E8-262 CAD for High Speed Circuits

Homework I : Spice program

Ashwin Rajesh

M.Tech Microelectronics and VLSI Design

Indian Institute of Science, Bangalore

Problem statement

Consider only R, C, G, V elements. Write a SPICE program in MATLAB/C for time domain simulation. Your code should do the following:

1. Parse net list (support for only the supplied SPICE files is OK: P2test1,2,3.sp)
2. DC operating point analysis
3. AC analysis

Theory

This implementation of spice uses the modified nodal analysis method. It involves solving linear equations that are formed out of nodal analysis equations on the whole circuit. The node voltages are the values being solved for. However, some elements like the voltage source break pure nodal analysis where conventionally "supernodes" have to be considered. In modified nodal analysis, "virtual nodes" of variables like branch current is used to circumvent this.

Each component in the circuit has a "stamp" which consists of elements of the LHS matrix and the RHS vector. The stamps are applied based on the connectivity of the component. The stamps required here are given below (R, C, G, V from top left in clockwise order).

	N+	N-	
N+	1/R	-1/R	
N-	-1/R	1/R	

	N+	N-	
N+			-Ik
N-			+Ik

	N+	N-	i_k	
N+			+1	
N-			-1	
i_k	+1	-1		V_k

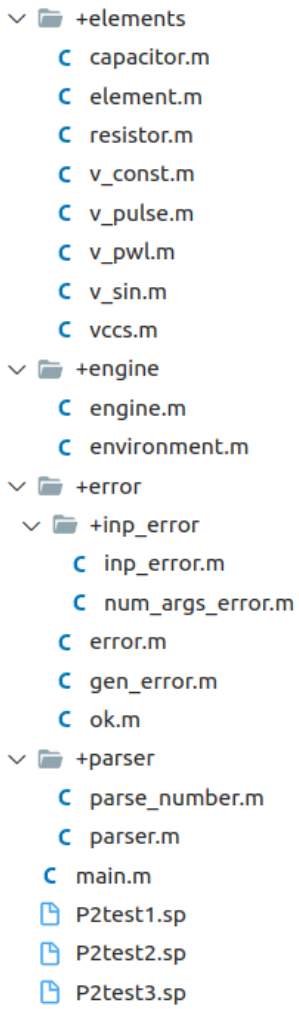
	N+	N-	Nc+	Nc-	
N+			+G_k	-G_k	
N-			-G_k	+G_k	
Nc+					
Nc-					

It can be noticed that the capacitor needs the previous branch voltage, and that the voltage source needs a virtual node. Capacitor also has a virtual node in our implementation to read the current flowing through it.

Implementation

The implementation was done with extensibility in mind. Since only a subset of the components have been covered, the code was written such that more components and features could be easily added. The functionality was divided into different modules, as listed below :

- **error** : These are objects returned by functions that indicate presence or absence of errors and some printable message
    - **inp_error** : Errors in the input spice file. Has line number and file name as properties to print.
- **parser** : The front-end of the program that reads the spice files and creates the other objects. The parse_number function parses numeric values with units and prefixes to float values.
- **elements** : Classes representing circuit components
- **engine** : The actual solver (engine class) and the environment that stores the components and generates the stamp matrix



The **main.m** file contains a function, **main(file_name)** which is the entry point of the program. It receives the path to the file and performs the simulation as per the commands in the file. It creates a **parser** object which reads the spice file line-by-line and creates the **environment** object which contains the circuit components and then

an **engine** object which contains the simulation settings. The main steps in the program is shown in the flowchart below.

