AUTOMATIC SIGMA-DELTA ANALOG TO DIGITAL CONVERTER ARCHITECTURE GENERATOR

by

Ali Murat Gök

B.S., Electrical and Electronics Engineering, Boğaziçi University, 2010

Submitted to Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Electrical and Electronics Engineering
Boğaziçi University
2012

Dedicated to science…

ACKNOWLEDGEMENTS

I would like to thank Prof. Günhan Dündar for his help, support and patience throughout the thesis.

I would also like the thank Tayyar Oğuz Karaduman for his support and warning me when MATLAB crashes occur.

I would also like to thank Feyyaz Melih Akçakaya for his help during my thesis.

ABSTRACT

AUTOMATIC SIGMA-DELTA ANALOG TO DIGITAL CONVERTER ARCHITECTURE GENERATOR

Sigma-delta A/D converters constitute a popular method of A/D conversion since they can achieve higher resolution with simpler analog circuitry. Using feedback and feedforward paths, their performance can be increased even further. However there are quite numerous possible feedback and feedforward paths and it is an important concern to find the gain values of these paths. In this thesis, an automatic tool is presented to find the optimum Sigma-Delta A/D converter topologies for desired specifications. The tool is then used to find some desirable topologies for second order Sigma-Delta A/D converters. Some different approaches are needed in order to find solutions. Then the solution topologies are simulated and compared. The tool presented is quite versatile and may even be used for other kind of topologies in z-domain.

ÖZET

SİGMA-DELTA KİPLEYİCİLER İÇİN OTOMATİK MİMARİ GELİŞTİRİCİ

Sigma Delta analog sayısal çeviriciler daha basit analog devrelerle daha yüksek çözünürlüğe erişebildikleri için analog sayısal çeviricilerin yaygın bir türüdür. Geri ve ileri besleme yolları kulanılması performanslarını daha da arttırmaktadır. Ancak olası geri ve ileri besleme yollarının sayısı oldukça fazla olduğundan bu yollardaki kazanç değerlerinin bulunabilmesi önemli bir konudur. Bu tezde istenen özellikte topolojiler bulan otomatik bir Sigma Delta analog sayısal çevirici yazılımı ortaya konmuştur. Sonra bu yazılım ikinci derece Sigma Delta analog sayısal çevirici topolojileri bulmak için kullanılmıştır. Bu sırada topolojilerin bulunabilmesi için bazı farklı yaklaşımlara ihtiyaç duyulmuştur. Sonra bulunan topoloji çözümleri simüle edilmiş ve karşılaştırılmıştır. Bu tezde bahsedilen yazılım çok amaçlıdır ve z uzayındaki başka tür topolojilerin bulunmasında da kullanılabilir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS/ABBREVIATIONS

AAG	Automatic Architecture Generator
AAG2	Automatic Architecture Generator 2 (introduced in this thesis)
A/D	Analog to Digital
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
FFT	Fast Fourier Transform
GUI	Graphical User Interface
NTF	Noise Transfer Function
PSD	Power Spectral Density
SD	Sigma-Delta
SNR	Signal-to-Noise Ratio
STF	Signal Transfer Function
TF	Transfer Function

1. INTRODUCTION

1.1. Background

Analog to digital converters are commonly used in electronic engineering since digital signals are usually easier to process. Computers hold and process the data digitally whereas the signal itself is sometimes digital, but sometimes not. When the signal is analog, we need to convert it into digital in order to process it easily.

There are various methods used for A/D conversion, but they can be divided into two main groups; Nyquist rate converters and oversampling converters.

Generally used Nyquist rate converter architectures are DAC based, Subranging, Integrating Type, Pipelined, Charge Redistribution, Successive Approximation and Flash. Since Nyquist rate converters require precise matching of components, they cannot achieve high bits of resolution unless some advanced and expensive techniques are used. The block diagram of a general Nyquist rate converter is in Figure 1.1.

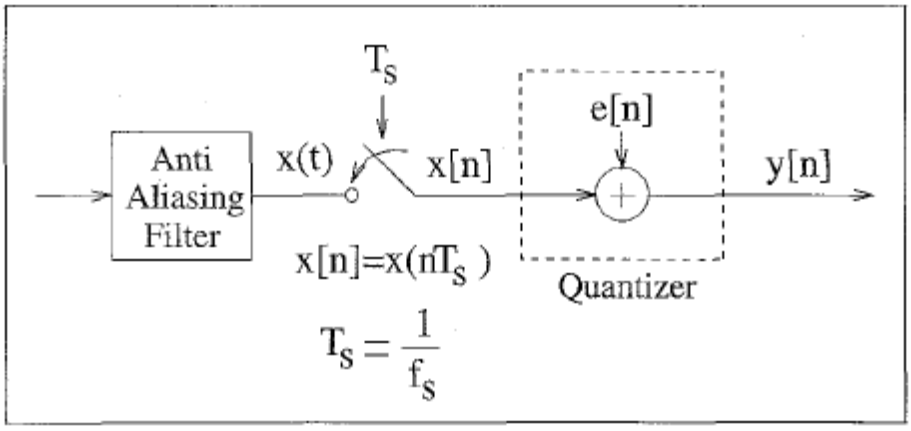

Figure 1.1. Block diagram of a Nyquist rate converter [1].

Oversampling converters sample the signal at a frequency much greater than Nyquist frequency. They have relatively lower bandwidth but higher resolution. As seen in Figure 1.2, oversampling converters trade bandwidth with resolution.

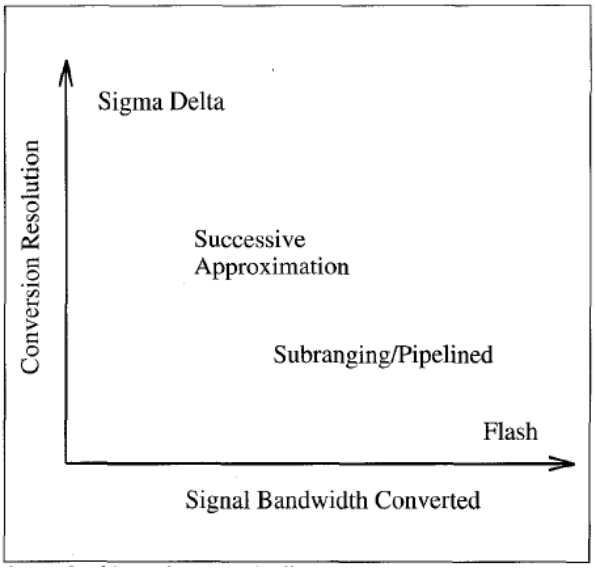

Figure 1.2. Bandwidth resolution tradeoffs [1].

Nyquist rate converters are not capable of achieving 10 to 12 bits of resolution unless trimming or special fabrication techniques are used [9,10]. High accuracy may also be achieved by using complex digital self-calibration algorithms [11].

Oversampling converters require less complex analog circuitry but faster digital circuitry [12]. Also different from Nyquist rate converters, the specifications on the anti-aliasing filter are more relaxed. The block diagram of an oversampling converter is given in Figure 1.3.

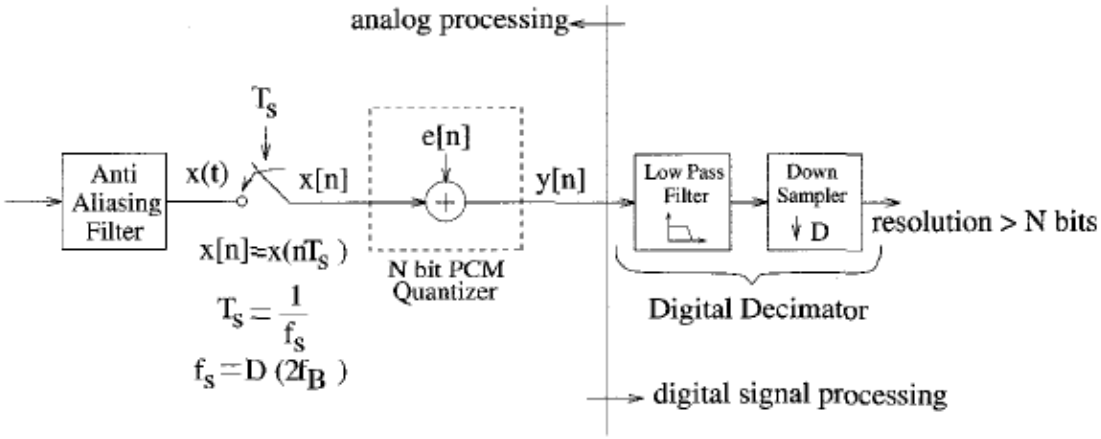

Figure 1.3. Block diagram of an oversampling converter [1].

An oversampling converter has analog and digital sections. In the analog section an anti aliasing filter is applied. Then, the signal is sampled at a much greater rate than the

Nyquist rate. Since the signal is sampled at a high frequency, the cutoff frequency of the anti aliasing filter is much higher than the Nyquist rate converter case. Then, quantization is performed. Quantization process generates white noise uniformly distributed over the frequency band [1,5,12], so the noise power spectral density for the Nyquist rate and oversampling converters are given in Figure 1.4.

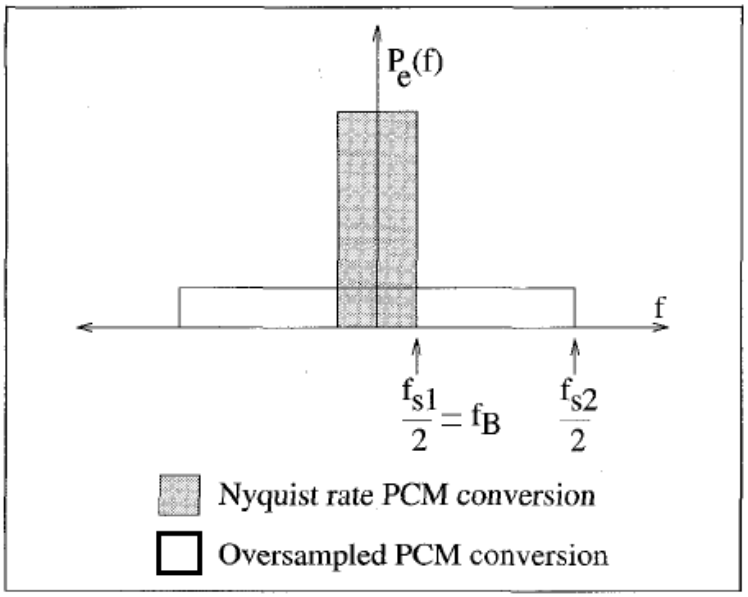

Figure 1.4. Noise power spectral density for Nquist rate and oversampled converters [1].

Figure 1.5 shows a signal that is sampled above the Nyquist frequency, but not oversampled too much. In Figure 1.6, the same signal is shown when the oversampling ratio is much higher.

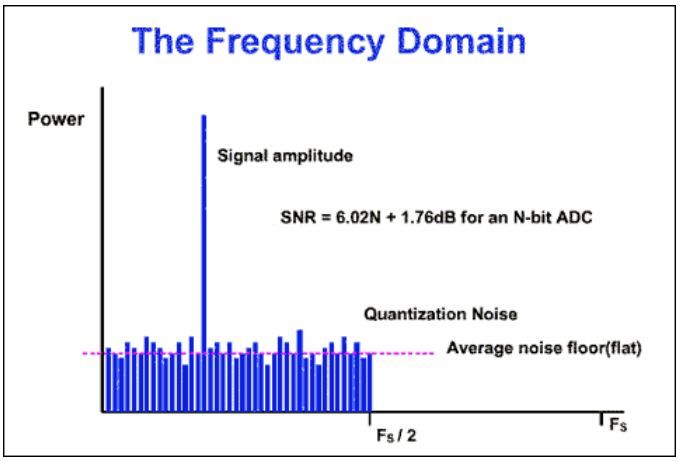

Figure 1.5. A signal sampled with a small oversampling ratio [4].

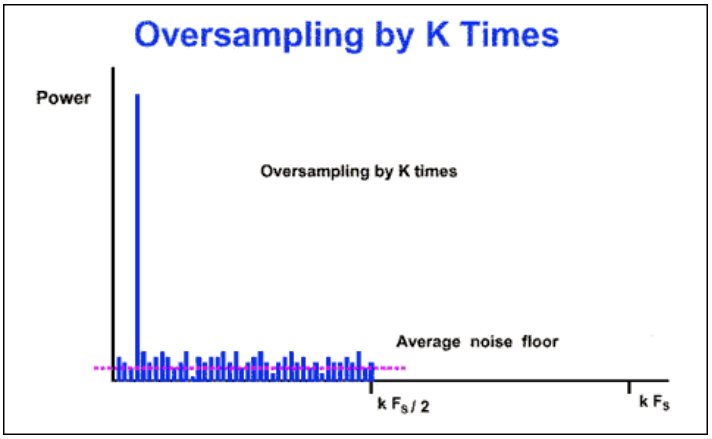

Figure 1.6. The same signal sampled with a much higher oversampling ratio [4].

If a low pass filtering is applied to this signal, most of the noise will be gone. This situation is shown in Figure 1.7. Since the signal is digital, the low pass filter can be implemented in the digital domain.

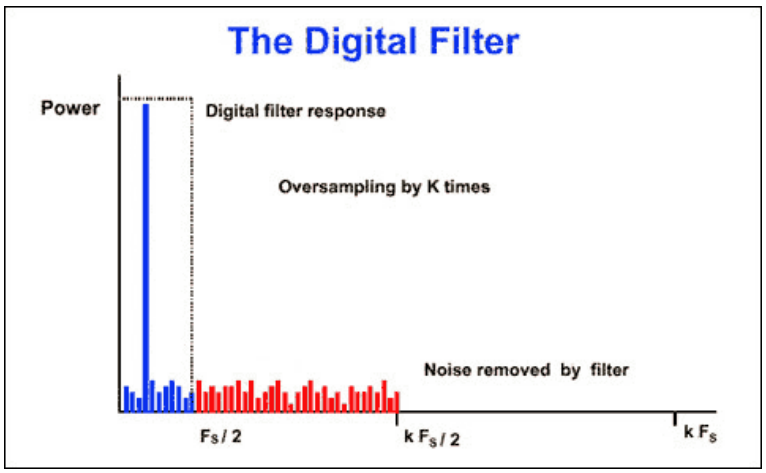

Figure 1.7. The noise is removed if a low pass filter is applied [4].

The noise spectrum is not required to be flat. If sigma delta ADCs are designed to have a high pass noise shape, when the low pass filter is applied the noise reduction would be much higher. This situation is shown in Figure 1.8.

After quantization is performed, only a part of the frequency spectrum is needed. For this purpose, a low pass filter is applied. After the low pass filter, most of the quantization noise will be removed because the noise was distributed over a large frequency spectrum. Since most of the noise is removed, higher bit resolutions can be reached.

