DEVSIM

Version: Beta 0.01

User Guide

Contact

Web:
http://www.devsim.com

Email:
info@devsim.com

Open Source Project:
http://www.devsim.org

Copyright

Disclaimer

Trademark

Contents

List of Figures

List of Tables

Part I

User Guide

Chapter 1

Release Notes

1.1 Introduction

DEVSIM download and installation instructions are located in *Chapter 11, Installation on page 73*. The following sections list bug fixes and enhancements over time. The official website for this project is located at http://www.devsim.org.

1.2 December 22, 2013

1.2.1 Binary Availability

Binary versions of the DEVSIM software are available for download from http://sourceforge.net/projects/devsim. Current versions available are for

- Mac OS X 10.9 (Mavericks)
- Red Hat Enterprise Linux 6.5
- Ubuntu 12.04 (LTS)

Please see *Chapter 11, Installation on page 73* for more information.

1.2.2 Platforms

Mac OS X 10.9 (Mavericks) is now supported. Support for 32 bit is no longer supported on this platform, since the operating system is only released as 64 bit.

Regression data will no longer be maintained in the source code repository for 32 bit versions of Ubuntu 12.04 (LTS) and Red Hat Enterprise Linux 6.5. Building and running on these platforms will still be supported.

1.2.3 Source code improvements

The source code has been improved to compile on Mac OS X 10.9 (Mavericks) and to comply with C++11 language standards. Some of the structure of the project has been reorganized. These changes to the infrastructure will help to keep the program maintainable and useable into the future.

1.3 September 8, 2013

1.3.1 Convergence

If the simulation is diverging for 5 or more iterations, the simulation stops.

1.3.2 Bernoulli Function Derivative Evaluation

The `dBdx` math function has been improved to reduce overflow.

1.3.3 Default Edge Model

The `edge_index` is now a default edge models created on a region (Table 3.2, *Edge models defined on each region of a device.* on page 15).

1.4 August 14, 2013

1.4.1 SYMDIFF functions

The `vec_max` and `vec_min` functions have been added to the SYMDIFF parser (Table 9.2, *Predefined Functions.* on page 65). The `vec_sum` function replaces `sum`.

1.4.2 Default Node Models

The `coordinate_index` and `node_index` are now part of the default node models created on a region (Table 3.1, *Node models defined on each region of a device.* on page 15).

1.4.3 Set Node Value

It is now possible to use the `set_node_value` command (page 33) to set a uniform value or indexed value on a node model.

1.4.4 Fix Edge Average Model

Fixed issue with `edge_average_model` command (page 26) during serialization to the DEVSIM format.

1.5 July 29, 2013

1.5.1 DEVSIM is open source

DEVSIM is now an open source project and is available from http://www.github.com/devsim/devsim. License information may be found in *Section 12.1,* DEVSIM License *on page 75*. If you would like to participate in this project or need support, please contact us using the information in the front cover of this manual. Installation instructions may be found in *Chapter 11,* Installation *on page 73*.

1.5.2 Build

The Tcl interpreter version of DEVSIM is now called devsim_tcl, and is located in `/src/main/` of the build directory. Please see the `INSTALL` file for more information.

1.5.3 Contact Material

Contacts now require a material setting (e.g. `metal`). This is for informational purposes. Contact models still look up parameter values based on the region they are located.

1.5.4 External Meshing

Please see *Section 6.3,* Using an external mesher *on page 47* for more information about importing meshes from other tools.

Genius Mesh Import DEVSIM can now read meshes written from Genius Device Simulator. More information about Genius is in *Section 6.3.1,* Genius *on page 47*.

Gmsh Mesh Import DEVSIM reads version 2.1 and 2.2 meshes from Gmsh. Version 2.0 is no longer supported. Please see *Section 6.3.2,* Gmsh *on page 48* for more information.

1.5.5 Math Functions

The `acosh`, `asinh`, `atanh`, are now available math functions. Please see Table 9.2, *Predefined Functions.* on page 65.

1.5.6 Test directory structure

Platform specific results are stored in a hierarchical fashion.

Chapter 2

Introduction

2.1 Overview

DEVSIM is a technology computer-aided design (TCAD) software for semiconductor device simulation. While geared toward this application, it may be used where the control volume approach is appropriate for solving systems of partial-differential equations (PDE's) on a static mesh. After introducing DEVSIM, the rest of the manual discusses the key components of the system, and instructions for their use.

> *DEVSIM is available from* http://www.devsim.org. *The source code is available under the terms of the GNU Lesser General Public License Version 3 [1]. Examples are released under the Apache License Version 2.0 [2]. Contributions to this project are welcome in the form of bug reporting, documentation, modeling, and feature implementation.*

2.2 Goals

The primary goal of DEVSIM is to give the user as much flexibility and control as possible. In this regard, few models are coded into the program binary. They are implemented in human-readable scripts that can be modified if necessary.

DEVSIM is embedded within a scripting language interface (*Chapter 8,* User Interface *on page 59*). This provides control structures and language syntax in a consistent and intuitive manner. Taking a hierarchical approach, the user is provided an environment where they can implement new models on their own. This is without requiring extensive vendor support or use of compiled programming languages.

SYMDIFF (*Chapter 9,* SYMDIFF *on page 63*) is the symbolic expression parser used to allow the formulation of device equations in terms of models and parameters. Using symbolic differentiation, the required partial derivatives can be generated, or provided by the user. DEVSIM then assembles these equations over the mesh.

2.3 Structures

Devices A device refers to a discrete structure being simulated. It is composed of the following types of objects.

Regions A region defines a portion of the device of a specific material. Each region has its own system of equations being solved.

Interfaces Interfaces connect two regions together. At the interfaces, equations are specified to account for how the flux in each device region crosses the region boundary.

Contacts Contacts specify the boundary conditions required for device simulation. It also specifies how terminal currents are are integrated into an external circuit.

2.4 Equation assembly

Equation assembly of models is discussed in *Chapter 3,* Equation and Models *on page 11*.

2.5 Parameters

Parameters may be specified globally, or for a specific device or region. Alternatively, parameters may be based on the material type of the regions. Usage is discussed in *Chapter 4,* Model Parameters *on page 37*.

2.6 Circuits

Circuit boundary conditions allow multi-device simulation. They are also required for setting sources and their response for AC and noise analysis. Circuit elements, such as voltage sources, current sources, resistors, capacitors, and inductors may be specified. This is further discussed in *Chapter 5,* Circuits *on page 41*.

2.7 Meshing

Meshing is discussed in *Chapter 6,* Meshing *on page 45*.

2.8 Analysis

DEVSIM offers a range of simulation algorithms. They are discussed in more detail in *Chapter 7,* Solver *on page 55*.

DC The DC operating point analysis is useful for performing steady-state simulation for a different bias conditions.

AC At each DC operating point, a small-signal AC analysis may be performed. An AC source is provided through a circuit and the response is then simulated. This is useful for both quasi-static capacitance simulation, as well as RF simulation.

Noise/Sensitivity Noise analysis may be used to evaluate how internal noise sources are observed in the terminal currents of the device or circuit. Using this method, it is also possible to simulate how the device response changes when device parameters are changed.

Transient DEVSIM is able to simulate the nonlinear transient behavior of devices, when the bias conditions change with time.

2.9 Scripting interface

The scripting interface to DEVSIM is discussed in *Chapter 8,* User Interface *on page 59*.

2.10 Expression parser

The expression parser is discussed in *Chapter 9,* SYMDIFF *on page 63*.

2.11 Visualization and postprocessing

Visualization is discussed in *Chapter 10,* Visualization *on page 71*.

2.12 Installation

Installation is discussed in *Chapter 11,* Installation *on page 73*.

2.13 Additional information

Additional information is discussed in *Chapter 12,* Additional Information *on page 75*.

2.14 Examples

Examples are discused in the remaining chapters beginning with *Chapter 13,* Example Overview *on page 81*.

Chapter 3

Equation and Models

3.1 Overview

DEVSIM uses the control volume approach for assembling partial-differential equations (PDE's) on the simulation mesh. DEVSIM is used to solve equations of the form:

$$\frac{\partial X}{\partial t} + \nabla \cdot \vec{Y} + Z = 0 \tag{3.1}$$

Internally, it transforms the PDE's into an integral form.

$$\int \frac{\partial X}{\partial t} \partial r + \int \vec{Y} \cdot \partial \vec{s} + \int Z \partial r = 0 \tag{3.2}$$

Equations involving the divergence operators are converted into surface integrals, while other components are integrated over the device volume.