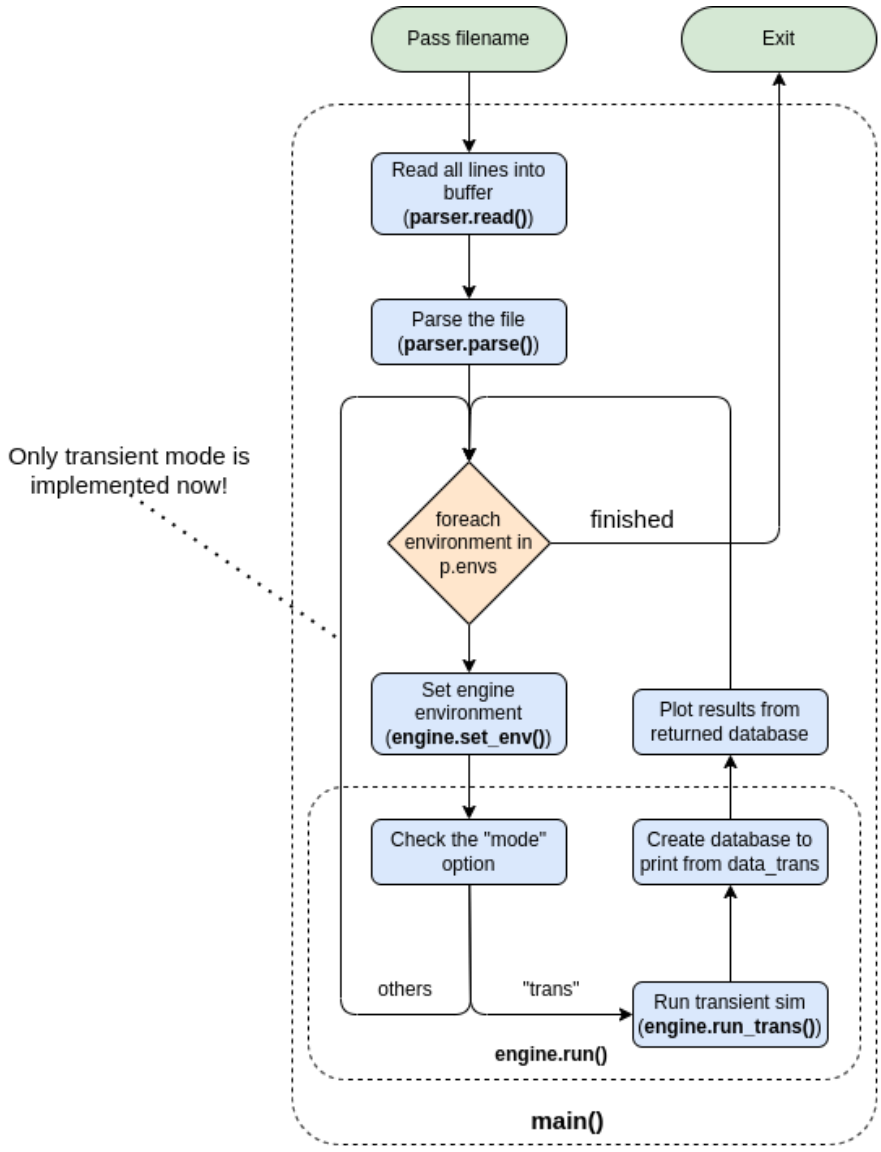


Figure : Flowchart for the spice program

The parser.parse() and engine.run_trans() functions are where most of the work happens. The flowcharts for these are given below. Error reporting is not shown here for simplicity.

