CMVSIM User's Guide

A. Lumsdaine, M. Silveira, J. White

RLE Technical Report No. 590

December 1994

The Research Laboratory of Electronics
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139-4307

This work was supported in part by the National Science Foundation under Grant MIP 91-17724.

CMVSIM USER'S GUIDE

A. Lumsdaine, M. Silveira, J. White

Research Laboratory of Electronics
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Draft of December 12, 1994

Abstract

This manual describes how to use the CMVSIM program for simulating grid-based analog circuit arrays on the Connection Machine®. The program was originally designed with robotic vision chips in mind, so the simulator has built-in functionality for handling images as input and as output.

CMVSIM has two intended uses. The first use is in simulating algorithms, whereby CMVSIM is used to simulate grids of idealized circuit elements so that a designer can tune the idealized network realization of a particular algorithm. The second, and more important, use is in simulating actual VLSI circuits, whereby CMVSIM is used to simulate a VLSI circuit at an analog level. It is in this arena where CMVSIM's capabilities are so important, because VLSI realizations of vision circuits (for example) can easily have hundreds of thousands of devices, and yet the circuit must be simulated in its entirety at an analog level. Such a capability is well beyond that of a conventional circuit simulation program running on a serial computer. Since CMVSIM can simulate very large circuits in a reasonable amount of time, it allows the incorporation of a simulation phase into the design cycle of grid-based analog signal processors.

CMVSIM is a superset of the SIMLAB program. This manual primarily describes the functionality particular to CMVSIM and assumes the reader is familiar with SIMLAB and has access to the *SIMLAB User's Guide*.

For information on extending CMVSIM, see the *CMVSIM Programmer's Guide* and the *SIMLAB Programmer's Guide*.

Contents

1 Introduction

CMVSIM is a program for simulating large grid-based analog circuit arrays on the Connection Machine®[1][2]. The program was originally developed for the MIT Vision Chip Project [16], so vision chips as an application have had an obvious influence on the design of the simulator. However, the simulator is by no means restricted only to vision chips.

As part of the MIT Vision Chip Project, CMVSIM has had two uses. The first use is in simulating algorithms, whereby CMVSIM is used to simulate grids of idealized circuit elements so that a designer can tune the idealized network realization of a particular algorithm. The second, and more important, use is in simulating actual VLSI circuits, whereby CMVSIM is used to simulate a VLSI circuit at an analog level. It is in this arena where CMVSIM's capabilities are so important, because VLSI realizations of vision circuits (for example) can easily have hundreds of thousands of devices, and yet the circuit must be simulated in its entirety at an analog level. Such a capability is well beyond that of a conventional circuit simulation program running on a serial computer. Since CMVSIM can simulate very large circuits in a reasonable amount of time, it allows the incorporation of a simulation phase into the design cycle of grid-based analog signal processors.

CMVSIM is a superset of the SIMLAB program. This manual primarily describes the functionality particular to CMVSIM and assumes the reader is familiar with SIMLAB and has access to the *SIMLAB User's Guide* [10].

The CMVSIM program was developed as a research project. It is not intended to be a production system, although serious effort has been made to insure the correctness of the program (it works for at least one test case). Constructive suggestions and bug reports regarding CMVSIM are welcomed. Comments and suggestions can be sent by electronic mail to cmvsim@sobolev.mit.edu; bug reports should be sent to bug-cmvsim@sobolev.mit.edu.

In Section 2 we provide a brief overview of the CMVSIM program. Section 2.1 describes a model problem, Section 2.2 describes the numerical algorithms used by CMVSIM, and Section 2.3 describes the data to processor mapping. The advanced user may skip these parts of the guide and direct him/herself to the sections containing specific directions for his/her problem. It is our belief, however, that this guided tour of the simulator's architecture and capabilities will help the new user to obtain a better understanding of what CMVSIM can do.

The CMVSIM program was developed at MIT (under support from the NSF PYI program, and DARPA contract N00014-87-K-825) and at Thinking Machines Corporation.

[1]Connection Machine is a registered trademark of Thinking Machines Corporation.

2 Overview

The recent success using one and two dimensional resistive grids to perform certain filtering tasks required for early vision [7] has sparked interest in general analog signal processors based on arrays of analog circuits coupled by resistive grids. As is usually the case, before fabricating these analog signal processors, substantial circuit-level simulation must be performed to insure correct functionality. Although desirable, simulation of *complete* signal processors has not been attempted because of the computational cost. Ambitious circuits consist of arrays of cells where the array size can be as large as 256×256, and each cell may contain up to a few dozen devices [16]. Therefore, simulation of a complete signal processor requires solving a system of differential equations with *hundreds of thousands* of unknowns.

The structure of grid-based analog signal processors is such that they can be simulated quickly and accurately with specialized algorithms tuned to certain parallel computer architectures. In particular, the coupling between cells in the analog array is such that a block-iterative scheme can be used to solve the equations generated by an implicit time-discretization scheme, and furthermore, the regular structure of the problem implies that the simulation computations can be accelerated by a massively parallel SIMD computer, such as the Connection Machine ®[2][2].

2.1 Model Problem

Consider the circuit in Figure 1, an idealized version of a grid-based analog signal processor used for two-dimensional image smoothing and segmentation [5]. The node equation for a grid point i, j in the network is

$$\begin{aligned} c\dot{v}_{i,j} &= g_f(v_{i,j} - u_{i,j}) \\ &+ g_s(v_{i,j} - v_{i+1,j}) + g_s(v_{i,j} - v_{i-1,j}) \\ &+ g_s(v_{i,j} - v_{i,j+1}) + g_s(v_{i,j} - v_{i,j-1}) \end{aligned} \tag{1}$$

where $u_{i,j}$ represents the image data at the grid point i, j, $v_{i,j}$ is the output voltage at node i, j, g_f is the input source impedance, c is the parasitic capacitance from the grid node to ground, and $g_s(\cdot)$ is a nonlinear "fused" resistor. In this circuit, the g_s resistors pass currents in such a way as to force $v_{i,j}$ to be a spatially smoothed version of $u_{i,j}$, unless the difference between neighboring $u_{i,j}$'s is very large. In that case, g_s no longer conducts, there is no smoothing, and the image is said to be "segmented" at that point.

In a more complete representation of the image smoothing and segmentation circuit, the voltage source $u_{i,j}$ and the source impedance g_f are replaced with a subcircuit which

[2]Connection Machine is a registered trademark of Thinking Machines Corporation.

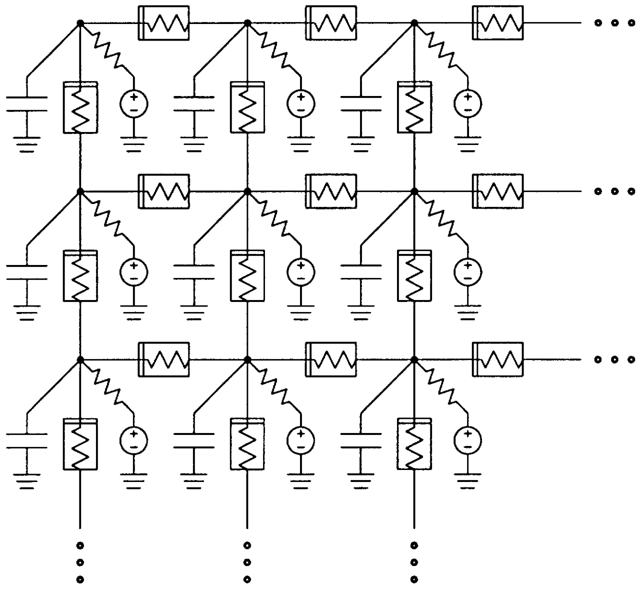

Figure 1: Grid of nonlinear resistors.

typically contains operational amplifiers and a phototransistor. If such a subcircuit has M internal nodes and contains only voltage-controlled elements, then it can be described by a differential equation system of the form

$$\frac{d}{dt} q_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t) = f_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t) \tag{2}$$

where $\tilde{v}_{i,j} \in \mathbb{R}^M$ is the vector of the i, j^{th} subcircuit's internal node voltages, and $q_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t)$, $f_{sub}(\tilde{v}_{i,j}(t), v_{i,j}(t), t) \in \mathbb{R}^m$ are the vectors of sums of charges and sums of resistive currents, respectively, at each of the subcircuit's internal nodes. Incorporating the subcircuit's behavior into the equation for grid point i, j leads to

$$\begin{aligned} c\dot{v}_{i,j} &= i_{sub}(v_{i,j}, \tilde{v}_{i,j}) \\ &+ g_s(v_{i,j} - v_{i+1,j}) + g_s(v_{i,j} - v_{i-1,j}) \\ &+ g_s(v_{i,j} - v_{i,j+1}) + g_s(v_{i,j} - v_{i,j-1}), \end{aligned} \tag{3}$$

where $i_{sub}(v_{i,j}, \tilde{v}_{i,j})$ is the current entering subcircuit i, j from grid node i, j.

For our purposes, an $N \times N$ grid-based analog signal processor, or analog array, is any circuit that can be described by a system of equations generated by replicating (2) and (3) for each $i, j \in 1, \ldots, N$.

2.2 Numerical Algorithms

For notational simplicity, the system of equations that describe an $N \times N$ grid-based analog signal processor, defined in the previous section, will be written compactly as

$$\frac{d}{dt} q(v) = f(v(t)), \tag{4}$$

where $q(v(t)), f(v(t)) \in \mathbb{R}^{N^2 \times (M+1)}$ are the vectors of sums of node charges and node resistive currents.