In Figure 3.1, 2D mesh elements are depicted. The shaded area around the center node is referred to as the node volume, and it is used for the volume integration. The lines from the center node to other nodes are referred to as edges. The flux through the edge are integrated with respect to the perpendicular bisectors (dashed lines) crossing each triangle edge.

In this form, we refer to a model integrated over the edges of triangles as edge models. Models integrated over the volume of each triangle vertex are referred to as node models. Element edge models are a special case where variables at other nodes off the edge may cause the flux to change.

There are a default set of models created in each region upon initialization of a device, and are typically based on the geometrical attributes. These are described in the following sections. Models required for describing the device behavior are created using the equation parser described in *Chapter 9,* SYMDIFF *on page 63*. For special situations, custom matrix assembly is also available and is discussed in *Section 3.5,* Custom matrix assembly *on page 19*.

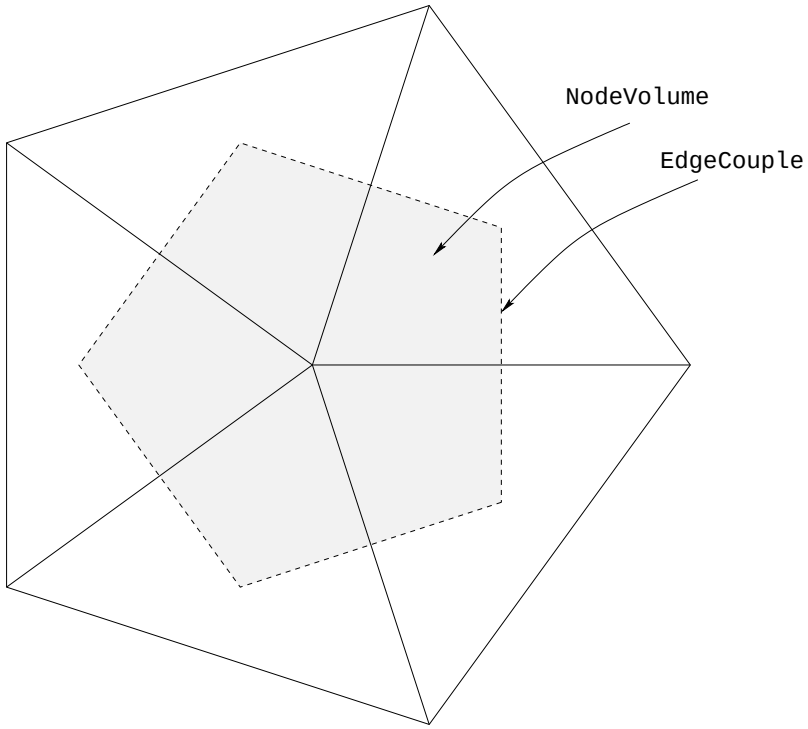

Figure 3.1: Mesh elements in 2D.

3.2 Bulk models

3.2.1 Node models

Node models may be specified in terms of other node models, mathematical functions, and parameters on the device. The simplest model is the node solution, and it represents the solution variables being solved for. Node models automatically created for a region are listed in Table 3.1.

In this example, we present an implementation of Shockley Read Hall recombination [3].

```
USRH="-ElectronCharge*(Electrons*Holes - n_i^2)/(taup*(Electrons + n1) \
        + taun*(Holes + p1))")
dUSRHdn="simplify(diff(%s, Electrons))" % USRH
dUSRHdp="simplify(diff(%s, Holes))" % USRH
ds.node_model(device='MyDevice', region='MyRegion', name="USRH", equation=USRH)
ds.node_model(device='MyDevice', region='MyRegion', name="USRH:Electrons", equation=dUSRHdn)
ds.node_model(device='MyDevice', region='MyRegion', name="USRH:Holes", equation=dUSRHdp)
```

The first model specified, `USRH`, is the recombination model itself. The derivatives with respect to electrons and holes are `USRH:Electrons` and `USRH:Holes`, respectively. In this particular example `Electrons` and `Holes` have already been defined as solution variables. The remaining variables in the equation have already been specified as parameters.

The `diff` function tells the equation parser to take the derivative of the original expression, with respect to the variable specified as the second argument. During equation assembly, these

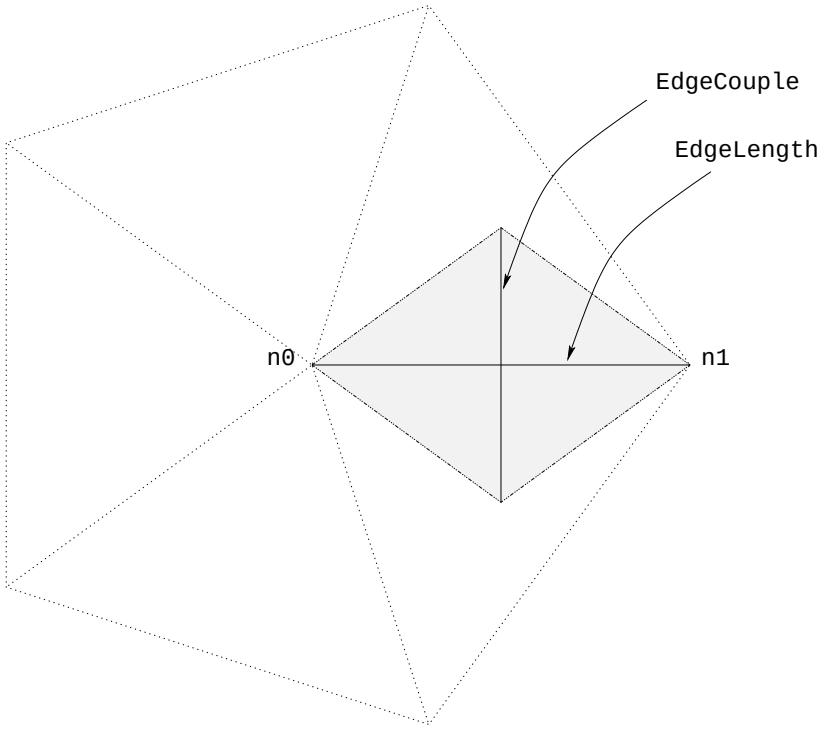

Figure 3.2: Edge model constructs in 2D.

derivatives are required in order to converge upon a solution. The `simplify` function tells the expression parser to attempt to simplify the expression as much as possible.

3.2.2 Edge models

Edge models may be specified in terms of other edge models, mathematical functions, and parameters on the device. In addition, edge models may reference node models defined on the ends of the edge. As depicted in Figure 3.2, edge models are with respect to the two nodes on the edge, `n0` and `n1`.

For example, to calculate the electric field on the edges in the region, the following scheme is employed:

```
ds.edge_model(device="device", region="region", name="ElectricField",
           equation="(Potential@n0 - Potential@n1)*EdgeInverseLength")
ds.edge_model(device="device", region="region",
           name="ElectricField:Potential@n0", equation="EdgeInverseLength")
ds.edge_model(device="device", region="region",
           name="ElectricField:Potential@n1", equation="-EdgeInverseLength")
```

In this example, `EdgeInverseLength` is a built-in model for the inverse length between nodes on an edge. `Potential@n0` and `Potential@n1` is the `Potential` node solution on the nodes at the end of the edge. These edge quantities are created using the `edge_from_node_model` command (page 26). In addition, the `edge_average_model` command (page 26) can be used to create edge models in terms of node model quantities.