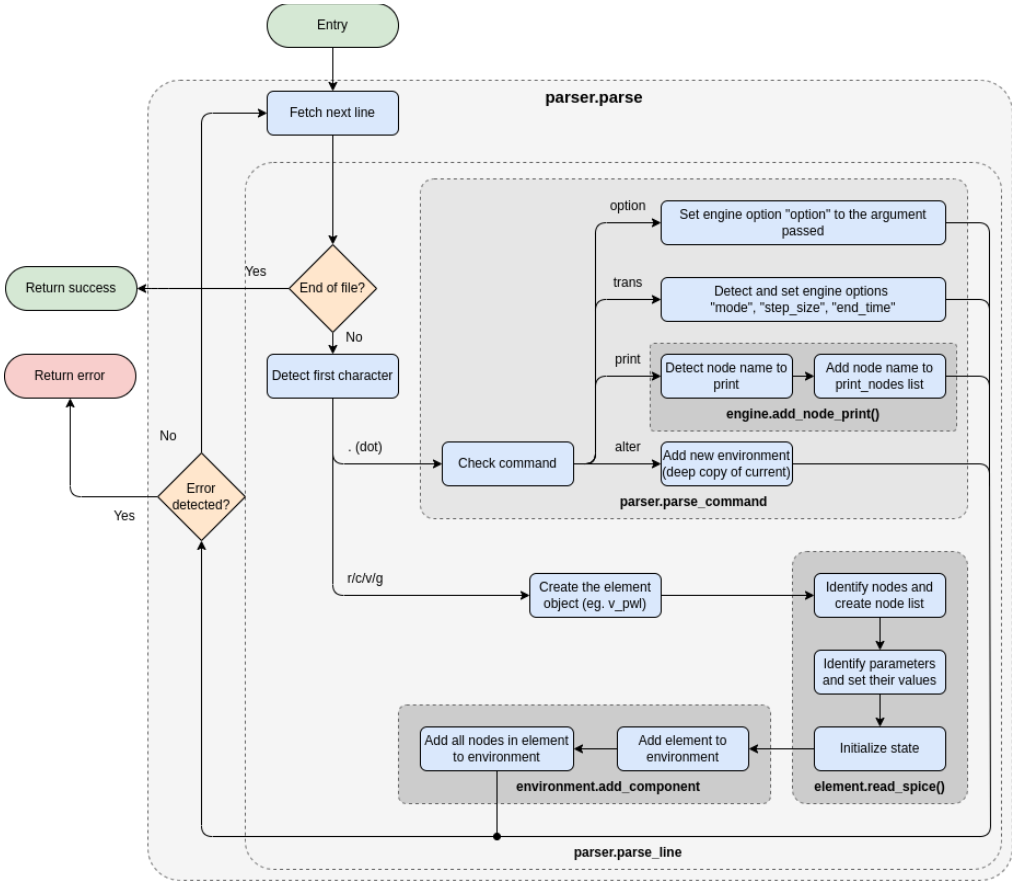


Figure : Flowchart for the parse() function of the parser object

