple transfer function:

$$F(s) = g_1 + \frac{g_2}{s}, \tag{6.4}$$

where g_1 and g_2 are selectable gains. PI filters produce second-order PLLs and only consist of a single pole in their transfer function; therefore, they are relatively easy to analyze. Since we are dealing with discrete time signals a z-domain representation is preferable:

$$F(z) = G_1 + \frac{G_2}{1 - z^{-1}}, \tag{6.5}$$

where $G_1 \neq g_1$ and $G_2 \neq g_2$.[1] The fractional portion of (6.5) can be represented nicely by a biquad filter.[2] For the calculation of the gain values (G_1, G_2) utilize the following equations based on a preferred damping factor ζ and loop bandwidth B_{Loop}:

$$\theta = \frac{B_{Loop}}{M(\zeta + 0.25/\zeta)} \qquad \Delta = 1 + 2\zeta\theta + \theta^2 \tag{6.6}$$

$$G_1 = \frac{4\zeta\theta/\Delta}{M} \qquad G_2 = \frac{4\theta^2/\Delta}{M} \tag{6.7}$$

where M is the samples per symbol associated with the input signal. Note that B_{Loop} is a normalized frequency and can range $B_{Loop} \in [0, 1]$. If you are interested in how these are derived, see [6, Appendix C]. For the selection of ζ:

$$\zeta = \begin{cases} < 1, \text{Underdamp} \\ = 1, \text{Critically Damped} \\ > 1, \text{Overdamped}, \end{cases} \tag{6.8}$$

which will determine the responsiveness and stability of the PLL.

6.3.2 Feedback Timing Correction

The timing synchronization PLL used in all three algorithms consists of four main blocks: interpolator, timing ED (TED), loop filter, and an interpolator controller. Their interaction and flow is presented in Figure 6.14. These operations first estimate an unknown offset error, scale the error proportionally, and apply an update for future symbols to be corrected. To provide necessary perspective on the timing error, let us considered the eye diagram presented in Figure 6.13. This eye diagram has been upsampled by twenty times so we can examine the transitions more closely, which we notice are smooth unlike Figure 6.6. In the optimal case, we chose to sample our input signal at the dark gray instances at the widest openings of the eye. However, there will be an unknown fractional delay τ that shifts this sampling period. This shifted sampling position is presented by the light gray selections. To help work around this issue, our receive signal is typically not decimated fully,

1. A simple way to translate between (6.4) and (6.5) is to utilize a bilinear transform.
2. See `dsp.BiquadFilter` for a simple realization of a biquad filter.

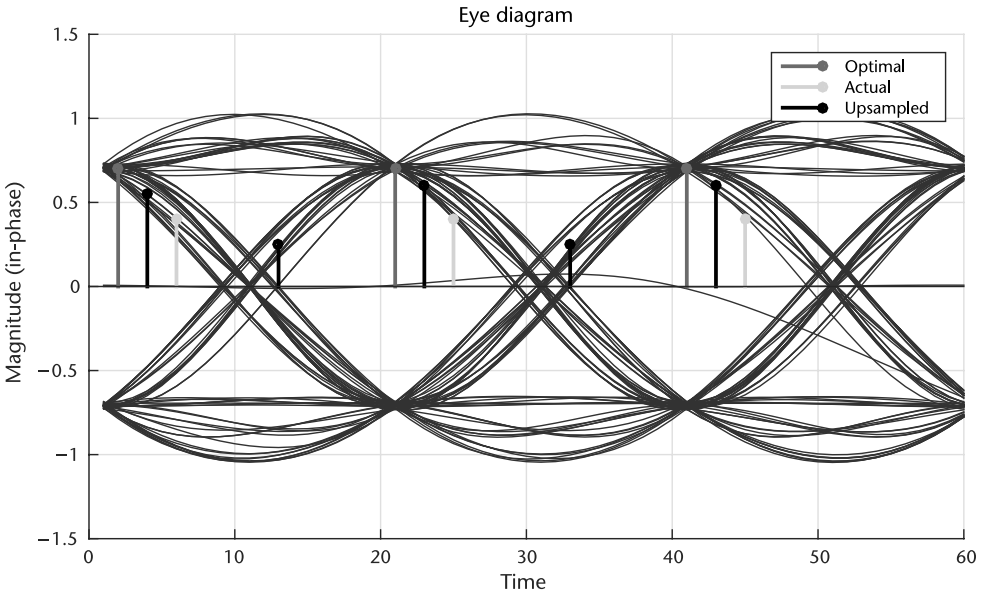

Figure 6.13 Eye diagram of received signal marking positions where received samples may exist. This figure is highly oversampled to show many positions, but a received sample could lie anywhere on the eye.