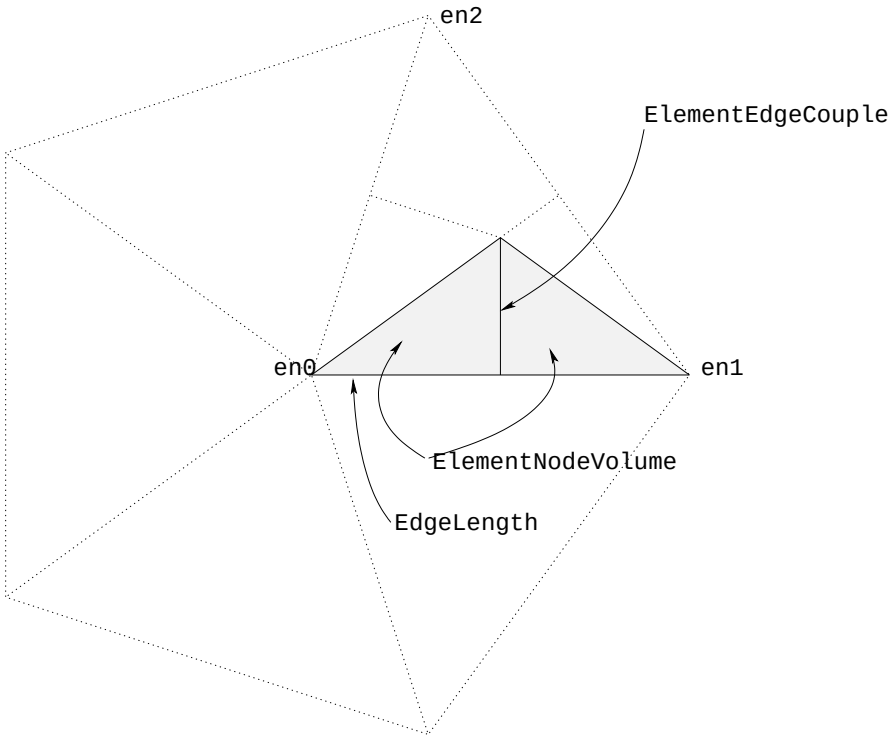

Figure 3.3: Element edge model constructs in 2D.

Edge models automatically created for a region are listed in Table 3.2.

3.2.3 Element edge models

Element edge models are used when the edge quantitites cannot be specified entirely in terms of the quantities on both nodes of the edge, such as when the carrier mobility is dependent on the normal electric field. In 2D, element edge models are evaluated on each triangle edge. As depicted in Figure 3.3, edge models are with respect to the three nodes on each triangle edge and are denoted as `en0`, `en1`, and `en2`. Derivatives are with respect to each node on the triangle.

In 3D, element edge models are evaluated on each tetrahedron edge. Derivatives are with respect to the nodes on both triangles on the tetrahedron edge. Element edge models automatically created for a region are listed in Table 3.3.

As an alternative to treating integrating the element edge model with respect to `ElementEdgeCouple`, the integration may be performed with respect to `ElementNodeVolume`. See `equation` command (page 22) for more information.

3.2.4 Model derivatives

To converge upon the solution, derivatives are required with respect to each of the solution variables in the system. DEVSIM will look for the required derivatives. For a model `model`, the derivatives with respect to solution variable `variable` are presented in Table 3.4.

Node Model	Description
AtContactNode	Evaluates to 1 if node is a contact node, otherwise 0
NodeVolume	The volume of the node. Used for volume integration of node models on nodes in mesh
NSurfaceNormal_x	The surface normal to points on the interface or contact (2D and 3D)
NSurfaceNormal_y	The surface normal to points on the interface or contact (2D and 3D)
NSurfaceNormal_z	The surface normal to points on the interface or contact (3D)
SurfaceArea	The surface area of a node on interface and contact nodes, otherwise 0
coordinate_index	Coordinate index of the node on the device
node_index	Index of the node in the region
x	x position of the node
y	y position of the node
z	z position of the node

Table 3.1: Node models defined on each region of a device.

Edge Model	Description
EdgeCouple	The length of the perpendicular bisector of an element edge. Used to perform surface integration of edge models on edges in mesh.
EdgeInverseLength	Inverse of the EdgeLength.
EdgeLength	The distance between the two nodes of an edge
edge_index	Index of the edge on the region
unitx	x component of the unit vector along an edge
unity	y component of the unit vector along an edge (2D and 3D)
unitz	z component of the unit vector along an edge (3D only)

Table 3.2: Edge models defined on each region of a device.

3.2.5 Conversions between model types

The edge_from_node_model command (page 26) is used to create edge models referring to the nodes connecting the edge. For example, the edge models Potential@n0 and Potential@n1 refer to the Potential node model on each end of the edge.

The edge_average_model command (page 26) creates an edge model which is either the arithmetic mean, geometric mean, gradient, or negative of the gradient of the node model on each edge.

When an edge model is referred to in an element edge model expression, the edge values are implicity converted into element edge values during expression evaluation. In addition, derivatives of the edge model with respect to the nodes of an element edge are required, they are converted as well. For example, edgemodel:variable@n0 and edgemodel:variable@n1 are implicitly converted to edgemodel:variable@en0 and edgemodel:variable@en1, respectively.

The element_from_edge_model command (page 28) is used to create directional components of an edge model over an entire element. The derivative option is used with this command to create the derivatives with respect to a specific node model. The element_from_node_model command (page 29) is used to create element edge models referring to each node on the element of the element edge.

Element Edge Model	Description
ElementEdgeCouple	The length of the perpendicular bisector of an edge. Used to perform surface integration of element edge model on element edge in the mesh.
ElementNodeVolume	The node volume at either end of each element edge.

Table 3.3: Element edge models defined on each region of a device.

Model Type	Derivatives Required
Node Model	model:variable
Edge Model	model:variable@n0 model:variable@n1
Element Edge Model	model:variable@en0 model:variable@en1 model:variable@en2 model:variable@en3 (3D)

Table 3.4: Required derivatives for equation assembly. model is the name of the model being evaluated, and variable is one of the solution variables being solved at each node.

3.2.6 Equation assembly

Bulk equations are specified in terms of the node, edge, and element edge models using the equation command (page 22). Node models are integrated with respect to the node volume. Edge models are integrated with the perpendicular bisectors along the edge onto the nodes on either end.

Element edge models are treated as flux terms and are integrated with respect to ElementEdgeCouple using the element_model option. Alternatively, they may be treated as source terms and are integrated with respect to ElementNodeVolume using the volume_model option.

In this example, we are specifying the Potential Equation in the region to consist of a flux term named PotentialEdgeFlux and to not have any node volume terms.

```
ds.equation(device="device", region="region", name="PotentialEquation",
        variable_name="Potential", edge_model="PotentialEdgeFlux", variable_update="log_damp" )
```

In addition, the solution variable coupled with this equation is Potential and it will be updated using logarithmic damping.

3.3 Interface

3.3.1 Interface models

Figure 3.4 depicts an interface in DEVSIM. It is a collection of overlapping nodes existing in two regions, r0 and r1.

Interface models are node models specific to the interface being considered. They are unique from bulk node models, in the sense that they may refer to node models on both sides of the interface. They are specified using the interface_model command (page 31). Interface models may

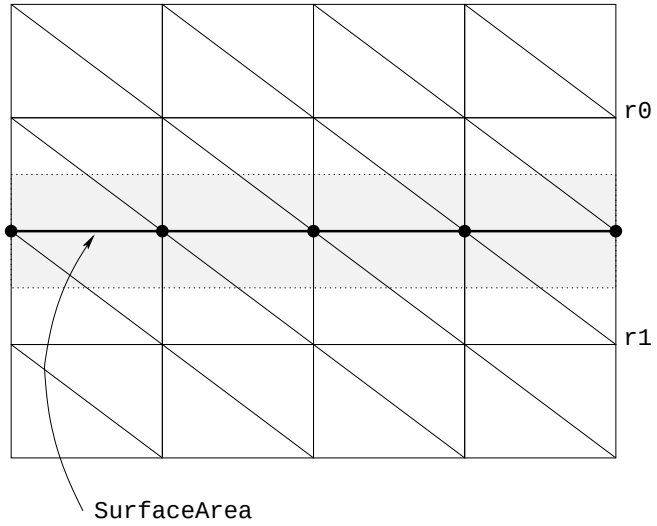

Figure 3.4: Interface constructs in 2D. Interface node pairs are located at each •. The `SurfaceArea` model is used to integrate flux term models.