The transient simulation of the analog grid involves numerically solving (4). To compute the solution, it is possible to use simple explicit or semi-implicit numerical integration algorithms, but for these types of circuits, experiments show that an implicit method like the trapezoidal rule is substantially more efficient [9]. The trapezoidal rule leads to the following algebraic problem at each time step h:

$$q(v(t+h)) - q(v(t)) + \frac{1}{2h}[f(v(t+h)) + f(v(t))] = 0. \tag{5}$$

As is standard, the algebraic problem is solved with Newton's method,

$$J_F(v^m(t+h))[v^{m+1}(t+h) - v^m(t+h)] = -F(v^m(t+h)) \tag{6}$$

where

$$F(v(t+h)) = [q(v(t+h)) - q(v(t))] + \frac{1}{2h}[f(v(t+h)) + f(v(t))] \tag{7}$$

and the Jacobian $J_F(v(t))$ is

$$J_F(v(t+h)) = \frac{\partial q(v(t+h))}{\partial v} + \frac{1}{2h}\frac{\partial f(v(t+h))}{\partial v} \tag{8}$$

In "classical" circuit simulators such as SPICE [8], the linear system of equations for each Newton iteration is solved by some form of sparse Gaussian elimination. When simulating grid-based signal processors, where the coupling between subcircuits is restricted to nonlinear resistors, the Newton iteration equation will be such that its solution can be efficiently computed by iterative algorithms like conjugate-gradient squared (CGS) [11, 1]. To demonstrate this, in Table 1, we compare the CPU time required to compute the transient analysis of the network in Figure 1 using several different matrix solution algorithms to solve the Newton iteration equation. This problem is hard for an iterative method because, though not described here, the transient analysis is performing a continuation on the nonlinear resistor elements that changes the conditioning of the matrix with time (see [5] for details). As the table indicates, sparse Gaussian elimination is much slower than CGS[3] or ILU preconditioned CGS, both of which perform almost identically. This is a fortunate result, because our goal is to develop an efficient parallel simulator, and unpreconditioned CGS is easiest to parallelize.

[3]This problem is symmetric, so CGS and standard conjugate-gradient are equivalent

Size	Direct	CG	ILUCG
16×16	16.53	11.72	10.27
32×32	156.57	60.72	50.75
64×64	1856.12	272.30	224.12

Table 1: Comparisons of serial execution time for direct, CG, and ILUCG linear system solvers when used for the transient simulation of the circuit in Figure 1, where $g_f = 3.0e-5$ and g_s has a conductance of $1e-3$ when linearized about zero.

2.3 CM Implementation

The Connection Machine model CM-2 is a single-instruction multiple data (SIMD) parallel computer consisting of 65,536 bit-serial processors and 2048 Weitek floating-point processors. The bit-serial processors are clustered together into groups of 16 to make a single integrated circuit, and these IC's are connected together in a 12-dimensional hypercube. Two IC's, or 32 processors, share a single Weitek IC. Since the CM-2 contains 2048 Weitek IC's, a speedup of a factor of 2048 over conventional computers containing a single Weitek IC (e.g., a SUN-4) is conceivable.

For an algorithm to approach this peak parallel performance on the CM, it must satisfy three requirements. First, the problem must have enough parallelism to use all the available processors. Second, the algorithm can depend only on local or infrequent interprocessor communication, like on any parallel machine. And third, the algorithm must be mostly *data-parallel* because of the SIMD nature of the Connection Machine. By data-parallel we mean:

- One can identically map individual pieces of data to individual processors for all relevant processors and
- One can operate identically on the data with all the relevant processors

The general circuit simulation problem violates all three of the above constraints, and previous attempts at circuit simulation on the Connection Machine have not yielded impressive results [15, 9]. As we will show in the rest of this section, simulation of grid-based analog signal processors *is* well suited to the CM. These circuits are large, and can be simulated with algorithms that are mostly data-parallel and which depend on mostly nearest-neighbor communication between processors.

2.3.1 Data to Processor Mapping

The two-dimensional nature of grid-based analog signal processing circuits naturally maps into a two-dimensional geometry on the CM, in such a way as to maintain data parallelism and locality. The circuit is divided into identical cells (as shown in Figure 2) and each processor is assigned the data associated with each cell, with nearest-neighbor grid cells being mapped to nearest-neighbor processors. It is an important point that the assigned data includes the node voltages, currents, charges and derivatives, *but not* a complete description of the cell, only the CM's front-end computer has that.

It can be seen from Figure 2 that some elements in each cell cross the cell boundaries, and the communication so implied must be organized carefully to maintain maximum data-parallelism. In our approach, copies are made of *shared-nodes*, by which we mean nodes within each cell to which elements from other cells are connected. These copies are referred to as *pseudo-nodes*. As can be seen in Figure 3, using pseudo-nodes implies that the data for the cell devices is contained completely within the cell.

Two types of consistency between the shared-nodes and pseudo-nodes must be maintained through interprocessor communication, namely:

Voltage Consistency: Pseudo-Nodes must have the same voltage as their corresponding shared nodes.

Charge and Current Consistency: Charges and currents flowing into the pseudo-nodes are summed at the corresponding shared nodes.

This particular mapping of the circuit data insures that the many cells in a large grid can be simulated in a data-parallel fashion. That is, simulation of the entire grid is accomplished by simulating a simple cell using many copies of data, and then enforcing the voltage, charge, and current consistencies for the shared nodes and pseudo-nodes.

2.3.2 Device Evaluation

Evaluating the right-hand side and the Jacobian for the Newton iteration, equation (7), involves computing sums of device currents and charges. Given the previous discussion of the data to processor mapping, the device evaluation portion of the simulation is obvious:

1. Copy node voltages from shared nodes to pseudo-nodes (voltage consistency step)
2. Evaluate cell devices in parallel
3. Sum node charges and currents from pseudo-nodes to shared nodes (charge and current consistency step)

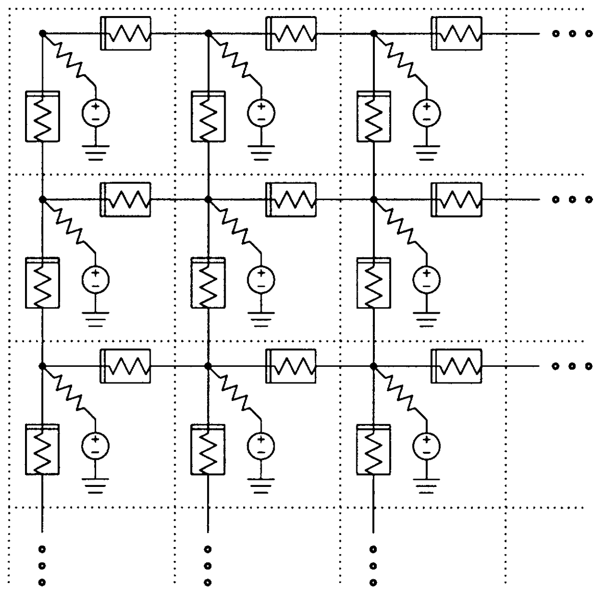

Figure 2: Grid of nonlinear resistors and its division into identical cells. The capacitors have been omitted for clarity.

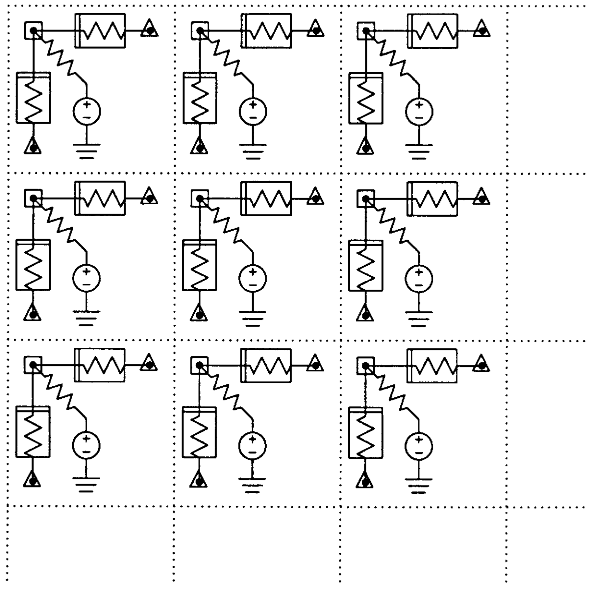

Figure 3: Separation of the cells by duplicating nodes. The shared nodes and pseudo-nodes are outlined with squares and triangles, respectively.

2.3.3 Linear System Solution

As mentioned in the previous section, for the case of grid-based analog circuits, solving the linear Newton iteration equation (6) using CGS is not only easy to parallelize, it is faster than using sparse Gaussian elimination, and nearly as fast as using ILUCGS. There are two parts of the CGS iteration which involve parallel data: the vector inner product and the matrix-vector product. The vector inner-product is accomplished with an in-place multiply and a global sum. The matrix-vector product $y = Ax$ is accomplished with the following sequence of operations:

1. Copy x values from shared nodes to pseudo-nodes (voltage consistency step)
2. Perform matrix-vector product with simple-cell matrix
3. Sum y values from pseudo-nodes to shared nodes (current consistency step)

That the operations involved in the matrix vector product are similar to those required for the device evaluation should come as no surprise. The communication steps are still required for consistency, and the device evaluation step is now replaced by an in-place matrix-vector product where the local matrix corresponds to the linearized conductance matrix of the simple cell circuit.

2.3.4 Grid Boundaries

To this point, we have glossed over what happens on the east and south boundary of the grid. Note that in the actual circuit, the cells on the east and south circuit boundaries respectively do not have east and south connecting elements. However, because all processors contain identical data, the processors on the east and south boundaries do, in effect, have these elements. In order to properly model the boundary behavior on the CM, boundary processors are turned off whenever data corresponding to non-existent elements is manipulated.

For instance, when the devices which connect processors to their east neighbors are evaluated, all processors along the east boundary are turned off. Similarly, when the devices which connect processors to their south neighbors are evaluated, all processors along the south boundary are turned off. See Figure 4. This contextualization in effect produces a representation of the original circuit in which the east and south borders do not have east and south connecting elements, respectively.

