A Mathematica-Based Approach to Designing Receiver and Transmitter Gain Tables

Jesse E. Chen
Senior Staff Engineer, Qualcomm Campbell, CA
jessec@qualcomm.com

ABSTRACT

This paper describes a few intrinsic Mathematica functions particularly well suited for designing a gain line up by exhaustively searching through all possible gain states of a given architecture. The key Mathematica function is the "Outer" function. Armed with the proper performance functions and input data, the Outer function fully characterizes a given architecture in one command. Since the arguments and operators for the Outer function are lists, the approach is extremely flexible. For example, one can quickly assess architectural changes simply by manipulating lists.

1. INTRODUCTION

Receiver and transmitter chains attain wide dynamic ranges by including one or more programmable gain blocks. For any power level within the specified dynamic range, the receiver/transmitter must find a gain distribution (i.e. gain state) that produces the desired power level with an acceptable amount of overall impairment. Intermediate values of composite gain can almost always be achieved by multiple gains states. I steal terminology from quantum physics to define gain states with the same composite gain (within some tolerance) as degenerate gain states. Although degenerate gain states have the same gain, they do not perform the same; different degenerate gain states have different degrees of impairments. A gain table describes how gain is distributed along a receiver/transmitter chain for each input/output power level in the desired dynamic range.

The traditional RF systems tool is the spreadsheet [1, 2]. Spreadsheets are interactive, easy to use, and inexpensive. The classic RF system spreadsheet analyzes one gain state at a time with a level diagram [3]. A level diagram shows the power levels of the desired signal, noise floor, compression point, and for receivers, perhaps a few undesired signals (blockers) along the chain. The level diagram can be enhanced to display curves for multiple gain states but the best gain state for each input power level is not immediately apparent. Furthermore, architectural changes can require major spreadsheet carpentry. Spreadsheets are justifiably the main work horse for designing RF systems but they can quickly become unwieldy when trying to compare various architectures over the entire dynamic range.

This paper presents an approach to gain table design based on Mathematica [4]. Mathematica is a programming language with extensive graphics and word processing features. Mathematica includes a rich vocabulary of functions extremely well suited to RF systems analysis. The resulting code is concise, easily documented, and easily maintained. The problem with Mathematica is that the key functions do not leap out from the rather formidable four-inch manual. In fairness to Mathematica, the manual is thick because Mathematica is capable of much more than RF systems analysis. This paper highlights the small subset of Mathematica functions particularly well suited to RF systems analysis.

I explain the Mathematica approach to gain table design primarily through a receiver example. However, I also briefly mention a transmitter application. The design objective for the receiver example is to convert a set of standard RF specifications for each gain state of each block in the chain into a table relating input power to gain state.

The first step is the key to obtaining a gain table for a given architecture that is "best" in some sense. The first step characterizes all possible gain states. This approach differs significantly from an evolutionary algorithm [5]. An evolutionary algorithm involves new generations, mutations, and natural selection. The approach discussed below computes all "mutations" up front in one generation and then sequentially weeds out all but the best states; there are no new generations. I characterize all possible gain states with Mathematica's Outer function. The Outer function generates a list of composite RF metrics for every possible gain state. The list of metrics includes composite gain, noise figure, and compression point over several frequencies. The list also includes a label. The label is an important diagnostic tool because it records the gain state of each block. The label could be the actual digital command for that gain state. The compression point frequencies correspond to the in-band signal and several blockers. Regardless of the blocker problem (desensitization, reciprocal mixing, etc.), I assume the severity of the problem increases with decreasing backoff from the compression point and that the backoff pass/fail threshold for each blocker frequency is known.

With all possible states of the candidate architecture fully characterized, I need only weed out unfit and/or inferior states to reveal the best states. The surviving states are "best" by definition because we started with all possible states.

The paper is organized as follows: Section 2 explains the details of the weeding process. Section 3 describes the inputs to the design process. Section 4 describes the state function. The state function operates on a single composite state to generate end-to-end specifications. Section 5 describes the key Mathematica functions, including the "Outer" and "FoldList" functions. These two key functions appear to be unique to Mathematica. Section 6 makes some concluding remarks.