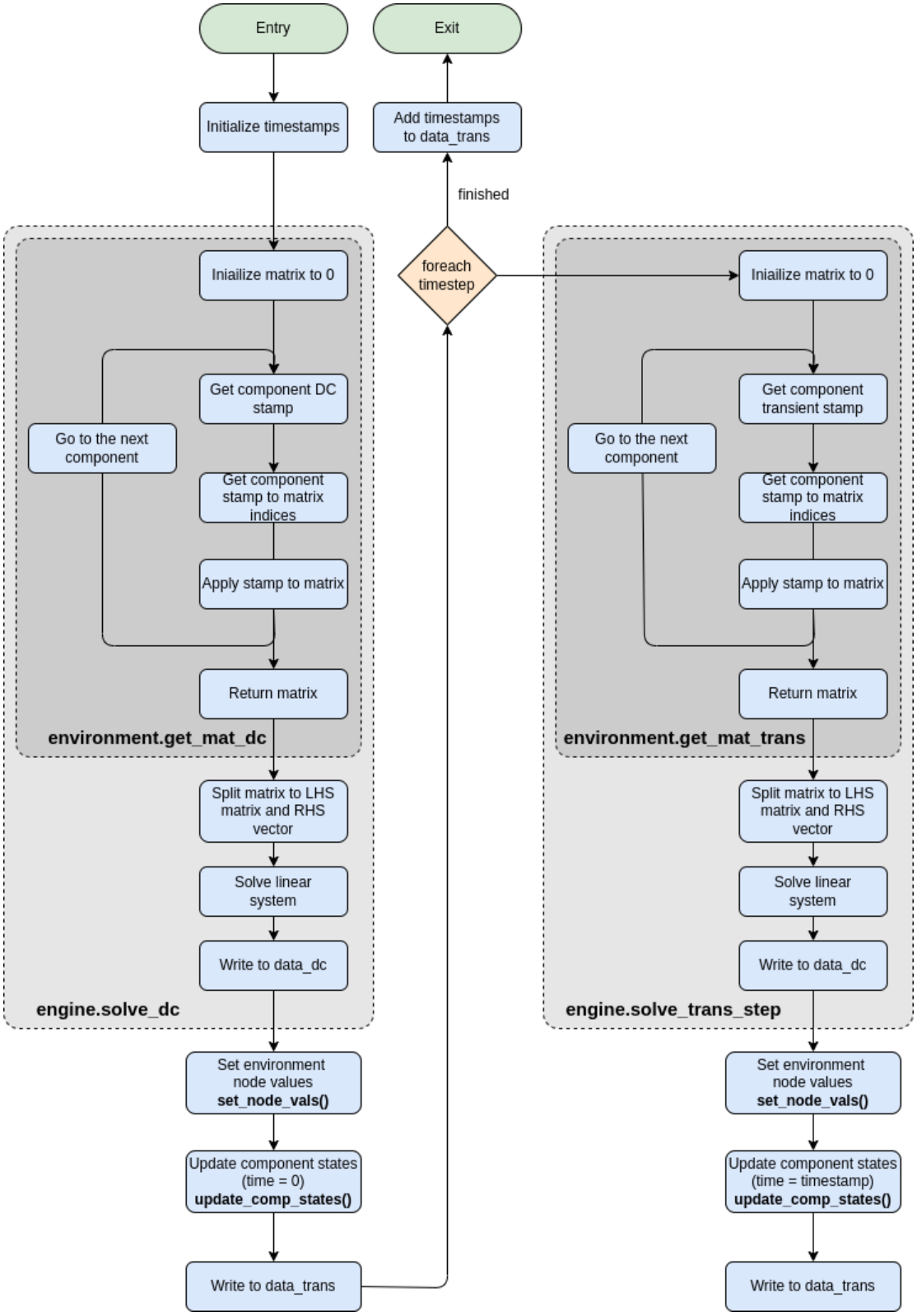
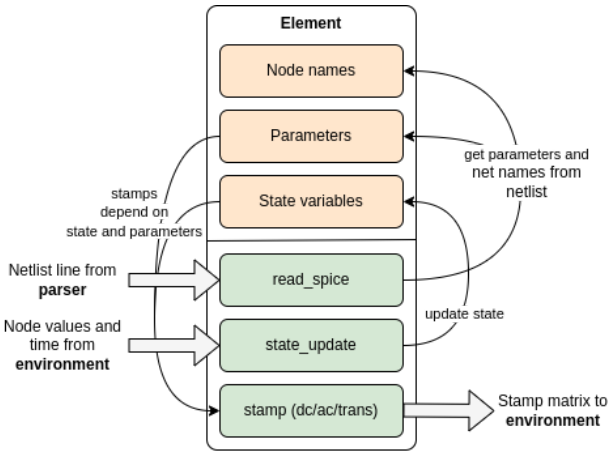