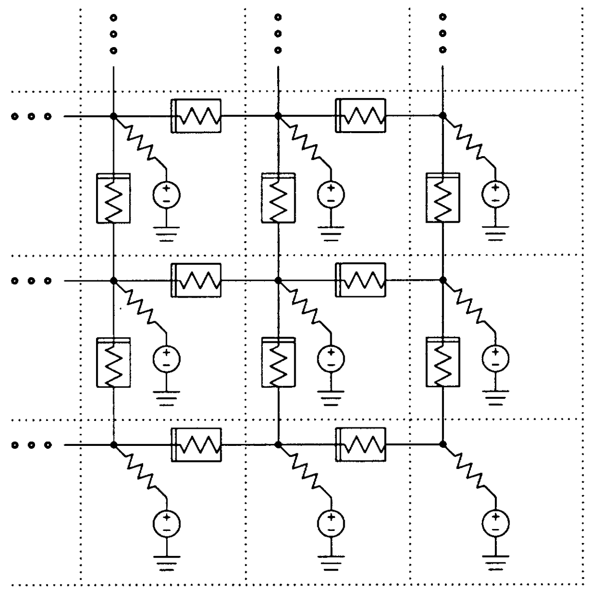

Figure 4: Although every processor must in effect contain the east and south connecting elements, these connecting elements are removed from the simulation by turning off border processors at appropriate times. When the devices which connect processors to their east neighbors are evaluated, the processors along the east boundary are turned off. When the devices which connect processors to their south neighbors are evaluated, all processors along the south boundary are turned off.

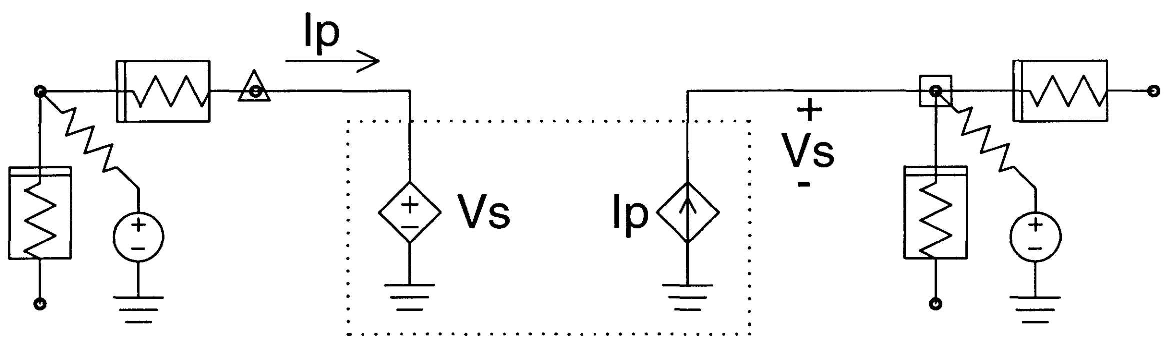

Figure 5: Circuit interpretation for the shared-node / pseudo-node formalism. Here, a single shared-node / pseudo-node pair is shown. Voltage consistency is maintained by a voltage-controlled voltage source (VCVS) connected to the pseudo-node; its voltage is set by the shared node. Current consistency is maintained by a current-controlled current source (CCCS) connected to the shared node; its current is set by the pseudo-node. Charge consistency is maintained in a similar manner to current consistency.

2.3.5 Circuit Interpretation

There is an interesting circuit interpretation to the shared-node / pseudo-node formalism. One can think of the voltage consistency as being maintained by a voltage-controlled voltage source (VCVS) connected to each pseudo-node and whose voltage is set by the appropriate shared nodes. Likewise, one can think of the current consistency as being maintained by a current-controlled current source (CCCS) connected to each shared node and whose current is set by the appropriate pseudo-nodes. An analogous circuit interpretation holds for charge consistency as well, although a charge-controlled charge source is not a standard circuit device. Figure 5 shows a diagram of the circuit interpretation for a single shared-node / pseudo-node pair.

The advantage of this circuit interpretation is that one can thereby rigorously justify the circuit tearing in the sense that the subset of nodes in the torn circuit which correspond to the nodes of the original circuit will have the same voltages before and after tearing. Furthermore, one can construct a "connecting device" with the VCVCS/CCCS pair and use it as a simple way of describing connections between cells for the input circuit specification for CMVSIM.

3 Using CMVSIM

In the following sections, we describe how to use CMVSIM. In general, we will only describe in detail those features of CMVSIM which are not found in SIMLAB. Before starting, the new user is advised to have on hand the *SIMLAB User's Guide* [10] to refer to features of SIMLAB and the *Connection Machine System Software Summary* [13] to refer to interacting with the Connection Machine.

3.1 Starting CMVSIM

CMVSIM is a superset of SIMLAB. As such, it is invoked and used in exactly the same way as SIMLAB and can be used in interactive mode, in batch mode, or both.

CMVSIM is invoked from the command line as follows:

```
cmvsim [-c circuit_file] [-f diary_file] [-vV] [config_file]
```

The command line arguments are:

`-c circuit` specifies a circuit file to be read initially. This option is present in CMVSIM as a holdover from SIMLAB — see the **`circuit`** command in the SIMLAB User's Guide. Note that this circuit will be read initially in the *serial* environment of CMVSIM, i.e., this will not load a grid for simulation on the CM.

`-f diary_file` specifies a diary file to be used initially. See the **`diary`** command in Section A

`-v` turns on verbose mode, in which case CMVSIM emits certain diagnostic messages. See the description of the **`verbose`** environment variable in [10].

`-V` turns on debug mode, in which case CMVSIM emits many diagnostic messages. See the description of the **`simdebug`** environment variable in [10].

`-q` turns off interactive mode. When this command line argument is given, CMVSIM will exit after executing the commands contained in the specified configuration file. If no configuration file is given this argument is ignored.

`config_file` is an initial configuration file read by CMVSIM. If a circuit file is also specified, the configuration file is read *after* the circuit file. CMVSIM will read and execute each line of the configuration file just as if the commands were interactively entered. If a `quit` command is not given in the configuration file, CMVSIM will enter interactive mode after reading it (and executing any commands therein) unless the command line argument -q is given, in which case CMVSIM exits after executing the commands contained in the configuration file.

In general, an interactive simulation session will proceed as follows:

1. Start CMVSIM (perhaps with a configuration file);
2. Configure the simulation environment (interactively, with a configuration file, or both);
3. Run simulation;
4. Plot simulation results;
5. If not done, change circuit parameters and go to 2

3.2 CM Environment

Since CMVSIM is a superset of SIMLAB, it can be used as normal SIMLAB running on a serial machine. In fact, this is the default action — when CMVSIM is initially started, it will be in serial mode and will act just like SIMLAB. To put CMVSIM into parallel mode, one must set the `environment` to `cm`. This is done with the command:

```
set environment cm
```

Several special things happen once CMVSIM is operating under the `cm` environment. First, circuits are no longer loaded with the `circuit` command. Rather, the `grid` command is used (see Section A for a description of the `grid` command and Section 3.3 for a description of CMVSIM circuit files). Second, fewer solution methods are available than with normal SIMLAB (see Section 3.2.1). Finally, extra CM-specific information is printed with the normal simulation statistics listing (see Section 3.2.2).

3.2.1 Available Algorithms

At present, the following algorithms are available for use when CMVSIM is operating under the `cm` environment:

simulate : `pt`

integrate : `trap, gear` (default: `trap`)

solve_nonlinear : `newton` (default: `newton`) `solve_linear` : `cg, cgs, cgnr, pcg, gj` (default: `cg`)

`precondition` : `none, diagonal, block, iblu, miblu` (default: `none`)

Naturally, all of SIMLAB's normal algorithms are available in CMVSIM when CMVSIM is in the `serial` environment. Note that no direct methods are available for `solve_linear` and that `cg` is the default.

3.2.2 Timing

CMVSIM adds some extra timing information to the normal set of simulation statistics printed by SIMLAB. Unfortunately, since the CM and its front end operate asynchronously, there is no single number one can use to absolutely gauge the CM execution time. The extra CM-specific fields are:

Wall clock time: This is the actual elapsed time that was passed between the start and the end of a simulation.

CM busy time: This is the amount of time the CM was busy. This number should always be less than the wall clock time. (Author's note: there seem to be some software bugs in the Paris library which contains the timing functions. This number is not always reliable.)

User time: This is the amount of time that the CMVSIM process was executing instructions. This number should always be less than the wall clock time.

System time: This is the amount of time that the operating system was executing instructions on behalf of the CMVSIM process. This number should always be less than the wall clock time.

If the front-end is running CMVSIM as its only process, the wall clock time, CM busy time, and user time should approximately be the same. In such a case, we generally take the CM busy time to be **the** execution time.

3.3 Circuit Files

Although the user interacts with CMVSIM in batch or interactive mode, circuits are read from grid description files (specified with the `grid` command). At the present time, CMVSIM circuits are required to be Manhattan grids of identical simple cells. CMVSIM constructs the vision circuit grid by reading a description of the simple cell, which is contained in three

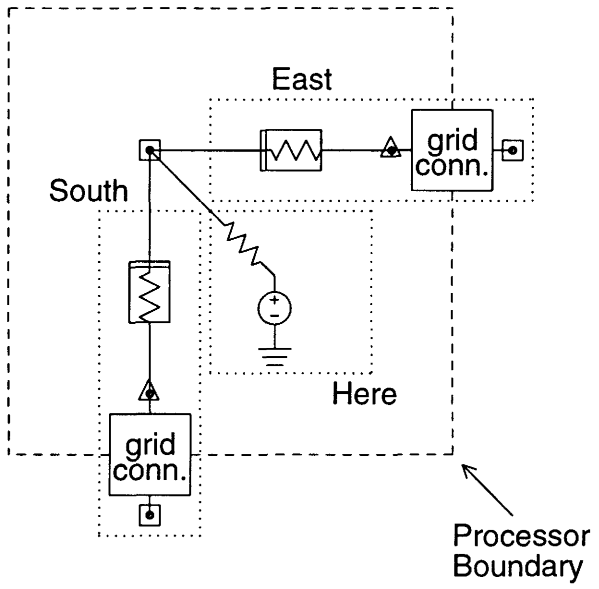

Figure 6: The simple cell is divided into three separate SIMLAB circuits: **east**, **south**, and **here**. The **east** and **south** circuits should contain grid connectors. Although the connector element in some sense crosses the processor boundary, all the information related to it is contained in a single processor.

separate circuit files. Each of the simple cell files contains a complete circuit given in SIMLAB circuit syntax. The three files are denoted to be **here**, **east**, or **south**, and contain the corresponding sub-circuit of the simple cell.

