TSTE87 Laboratory Work – Lab 1

Oscar Gustafsson, Kenny Johansson, and Erik Bertilsson

- Student name: ....
- Student personal id: ....
- Passed: ....

Goals

To get an overview of the DSP toolbox for MATLAB. Use the DSP toolbox to simulate signal flow graphs and investigate the scaling and overflow problems using finite word length. Compute precedence graphs and investigate the effect of pipelining.

Guidelines

Write proper code for the labs. For example, it should not be required to comment code in/out when showing it to the lab assistants. The Matlab editor has a quite good code alignment button. It is possible to use sections in the Matlab code by starting a line with `%%`. Pressing ctrl+⏎ will execute the code in the current section.

Preparations

1. Read Chapters 3–5 in Wanhammar, DSP Integrated Circuits.
2. Read "Introduction to the DSP toolbox" below.

Online Mode

Use Teams and the team "TSTE87 2021VT Application-Specific Integrated Circuits" to communicate with the lab assistants.

To work remotely, follow the instructions below. A good ssh-client for Windows is MobaXTerm, `https://mobaxterm.mobatek.net/`.

1. Log in to LiU using ThinLinc or SSH (see link on Lisam page), do NOT use RDP

2. Log in to a computer in Muxen 1 or 2 using ssh with compression and X-forwarding:
   `ssh -XC muxenY-0ZZ.ad.liu.se`
   where `Y` is `1` or `2` and `ZZ` in `01` to `16`

Introduction to the DSP toolbox

The DSP toolbox provides a way of manipulating and simulating signal flow graphs, and functions to evaluate properties of signal flow graphs. The signal flow graph can later on be turned into a computational graph (where the operations have related timing information) for synthesis of architectures.

Signal flow graph (SFG) creation

The signal flow graph (SFG) is constructed by adding operations connected to nodes, as in, e.g., the circuit simulator SPICE. In MATLAB the SFG is represented as an array. However, there are functions for manipulating the SFG, so there is no need to have an in-depth knowledge about the format. An initial SFG is obtained using an empty matrix

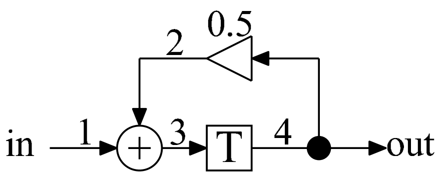
```
sfg = []
```

To add an operation, the function addoperand can be used as follows

```
sfg = addoperand(sfg, operandname, id-number, innodes, outnodes,
                 operanddata, operandtype)
```

where `sfg` is the SFG to add the operand to, `operandname` is the name of an operand (see Appendix for available operands), `id-number` is an identifying number for the operation, `innodes` is one or more nodes to connect to the input(s), `outnodes` similarly for the output(s), `operanddata` is an optional data for certain operations (e.g. the coefficient for a multiplier or WDF adaptor), while `operandtype` is used for certain operands which may have several different types (e.g. different types of quantization). More on possible `operanddata` and `operandtype` are available in the Appendix.

To see how we can create an SFG, let us use the simple filter below, where we have numbered the nodes 1 to 4. Each operand also has an identifying number, but as only operands of the same type must have different numbers all operands are using 1 as the identifying number.



To create the corresponding SFG we can use the following MATLAB code

```
sfg = [];
sfg = addoperand(sfg, 'in', 1, 1);
sfg = addoperand(sfg, 'add', 1, [1 2], 3);
sfg = addoperand(sfg, 'delay', 1, 3, 4);
sfg = addoperand(sfg, 'constmult', 1, 4, 2, 0.5);
sfg = addoperand(sfg, 'out', 1, 4);
```

To obtain the SFG in readable form we can use the command `printsfg` as

```
printsfg(sfg)
```

which will give an output of

```
1. in        id: 1   out: 1
2. add       id: 1   in: 1, 2   out: 3
3. delay     id: 1   in: 3      out: 4
4. constmult id: 1   in: 4      out: 2   coeff: 0.5
5. out       id: 1   in: 4
```

Always make sure that the created SFG is valid, i.e., that no nodes have multiple drivers and no dangling (unconnected) nodes are included, by using the function

```
errorlist = checknodes(sfg)
```

For a correct SFG the result, `errorlist`, is empty, otherwise a description of the error(s) is printed at the prompt.