providing the receiver with multiple samples per symbol (this is not always the case). Therefore, if we are straddling the optimal sampling position instead as in the black markers, we can simply interpolate across these points to get our desired period. This interpolation has the effect of causing a fractional delay to our sampling, essentially shifting to a new position in our eye diagram. Since τ is unknown we must weight this interpolation correctly so we do not overshoot or undershoot the desired correction. This is similar to the idea presented at the end of Section 6.2. Finally, controlling the instances in time when an output is produced or sampled from the input data is the function of the interpolator control block, which will be at the symbol rate. This correction loop, when implemented properly, that will cause the eye diagram to open for input signals with clock timing missmatches. However, a constellation diagram may also be useful tool for evaluating timing correction as presented in Figures 6.4 and 6.9.

We will initially discuss the specifics of the blocks in Figure 6.12 through the perspective of the zero-crossing (ZC) method, since it is the most straightforward to understand. Then we will provide extensions to the alternative methods. ZC, as the name suggests, will produce an error signal $e(n)$ of zero when one of the sampling positions is at the zero intersection. ZC requires two samples per symbol or more, resulting in the other sampling position occurring at or near the optimal position. The TED for ZC [4] is evaluated as

$$\begin{aligned} e(n) =& Re(y((n-1/2)T_s+\tau))[sgn\{Re(y((n-1)T_s+\tau))\} - sgn\{Re(y(nT_s+\tau))\}] \\ &+ Im(y((n-1/2)T_s+\tau))[sgn\{Im(y((n-1)T_s+\tau))\} \\ &- sgn\{Im(y(nT_s+\tau))\}], \end{aligned} \tag{6.9}$$

where *Re* and *Im* extract the real and imaginary components of a sample, and *sgn* process the sign (−1 or 1) for the sample. In (6.9) it is important to note that these

indexes are with respect to samples, not symbols, and to be specifc $y(nT_s + \tau)$ is simply the latest output of the interpolator filter. Looking at (6.9) it first provides a direction for the error with respect to the *sgn* operation, and the shift required to compensate is determined by the midpoints. The in-phase and quadrature portions operate independently, which is desirable.

Once the error is calculated it is passed to the loop filter, which we can entirely borrow from Section 6.3.1. The same principles apply here, with a near identical formulation for our equations we have provided in a slightly more compact form.

$$G_1 = \frac{-4\zeta\theta}{G_D N\Delta} \qquad G_2 = \frac{-4\theta^2}{G_D N\Delta} \tag{6.10}$$

Here B_{Loop} is the normalized loop bandwidth, ζ is our damping factor, N is our samples per symbol, and G_D is our detector gain. The new variable G_D provides an additional step size scaling to our correction. Again the loop filter's purpose is to maintain stability of the correction rate. This filter can be implemented with a simple linear equation:

$$y(t) = G_1 x(t) + G_2 \sum_{n=0} y(n), \tag{6.11}$$

or with a biquad filter.

The next block to consider is the Interpolation Controller, which is responsible to providing the necessary signaling to the interpolator. With respect to our original PLL structure in Figure 6.11 the interpolation controller takes the place of the correction generator. Since the interpolator is responsible for fractionally delaying the signal, this controller must provide this information and generally the starting interpolant sample. By starting interpolant sample we are referring to the sample on the left side of the straddle, as shown by the second black sampling position from the left in Figure 6.13. The interpolation controller implemented here will utilize a counter-based mechanism to effectively trigger at the appropriate symbol positions. At these trigger positions the interpolator is signaled and updated, as well as an output symbol is produced from the system.