The **east** and **south** circuit files contain the sub-circuits which connect cells, as described in Section 2.3.1. In order to connect cells, the **east** and **south** circuits should contain connector elements. The connector elements in the **east** and **south** circuit files are given the corresponding communication direction (see Sections 2.3.5 and 3.4.2). See Figure 6.

Since the simple cell is divided into three separate circuit files, CMVSIM must be told which nodes are common to all three files. This is done with a "common" comment in the **here** circuit file. The "common" comment has the form:

; **common** ⟨ *node1* ⟩ [⟨ *node2* ⟩ ...]

All common nodes must be listed on the same line with a single common statement. Normally, the common nodes correspond to the shared nodes.

The circuit files must also conform to a special naming convention. In SIMLAB circuit files have a ".rel" extension. In CMVSIM, the grid description files for a particular grid have the same prefix, corresponding to the name of the grid (as given in the **grid** command) and the extensions ".rel.h", ".rel.e", and ".rel.s", for the `here`, `east`, and `south` circuit files, respectively. So if you issue the command

```
grid test
```

to CMVSIM, the program will look for the files test.rel.h, test.rel.e, and test.rel.s with which to construct the vision circuit grid.

Remember, each simple cell sub-circuit file is a complete SIMLAB circuit (see the *SIMLAB User's Guide*). Just as in SIMLAB, each circuit description file contains element models, circuit elements, and subcircuit definitions. Selected simulation parameters can also be specified (see however, Section 3.3.1). Example circuit files can be found in Appendix D.

3.3.1 SIMLAB Control Statements

CMVSIM recognizes the SIMLAB `plot` and `options` control statement. The `plot` statement can appear in any or all of the grid circuit files. However, the `option` statement should only appear in the `here` circuit. If the `option` statement appears in the other grid circuit files, it will be ignored.

3.4 CMVSIM Devices

In addition to the circuit elements available with SIMLAB, the following elements can be used to construct circuits for CMVSIM: grid connector, voltage controlled linear conductance, voltage controlled nonlinear resistor, and dynamic image input source. The voltage controlled linear conductance and voltage controlled nonlinear resistor are idealized circuit elements useful for studying the behavior of certain grid-based vision algorithms (such as those described in [5]).

3.4.1 Devices in Common with SIMLAB

CMVSIM can simulate circuits constructed with any of the the circuit elements available with SIMLAB. These include:

- Level 1 and Level 3 MOS devices (nmos and pmos),
- Bipolar junction transistors (npn and pnp),
- PN junction diodes,

- Linear resistors,
- Linear capacitors,
- Voltage controlled current sources,
- Constant current sources,
- Constant voltage sources,
- Sinusoidal voltage sources, and
- Piecewise linear voltage sources.

The proper usage of these devices is described in the *Simlab User's Guide* [10]. CMVSIM and SIMLAB were both written to be easily extensible — adding new devices is a straightforward task (see the *SIMLAB Programmer's Guide* [4] and the *CMVSIM Programmer's Guide* [3]).

3.4.2 Grid Connector

The grid connector element is used for tearing nodes across processor boundaries, that is, it connects pseudo-nodes to shared nodes (see Section 2.3.1 for a description of pseudo-nodes and shared nodes and Section 2.3.5 for a description of the theory behind the grid connector).

Grid Connector

Element Definition:
⟨ *element name* ⟩ ⟨ *pseudo* ⟩ ⟨ *shared* ⟩ **connector**

Example:
c0 ps0 sh0 **connector**
c1 ps1 sh1 **connector**

⟨ *pseudo* ⟩ and ⟨ *shared* ⟩ are the nodes for which the connector will enforce voltage, current, and charge consistency. Using the pseudo-node / shared node formalism, ⟨*pseudo*⟩ corresponds to the pseudo-node and ⟨ *shared* ⟩ corresponds to the shared node. Presently, the consistency step is performed with nearest-neighbor communication operations. The communication direction is the direction of the circuit file in which the particular instance of the connector is given. That is, all connectors given in the circuit file with the ".e" extension will communicate in the east direction, etc. The communication distance is always unity. Future enhancements of CMVSIM may include means for specifying arbitrary directions and distances. It is meaningless to specify a connector in a ".h" grid circuit file.

3.4.3 Voltage Controlled Conductance

The voltage controlled conductance element is a linear conductance element whose conductance value is proportional to the value of a controlling voltage. This type of element is useful in realizing certain continuations for image smoothing and segmentation (see [5]).

Voltage Controlled Conductance

VCG Model Definition:
model ⟨*name*⟩ **vcg** [**gm** = ⟨*value*⟩]

VCG Element Definitions:
⟨*name*⟩ ⟨*i*⟩ ⟨*j*⟩ ⟨*k*⟩ ⟨*l*⟩ ⟨*model name*⟩ [*gm* = ⟨*value*⟩]
⟨*name*⟩ ⟨*i*⟩ ⟨*j*⟩ ⟨*k*⟩ ⟨*l*⟩ **vcg** [*gm* = ⟨*value*⟩]

Example:
model rt **vcg** gm = 1.0
vcg0 1 2 ctrl 0 rt
vcg1 3 4 ctrl 0 **vcg** gm = 2.0

The voltage controlled conductance element is a linear conductance element connected between nodes ⟨*i*⟩ and ⟨*j*⟩, whose conductance value is determined by the voltage between nodes ⟨*k*⟩ and ⟨*l*⟩. The constitutive relation of the vcg is:

$$i = \mathrm{gm} * (v_k - v_l) * (v_i - v_j).$$

3.4.4 Voltage Controlled Nonlinear Resistor

The voltage controlled nonlinear resistor is a parameterized nonlinear circuit element whose nonlinear characteristics are determined by the value of a controlling voltage. This type of element is useful in realizing certain continuations for image smoothing and segmentation (see [5]).

Voltage Controlled Nonlinear Resistor

NLR4 Model Definition:
model ⟨ *name* ⟩ **nlr4** [*alpha* = ⟨ *value* ⟩ *beta* = ⟨ *value* ⟩ *gamma* = ⟨ *value* ⟩]

NLR4 Element Definitions:
⟨ *name* ⟩ ⟨ *i* ⟩ ⟨ *j* ⟩ ⟨ *k* ⟩ ⟨ *l* ⟩ ⟨ *model name* ⟩ \
    [*alpha* = ⟨ *value* ⟩ *beta* = ⟨ *value* ⟩ *gamma* = ⟨ *value* ⟩]
⟨ *name* ⟩ ⟨ *i* ⟩ ⟨ *j* ⟩ ⟨ *k* ⟩ ⟨ *l* ⟩ **nlr4** \
    [*alpha* = ⟨ *value* ⟩ *beta* = ⟨ *value* ⟩ *gamma* = ⟨ *value* ⟩]

Example:
model rt **nlr4** alpha = 1.e-3 gamma = 2e-5 beta = 5e5
nlr40 1 2 ctrl 0 rt
nlr41 3 4 ctrl 0 **nlr4** alpha = 2.e-3 gamma = 1e-5 beta = 1e6

The voltage controlled nonlinear resistor is a nonlinear conductance element connected between nodes ⟨ *i* ⟩ and ⟨ *j* ⟩, parameterized by the voltage between nodes ⟨ *k* ⟩ and ⟨ *l* ⟩. The constitutive relation of the nlr4 is:

$$i = \frac{\text{alpha} * (v_i - v_j)}{1 + \exp\{-\text{beta} * (v_k - v_l)[\text{gamma} - \text{alpha} * (v_i - v_j)^2]\}}.$$

3.4.5 Image Source Motion Element

The IMGSRC motion element is in fact a piecewise-linear voltage source which is distinguishable by placing a *magic number* as its first voltage value. IMGSRC elements are read by the SIMLAB engine as simple pwl's, but when built on the *Connection Machine*, the *magic number* allows us to separate them from the other pwl's. IMGSRC act as DC voltage sources unless a motion command is issued. The image values read, at appropriate times, from the files indicated in the motion description file are stored in an IMGSRC (see Section A) and the source performs like a pwl. The use of IMGSRC is meaningless unless a simulation with moving images is being performed. Similarly the motion command will have no effect unless an IMGSRC is present in the circuit.

Image Source Motion Element

IMGSRC Element Definitions:

$\langle\, name \,\rangle\ \langle\, Node\ i \,\rangle\ \langle\, Node\ j \,\rangle$ **pwl** $[delay = \langle\, value \,\rangle period = \langle\, value \,\rangle] \backslash$
$t0 = \langle\, value \,\rangle v0 = \langle\, magicnumber \,\rangle [...t24 = \langle\, value \,\rangle v24 = \langle\, value \,\rangle]$

Example:

imgin 1 0 **pwl** delay=0ns period=100ns \
t0=0ns v0=1.23e45

$\langle\, Node\ i \,\rangle$ is the positive node of the voltage source, and $\langle\, Node\ j \,\rangle$ must be the ground node. There are three types of parameters for the IMGSRC just as for a normal **pwl**. However, although there can be up to 25 (0 through 24) breakpoints in the piecewise linear waveform, which are pairs (ti=sec vi=volts), for an IMGSRC only the first voltage value is relevant since the real voltage values will be read from image files. The magic number is historically kept at 1.23e45 but can in fact be any number larger than 1.0e12. There is also a **delay** parameter, which causes each breakpoint is delayed by the amount specified. Finally, there is the **period** parameter, which, when specified, causes the waveform to repeat after the given number of seconds. Not that the delay and period parameter are relevant and will be considered whenever specified. Special care is necessary to ensure that the period given in the element definition line is not smaller than the time specified for the last image in the motion image description file.