Model Type	Model Name	Derivatives Required
Node Model (region 0)	`nodemodel@r0`	`nodemodel:variable@r0`
Node Model (region 1)	`nodemodel@r1`	`nodemodel:variable@r1`
Interface Node Model	`inodemodel`	`inodemodel:variable@r0` `inodemodel:variable@r1`

Table 3.5: Required derivatives for interface equation assembly. The node model name `nodemodel` and its derivatives `nodemodel:variable` are suffixed with `@r0` and `@r1` to denote which region on the interface is being referred to.

refer to node models or parameters on either side of the interface using the syntax `nodemodel@r0` and `nodemodel@r1` to refer to the node model in the first and second regions of the interface. The naming convention for node models, interface node models, and their derivatives are shown in Table 3.5.

```
ds.interface_model(device="device", interface="interface",
                name="continuousPotential", equation="Potential@r0-Potential@r1")
```

3.3.2 Interface model derivatives

For a given interface model, `model`, the derivatives with respect to the variable `variable` in the regions are

- `model:variable@r0`
- `model:variable@r1`

```
ds.interface_model(device="device", interface="interface",
```


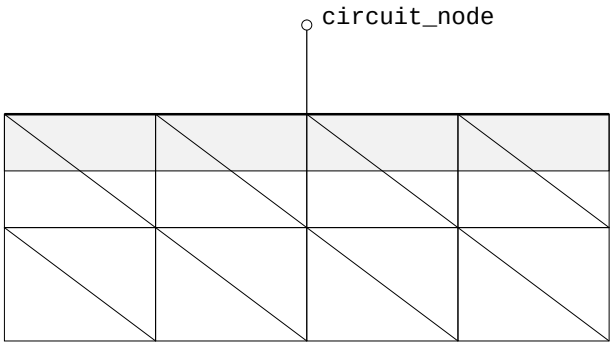

Figure 3.5: Contact constructs in 2D.

```
                name="continuousPotential:Potential@r0", equation="1")
ds.interface_model(device="device", interface="interface",
                name="continuousPotential:Potential@r1", equation="-1")
```

3.3.3 Interface equation assembly

There are two types of interface equations considered in DEVSIM. They are both activated using the `interface_equation` command (page 23).

In the first form, `continuous`, the equations for the nodes on both sides of the interface are integrated with respect to their volumes and added into the same equation. An additional equation is then specified to relate the variables on both sides. In this example, continuity in the potential solution across the interface is enforced, using the `continuousPotential` model defined in the previous section.

```
ds.interface_equation(device="device", interface="interface", name="PotentialEquation",
                variable_name="Potential", interface_model="continuousPotential",
                type="continuous")
```

In the second form, `fluxterm`, a flux term is integrated over the surface area of the interface and added to the first region, and subtracted from the second.

3.4 Contact

3.4.1 Contact models

Figure 3.5 depicts how a contact is treated in a simulation. It is a collection of nodes on a region. During assembly, the specified models form an equation, which replaces the equation applied to these nodes for a bulk node.

Contact models are equivalent to node and edge models, and are specified using the `contact_node_model` command (page 23) and the `contact_edge_model` command (page 23), respectively. The key difference is that the models are only evaluated on the contact nodes for the contact specified.

3.4.2 Contact model derivatives

The derivatives are equivalent to the discussion in *Section 3.2.4, Model derivatives on page 14*. If external circuit boundary conditions are being used, the model `model` derivative with respect to the circuit node `node` name should be specified as `model:node`.

3.4.3 Contact equation assembly

The `contact_equation` command (page 21) is used to specify the boundary conditions on the contact nodes. The models specified replace the models specified for bulk equations of the same name. For example, the node model specified for the contact equation is assembled on the contact nodes, instead of the node model specified for the bulk equation. Contact equation models not specified are not assembled, even if the model exists on the bulk equation for the region attached to the contact.

As an example

```
ds.contact_equation(device="device", contact="contact", name="PotentialEquation",
                variable_name="Potential", node_model="contact_bc",
                edge_charge_model="DField")
```

Current models refer to the instantaneous current flowing into the device. Charge models refer to the instantaneous charge at the contact.

During a transient, small-signal or ac simulation, the time derivative is taken so that the net current into a circuit node is

$$I(t) = i(t) + \frac{\partial q(t)}{\partial t} \tag{3.3}$$

where i is the integrated current and q is the integrated charge.

3.5 Custom matrix assembly

The `custom_equation` command (page 21) command is used to register callbacks to be called during matrix and right hand side assembly. The Python procedure must expect to receive two arguments and return two lists. For example a procedure named `myassemble` registered with

```
ds.custom_equation(name="test1", procedure="myassemble")
```

must expect to receive two arguments

```
def myassemble(what, timemode):
  .
  .
  .
  return [rcv, rv]
```

where `what` may be passed as one of

```
MATRIXONLY
RHS
MATRIXANDRHS
```

and `timemode` may be passed as one of

```
DC
TIME
```

When `timemode` is `DC`, the time-independent part of the equation is returned. When `timemode` is `TIME`, the time-derivative part of the equation is returned. The simulator will scale the time-derivative terms with the proper frequency or time scale.

The return value from the procedure must return two lists of the form

```
[1 1 1.0 2 2 1.0 1 2 -1.0 2 1 -1.0 2 2 1.0] [1 1.0 2 1.0 2 -1.0]
```

where the length of the first list is divisible by 3 and contains the row, column, and value to be assembled into the matrix. The second list is divisible by 2 and contains the right-hand side entries. Either list may be empty.

The `get_circuit_equation_number` command (page 42) may be used to get the equation numbers corresponding to circuit node names. The `get_equation_numbers` command (page 22) may be used to find the equation number corresponding to each node index in a region.

The matrix and right hand side entries should be scaled by the `NodeVolume` if they are assembled into locations in a device region. Row permutations, required for contact and interface boundary conditions, are automatically applied to the row numbers returned by the Python procedure.

3.6 Cylindrical Coordinate Systems

In 2D, models representing the edge couples, surface areas and node volumes may be generated using the following commands:

- `cylindrical_edge_couple` command (page 24)
- `cylindrical_node_volume` command (page 24)
- `cylindrical_surface_area` command (page 25)

In order to change the integration from the default models to cylindrical models, the following parameters may be set

```
set_parameter(name="node_volume_model",          value="CylindricalNodeVolume")
set_parameter(name="edge_couple_model",          value="CylindricalEdgeCouple")
set_parameter(name="element_edge_couple_model",  value="ElementCylindricalEdgeCouple")
set_parameter(name="element_node0_volume_model", value="ElementCylindricalNodeVolume@en0")
set_parameter(name="element_node1_volume_model", value="ElementCylindricalNodeVolume@en1")
```

3.7 Equation commands

contact_equation

```
ds.contact_equation(device=STRING, contact=STRING, name=STRING,
                    variable_name=STRING,
                    circuit_node=STRING,
                    edge_charge_model=STRING,
                    edge_current_model=STRING,
                    edge_model=STRING,
                    element_charge_model=STRING,
                    element_current_model=STRING
                    element_model=STRING,
                    node_charge_model=STRING,
                    node_current_model=STRING,
                    node_model=STRING)
```

device=	Device on which to apply this command
contact=	Contact on which to apply this command
name=	Name of the contact equation being created
variable_name=	The variable name is used to determine the bulk equation we are replacing at this contact
circuit_node=	Name of the circuit we integrate the flux into
edge_charge_model=	Name of the edge model used to determine the charge at this contact
edge_current_model=	Name of the edge model used to determine the current flowing out of this contact
edge_model=	Name of the edge model being integrated at each edge at this contact
element_charge_model=	Name of the element edge model used to determine the charge at this contact
element_current_model=	Name of the element edge model used to determine the current flowing out of this contact
element_model=	Name of the element edge model being integrated at each edge at this contact
node_charge_model=	Name of the node model used to determine the charge at this contact
node_current_model=	Name of the node model used to determine the current flowing out of this contact
node_model=	Name of the node_model being integrated at each node at this contact

custom_equation

```
ds.custom_equation(name=STRING, procedure=STRING)
```

name=	Name of the custom equation being created
procedure=	The Python procedure to be called. See *Section 3.5,* Custom matrix assembly *on page 19* for a description of how the function should be structured.

