MSPICE Manual v2.0

Justin Reiher

Contents

Preface

MSPICE is based on the work of Yan [11] and has been extended as part of a MSc thesis by Reiher [8]. The original repository can be found at: `https://github.com/coho-tools/ccm`. The extension provides an implementation of the nested bisection algorithm for synchronizer metastability simulations developed by Yang & Greenstreet [12] and its analysis by Reiher *et al.* [6, 7]. Additional debugging, diagnostics and plotting routines have been added to facilitate information extraction. The extensions have been tested and run using MATLAB R2017b [4] and known to be working with MATLAB R2019b.

The transistor models are derived from the Predictive Technology Model (PTM) developed by Arizona State University [3]. In particular the examples and models found in this manual make use of the PTM 45nmHP model card.

This manual is intended to provide insight on how to create circuits, run simulations for general circuits and synchronizers. The options available to the user are outlined in this manual with detailed examples as guides to follow to get the user started on their own experiments.

The synchronizer analysis, small-signal model creation and optimization routines outlined in this manual require the Automatic Differentiation (AD) package found at `http://www.ti3.tu-harburg.de/intlab/` by Rump [9].

1 Introduction

The MSPICE code base is a MATLAB implementation of a circuit simulator such as LTSPICE [1], HSPICE [2] or ngSPICE [10]. MSPICE is open source and extensible to meet your needs. The version outlined in this manual seeks to facilitate further experiments and extensions by providing detailed explanations to the features and routines found within, while also providing insight on how one might go about making their own extensions.

MSPICE is not intended to compete with other open source or commercially available SPICE programs but is designed to be useful in the exploration of new circuit analysis techniques by providing a framework which gives users the ability to use the wealth of routines and functions found within MATLAB. The framework gives the user complete access to all data structures (with the appropriate methods) to be manipulated as seen fit. MSPICE should be viewed as a testing ground to try new algorithms and explore their consequences in an easy to debug framework before spending the effort in improving the computational efficiency in a language such as C/C++ or before extending other SPICE programs.

The manual is organized into quick start guides with instructions on how to use the framework as well as a comprehensive list of methods that explain what they do and how they work.

The intent of this manual is to be used as a reference with some detailed examples to unpack some of the steps necessary in using MSPICE. However I recommend browsing the files running some scripts and refer to the manual to obtain details about specific parts of the code base, or to consult it if there is information you want to obtain from a circuit, simulation or synchronizer metastability simulation that is not obvious. It is quite possible that what you are looking for has been implemented.

If it has not, please feel free to add your own methods and routines!

1.1 MSPICE Setup

First clone the repository to your computer.

```
git clone https://github.com/justinreiher/ccm.git
```

The above will clone the repository to your computer `./ccm`. All the contents of MSPICE and routines are found within.

1. Open MATLAB, from the command line run:
   `addpath(genpath('<<insert_your_path>>/ccm'))`
   Where `<<insert_your_path>>` is the path to the `ccm` folder which has just been cloned. This will add to the MATLAB path all folder and sub-folders within the folder `ccm`.

2. Alternatively you can use MATLAB's gui and click on the *Set Path* button under the *Home* tab, press *Add with Subfolders...* and navigate to where the `ccm` folder was cloned. Press *Save* and now when you open MATLAB you should have all `ccm` functions and routines available.

WARNING: If your MATLAB path includes the `RF Toolbox`, it will interfere with the MSPICE `circuit.m` class. To prevent this either remove the `RF Toolbox` or remove from MATLAB's path:

```
./MATLAB/201xx/mcr/toolbox/rf/rf
```

and save the updated MATLAB path.

2 MSPICE: Quick Start Guide Three Stage Ring Oscillator

Consider a simple three stage ring oscillator shown in Figure 1 to illustrate creating a circuit, a testbench, and running a simulation. This quick start guide is intended to distil the features available to demonstrate how to build a circuit in MSPICE (the three stage ring oscillator), create a testbench and run a simulation. Section 4 provides a comprehensive list of features and tools available for the user.

In this guide I will show you how to define the three stage ring oscillator from a template file by first defining the inverter `INV4.,` component and then the three stage ring oscillator `THREE_RING_OSC.m`. Section 2.2 describes the details on setting up a `testbench` with the three stage ring oscillator and how to

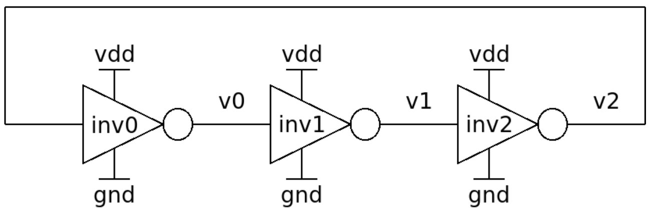

Figure 1: Three Stage Ring Oscillator

obtain the results shown in Figure 2.
The inverter component definition is found in:

`./ccm/libs/coho/components/INV4.m`

The three stage ring oscillator is found in:

`./ccm/libs/coho/circuits/THREE_RING_OSC.m`

And the script to run the example is found in:

`./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`

The tutorial starts with defining the circuit definition for `INV4.m` and demonstrates how once we have a component definition this can be used to compose larger circuits like the three stage ring oscillator shown in Figure 1. You can open these files and follow along in the tutorial to understand how the individual piece work instead of re-creating them from the `TEMPLATE.m` file.