3.4.6 Idealized Mead HRES

The IHRES element is an idealization of Mead's HRES circuit [7]. It behaves as a linear resistor for small voltage difference and has an hyperbolic tangent characteristic function for larger currents.

Idealized HRES

IHRES Model Definition:

model ⟨ *name* ⟩ **ihres** [*isat* = ⟨ *value* ⟩]

IHRES Element Definitions:

⟨ *name* ⟩ ⟨ *i* ⟩ ⟨ *j* ⟩ ⟨ *model name* ⟩ \
    [*isat* = ⟨ *value* ⟩]

⟨ *name* ⟩ ⟨ *i* ⟩ ⟨ *j* ⟩ **ihres** \
    [*isat* = ⟨ *value* ⟩]

Example:

model hr **ihres** isat = 2.e-8
ihres0 1 2 hr
ihres1 3 4 **hres** isat=2.25.e-8

The ideal horizontal resistor element is a nonlinear conductance element connected between nodes ⟨ *i* ⟩ and ⟨ *j* ⟩, whose constitutive relation is:

$$i = \text{isat} * tanh(\frac{v_i - v_j}{2V_T})$$

where V_T is the thermal voltage.

3.5 Image Inputs

3.5.1 Static Images

In order to test the correctness of a vision circuit one must check that its functionality is correct, i.e., that it processes an image in the desired manner. Several functions are therefore provided in CMVSIM for handling images.

An image is usually used as input to CMVSIM in order to specify some initial condition on the circuit or as a source value. CMVSIM is able to read image files from a front-end computer disk and from the DataVault [12]. Images stored in a front-end computer disk can be in a variety of formats, namely "ascii", "img" and "matlab" formats. See Section 3.5.3.

Images stored in the DataVault must be stored as 8-bit quantities written from a CM memory location. When they are read, they are converted into voltage values and stored into the proper location.

An image is read into CMVSIM by means of the `image` command or the `pimage` command. Both commands take the name of the file containing the image data as an argument. In the case of the `image` command, the file resides on a disk in the front-end. For the `pimage` command, on the other hand, the image file resides in the DataVault. In both cases, the image is read into some CM memory location according to the following rules:

- If an imgsrc exists in the circuit, the image is loaded into that source which is then used as a DC source;
- Otherwise, if a DC source exists in the circuit, the image is loaded into that source;

If no circuit has been read, or if no source exists to accept the image, an error condition results.

CMVSIM also has the ability to store processed (or unprocessed) images onto either a front-end disk or the DataVault. This is done with the `iplot` command or the `piplot` command. The `iplot` command saves the output image, i.e., the voltage values of the nodes specified for plotting (with the circuit plot statement) from each processor into a file on the front-end computer. The format used for the output image is the same as that of the last input image read. Note that if the last image loaded was read from the DataVault (either by using `pimage` or caused by a `pmotion` command), the output image will be stored in the ".img" format.

If the `iplot` or `piplot` command is issued before a circuit has been read, an error is issued. Otherwise, either command will create of a set of images files, one for each node marked for plotting, containing the voltage values at those nodes. The syntax for the names of these files is obtained by prefixing the node number with the name of the last image read, as in:

```
  image_name_node_num.ext
```

where ".ext" is an extension referring to the format of the image being written.

3.5.2 Dynamic Images

The ability to process dynamic images is one of the strongest features of CMVSIM. It allows for the simultaneous verification of a circuit's functionality and correctness. By dynamic images, we mean a time sequence of image frames which are sequentially loaded into CMVSIM as input. To approximate the continuous nature of an actual moving image, the input values at intermediate times (i.e., at times between image frames) are derived by linearly interpolating between the nearest timepoint values at each pixel. Note that the interpolations are therefore *temporal*, not spatial.

Processing dynamic images is done with the `pmotion` command. This command takes as argument a description file that contains information about the sequence of images to be loaded.

```
=> pmotion motion.dsc
```

The description file will contains pairs of values ⟨*time image−file*⟩ indicating which images are to be read at what times. Each line contains a pair of values separated by spaces. At the times specified by the description file (the times are the first value of each tuple), two things happen. First, the specified input image (the file whose name is the second tuple value) is read from the DataVault and loaded onto the vision grid. Second, an output file for each of the plottable nodes is written to the DataVault.

A description file might look like:

```
; Image of moving object.
; Frames obtained at a rate of 10 per second.
0.0      object0.img
0.1      object1.img
0.2      object2.img
0.3      object3.img
0.4      object4.img
```

The times in the description file should always be strictly increasing.

Reading each image is similar to issuing a `pimage` command (see Appendix A). The image file stored in the DataVault contains 8-bit quantities which are read, converted into voltage values and stored into the proper location. If a simulation with dynamic images is to be performed, a special device called an Image Source (`imgsrc`) must exist in the circuit being simulated. If no such source exists, an error condition is issued and the simulation is aborted. For a description of `imgsrc` devices, see Section 3.4.5.

It is extremely important to recall that an image (like any other file) stored in the DataVault, has a geometry associated with it and confusion will arise when trying to read an image which was stored with geometry information different from the one currently being used. Since most images initially exist in the front-end computer we suggest that a fake simulation should be run initially that merely uses CMVSIM to read the sequence of images and store them in the DataVault (via a sequence of `image` and `piplot` commands). This will ensure that the images to be used have the correct geometry. For more information refer to the proper CM documentation [12].

(**Authors' note:** Thinking Machines has recently developed an image file interface [14]. CMVSIM will probably migrate to that format sometime in the future.)

3.5.3 File Formats

A conversion program for converting between the three image input formats is included in the CMVSIM distribution and should be in the util directory.

Img Format

The img file format is a simple grey-scale format. The file contains two 16-bit words as a header, indicating the image width and height, respectively, followed by the image data. The image data is stored row-wise in typical raster-scan fashion, with one byte per pixel. The voltage range of the image is $0 - 5\ V$, that is, a zero pixel is $0\ V$ and a 255 pixel is $5\ V$.

The following chart shows the img file format:

Byte	Contents
0	width MSB
1	width LSB
2	height MSB
3	height LSB
4 - EOF	"height" rows of "width" bytes of image data

The authors have miscellaneous conversion programs for converting between popular image formats (such as tiff) and img. Since these programs require other software libraries (such as the tiff libraries) to work properly, they are not included with the CMVSIM distribution, but are available on request.

Matlab Format

When developing and testing vision algorithms with CMVSIM, it is often desirable to use artificial input images. We found that the MATLAB program [6] was useful for this. Typically, one will create a matrix in MATLAB which is to be the artificial image and save the matrix with MATLAB's `save` command. CMVSIM can then read in the matrix and use it as an input image. Since the MATLAB matrices are stored as floating-point quantities, the matrix entry values correspond directly to voltages.

Output results from CMVSIM runs can be loaded into MATLAB with MATLAB's `load` command.

Ascii Format

One slight problem with the MATLAB format is that it is machine dependent. This can be a problem if matrices are constructed on one machine and CMVSIM run on another. To make the MATLAB matrices somewhat more portable, an equivalent ascii format was developed. A filter program is then used to convert back and forth between matlab and ascii formats. (The filter program is available upon request.)

3.6 Plotting

Outputting the results of a simulation run in CMVSIM is done in a manner compatible with SIMLAB.

A node within a simple cell is marked for plotting if it appears with a plot statement in any of the grid circuit files. Note that if a shared node appears in a plot statement in more than one of the grid circuit files, it will be multiply plotted, a situation that should be avoided to save disk space and execution time. Once a simple cell node is marked for plotting, it will will only be plotted from the simple cells corresponding to the processors which are marked for plotting. By default, only four processors are initially marked for plotting, namely the centers of the four grid quadrants (i.e., processors with coordinates $(\frac{N}{4}, \frac{N}{4})$, $(\frac{N}{4}, \frac{3N}{4})$, $(\frac{3N}{4}, \frac{N}{4})$ and $(\frac{3N}{4}, \frac{3N}{4})$, where N is the grid size in each dimension).

The command `procplot` allows the user to select a different set of processors to have their voltages plotted, by specifying a list of processors (and correspondingly grid circuitry) to be set for plotting. The syntax of this command which is shown in Appendix A consists of the command itself followed by a list of x, y coordinate pairs inside parenthesis. An example is

```
procplot (1,1) (2,3) (5,7) (25,32)
```

Once a set of processors is selected for plotting, that selection remains in effect until a new `procplot` command is issued.

The plotted values are stored in a file called *circuit.*"trans" (where *circuit* is the circuit name, that is, the prefix of the grid circuit files) which is kept in the current working directory. The format of the output file consists of a sequence of three-tuples as follows:

$$(xproc, yproc)node-name \;\; time \;\; voltage$$

For example,

```
(24,24)input  0.0  5.0
(24,24)output 0.0  3.14
(48,2)input   0.0  2.25
(48,2)output  0.0  2.55
(24,24)input  0.1  5.0
(24,24)output 0.1  3.15
(48,2)input   0.1  2.25
```

```
(48,2)output  0.1  2.75
```

The contents of this file can be previewed with the help of some plotting tool such as SIMGRAPH (see [10]) or the user's favorite two-dimensional plotting package (provided it is able to read the format of the plot file). Consult [10] for further information about viewing the simulation results.

Note that image plotting and motion image plotting (`iplot`, `piplot` and `pmotion`) use the circuit plot statement to determine which nodes will be recorded for the image output files. However, in these cases, the voltage from the watched nodes is recorded from *every* processor. Each image file will be rather large; if too many nodes are contained in the plot statement, many large files will be created, degrading performance and wasting disk space. We therefore recommend judicious use of the plot statement when using `iplot`, `piplot` and `pmotion`.