equation

```
ds.equation(device=STRING, region=STRING, name=STRING,
         variable_name=STRING, node_model=STRING,
         edge_model=STRING, time_node_model=STRING,
         element_model=STRING, volume_model=STRING,
         variable_update=OPTION)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the equation being created
variable_name=	Name of the node_solution being solved
node_model=	Name of the node_model being integrated at each node in the device volume
edge_model=	Name of the edge model being integrated over each edge in the device volume
time_node_model=	Name of the time dependent node_model being integrated at each node in the device volume
element_model=	Name of the element_model being integrated over each edge in the device volume
volume_model=	Name of the element_model being integrated over the volume of each edge in the device volume
variable_update=	update type for circuit variable
"default"	Variable can be positive or negative
"log_damp"	Variable update is damped
"positive"	Solution update results in positive quantity

get_equation_numbers

```
ds.get_equation_numbers(device=STRING, region=STRING, equation=STRING,
                     variable=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
equation=	Name of the equation (optional)
variable=	Name of the variable (optional)

Returns a list of the equation numbers corresponding to each node in a region. Values are only valid when during the course of a solve.

interface_equation

```
ds.interface_equation(device=STRING, interface=STRING, name=STRING,
                  variable_name=STRING,
                  interface_model=STRING,
                  type=OPTION)
```

device=	Device on which to apply this command
interface=	Interface on which apply this command
name=	Name of the interface equation being created
variable_name=	The variable name is used to determine the bulk equation we are coupling this interface to
interface_node_model=	When specified, the bulk equations on both sides of the interface are integrated together. This model is then used to specify how nodal quantities on both sides of the interface are balanced
type=	Specifies the type of boundary condition
"continuous"	Equations of the same name in the two regions are added. The interface_model is an additional equation is created to specify how quantities across the interface are solved
"fluxterm"	The interface_model is added to the bulk equation in the first region, and subtracted from the second

3.8 Model commands

contact_edge_model

```
ds.contact_edge_model(device=STRING, contact=STRING, name=STRING,
                  equation=STRING, display_type=OPTION)
```

device=	Device on which to apply this command
contact=	Contact on which to apply this command
name=	Name of the contact edge model being created
equation=	Equation used to describe the contact edge model being created
display_type=	Option for output display in graphical viewer
"nodisplay"	Data on edge will not be displayed
"scalar"	Data on edge is a scalar quantity
"vector"	Data on edge is a vector quantity (default)

contact_node_model

```
ds.contact_node_model(device=STRING, contact=STRING, name=STRING,
                  equation=STRING, display_type=OPTION)
```

device=	Device on which to apply this command
contact=	Contact on which to apply this command
name=	Name of the contact node model being created
equation=	Equation used to describe the contact node model being created
display_type=	Option for output display in graphical viewer
"nodisplay"	Data on node will not be displayed
"scalar"	Data on node is a scalar quantity (default)

cylindrical_edge_couple

```
ds.cylindrical_edge_couple(device=STRING, region=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command

This model is only available in 2D. The created variables are

- ElementCylindricalEdgeCouple (Element Edge Model)
- CylindricalEdgeCouple (Edge Model)

The set_parameter command (page 38) must be used to set

- raxis_variable, the variable (x or y) which is the radial axis variable in the cylindrical coordinate system
- raxis_zero, the location of the z axis for the radial axis variable

cylindrical_node_volume

```
ds.cylindrical_node_volume(device=STRING, region=STRING,
                zaxis_variable=STRING, raxis_zero=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command

This model is only available in 2D. The created variables are

- ElementCylindricalNodeVolume@en0 (Element Edge Model)
- ElementCylindricalNodeVolume@en1 (Element Edge Model)
- CylindricalNodeVolume (Edge Model)

The ElementCylindricalNodeVolume@en0 and ElementCylindricalNodeVolume@en1 represent the node volume at each end of the element edge.

The set_parameter command (page 38) must be used to set

- raxis_variable, the variable (x or y) which is the radial axis variable in the cylindrical coordinate system
- raxis_zero, the location of the z axis for the radial axis variable

cylindrical_surface_area

```
ds.cylindrical_surface_area(device=STRING, region=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command

This model is only available in 2D. The created variables are

- CylindricalSurfaceArea (Node Model)

and is the cylindrical surface area along each contact and interface node in the device region.

The set_parameter command (page 38) must be used to set

- raxis_variable, the variable (x or y) which is the radial axis variable in the cylindrical coordinate system
- raxis_zero, the location of the z axis for the radial axis variable

delete_edge_model

```
ds.delete_edge_model(device=STRING, region=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the edge model being deleted

delete_interface_model

```
ds.delete_edge_model(device=STRING, interface=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
interface=	Interface on which apply this command
name=	Name of the interface model being deleted

delete_node_model

```
ds.delete_node_model(device=STRING, region=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the node model being deleted

edge_average_model

```
ds.edge_average_model(device=STRING, region=STRING,
                      node_model=STRING, edge_model=STRING,
                      average_type=STRING, derivative=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
node_model=	The node model from which we are creating the edge model. If derivative is specified, the edge model is created from nodeModel:derivativeModel
edge_model=	The edge model name being created. If derivative is specified, the edge models created are edgeModel:derivativeModel@n0 edgeModel:derivativeModel@n1, which are the derivatives with respect to the derivative model on each side of the edge
derivative=	The node model of the variable for which the derivative is being taken. The node model nodeModel:derivativeModel is used to create the resulting edge models.
average_type=	The node models on both sides of the edge are averaged together to create one of the following types of averages.
"arithmetic"	The edge model is the average of the node model on both sides (default)
"geometric"	The edge model is the square root of the product of the node model evaluated on each side
"gradient"	The edge model is the gradient along the edge with respect to the distance between the two nodes.
"negative_gradient"	The edge model is the negative of the gradient along the edge

For a node model, creates an 2 edge models referring to the node model value at both ends of the edge. For example, to calculate electric field:

```
ds.edge_average_model(device=device, region=region, node_model="Potential",
                      edge_model="ElecticField", average_type="negative_gradient")
```

and the derivatives ElectricField:Potential@n0 and ElectricField:Potential@n1 are then created from

```
ds.edge_average_model(device=device, region=region, node_model="Potential",
                      edge_model="ElecticField", average_type="negative_gradient",
                      derivative="Potential")
```

edge_from_node_model

```
ds.edge_from_node_model(device=STRING, region=STRING,
                        node_model=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
node_model=	The node model from which we are creating the edge model

For a node model, creates an 2 edge models referring to the node model value at both ends of the edge. For example, to calculate electric field:

```
ds.edge_from_node_model(device=device, region=region, node_model="Potential")
```

edge_model

```
ds.edge_model(device=STRING, region=STRING, name=STRING,
           equation=STRING, display_type=OPTION)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the edge model being created
equation=	Equation used to describe the edge model being created
display_type=	Option for output display in graphical viewer
"nodisplay"	Data on edge will not be displayed
"scalar"	Data on edge is a scalar quantity (default)
"vector"	Data on edge is a vector quantity (deprecated)

The vector option uses an averaging scheme for the edge values projected in the direction of each edge. For a given model, model, the generated components in the visualization files is:

- model_x_onNode
- model_y_onNode
- model_z_onNode (3D)

This averaging scheme does not produce accurate results, and it is recommended to use the element_from_edge_model command (page 28) to create components better suited for visualization. See *Chapter 10, Visualization on page 71* for more information about creating data files for external visualization programs.

element_model

```
ds.element_model(device=STRING, region=STRING, name=STRING,
           equation=STRING, display_type=OPTION)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the element edge model being created
equation=	Equation used to describe the element edge model being created
display_type=	Option for output display in graphical viewer
"nodisplay"	Data on edge will not be displayed
"scalar"	Data on edge is a scalar quantity (default)

element_from_edge_model

```
ds.element_from_edge_model(device=STRING, region=STRING,
                     edge_model=STRING,
                     derivative=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
edge_model=	The edge model from which we are creating the element model
derivative=	The variable we are taking with respect to edge_model

For an edge model `emodel`, creates an element models referring to the directional components on each edge of the element:

- `emodel_x`
- `emodel_y`

If the `derivative variable` option is specified, the `emodel@n0` and `emodel@n1` are used to create:

- `emodel_x:variable@en0`
- `emodel_y:variable@en0`
- `emodel_x:variable@en1`
- `emodel_y:variable@en1`
- `emodel_x:variable@en2`
- `emodel_y:variable@en2`

in 2D for each node on a triangular element. and

- `emodel_x:variable@en0`
- `emodel_y:variable@en0`
- `emodel_z:variable@en0`
- `emodel_x:variable@en1`
- `emodel_y:variable@en1`
- `emodel_z:variable@en1`
- `emodel_x:variable@en2`
- `emodel_y:variable@en2`
- `emodel_z:variable@en2`

- emodel_x:variable@en3
- emodel_y:variable@en3
- emodel_z:variable@en3

in 3D for each node on a tetrahedral element.

The suffix en0 refers to the first node on the edge of the element and en1 refers to the second node. en2 and en3 specifies the derivatives with respect the variable at the nodes opposite the edges on the element being considered.

element_from_node_model

