Receiver Modeling with OCS

John Zweck

April 1, 2003

In OCS applications a receiver for a WDM system typically consists of a tunable optical filter from the class `OptFilter`, which is used by an instance of the optical demuxer class `OptDemuxer`, a `Photodetector`, an electric filter from the class `ElecFilter`, and an instance of the `ElecSignalStat` class which is used to compute the statistics of the received current. In the presence of optical noise we either use Monte Carlo simulations, the semi-analytical receiver model of Winzer/Lima, or the generalized χ^2 receiver model as described in the thesis of Ronald Holzlöhner.

1 Receiver model for Monte Carlo simulations

Code for the receiver in the case that Monte Carlo simulations are used can be found in the Tyco function of MyApp.cc.

The major function calls are as follows:

1. `OptFilter TunableFilter("TunableOptFilter.in", Signal->GetTypeSimulation());`
   Constructs an instance `TunableFilter` of an `OptFilter`. The main input parameters are:
   - `$TypeOptFilter 1`
     1 = (Super)Gaussian is only real option right now.
   - `$OrderOptFilter 1`
     1 = Gaussian, $n > 1$ gives a SuperGaussian of order n in a new version of the code you don't have yet.
   - $FreqFWHM_OptFilter 60e9
     Full-width at half maximum in Hz of the power of the optical filter transfer function in the frequency domain.
2. `OptDemuxer oOptDemuxer(Signal, TunableFilter);`
   Constructor for an `OptDemuxer`.

3. `SCSignal = new OptSignal(oOptDemuxer.ExtractChannel(0));`
   Constructs a single channel signal `SCSignal` by demuxing out the channel with physical index 0 from the WDM (or single channel) `Signal`.

   - If you specify the flag `EvenlySpacedFreqsFlag = 0` in `Signal.in` then the channel with wavelength given by `$WavelengthChann1` will be demuxed.
   - If you specify the flag `EvenlySpacedFreqsFlag = 1` then the center channel at `Signal::CenterFreq` will be demuxed.

   To compute the statistics of the received current we need to know the input `BitString` in the channel to be demuxed. Among other things, the method `OptDemuxer::ExtractChannel(int Index)` makes sure that the `BitString` of the demuxed channel is correctly extracted from that of the WDM signal.

4. `Photodetector oPhotodetector("Photodetector.in",SCSignal,RNG);`
   `oPhotodetector.DetectOptSignal();`
   These lines pass `SCSignal` through a square-law photodetector. The data for the electrical current is stored in an array of length `qtPoints` in the `Photodetector`.

5. Although the data array used to store the electrical current is `cplx`-valued (so we can easily take Fourier transforms), every method operating on the current always outputs a `real`-valued current. In particular, as is the case with Bessel filtering, the current can become negative.

6. We usually ignore detector noise and all other electrical noise in the receiver and we set the photodetector responsivity to 1.[1] Therefore you don't need to pre-amplify the optical signal, unless you want to ensure a fixed average received current.

7. `oElecFilter = new ElecFilter(InDir + "ElecFilter.in",&oPhotodetector);`

   This line constructs an `ElecFilter` object which acts on the data in `oPhotodetector`. The main parameters read in from `"ElecFilter.in"` are

   - `$TypeElecFilter 2`:
     1 = Gaussian, 2 = Bessel, 3 = Measured, 4 = IntegrateAndDump
     For Option 3 see documentation of the method
     `void ElecFilter::ReadMeasuredFilterData(void)`
     Use option 4 with caution; Read code to see exactly what it does.
   - `$OrderElecFilter`: For a Bessel filter you can specify the order to be 4 or 5. [2]
   - `$f_3dB`: The half-width at half maximum of the power spectrum of the filter. We specify a half width because electric filters are usually centered at frequency 0 and are one sided (low pass).
   - `$CenterFreqElecFilter`: Set to 0 unless you have a good reason not to.

[1] The code in the `Photodetector` to add shot noise has not been debugged or validated.