Figure : Flowchart of the run_trans() function of the engine object

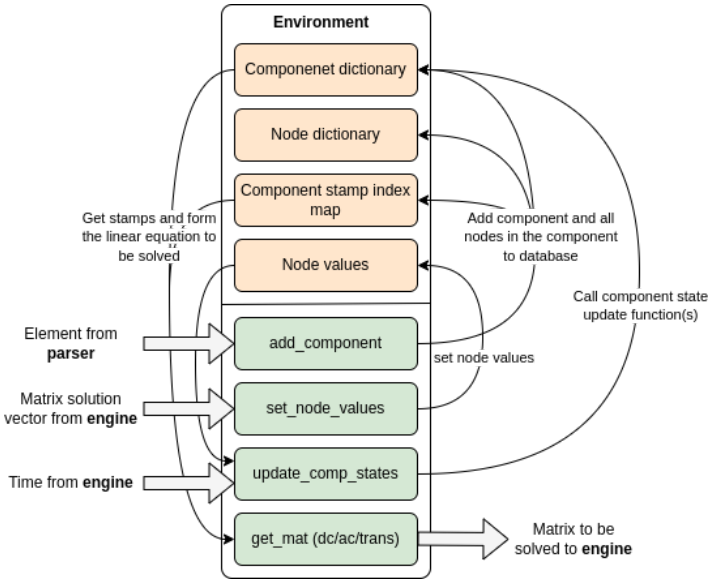
Circuit elemenets

The circuit elements are statefull objects which store state values like charge (for capacitor), time (for transient voltage source) or current (for inductor) and parameters like capacitance, resistance, etc. They have functions that return the stamps for AC, DC and transient simulations for the given state. This enables easy implementation of any element including non-linear elements but with some performance penalty (due to greater memory requirements and function calls). The states are updated in each element from the environment which calls a function in all the elements by passing a hasmap object with the values of all the nodes and virtual nodes (additional values like branch currents) that were solved by the engine. The elements store the names of all the nodes and virtual nodes and retrieve the required values from the hashmap and update their internal state accordingly.



Environment

The environment stores all the components and the nodes they use (including virtual nodes like branch currents) as a dictionary. Each node name is mapped to an index which is its index in the matrix to be solved and each component name is mapped to the component object. The values of the



nodes are also stored as dictionaries. This is useful in the state update phase where the engine passes the solution of the matrix to the environment and the environment saves it in the dictionary according to the indices of the nodes. Dictionaries are implemented using hashmaps. Then the engine calls the update_comp_states function with the current sim time which accesses all the components and passes the node values dictionary and the simulation time to them. After this, the engine calls the get_mat function for AC, DC or transient which returns the matrix to be solved. The environment encapsulates all the information and state associated with the circuit at a point of time for easy manipulation by the solver engine. The stamp index map stores a list for each component which indicates how the stamp matrix returned by the component should be stamped into the final matrix (it caches the indices of the nodes in the solution matrix)

Engine

The engine object contains an environment object and simulation settings. Using this, it performs the simulations and records the required data. It interfaces with the environment object to get the matrix to be solved, solves it and passes it back to update its internal state. It stores the names of the nodes to be printed and stores them in a transient simulation and returns it as a dictionary along with the timestamps to be plotted, after the simulation is completed. The transient simulation starts with a DC solution at t = 0 and then proceeds with the loop of matrix retrieval, matrix solution and state updation. The solution is stored in an array that is later processed to return a dictionary of values.

Parser

The parser is responsible for building the engine and environment objects by reading the spice netlist. It first cleans the input by removing comments and unnecessary whitespaces. Then it detects if the line is a command (starting with .) or an element description. If it is an element

description, it detects which element it is (using the first letter) and passes the line to the concerned element's read_spice function to form the element with its nodes and parameters. It adds this to the present environment variable.

If it is a command, it parses the line and adds the required options (like step size and end time) in the options dictionary of the engine. The parser also keeps track of the current line number and file name for error reporting. This has been implemented currently only for the circuit element parsing and not for the command parsing. If the .alter command is encountered, it adds one more environment to the environments list that is a copy of the current environment. Any changes or additions made further will be on this new environment. This environment has to be linked with the engine externally.

Main program

The main program initializes the parser and passes the spice filename to be read. It handles error reporting from the parser. Then once it creates the environment and engine, the main program calls the run function in the engine and gets the output and displays it. It repeats this for all the environments detected (for each alter command).

Testcases

1) **P2test1.sp**

```
* Time Domain Simulation
R1      1       0       100k
c1      1       2       100u
c2      2       3       35u
r2      2       0       5
va      3       0       10V


.option post
.tran 0.005ms 5ms
```

```
.alter
va 3 0  pulse ( 0 12V 2ms 0.1ms 0.1ms 1ms 2ms )

.alter
va 3 0  pwl ( 2ms,0v 2.5ms,12v 2.75ms,5v 3ms,-5V 3.5ms,12v 3.6ms,5v 4ms,0v
4.5ms,3v)

.alter
va 3 0  sin(0 12V 1kHz 2ms 10 45)

.print v(0)
.print v(1)
.print v(2)
.print v(3)

.end
```