References

[1] R. Burch, K. Mayaram, J.-H. Chern, P. Yang, and P. Cox, *PGS and PLUCGS - Two new matrix solution techniques for general circuit simulation*, in International Conference on Computer Aided-Design, Santa Clara, California, November 1989, pp. 408–411.

[2] W. D. Hillis, *The Connection Machine*, MIT Press, New Haven, CT, 1985.

[3] A. Lumsdaine, M. Silveira, and J. White, *CMVSIM programmer's guide.* Research Laboratory of Electronics, Massachusetts Institute of Technology. Unpublished, 1990.

[4] ——, *SIMLAB programmer's guide.* Research Laboratory of Electronics, Massachusetts Institute of Technology. Unpublished, 1990.

[5] A. Lumsdaine, J. Wyatt, and I. Elfadel, *Nonlinear analog networks for image smoothing and segmentation*, in International Symposium on Circuits and Systems, New Orleans, Louisiana, May 1990, pp. 987–991.

[6] The MathWorks, *Matlab User's Guide*, Waltham, MA, June 1990.

[7] C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley, Reading, MA, 1988.

[8] L. W. Nagel, *SPICE2: A computer program to simulate semiconductor circuits*, Tech. Report ERL M520, Electronics Research Laboratory Report, University of California, Berkeley, Berkeley, California, May 1975.

[9] L. M. Silveira, *Circuit simulation algorithms for massively parallel processors*, master's thesis, Massachusetts Institute of Technology, May 1990.

[10] M. Silveira, A. Lumsdaine, and J. White, *SIMLAB users' guide.* Research Laboratory of Electronics, Massachusetts Institute of Technology, 1990.

[11] P. Sonneveld, *CGS, a fast Lanczos-type solver for nonsymmetric linear systems*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 36–52.

[12] Thinking Machines Corporation, *CM I/O System Programming Guide*, Cambridge, MA, July 1990.

[13] ——, *Connection Machine System Software Summary*, Cambridge, MA, November 1990.

[14] ——, *Image File Interface Reference Manual*, Cambridge, MA, November 1990.

[15] D. M. Webber and A. Sangiovanni-Vincentelli, *Circuit simulation on the Connection Machine*, in 24^{th} ACM/IEEE Design Automation Conference, June 1987.

[16] J. L. WYATT JR., *et al*, *The first two years of the MIT vision chip project*, Tech. Report VLSI Memo 90-605, Massachusetts Institute of Technology, October 1990.

A Commands

A.1 Commands in Common with SIMLAB

The following is a list of commands that CMVSIM has in common with SIMLAB. For more information about any of these, see [10].

- `cd` – change current working directory,
- `circuit` – load circuit (for serial mode only),
- `continue` – continue transient simulation,
- `diary` – record session to file,
- `help` – provide help,
- `history` – list previous command lines,
- `plot` – plot transient output,
- `printenv` – print value of environment variable,
- `quit` – quit CMVSIM,
- `run` – perform simulation,
- `set` – set simulation algorithm,
- `setenv` – set environment variable,
- `sh` – execute shell command,
- `show` – show simulation algorithms and values,
- `source` – execute commands from a file,
- `unsetenv` – unset environment variable,
- `who` – display variables.

A.2 Commands Unique to CMVSIM

This section contains a description of commands unique to CMVSIM.

---

grid grid

Purpose:
Load vision grid for simulation.

Synopsis:
`grid` basename [size]

Description:
`grid` loads a vision grid into CMVSIM for simulation. "basename" specifies the base name for the grid configuration files. The program will look for the files basename.rel.h, basename.rel.e, and basename.rel.s (all of which must exist) with which to construct the vision circuit grid. Some application variables may be affected by the grid configuration files.

The optional parameter "size" is an integer which specifies the size of the grid to be size $\times$ size. If no value is given, a default size of 64 is used.

CMVSIM recognizes the SIMLAB `plot` and `options` control statement. The `plot` statement can appear in any or all of the grid circuit files. However, the `option` statement should only appear in the ".h" (the `here`) circuit. If the `option` statement appears in the other grid circuit files, it will be ignored.

Diagnostics:
The grid configuration files must be specified in SIMLAB format. See Section 3.3 for more information.

image image

Purpose:
Load an image from the front-end onto a vision grid.

Synopsis:
`image` filename

Description:
The **`image`** command loads an image from a front-end file onto the current vision grid. If no grid has been loaded, an error will be indicated. The image file can be in one of two formats: .img, .mat, or .asc (see Section 3.5.3). The image file resides in the front-end computer and it is loaded into one of the following by order: an image source device (known as "imgsrc"), a DC source with a magic value (1.23e45 or in fact aything larger than 1.0e12), the first DC source in the internal list. Make sure the magic value is set on one, and only one of the sources.

Diagnostics:
The image should be the same size as the grid. A warning is given in case the magic value is not found.

Purpose:
Save output image to the front-end.

Synopsis:
iplot

Description:
The `iplot` command saves the output image, i.e., the voltage values of the `here` nodes in each processor. The output image is saved in the same format as the last image read with either an `image` or `pimage` command (note that `pmotion` commands are handled as sequences of `pimage` commands). The output image is stored on a file in the front-end computer.

Diagnostics:
If no image was loaded, a warning is given.

pimage pimage

Purpose:
Load an image from the DataVault onto a vision grid.

Synopsis:
`image` filename

Description:
The `pimage` command loads an image from a DataVault file onto the current vision grid. If no grid has been loaded, an error will be indicated. The image file resides in the DataVault and it is loaded into one of the following by order: an image source device (known as "imgsrc"), a DC source with a magic value (1.23e45 or in fact aything larger than 1.0e12), the first DC source in the internal list. Make sure the magic value is set on one, and only one of the sources.

Diagnostics:
The image should be the same size as the grid. A warning is given in case the magic value is not found. In order to properly operate with the DataVault, the proper Unix environment variables must be set. That includes DVHOSTNAME with the DataVault's name, and DVWD with the working directory where the files reside on the DataVault.

piplot piplot

Purpose:
Save output image to the DataVault.

Synopsis:
piplot

Description:
The `piplot` command saves the output image, i.e., the voltage values of the `here` nodes in each processor. The output image is saved to a file on the DataVault.

Diagnostics:
If no image was loaded, a warning is given. As with any operation involving the DataVault, the proper Unix environment variables must be set. That includes DVHOSTNAME with the DataVault's name, and DVWD with the working directory where the files reside on the DataVault.

pmotion

Purpose:
Define parameters for image motion.

Synopsis:
`pmotion` filename

Description:
The `pmotion` command loads information about image motion. The description file that is read contains pairs $\langle time\ image-file\rangle$ of values indicating which images are to be read at what times. Each line contains a pair of values which are separated by blank spaces. At the times specified by the description file (the times are the first value of each tuple), two things happen. First, the specified input image (the file whose name is the second tuple value) is read from the DataVault and loaded onto the vision grid. Second, an output file for each of the plottable nodes is written to the DataVault.

Reading each image is similar to issuing a `pimage` command.

Diagnostics:
Times must be monotonically increasing. Only the first file is checked for existence.

Purpose:
Select a set of processors to be marked for plotting

Synopsis:
procplot [(x1,y1) [(x2,y2) [...]]]

Description:
The `procplot` command selects a set of processors in the grid to be marked for plotting. The processors are given in a list of coordinates pairs separated by spaces. The voltage of each node marked for plotting in each of the selected processors, will be output at the correct points in time. A node is marked for plotting if it appears in a plot statement in one of the circuit files.

If no arguments are given, the currently selected set of processors will be listed.

Diagnostics:
If no nodes are marked for plotting, a warning is issued. If a processor is marked more than once for plotting, it will be plotted more than once (i.e., there is not yet any checking of repetitions).

B Variables

B.1 Variables in Common with SIMLAB

The following is a list of variables that CMVSIM has in common with SIMLAB. For more information about any of these, see [10].

- `True, False, pi` – system constants,
- `stop` – final simulation time,
- `cmin` – minimum capacitance to ground at each node,
- `gmin` – minimum conductance to ground at each node,
- `nrvabs`, `nrvrel` – Newton-Raphson absolute and relative voltage convergence criteria,
- `nrcabs`, `nrcrel` – Newton-Raphson absolute and relative current convergence criteria,
- `lterel`, `lteabs` – absolute and relative local truncation error,
- `nralpha` – Newton-Raphson ΔV step limit,
- `newjacob` – Number of Newton iterations before updating the Jacobian,
- `maxtrnr`, `maxdcnr` – Maximum number of *transient* and *dc* Newton-Raphson iterations allowed,
- `dodc`, `dotran` – flags indicating *dc* or *transient* solution is desired,
- `verbose` – flag indicating verbose mode,
- `simdebug` – flag indicating debug mode.

B.2 Variables Unique to CMVSIM

cmdebug

Description:
Flag indicating cmdebug mode. This is a debugging mode which will print out lots of information about the CM as the simulation is progressing.

Default:
`cmdebug = False`

cmmem

Description:
A diagnostic variable which shows how much of the CM memory has been allocated by the simulator with explicit calls to `palloc()`. It does *not* represent the total amount of CM memory which is being used by CMVSIM, since some memory may be used by the stack, by automatic variables, etc. You should never set this variable.

Default:
N/A

maxcg, maxdccg

Description:
These variables give the maximum number of iterations that CMVSIM will use in the CG algorithm for linear system solution. `maxcg` is used for transient simulation; `maxdccg` is used for DC simulation. If a value of -1 is given, the CG algorithm will take N iterations, which should guarantee convergence. Note, however, that this is not practical for large circuits.

Default:

`maxcg`	=	32
`maxdccg`	=	-1

Diagnostics:
These variables will only appear in the CMVSIM environment when `solve_linear` is set to `cg`.

maxcgnr, maxdccgnr

Description:
These variables give the maximum number of iterations that CMVSIM will use in the CGNR algorithm for linear system solution. `maxcgnr` is used for transient simulation; `maxdccgnr` is used for DC simulation. If a value of -1 is given, the CGNR algorithm will take N iterations, which should guarantee convergence. Note, however, that this is not practical for large circuits.