[2] At the moment because of a bug I think you also need to set OrderElecFilter to be 4 or 5 for a Gaussian filter, but the value is meaningless in that case.

8. `oElecFilter->FilterElecSignal();`
   This line does the filtering.

9. `oElecSignalStat = new ElecSignalStat(SCSignal,&oPhotodetector);`
   Constructor for the `ElecSignalStat` class. This class does not have any input parameters!!

10. The `ElecSignalStat` class recovers the clock (see below) and computes the statistics of the received current at the clock recovery time. This is all done when you call the method: `oElecSignalStat->GetSingleStringElecSignalStatConvolutingNEW();`.

11. You can write out an eye diagram using the method `oElecSignalStat->WriteFileEyeDiagram(Job+"Eye.dat");`. This method is particularly useful when doing Monte Carlo simulations with a limited number (less than 100) random realizations of the system. With more random realizations the output eye diagram file gets very large. In that case I recommend collecting the eye data using a `Histogram2D` object (see below).

12. The `ElecSignalStat` class has many `Get...` methods which you can then call to print out statistics obtained using Monte Carlo simulations. For example: `GetBit0_PowerMean(); GetBit1_PowerMean(); GetBit0_PowerStdDev(); GetBit1_PowerStdDev)(); GetElecTimeDomainSNR(); GetMinAmplitudeMargin(); GetQ_Factor(); GetBitErrorRate();`[3] `GetClockTime(); GetClockCurrent();`

All the above is actually for a single realization in a Monte Carlo simulation. To accumulate statistics of the system performance using many system realizations we do something like the following.

1. At the start of the program we call all constructors of all objects used to model the system

2. `for(int RealizationNumber = 1; RealizationNumber <= Startup->GetNumMonteCarloExpts(); RealizationNumber++)` Within this `for` loop you need to:

   - Pass the optical signal through the fibers and amplifiers
   - Pass the optical signal through the optical filter and photodetector to generate an electrical signal
   - Pass the electrical signal through the electrical filter
   - Update the statistics. At the moment the way this is done is by calling the method `ElecSignalStat::AddSignalStringSampleConvoluting();` within the `for` loop. However we have a bug in our clock recovery method. To get around this bug you will need to add a new method to `ElecSignalStat` called `AddSignalStringSampleNEW` which is implemented as:

[3] Assuming Gaussian distributions of the received current

```
void ElecSignalStat::AddSignalStringSampleNEW(void)
{
ClockRecoveryTime = GetTargetTime();
oPhotodetector->TimeShiftSignal( -ClockRecoveryTime );
AddSignalStringSample();
}
```

Note that before entering the `for` loop you should call the method `ElecSignal-Stat::Clear()`.

- If you are also collecting samples for a 2D contour plot of an eye diagram pdf you will need to add a call to `Histogram2D::UpdateEyeDiagram()`
- Finally at the end of the `for` loop you need to construct a fresh copy of the signal to be passed trhough the system with the next random realization of the system parameters, i.e., call `OptSignal::RegenerateSignalString()`.

3. Finally we need to output the statistics of the received current. To do this we first call `ElecSignalStat::GetElecSignalStat()` and if you are computing a 2D contour eye diagram you will also need to call `Histogram2D::WriteHistogram(string OutFileName)`. Second, call any of the `Get...` functions decribed above.

1.1 SURGEON GENERAL'S WARNING

Using Monte Carlo simulations to accumulate statistics of the system performance by obtaining many different random realizations of the system parameters is "*an extremely bad method. It should only be used when all alternative methods are worse.*" (Sokal, 1997). Although it is easy to understand and to code up, it converges slowly[4] and does not necessarily help us gain a fundamental understanding of the system.

1.2 Clock recovery

In the code we use two different types of clock recovery.
For NRZ we use `TypeClockRecovery = FREQ_DOUBLING`
For all other formats we use `TypeClockRecovery = RZ_SIGNAL_FREQ`. Ivan Lima and John Zweck tried various different methods for recovering the clock. The ones that are used by default in the code were found to work the best of the methods we tried.