2. THE WEEDING PROCESS

This section assumes all possible states have already been fully characterized and sorted into gain bins. Subsequent sections describe how to characterize all possible gain states. In the following example, the gain bins are 2dB wide.

Figure 1 shows the input referred target output (IRTO) and the input referred minimum detectable signal (MDS) versus gain for all possible gain states. The horizontal axis is the gain of each bin. The vertical axis is input referred power. The IRTO equals the desired receiver output in dBm, minus the composite gain. In this example the IRTO points lie on a straight line with unity slope because the target output is constant. The MDS equals the noise floor plus the minimum required signal-to-noise ratio (SNR). The receiver chain had five cascaded blocks. The blocks had 2, 2, 13, 2, and 13 gain states respectively, giving a total of 1352 possible gain states.

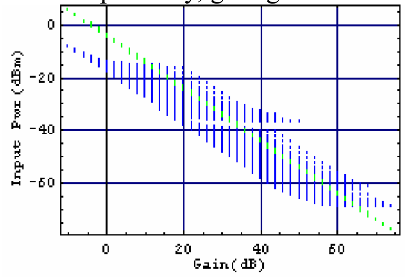

Figure 1 shows input power levels and MDS points for all possible gain states.

The first weeding process removes gain states for which the IRTO lies below the MDS. Figure 2 shows the surviving 1006 gain states.

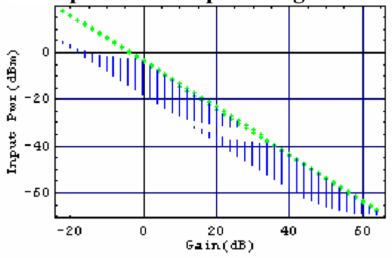

Figure 2 shows the states surviving screening for minimum SNR.

The next weeding process removes gain states violating in-band linearity. Figure 3 shows the IRTO and the input referred maximum allowable signal (MAS) for all possible gain states. The MAS equals the in-band one dB compression point minus the minimum required backoff. I weed out gain states for which the IRTO lies above the MAS.

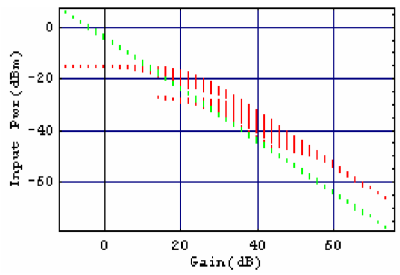

Figure 3. Some states violate backoff.

Figure 4 shows the IRTO, MAS, and MDS points for the 756 gain states surviving the first two weeding processes. Note that there are no points for composite gains below about 12 dB. Figure 3 shows why: for gains below 12dB, the MAS points lie below the IRTO points.

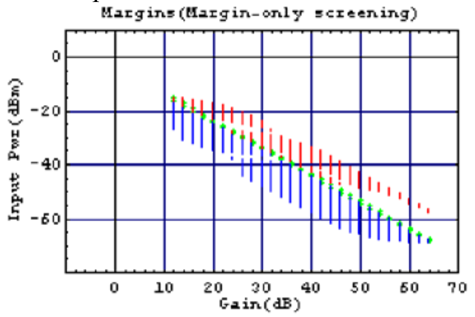

Figure 4. 756 of the 1352 original gain states survive margin weeding.

The next weeding process checks for blocker violations. The compression point and backoff depend on blocker frequency. A blocker violation occurs when two things happen: 1) the blocker power at the LNA input exceeds the one dB compression point minus the specified minimum backoff; 2) the computed desired input signal level lies above the desired signal level specified for blocker testing. For example, suppose:

1. IRTO equals -70dBm;
2. The specified input signal power during blocker testing equals -60dBm.
3. the blocker at an offset frequency of 30MHz equals -40dBm;
4. The input referred compression point at 30MHz offset is -35dBm.
5. The required backoff is 10dB.

In this case, weeding for blocker performance would not weed out the gain state because the IRTO power (-70dBm) lies below the input signal power specified for the test (-60dBm); the test does not apply here. If we drop the power listed in item 2 to -70dBm, the blocker test would apply and it would remove this gain state because -35dBm -10dB = -45dBm < - 40dBm. Getting back to the main example, all states passed the blocker tests.