Opening and running the script found in:
`./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`
will produce the results found in Figure 2. If you do not have Rump's IntLAB Automatic Differentiation (AD) package [9], then comment out line 26 of the file:

```
[tMvsFull,VmvsFull]     = tb_mvsFull.simulate([0 2e-10],[0,0,0,1,0]);
```

and comment out lines 55 to 64:

```
subplot(4,1,3)
hold on
plot(timeSim,Vo3,'o')
plot(tMvsFull-1.8e-11-1.1e-12,VmvsFull(:,5))
plot(tMvsFull,0.5*ones(1,length(tMvsFull)),'r--')
ylabel('Output Voltage [V]')
legend('ngSpice','MVS Model - Full')
set(gca,'fontsize',18)
axis([0,timeSim(end),-inf,inf])
set(findall(gca,'type','line'),'linewidth',1.2)
```

There will be a missing subplot after the script is completes running, however you will get an idea of the differences between the models considered.

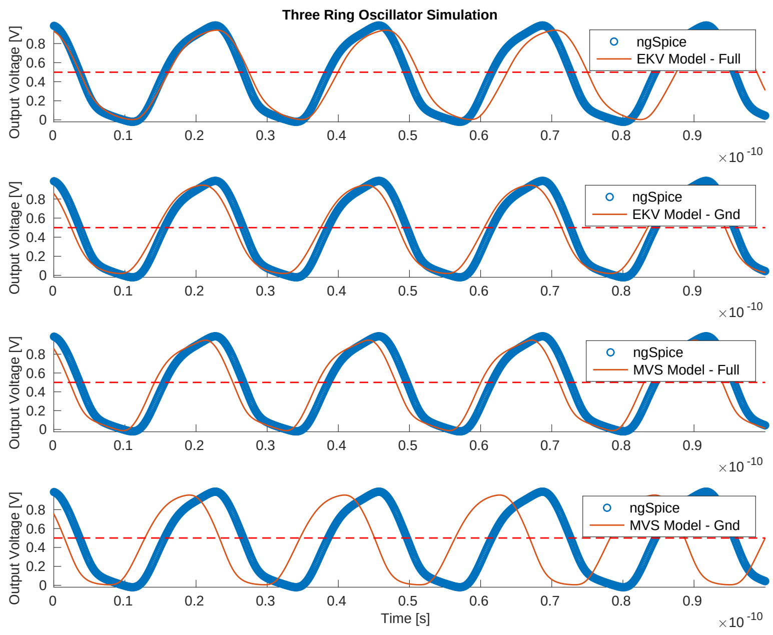

Figure 2: Three Stage Ring Oscillator model comparisons

2.1 Defining the Circuit

To define the three stage ring oscillator, open the circuit template file found in:

`./ccm/libs/coho/circuits/TEMPLATE.m`

This template file has all the necessary scaffolding to define your own circuit and should look like:

```
% Template to be used for your own circuit
classdef TEMPLATE < circuit

    properties (GetAccess = 'public', SetAccess = '
        private')
        %Input; %outputs
    end

    methods
        function this = TEMPLATE(name,wid,rlen)
            if(nargin <2||isempty(name) || isempty(wid))
                error('must provide name and width');
            end
            if(nargin <3||isempty(rlen)), rlen = 1; end
            if(numel(wid)==1)
            else
                assert(length(wid) == 0) %replace 0 with
                    the correct number
            end
            this = this@circuit(name);

            this.finalize;

        end

    end
end
```

Notice that the three stage ring oscillator is made of 3 inverters, so let's use the above template twice. Once to define a simple inverter circuit and a second time to define the three stage ring oscillator. At the end of this portion of the guide you should have circuit definitions that look like the already defined circuits `INV4.m` and `THREE_RING_OSC.m` found in `./ccm/libs/coho/components/INV4.m` and `./ccm/libs/coho/circuits/THREE_RING_OSC.m` respectively.

2.1.1 Inverter `INV4.m`

An inverter is made of two transistors as shown in Figure 3. The following list of instructions show how to fill in the template file for the definition of an inverter circuit explaining the methods along the way. At the end of following these instructions you will have written the definition for the inverter circuit found in `./ccm/libs/coho/circuits/INV4.m`.