The clock recovery is done in the method `double ElecSignalStat::GetTargetTime()` as follows. The simplistic idea is that the clock recovery time should be determined by the phase of the electric current at the frequency given by the bit rate.

[4]The error in the average of a quantity that is computed using Monte Carlo converges to zero no faster than $C/\sqrt{N}$, where N is the number of samples, and C is a problem dependent quantity. This fact follows from the Central Limit Theorem in probability theory.

1. Call `int ElecSignalStat::GetTargetFreqIndex()` which returns the index (array address) for the frequency in the power spectrum of the received electrical current that corresponds to the bit rate.

2. Call `PhaseTargetFreq = ComputePhaseTargetFreq()`.
   For `RZ_SIGNAL_FREQ` this method returns the phase of the power spectrum of the received electrical signal at the frequency given by the bit rate.
   For NRZ signals the power at the bit rate is often small and can be dominated by noise. This makes it hard to recover the clock. So instead for `FREQ_DOUBLING` we compute a new current that is equal to square of the difference of the current and a replica of the current that is delayed by half a bit period. If you start with a rectangular NRZ signal, the resulting current is another rectangular signal which is non-zero in the half-bit slots immediately prior to a transition from a mark to a space and from a space to a mark. This new signal looks like an RZ signal and typically has a much stronger power at the frequency given by the bit rate than the original NRZ signal did. We compute the phase of this current at the frequency given by the bit rate and use that to help us reciver the clock.

3. Given the phase of the received current at the target frequency we can then compute the clock recovery time using the method `double ElecSignalStat::GetTargetTime(void)`.

4. Brian found that it is very important to compute this time as a continuous quantity. It is *not* enough to compute the discrete index of the array that gives electrical current as a function of time — you won't get enough accuracy by doing that. Instead we compute the continuous clock recovery time and then shift the current in time by the `TargetTime`, so that the clock time now lies on the discrete time grid. Note that Brian still has not completely incorporated this improvement into the code.

1.3 Computing the statistics of the marks and spaces

To compute the *Q*-factor or the eye opening[5] we need to compute the means and standard deviations of the currents at clock recovery time as well as the maximum and minimum currents in the marks and spaces at the clock recovery time.

Once we have the clock recovery time we can form an array `BitIntensity` that consists of the received electrical currents at the clock recovery time in each of the bits.

To compute the statistics of the marks and spaces we need to know which entries of the `BitIntensity` array correspond to transmitted marks and which to transmitted spaces.[6] Because we know the original `BitString` in the transmitter we just need to work out the offset between the transmitted `BitString` array and the received `BitIntensity` array that would be required to line the `BitString` up with the `BitIntensity` array. Ivan Lima devised the following

[5] In the code rather than computing the eye opening per-se we compute the minimum amplitude margin `MinAmpMargin` which is defined to be the difference of the currents in the lowest mark and highest space at the clock recovery time. This is not necessarily the same as the eye opening, since you could imagine a situation in which some marks got changed to spaces and vice versa. In that case the eye would still look open but the `MinAmpMargin` would be negative!

[6] In an experiment a bit-error rate tester would work this out.

method to compute this offset. We do a `for` loop over all possible offsets and compute the *Q*-factor for each offset. We then choose the offset that gives us the maximum *Q*-factor. We have found that this method works very well.

Finally, we can now easily update the variables that are used to compute the statistics of the marks and spaces. Note that in a Monte Carlo simulation, for each system realization we repeat the entire proceedure above, and get a possibly different clock recovery time for each system realization. Outside the `for` loop for the Monte Carlo realizations we can call the method `double ElecSignalStat::GetClockTime(void)` which returns the average clock recovery time, averaged over all Monte Carlo realizations.