The last weeding process removes any remaining degeneracy by removing all but the best state within each gain bin. This step requires a definition of "best". For receivers I apply two definitions, one based on lowest noise figure and one based on highest in-band compression point. Figure 5 shows the least noisy gain states while Figure 6 shows the most linear gain states.

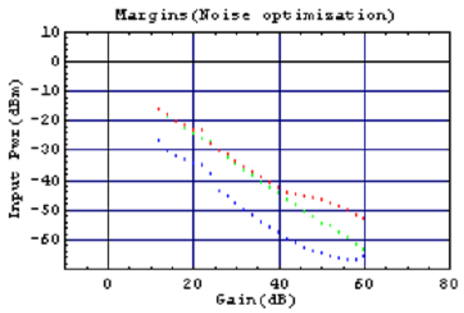

Figure 5. Only 25 states survive noise optimization.

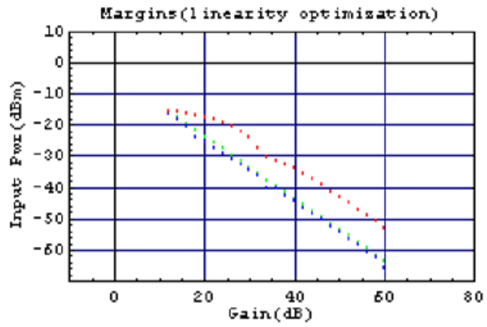

Figure 6. Degeneracy can also be removed through linearity optimization.

Figure 7 shows similar results for a transmitter. The vertical axis in Figure 7 is normalized output power instead of input power. The lower curve is the noise floor plus the minimum required SNR. The upper curve shows the output compression point minus the minimum required backoff from the one dB compression point. The center curve shows the output power. Here, the input power is fixed and the output power varies with gain. In this example I also applied noise and linearity margins but I optimized for current. For each gain state of each block in the chain, I added current consumption to the list of specifications. I also added a function to the list of state function to compute total current consumption of the chain. After weeding for noise and linearity margins, I remove the remaining degeneracy in each gain bin by selecting the state with minimum total current. Figure 8 shows the resulting current as a function of output power.

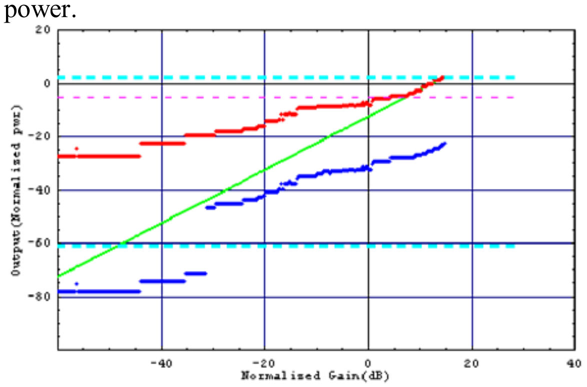

Figure 7. The method can also be applied to transmitters.

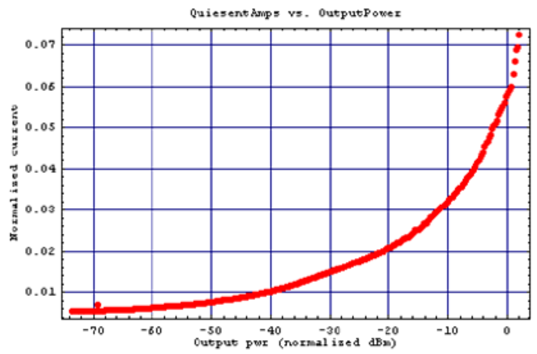

Figure 8. This transmitter was optimized for total current consumption.

3. BLOCK LEVEL INPUTS

This section describes the structure of the input data. The block level inputs are specified by lists of lists with the format shown in listing 1. Listing 1 is pseudo-code. In Mathematica, a list is a sequence of comma-separated elements enclosed in curly brackets.