You can skip reading these instruction and look at the final version of the code listing found at the end of this section. The instruction provides detailed explanations on what each line of code is doing with regards to building the circuit definition and will be assumed to be known when we continue to define the three stage ring oscillator in Section 2.1.2.
The following steps are instruction on defining `INV4.m`:

1. Add some documentation to explain what this circuit is/does and details

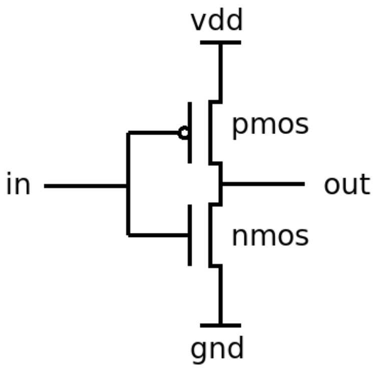

Figure 3: Inverter circuit

on the constructor as shown:

```
% This class defines the 'inverter' circuit for COHO
    library
% An 'inverter' has two nodes: i(input) and o(output)
% To create an inveter, need to provide
%   1. name:  circuit name
%   2. wid:   circuit width
%             wid(1) is the nmos width (nmos)
%             wid(2) is the pmos width (pmos)
%   3. rlen: (circuit length)/(minimum length), use 1
     by default
% E.g. inv = INV('inv',[1e-5,2e-5],1)
% NOTE: It's different from 'inverter' that 'INV'
    consists of 'nmos' and 'pmos'.
%  i.e. a function call of 'INV' requires two
    function calls from 'nmos','pmos'.
%  while a function call of 'inverter' issues one
    call from the mat file directly.
```

2. Then replace the keyword `TEMPLATE` in lines 2 and 9 (of the template file Listing 2.1) with the name of the circuit: in this case `INV4`. (There already exists an INV element for 3 terminal transistors - legacy of coho v1)

```
classdef INV4 < circuit
```

```
    function this = INV4 (name,wid,rlen)
```

3. Save the file as `INV4.m`, this is required so that MATLAB can properly call the constructor of `INV4`.

4. Now we add some properties to the circuit, for the inverter we have as inputs `vdd`, `gnd`, `i`, and we have the output `o`. We add these under the properties of line 5 of the template, as:

```
  properties (GetAccess='public', SetAccess='private'
      );
```

```
        vdd,gnd,i; o;
```

5. To allow for different invocations of inverters we add some guards to throw errors and provide default values where appropriate:

```
        if(nargin<2||isempty(name)||isempty(wid))
          error('must provide name and width');
        end
```

   The above snippet will throw an error if we try to create an inverter without a name and width.

6. The convention used in MSPICE is that if a single value for the width is provided in the constructor, it applies that width to all transistors in the circuit definition. Otherwise we require the length of *wid* to be of the number of devices in the circuit definition to do this we add a guard as follows:

```
        if(nargin<3||isempty(rlen)), rlen = 1; end
        if(numel(wid)==1)
            wid = [1;1]*wid;
        else
            assert(length(wid)==2);
        end
```

   Note that it is the user's responsibility to define the widths in *wid* in the correct order for the circuit definition. Also notice that if *rlen* is not defined, by default it is set to 1. The parameter *rlen* allows the user to modify the length of the device. In all the examples provided here it is set to 1.

7. Finally we now invoke the constructor of the `circuit` class and assign it the variable *this* as shown:

```
        this     = this@circuit(name);
```

By following the above instruction our `TEMPLATE` file should look like:

```
% This class defines the 'inverter' circuit for COHO
   library
% An 'inverter' has two nodes: i(input) and o(output)
% To create an inveter, need to provide
%   1. name: circuit name
%   2. wid:  circuit width
%            wid(1) is the nmos width (nmos)
%            wid(2) is the pmos width (pmos)
%   3. rlen: (circuit length)/(minimum length), use 1 by
   default
% E.g. inv = INV('inv',[1e-5,2e-5],1)
% NOTE: It's different from 'inverter' that 'INV'
   consists of 'nmos' and 'pmos'.
%  i.e. a function call of 'INV' requires two function
   calls from 'nmos','pmos'.
```

```
% while a function call of 'inverter' issues one call
   from the mat file directly.
classdef INV4 < circuit
  properties (GetAccess='public', SetAccess='private');
    vdd,gnd,i; o;
  end
  methods
    % wid: wid(1) is the width of nmos, and wid(2) is the
        width of pmos
    %     if wid(2) is not provided, 2*wid(1) is used by
        default
    % rlen: relative transitor length, must be scaler
    function this = INV4 (name,wid,rlen)
      if(nargin<2||isempty(name)||isempty(wid))
        error('must provide name and width');
      end
      if(nargin<3||isempty(rlen)), rlen = 1; end
      if(numel(wid)==1)
          wid = [1;1]*wid;
      else
          assert(length(wid)==2);
      end

      this    = this@circuit(name);
```

We can now define circuit nodes, and make connections to define our circuit.