Simulation

Now, we would like to simulate the SFG. For this the function `simulate` is available. In its simplest form it is used as

```
output = simulate(sfg, inputvalues)
```

where `inputvalues` is the sequence of samples to be used as input.

Two commonly used input data are an impulse and random data (uniformly distributed on the interval $[-1, 1]$) which can be obtained as

```
impulse = [1, zeros(1,N-1)]
random = 2*rand(1,N)-1
```

In its most complex form `simulate` is used as

```
[outputs, outputids, delays, delayids, nodes, nodeids] =
simulate(sfg, inputvalues, inputids, initialdelays, delayids, wordlength)
```

where it has support for multiple inputs and outputs, initializing delay elements, and tracking values in delay elements and nodes. If the `wordlength` input value is specified, the simulation will use finite word lengths for the nodes. The `wordlength` input can be either `[Wi Wf]`, where `Wi` denotes the number of integer bits and `Wf` fractional bits, or `Wf`, where it is assumed that `Wi` is 1. Note that `initialdelays` and `delayids` can be set to an empty vector `[]` if they are not needed, but you want to set the word length.

The `outputs` variable contains a 2D array, where each row contains the output values of the output with identifying number given at the same row of `outputids`. Similarly, `delays-delayids` and `nodes-nodeids` have the same structure. To obtain the values of a certain output/delay/node the function `getnodevalues` can be used as

```
values = getnodevalues(outputs, outputids, outputidentifier)
values = getnodevalues(delays, delayids, delayidentifier)
values = getnodevalues(nodes, nodeids, nodenumber)
```

The functions `impulseresponse` and `stepresponse` are available for computing the impulse response and step response, respectively. They are used as

```
output = impulseresponse(sfg, numberofsamples)
```

where `numberofsamples` is the number of samples of the impulse response to be computed.

SFG modification

For scaling and pipelining purposes it is possible to insert a single input-single output operation at a node. This is done with `insertoperand` as

```
sfg = insertoperand(sfg, operandname, idnumber, node, operanddata,
                    operandtype)
```

To insert a pipeline delay at the input of our previous SFG we would type

```
sfg = insertoperand(sfg, 'delay', 2, 1)
```

Note that `insertoperand` adds a new node to the SFG for the input of the new operand.

For the case when a node is connected to more than one destination operands and the new operand should only be connected to one of the destination operands, there is a longer form available as

```
sfg = insertoperand(sfg, operandname, idnumber, node, operanddata,
                    operandtype, destinationoperandname,
                    destinationidnumber, destinationinputnumber)
```

where `destinationoperandname` and `destinationidnumber` is the name and id of the operand to insert the new operand before. `destinationinputnumber` is the sequential input number, a two-input adder has two inputs: 1 and 2, of the operand that the new operand should be connected to. However, this is only needed when a node is connected to more than one input of an operand and the new operand should only be inserted to one of the inputs. If the new operand does not have `operanddata` or `operandtype`, just leave and empty array `[]` for those arguments.

If you need to change the SFG further, you can use the functions `removeoperand` (opposite to `addoperand`) or `changeoperand`, which are used as

```
sfg = removeoperand(sfg, operandname, idnumber)
sfg = changeoperand(sfg, operandname, idnumber, operanddata, operandtype)
```

There is also one function to replace a single input-single output operation with a short circuit, i.e., the opposite to `insertoperand`. It is used as

```
sfg = deleteoperand(sfg, operandname, idnumber)
```

Some operations, e.g., butterflies and adaptors, are composed of lower level operations (additions and multiplications). It is possible to transform these operations to the lower level operations using the function `flattensfg`, which is used as

```
flatsfg = flattensfg(sfg)
```

Precedence form

To see the precedence relations between operands there are two functions available, `printprecedence` and `plotprecedence`. Both functions takes an SFG as input.