Default:

`maxcgnr`	=	32
`maxdccgnr`	=	-1

Diagnostics:
These variables will only appear in the CMVSIM environment when `solve_linear` is set to `cgnr`.

maxcgs, maxdccgs

Description:

These variables give the maximum number of iterations that CMVSIM will use in the CGS algorithm for linear system solution. `maxcgs` is used for transient simulation; `maxdccgs` is used for DC simulation. If a value of -1 is given, the CGS algorithm will take N iterations, which should guarantee convergence. Note, however, that this is not practical for large circuits.

Default:

`maxcgs`	=	32
`maxdccgs`	=	-1

Diagnostics:

These variables will only appear in the CMVSIM environment when `solve_linear` is set to `cgs`.

maxpcg, maxdcpcg

Description:

These variables give the maximum number of iterations that CMVSIM will use in the PCG algorithm for linear system solution. `maxpcg` is used for transient simulation; `maxdcpcg` is used for DC simulation. If a value of -1 is given, the PCG algorithm will take N iterations, which should guarantee convergence. Note, however, that this is not practical for large circuits.

Default:

`maxpcg`	=	32
`maxdcpcg`	=	-1

Diagnostics:

These variables will only appear in the CMVSIM environment when `solve_linear` is set to `pcg`.

block_steps

Description:
This variable specifies how many times to repeat the block diagonal preconditioning step during each application of the block diagonal preconditioner. A value of one seems to give the best results.

Default:
`block_steps` = 1

Diagnostics:
This variable will only appear in the CMVSIM environment when `solve_linear` is set to `pcg` or `cgs` and `precondition` is set to block.

C Algorithms

One of the most powerful features of CMVSIM is the ability to specify the algorithms to be used at different levels of the simulation for the various simulation modes.

In this section, the functional controls for CMVSIM are described, along with valid values for them when operating in the `cm` environment.

The combination which seems to work the best for VLSI circuits is:

`environment: cm`

`simulate: pt`

`integrate: trap`

`solve_nonlinear: newton`

`solve_linear: cgs`

`precondition: block`

Note that these are the defaults values for **environment**, **simulate**, **integrate**, and **solve_nonlinear**. The others would need to be set before simulating any circuits, with the following sequence of commands:

```
set solve_linear cgs
set precondition block
```

The CMVSIM variable **block_steps** controls how many preconditioning iterations are done per linear solution iteration. The best value for this seems to be one, which is the default.

environment

Description:
Specifies what type of environment CMVSIM should run under. In serial mode, CMVSIM is identical to SIMLAB. Different environments cannot be specified for DC and Transient simulation. To simulate grid-based analog arrays on the Connection Machine, `environment` must be set to `cm`.

Values:

`cm`	(Connection Machine environment)
`serial`	(sequential computer environment)

Default:
`cm`

simulate

Description:

Specifies what type of simulation structure to use. CMVSIM only supports pointwise simulation methods at the moment, so **`simulate`** must be left as `pt`.

Values:

`(dc)`	`pt`	(pointwise solution in time)
`(tran)`	`pt`	(pointwise solution in time)

Default:

`(dc)`	`pt`
`(tran)`	`pt`

integrate

Description:

Specifies which integration method to use. Has no meaning except in transient mode.

Values:

`trap`	(trapezoidal integration)
`gear2`	(second-order backward-difference integration)

Default:

`trap`

solve_nonlinear

Description:

Specifies function to be used for nonlinear system solution during simulation. Different functions may be specified for DC and Transient simulation. At present, CMVSIM only includes a Newton-Raphson solver, however.

Values:

`(dc)`	`newton`	(Newton-Raphson iteration)
`(tran)`	`newton`	(Newton-Raphson iteration)

Default:

`(dc)`	`newton`
`(tran)`	`newton`

solve_linear

Description:

Specifies function to be used for linear system solution during simulation. Different functions can be specified for DC and Transient simulation. Note that the default method, `cg` is intended for symmetric systems. Non-symmetric systems should use `cgnr` or `cgs`. Preconditioners can be used with `pcg` and `cgs`.

Values:

`(dc)`	`cg`	(conjugate gradient)
`(tran)`	`cg`	(conjugate gradient)
`(dc)`	`cgnr`	(CG applied to normal eqations)
`(tran)`	`cgnr`	(CG applied to normal eqations)
`(dc)`	`pcg`	(preconditioned conjugate gradient)
`(tran)`	`pcg`	(preconditioned conjugate gradient)
`(dc)`	`cgs`	(conjugate gradient squared)
`(tran)`	`cgs`	(conjugate gradient squared)

Default:

`(dc)`	`cg`
`(tran)`	`cg`

precondition

Description:

Specifies function to be used for preconditioning during linear system solution. The `precondition` control is only available for the `pcg` and `cgs` linear solvers. Different preconditioners can be specified for DC and Transient simulation. The preconditioners `block`, `iblu`, and `miblu` use the CMVSIM variable `block_steps` to specify the number of preconditioning iterations taken per linear solution iteration (the default is one).

Values:

(dc)	none	(identity)
(tran)	none	(identity)
(dc)	diag	(diagonal)
(tran)	diag	(diagonal)
(dc)	block	(block diagonal)
(tran)	block	(block diagonal)
(dc)	iblu	(incomplete block diagonal)
(tran)	iblu	(incomplete block diagonal)
(dc)	miblu	(incomplete block diagonal, markowitz order)
(tran)	miblu	(incomplete block diagonal, markowitz order)

Default:

(dc)	none
(tran)	none

D Examples

D.1 Starting a CMVSIM Session

In order to run a CMVSIM session, you must first be attached to a Connection Machine by using the command **cmattach**. One can attach in either interactive or batch mode. For instance, if at the shell prompt you type:

```
% cmattach
```

you will be attached to a Connection Machine in a new shell. You can now start up and run CMVSIM (in either interactive or batch mode).

Before attempting to attach, you might want to check if there are any free Connection Machines available on your host. This is done with the command:

```
% cmfinger
```

One useful feature is to attach to a Connection Machine in batch mode to run a CMVSIM script in batch mode. For instance, to run the CMVSIM script `batch.cfg`, you would type something like:

```
% cmattach cmvsim -q batch.cfg
```

This is handy for running unattended CMVSIM jobs. The output can even be piped to a file (or `/dev/null` and the entire job put in the background, as with:

```
% cmattach cmvsim -q batch.cfg > batch.out &
```

The `cmattach` command has several command line options with which to specify which Connection Machine to attach to, which interface to use, and so forth. To get a quick listing of these, type

```
% cmattach -h
```

For more detailed information, see [13].

D.2 Example Input Files

In this section, we present an assortment of example CMVSIM scripts.

The first example, in file `ex1.cfg`, shows a simulation run in which a 32×32 resistive grid is simulated with an input image.

ex1.cfg

```
% Example CMVSIM script

% Save output to diary file
diary ex1.doc

% Load 32 x 32 grid
grid lres 32

% Load input image
image rand32.mat

% Set up some simulation parameters
cmin = 1.e-7
gmin = 0
maxdcnr = 20
stop = 1

% Set up max iteration variables
maxcg = 32
maxdccg = 256

% Only do dc simulation
dodc = 1
dotran = 0
run

% Save the output image
iplot

% Now do dc and transient
dodc = 1
dotran = 1
run
% Save the output image
iplot

diary off
```

The second example, in file `ex2.cfg`, shows a simulation run in which a 201×201 retina chip is simulated with a dynamic input image. The input image description file is shown in soda.dsc.

ex2.cfg

```
% Example CMVSIM script

% Save output to diary file
diary ex2.doc

% Load 201 x 201 grid
grid ret 201

% Use cgs solver with block diagonal preconditioner
set solve_linear cgs
set precondition block

% Don't do dc
dodc = 0

% Specify the dynamic input image
pmotion soda.dsc

run
```

soda.dsc

```
; image of a soda can undergoing a translation
; speed is 1 frame per 0.03 seconds (approx.  1/30)
0.0 soda00.img
0.03 soda11.img
0.06 soda22.img
0.09 soda33.img
0.12 soda44.img
0.15 soda55.img
0.18 soda66.img
0.21 soda77.img
0.24 soda88.img
0.27 soda99.img
0.30 soda1010.img
```

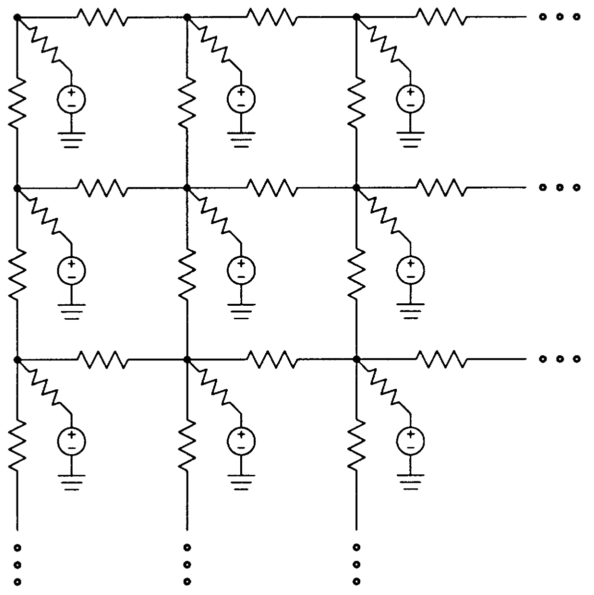

Figure 7: Grid of linear resistors.

D.3 Example Circuit Files

In this section, we give examples of some CMVSIM circuits. The most important detail to watch is how the connector elements are used. CMVSIM circuit files are specified in SIMLAB circuit format. For more information about specifying SIMLAB circuits, see the *SIMLAB User's Guide* [10].