1. First we need to add nodes and ports to the circuit by creating nodes via `node` and adding ports via `this.add_port(`*`node_object`*`)`. These can be nested. In the inverter example we do the following to add our inputs and outputs to the circuit:

   ```
           this.vdd = this.add_port(node('vdd'));
           this.gnd = this.add_port(node('gnd'));
           this.i   = this.add_port(node('i'));
           this.o   = this.add_port(node('o'));
   ```

   The above creates nodes and ports *vdd*, *gnd*, *i*, *o* for the circuit. A circuit port requires a `node` object.

2. We then create the devices which make up the inverter and distribute the widths to their appropriate location. An inverter is made of one `nmos` device and one `pmos` device. To add these devices to our circuit we do the following:

   ```
           widN = wid(1);
           widP = wid(2);

           n = nmos('n','wid',widN,'rlen',rlen); this.
               add_element(n);
           p = pmos('p','wid',widP,'rlen',rlen); this.
               add_element(p);
   ```

Each transistor device has nodes d (drain), g (gate), s (source), b (body) which need to be connected. Note also that after a device is created e.g. `n = nmos('n','wid',widN,'rlen',rlen)` it needs to be added to the circuit by calling `this.add_element(device)`. Transistor devices are created using the MATLAB convention of `<'string',value>` pairs. As shown in the above snippet in lines 4 and 5. The constructor of a `(p/n)mos` device has the following signature:

`(p/n)mos(name,<wid,wid_value>,<rlen,rlen_value>)`

3. Once all the devices have been created, we proceed to connecting the device nodes to the nodes within our circuit. Connections are made by calling `this.connect(node_1,node_2,⋯,node_n)`, which connect all nodes$_{1,2,\cdots,n}$ together. The connections in the inverter circuit are defined as follows:

```
        this.connect(this.i,n.g,p.g);
        this.connect(this.o,n.d,p.d);
        this.connect(this.vdd,p.s,p.b);
        this.connect(this.gnd,n.s,n.b);
```

   The input node *this.i* is connected to `pmos` and `nmos` nodes g and so on.

4. To finalize the circuit to be used in a testbench, the circuit class method `this.finalize` needs to be called as shown in line 20 of the `TEMPLATE.m` file.

```
        this.finalize;
```

Having followed the above steps, the circuit definition for `INV4.m` should look as follows:

```
% This class defines the 'inverter' circuit for COHO library
% An 'inverter' has two nodes: i(input) and o(output)
% To create an inveter, need to provide
%   1. name: circuit name
%   2. wid:  circuit width
%             wid(1) is the nmos width (nmos)
%             wid(2) is the pmos width (pmos)
%   3. rlen: (circuit length)/(minimum length), use 1 by default
% E.g. inv = INV('inv',[1e-5,2e-5],1)
% NOTE: It's different from 'inverter' that 'INV' consists of 'nmos' and 'pmos'.
%  i.e. a function call of 'INV' requires two function calls from 'nmos','pmos'.
%  while a function call of 'inverter' issues one call from the mat file directly.
classdef INV4 < circuit
  properties (GetAccess='public', SetAccess='private');
    vdd,gnd,i; o;
  end
  methods
    % wid: wid(1) is the width of nmos, and wid(2) is the width of pmos
    %      if wid(2) is not provided, 2*wid(1) is used by default
    % rlen: relative transitor length, must be scaler
    function this = INV4 (name,wid,rlen)
      if(nargin<2||isempty(name)||isempty(wid))
        error('must provide name and width');
      end
      if(nargin<3||isempty(rlen)), rlen = 1; end
      if(numel(wid)==1)
          wid = [1;1]*wid;
      else
          assert(length(wid)==2);
      end

      this     = this@circuit(name);

      this.vdd = this.add_port(node('vdd'));
      this.gnd = this.add_port(node('gnd'));
      this.i   = this.add_port(node('i'));
      this.o   = this.add_port(node('o'));

      %create devices wid is of the form wid = [widN,widP] if wid is
      %called as a single number then both the P and N device will
      %have the same width.

      widN = wid(1);
      widP = wid(2);

      n = nmos('n','wid',widN,'rlen',rlen); this.add_element(n);
      p = pmos('p','wid',widP,'rlen',rlen); this.add_element(p);

      this.connect(this.i,n.g,p.g);
      this.connect(this.o,n.d,p.d);
      this.connect(this.vdd,p.s,p.b);
      this.connect(this.gnd,n.s,n.b);
      this.finalize;
    end
  end
end
```