The main idea behind a counter-based controller is to maintain a specific triggering gap between updates to the interpolator, with an update period on average equal to symbol rate N of the input stream. In Figure 6.14 a logical flowchart of the interpolation controller is provided to better understand the complex flow. If we consider the case when the timing is optimal and the output of the loop filter $g(n)$ is zero, we would want to produce a trigger every N samples. Therefore, it is logical that the weighting or decrement for the counter would be

$$d(n) = g(n) + \frac{1}{N}. \tag{6.12}$$

resulting in a maximum value of 1 under modulo-1 subtraction of the counter $c(n)$, where wraps of the modulus occur every N subtractions. This modulus counter update is defined as

$$c(n+1) = (c(n) - d(n)) \mod 1. \tag{6.13}$$

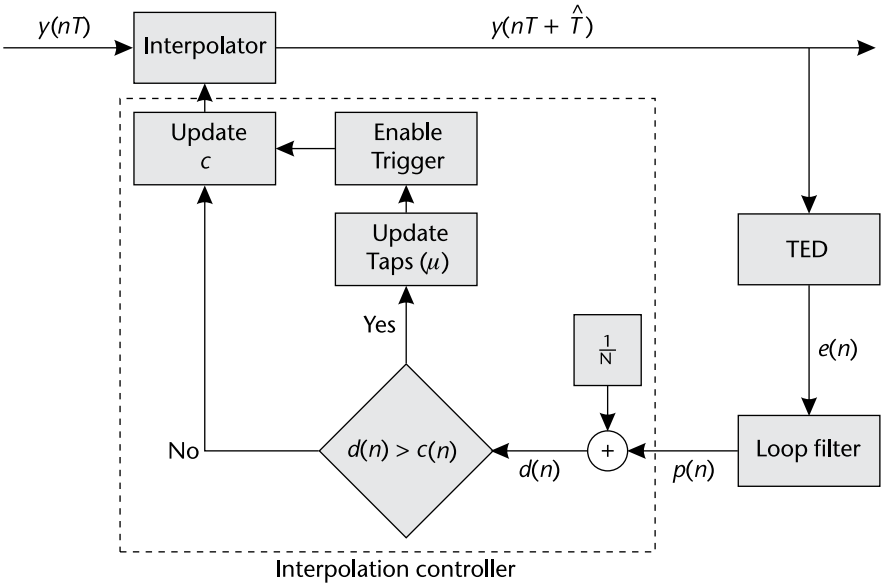

Figure 6.14 Timing recovery triggering logic used to maintain accurate interpolation of input signal.

We determine a trigger condition, which is checked before the counter is updated, based on when these modulus wraps occur. We can easily check for this condition before the update, such as

$$Trigger = \begin{cases} c(n) < d(n) & True \\ \text{Otherwise} & False \end{cases}. \tag{6.14}$$

This triggering signal is the method used to define the start of a new symbol; therefore, it can also be used to make sure we are estimating error over the correct samples. When the trigger occurs we will update $\mu(n)$ our estimated gap between the interpolant point and the optimal sampling position. This update is a function of the new counter step $d(n)$ and our current count $c(n)$:

$$\mu(k) = c(n)/d(n). \tag{6.15}$$

This μ will be passed to our interpolator to update the delay it applies.

We want to avoid performing timing estimates that span over multiple symbols, which would provide incorrect error signals and incorrect updates for our system. We can avoid this by adding conditions into the TED block. We provide additional structure to the TED block in Figure 6.15, along with additional logic to help identify how we can effectively utilize our trigger signals. Based on this TED structure, only when a trigger occurs the output error e can be nonzero. Looking downstream in Figure 6.14 from the TED, since we are using a PI loop filter only nonzero inputs can update the output, and as a result modify the period of the triggering associated d. When the system enters steady state, where the PLL has locked, the TED output can be nonzero every N samples.

The final piece of the timing recovery we have not yet discussed is the interpolator itself. With respect to our original PLL structure in Figure 6.11 the interpolator takes the place of the corrector. Interpolation here is simply a linear

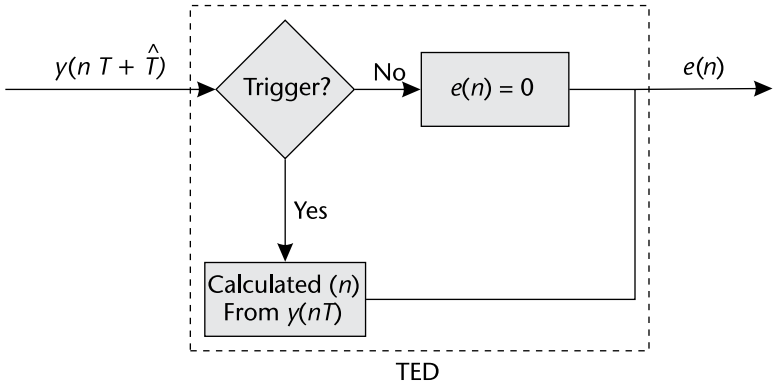

Figure 6.15 An internal view of the timing error detector to outline the error and triggering control signals relation to operations of other blocks in Figure 6.14.