Listing 1. Block Specifications

```
Block-n=
        {List-for-gain-state-1,
        List-for-gain-state-2,
        etc};

List-for-gain-state-k =
        {Gain-list,
        Noise figure,
        Compression-point-list,
        Label};

Gain-list=
        {In-band gain,
        Gain @ Blocker 1,
        Gain @ Blocker 2,
        Etc.}

Noise figure  = scalar, not a list, no curly brackets.

Compression-point-list=
        {In-band  compression point (i.e. Pi1dB),
        Pi1dB @ Blocker 1,
        Pi1dB @ Blocker 2,
        Etc.}

Label="010", for example.
```

To add more specifications to the analysis just expand the appropriate list. Since the specifications are referenced by their position in a list, you can minimize the amount of code you must change to accommodate new metrics by always adding to the end of the lists and never referencing any item with Mathematica's "Last" function. The order of these lists is arbitrary but once set the order must be maintained because everything else depends on it.

4.0 THE STATE FUNCTION

The state function operates on one state from each block to characterize end-to-end performance of that composite gain state. The state function is a list of multi-input functions that produces a list of input referred end-to-end performance metrics. Listing 2 shows the structure of the state function for our receiver example.

Listing 2. State Function

```
State function =
{Composite gain,
Composite noise figure,
List of composite compression points,
Composite label or gain command,
In-band signal level along the chain,
Blocker levels along he chain,
Gain distribution}
```

The argument to each function within state function is also list. Listing 3 shows one such argument; blocks 1,2,3,4 and 5 are in states 2, 1, 9, 1, and 6 respectively using the notation from Listing 1.

Listing 3. Function arguments.

```
State function argument =
{Block1.list-for-gain-state-2,
Block2.list-for-gain-state-1,
Block3.list-for-gain-state-9,
Block4.list-for-gain state-1,
Block5.list-for-gain-state-6}
```

To add another end-to-end performance metric to the analysis, just add the appropriate function to the state function.

5.0 KEY MATHEMATICA FUNCTIONS

I use several Mathematica functions but two functions stand out because they appear to be extremely well suited to RF systems analysis and unique to Mathematica. These are the "FoldList" and "Outer" functions which I describe in this section.

5.1 The FoldList Function

Most if not all end-to-end metrics of a receiver or transmitter chain are conveniently expressed as a recursive calculation. The Friis formula [2,3] for the cascaded noise figure is a good example. The FoldList function performs recursive calculations. The FoldList function takes three arguments: a function, a list, and an initial element. The FoldList function starts with the initial element and recursively applies the function to the list. The functional argument itself has two arguments, usually the previous output of the function and a new input. FoldList is best explained through example.

Suppose we are only interested in composite gain and label for a 10-block chain. For simplicity, assume each gain is a scalar (instead of a list). Let the input architectural data be as shown in Listing 4. Gains are in dB.

Listing 4. Simplified State Function Input

```
chain={{1,a},{2,b},{3,c},{4,d},{5,e},{6,f},{7,g},{8,h},{9,i},{10,j}};
```

The composite gain equals the sum of block gains, which equals 55 in this case. The composite label is "abcdefghij". Listing 5 shows the FoldList command to compute the composite gain and label, along with the result. The line with the "#" signs is what Mathematica calls a "pure function". This particular pure function takes two inputs (for example, {1,"a"} and {2,"b"}). This pure function also has two outputs. The first output equals the sum of the first components of the two inputs; "#1[[1]]" is the first component of the first input, which equals 1 in this case, and "#2[[1]]" equals the first component of the second input, or 2 in this case. The second output is the concatenation of the two labels, which equals "ab" for the two inputs above. The second output operates on the second component of each of the two inputs; it concatenates #1[[2]] and #2[[2]].

Note that the FoldList function computes all intermediate results too. Intermediate results are important when computing distributed gain, signals along the chain, or cumulative compression points and noise figures.

Listing 5. Application of FoldList.

```
FoldList[
        {#1[[1]]+#2[[1]],StringJoin[#1[[2]],#2[[2]]]}&,
        First[chain],Rest[chain]
]

(*Result*)
{{1,a},{3,ab},{6,abc},{10,abcd},{15,abcde},{21,abcdef},{28,abcdefg},{
36,abcdefgh},{45,abcdefghi},{55,abcdefghij}}
```

If you don't want the intermediate results, the "Fold" function works like the FoldList function except that it outputs only the last result. However, intermediate results make quick work of level diagrams. Figure 9 shows the signal levels along the chain for all 25 gain states as well as the level of a blocker all along the chain. The curves converging at the right show desired signal over the dynamic range hitting a fixed target output level. The curves diverging from the left show the blocker level along the chain for the 25 surviving gain states. (In reality, the blocker would likely start out larger than the desired signal and cross over the desired signal after passing through a filter stage. I chose the low blocker level just to make the two families of curves easily discernable.)