2.1.2 Three Ring Stage Oscillator: `THREE_RING_OSC.m`

With the `INV4.m` circuit definition complete we can now use it to compose other circuits. We finish the circuit definition part of the quick start guide by filling in a new `TEMPLATE.m` file so that we end up with the circuit definition found in `./ccm/libs/coho/circuits/THREE_RING_OSC.m`. The following steps gets us to the final circuit definition which will then be used to create a testbench and simulate the circuit using different models and settings in Section 2.2. Building on the knowledge from defining the `INV4.m` circuit definition, we define `THREE_RING_OSC.m` by:

1. Replace lines 2 and 9 of `TEMPLATE.m` with `THREE_RING_OSC`.
2. Add properties *vdd* and *gnd* to line 5.
3. At line 14 add:

```
                    widInv0=[1;1]*wid;
                    widInv1=[1;1]*wid;
                    widInv2=[1;1]*wid;
```

4. Change line 16 with:

```
                    assert(length(wid) == 6)
                    widInv0 = wid(1:2);
                    widInv1 = wid(3:4);
                    widInv2  = wid(5:end);
```

5. Add after line 18 of the `TEMPLATE.m`: `this = this@circuit(name);` what to do with *rlen*:

```
                rlen0 = rlen;
                rlen1 = rlen;
                rlen2 = rlen;
                if(iscell(rlen))
                    rlen0 = rlen{1};
                    rlen1 = rlen{2};
                    rlen2 = rlen{3};
                end
```

6. Add ports *vdd*, and *gnd* as:

```
                this.vdd = this.add_port(node('vdd'));
                this.gnd = this.add_port(node('gnd'));
```

7. Add internal nodes *v0*, *v1*, *v2* and `INV4.m` devices *inv0*, *inv1* and *inv2* as:

```
                v0 = this.add_port(node('v0'));
                v1 = this.add_port(node('v1'));
                v2 = this.add_port(node('v2'));

                inv0 = INV4('inv0',widInv0,rlen0); this.
                    add_element(inv0);
```

```
            inv1 = INV4('inv1',widInv1,rlen1); this.add_element(inv1);
            inv2 = INV4('inv2',widInv2,rlen2); this.add_element(inv2);
```

8. Next we connect the output of *inv2* to node *v2* and to the input of *inv1*:

```
            this.connect(v2,inv2.o,inv0.i);
```

connect the output of *inv0* to node *v0* and the input to *inv1*:

```
            this.connect(v0,inv0.o,inv1.i);
```

connect the output of *inv1* to node *v1* and the input to *inv2*:

```
            this.connect(v1,inv1.o,inv2.i);
```

and lastly we need to connect *vdd* and *gnd* to power the inverters:

```
            this.connect(this.vdd,inv0.vdd,inv1.vdd,inv2.vdd);
            this.connect(this.gnd,inv0.gnd,inv1.gnd,inv2.gnd);
```

9. The last step is to finalize the circuit which is in line 20 of `TEMPLATE.m`.

With the appropriate documentation your version of the three stage ring oscillator circuit should look like:

```
% Definition of a THREE_RING_OSC a three stage ring oscillator with
% INV4 elements
%
% The three ring oscillator has 2 input nodes:
% 1. vdd:    power supply source
% 2. gnd:    circuit ground
% The three ring oscillator has 3 output nodes:
% 1. v1:     the first output of the oscillator
% 2. v2:     the second output of the oscillator
% 3. v3:     the third output of the oscillator
%
% To create a THREE_RING_OSC, requires
% 1. name: oscillator name
% 2. wid: circuit width
%       - if one wid is given that width is applied to all devices
%       - otherwise width needs to be of length 6:
%            wid(1:2) = is for inv0 (INV4)
%            wid(3:4) = is for inv1 (INV4)
%            wid(5:6) = is for inv2 (INV4)
% 3. rlen: relative device length, use 1 by default.
% E.g. myOsc = THREE_RING_OSC('osc0',450e-7,1)
classdef THREE_RING_OSC < circuit
```

```

    properties (GetAccess = 'public', SetAccess = 'private')
        vdd,gnd; %Input
    end

    methods
        function this = THREE_RING_OSC(name,wid,rlen)
            if(nargin <2||isempty(name) || isempty(wid))
                error('must provide name and width');
            end
            if(nargin <3||isempty(rlen)), rlen = 1; end
            if(numel(wid)==1)
                widInv0=[1;1]*wid;
                widInv1=[1;1]*wid;
                widInv2=[1;1]*wid;
            else
                assert(length(wid) == 6)
                widInv0 = wid(1:2);
                widInv1 = wid(3:4);
                widInv2  = wid(5:end);
            end
            this = this@circuit(name);
            %set the relative lengths by default to be be the same
            rlen0 = rlen;
            rlen1 = rlen;
            rlen2 = rlen;
            if(iscell(rlen))
                rlen0 = rlen{1};
                rlen1 = rlen{2};
                rlen2 = rlen{3};
            end
            %define circuit nodes
            %inputs
            this.vdd = this.add_port(node('vdd'));
            this.gnd = this.add_port(node('gnd'));

            %internal node/outputs
            v0 = this.add_port(node('v0'));
            v1 = this.add_port(node('v1'));
            v2 = this.add_port(node('v2'));

            inv0 = INV4('inv0',widInv0,rlen0); this.add_element(inv0);
            inv1 = INV4('inv1',widInv1,rlen1); this.add_element(inv1);
            inv2 = INV4('inv2',widInv2,rlen2); this.add_element(inv2);

            this.connect(v2,inv2.o,inv0.i);
            this.connect(v0,inv0.o,inv1.i);
            this.connect(v1,inv1.o,inv2.i);
            this.connect(this.vdd,inv0.vdd,inv1.vdd,inv2.
                vdd);
            this.connect(this.gnd,inv0.gnd,inv1.gnd,inv2.
                gnd);


            this.finalize;

        end

    end
end
```