2) **P2test2.sp**

```
* Time Domain Simulation
r2      1  2  5e3
r3      2  3  400
r4      3  4  1e3
c1      4  0  9e-5
vdyn    1  0  10v

.options post
.TRAN  0.1ms 800ms

.alter
vdyn 1 0 PULSE( 1V 3V 0.1ms 0.1ms 0.1ms 40ms 100ms)

.alter
vdyn 1 0 pwl(0ms,3v 100ms,2v 200ms,0v 250ms,2v 300ms,3.5v 400ms,2V 500ms,0v
600ms,0v 700ms,3v 705ms,0v)

.alter
vdyn 1 0 sin(2v 2v 25Hz 2ms 0 90 )

.print v(0)
.print v(1)
.print v(2)
.print v(3)
.print v(4)

*.end
```

3) **P2test3.sp**

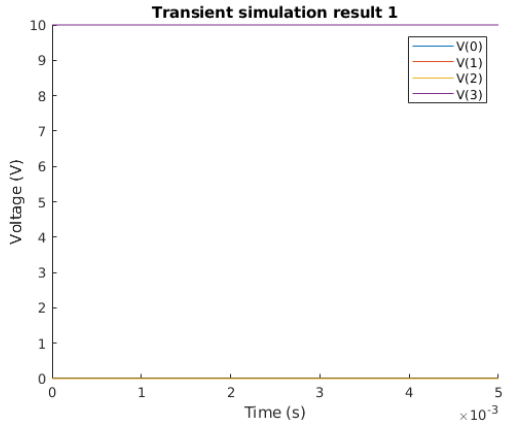
```
* Pseudo Butterworth Filter

R1      1 2 1K
R2      2 3 1K
C1      2 4 126n
C2      3 0 113n
g1      4 0 3 4 100K
rbyp    4 0 900k
VIN     1 0 10

.option post
.TRAN 100NS 2MS
.alter
VIN     1 0  PULSE(0 1V 0ms 0.1ms 0.1ms 1ms 5ms)

.alter
VIN     1 0  PWL(0ms,0V 0.5ms,1.1V 1ms,1.1V 1.3ms,0.2v )

.alter
VIN     1 0  SIN(0 1V 1khz 0ms 0 0 )

.print v(0)
.print v(1)
.print v(2)
.print v(3)
.print v(4)

.END
```