Executing `printprecedence(sfg)` to the example SFG with an inserted delay yields the following output

```
-----------------------------------------------------
1.1  in        id: 1   out: 1
-----------------------------------------------------
2.1  constmult id: 1   in: 4      out: 2   coeff: 0.5
2.2  out       id: 1   in: 4
-----------------------------------------------------
3.1  add       id: 1   in: 5, 2   out: 3
-----------------------------------------------------
4.1  delay     id: 1   in: 3      out: 4
4.2  delay     id: 2   in: 1      out: 5
-----------------------------------------------------
```

The functions `printprecedence` and `plotprecedence` also returns an SFG sorted in precedence order.

Generating an image of the SFG

To more easily see and identify the new node numbers after flattening or inserting new operands, there is a command `dotsfgplot` to generate an image of the SFG. This is based on the graph drawing toolkit Graphwiz, so this must be available. In its simplest form, the command can be executed as

```
dotsfgplot(sfg)
```

Invoked in this form, the program shows the generated .eps-file (as it currently invokes `gv` you may want to select “pixel-based” in the scaling menu to make it look better). It can also be invoked as

```
filename = dotsfgplot(sfg)
```

where the name of the generated file is returned for external viewing.

It is also possible to generate a .dot-file for further modifications as

```
filename = dotsfgplot(sfg, 'dot')
```

where now the name of the .dot-file is returned.

Since automatically drawing graphs in a “nice” way is a fundamentally hard problems, there are a few different ways to draw graphs provided by Graphviz. It may be worthwhile trying:

```
dotsfgplot(sfg, 'eps', 'neato')
```

to use the `neato` command or

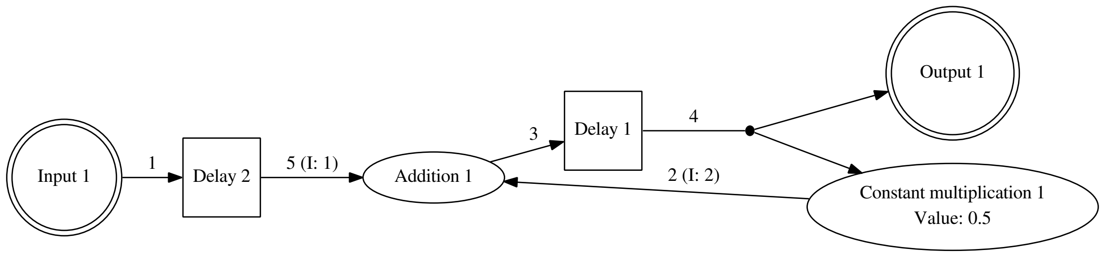
```
dotsfgplot(sfg, 'eps', 'fdp')
```

to use `fdp`.

Finally, it is possible to provide a filename (the correct extension is automatically added) as (with default settings)

```
dotsfgplot(sfg, 'eps', 'dot', 'myfilename')
```

The result can be seen below for the default setting. As can be seen, the edges include the node numbers and there is an indication which input of the operation (where applicable) a specific node is connected to.



Tasks

Never execute commands at the prompt. Instead, write all the commands in a textfile and run that file. You will have to use some of the functions in the Signal Processing Toolbox, e.g., `freqz`.

1. Start MATLAB and add the path to the DSP toolbox in MATLAB. This can be done using

   ```
   addpath /courses/TSTE87/matlab/
   ```

   To save the path for later laboratory works you can use `savepath`

2. Simulate and plot the impulse response of the example in "Introduction to the DSP toolbox" above. stem is a good function for plotting discrete signals. Also plot the frequency response (hint: use `freqz` for a long impulse response). The SFG is available in the file

   ```
   /courses/TSTE87/labs/lab1/introduction.m
   ```

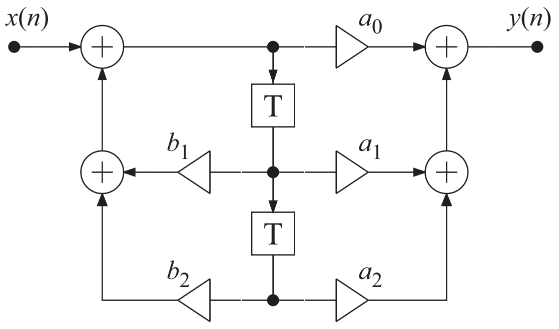
   (a) Give the exact value of all samples in the impulse response that are larger than 0.1.