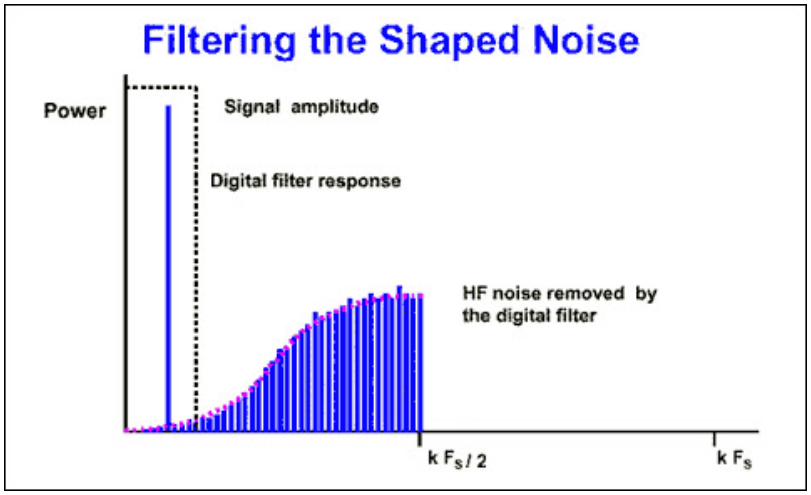

Figure 1.8. The noise is shaped and gathered at higher frequencies, so the noise reduction done by low pass filter is much greater [4].

1.2. Problem Definition and Outline of Thesis

In a Sigma Delta A/D Converter design, there are a lot of combinations of all possible feedback and feedforward paths. Checking all the combinations analytically is impractical. In order to solve this problem, a CAD tool which is used to design a Sigma Delta ADC will be introduced in this thesis. The tool will take the netlist and the design parameters from the user and check if it is possible to obtain a block level design that satisfies desired functions. The user will be able to choose to get a parametric solution or a numeric solution.

The tool that will be introduced will be named as "Automatic Architecture Generator 2" and will be referred as AAG2. This tool is written in MATLAB and is based on [3]. The tool introduced in [3] will be referred as AAG.

AAG2 also designed such that it can handle non idealities of integrators. As explained in [3,6], the non idealities of integrators may be gathered into two coefficients which take place in the transfer function of the integrator.

Chapter 2 will explain the algorithm of AAG2 in detail. This chapter also contains the input formats, output formats and GUI explanation.

Chapter 3 shortly demonstrates how to run AAG2. Although nearly every step of running AAG2 is explained in Chapter 2, a practical example is given in this chapter.

Chapter 4 includes applications of AAG2 and some solutions found by its application. The aim of these applications is to find a second order topology that has low power consumption and required small area to implement. For this purpose some constraints are applied to the output of the first integrator. The Simulink simulation results of the solutions are also given in this chapter.

One of the solutions found by AAG2 is selected and simulated by SPICE. The gain components are implemented by using switched capacitors. These simulation results are given in Chapter 5.

Chapter 6 summarizes this thesis and the features of AAG2.

2. SIGMA-DELTA A/D CONVERTERS AND REALIZATION

2.1. Sigma-Delta A/D Converters

As explained in the previous section, oversampling converters have some advantages over Nyquist rate converters when high resolution is needed.

The conceptual block diagram of a sigma delta converter is given in Figure 2.1. The order of a sigma delta ADC is determined by the number of integrators in it. There may be feedback or feedforward paths with certain gains in the block diagram. In order to suppress noise better, usually a second or higher order sigma delta ADC is required. Although a classical second order sigma delta ADC is given in Figure 2.2, the block diagram becomes Figure 2.3 when all the possible paths are included.

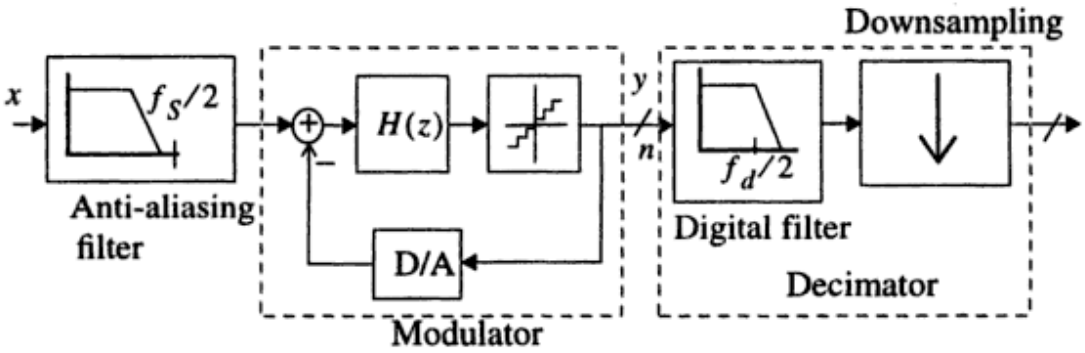

Figure 2.1. Sigma-Delta ADC block diagram [5].

The relation between input, x and output of the modulator, y in z-domain is:

$$Y(z) = STF(z) * X(z) + NTF(z) * E(z)$$

where STF is the Signal Transfer Function, NTF is the Noise Transfer Function and E(z) represents the z-transform of quantization noise. STF and NTF can be defined as

$$STF(z) = \left.\frac{Y(z)}{X(z)}\right|_{E(z)=0} \text{ and } NTF(z) = \left.\frac{Y(z)}{E(z)}\right|_{X(z)=0}.$$

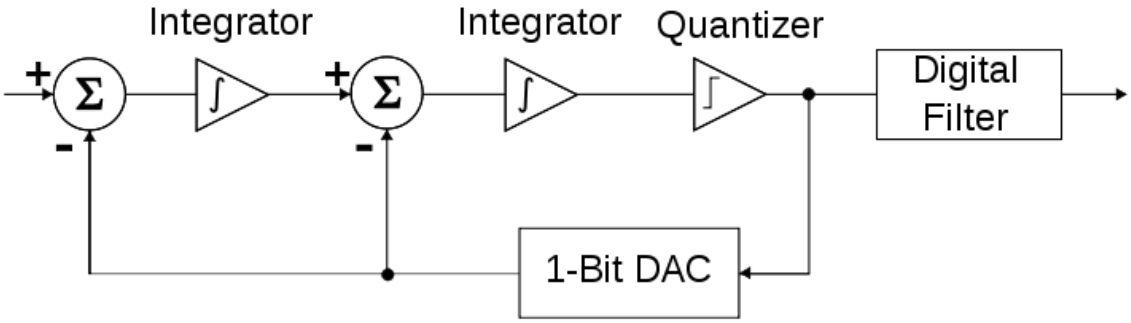

Figure 2.2. A simple second order sigma delta ADC [2].

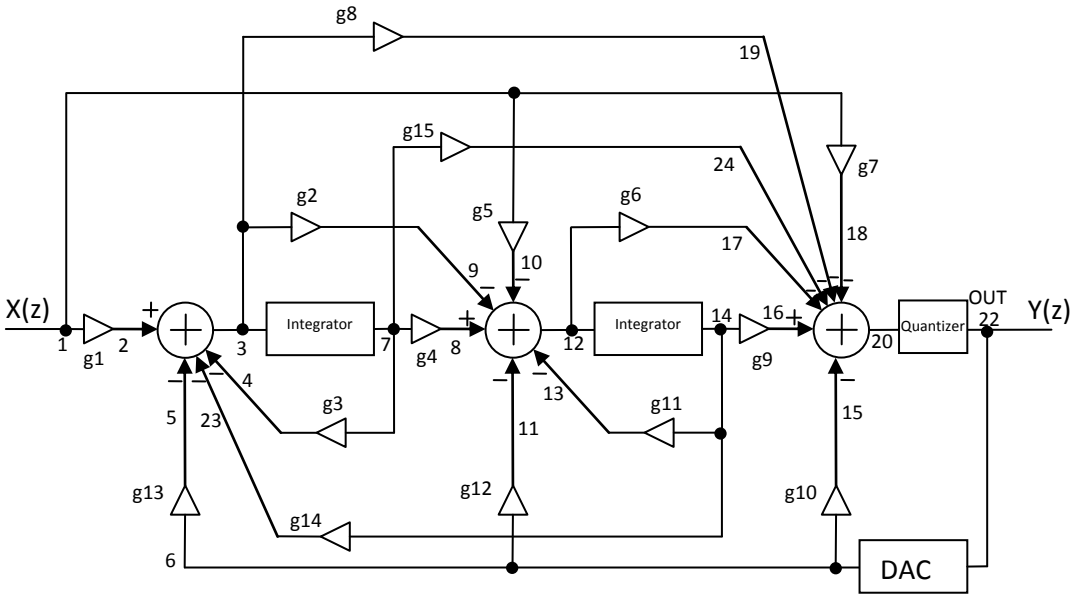

Figure 2.3. A second order sigma delta ADC with
all possible feedback and feedforward paths.

Different topologies are examined more in detail in [5,9,10,12].

2.2. Switched Capacitors

In modern VLSI implementations, main components are transistors and capacitors. Since implementing resistors is undesired in VLSI design, capacitors with MOSFET switches can be used instead of resistors. These switches are turned ON and OFF periodically [14]. Figure 2.4 shows two different topologies for clocking scheme for simulating resistors.

In Figure 2.4a, the capacitor is charged when Q1 is conducting. Then assuming $V_2 > V_1$, the capacitor is discharged to V_2 when Q2 is conducting. In Figure 2.4b, the capacitor is charged to voltage V_1-V_2 when Q1 is conducting and discharged when Q2 is conducting. Q1 and Q2 never conduct at the same time.

Switched capacitor technique also reduces the area required. As explained in [14], for R=$10^7\Omega$ and assuming clock frequency of 100 *kHz*, a capacitance of 1 *pF* is required. This capacitor occupies an area of 3 mil^2, whereas implementing the corresponding resistor would require 1600 mil^2.

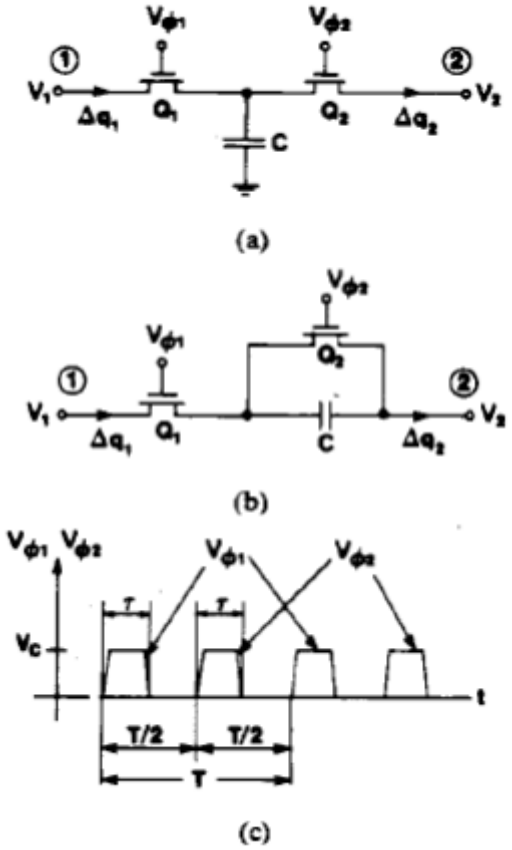

Figure 2.4. Switched capacitor circuits (a) Shunt circuit (b) Series circuit (c) Clock waveforms [13].

A switched capacitor integrator circuit diagram is given in Figure 2.5a. C_1 is charged when Q1 is conducting. Then, when Q2 is conducting the charge on C_1 is transferred to C_2. If examined in more detail, at time t_{n-1}, which is assumed to be the end of the Φ_1 pulse, C_1 is charged with charge $C_1*V_{in}(t_{n-1})$. At t= t_{n-1}+(T/2), this charge is added to C2. The output voltage at t=t_n then becomes [13]

$$vout(t_n) = -v_{C2}(tn) = -\frac{C_1}{C_2} v_{in}(t_{n-1}) .$$

Solving this equation in z-domain gives the transfer function

$$H(z) = \frac{V_{out}(z)}{V_{in}(z)} = -\frac{C_1}{C_2} \frac{z^{-1}}{1 - z^{-1}} .$$

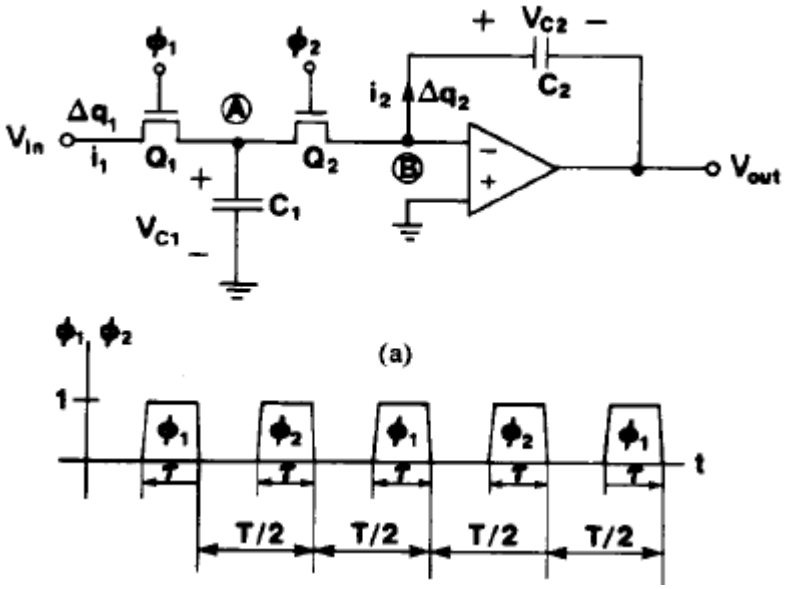

Figure 2.5. Switched capacitor integrator using shunt switched capacitor circuit (a) Circuit diagram (b) Clock waveforms [13].

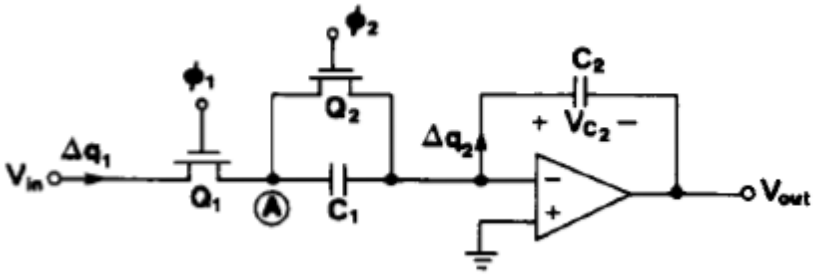

Figure 2.6. Switched capacitor integrator using shunt switched capacitor circuit diagram.

Similarly, switched capacitor integrator using series switched capacitor circuit is given in Figure 2.6. Doing a similar analysis, the transfer function is obtained as

$$H(z) = \frac{V_{out}(z)}{V_{in}(z)} = -\frac{C_1}{C_2} \frac{1}{1 - z^{-1}} .$$

2.3. Integrator Non-Idealities

The transfer function of an ideal delayless integrator is:

$$H(z) = \frac{1}{1 - z^{-1}} = \frac{z}{z - 1}.$$

However, sometimes this non-idealities should be considered when finding solutions. Non-idealities in an integrator may be expressed in its transfer function. For this purpose two coefficients may be used to express the non-idealities:

$$H(z) = \frac{b * z}{z - c}.$$

where *b* and *c* would express the non-idealities. The coefficient *c* is related to current leakage and the coefficient *b* is related to gain. When *b* and *c* both are 1, the integrator is ideal.

3. AUTOMATIC ARCHITECTURE GENERATOR 2

3.1. Automatic Architecture Generator

The Automatic Architecture Generator 2 which will be referred as AAG2 proposed in this thesis is based on the Automatic Architecture Generator (AAG) in [3]. AAG was written in MATLAB and the approach may be divided into 3 steps:

- Create and solve node equations from given netlist
- Create and solve gain equations by matching node transfer functions to desired functions
- Choose useful solutions among all solutions

In the first step, from a given netlist, the program creates node equations in z-domain. Then it solves these equations and finds the transfer functions for all nodes.