2 Semi-analytical receiver model

The semi-analytical receiver model is based on work of Peter Winzer *et al.*. Winzer's model has been extended in several ways in a series of papers from our group by Ivan Lima, Aurenice Lima, Yu Sun, Hua Jiao, John Zweck and Curtis Menyuk. The paper *"Performance characterization of chirped return-to-zero modulation format using an accurate receiver model"* which will soon appear in PTL explains the basic theory and gives references to Winzer's work. You can find it on the course web page. The inputs to this model are the noise-free signal, the noise spectral density, and the shapes of the optical and electrical filters in the receiver. The output consists of the means and standard deviations of the received electrical current at a user-specified time t. [7] The main assumption of the model is that the optical noise is additive white Gaussian noise. For this discussion let us assume that the signal is polarized and the noise is unpolarized.[8] The statistics can be used in various ways.

The statistics are computed analytically. No Monte Carlo simulations are required. Consequently the method is much faster and much more accurate than Monte Carlo simulations of a system in which the optical noise is additive white Gaussian.

To use the model as implemented in the OCS code you can do the following.

1. Pass the optical signal `Signal` through all elements of the system up to but not including optical filter. Do *not* include noise in this simulation!!

2. Construct `OptFilter, Photodetector, ElecFilter`, and `ElecSignalStat` objects as described above.

3. Call `Stats.SetOptElecFiltersForSemiAnalyticalStatistics (&oOptFilter,&oElecFilter);`, where `Stats` is the ElecSignalStat object constructed above.

4. Compute the `TotalNoiseSpectralDensity` at the receiver somehow. This is the sum of the noise spectral densities in the X- and Y-polarizations of the Jones space representation of the signal.

[7] The model also computes the *Q*-factor from the OSNR, but I'm not focusing on that aspect here.

[8] The model can also deal with partially polarized noise.

5. To compute the mean received current at a time t = `jj*Signal.GetDeltaTime()` with time index `int jj` add the value of `real(oPhotodetector.sfftPM.tPowerSignal[jj])` to `Stats.GetMeanASE_ASE(TotalNoiseSpectralDensity)`.

6. Compute the time-independent quantity
   `double StdDev_ASE_ASE = Stats.GetStdDevASE_ASE(TotalNoiseSpectralDensity)`

7. Compute the time-dependent quantity
   `StdDev_Signal_ASE[jj] = Stats.GetStdDevSignal_ASE_TimeIndex(jj, 0.5*TotalNoiseSpectralDensity,0.5*TotalNoiseSpectralDensity)`

8. The standard deviation of the current at time t = is then given by
   `sqrt(sq(StdDev_ASE_ASE) + sq(StdDev_Signal_ASE[jj]))` .

Using this receiver model I have computed semi-analytical eye diagrams for a long-haul transmission system assuming that the noise is unpolarized additive white Gaussian white and that the signal is polarized. The result is shown in Fig. 1. The agreement on a linear scale with the eye in Fig. 2 obtained from Monte Carlo simulations with 10,000 realizations of a PRBS of length 32 in which noise is added just prior to the receiver is excellent. Even if we add noise at each of the amplifiers which can then propagate and interact nonlinearly with the signal the agreement looks very good on a linear scale, see Fig. 3.