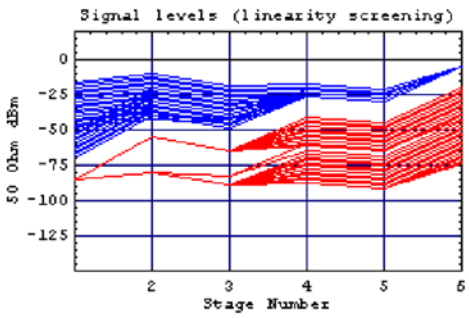

Figure 9. Intermediate FoldList results produce level diagrams.

5.2 The Outer Function

The Outer function is the central function of this paper. The Outer function generates a list of all possible gain states along with all metrics of interest for each gain state. The Outer function is a general form of the conventional outer product function, or dyad. Consider the conventional dyad of two vectors, {1,2} and {3,4}. Listing 6 shows how to generate the dyad using Mathematica's Outer function. Listing 6 also shows that we can chose a different operator than "Times", such as the "Plus" operator.

Listing 6. Simple Application of the Outer Function.

```
Outer[Times,{1,2},{3,4}]

(*Result*)
{{3,4},{6,8}}

Outer[Plus,{1,2},{3,4}]

(*Result*)
{{4,5},{5,6}}
```

I can also apply the Outer function to more than two lists at a time. At this point the output is easier to read if we "flatten" it to express it as a single list (only one set of curly brackets). Listing 7 shows how to use the Outer function to compute all possible combinations of three vectors representing three bits.

Listing 7. Applying the outer function to more than two inputs.

```
Flatten[Outer[StringJoin,{"0","1"},{"0","1"},{"0","1"}],2]

(*Result*)
{000,001,010,011,100,101,110,111}
```

The functional argument to the Outer function, which was "StringJoin" in listing 7, is not limited to single-input-single output functions. We can use multi-input-multi-output functions. I expand on the example in listing 7 to illustrate that point. The input lists in listing 7 were strings ("0" and "1"). Let's add variables to the input data that represent numbers. Suppose we want to sum the numbers for each combination. Listing 8 shows the code and the results.

Listing 8. A simple multi-input-multi-output example.

```
Flatten[Outer[{#1[[1]]+#2[[1]]+#3[[1]],StringJoin[#1[[2]],#2[[2]],#3[[2]]
]}&,{{a,"0"},{b,"1"}},{{c,"0"},{d,"1"}},{{e,"0"},{f,"1"}},1],2]

(*Results*)
{{a+c+e,000},{a+c+f,001},{a+d+e,010},{a+d+f,011},{b+c+e,100},{b+
c+f,101},{b+d+e,110},{b+d+f,111}}
```

If we assume {a,"0"} represents the gain and label the first gain state of the first block, and {b,"1"} does the same for the second gain state of the first block, {c,"0"} does the same the first gain state of the second block and so on, the results in listing 8 show that the Outer function computed the composite gain and label for all possible gain states of this simple 3-block example of a chain..

To characterize all possible gain states of a real receiver chain, I replace the multi-input-multi-output pure function in listing 8 with the state function of section 4 and I replace the input list with a list of elements as described in section 3.

5.3 A Few Other Handy Functions

This section explains a few details that make the overall approach more flexible. In listing 8 I called out the arguments explicitly. For example, to sum the gains I used the pure function #1[[1]]+#2[[1]]+#3[[1]]. This approach is not flexible because I must keep track of the number of blocks in the chain. Listing 9 shows the commands to define the complete state function, specify the receiver chain, and characterize all possible gain states. (For brevity, I do not explicitly show the definitions of all individual functions within the state function or definitions of all the individual blocks.) For a particular end-to-end state, the "Composition" function in listing 9 combines the specifications for a particular state of each block into a single list before calling the state function (ChainMetrics). Passing the architectural data to the state function in this fashion saves me from having to modify all functions within the state function whenever the number of blocks changes; I can change the architecture just by adding or deleting elements in "AllChain" list.