In the second step, the desired responses are matched with node transfer functions and matching equations are obtained. Then, these equations are solved and required gain values are found. These values are sometimes not numeric, but parametric.

The third step contains choosing only the useful solutions by eliminating duplicate solutions, solutions with imaginary gain values, and solutions with delayless loops.

3.2. Automatic Architecture Generator 2 Algorithm

In this thesis an improved version of AAG called Automatic Architecture Generator 2 is proposed. Automatic Architecture Generator version 2 will be referred as AAG2 in this thesis.

Although the main part of the approach in AAG2 is the same as AAG which is generating and solving equations in z-domain using MATLAB, it also has a lot of differences. First of all, the code of AAG acquired to make a basis for this thesis had a lot

of missing files and the existing files had a lot of bugs. In fact there were only 3 files that were useful from the code of AAG: “tf_gen.m” which holds netlist component definitions, “equgen.m” which is about matching coefficients and “coeffsort.m” which sorts coefficients of z polynomials. The last file was written by Frank Fisher and modified by the author of this thesis.

In the final version of AAG2, all the files have been changed and improved. All of the bugs have been cleared, the code was completed and improved, a lot of features added, and a GUI is created for ease in use.

AAG2 also has 3 steps:

- Create and solve node equations from a given netlist
- Create and solve gain equations by matching node transfer functions to desired functions, thereby obtaining parametric solutions, then clearing duplicate parametric solutions
- From parametric solutions, find useful numeric solutions, then clear duplicate numeric solutions

The third step which is “Choose useful solutions among all solutions” in the AAG’s steps [3] is not in AAG2’s steps. This is not because AAG2 does not select useful solutions, but because there is not a separate step for it. It is implemented in the algorithm, but not as a separate step. Clearing duplicate parametric and numeric solutions and finding useful numeric solutions are also parts of this process.

The GUI of AAG2 is depicted in Figure 3.1. It has a quite useful feature, its steps can be run separately. AAG2 performs only the steps whose checkboxes are checked. For example, finding parametric solutions may take a lot of time and the user may want to find corresponding numeric solutions later. In such a situation, the checkbox of “Solve Parametric” should be checked, and the checkbox of “Solve Numeric” should be unchecked. The GUI also has a very useful feature; it permits access to textboxes only if the corresponding files whose names are in those textboxes are used.

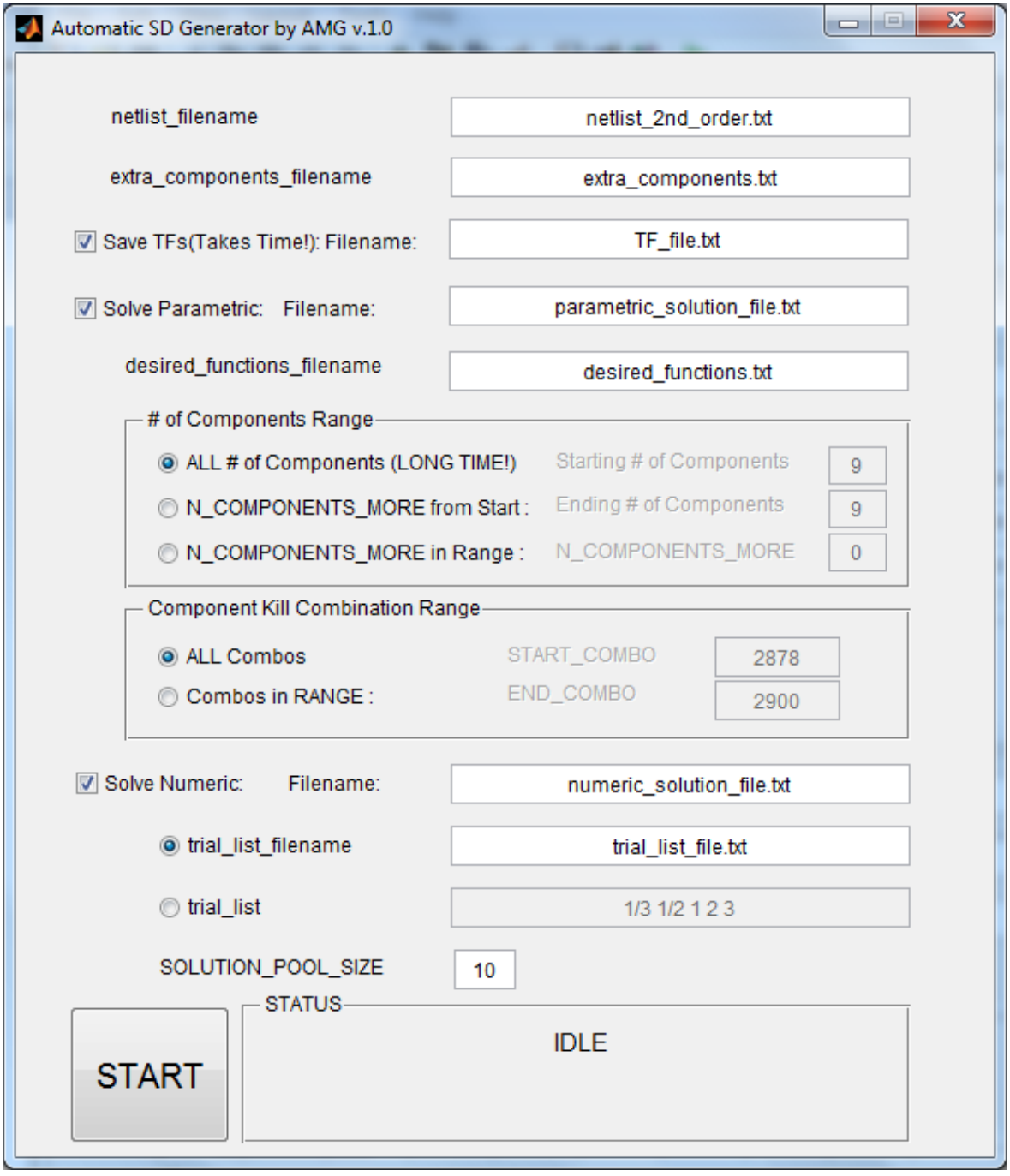

Figure 3.1. The GUI of AAG2.

First of all, if checkboxes of either “Save TFs” or “Find Parametric” are checked, then the program creates and solves netlist equations. Let us assume such a situation. In this stage, the filenames in the “netlist_filename” and “extra_components_filename” textboxes are used. “netlist_filename” is the filename of the netlist to be used in the program. For now let us assume the file indicated in the “extra_components” textbox is empty for simplicity.

From the given netlist, AAG2 creates node equations in z-domain. Valid component types are:

- GAIN: One input, one output. During this stage the gain parts do not have numerical values. The program assigns values g1,g2,… to the gains and considers these values as constants for now.
- ADDER: Multiple input, one output. The inputs may have a minus "-" sign indicating subtraction. The number of inputs may be arbitrary.
- INTEGRATOR: A delayless integrator which has transfer function of $\frac{1}{1-z^{-1}}$.
- INTEGRATORD: An integrator with unit delay which has transfer function of $\frac{z^{-1}}{1-z^{-1}}$.
- NONIDEAL_INTEGRATOR: A non-ideal integrator without delay with transfer function of $\frac{b}{1-cz^{-1}}$. The non-idealities are modeled with coefficients b and c.
- NONIDEAL_INTEGRATORD: A non-ideal integrator with unit delay with transfer function of $\frac{bz^{-1}}{1-cz^{-1}}$. The non-idealities are modeled with coefficients b and c.

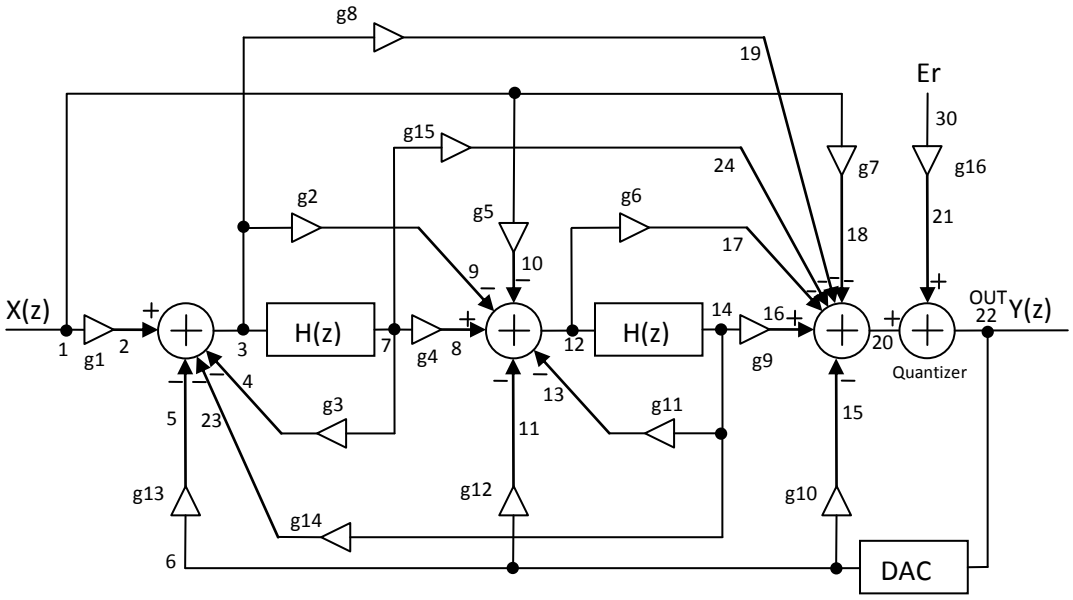

Figure 3.2. The corresponding block diagram of the general 2^{nd} order sigma-delta A/D converter netlist.

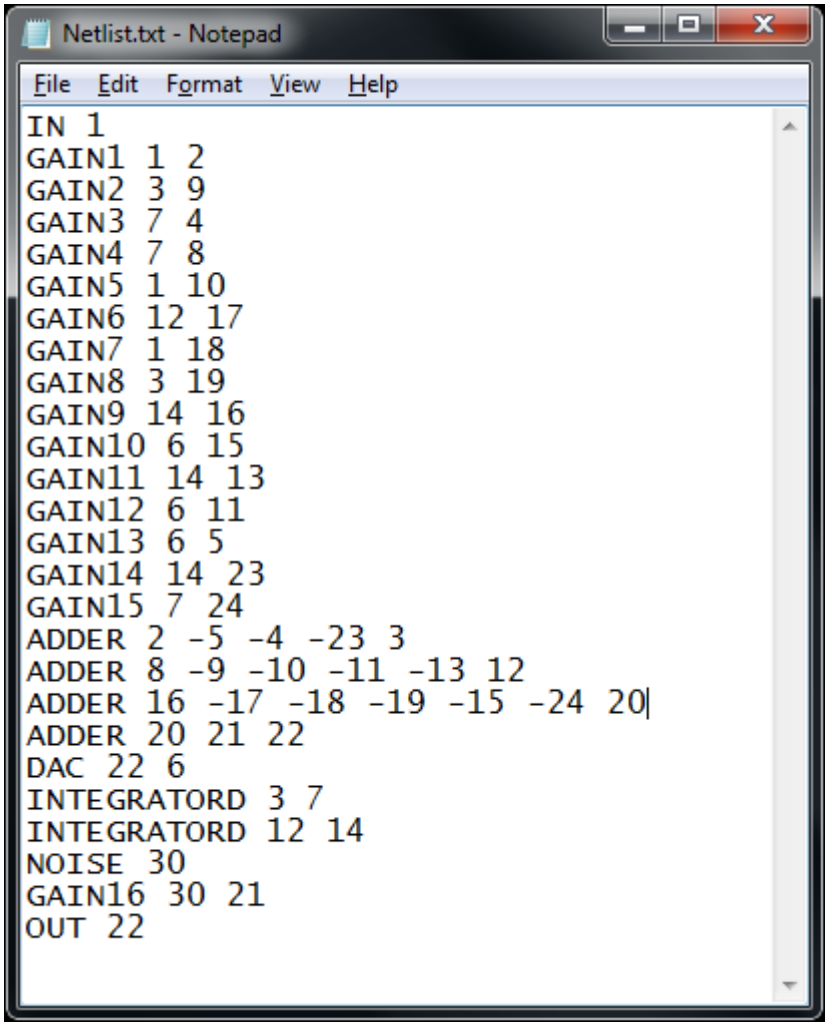

Figure 3.3. An example netlist.

An example netlist is given in Figure 3.3. This is the netlist for the most general 2nd order sigma-delta A/D converter. In this netlist, all possible feedback and feedforward paths are included. The corresponding schematic is given in Figure 3.2.

As seen in the netlist, there are a few more components defined which were not in the valid components list above. Those are usually just used for assigning a name to a node.

If the lines in the netlist are examined:

IN 1

In this line, the component IN is used just to determine that node 1 is the input node. The nodes are named as “x” followed by the node number in the netlist. IN is defined as a variable in AAG2, so this line generates the equation

IN = x1.

On the other hand, if we examine the line:

GAIN2 3 9

This line defines a gain component between nodes 3 and 9, and the id of the gain component is given after the command GAIN. The id should be an integer between 1 and 99. The first node is the input and the second node is the output, so this line generates the equation

x9 = x3*g2.

Adder components are quite interesting compared to the components mentioned above. They may have arbitrary number of inputs but one output. So let us examine the line:

ADDER 2 -5 -4 -23 3

The command ADDER defines an adder, with the last number on this line as output node and all the other numbers as input nodes. The “–“ signs indicate subtraction. So this line generates the equation

x3 = x2 – x5 – x4 – x23.

In z-domain, DAC does nothing, but is included in the netlist anyway. This is because we may want to define a non-ideal DAC with some error later. But for now, it can be considered as a unity gain component. So the line:

DAC 22 6

generates the equation

x6 = x22.

Now, the next line:

INTEGRATORD 3 7

This line indicates that there is a delayed integrator in between nodes 3 and 7. Since such an integrator has transfer function of $\frac{z^{-1}}{1-z^{-1}}$, this line generates the equation

$x7 = (x3 * z^{-1}) / (1 - z^{-1})$.

The error node is defined in the line:

NOISE 30

The noise is also referred to as Error in this thesis. Thus, this line generates the equation

Er = x30.

Finally, the output node is defined in the line:

OUT 22

which generates the equation

OUT = x22.

All these generated equations are gathered together and all the node names are gathered together separately. Then, solve command of MATLAB is used to solve the set of equations with respect to the set of node names. IN and Er is defined as variables in MATLAB, so the results found will be in terms of IN and Er in addition to gain variables and z. Thus, all the node transfer functions are found by now.

The coefficients of IN and Er are in the form of a polynomial of z which have g1,g2,… and constants as coefficients. The nodes have names x1,x2,… Hence, an example for a node transfer function would be

x5= [((g1*g2)*z^2 + (g5+2)*z + 1) / ((g13-1)*z + 1)] * IN
+[(g2*z + g11) / ((g1+1)*z^2 + g5*z + g2)] * ERROR .

In this step, if a real netlist with real components is used, the solve command of MATLAB should not need any parameters for solutions hence all the solutions obtained by this command can be used directly.

AAG2 is intended to find gain values for S/D ADC architecture but can be used for any architecture in z-domain. This may require new parts to be defined. The user may need some new parts when designing a S/D ADC. So in case the user needs new components to be defined, this can be done in two methods:

- New components can be defined in “tf_gen.m”. This file can also be examined first if any help required about writing or modifying a netlist.
- “extra_components” feature can be used. This feature is explained below.