   . . . .

   (b) What type of filter is this (lowpass, highpass, bandpass, or bandstop)?

   . . . .

3. Consider the following second-order direct form II IIR filter, where $a_0 = a_2 = 57/256$, $a_1 = 55/128$, $b_1 = 179/512$, $b_2 = -171/512$. The objective is to study the precedence relations, to scale the filter, and to study effects of overflow.



(a) Number the nodes and operations and create the signal flow graph.

(b) Print and plot the precedence graph.

(c) Simulate the filter using an impulse.

(d) What is the passband edge? $A_{\max} = 1.023$ dB

. . . .

(e) How large is the stopband attenuation? $\omega_s T = 0.88\pi$ rad

. . . .

(f) Plot the discrete values of all interesting nodes.

(g) Identify nodes that are possibly badly scaled under the safe scaling criteria, i.e., critical nodes where the sum of absolute node values are greater than one.

. . . .

(h) Scale all critical nodes using safe scaling only using power-of-two scaling coefficients. Indicate where you introduce scaling.

(i) Simulate the filter again. Are the nodes correctly scaled? What is the sum of the absolute node values in the critical nodes?

. . . .

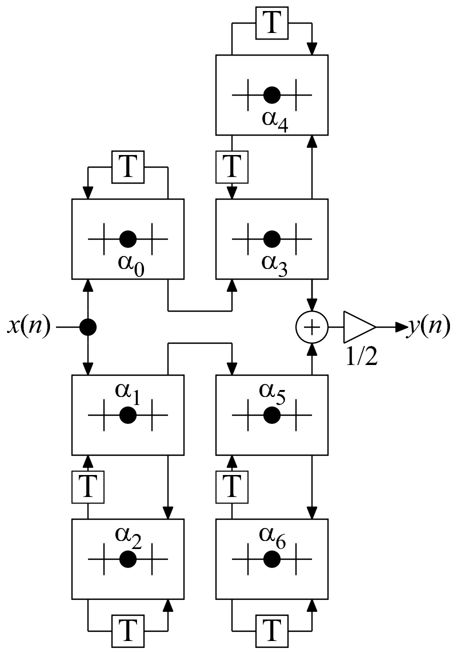
(j) How does scaling affect the magnitude response?

. . . .

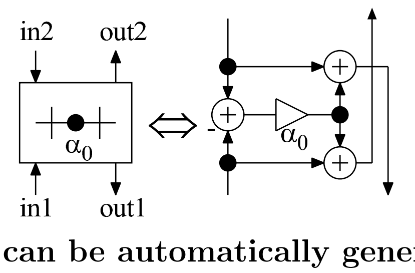
(k) Simulate the original and scaled filters using the same random data for both. **Use 1 integer bit and 15 fractional bits.** Compare the discrete outputs by plotting them in the same figure with different markers, e.g., `'x'` and `'o'`, respectively (use `hold on` to plot several plots in the same figure). Comments?

. . . .

4. In this problem we consider a WDF based on first- and second-order allpass sections using symmetric two-port adaptors as shown below. The coefficients are $\alpha_0 = 167/256$, $\alpha_1 = -135/256$, $\alpha_2 = 1663/2048$, $\alpha_3 = -1493/2048$, $\alpha_4 = 669/1024$, $\alpha_5 = -117/128$, $\alpha 6 = 583/1024$.



   (a) Number the nodes and operations and create the signal flow graph. The inputs and outputs of the symmetric two-port adaptor are related and numbered as shown below. Note that the position of the adaptor coefficient, α_0, denotes which input to be subtracted.



   Alternatively, **the SFG can be automatically generated** by

   `sfg = sfg_LWDF(coeff)`

   where `coeff` is a vector containing the adaptor coefficients in order as

   `coeff = [alpha0, alpha1, ..., alpha6]`

   Using this approach means that you will have to figure out the node numbers of the interesting nodes on your own.