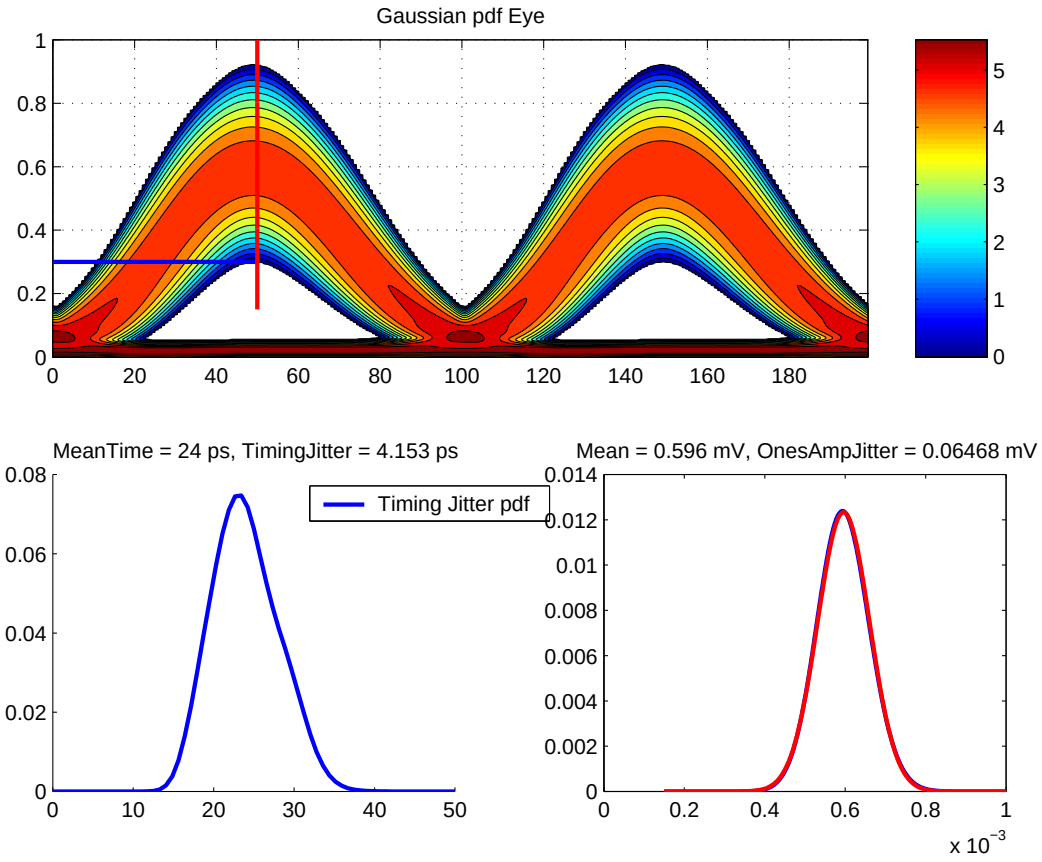

Figure 1: Semi-analytical eye

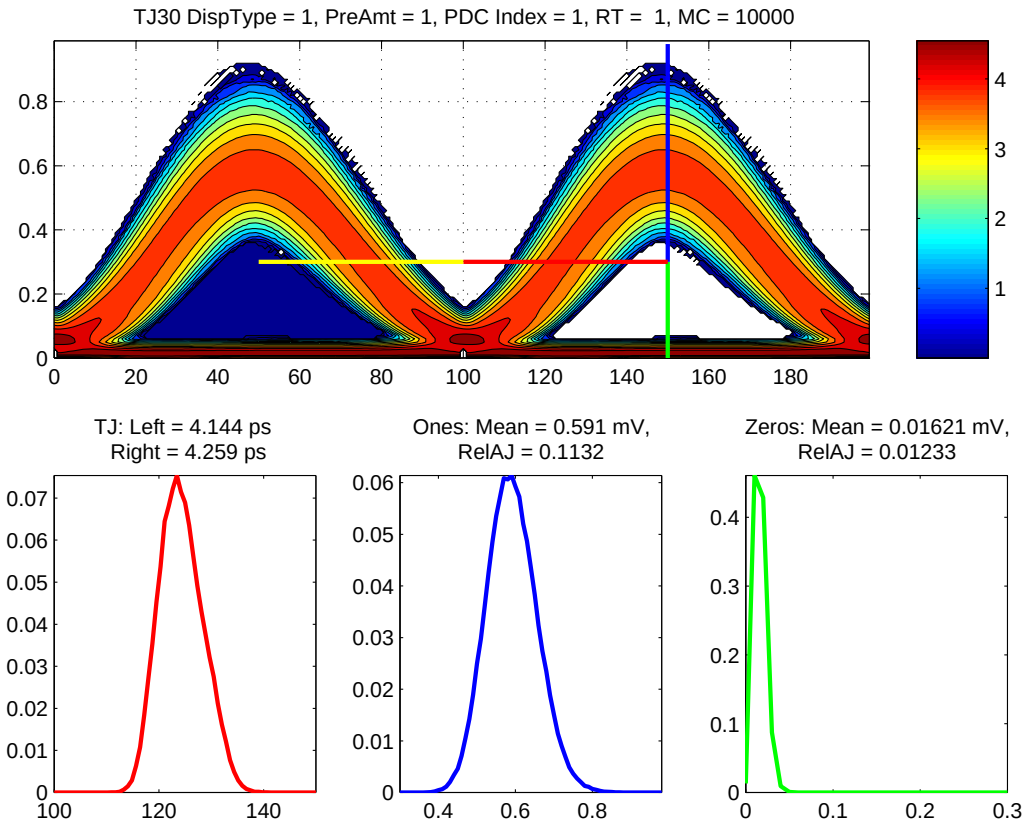