combination of the current and past inputs y, which in essence can be thought of as a filter. However, to create a FIR filter with any arbitrary delay $\tau \in [0, ..., T_s]$ cannot be realized [7]. Realizations for ideal interpolation IIR filters do exist, but the computation of their taps are impractical in real systems [8]. Therefore, we will use an adaptive implementation of a FIR lowpass filter called a piecewise polynomial filter (PPF) [6]. The PPF can only provide estimations of offsets to a polynomial degree. Alternative implementations exists such as polyphase-filterbank designs, but depending on the required resolution the necessary phases become large. However, they can be straightforward to implement [9].

The PPF are useful since we can easily control the form of interpolations by determining the order of the filter, which at most is equivalent to the order of the polynomial used to estimate the underlying received signal. Here we will use a second order, or quadratic, interpolation requiring a four-tap filter. The general form of the interpolator's output is given by

$$y(kT_s + \mu(k)T_s) = \sum_{n=1}^{2} h(n)y((k-n)T_s), \tag{6.16}$$

where h_k are the filter coefficients at time instance k determined by [10]:

$$\begin{aligned} h = &[\alpha\mu(k)(\mu(k)-1), \\ &-\alpha\mu(k)^2 - (1-\alpha)\mu(k) + 1, \\ &-\alpha\mu(k)^2 + (1+\alpha)\mu(k), \\ &\alpha\mu(k)(\mu(k)-1)], \end{aligned} \tag{6.17}$$

where $\alpha = 0.5$. $\mu(k)$ is related to the fractional delay, which is provided by the interpolator control block, which relates the symbol period T_s to the estimated offset. Therefore, we can estimate the true delay τ as

$$\hat{\tau} \sim \mu(k)T_s. \tag{6.18}$$

Without any offset ($\mu = 0$), the interpolator acts as a two-sample delay or single-symbol delay for the ZC implementation. We can extend the PPF to utilize

more samples creating cubic and greater interpolations, but their implementations become more complex. The underlying waveform should be considered when determining the implementation of the interpolator as well as the required degrees of freedom to accurately capture the required shape.

This design using four samples in a quadratic form can be considered irregular, since the degree of taps does not reach three. However, odd length realizations (using an odd number of samples) are not desirable since we are trying to find values in-between the provided samples. We also do not want a two-sample implementation due to the curvature of the eye in Figure 6.13.

In MATLAB we can realize this interpolator with a few lines of code that are dependent on the input data y and the last output of the interpolator controller μ provided in Code 6.2.

Code 6.2 Interpolator: **interpFilter.m**

```
% Define interpolator coefficients
alpha = 0.5;
InterpFilterCoeff = ...
    [ 0,       0,         1,       0;    % Constant
    -alpha, 1+alpha, -(1-alpha), -alpha; % Linear
    alpha,  -alpha,    -alpha,   alpha]; % Quadratic
% Filter input data
ySeq = [y(i); InterpFilterState]; % Update delay line
 % Produce filter output
filtOut = sum((InterpFilterCoeff * ySeq) .* [1; mu; mu^2]);
InterpFilterState = ySeq(1:3); % Save filter input data
```

From this output *filtOut* we can drive our TED using the ZC equation (6.9) to create error signals for the loop filter and interpolator controller. Based on Figure 6.14 we know that this TED calculation will be based on a triggered signal from the interpolator controller. Additional historical triggers are also checked which prevent driving the output of the timing loop faster than the symbol rate. This logic and TED measurement is captured in Code 6.3.

Additional logic is added to the TED from lines 13 to 22, which manage symbol stuffing. Symbol stuffing is basically forcing an additional trigger from the synchronizer routine. This is necessary when clock deviations force the interpolator to minimally straddle the symbol of interest. To compensate we must insert an additional output symbol. Note that at the output of the system, the sample rate will equal the symbol rate, essentially downsampling our signal when $N > 1$.

Following the TED is the loop filter, which has already been discussed in Section 6.3.1. Since the filter is quite simple it can be implemented in a straightforward way without filter objects. However, using a biquad filter object provides more compact code as shown Code 6.4.