2.2 Simulating the Three Stage Ring Oscillator

With the completed circuit definition, we are now ready to use it in a `testbench`. The `testbench.m` class is the backbone of the MSPICE simulator. The signature for creating a `testbench` is as follows:

`testbench(`*`circuit`*`,`*`ports`*`,`*`sources`*`,`*`model`*`,`*`process`*`,`*`options`*`)`
Creating a `testbench` requires:

1. A *`circuit`* definition such as the one described in Section 2.1.2.
2. The *`circuit`*'s *`ports`* are an ordered cell array, i.e. {`myOsc.vdd,myOsc.gnd`}.
3. The *`sources`* are external sources that connect to the *`circuit`*'s *`ports`* in the order they appear as a cell array, i.e. {`Vdd,Gnd`}.
4. The *`model`* is the default `model` that the `testbench` uses for `simulation`.
5. The *`process`* is the `process` for the `testbench`'s *`model`*
6. The *`options`* is the `option` object that determine parameters for the `testbench` at simulation time.

The *`circuit`* we will use is the `THREE_RING_OSC.m` circuit which we defined in Section 2.1.2. The *`ports`* of `THREE_RING_OSC.m` are *vdd* and *gnd*, but we need to create voltage sources for the `testbench` to connect to the ports *vdd* and *gnd* of the `THREE_RING_OSC.m`. Once the voltage sources are created, we make a `testbench` option object to setup the `testbench` with its circuit and voltage sources. Finally we select our `model` and `process`. Once the `testbench` is created we can use the `simulate` method to run our simulation. A full list of methods for the `testbench` class is found in Section 4.4.

2.2.1 Creating Voltage Sources

There are a number of voltage sources available in MSPICE, a full list is found in Section 4.3.1. Here we consider only those that are constant voltage sources used to power the `THREE_RING_OSC.m` circuit which are created by calling

`vsrc('`*`name`*`',`*`value`*`,`*`ctf`*`)`

where '*name*' is the name of the voltage source, *value* is the DC voltage value the source outputs, and *ctg* (connect-to-ground) is a flag which if true connects the negative terminal of the voltage source to ground. To create the necessary sources for the *vdd* and *gnd* ports of our `THREE_RING_OSC.m` circuit the following lines of code in `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m` define these voltage sources:

```
Vdd = vsrc('vdd',1.0,1);
Gnd = vsrc('gnd',0.0,1);
```

2.2.2 Creating the circuit

To create a circuit object which is passed into the `testbench` we simply instantiate our circuit definition as:

```
ringOsc = THREE_RING_OSC('osc',450e-7,1);
```

which creates our three stage ring oscillator with all transistor widths set to $450nm$.

2.2.3 Creating the testbench options

All the `testbench` settings that are available are outlined in Section 4.4.1. The settings used to setup the testbenches in `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m` are as follows:

```
tbOptionsFull = struct('capModel','full','capScale',1,'
    vdd',1,'temp',298,'numParallelCCTs',1,'debug',false);
tbOptionsGnd  = struct('capModel','gnd','capScale',1,'vdd
    ',1,'temp',298,'numParallelCCTs',1,'debug',false);
```

the important difference between these settings is in the capacitance models employed:

1. `<'capModel','gnd'>`
2. `<'capModel','full'>`

The difference between these two capacitance models for the MOS devices within the test bench is weather inter-node capacitance which models effects such as Miller capacitance is considered. The capacitance model `<'capModel','gnd'>` only considers capacitances with respect to ground, while `<'capModel','full'>` considers internet node capacitances where applicable.