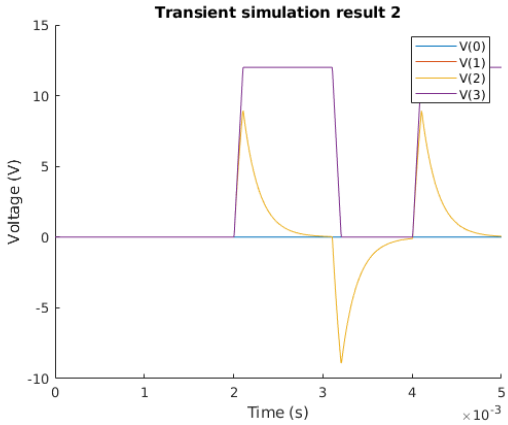
Results

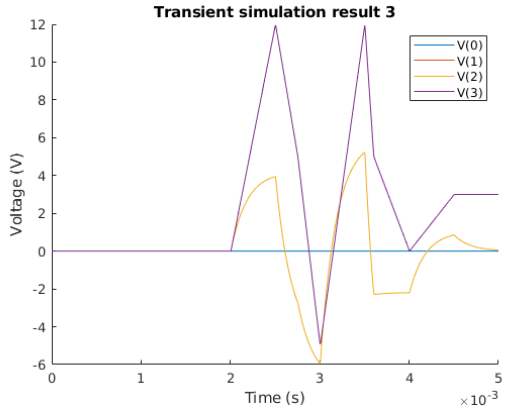
1) **P2test1.sp**

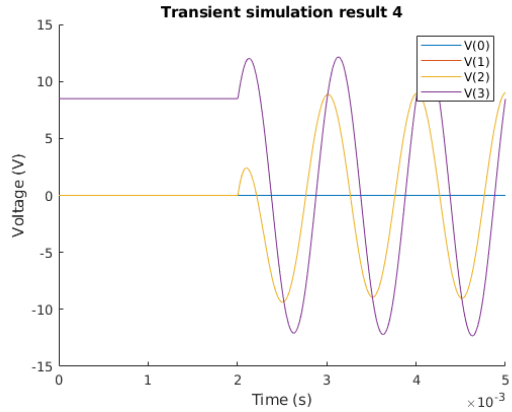
```
Reading file...
Parsing...
Solving alternative 1
DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
```

V(3) : 10
Solving alternative 2
DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
V(3) : 0
Solving alternative 3
DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
V(3) : 0
Solving alternative 4
DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
V(3) : 8.4853



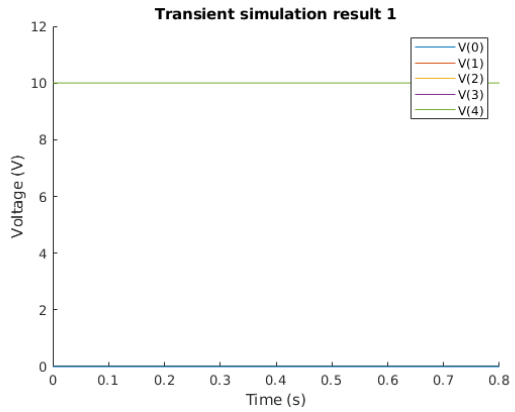


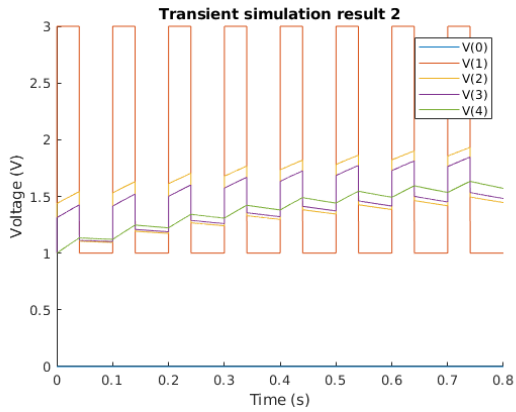


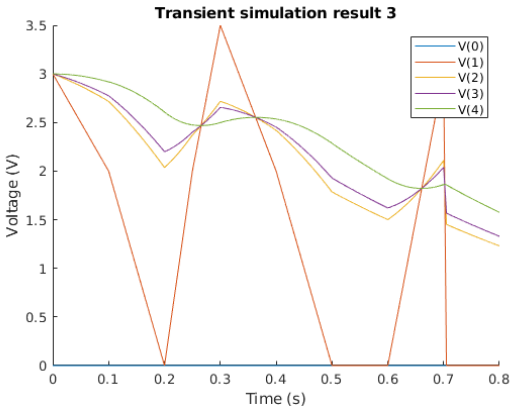


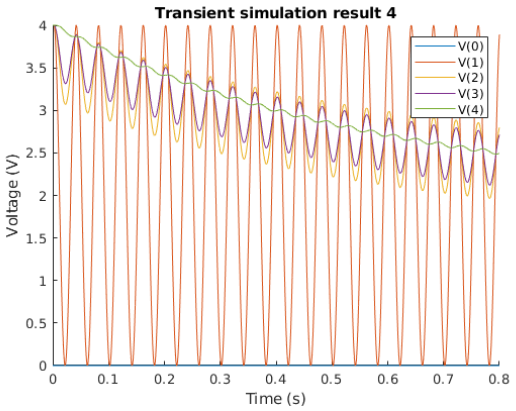
2) P2test2.sp

```
Reading file...
Parsing...
Solving alternative 1
DC operating point
V(0) : 0
V(1) : 10
V(2) : 10
V(3) : 10
V(4) : 10
Solving alternative 2
DC operating point
V(0) : 0
V(1) : 1
V(2) : 1
V(3) : 1
V(4) : 1
Solving alternative 3
DC operating point
V(0) : 0
V(1) : 3
V(2) : 3
V(3) : 3
V(4) : 3
Solving alternative 4
DC operating point
V(0) : 0
V(1) : 4
V(2) : 4
V(3) : 4
V(4) : 4
```