Finally, we can evaluate the filtered error at the interpolator control block. In steady state this block should produce a trigger every N input samples. This trigger signal can be considered a valid output signal, which will concide with output data from the whole algorithm. In the coding context here, when *Trigger* is true at time n the output of the interpolation filter at input $n + 1$ should be processed

Code 6.3 ZC TED: **`zcTED.m`**

```
% ZC-TED calculation occurs on a strobe
if Trigger && all(~TriggerHistory(2:end))
    % Calculate the midsample point for odd or even samples per symbol
    t1 = TEDBuffer(end/2 + 1 - rem(N,2));
    t2 = TEDBuffer(end/2 + 1);
    midSample = (t1+t2)/2;
    e = real(midSample)*(sign(real(TEDBuffer(1)))-sign(real(filtOut))) ...
        imag(midSample)*(sign(imag(TEDBuffer(1)))-sign(imag(filtOut)));
else
    e = 0;
end
% Update TED buffer to manage symbol stuffs
switch sum([TriggerHistory(2:end), Trigger])
    case 0
      % No update required
    case 1
      % Shift TED buffer regularly if ONE trigger across N samples
      TEDBuffer = [TEDBuffer(2:end), filtOut];
    otherwise % > 1
      % Stuff a missing sample if TWO triggers across N samples
      TEDBuffer = [TEDBuffer(3:end), 0, filtOut];
end
```

Code 6.4 Loop Filter: **`loopFilter.m`**

```
% Loop filter
loopFiltOut = LoopPreviousInput + LoopFilterState;
g = e*ProportionalGain + loopFiltOut; % Filter error signal
LoopFilterState = loopFiltOut;
LoopPreviousInput = e*IntegratorGain;
% Loop filter (alternative with filter objects)
lf = dsp.BiquadFilter('SOSMatrix',tf2sos([1 0],[1 -1])); % Create filter
g = lf(IntegratorGain*e) + ProportionalGain*e; % Filter error signal
```

downstream. The interpolation controller itself will utilize the filtered error signal g and will update the internal counter as data is processed in Code 6.5.

Code 6.5 Interpolator Control Logic: **`interpControl.m`**

```
% Interpolation Controller with modulo-1 counter
d = g + 1/N;
TriggerHistory = [TriggerHistory(2:end), Trigger];
Trigger = (Counter < d); % Check if a trigger condition
if Trigger % Update mu if a trigger
    mu = Counter / d;
end
Counter = mod(Counter - d, 1); % Update counter
```

The overall design of the timing synchronizer can be complex and implementations do operate at different relative rates. Therefore, we have provided Table 6.1 as a guide to a recommended implementation. These rates align with

Table 6.1 Operational Rates of Timing Recovery Blocks

Block	*Operational rate*
Interpolator	Sample rate
TED	Symbol rate
Loop filter	Symbol rate
Interpolator controller	Sample rate

the trigger implementation outlined in Figure 6.14. This system will result in one sample per symbol (N) when output samples of the interpolator are aligned with the triggers.

Starting with script `TimingError`, which models a timing offset, implement ZC timing correction.

6.4 Alternative Error Detectors and System Requirements

Within the discussed PLL framework alternative TEDs can be used if the application or system arrangement is different. For example, the discussed method of ZC cannot operate under carrier phase or frequency offsets. Therefore, such a nonideality would require compensation first before application of ZC, which is not true for other methods. Besides carrier offsets, a requirement of the ZC method is an upsample factor N of at least two, which may not be possible for certain systems due to bandwidth and data rate constraints.

6.4.1 Gardner

The second TED we will considered is called Gardner [11], which is very similar to ZC. The error signal is determined by

$$\begin{aligned} e(n) = Re(y((n-1/2)T_s+\tau))\big[Re(y((n-1)T_s+\tau)) - Re(y(nT_s+\tau))\big]+ \\ Im(y((n-1/2)T_s+\tau))\big[Im(y((n-1)T_s+\tau)) - Im(y(nT_s+\tau))\big]. \end{aligned} \tag{6.19}$$

This method also requires two samples per symbol and differs only in the quantization of the error direction from ZC. One useful aspect of Gardner is that it does not require carrier phase correction and works especially well with BPSK and QPSK signals. However, since Gardner is not a decision-directed method, for best performance the excess bandwidth of the transmit filters should be $\beta \in (0.4, 1)$.