If you do not have Rump's AD package [9] comment out line 26:

```
[tMvsFull,VmvsFull]     = tb_mvsFull.simulate([0  2e
    -10],[0,0,0,1,0]);
```

The `testbench` options are structured in the MATLAB convention of `<'string',value>` pairs stored in a MATLAB `struct` object. Shown in the setup for the `THREE_RING_OSC.m` circuit are the following pairs:

1. `<'capModel','gnd/full'>`, the options are *gnd* or *full*, where *gnd* defines capacitances with respect to ground and *full* defines capacitance with respect to ground and inter-node.

2. `<'capScale',value>`, *value* scales $\frac{dV}{dt}$ by *value* for better scaling of numerical integration.

3. `<'vdd',value>`, *value* is the global value for the circuit's *vdd*, legacy of v1 of coho where needed for auto-connection. In the nested bisection algorithm (see Section 4.5), it is useful to have knowledge of *vdd* of the circuit.

4. `<'temp',value>`, *value* is the temperature the circuit is running at in degrees Kelvin.

5. `<'numParallelCCTs',value>`, *value* is the number of circuits being simulated in parallel. This feature allows the user to simulate multiple copies of the circuit with differing initial conditions - a necessary feature for the nested bisection algorithm described in Section 4.5

6. `<'debug',true/false>`, the *debug* flag turns on or off debug mode of the `testbench` . When *debug* mode is *true*, all numerically integrated data points and intermediate calculations such as *Vin*, *I* and *C* are saved in a *debug* cell array.

2.2.4 Creating the `testbench`

In `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`, each `testbench` is created as shown in lines 18 through 22:

```
tb_ekvFull = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
    },{Vdd,Gnd},'EKV','PTM 45nmHP',tbOptionsFull);
tb_ekvGnd  = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
    },{Vdd,Gnd},'EKV','PTM 45nmHP',tbOptionsGnd);

tb_mvsFull  = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
    },{Vdd,Gnd},'MVS','PTM 45nmHP',tbOptionsFull);
tb_mvsGnd  = testbench(ringOsc,{ringOsc.vdd,ringOsc.gnd
    },{Vdd,Gnd},'MVS','PTM 45nmHP',tbOptionsGnd);
```

The `THREE_RING_OSC.m` ports *vdd* and *gnd* correspond to the order in which the voltage sources *Vdd* and *Gnd* appear. These are passed into the `testbench` constructor as cell arrays. The *model* and *process* parameters are passed in as strings and finally the *options* is a `struct` as defined in Section 2.2.3.

In this example, I compare the results obtained by simulating `THREE_RING_OSC.m` using my simplified *EKV* model described in [8] and the adapted *MVS* model [5] which have been fit to data obtained using ngSPICE [10] and the PTM 45nmHP model card. The results in Figure 2 show the differences in these two models when using a fully connected capacitance model vs capacitance referenced to ground only as compared to what ngSPICE produces.

The `testbench` parameters *model* and *process* signal to the `modelDictionary.m` class (see Section 5) which *model* and *process* to use. Here these are *EKV* and *MVS* models and the *PTM 45nmHP* process.

2.2.5 Simulating `THREE_RING_OSC.m`

Once the `testbench` is created such as `tb_ekvFull` as shown in Section 2.2.4 then to obtain a time series result a call to the `simulate` method invokes the `testbench`'s integrator to simulate the `circuit` defined in the `testbench` over the specified time interval, i.e:

```
myTestbench.simulate([starTime,stopTime],v0,options)
```

The parameters *v0* and *options* are optional in this case and can be used to specify initial conditions and integrator options respectively. In `./ccm/Examples/Three Ring Oscillator/threeRingOsc.m`, the simulations are defined as:

```
[tEkvFull,VekvFull] = tb_ekvFull.simulate([0 2e-10],[0,0,0,1,0]);
[tEkvGnd,VekvGnd]   = tb_ekvGnd.simulate([0 2e-10],[0,0,0,1,0]);
[tMvsFull,VmvsFull]   = tb_mvsFull.simulate([0 2e-10],[0,0,0,1,0]);
[tMvsGnd,VmvsGnd]   = tb_mvsGnd.simulate([0 2e-10],[0,0,0,1,0]);
```

where *startTime* is 0 and *stopTime* is 2×10^{-10} seconds. The initial conditions *v0* specifies the starting condition for the voltage sources and the internal circuit nodes *v0*, *v1* and *v2*. In this case *v0* starts at 0, *v1* starts at 1 and *v2* starts at 0. Notice that no integrator options are specified, thus internal defaults are selected. The results of the simulation are shown in Figure 2 at the beginning of this section.