Figure 2: Monte Carlo with AWGN added just prior to receiver

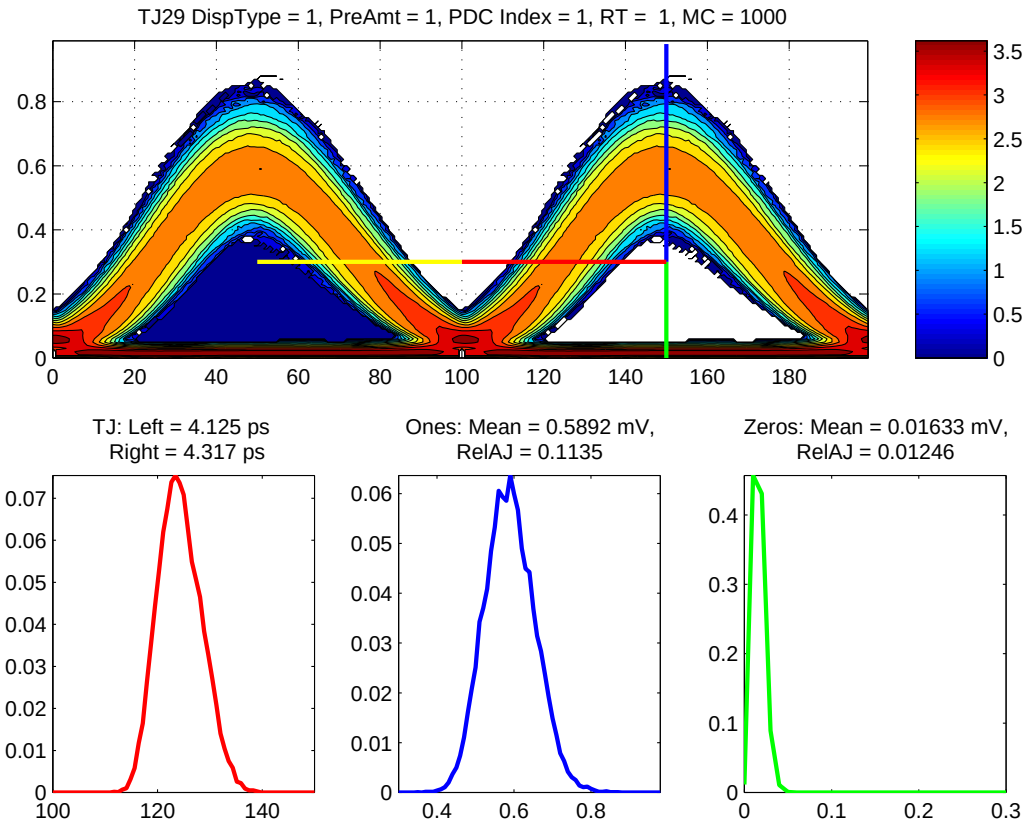

Figure 3: Monte Carlo with noise added in-line