Implement the Gardner TED inside your existing timing error detector. Introduce a small phase shift into the received signal of $\pi/4$. Compare ZC and Gardner in these cases.

6.4.2 Müller and Mueller

Next is the Müller and Mueller (MM) method named after Kurt Mueller and Markus Müller [12]. This can be considered the most efficient method since it does

not require upsampling of the source data, operating at one sample per symbol. The error signal is determined by [6]

$$\begin{aligned} e(k) = & Re(y((k)T_s + \tau)) \times sgn\{Re(y((k-1)T_s + \tau))\} \\ & - Re(y((k-1)T_s + \tau)) \times sgn\{Re(y((k)T_s + \tau))\} \\ & + Im(y((k)T_s + \tau)) \times sgn\{Im(y((k-1)T_s + \tau))\} \\ & - Im(y((k-1)T_s + \tau)) \times sgn\{Im(y((k)T_s + \tau))\}. \end{aligned} \tag{6.20}$$

MM also operates best when the matched filtering used minimizes the excess bandwidth, meaning β is small. It is important to note when the excess bandwidth of the receiver or transmitter filters is high the decisions produced by the *sgn* operation can be invalid. Therefore, this trade-off must be considered during implementation. However, even though MM is technically most efficient performance can be questionable at $N = 1$ due to the lack of information available per symbol.

Add phase and frequency offsets to the input signal and compare the performance of ZC, Gardner, and MM estimation methods. Do this for fixed fractional delays $\frac{T_s}{2}$, $\frac{T_s}{4}$, $\frac{T_s}{5}$ in the channel and plot the error output of the TEDs for Gardner and ZC.

6.5 Putting the Pieces Together

Throughout this chapter we have outlined the structure and logic behind a PLL-based timing recovery algorithm and the associated MATLAB code. In the remaining sections we will discuss putting the algorithmic components together and provide some intuition on what happens during evaluation. Here we will also address parameterization and the relation to system dynamics.

The system-level scripts have shown a constant theme throughout where data is modulated, transmit filtered, passed through a channel with timing offset, filtered again, then is timing recovered. Many rate changes can happen in this series of steps. To help understand these relations better we can map things out as in Figure 6.16, which takes into account these stages. Here the modulator produces symbols equal to the sample rate. Once passing through the transmit filter we acquire our upsampling factor N, which increases our samples per symbol to N. At the receiver we can perform decimation in the receive filter by a factor N_F where $N_F \leq N$. Finally, we will perform timing recovery across the remaining samples and remove the fractional offset τ, returning to the original rate of one sample per symbol. The rate pattern outlined in Figure 6.16 is identical to that of the first MATLAB script in Code 6.1. That script can be modified to produce a slightly dynamic timing offset, which we provide below:

From Code 6.6 we can evaluate the receive filtered signal with a variable offset over time. Figure 6.17(a) provides the direct output of `rxFilt` when `samplesPerSymbol` is equal to `decimation`, where we can observe the constellation of the signal collapsing into constellations over time similar to Figure 6.9. This is essentially when no timing recovery is being used. Next,

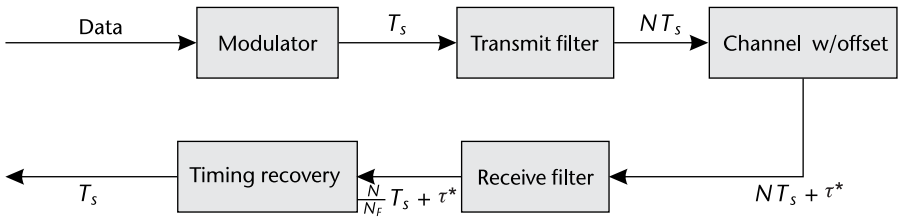

Figure 6.16 Relative rates of transmit and receive chains with respect to the sample rate at different stages. Here τ^* represents a timing shift not an increase in the data rate. This is a slight abuse of notation.