3 Synchronizer Metastability Simulation and Analysis: Quick Start Guide Passgate Latch Synchronizer

This quick guide demonstrates the use of the `nestedBisection.m` and `nestedBisectionAnalysis.m` `testbench` subclasses on a passgate latch synchronizer shown in ??.

4 MSPICE Features and Functions

This section is intended to be a reference to various files, functions/methods available as part of the MSPICE framework. The sections are sub-divided by class and their subclasses. Included within is a comprehensive list of methods that are available to the user. The private methods of the classes will be denoted as such so that a user looking to extend functionality is aware of their existence and will be clearly marked when being described.

4.1 `circuit.m` class

The `circuit.m` class in MSPICE is the class which defines circuits, creates connectivity matrices, a circuit path object and provides methods to query information about the circuit to extract specific information relevant to the defined circuit. It has as sub-classes: `coho_leaf.m` and `coho_vsrc.m`.

4.1.1 Circuit `TEMPLATE.m`

Found in `./ccm/libs/coho/circuits/TEMPLATE.m` is a blank template file that can be used to define your own circuit definitions. If these are deemed to be components that make up larger circuits they can be saved in `./ccm/libs/coho/components`. Such examples of components are `INV4.m`, `NAND2.m`, ect. A full list of available components as v2.0 of this manual can be found in **??**. The template can be decomposed in the following:

- The template is of class `circuit`
- The properties by default are *get* 'public' and *set* 'private'. These properties are generally inputs and outputs of the circuit.
- The constructor for the circuit: `TEMPATE(`*`name,wid,rlen`*`)` requires a circuit *name*, *wid* (for transistor widths), and *rlen* an optional parameter that can modify the relative length of the transistor sizes (legacy from version 1 of coho - generally not provided or set to 1).
- Following the constructor definition, is code to provide checks and throw an error if the inputs are incorrect, along with the ability to automatically set default values.
- This is then followed by constructing the superclass `circuit` object which gets assigned the variable *this* which is used to access methods of the `circuit` class and any methods unique to the current subclass.
- Finally the body of the circuit definition is filled in and then must be finalized by calling `this.finalize` at the end of the circuit definition.

4.2 `coho_leaf.m` class

The `coho_leaf.m` class is a subclass of `circuit.m` and describes MOSFET components, resistors, capacitors and inductors.

4.2.1 MOS devices

constructor and definitions for mos devices

4.3 `coho_vsrc.m` class

The `coho_vsrc.m` class is a subclass of `circuit.m` and describes voltage sources for MSPICE.

4.3.1 Voltage Sources

constructor and definitions for voltage sources

4.4 `testbench.m` class

4.4.1 Testbench Options

describe the testbench option struct, and options available.

4.5 `nestedBisection.m` class

4.5.1 `nestedBisectionWithKick.m` class

4.6 `nestedBisectionAnalysis.m` class

5 MSPICE Models

describe the models

5.1 Model Dictionary class

describe the model dictionary class and setup

5.2 Model interface

describe the model interface

5.3 Model cards

6 MSPICE Library contents

List of all library contents

6.1 Circuits

6.2 Components

6.3 Leaves

6.4 Sources

References

[1] L. M. engelhardt. *version XVII.* Linear Techology Corporation - now part of Analog Devices, 2019.

[2] HSPICE. *the gold standard for accurate circuit simulations.* Synopsis Inc., Mountain View, California, 2019.

[3] N. Integration and M. N. Group. Predictive technology model, June 2011.

[4] MATLAB. *version 9.3.0.713579 (R2017b).* The MathWorks Inc., Natick, Massachusetts, 2017.

[5] S. Rakheja and D. Antoniadis. MVS nanotransistor model (silicon), Dec 2015.

[6] J. Reiher and M. R. Greenstreet. Optimization and comparison of synchronizers. In *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2020.

[7] J. Reiher, M. R. Greenstreet, and I. W. Jones. Explaining metastability in real synchronizers. In *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 59–67, May 2018.

[8] J. J. Reiher. Synchronizer analysis and design tool: an application to automatic differentiation. Master's thesis, University of British Columbia, 2020.

[9] S. M. Rump. *INTLAB — INTerval LABoratory*, pages 77–104. Springer Netherlands, Dordrecht, 1999.

[10] H. Vogt, M. Hendrix, and P. Nenzi. *ngSPICE - version 30.* 2019.

[11] C. Yan. *Reachability Analysis Based Circuit-Level Formal Verification.* PhD thesis, The University of British Columbia, 2011.

[12] S. Yang and M. R. Greenstreet. Simulating improbable events. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 154–157, New York, NY, USA, 2007. ACM.