OCS Implementation of Amplifiers

April 1, 2003

1 Overview of the lecture

This lecture is about how we have implemented erbium-doped fiber amplifiers in our OCS code. The topics we will discuss are:

1. The random number generator constructor, the amplifier constructor, and the parameters in the amplifier input file
2. The OCS amplifier types and implementations
3. The OCS amplifier noise types and implementations
4. Monte Carlo simulation (briefly)

2 Amplifier constructor and the input file

You already implemented an amplifier in the Tyco simulation you did in the previous homework. The amplifier we used in the homework is *very* simple, in that it simply multiplies the power of the signal by a gain value specified in the amplifier's input file. The gain value was chosen to exactly compensate the loss in the fibers during transmission. Although this is a simple model, it is quite useful in many cases.

In the OCS code, the option for this type of amplifier is called `SCALAR_SIMPLE` for the scalar (polarization-preserving) model of the system, and `VECTOR_SIMPLE` for the vector (polarization-dependent) model. Also, in your homework, no noise was added at the amplifiers. That is, in the input file for the amplifiers, `$TypeAmplifier-Noise` was set to 0, corresponding to `NOISE_OFF` in the code.

Of course, amplifiers always add some amount of random noise to the signal. When modeling amplifiers using OCS, you must first call a constructor for a random number generator (RNG), whether modeling noise or not. The RNG is used to model the random noise process, but it must be set up first whether the noise is turned off or not. In the Tyco system, it was implemented in the following line of C++ code:

```
RanNumGen RNGAmps(InDir + "RanNumGenAmps.in");
```

More about the RanNumGenAmps.in input file later, but for now, we have called an instance of a RNG called RNGAmps for use with the amplifiers. You will notice that we also set up an RNGSignal and RNGFibers in the code as well for random behavior in the signal bit string and in the polarization properties of the fiber as well.

Similarly, in OCS, you must also first call the constructor for the optical signal prior to calling the amplifier constructor. The code must know what field it is applying gain and noise to. In the Tyco system, this constructor was called by the following line of C++ code:

```
Signal = new OptSignal("Signal.in",&RNGSignal);
```

The constructor for the amplifier is similar, and its arguments are the input file describing the amplifier, the pointer to the signal object it acts on, and the pointer to the RNG:

```
OptAmplifier Amp(InDir + "Amp.in",Signal,&RNGAmps);
```

The amplifier input file has a bunch of parameters of the amplifier in it. The input files we used for the Tyco system should be a good template for whatever you want to do. First, the parameter `$TypeAmplifier` tells the code what type of amplifier we are using. See the file ocsOptAmplifier.hh for more information about the different amplifier types and some documentation about how to use the class. Here are the possible types of amplifiers implemented in the code so far:

`$TypeAmplifer`	=	0	NO_AMPLIFICATION
	=	1	SCALAR_SIMPLE
	=	2	SCALAR_SATURATED
	=	3	VECTOR_SIMPLE
	=	4	VECTOR_SATURATED
	=	5	VECTOR_FIXED_OUTPUT_POWER

Adding noise to the amplifier is straightforward by changing the value of `$TypeAmplifierNoise` in the input file. The possible values are

`$TypeAmplifierNoise`	=	0	NOISE_OFF
	=	1	NOISE_ON_CONST_POWER_RANDOM_PHASE
	=	2	NOISE_ON_GAUSSIAN_WHITE
	=	3	NOISE_ON_SEMI_ANALYTICAL

I will discuss more about the noise implementations later.

The rest of the parameters in the input file for the amplifiers depend on the choices you make for the amplifer type and the noise simulation type.

If one of the SIMPLE gain type amplifiers is used, you only need to specify the gain of the amplifier in `$Gain-OptAmplifier` in dB.

For simulations with noise turned on, one must specify either the amplifier's $n_{\rm sp}$, `$SpontEmissionFactor`, or the amplifier's noise figure, `$NoiseFigOptAmplifier`. The other parameter should be set to 0 so that the code knows which one to take. If both are non-zero, an error is generated when you run the code.