Code 6.6 Transmit Filter Data: **`srrcFilterData.m`**

```
% User tunable (samplesPerSymbol>=decimation)
samplesPerSymbol = 4; decimation = 2;
% Create a QPSK modulator System object and modulate data
qpskMod = comm.QPSKModulator('BitInput',true);
% Set up filters
rctFilt = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol', samplesPerSymbol);
rcrFilt = comm.RaisedCosineReceiveFilter( ...
    'InputSamplesPerSymbol',  samplesPerSymbol, ...
    'DecimationFactor',       decimation);
% Set up delay object
VFD = dsp.VariableFractionalDelay;
% Delay data with slowly changing delay
rxFilt = [];
for index = 1:1e3
    % Generate, modulate, and tx filter data
    data = randi([0 1],100,1);
    modFiltData = rctFilt(qpskMod(data));
    % Delay signal
    tau_hat = index/30;
    delayedsig = VFD(modFiltData, tau_hat);
    rxSig = awgn(delayedsig,25); % Add noise
    rxFilt = [rxFilt;rcrFilt(rxSig)]; % Rx filter
end
```

taking the lessons from this chapter and utilizing the implementation of timing recovery proposed, we can adapt to this changing fractional delay. Figure 6.17(b) demonstrates the recovery for the ZC technique when $\frac{N}{N_F} = 2$. Here we can observe clear division between the level for the real component of the signal, meaning our output constellation is collapsing to a correct QPSK signal. In Figure 6.17(c) we increase B_{Loop} from 0.001 to 0.01, which causes the system to react faster. However, for $B_{Loop} = 0.001$ once converged the residual symbols have less noise than for $B_{Loop} = 0.01$.

Regenerate Figure 6.17 and utilize alternative $\zeta = \{0.5, \sqrt{2}, 10, 20\}$. Comment on the dynamic of the recovery algorithm.