“extra_components” feature give user the control to add new netlist equations before solving them. The “extra_components_filename” should always contain a valid filename that corresponds to an existing file. If the user does not want to use this feature, the textfile should exist but be empty.

The extra component file has syntax of 3 lines.

- The first line should contain new node names that are introduced in the third line. These node names should be in the form of “x” followed by a number. There should be a white space between node names. If there are no new nodes, then this line should be left empty.
- The second line should contain new variables that will be matched during matching equations. These variables will be treated same as gain variables (g1,g2,…). There is no limitation to the name format of these variables but they cannot contain white space or strings that may be meaningful to MATLAB. In order to avoid problems, only letters and numbers but no other characters to be used in the variable names is recommended. These variables too should contain a white space between each other in the second line.
- The third line should contain the equations to be added to the netlist equations before solving them and finding node transfer functions. The equations should have a comma in between them. In order to avoid problems, avoid using white spaces.

An example for the extra component file:

```
x99
A B C
x6=(A*x99)+(B*(z^-1)*x22)+(C*(z^-2)*x22),x99=x22+x3
```

If the extra component file is as given above, x99 will be treated the same as any other node (x1,x2,…). This includes finding x99’s node transfer function just like other nodes. AAG2 will treat A,B and C same as gain variables (g1,g2,…) and will try to find their values after matching node transfer functions to the desired functions. In the last line, there are two equations that are to be added to the set of node equations. As seen in the example, these extra equations may contain polynomials of z. The user should be able to enter any kind of equation, but that is out of scope of this thesis.

3.2.1. Save Transfer Functions

If the first checkbox, “Save TFs”, is checked, the program saves the node transfer functions to the file whose name is in the textbox. In the figure above, it is “TF_file.txt”.

All of the node transfer functions will be saved. The node transfer functions will be in the form mentioned above. An example would be

x5= [((g1*g2)*z^2 + (g5+2)*z + 1) / ((g13-1)*z + 1)] * IN
+[(g2*z + g11) / ((g1+1)*z^2 + g5*z + g2)] * ERROR .

3.2.2. Find Parametric Solutions

If the second checkbox, “Find Parametric”, is checked, the program finds parametric solutions. In this step the node transfer functions are matched to the desired functions. The desired functions should be entered into the file whose name is written in the “desired_function_filename” textbox.

The program already found node transfer functions which have STF and NTF parts. STF part is the value when noise is zero, and NTF part is the value when the signal is zero. These STF and NTF parts are in the form of polynomials of z which have g1,g2,… and constants as coefficients. For the example above, STF and NTF of x5 would be

x5_STF = ((g1*g2)*z^2 + (g5+2)*z + 1) / ((g13-1)*z + 2)
x5_NTF = (g2*z + g11) / ((g7+1)*z^2 + g5*z + g8) .

In this step, the program matches these polynomials with the desired polynomials. The desired polynomials are entered by user. A desired polynomial indicates that the STF or NTF of that node is desired to be the entered polynomial. Thus, the user enters a polynomial with numeric coefficients and AAG2 matches these numeric values with the coefficients in the node transfer functions found by AAG2. During this stage, nonlinear equations which are in the form of different multiplications and additions of g1,g2,… and numeric values are obtained and then this set of nonlinear equations are solved by MATLAB’s solve command. Unlike the previous case, solving netlist equations, most of the solutions found by the solve command would be parametric, so they cannot be used directly.

Desired functions are entered as follows:

NODE MODE NUMERATOR / DENOMINATOR

There are some important points in this format:

- The user should put a white space both before and after "/".
- The user may enter multiple desired functions by entering them in separate lines.

So an example would be:

x5 STF (3*z^2 + 1) / (z+2)
x5 NTF 2*z+1 / 1

The program matches coefficients and generates these equations:

g1*g2 = 3
g5+2 = 0
g13-1 = 1
g2 = 2
g11 = 1
g7 + 1 = 0
g5 = 0
g8 = 1

AAG2 does not generate the equations 1=1 or 2=2 because both sides are numeric. In such cases, it uses a scale factor which will be explained later in this chapter.

AAG2 tries to find the solution with the least number of components. For this purpose, it tries to make zero as many gain components as possible. It may run in different modes of combinations, which will be explained in Section 2.3.5. For now, let us assume "All # of Components" and "All Combos" are selected.

For every number *k* from 1 to number of components, AAG2 first generates k-combinations of components. Then, for every combination, it makes all the components in that combination zero and tries to solve the equations in this situation. It replaces zero with those components in matching equations and adds required equations which indicate the components in the combination is zero. An example for such equations would be: "g1=0".

After AAG2 generated all the equations, MATLAB's solve command is used to solve this set of equations. Unlike netlist equations, these matching equations may or may not be linear. In most cases they are not linear.

AAG2 usually finds parametric solutions for these matching equations. Before continuing to the next step, the duplicate parametric solutions should be removed. In most cases, the duplicate parametric solutions have different parameter names. Because of this, the parameter names are first replaced according to a rule. This rule is simple, just rename the first parameter that appears on the solution as PARA_1 and second one as PARA_2 and go on like this. After parameter name replacement, the duplicate solutions are cleared easily by string matching term by term.

The resulting parametric solutions are written into the file whose name is written in the textbox in the same line as "Solve Parametric". On the output file, first line is reserved for the names of gain components. Starting from the second line, every new line contains a new parametric solution. In each line, the n^{th} term corresponds to the n^{th} gain component in the first line. So an example would be:

g1 g2 g3 A B C
PARA_1 1 0 0 PARA_2 PARA_1*PARA_2
PARA_1 PARA_2 PARA_3 0 0 1

3.2.3. Find Numeric Solutions

In this step, numeric solutions are found. AAG2 uses parametric solutions found earlier to find the numeric solutions. For every parametric solution, a numeric solution is found. Then, duplicate numeric solutions are deleted. So, although for every parametric

solution a numeric solution is found, in the end there may be less numeric solutions than parametric ones. Assuming a parametric solution is given, the corresponding numeric solution is found by:

- From given trial list, generate all the numeric solutions. This is done by giving all combinations of trial list members to parameters in the parametric solution.
- From this numeric solution set, create a solution pool with the best "SOLUTION_POOL_SIZE" number of elements according to "criteria_1.m"
- From this solution pool, get the best numeric solution according to "criteria_2.m"

The criteria mentioned above are:

- criteria_1.m : The absolute value of the proportion of the biggest gain value to the smallest gain value. Smaller is better.
- criteria_2.m : The sum of absolute values of gain values. Smaller is better.

The order of criteria can be changed very easily by just renaming corresponding ".m" files. If needed even new criteria may be defined under the same names, but the user should be careful about the fact that the program tries to minimize the criteria values. If maximization is needed, changing only "criteria_1.m" and "criteria_2.m" would not be enough. In such a case, the user should modify "find_best_solution.m". In that file, "trial_solution_array" is the set of numeric solutions that were found by assigning all the combinations of trial list members to all paramaters.

Trial list and "SOLUTION_POOL_SIZE" and are entered by user. Trial list is a list of numbers and "SOLUTION_POOL_SIZE" is just an integer.

Let us assume the given parametric solution is:

g1 = PARA_1
g2 = PARA_1 + 1
g3 = PARA_1 * PARA_2
g4 = PARA_3

g5 = PARA_1 * PARA_2 + PARA_3

and given trial list is:

1 2

and SOLUTION_POOL_SIZE is 3.

The program generates the solutions given in Table 3.1.

Table 3.1. Parameters and Corresponding Solutions.

PARAMETERS			SOLUTIONS				
PARA_1	PARA_2	PARA_3	g1	g2	g3	g4	g5
1	1	1	1	2	2	1	3
1	1	2	1	2	2	2	4
1	2	1	1	2	2	1	3
1	2	2	1	2	2	2	4
2	1	1	2	3	6	1	7
2	1	2	2	3	6	2	8
2	2	1	2	3	6	1	7
2	2	2	2	3	6	2	8

Then the best 3 solutions are selected according to “criteria_1.m”, which was minimizing the gain proportions. The solution pool would be:

- {1,2,2,1,3}
- {1,2,2,2,4}
- {1,2,2,1,3}

Then, the best one is found as {1,2,2,1,3} since “criteria_2.m” was to minimize the sum of gains. For better understanding of this algorithm, “find_best_solution.m” could be examined.

One another important fact is that changing “SOLUTION_POOL_SIZE” may sometimes change the resulting numeric solution.

After finding numeric solutions, the program eliminates the duplicate numeric solutions, if there is any.

3.3. Other Features

This section tries to explain other features of AAG2 which were not covered since the beginning of this chapter.

3.3.1. Numeric Scaling

Matching polynomial coefficients may look like a trivial task at first but it is not as trivial as it looks. Let us consider these two polynomial divisions

$$\frac{(g1 * z) - 1}{(g2 * z) + g3} \text{ and } \frac{z - 1}{z + 1}.$$

Matching these two polynomials gives the set of equations

g1=1 -1=-1 g2=1 g3=1

which are solvable.

Now let us consider matching these two:

$$\frac{(-g1 * z) + 1}{(-g2 * z) - g3} \text{ and } \frac{z - 1}{z + 1}.$$

If the coefficients are matched directly, the resulting set of equations would be:

-g1=1 1=-1 -g2=1 -g3=1

which are not solvable because of 1=-1 contradiction. But the first polynomial division is the same as the one in the previous example, so the system should be solvable because there is a solution: g1=1, g2=1, g3=1.

The minus sign may not be the only problem. Any numerical value may cause the same problem. Let us consider P to be a real number. Now let us consider these two:

$$\frac{(P * g1 * z) + P}{(P * g2 * z) - (P * g3)} \text{ and } \frac{z-1}{z+1}$$

The resulting set of equations would be

P*g1 = 1 P = -1 P*g2 = 1 -P*g3=1.

This set is not solvable unless P=-1. When MATLAB's solve command solves the set of equations, although the resulting polynomial divisions are correct they may not always be in the desired form.

It may be assumed that MATLAB has collect and simplify commands that would solve this problem. However, this is not the case since the biggest powers of *z* may be a variable such as *g1*. In such a case, we cannot simply put the results in a desired form.

Hence, when matching two polynomial divisions, we have to consider a numeric scaling factor. AAG2 does this by defining a variable called scale, then defining its value when there are only numeric values at the same spot of equations to be matched.

Also, let us consider these two polynomial divisions

$$\frac{(g1 * z) + 1}{(g2 * z) - g3} \text{ and } \frac{z-2}{z+1}.$$

Matching coefficients directly gives us

g1=1 1=-2 g2=1 -g3=1

which is not solvable. But if we consider the same two divisions as

$$\frac{(-2 * g1 * z) - 2}{(-2 * g2 * z) + (2 * g3)} \text{ and } \frac{z-2}{z+1},$$

the resulting set of equations would be

-2*g1=1 -2=-2 -2*g2=1 -2*g3=1

which is solvable with solution: g1=-0.5, g2=-0.5, g3=0.5 .

AAG2 does two scale checks, one while matching equations and one after making variables zero and just before using solve command for matching equations. This will be explained more detail in 2.3.3.

3.3.2. z-scaling

Let us consider matching the transfer functions

$$\frac{(g3 * z^2) + (g2 * z) + g1}{(g5 * z^2) + (g4 * z)} \text{ and } \frac{z-1}{z+1}.$$

The first one can be considered as the transfer function of a node and the second one as the desired transfer function for that node. There is a solution for this case, but it requires more than just matching polynomials.

The algorithm explained so far generates the equations

g3=0 g2=1 g1=-1 g5=0 g4=1 0=1.

This is a contradiction, leading to no solution. AAG2 does matching before trying to kill as many components as possible, so it cannot find any solutions for this case. It is not practical to make the matching after killing components because matching takes some time and doing it for every zero combination would increase running time to a non practical value.

But there is a solution which is

g3=1 g2=-1 g1=0 g5=1 g4=1 .

This solution requires canceling out *z* both from numerator and denominator.

There may be many combinations of cancel outs in generic cases, but on the other hand for given transfer functions to match the number of possible cancel outs are generally a few. So this situation can be handled by user quite easily.

AAG2 considers numerator and denominator of the desired transfer function separately, so the desired function entered by the user is not simplified or z's are not cancelled out even it is possible.

For the example above, the user should enter

$$\frac{z^2 - z}{z^2 + z} \text{ instead of } \frac{z - 1}{z + 1}.$$

Now the AAG2 may do the matching correctly and find the solution

$$\frac{(g3 * z^2) + (g2 * z) + g1}{(g5 * z^2) + (g4 * z)} \text{ and } \frac{z^2 - z}{z^2 + z}.$$

And generates the equations

g3=1 g2=-1 g1=0 g5=1 g4=1

which would lead to the solution.

In order to find all the solutions, the user should follow these steps:

- Run the program just to obtain TF's
- Examine the TF's of the nodes and the desired functions of corresponding nodes
- Check if there could be a "z-scaling" explained above
- If there is a "z-scaling" then the user should generate required combinations of z-expanded desired transfer functions and run AAG2 for all of them

3.3.3. Running time

Running time is a quite important issue for AAG2 because there are a lot of zero combinations of components.

When solving the netlist and finding the transfer function of the nodes, MATLAB is quite fast and solves the netlist very quickly. But to write the results in an order "simplify" and "collect" commands are used and these commands may require a few seconds. They are called for every node, so "Save TF's" step may take some time, but rarely more than a minute.

For parametric solutions, if there are n components then there are 2^n possible zero combinations of components. So for about 20 components the number of all combinations would be over one million. AAG2 uses "solve" command of MATLAB which is quite fast, but sometimes detecting the situations when there is no solution and omitting using "solve" command makes AAG2 even faster. Running speed almost tripled after omitting unnecessary "solve" commands. AAG2 does not call "solve" command if during second "scaling", there is a scale mismatch. This is detected if scale is changed more than once.

Currently AAG2 is able to handle around 5500 combinations per hour. Calling "solve" command is not required for most of these.

For numeric solutions, if there are *n* parameters in a parametric solution and *m* values in the trial list, there are m^n different combinations to check. This should be done for every parametric solution. Unfortunately, there is no way to decrease the running time for this situation. If the running time is too long, the user may consider shortening the trial list or running AAG2 for just a few parametric solutions not for an entire set found in the "Solve Parametric" step.

3.3.4. GAIN16 Component

There is a gain component just after the error input called GAIN16. This is not a real gain component. Sometimes there is only one desired function corresponding to NTFs:

OUT STF z^2 / z^4
OUT NTF z+1 / z
x8 STF z^2+z+1 / z^2

Generally the sign of an NTF is not important since the corresponding SNR is the same. So the user may need to run AAG2 again for the same design using desired functions:

OUT STF z^2 / z^4
OUT NTF -z-1 / z
x8 STF z^2+z+1 / z^2

Sometimes one of these set of desired functions may have no solution while the other one has. So the user needs to run for both of them. When GAIN16 is added, the user does not need to run AAG2 for 2 cases. If needed, minus sign is provided by GAIN16. But the drawback is that the user should check GAIN16 manually after the solutions are found to be sure that it is either 1 or -1. It is usually 1 or -1 if desired functions are selected with some knowledge about the circuit. But if it is not 1 or -1, then the whole noise should be considered as multiplied by the value of GAIN16.

This feature may be disabled by modifying the netlist and removing GAIN16.

3.3.5. “Solve Parametric” Modes

Under “Solve Parametric” step, there are several running modes. These modes are about running AAG2 in smaller steps since the runtime is usually very high. The first set is about number of components:

- All # of Components: If this one is selected, AAG2 runs for all number of components, starting from one.