For simulations using the vector model, the polarization-dependent gain, `$PolDepGainOptAmplifier` of the amplifier may be given in dB in the input file. Alternatively, of course, you can set this to 0 dB so that you have no PDG.

For simulations with gain saturation in the EDFAs, several parameters must be set. Recall from the last lecture that a simple gain saturation model is given by the solution of the ordinary differential equation

$$\frac{dP}{dz} = \frac{g_0 P(z)}{1 + (T_{\rm amp}/U_{\rm sat})P(z)}. \tag{1}$$

The value `$NumZSteps` tells the code how many steps along the erbium-doped fiber must be taken in the simple gain saturation model. In OCS, we use an Euler method to solve the ODE for gain saturation, and the number of steps taken should be fairly large, say about 100, so that we get a good numerical solution for the correct saturated gain. The saturation power, `$SaturatingPowerdBm`, is the ratio $U_{\rm sat}/T_{\rm amp}$, expressed in dBm. Finally, the small-signal gain, `$UnsaturatedGaindB`, of the amplifier should be specified in dB. The small-signal gain in an amplifier of length L is related to the parameter g_0 by the equation $G_0 = \exp(g_0 L)$.

The amplifier type that uses the fixed output power is used to pin the amplifier's output power to a particular level. I will not go into how this option works in this lecture. See the code for further details on this model.

As yet, we do not have an amplifier model that solves the rate equations in the ocsOptAmplifier class, but Jonathan Hu has implemented this using measured data for the absorption and emission spectra of the erbium-doped fibers we use in experiments.

3 Random number generation

Random number generators are commonly used in computer science and applied math in many applications. In fact, many compilers have built-in random number generators (in C/C++, it's the `rand()` function), but these tend to be very simple implementations. Much better generators exist, and for a reasonably complete discussion about them (as well as why you should *never* use the built-in versions), read one of the *Numerical Recipes* books by Press, et al., about the three or four generators they suggest. The first thing to know about random number generators on computers is that, for *most* of them, they're not really random! In fact, most "random" number generators are periodic, with a very large period, and good decorrelation between any two numbers within the period. There are some aperiodic exceptions based on using the system clock or, in one extreme case, using the digitized image of a lava lamp [see "Lava Lamp Randomness" in Science News, **159** (19), 2001], but most random number generators produce a periodic sequence.

In OCS, we use the `ran2` routine from *Numerical Recipes*, which has a very large period of greater than 2×10^{18} and very good autocorrelation behavior. The authors even advertise a $1000 prize for anyone who finds a statistical test it fails.

The amplifier code uses the member functions of the random number generator class, `GetRanNum()` and `GetGaussianDeviate()`, which generate uniformly-distributed and Gaussian-distributed random numbers, respectively. Depending on the noise model used, either of these functions may be called to generate the effect of randomness in the noise on the signal.

Random number generators typically rely on a *seed* (or a set of seeds), provided by the user, that corresponds to one particular number output from the random number generator. The sequence of generated random numbers that follows is *always the same* for the same seed value. Therefore, if you want to run a simulation with a random process, you need to reset the seed for each simulation or you will get *exactly* the same result. Note that sometimes this is desirable! We have some automation of the process of resetting the seed available in the OCS code, that can be set in the random number generator input file.

In the Tyco simulation, in the RanNumGenAmps.in file, there are parameters `$OperationMode` and `$Seed`. The value of `$OperationMode` is described in the input file:

`$OperationMode`	=	1	Initialize using `$Seed`
	=	2	Continue random sequence from the previous run using seed values in the XXX.continue file
	=	3	Restart the random sequence where the previous run started using the seed values in the XXX.restart file

First note that you should *never* edit the XXX.continue and XXX.restart files. Let the code do that for you. Do not touch them.