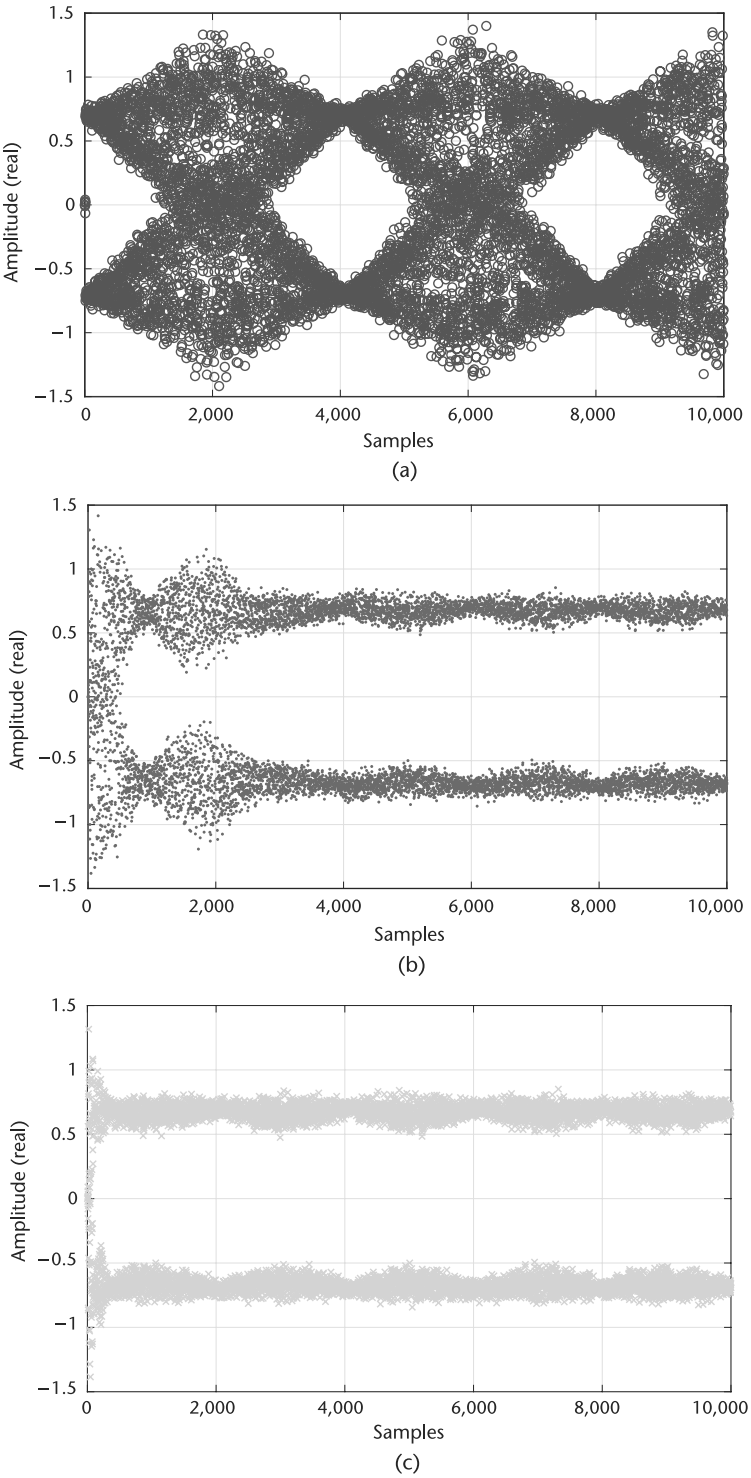

Figure 6.17 Comparison of a signal that requires timing recovery, and outputs of two parameterization of ZC timing recovery after application. (a) Receive signal without timing recovery, (b) receive signal with ZC timing recovery for parameterization $\{N, \zeta, B_{Loop}, G_D\} = \{2, 1, 0.001, 2.7\}$, and (c) receive signal with ZC timing recovery for parameterization $\{N, \zeta, B_{Loop}, G_D\} = \{2, 1, 0.01, 2.7\}$.

Regenerate Figure 6.17, but utilize Pluto SDR in loopback as the channel. Tune the recovery algorithm (N, ζ, B_{Loop}, G_D) and measure the best conference you can achieve.

6.6 Chapter Summary

Timing recovery is a fundamental tool for successful transmission between nodes with independent oscillators. In this chapter, a model for timing offset was introduced mathematically, in simulation, and demonstrated with Pluto SDR. To combat this offset, a PLL-based timing recovery methodology was introduced that included several timing error detectors. This included an review and extension to matched filtering introduced in Chapter 6. MATLAB code was provided for the different components of the timing recovery algorithms, and a considerable amount of time was spent examining their configuration and interactions. Finally, once all the components were investigated, portions of the design's parameterization were explored. In subsequent chapters, the implementations developed here will be utilized to created a full receiver design which can recover signals transmitted between separate Pluto SDR devices.

References

[1] Proakis, J., and M. Salehi, *Digital Communications*, Fifth Edition, Boston: McGraw-Hill, 2007.

[2] Saleh, A. A. M., "Frequency-Independent and Frequency-Dependent Nonlinear Models of TWT Amplifiers," *IEEE Transactions on Communications*, Vol. 29, No. 11, November 1981, pp. 1715–1720.

[3] Boumaiza, S., T. Liu, and F. M. Ghannouchi, "On the Wireless Transmitters Linear and Nonlionear Distortions Detection and Pre-correction," in *2006 Canadian Conference on Electrical and Computer Engineering*, May 2006, pp. 1510–1513.

[4] Mengali, U., *Synchronization Techniques for Digital Receivers*, Applications of Communications Theory, New York: Springer, 2013.

[5] Oerder, M., and H. Meyr, "Digital Filter and Square Timing Recovery," *IEEE Transactions on Communications*, Vol. 36, No. 5, May 1988, pp. 605–612.

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

[7] Laakso, T. I., V. Valimaki, M. Karjalainen, and U. K. Laine, "Splitting the Unit Delay [FIR/All Pass Filters Design]," *IEEE Signal Processing Magazine*, Vol. 13, No. 1, January 1996, pp. 30–60.

[8] Thiran, J. P., "Recursive Digital Filters with Maximally Flat Group Delay," *IEEE Transactions on Circuit Theory*, Vol. 18, No. 6, November 1971, pp. 659–664.

[9] Rice, M., and F. Harris, "Polyphase Filterbanks for Symbol Timing Synchronization in Sampled Data Receivers," in *MILCOM 2002, Proceedings*, Vol. 2, October 2002, pp. 982–986.

[10] Erup, L., F. M. Gardner, and R. A. Harris, "Interpolation in Digital Modems. ii. Implementation and Performance," *IEEE Transactions on Communications*, Vol. 41, No. 6, June 1993, pp. 998–1008.

[11] Gardner, F., "A BPSK/QPSK Timing-Error Detector for Sampled Receivers," *IEEE Transactions on Communications*, Vol. 34, No. 5, May 1986, pp. 423–429.

[12] Mueller, K., and M. Muller, "Timing Recovery in Digital Synchronous Data Receivers," *IEEE Transactions on Communications*, Vol. 24, No. 5, May 1976, pp. 516–531.