```
ds.element_from_node_model(device=STRING, region=STRING,
                     node_model=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
node_model=	The node model from which we are creating the edge model

This command creates an element edge model from a node model so that each corner of the element is represented. A node model, nmodel, would be be accessible as

- nmodel@en0
- nmodel@en1
- nmodel@en2
- nmodel@en3 (3D)

where en0, and en1 refers to the nodes on the element's edge. In 2D, en2 refers to the node on the triangle node opposite the edge. In 3D, en2 and en3 refers to the nodes on the nodes off the element edge on the tetrahedral element.

get_edge_model_list

```
ds.get_edge_model_list(device=STRING, region=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command

Returns a list of the edge models on the device region

get_edge_model_values

```
ds.get_edge_model_values(device=STRING, region=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the edge model values being returned as a list

get_element_model_list

```
ds.get_edge_model_list(device=STRING, region=STRING)
```

`device=` Device on which to apply this command
`region=` Region on which to apply this command
Returns a list of the element edge models on the device region

get_element_model_values

```
ds.get_edge_model_values(device=STRING, region=STRING,
                         name=STRING)
```

`device=` Device on which to apply this command
`region=` Region on which to apply this command
`name=` Name of the element edge model values being returned as a list

get_interface_model_list

```
ds.get_interface_model_values(device=STRING, interface=STRING)
```

`device=` Device on which to apply this command
`interface=` Interface on which apply this command
Returns a list of the interface models on the interface

get_interface_model_values

```
ds.get_interface_model_values(device=STRING, interface=STRING,
                         name=STRING)
```

`device=` Device on which to apply this command
`interface=` Interface on which apply this command
`name=` Name of the interface model values being returned as a list

get_node_model_list

```
ds.get_node_model_list(device=STRING, region=STRING)
```

`device=` Device on which to apply this command
`region=` Region on which to apply this command
Returns a list of the node models on the device region

get_node_model_values

```
ds.get_node_model_values(device=STRING, region=STRING,
                         name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the node model values being returned as a list

interface_model

```
ds.interface_model(device=STRING, interface=STRING,
                         name=STRING, equation=STRING)
```

device=	Device on which to apply this command
interface=	Interface on which apply this command
equation=	Equation used to describe the interface node model being created

interface_normal_model

```
ds.interface_normal_model(device=STRING, region=STRING, interface=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
interface=	Interface on which apply this command

This model creates the following edge models:

- iname_distance
- iname_normal_x (2D and 3D)
- iname_normal_y (2D and 3D)
- iname_normal_z (3D only)

where iname is the name of the interface. The normals are of the closest node on the interface. The sign is toward the interface.

node_model

```
ds.node_model(device=STRING, region=STRING, name=STRING,
              equation=STRING, display_type=OPTION)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the node model being created
equation=	Equation used to describe the node model being created
display_type=	Option for output display in graphical viewer
"nodisplay"	Data on node will not be displayed
"scalar"	Data on node is a scalar quantity (default)

node_solution

```
ds.node_solution(device=STRING, region=STRING, name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the solution being created

print_node_values

```
ds.print_node_values(device=STRING, region=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the node model values being printed to the screen

print_edge_values

```
ds.print_edge_values(device=STRING, region=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the edge model values being printed to the screen

print_element_values

```
ds.print_element_values(device=STRING, region=STRING,
                     name=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the element edge model values being printed to the screen

register_function

```
ds.register_function(name=STRING, nargs=STRING)
```

name=	Name of the function
nargs=	Number of arguments to the function

This command is used to register a new Python procedure for evaluation by SYMDIFF.

set_node_value

```
ds.set_node_values(device=STRING, region=STRING, name=STRING,
                index=INTEGER, value=FLOAT)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the node model being whose value is being set
index=	Index of node being set
value=	Value of node being set

A uniform value is used if index is not specified. Note that equation based node models will lose this value if their equation is recalculated.

set_node_values

```
ds.set_node_values(device=STRING, region=STRING, name=STRING,
                init_from=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
name=	Name of the node model being initialized
init_from=	Node model we are using to initialize the node solution

symdiff

```
ds.symdiff(expr=STRING)
```

expr=	Expression to send to SYMDIFF

This command returns an expression. All strings are treated as independent variables. It is primarily used for defining new functions to the parser.

vector_element_model

```
ds.vector_element_model(device=STRING, region=STRING,
                     element_model=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
element_model=	The element model for which we are calculating the vector compoenents

This command creates element edge models from an element model which represent the vector components on the element edge. An element model, `emodel`, would then have

- `emodel_x`
- `emodel_y`
- `emodel_z` (3D only)

The primary use of these components are for visualization.

vector_gradient

```
ds.vector_gradient(device=STRING, region=STRING,
                node_model=STRING, calc_type=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
node_model=	The node model from which we are creating the edge model
calc_type=	The node model from which we are creating the edge model
"default"	Consider all nodes for calculating the gradient field (default)
"avoidzero"	Do not take gradient at nodes where the node_model is zero

Used for noise analysis. The avoidzero option is important for noise analysis, since a node model value of zero is not physical for some contact and interface boundary conditions. For a given node model, model, a node model is created in each direction:

- model_gradx (1D)
- model_grady (2D and 3D)
- model_gradz (3D)

It is important not to use these models for simulation, since DEVSIM, does not have a way of evaluating the derivatives of these models. The models can be used for integrating the impedance field, and other postprocessing. The element_from_edge_model command (page 28) command can be used to create gradients for use in a simulation.

3.9 Geometry commands

get_device_list

```
ds.get_device_list()
```

Gets a list of devices on the simulation.

get_region_list

```
ds.get_region_list(device=STRING, contact=STRING, interface=STRING)
```

device=	Device on which to apply this command
contact=	If specified, gets the name of the region belonging to this contact on the device (optional)
interface=	If specified, gets the name of the region belonging to this interface on the device (optional)

get_contact_list

```
ds.get_contact_list(device=STRING)
```

device=	Device on which to apply this command

get_interface_list

```
ds.get_interface_list(device=STRING)
```

`device=` Device on which to apply this command

Chapter 4

Model Parameters

Parameters can be set using the commands in *Section 4.4,* Material commands *on page 38*. There are two complementary formalisms for doing this.

4.1 Parameters

Parameters are set globally, on devices, or on regions of a device. The models on each device region are automatically updated whenever parameters change.

```
ds.set_parameter(device="device", region="region", name="ThermalVoltage", value=0.0259)
```

4.2 Material database entries

Alternatively, parameters may be set based on material types. A database file is used for getting values on the regions of the device.

```
ds.create_db(filename="foodb")
ds.add_db_entry(material="global", parameter="q", value=1.60217646e-19,
        unit="coul", description="Electron Charge")
ds.add_db_entry(material="Si", parameter="one", value=1, unit="", description="")
ds.close_db
```

When a database entry is not available for a specific material, the parameter will be looked up on the `global` material entry.

4.3 Discussion

Both parameters and material database entries may be used in model expressions. Parameters have precedence in this situation. If a parameter is not found, then DEVSIM will also look for a circuit node by the name used in the model expression.

4.4 Material commands

get_dimension

```
ds.get_dimension (device=STRING)
```

`device=` Device on which to apply this command

get_parameter

```
ds.get_parameter (device=STRING, region=STRING, name=STRING)
```

`device=` Device on which to apply this command
`region=` Region on which to apply this command
`name=` Name of the parameter name being retrieved

Note that the `device` and `region` options are optional. If the region is not specified, the parameter is retrieved for the entire device. If the device is not specified, the parameter is retrieved for all devices. If the parameter is not found on the region, it is retrieved on the device. If it is not found on the device, it is retrieved over all devices.

get_parameter_list