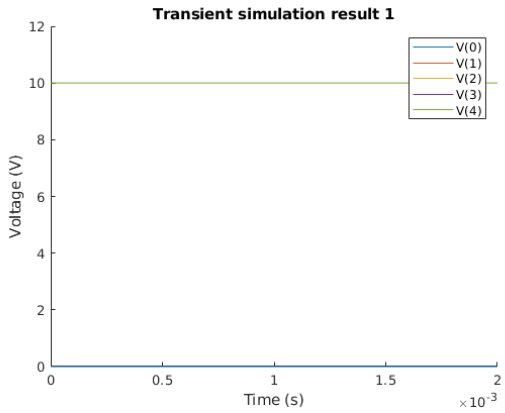


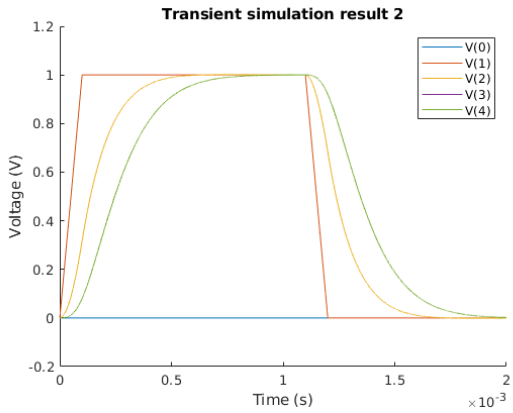


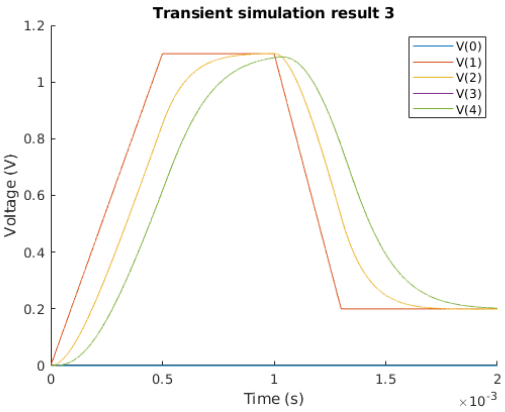
3) P2test3.sp

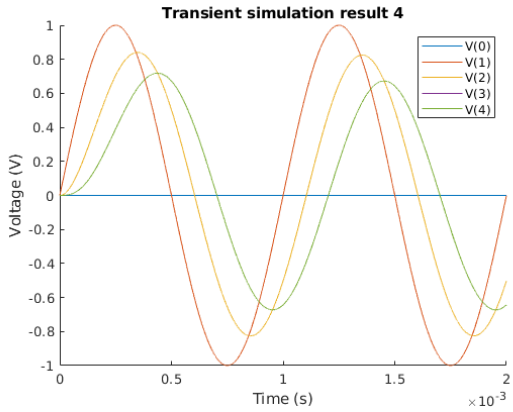
Reading file...
Parsing...
Solving alternative 1
DC operating point
V(0) : 0
V(1) : 10
V(2) : 10
V(3) : 10
V(4) : 10
Solving alternative 2
DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
V(3) : 0
V(4) : 0
Solving alternative 3

DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
V(3) : 0
V(4) : 0
Solving alternative 4
DC operating point
V(0) : 0
V(1) : 0
V(2) : 0
V(3) : 0
V(4) : 0









References

1) *Dr. Dipanjan Gope*, "**E8 262: CAD for High Speed Chip-Package-Systems**" Lecture notes (Lec 4, 5),