- N_COMPONENTS_MORE from Start: If this one is selected the user is required to specify starting number of components and N_COMPONENTS_MORE. AAG2 will run for: {Start, Start+1,…,Start+N_COMPONENTS_MORE} number of components.

- N_COMPONENTS_MORE in Range: If this one is selected, the user is required to specify starting number of components, ending number of components and N_COMPONENTS_MORE. AAG2 will run like the previous mode explained above, but this time there is one more stopping condition; reaching the ending number of components.

The second set of modes are about combination ranges. Assuming there are three components, {g1,g2,g3} in the circuit, AAG2 uses “nchoosek” command which generates combinations of the components. For example

```
nchoosek({'g1','g2','g3'},2)
```

gives the result

```
'g1'    'g2'
'g1'    'g3'
'g2'    'g3'
```

which is a two dimensional vector. Each line corresponds to a combination. The components in these combinations are considered as zero and then the matching equations are solved. Sometimes, even when running just for a specific number of components, the user may need to run for a specific situation. The user should use “nchoosek” command to find the line number in the two dimensional vector which corresponds to the required situation. For example, if the user wants to consider g1 and g3 as zero and solve the matching equations with only g2, then the combination number would be 2 because “g1 g3” corresponds to the second line. Of course, the number of components is 1 here, so the appropriate options should be selected in the “Number of Components” modes.

Combination modes are:

- ALL Combos: AAG2 runs for all combinations.

- Combos in RANGE: AAG2 runs for combinations just in the specified range. It is important to keep in mind that if the number of components changes AAG2 will still run in the same combination range. So using this mode is advised to run only for a specific number of components.

4. EXAMPLE RUN

This chapter contains an example run of the program.

First of all the user should open MATLAB and select the active directory as the directory of the project files. Then, the user should open menu.fig in GUIDE. This can be done by right clicking menu.fig and selecting “Open in GUIDE”. If directly opened instead of using GUIDE, the GUI does not work because of MATLAB’s limitations. After menu.fig is opened in GUIDE, “Run Figure (Ctrl + F7)” should be selected in order to run the GUI .

First of all, only “Save TF’s” should be selected. In this situation, only the names of the input netlist, the extra components file and the TF output file should be specified. All the input files should be and output files will be created in the active directory. When file names are entered, clicking “START” stats running AAG2. In the example, a generic second order Sigma Delta ADC netlist is used and extra components file is empty. This netlist does not contain GAIN16 component. After TF’s are saved, the user should check the TF’s of the required nodes. In the example, desired functions file contains:

```
OUT STF 1 / z^2
OUT NTF z^2-2*z+1 / z^2
```

The user should check the TF file and if there are possible z scales, new desired function files should be created. In the TF file, there is no term which has greater power of z than 2. So, there are no z scales possible. Now, the user may continue to use AAG2.

In this step, since TF’s are already saved, the user should select only “Solve parametric”. File names of desired functions and parametric solution outputs should be specified. As example, AAG2 will run for a few specific topologies. These topologies all have 9 components, so “N_COMPONENTS_MORE in Range” is selected. “Starting # of Components” and “Ending # of Components” should be 9. “N_COMPONENTS_MORE” can be any number. 0 is recommended. Under “Component Kill Combination Range”,

“Combos in Range” is selected with “START_COMBO” as 2878 and “END_COMBO” as 2900. As introduced in [3], there is a solution is this combination range. It occurs exactly at combo number 2879, which corresponds to equaling g2, g5, g6, g7, g8, g9, g10, g13 and g15 to zero.

When “START” is clicked, AAG2 starts to find parametric solutions in this range. When AAG2 is finished, the parametric solutions are written into the parametric solution output file. The user may check the parametric solutions there. In the range from 2878 to 2900, there are four parametric solutions but two of them are identical, so there should be three different parametric solutions in the output file.

Table 4.1. Parametric Solutions Found by AAG2.

	Solution 1	**Solution 2**	**Solution 3**
g1	PARA_1/(2*PARA_2)	PARA_1	PARA_1/(2*PARA_2)
g2	0	0	0
g3	1/2	0	1/2
g4	PARA_2	1/(PARA_2*PARA_1)	PARA_2
g5	0	0	0
g6	0	0	0
g7	-4	0	0
g8	(8*PARA_2)/PARA_1	0	0
g9	0	PARA_2	2/PARA_1
g10	0	0	0
g11	-1/2	0	-1/2
g12	PARA_1	2/PARA_2	PARA_1
g13	0	PARA_1	0
g14	1/(4*PARA_2)	0	1/(4*PARA_2)
g15	(4*PARA_2)/PARA_1	0	0

The parametric solutions found are given in Table 4.1. The solution 1 is the same as the 4th solution introduced in p.42 of [3].

The user may then proceed to find numeric solutions from parametric solutions. For this, only “Solve Numeric” should be selected. When selected, the user is required to enter the output file name for numeric solutions. There are two options in this part. If “trial_list_filename” is selected, the user should enter a filename in which the trial numbers are written. If “trial_list” is selected, the user should enter the trial numbers into the textbox. Solution pool size is the size of pool created after applying the first criterion. The second criterion is applied to this pool to find the one best solution. One numeric solution is found per parametric solution. In the example, solution pool value is 10 and the trial list consist of 1/3, 1/2, 1, 2 and 3. There is no difference entering trial list into the textbox or as a file as long as the same trial list is used.

“START” button is clicked again. After AAG2 finishes, the numeric solutions found are written into the numeric solution output file. These solutions are given in Table 4.2.

Table 4.2. Numeric Solutions Found by AAG2.

	Solution 1	**Solution 2**	**Solution 3**
g1	1.500000	0.500000	0.500000
g2	0.000000	0.000000	0.000000
g3	0.500000	0.000000	0.500000
g4	0.333333	1.000000	1.000000
g5	0.000000	0.000000	0.000000
g6	0.000000	0.000000	0.000000
g7	-4.000000	0.000000	0.000000
g8	2.666667	0.000000	0.000000
g9	0.000000	2.000000	2.000000
g10	0.000000	0.000000	0.000000
g11	-0.500000	0.000000	-0.500000
g12	1.000000	1.000000	1.000000
g13	0.000000	0.500000	0.000000
g14	0.750000	0.000000	0.250000
g15	1.333333	0.000000	0.000000

5. RESULTS

AAG2 was run to generate the topologies of sigma delta ADCs which have good noise shaping, high SNR and low power consumption. In order to achieve this, second order topologies are searched. First order topologies are not sufficient and third or more order topologies have a lot of components, which makes the running time huge.

The first idea was always to keep the output requirements, because output node STF and NTF directly affect SNR and power consumption. For this purpose OUT STF was either z^{-1} or z^{-2}. This means output will have input unshaped but with one or two unit delay.

For OUT NTF, $(1-z^{-1})^2$ was the requirement that was never changed. $(1-z^{-1})$ means the change in the input in a unit time. This is assumed to be a small value because of oversampling. If this value is not small, then the input is probably not oversampled enough. Squaring such a small value will make it even smaller, resulting in more suppressed noise.

Table 5.1. Desired functions and corresponding number of solutions.

OUT STF	**OUT NTF**	**x8 STF**	**# of Solutions**
z^-1	(1-z^-1)^2	(1-z^-1)	0
z^-2	(1-z^-1)^2	(1-z^-1)	0
z^-1	(1-z^-1)^2	z^-1*(1-z^-1)	0
z^-2	(1-z^-1)^2	z^-1*(1-z^-1)	9
z^-1	(1-z^-1)^2	((1-z^-1)^2)	0
z^-2	(1-z^-1)^2	((1-z^-1)^2)	0

The next idea was to decrease the swing at the first integrator output as much as possible. In the netlist used, the output of the first integrator is node number 8. So x8 STF and x8 NTF are both required to be $(1-z^{-1})$, which would be a small value. But unfortunately there were no solution in this case. One delayed version of the difference,

which is $z^{-1}*(1-z^{-1})$ had been tried and there were still no solution. The combinations of these two signals had also been tried and still no solution was found.

Although a good idea, suppressing both x8 STF and x8 NTF did not give any solutions. So the next idea was to release one of those. Since the noise comes from quantization, it is still a smaller value compared to the signal. This proportion gets better if a higher bit quantizer is used. So x8 NTF is released while x8 STF is kept.

Table 5.2. 5 of the Solutions Found by AAG2.

	Solution 1	**Solution 2**	**Solution 3**	**Solution 4**	**Solution 5**
g1	1/P_1	1/P_1	1/P_1	1/P_1	1/P_1
g2	0	0	-2*P_1	0	0
g3	2	-2/3	2	-2/3	-2/3
g4	P_1	P_1	P_1	P_1	P_1
g5	0	-6	2	-6	-6
g6	0	0	0	1/6	2/3
g7	0	0	0	-1	-4
g8	0	0	0	0	0
g9	1	1/9	-1/3	0	-1/3
g10	0	0	0	0	0
g11	-2	2/3	2/3	2/3	2/3
g12	2	0	0	0	0
g13	-3/P_1	-3/P_1	-3/P_1	-3/P_1	-3/P_1
g14	4/P_1	4/(9*P_1)	-4/(3*P_1)	4/(9*P_1)	4/(9*P_1)
g15	0	(2*P_1)/3	0	P_1/2	0
g16	1	1	1	1	1

AAG2 was run for different combinations of OUT STF and x8 STF while OUT NTF is kept the same. Solutions were found for only one of these combinations. No solution is found up to 9 components and a solution is found with 9 components; so since N_COMPONENTS_MORE is selected as 2, the solutions are found up to 11 components.

The combinations entered as desired functions in NUM / DEN format. The trials and corresponding number of solutions are given in Table 5.1.

The generic second order sigma delta ADC netlist is used while running AAG2. This netlist is given in Figure 3.3.

The parametric solutions found are given in Table 5.2 and Table 5.3.

When AAG2 finds parametric solutions, the parameters are written in PARA_# format but because of space limitations they are written in P_# format in Table 4.1 and 4.2.

Table 5.3. Remaining 4 Solutions Found by AAG2.

	Solution 6	**Solution 7**	**Solution 8**	**Solution 9**
g1	-P_1/3	P_1/(4*P_2)	P_1/(4*P_2)	1/P_1
g2	P_2	-(18*P_2-2)/(3*P_1)	-(2*P_2-2)/P_1	0
g3	6/(2*P_2*P_1+3)	-6*P_2	2	-2/3
g4	-3/P_1	(4*P_2)/P_1	(4*P_2)/P_1	P_1
g5	0	-2/(3*P_2)	(P_2-1)/(2*P_2)	-6
g6	0	0	0	1/6-(3*P_2)/2
g7	0	0	0	9*P_2-1
g8	0	0	0	0
g9	3/(2*P_2*P_1+3)	P_2	P_2	P_2
g10	0	0	0	0
g11	-2	2/3	-2*P_2	2/3
g12	(4*P_2*P_1)/3+2	0	(3*P_2+1)/(2*P_2)	0
g13	P_1	-(3*P_1)/(4*P_2)	-(3*P_1)/(4*P_2)	-3/P_1
g14	-(4*P_1)/(2*P_2*P_1+3)	P_1	P_1	4/(9*P_1)
g15	-(3*P_2)/(2*P_2*P_1+3)	(6*P_2^2+2*P_2)/P_1	0	P_1/2+(3*P_2*P_1)/2
g16	1	1	1	1

An important thing about these parametric solutions is that they are not required to be solutions of a linear system. For example g5 of Solution 7 in Table 5.2 is (6*P_2^2+2*P_2)/P_1; which has a quadratic term.

After the parametric solutions are found, in order to simulate these solutions in Simulink, AAG2 is run for finding numeric solutions.

"1/5 1/4 1/3 1/2 1 2 3 4 5" is used for trial list.

The corresponding numeric solutions are given in Table 5.4.

Table 5.4. Corresponding Numeric Solutions.

	Soln 1	**Soln 2**	**Soln 3**	**Soln 4**	**Soln 5**	**Soln 6**	**Soln 7**	**Soln 8**	**Soln 9**
g1	0.333333	0.500000	1.000000	0.500000	0.500000	-0.333333	0.750000	0.500000	0.500000
g2	0.000000	0.000000	-2.000000	0.000000	0.000000	0.250000	-1.333333	1.000000	0.000000
g3	2.000000	-0.666667	2.000000	-0.666667	-0.666667	1.714286	-2.000000	2.000000	-0.666667
g4	3.000000	2.000000	1.000000	2.000000	2.000000	-3.000000	1.333333	2.000000	2.000000
g5	0.000000	-6.000000	2.000000	-6.000000	-6.000000	0.000000	-2.000000	-0.500000	-6.000000
g6	0.000000	0.000000	0.000000	0.166667	0.666667	0.000000	0.000000	0.000000	-0.333333
g7	0.000000	0.000000	0.000000	-1.000000	-4.000000	0.000000	0.000000	0.000000	2.000000
g8	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
g9	1.000000	0.111111	-0.333333	0.000000	-0.333333	0.857143	0.333333	0.500000	0.333333
g10	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
g11	-2.000000	0.666667	0.666667	0.666667	0.666667	-2.000000	0.666667	-1.000000	0.666667
g12	2.000000	0.000000	0.000000	0.000000	0.000000	2.333333	0.000000	2.500000	0.000000
g13	-1.000000	-1.500000	-3.000000	-1.500000	-1.500000	1.000000	-2.250000	-1.500000	-1.500000
g14	1.333333	0.222222	-1.333333	0.222222	0.222222	-1.142857	1.000000	1.000000	0.222222
g15	0.000000	1.333333	0.000000	1.000000	0.000000	-0.214286	1.333333	0.000000	2.000000
g16	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

All the solutions have the same OUT STF and OUT NTF, so their outputs and SNR are the same. But there is no condition on the NTF of the first integrator output (x8) , so x8 NTF may be different for them. These are given in Table 5.5. The FFT of the OUT node is shown in Figure 4.1. and the SNR value is found to be 72.7 dB.

Table 5.5. First Integrator Output NTFs for different solutions.

	x8 NTF
Solution 1	$3*z^{-1} - z^{-2}$
Solution 2	$3*z^{-1} - z^{-2}$
Solution 3	$(3*z -1) / (z^2 + (16/3)*z - (8/3))$
Solution 4	$(3*z -1) / (z^2 + (9/2)*z - (9/2))$
Solution 5	$3*z^{-1} - z^{-2}$
Solution 6	$-(3*z -1) / (-z^2 + (11/7)*z + (55/7))$
Solution 7	$((3/2)*z - (1/2)) / (z^2 -z + (1/2))$
Solution 8	$3*z^{-1} - z^{-2}$
Solution 9	$3*z^{-1} - z^{-2}$

All these solutions are simulated using Simulink tool of MATLAB. For this purpose, examples in [7] are modified and used. The quantizer used in these simulations has 1 bit resolution. If higher bit resolutions are used, much higher SNR values may be achieved.