Listing 9. The key commands.

```
(*Define the complete state function*)
Clear[ChainMetrics];
ChainMetrics[chain_]:=
 Module[{},
  {ChainGain[chain],
   ChainNF[chain],
   ChainPi1dB[chain],
   ChainCmd[chain],
   SigLevel[AdcTarget,chain],
   BlockerLevel[BlockerInputs,chain],
   GainDist[chain]}
  ]
```

```
(*Define the receiver chain*)
Clear[AllChain];
AllChain={FrntEnd,Filt1,Pga1,Filt2,Pga2};

(*Characterize all possible gain states*)
Clear[ChainData];
ChainData=
  Reverse[
   Sort[
    Flatten[
     Apply[Outer,
      Prepend[
       Append[AllChain,1],
       Composition[ChainMetrics,List]
       ]
      ],
     Length[AllChain]-1(*Level=Number of blocks-1*)
     ]
    ]
   ];
```

The "Reverse", "Sort", and "Flatten" functions merely rearrange the results of the Outer function. The "Apply", "Prepend", and "Append" functions assemble the input architectural data, the ChainMetrics function, and the word "Outer" into the proper syntax for the Outer function.

5.4 A Sample Element of the State Function

Listing 10 shows the first element of the state function. This function computes the end-to-end gain and then bins the result. The "Map" line creates a list of gains from the first element of each list of block level specifications. The "Total" line sums the gains from each block. The last line bins the data by rounding the gain to the nearest 2dB.

Listing 10. Sample element of the state function.

```
Clear[ChainGain];
ChainGain[chain_]:=
 Module[{gains,xx},
  gains=Map[First[#[[1]]]&,chain];
  xx=Total[gains];
  2 Round[xx/2.0]
  ]
```

5.5 A Sample Weeding Function

Listing 11 shows the code for selecting only those gain states that survive the first two weeding processes (noise and linearity margins). The "LimitFlags" function appends each state function output list with a True/False variable indicating whether the state violated (False) either the noise or linearity margins. The "MarginOnly" function selects those states for which the True/False variable is True and then drops the True/False variable.

Listing 11. Selection functions.

```
Clear[LimitFlags];
LimitFlags[ChainData_]:=
 Module[{},
  Map[Append[#,
     (AdcTarget-#[[1]]  #[[3,1]]-BackOff[[1]])&&
      (AdcTarget-#[[1]]  #[[2]]+NoiseInput+MinSnr)]&,
   ChainData]
  ]

Clear[MarginOnly];
MarginOnly[ChainData_]:=
 Module[{x,y},
  x=LimitFlags[ChainData];
  y=Select[x,Last[#]&];
  Map[Drop[#,-1]&,y]
  ]
```

6. CONCLUSION

Mathematica has a thick manual and a fairly steep learning curve but the small subset of functions described in this paper can make the investment worthwhile for RF systems engineers. In particular, the FoldList, and Outer functions allow the user to code up with just a few lines an algorithm to find optimal gain tables for a given architecture. The Mathematica approach to gain table design is very flexible because the user can modify the analysis simply by manipulating lists.

Although this paper did not discuss it, the function Outer can also be applied to frequency planning. For example, I have also used the Outer function to quickly generate all possible mixing spurs within a specified order and flag those spurs that fall into a vulnerable band.

ACKNOWLEDGEMENTS

I am grateful to several people for long discussions that influenced my work. Among them are Tim Nguyen, Lawrence Chang, Pengfei Zhang, Isaac Server, Patrick Vandenameele, and Anthony Tsangaropoulus.

REFERENCES

[1] Thomas R. Turlington, *Behavioral Modeling of Nonlinear RF and Microwave Devices*. Artech House.
[2] William F. Egan, *Practical RF Systems Design.* Wiley Interscience.
[3] Behzad Razavi, *RF Microelectronics*. Prentice Hall. 1998. First Edition.
[4] *Mathematica, http://www.wolfram.com/*
[5] http://en.wikipedia.org/wiki/Evolutionary_algorithm