```
ds.get_parameter_list (device=STRING, region=STRING)
```

`device=` Device on which to apply this command
`region=` Region on which to apply this command

Note that the `device` and `region` options are optional. If the region is not specified, the parameter is retrieved for the entire device. If the device is not specified, the parameter is retrieved for all devices. Unlike the `get_parameter` command (page 38), parameter names on the the device are not retrieved if they do not exist on the region. Similarly, the parameter names over all devices are not retrieved if they do not exist on the device.

set_parameter

```
ds.set_parameter (device=STRING, region=STRING, name=STRING, value=STRING)
```

`device=` Device on which to apply this command
`region=` Region on which to apply this command
`name=` Name of the parameter name being retrieved
`value=` value to set for the parameter

Note that the device and region options are optional. If the region is not specified, the parameter is set for the entire device. If the device is not specified, the parameter is set for all devices.

get_material

```
ds.get_material (device=STRING, region=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command

Returns the material for the specified region

set_material

```
ds.set_material (device=STRING, region=STRING, material=STRING)
```

device=	Device on which to apply this command
region=	Region on which to apply this command
material=	New material name

Sets the new material for a region.

create_db

```
ds.create_db (filename=STRING)
```

filename=	filename to create for the db

open_db

```
ds.open_db (filename=STRING, permission=OPTION)
```

filename=	filename to create for the db
permissions=	permissions on the db
"readwrite"	Open file for reading and writing
"readonly"	Open file for read only (default)

close_db

```
ds.close_db()
```

Closes the database so that its entries are no longer available.

save_db

```
ds.save_db()
```

Saves any new or modified db entries to the database file.

add_db_entry

```
ds.add_db_entry (material=STRING, parameter=STRING, \
             value=STRING, unit=STRING, description=STRING)
```

material=	Material name requested. global refers to all regions whose material does not have the parameter name specified
parameter=	Parameter name
value=	Value assigned for the parameter
unit=	String describing the units for this parameter name
description=	Description of the parameter for this material type.

The save_db command is used to commit these added entries permanently to the database.

get_db_entry

```
ds.get_db_entry (material=STRING, parameter=STRING)
```

material=	Material name
parameter=	Parameter name

This command returns a list containing the value, unit, and description for the requested material db entry.

Chapter 5

Circuits

5.1 Circuit elements

Circuit elements are manipulated using the commands in *Section 5.3, Circuit commands on page 41*. Using the `circuit_element` command (page 42) to add a circuit element will implicitly create the nodes being references.

A simple resistor divider with a voltage source would be specified as:

```
ds.circuit_element(name="V1", n1="1", n2="0", value=1.0)
ds.circuit_element(name="R1", n1="1", n2="2", value=5.0)
ds.circuit_element(name="R2", n1="2", n2="0", value=5.0)
```

Circuit nodes are created automatically when referred to by these commands. Voltage sources create an additional circuit node of the form `V1.I` to account for the current flowing through it.

5.2 Connecting devices

For devices to contribute current to an external circuit, the `contact_equation` command (page 21) should use the `circuit_node` option to specify the circuit node in which to integrate its current. This option does not create a node in the circuit. No circuit boundary condition for the contact equation will exist if the circuit node does not actually exist in the circuit. The `circuit_node_alias` command (page 42) may be used to associate the name specified on the contact equation to an existing circuit node on the circuit.

The circuit node names may be used in any model expression on the regions and interfaces. However, the simulator will only take derivatives with respect to circuit nodes names on models used to compose the contact equation.

5.3 Circuit commands

Circuit commands are for adding circuit elements to the simulation.

add_circuit_node

```
ds.add_circuit_node (name=STRING, value=FLOAT,
         variable_update=OPTION)
```

name=	Name of the circuit node being created
variable_update=	update type for circuit variable
"default"	Variable can be positive or negative
"log_damp"	Variable update is damped
"positive"	Solution update results in positive quantity
value=	initial value

circuit_alter

```
ds.circuit_alter (name=STRING, param=STRING,
                  value=FLOAT)
```

name=	Name of the circuit element being modified
param=	parameter being modified (default *value*)
value=	value for the parameter

circuit_element

```
ds.circuit_element (name=STRING, value=FLOAT,
               n1=STRING, n2=STRING, acreal=FLOAT, acimag=FLOAT)
```

name=	Name of the circuit element being created. A prefix of ”V” is for voltage source, ”I” for current source, ”R” for resistor, ”L” for inductor, and ”C” for capacitor.
value=	value for the default parameter of the circuit element
n1=	circuit node
n2=	circuit node
acreal=	real part of AC source for voltage
acimag=	imaginary part of AC source for voltage

circuit_node_alias

```
ds.circuit_node_alias (node=STRING, alias=STRING)
```

node=	circuit node being aliased
alias=	alias for the circuit node

get_circuit_equation_number

```
ds.get_circuit_equation_number (node=STRING)
```

`node=` Circuit node

Returns the row number correspond to circuit node in a region. Values are only valid when during the course of a solve.

get_circuit_node_list

Gets the list of the nodes in the circuit.

```
get_circuit_node_list()
```

get_circuit_node_value

```
ds.get_circuit_node_value(solution=STRING, node=STRING)
```

`solution=` name of the solution. ”dcop” is the name for the DC solution and is the default

`node=` circuit node of interest

get_circuit_solution_list

```
ds.get_circuit_solution_list()
```

Gets the list of available circuit solutions.

set_circuit_node_value

```
ds.set_circuit_node_value(solution=STRING, node=STRING,
                          value=FLOAT)
```

`solution=` name of the solution. ”dcop” is the name for the DC solution and is the default

`node=` circuit node of interest

`value=` new value

Chapter 6

Meshing

6.1 1D mesher

DEVSIM has an internal 1D mesher and the proper sequence of commands follow in this example.

```
ds.create_1d_mesh(mesh="cap")
ds.add_1d_mesh_line(mesh="cap", pos=0, ps=0.1, tag="top")
ds.add_1d_mesh_line(mesh="cap", pos=0.5, ps=0.1, tag="mid")
ds.add_1d_mesh_line(mesh="cap", pos=1, ps=0.1, tag="bot")
ds.add_1d_contact(mesh="cap", name="top", tag="top", material="metal")
ds.add_1d_contact(mesh="cap", name="bot", tag="bot", material="metal")
ds.add_1d_interface(mesh="cap", name="MySiOx", tag="mid")
ds.add_1d_region(mesh="cap", material="Si", region="MySiRegion", tag1="top", tag2="mid")
ds.add_1d_region(mesh="cap", material="Ox", region="MyOxRegion", tag1="mid", tag2="bot")
ds.finalize_mesh(mesh="cap")
ds.create_device(mesh="cap", device="device")
```

The `create_1d_mesh` command (page 50) is first used to initialize the specification of a new mesh by the name specified with the `mesh` option. The `add_1d_mesh_line` command (page 51) is used to specify the end points of the 1D structure, as well as the location of points where the spacing changes. The `tag` is used to create reference labels used for specifying the contacts, interfaces and regions.

The `add_1d_contact` command (page 51), `add_1d_interface` command (page 51) and `add_1d_region` command (page 51) are used to specify the contacts, interfaces and regions for the device.

Once the meshing commands have been completed, the `finalize_mesh` command (page 51) is called to create a mesh structure and then `create_device` command (page 53) is used to create a device using the mesh.