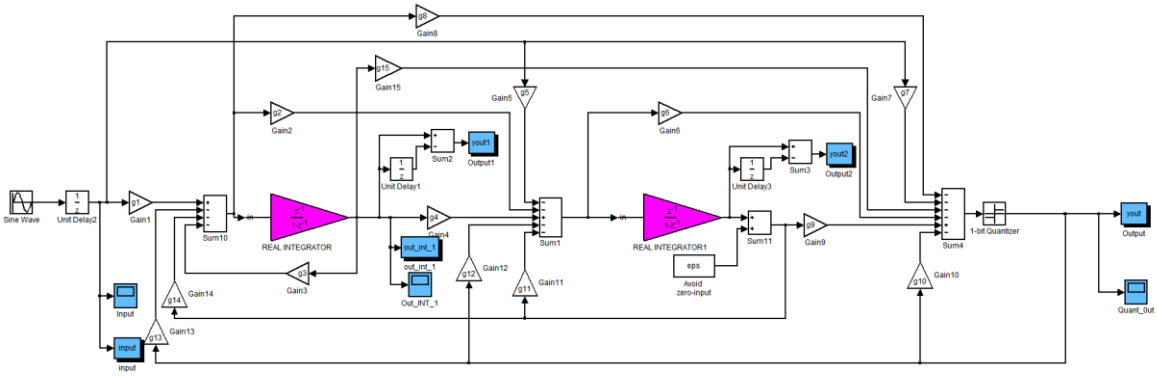

Figure 5.1. Simulink schematic used in simulations

The solutions are listed as AAG2 output, which means the solution with the least number of components is the first solution. So it is no surprise that Solution 1 has the least number of components, hence having the simplest topology. There are only feedbacks and no feedforwards in this solution.

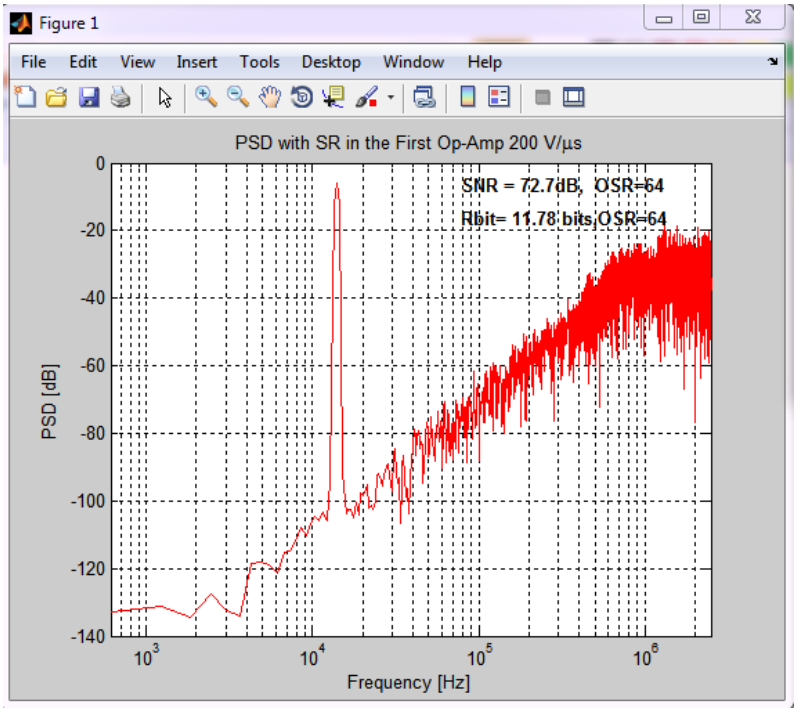

Figure 5.2. The FFT of the output

Although it seems good to have the least complex solutions, sometimes it may be better to find other solutions without making the topology complex. The advantage of this is the new solutions may result in better results. x8 NTF is still unrestricted, a solution with more components may have a much better x8 NTF suppression. Although this was not the case for this design, the user should keep this in mind. So it is advised to enter a positive number for N_COMPONENTS_MORE.

More parametric solutions also give a chance to have better numeric solutions.

Among the solutions, the most interesting one is Solution 4. There is no g9 component, meaning the output of the second integrator is not fed to the quantizer. In fact, the order of integrators can be considered in reverse order for this solution.

When examining the solutions, assuming undelayed, one delayed and two delayed signals are nearly the same and guessing the NTF performance may seem like a good approach but unfortunately it is not for the one bit quantizer case. Comparing Solution 1 and Solution 7, assuming z^{-1} (one delayed signal) and z^{-2} (two delayed signal) are both

equal to one (undelayed signal) results two in both Solution 1 and in Solution 7. But their NTF suppression performances are quite different and Solution 1 is much better.

After these solutions are found, in order to improve the design and find solutions for the case when both x8 STF and NTF are suppressed, extra components feature is used. With extra components feature, the output has been fed back with one and two delayed versions of itself. The extra components file was:

(EMPTY LINE!)
A B C
x6=(A*x22)+(B*(z^-1)*x22)+(C*(z^-2)*x22)

In this case, DAC is deleted from the netlist.

These extra components changes output feedback from just output to a linear combination of output, output with one delay and output with two delay.

Some desired function combinations have been tried for these extra components but no solutions were found. These cases are given in Table 5.5.

The next idea was to have two separate output feedback with delayed versions. For this purpose the extra components file used was:

(EMPTY LINE!)
A1 B1 C1 A2 B2 C2
x5=(A1*x22)+(B1*(z^-1)*x22)+(C1*(z^-2)*x22),x11=(A2*x22)+(B2*(z^-1)*x22)+(C2*(z^-2)*x22)

g12, g13 and DAC are deleted from the netlist for this case. AAG2 was run for the same combinations of desired functions given in Table 4.5, but unfortunately no solutions were found.

Table 5.6. Desired functions and corresponding number of solutions for modified netlist.

OUT STF	OUT NTF	x8 STF	x8 NTF	# of Solutions
z^{-2}	$(1-z^{-1})^2$	$(1-z^{-1})^2$	$z^{-1}*(1-z^{-1})$	0
z^{-2}	$(1-z^{-1})^2$	$(1-z^{-1})^2$	$-z^{-1}*(1-z^{-1})$	0

Adding more components increase running time a lot. If there will be no solution, the running time doubles with every new component. So searching for new solutions became more time demanding by adding more delayed feedback paths. Since the required time for each run without a crash exceeded one day, and there are much more combinations of desired functions to check after delayed feedback additions, not all of the combinations are tested. The most possible desired functions tested and no solutions found. But as a future work, with some educated guess and luck, some solutions may be found for suppressing both x8 STF and x8 NTF.

5.1. Solution 1

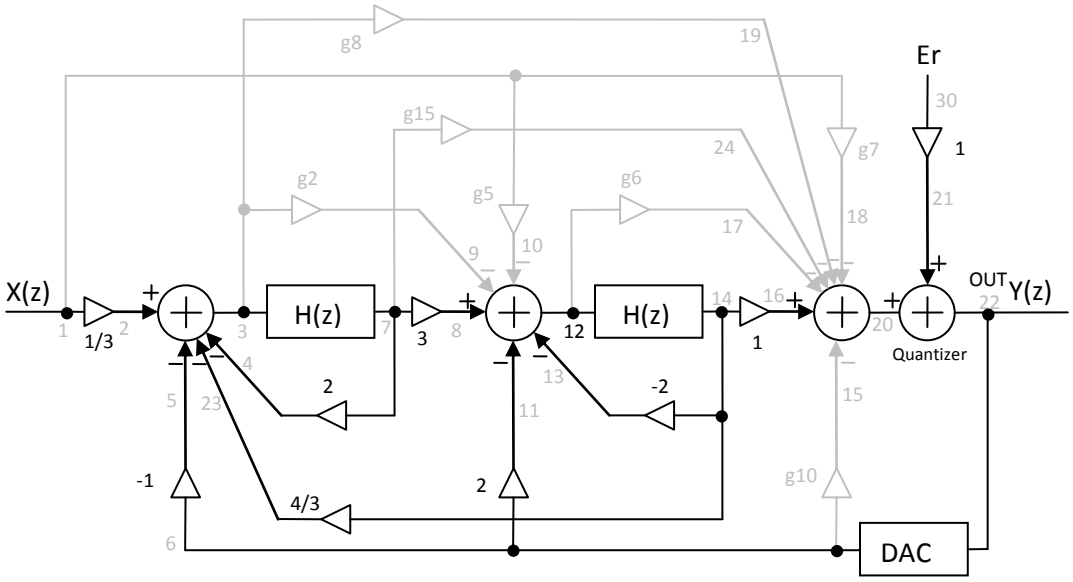

Figure 5.3 Solution 1.

min(out_int_1) = -1.9671 max(out_int_1) = 2.0712

mean(abs(out_int_1)) = 0.5261 x8 NTF = $3*z^{-1} - z^{-2}$

5.2. Solution 2

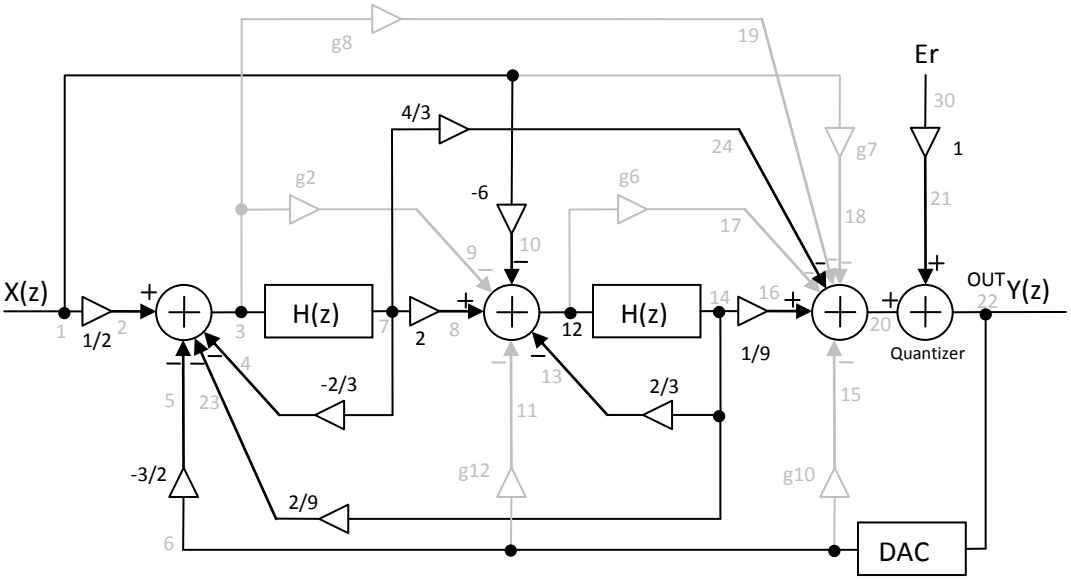

Figure 5.4. Solution 2.

min(out_int_1) = -3.0625 max(out_int_1) = 3.0949

mean(abs(out_int_1)) = 0.7949 x8 NTF = $3*z^{-1} - z^{-2}$

5.3. Solution 3

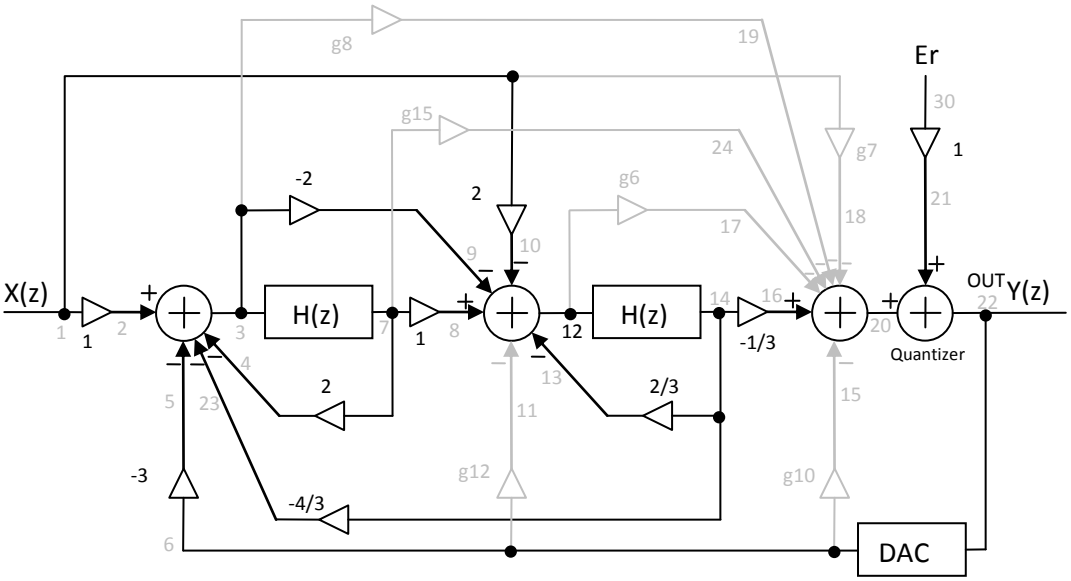

Figure 5.5. Solution 3.

min(out_int_1) = -5.0605 max(out_int_1) = 6.2785

mean(abs(out_int_1)) = 1.5694 x8 NTF = $(3*z -1) / (z^2 + (16/3)*z - (8/3))$

5.4. Solution 4

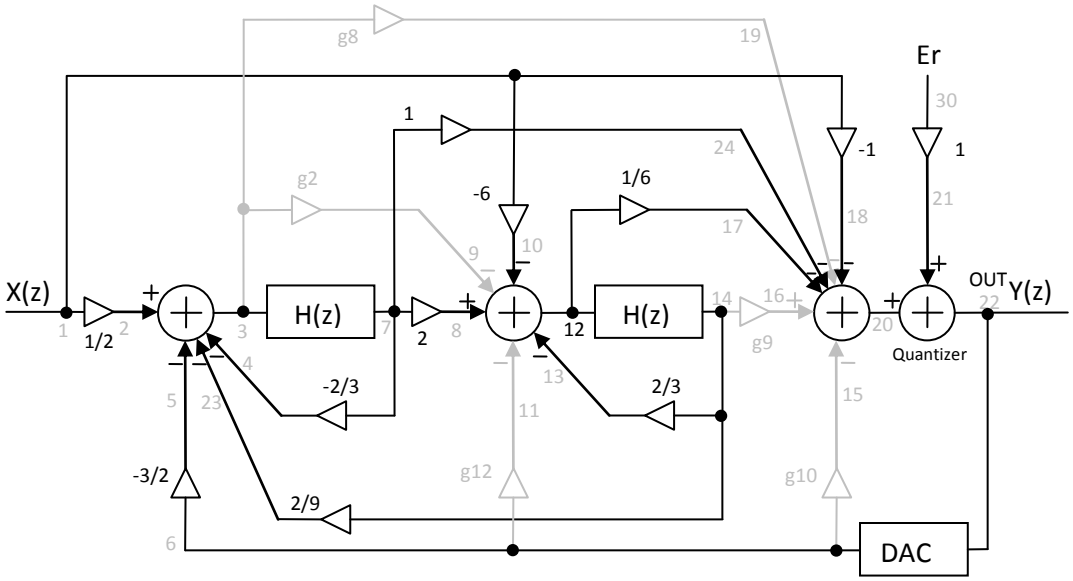

Figure 5.6. Solution 4.