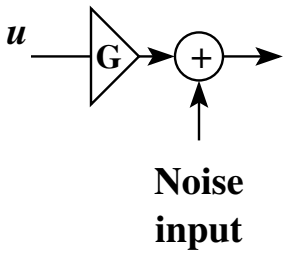

Figure 1: Schematic of noise addition in the OCS code.

When the class constructor is called, it figures out the state you want based on the operation mode you specified in the file, and then it writes the information for that state to the XXX.restart file in case you want to come back to that state. When the class destructor is called, OCS writes the seed information for the stopping point in the random number sequence to the XXX.continue file so that you can start at that point the next time you run your code.

4 Amplifier noise modeling in OCS

The first two of the noise models in the code, NOISE_ON_CONST_POWER_RANDOM_PHASE and NOISE_ON_-GAUSSIAN_WHITE, are the two models I will discuss here. For more information on the final method, see the code. In the first case, a constant noise power is added to the signal at the amplifier, and then the phase of each Fourier component is randomized in a uniform way. In the second case, a Gaussian-distributed white noise random variable is added to the real and imaginary parts of each Fourier component of the signal at the amplifier. The second case is a more physical model, but the first case is equivalent to the second when transmitting through a cascade of optical amplifiers.

The amount of noise added to the signal is determined in the code by the private variable `NoiseAmplitude-Factor` using one of the input parameters `$SpontEmissionFactor` or `$NoiseFigOptAmplifier` (the other one is set to 0). In the code, we define

$$\texttt{NoiseAmplitudeFactor} = \left[h n_{\rm sp} \Delta\nu (G-1)\right]^{1/2}, \tag{2}$$

when $n_{\rm sp}$ is given through `$SpontEmissionFactor` and

$$\texttt{NoiseAmplitudeFactor} = \left[\frac{h}{2}\,\mathrm{NF}\,\Delta\nu (G-1)\right]^{1/2}, \tag{3}$$

when the noise figure, `$NoiseFigOptAmplifier`, is specified. In the above, h is Planck's constant, $\Delta\nu$ is the frequency spacing in the FFT that we are using, and G is the amplifier gain on a linear scale (i.e., *not* in dB). The relationship between $n_{\rm sp}$ and noise figure given above assumes that we are in a high-gain regime, which is applicable for many modern-day communication systems. See Desurvire's book on EDFAs for more information. The square root is used because we add noise to the *electric field* rather than the *power*.

Figure 1 shows an illustration of how the noise addition is done at the amplifier. The signal is first amplified, and then the noise is applied. In the case of the constant noise power with a randomized phase, one adds `NoiseAmplitudeFactor` $\times\sqrt{\nu}\exp(i\phi)$ to each complex Fourier component, where ϕ is a uniformly-distributed random variable between 0 and 2π. In the case of Gaussian white noise, one adds `NoiseAmplitudeFactor` $\times\sqrt{\nu/2}\times Y$ where Y is a Gaussian-distributed random variable with zero mean and standard deviation of 1 to the real and imaginary parts of each Fourier component.

5 Monte Carlo simulations

In a Monte Carlo simulation, one uses the random number generator to produce many *realizations* of the noise in the amplifiers (or some other random process) in order to determine the impact of the randomization on some effect *on average*. As an example, one can use a Monte Carlo simulation to determine the average eye opening penalty due to amplifier noise, and with enough realizations, one can get an estimate for the variance, the skew, and other higher-order moments of the eye opening, which is itself a random process. With enough realizations, one can build a histogram of the simulated data, and from this, generate a probability density function (pdf) for any system property that depends on the random processes in the system.

This is a *very slow process*, and can often take *millions or billions* of realizations (or more!) to acquire meaningful results. In fact, it is truly impossible to evaluate bit error rates in this way. However, OCS has many functions that can aid in collecting statistics when performing Monte Carlo simulations and building pdfs. The ocsHistogram class can be used in such a way. See the ocsHistogram.hh and ocsHistogram.cc files for some information on what functionality is available.