   (b) Simulate the filter using an impulse and check the values of all interesting nodes. Comments?

. . . .

(c) Plot the phase-response of the two allpass branches in the same plot. The command `angle` determines the phase of the frequency response, which can be obtained as output from `freqz`. Alternatively, `phasez` can be used to directly obtain the phase response from the impulse response.

(d) How large is the difference in phase in the pass- and stopband?

. . . .

(e) Scale the filter using L_2-scaling (only use power-of-two scaling coefficients). Indicate where you introduce scaling. Note that all four sections should have as large dynamic range as possible.

(f) Simulate the scaled filter. What is the value of the L_2-sum in the different nodes? Are the nodes correctly scaled? How can you verify that the filter function is not changed?

. . . .

(g) Simulate the scaled filter with a random input. Comments? Can overflow occur in any node?

. . . .

(h) Introduce a pipelining delay in each branch and compare the output with the non-pipelined SFG. Comments?

. . . .

(i) Plot the precedence graph for the pipelined SFG.

(j) Flatten the SFG and plot the precedence graph.

Appendix

The following operations are available and can be used as an input to `addoperand`:

`operandname`	**Description**	**Operation**
`'in'`	Input of SFG, one output node.	
`'out'`	Output of SFG, one input node.	
`'add'`	Addition of two inputs	$y_1 = x_1 + x_2$
`'sub`	Subtraction of the second input from the first input.	$y_1 = x_1 - x_2$
`'constmult'`	Multiplication with constant coefficient `operanddata`.	$y_1 = \texttt{operanddata} \times x_1$
`'delay'`	Delay element delaying the signal one sample.	
`'quant'`	Quantization to `operanddata` fractional bits. Uses `operandtype`: ’`truncation`’, ’`rounding`’, or ’`magnitudetruncation`’.	
`'overflow'`	Overflow detection. Uses `operandtype`: ’`twosc`’ or ’`saturation`’.	
`'negate'`	Multiplication by -1.	$y_1 = -x_1$
`'shift'`	Shift by `operanddata` bits. Direction controlled by `operandtype`: `'left'` or `'right'`.	
`'twoport'`	Two-port WDF adaptor with coefficient `operanddata`. Uses `operandtype`: `'series'`, `'parallel'`, or `'symmetric'`.	
`'threeport'`		
`'fourport'`		
`'butterfly2'`	Radix-2 butterfly operation.	$\begin{aligned} y_1 &= x_1 + x_2 \\ y_2 &= x_1 - x_2 \end{aligned}$
`'constdiv'`	Division with constant coefficient `operanddata`.	$y_1 = \frac{x_1}{\texttt{operanddata}}$
`'mux'`		
`'demux'`		
`'constmac'`	Multiply-add with constant coefficient `operanddata`.	$y_1 = x_1 \times \texttt{operanddata} + x_2$
`'mult'`	Two-input multiplication.	$y_1 = x_1 \times x_2$
`'division'`	Two-input division.	$y_1 = \frac{x_1}{x_2}$
`'reciprocal'`	Reciprocal operation.	$y_1 = \frac{1}{x_1}$
`'mac'`	Multiply-add.	$y_1 = x_1 \times x_2 + x_3$
`'constant'`	Constant value.	$y_1 = \texttt{operanddata}$
`'addsub'`	Addition or subtraction depending on `operandtype`: `'add'` or `'sub'`.	$y_1 = x_1 \pm x_2$
`'conjugate'`	Complex conjugate.	$y_1 = x_1^*$
`'abs'`	Absolute value.	$y_1 = \lvert x_1 \rvert$
`'max'`	Maximal value.	$y_1 = \max\{x_1, x_2\}$
`'min'`	Minimal value.	$y_1 = \min\{x_1, x_2\}$
`'sqrt'`	Square root.	$y_1 = \sqrt{x_1}$
`'recipsqrt'`	Reciprocal square root.	$y_1 = \frac{1}{\sqrt{x_1}}$
`'square'`	Square.	$y_1 = x_1^2$