min(out_int_1) = -3.0625 max(out_int_1) = 3.0949

mean(abs(out_int_1)) = 0.7949 x8 NTF = $(3*z -1) / (z^2 + (9/2)*z - (9/2))$

5.5. Solution 5

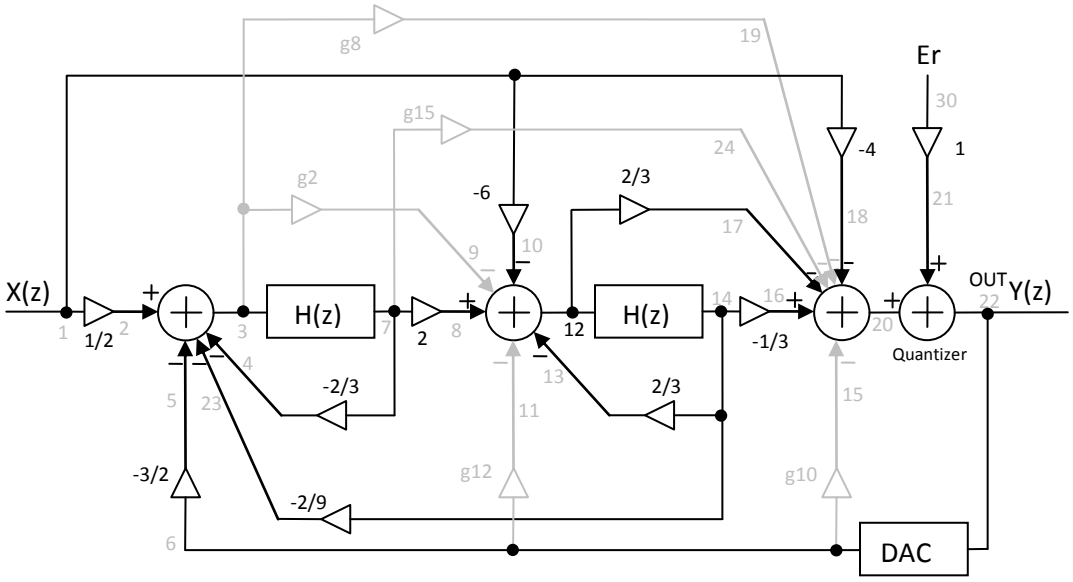

Figure 5.7. Solution 5.

min(out_int_1) = -3.0625 max(out_int_1) = 3.0949

mean(abs(out_int_1)) = 0.7949 x8 NTF = $3*z^{-1} – z^{-2}$

5.6. Solution 6

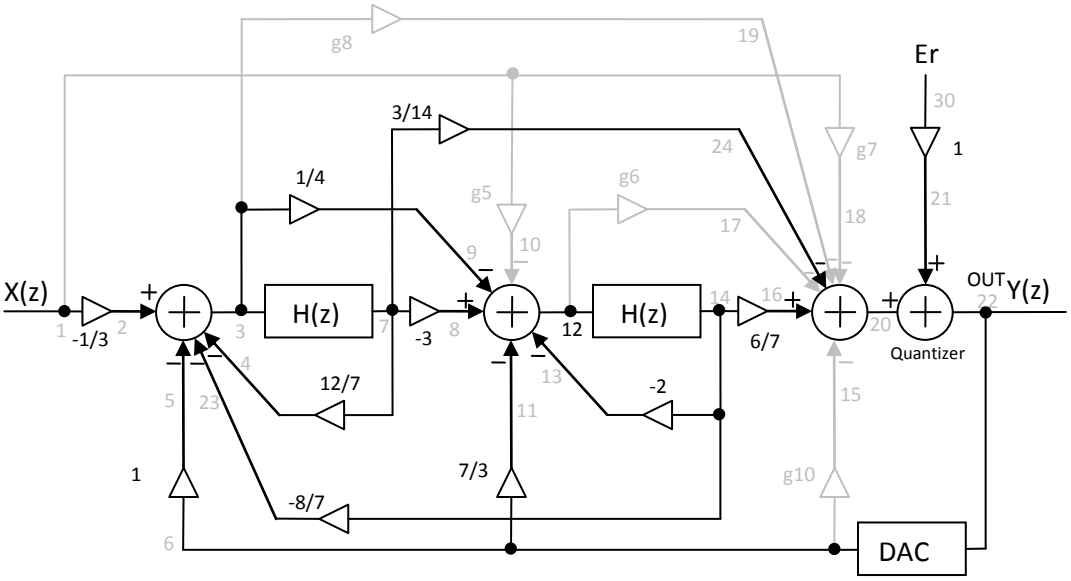

Figure 5.8. Solution 6.

min(out_int_1) = -2.0734 max(out_int_1) = 2.0595

mean(abs(out_int_1)) = 0.5266 x8 NTF = $-(3*z -1) / (-z^2 + (11/7)*z + (55/7))$

5.7. Solution 7

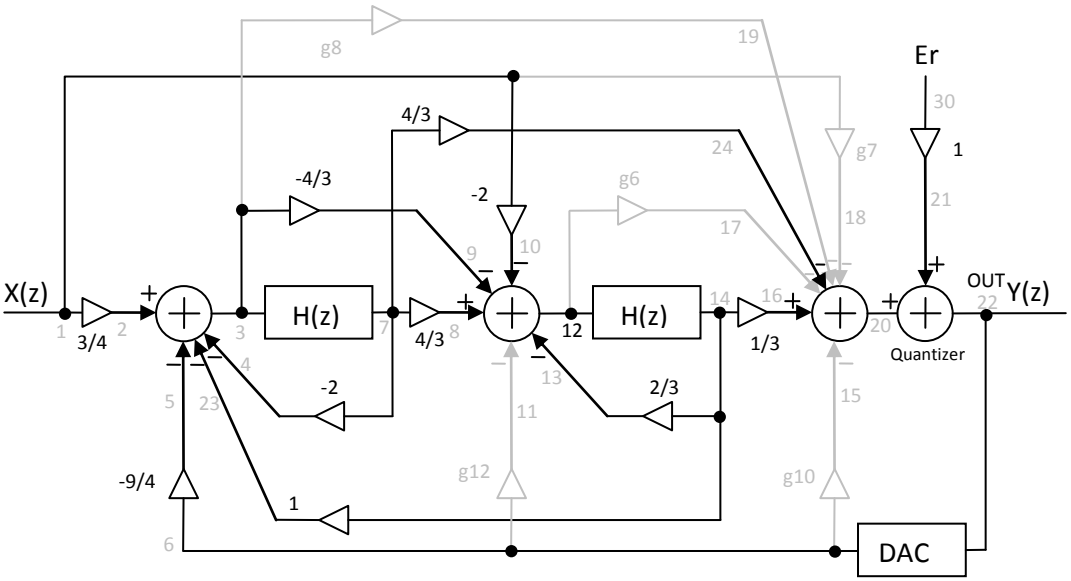

Figure 5.9. Solution 7.

min(out_int_1) = -4.3838 max(out_int_1) = 3.9028

mean(abs(out_int_1)) = 1.1886 x8 NTF = $((3/2)*z – (1/2)) / (z^2 -z + (1/2))$

5.8. Solution 8

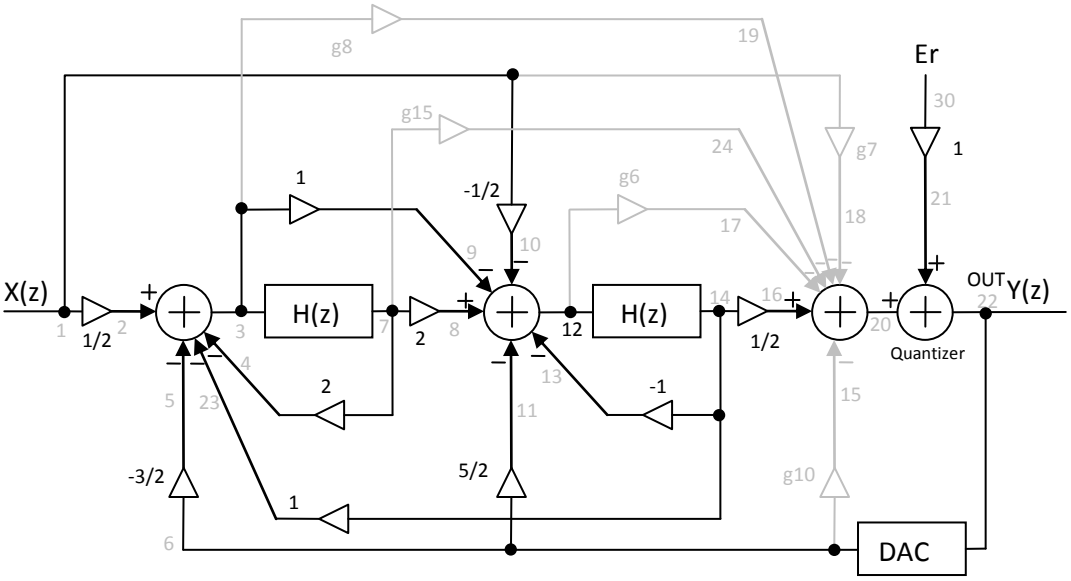

Figure 5.10. Solution 8.

min(out_int_1) = -2.9997 max(out_int_1) = 3.2828

mean(abs(out_int_1)) = 0.7893 x8 NTF = $3*z^{-1} - z^{-2}$

5.9. Solution 9

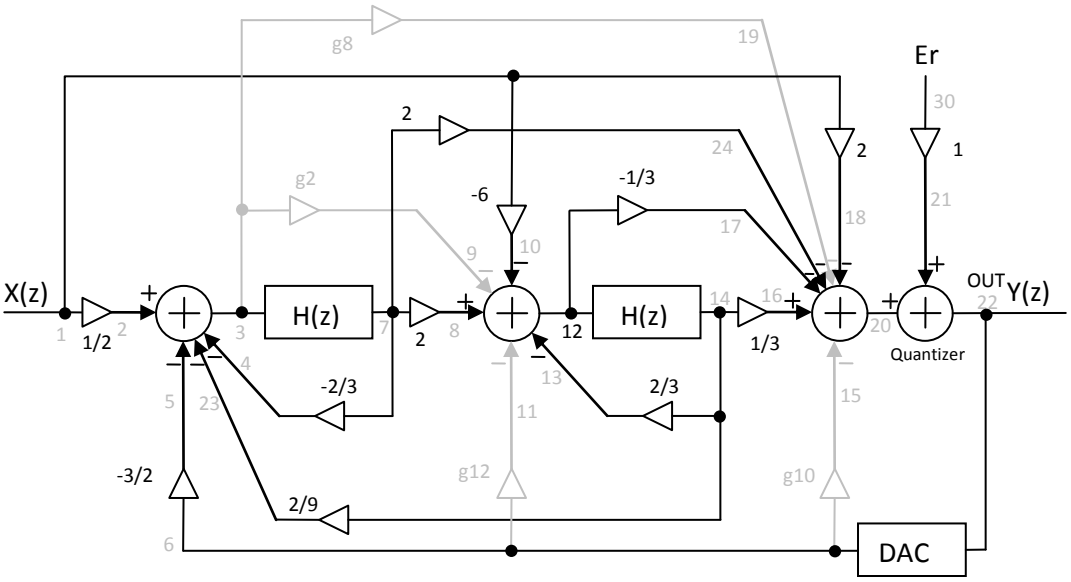

Figure 5.11. Solution 9.

min(out_int_1) = -3.0625 max(out_int_1) = 3.0949

mean(abs(out_int_1)) = 0.7949 x8 NTF = $3*z^{-1} - z^{-2}$

6. SPICE RESULTS

The solution in Figure 4.3 is designed and simulated in SPICE. The design is shown in Figures 6.1, 6.2 and 6.3.

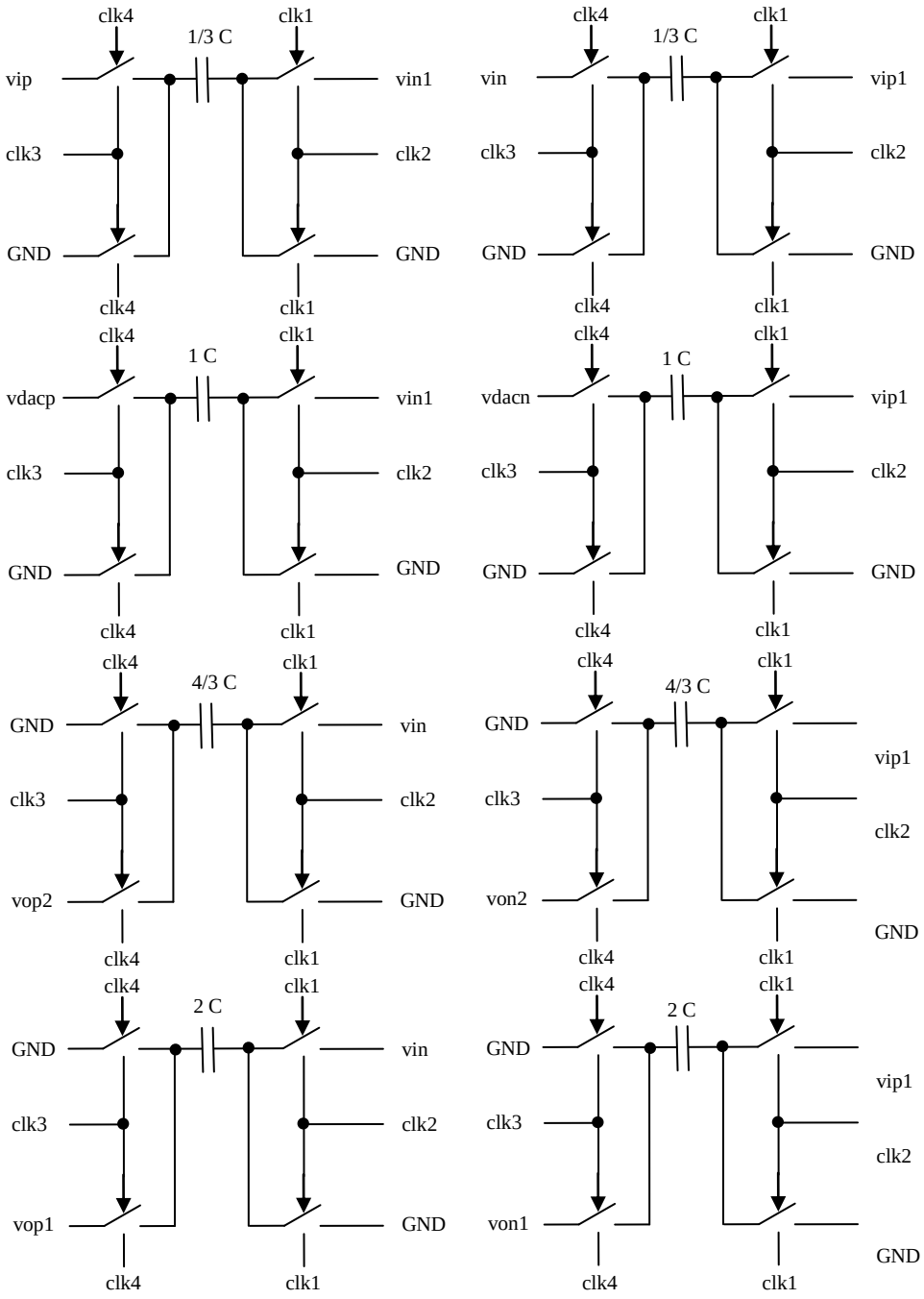

Figure 6.1. The first integrator inputs.

Switched capacitors are used to implement the gain components and adders. 1 *pF* is used as C value.

There are four clock signals but"clk3" is just a slightly delayed version of "clk1" and "clk4" is just a slightly delayed version of "clk2". As a requirement for switched capacitor design to operate correctly, "clk1" and "clk2" are never high at the same time. Similarly "clk3" and "clk3" are also never high at the same time.