D.3.1 Resistive Grid

The first example we want to examine is a simple resistive grid, as shown in Figure 7. The circuit is a resistive grid having 1 k resistors between grid nodes and 1 k resistors from the grid nodes to the voltage source inputs. The simple cell that is needed to construct such a circuit is shown in Figure 8.

Three circuit files are needed to construct the simple cell shown in Figure 8: `lres.rel.e`, `lres.rel.s`, and `lres.rel.h`. Note that since the node `here` is common to all three circuit files, it must be listed in the `common` statement in the ".h" file (lres.rel.h). The listings for these files are given as follows:

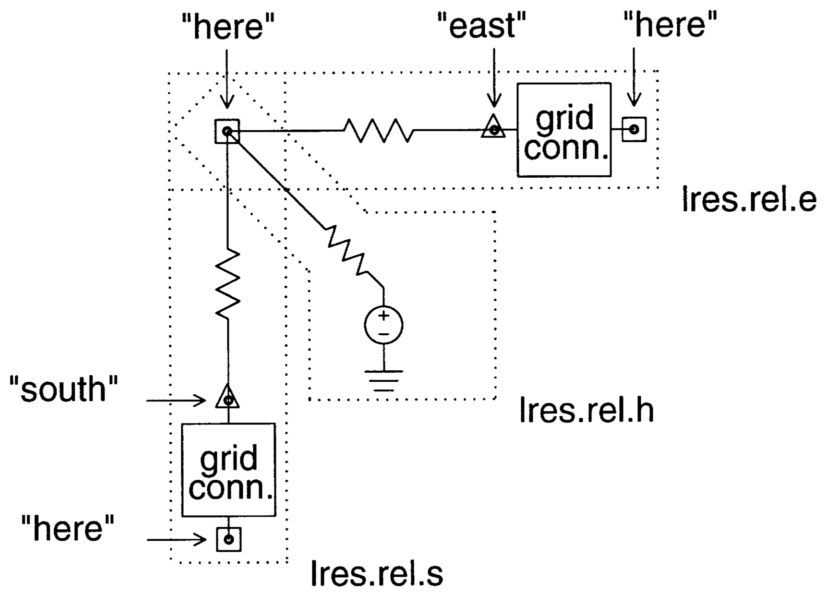

Figure 8: Example simple cell for constructing a resistive grid. Three circuit files are needed to construct the simple cell: `lres.rel.e`, `lres.rel.s`, and `lres.rel.h`. Note that the node **here** is common to all three circuit files.

`lres.rel.e`

```
global 0

r1 here east r r=1k
c0 east here connector
```

`lres.rel.s`

```
global 0

r1 here south r r=1k
c0 south here connector
```

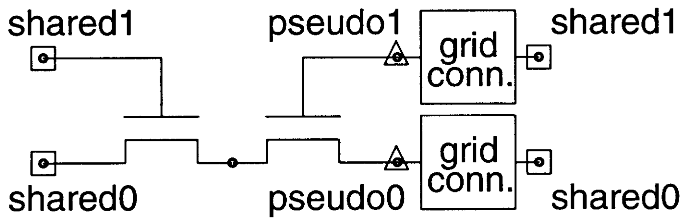

Figure 9: The east connecting HRES circuitry, contained in the file `ret.rel.e`.

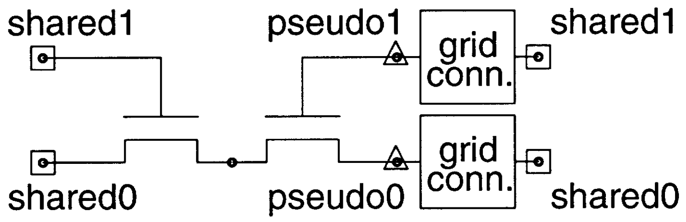

Figure 10: The south connecting HRES circuitry, contained in the file `ret.rel.s`.

```
lres.rel.h

global 0

; common here

r1 here 1 r r=1k
v1 1 0 dc v=2

cmvsim options cmin = 1.e-6
plot here
```

D.3.2 Silicon Retina

The next example circuit we want to consider is slightly more complicated — a silicon retina [7]. This circuit consists of a grid of transamp cells interconnected by small semiconductor circuits ("HRES" circuits in Mead's terminology). The simple cell to construct a retina grid is shown in Figures 9 – 11.

The following are the three circuit files which make up the silicon retina grid circuit. Note that the connecting circuitry has two connectors in each direction.

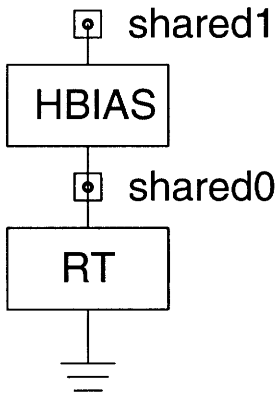

Figure 11: A block diagram of the circuitry contained in `ret.rel.h`. The diagram shows the retina input circuitry ("RT") and the HRES biasing circuitry ("HBIAS"). Note that the nodes `shared0` and `shared1` are common to all three grid circuit files and must be listed in the `common` statement in the ".h" file.

`ret.rel.e`

```
global 0

; nmos model
model modn nmos3 \
vto=0.7 tox=2.3e-8 nsub=1.0e15 uo=700 cgso=3.5e-10 \
cgdo=3.5e-10 cj=8.0e-5 mj=0.5 cjsw=5.0e-10 mjsw=0.5 \
xj=0.6u ld=0.3u \
nfs=1e10

; pmos model
model modp pmos3 \
vto=-0.7 tox=2.3e-8 nsub=1.0e16 uo=350 cgso=3.5e-10 \
cgdo=3.5e-10 cj=2.0e-4 mj=0.5 cjsw=1.5e-9 mjsw=0.5 \
xj=0.6u ld=0.3u \
nfs=1e10

define hres ( d1 d2 g1 g2 )
parameters w=2u l=8u
m1 d1 g1 2 2 modn w=w l=l
m2 d2 g2 2 2 modn w=w l=l
end hres

hres0 shared0 pseudo0 shared1 pseudo1 hres
c0 pseudo0 shared0 connector
c1 pseudo1 shared1 connector
```

ret.rel.s

```
global 0

; nmos model
model modn nmos3 \
vto=0.7 tox=2.3e-8 nsub=1.0e15 uo=700 cgso=3.5e-10 \
cgdo=3.5e-10 cj=8.0e-5 mj=0.5 cjsw=5.0e-10 mjsw=0.5 \
xj=0.6u ld=0.3u \
nfs=1e10

; pmos model
model modp pmos3 \
vto=-0.7 tox=2.3e-8 nsub=1.0e16 uo=350 cgso=3.5e-10 \
cgdo=3.5e-10 cj=2.0e-4 mj=0.5 cjsw=1.5e-9 mjsw=0.5 \
xj=0.6u ld=0.3u \
nfs=1e10

define hres ( d1 d2 g1 g2 )
parameters w=2u l=8u
m1 d1 g1 2 2 modn w=w l=l
m2 d2 g2 2 2 modn w=w l=l
end hres

hres0 shared0 pseudo0 shared1 pseudo1 hres
c0 pseudo0 shared0 connector
c1 pseudo1 shared1 connector
```

ret.rel.h

```
global 0

; common shared0 shared1

; nmos model
model modn nmos3 \
vto=0.7 tox=2.3e-8 nsub=1.0e15 uo=700 cgso=3.5e-10 \
cgdo=3.5e-10 cj=8.0e-5 mj=0.5 cjsw=5.0e-10 mjsw=0.5 \
xj=0.6u ld=0.3u nfs=1e10

; pmos model
model modp pmos3 \
vto=-0.7 tox=2.3e-8 nsub=1.0e16 uo=350 cgso=3.5e-10 \
cgdo=3.5e-10 cj=2.0e-4 mj=0.5 cjsw=1.5e-9 mjsw=0.5 \
xj=0.6u ld=0.3u nfs=1e10

;************************************
; Simple Transconductor

; transamp connections:
; 2 = v+, 4 = v-, 14 = vb, 17 = iout

;*************************************
define transamp ( 2 4 17 )
parameters ib=25n

; bias currents
ibias 1 14 i i=ib
m1 14 14 0 0 modn w=10u l=8u

; bias voltages
vdd 1 0 dc v=5
; devices
m2 65 14 0 0 modn w=10u l=8u
m3 16 2 65 65 modn w=4u l=6u
m4 17 4 65 65 modn w=4u l=6u
m5 16 16 1 1 modp w=8u l=6u
m6 17 16 1 1 modp w=8u l=6u
end transamp
```

ret.rel.h (continued)

```
;*************************************
; HRES bias circuitry
;**************************************
define hbias ( vnode vg )
parameters ib=25n
vdd vd 0 dc v=5
ibias vd vb i i=ib
mm vb vb 0 0 modn w=10u l=8u

mb 1 vb 0 0 modn w=10u l=8u
m1 2 vnode 1 1 modn w=4u l=6u
m2 4 4 1 1 modn w=4u l=6u
m3 2 2 vd vd modp w=8u l=6u
m4 vg 2 vd vd modp w=8u l=6u
md vg vg 4 4 modn w=4u l=6u
end hbias

;*************************************
; Retina Input Circuitry
;**************************************
define rt ( a b )
parameters vlo=0.8 vhi=0.8 c=1.e-12 vb=2.5
t0 vin a vout transamp ib=25n
t1 vin a a transamp ib=25n
c0 a b c c=c
v0 vt b dc v=vb
r0 vt vout r r=1e8

; Input image is loaded here
i0 vin b pwl v0 = 1.23e45 t0 = 0 end rt

;*************************************
; Connections to the Grid
;**************************************
rt0 shared0 0 rt
hbias0 shared0 shared1 hbias

plot shared0 shared1
cmvsim options dodc=0 dotran=1 cmin=1.e-15 stop=1.e-3
```