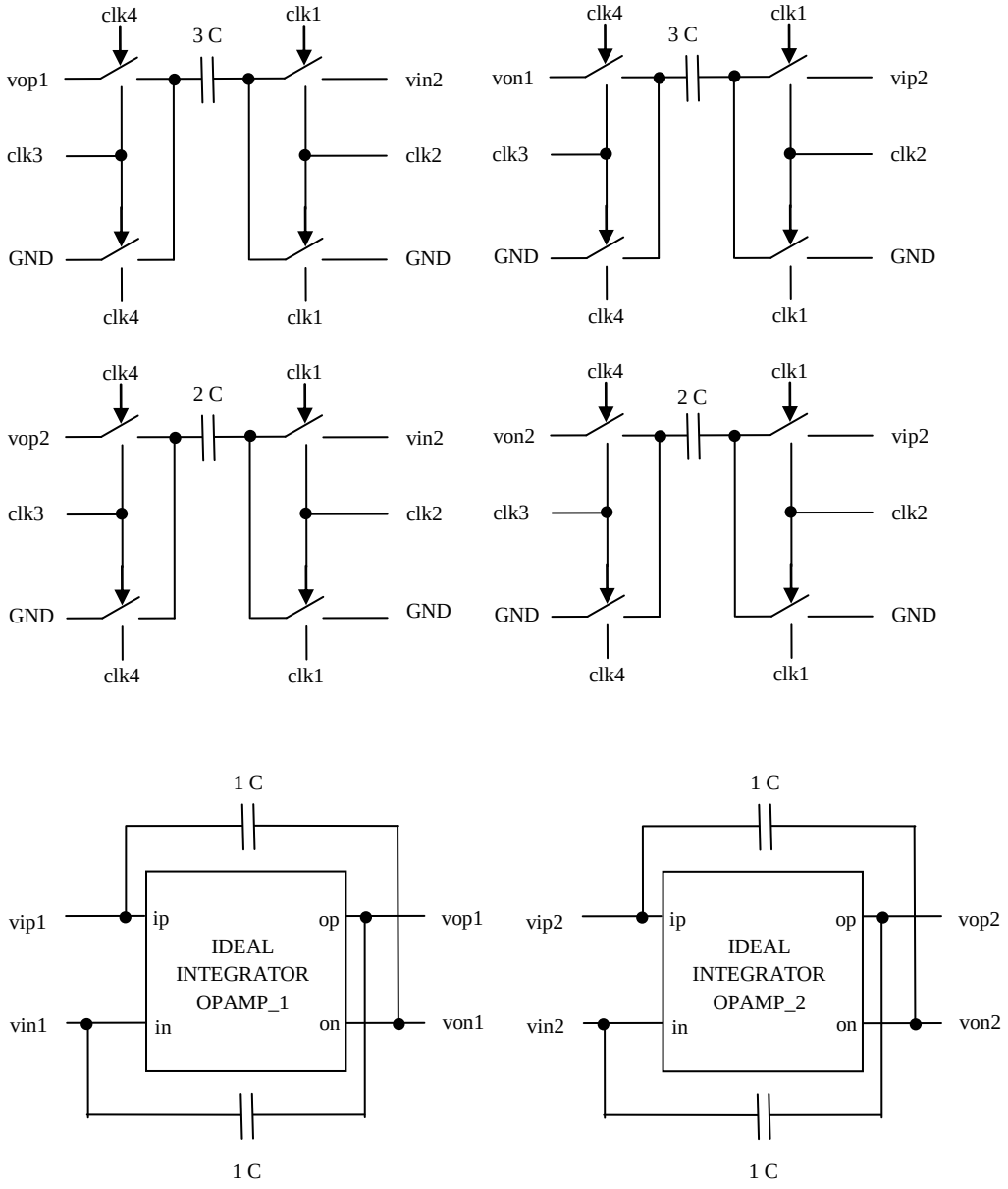

Figure 6.2. The first integrator, the second integrator and its inputs.

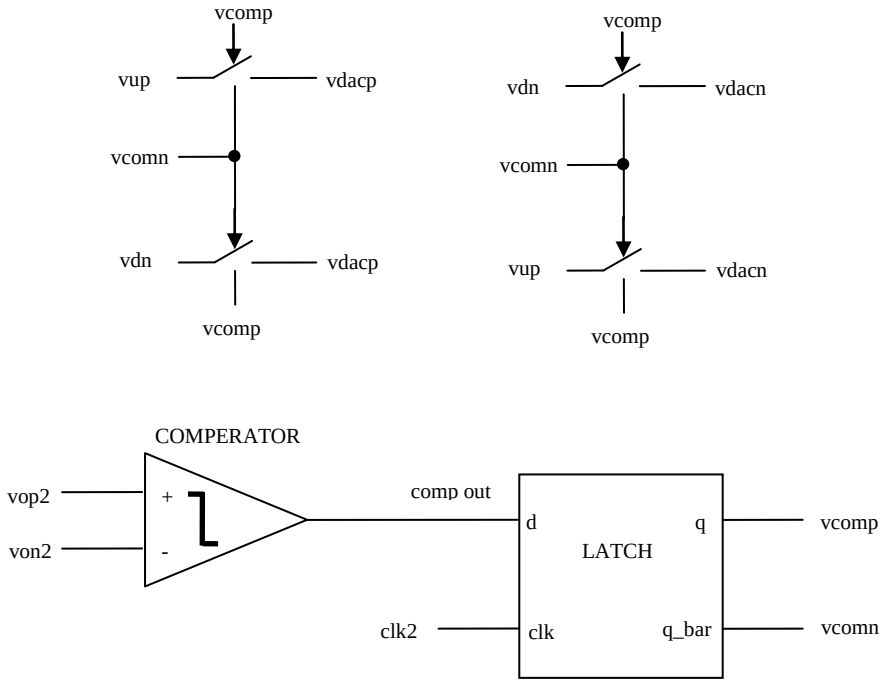

Figure 6.3. The comperator and the rest of the components.

For spice simulation, ideal components are used. After the simulation done, the input and the output waveforms are shown in Figure 6.4. Since the whole design is done differentially, there are 2 ports for each node: POSITIVE and NEGATIVE. VIN is the input , VCOMN is the output, VON1 is the output of the first integrator, VON2 is the output of the second integrator NEGATIVE values. POSITIVE values are just the opposite signed values. The output is usually stays at 0V when the input is low and usually stays at 5V when the input is high. There is less change in the output when there is less change in the input and there is more change in the output when there is more change in the input. This is compatible to how a sigma delta ADC operates. Average integrator output swings are calculated using calculator functions of Mentor and found as:

avg(abs(wf("<ff2_ideal_eldonet/TRAN>V(VON1")))=0.363

avg(abs(wf("<ff2_ideal_eldonet/TRAN>V(VON2")))=0.72

After the simulation is done, the output waveform is saved into a file and a MATLAB file is run to acquire the SNR value. This is shown in Figure 6.5. The SNR value is 63 dB.

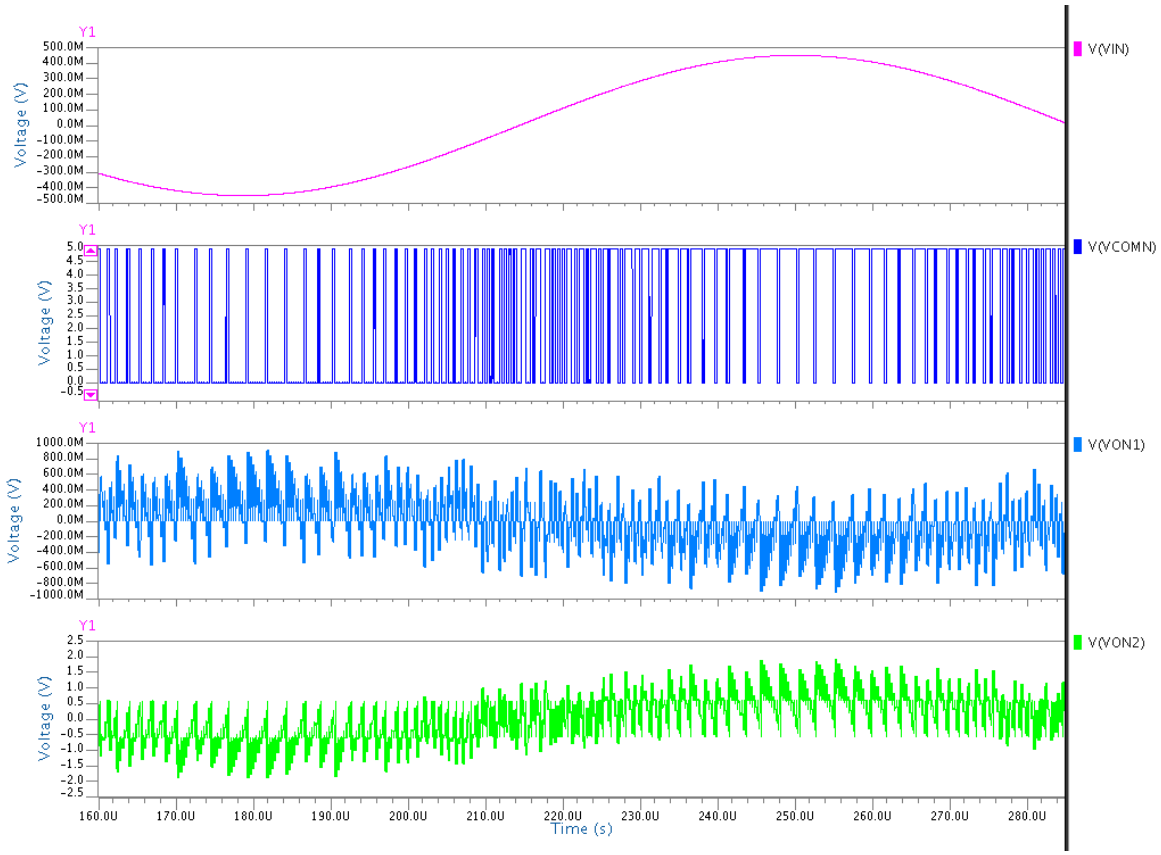

Figure 6.4. The input and the output waveforms.

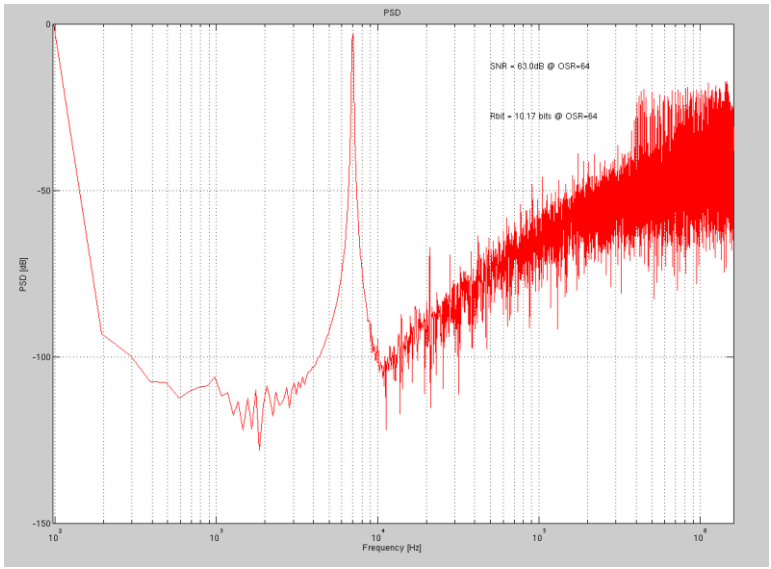

Figure 6.5. FFT of the output.

7. CONCLUSION

The software AAG2 devaloped in this thesis accomplishes finding both parametric and numeric solutions almost fully automatically. However, since there are a lot of combinations of possibilities, the user should truly have a grasp of sigma delta ADCs in order to avoid huge waste of time. There are a lot of different combinations of desired functions (and corresponding nodes), netlists and extra components to be used. Although the tool checks every possible solution and runs quite fast bypassing unnecessary commands, the runtime is still at the order of several hours, given a netlist with around 15 components. Each new component have a potential to double the running time.

AAG2 was run for putting different conditions on both STF and NTF of the output of the first integreator in order to decrease power consumption. When the general second order sigma delta ADC netlist is used, there were no solutions found. In order to find a solution, extra components were added to feedback paths which would allow different linear combinations of output, output with one delay and output with two delays to be fed back. Still there were no solutions found so far. Since there are many possibilities of desired functions, netlists and extra components; not all possibilities had been tried yet.

When the first integrator output NTF is released, 9 different solutions were found. These solutions are given in Chapter 4.

AAG2 functions perfectly and as proposed. As a feature, the user may choose to run it in separate steps. Running AAG2 in separate steps helps to protect against computer or MATLAB crashes. These crashes may cause the solutions to be lost.

AAG2 requires the user to interfere manually only in between the steps. When “Save TFs” step is done the user is required to check the TFs of required nodes as explained in Chapter 2. This manual step is required and without this step, some solutions may be missed. The second manual interference is up to the user. It is recommended that the user checks the parametric solutions before running AAG2 for numeric solutions.

To summarize, AAG2 is working as proposed, but because of the nature of sigma delta ADC topologies, the user should have a comprehensive knowledge of the subject. Otherwise although AAG2 runs perfectly, the user may not be able to find solutions. This is, as explained above, because usually only a few of many possibilities of input parameters (including netlist, desired functions and extra components) have solutions.

REFERENCES

1. Aziz, P. M., H. V. Sorensen and J. V. D. Spiegel, "An overview of sigma-delta converters", *Signal Processing Magazine, IEEE* , vol.13, no.1, pp.61-84, Jan 1996.

2. Anonymous, *Delta-Sigma Modulation*, 2012, http://en.wikipedia.org/wiki/Sigma_delta, accessed at May 2012.

3. Yetik, Ö., *An Automated Architecture Generator For Sigma-Delta Modulators Considering Component Non-Idealities*, Boğaziçi University, İstanbul, 2007.

4. Contadini, F., *Demystifying Delta-Sigma ADCs*, 2012, http://www.maxim-ic.com/app-notes/index.mvp/id/1870, accessed at May 2012.

5. Medeiro, F., B. Perez-Verdu and A. Rodriguez-Vazquez, *Top-Down Design of High-Performance Sigma-Delta Modulators*, Kluwer Academic Publishers, Dordrect, 1999.

6. Brigati, S., F. Francesconi, P. Malcovati, D. Tonietto, A. Baschirotto, F. Maloberti, "Modeling sigma-delta modulator non-idealities in SIMULINK(R)", *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium*, vol.2, no., pp.384-387 vol.2, Jul 1999.

7. Maloberti, F., *Integrated Micro Systems*, 2012, http://ims.unipv.it/Microelettronica/Dataconverters1.html, accessed at May 2012.

8. Maloberti, F., *Data Converters*, Springer, Dordrect, 2008.

9. Liu, M., *Demystifying Switched-Capacitor Circuits*, Newnes, Oxford, 2006.

10. Rabii, S. and B. Wooley, *The Design of Low-Voltage, Low-Power Sigma-Delta Modulators*, Kluwer Academic Publishers, Dordrect,1999.

11. Unbehauen, R. and A. Cichocki, *MOS Switched-Capacitor and Continuous-Time Integrated Circuits and Systems*, Springer-Verlag, New York, 1989.

12. Norsworthy, S., R. Schreier and G. Temes, *Delta-Sigma Data Converters : Theory, Design, and Simulation* , IEEE Circuit & Systems Society IEEE Press, Hoboken, 1997.

13. Gregorian, R., K. W. Martin and G. C. Temes, "Switched-capacitor circuit design", *Proceedings of the IEEE* , vol.71, no.8, pp. 941- 966, Aug. 1983.

14. Franco, S., *Design with Operational Amplifiers and Analog Integrated Circuits*, Mc-Graw Hill, New York, 2002.