6.2 2D mesher

Similar to the 1D mesher, the 2D mesher uses a sequence of non-terminating mesh lines are specified in both the x and y directions to specify a mesh structure. As opposed to using tags, the regions are specified using add_2d_region command (page 52) as box coordinates on the mesh coordinates. The contacts and interfaces are specified using boxes, however it is best to ensure the the interfaces and contacts encompass only one line of points.

```
ds.create_2d_mesh(mesh="cap")
ds.add_2d_mesh_line(mesh="cap", dir="y", pos=-0.001, ps=0.001)
ds.add_2d_mesh_line(mesh="cap", dir="x", pos=xmin, ps=0.1)
ds.add_2d_mesh_line(mesh="cap", dir="x", pos=xmax, ps=0.1)
ds.add_2d_mesh_line(mesh="cap", dir="y", pos=ymin, ps=0.1)
ds.add_2d_mesh_line(mesh="cap", dir="y", pos=ymax, ps=0.1)
ds.add_2d_mesh_line(mesh="cap", dir="y", pos=+1.001, ps=0.001)
ds.add_2d_region(mesh="cap", material="gas", region="gas1", yl=-.001, yh=0.0)
ds.add_2d_region(mesh="cap", material="gas", region="gas2", yl=1.0, yh=1.001)
ds.add_2d_region(mesh="cap", material="Oxide", region="r0", xl=xmin, xh=xmax,
  yl=ymid1, yh=ymin)
ds.add_2d_region(mesh="cap", material="Silicon", region="r1", xl=xmin, xh=xmax,
  yl=ymid2, yh=ymid1)
ds.add_2d_region(mesh="cap", material="Silicon", region="r2", xl=xmin, xh=xmax,
  yl=ymid2, yh=ymax)

ds.add_2d_interface(mesh="cap", name="i0", region0="r0", region1="r1")
ds.add_2d_interface(mesh="cap", name="i1", region0="r1", region1="r2", xl=0, xh=1,
  yl=ymid2, yh=ymid2, bloat=1.0e-10)
ds.add_2d_contact(mesh="cap", name="top", region="r0", yl=ymin, yh=ymin,
  bloat=1.0e-10, material="metal")
ds.add_2d_contact(mesh="cap", name="bot", region="r2", yl=ymax, yh=ymax,
  bloat=1.0e-10, material="metal")
ds.finalize_mesh(mesh="cap")
ds.create_device(mesh="cap", device="device")
```

In the current implementation of the software, it is necessary to create a region on both sides of the contact in order to create a contact using add_2d_contact command (page 53) or an interface using add_2d_interface command (page 52).

Once the meshing commands have been completed, the finalize_mesh command (page 51) is called to create a mesh structure and then create_device command (page 53) is used to create a device using the mesh.

6.3 Using an external mesher

DEVSIM supports reading meshes from Genius Device Simulator and Gmsh. These meshes may only contain points, lines, triangles, and tetrahedra. Hybrid meshes or uniform meshes containing other elements are not supported at this time.

6.3.1 Genius

Meshes from the Genius Device Simulatorsoftware (see *Section 12.3.1, Genius on page 75*) can be imported using the CGNS format. In this example, `create_genius_mesh` command (page 48) returns region and boundary information which can be used to setup the device.

```
mesh_name = "nmos_iv"
result = create_genius_mesh(file="nmos_iv.cgns", mesh=mesh_name)

contacts = {}
for region_name, region_info in result['mesh_info']['regions'].iteritems():
  add_genius_region(mesh=mesh_name, genius_name=region_name,
                    region=region_name, material=region_info['material'])
  for boundary, is_electrode in region_info['boundary_info'].iteritems():
    if is_electrode:
      if boundary in contacts:
        contacts[boundary].append(region_name)
      else:
        contacts[boundary] = [region_name, ]

for contact, regions in contacts.iteritems():
  if len(regions) == 1:
    add_genius_contact(mesh=mesh_name, genius_name=contact, name=contact,
      region=regions[0], material='metal')
  else:
    for region in regions:
      add_genius_contact(mesh=mesh_name, genius_name=contact,
        name=contact+'@'+region, region=region, material='metal')



for boundary_name, regions in result['mesh_info']['boundaries'].iteritems():
  if (len(regions) == 2):
    add_genius_interface(mesh=mesh_name, genius_name=boundary_name,
      name=boundary_name, region0=regions[0], region1=regions[1])

finalize_mesh(mesh=mesh_name)
create_device(mesh=mesh_name, device=mesh_name)
```

Example locations are available on 81.

6.3.2 Gmsh

The Gmsh meshing software (see *Section 12.3.2, Gmsh on page 75*) can be used to create a 1D, 2D, or 3D mesh suitable for use in DEVSIM. When creating the mesh file using the software, use physical group names to map the difference entities in the resulting mesh file to a group name. In this example, a mos structure is read in:

```
ds.create_gmsh_mesh(file="gmsh_mos2d.msh", mesh="mos2d")
ds.add_gmsh_region(mesh="mos2d" gmsh_name="bulk", region="bulk", material="Silicon")
ds.add_gmsh_region(mesh="mos2d" gmsh_name="oxide", region="oxide", material="Silicon")
ds.add_gmsh_region(mesh="mos2d" gmsh_name="gate", region="gate", material="Silicon")
ds.add_gmsh_contact(mesh="mos2d" gmsh_name="drain_contact", region="bulk",
    name="drain", material="metal")
ds.add_gmsh_contact(mesh="mos2d" gmsh_name="source_contact", region="bulk",
    name="source", material="metal")
ds.add_gmsh_contact(mesh="mos2d" gmsh_name="body_contact", region="bulk",
    name="body", material="metal")
ds.add_gmsh_contact(mesh="mos2d" gmsh_name="gate_contact", region="gate",
    name="gate", material="metal")
ds.add_gmsh_interface(mesh="mos2d" gmsh_name="gate_oxide_interface", region0="gate",
  region1="oxide", name="gate_oxide")
ds.add_gmsh_interface(mesh="mos2d" gmsh_name="bulk_oxide_interface", region0="bulk",
  region1="oxide", name="bulk_oxide")
ds.finalize_mesh(mesh="mos2d")
ds.create_device(mesh="mos2d", device="mos2d")
```

Once the meshing commands have been completed, the `finalize_mesh` command (page 51) is called to create a mesh structure and then `create_device` command (page 53) is used to create a device using the mesh.

6.4 Loading and saving results

The `write_devices` command (page 54) is used to create an ASCII file suitable for saving data for restarting the simulation later. The `"devsim"` format encodes structural information, as well as the commands necessary for generating the models and equations used in the simulation. The `"devsimdata"` format is used for storing numerical information for use in other programs for analysis. The `load_devices` command (page 53) is then used to reload the device data for restarting the simulation.

6.5 Meshing commands

create_genius_mesh

```
ds.create_genius_mesh(mesh=STRING, file=STRING)
```

`file=` name of the Genius mesh file being read into DEVSIM
`mesh=` name of the mesh being generated

This command reads in a Genius mesh written in the CGNS format. If successful, it will return a dictionary containing information about the regions and boundaries in the mesh. Please see the example in *Section 6.3.1, Genius on page 47* for an example of how this information can be used for adding contacts and interfaces to the structure being created.

> *If the CGNS file was created with HDF as the underlying storage format, it may be necessary to convert it to ADF using the `hdf2adf` command before reading it into DEVSIM. This command is available as part of the CGNS library when it is compiled with HDF support. Please* Section 12.4.2, *CGNS* on page 76 *for availablility.*

add_genius_contact

```
ds.add_genius_contact(mesh=STRING, genius_name=STRING,
                name=STRING, region=STRING, material=STRING)
```

`genius_name=` boundary condition name in the Genius CGNS file
`material=` material for the contact being created
`mesh=` name of the mesh being generated
`name=` name of the contact begin created
`region=` region that the contact is attached to

add_genius_interface

```
ds.add_genius_interface(mesh=STRING, genius_name=STRING,
                  name=STRING, region0=STRING, region1=STRING)
```

`genius_name=` boundary condition name in the Genius CGNS file
`mesh=` name of the mesh being generated
`name=` name of the interface begin created
`region0=` first region that the interface is attached to
`region1=` second region that the interface is attached to

add_genius_region

```
ds.add_genius_region(mesh=STRING, genius_name=STRING,
                name=STRING, region=STRING, material=STRING)
```

`genius_name=` region name in the Genius CGNS file
`mesh=` name of the mesh being generated
`region=` name of the region begin created
`material=` material for the region being created

create_gmsh_mesh

```
ds.create_gmsh_mesh(mesh=STRING, file=STRING)
```

file=	name of the Gmsh mesh file being read into DEVSIM
mesh=	name of the mesh being generated

add_gmsh_contact

```
ds.add_gmsh_contact(mesh=STRING, gmsh_name=STRING,
                name=STRING, region=STRING, material=STRING)
```

gmsh_name=	physical group name in the Gmsh file
material=	material for the contact being created
mesh=	name of the mesh being generated
name=	name of the contact begin created
region=	region that the contact is attached to

add_gmsh_interface

```
ds.add_gmsh_interface(mesh=STRING, gmsh_name=STRING,
                  name=STRING, region0=STRING, region1=STRING)
```

gmsh_name=	physical group name in the Gmsh file
mesh=	name of the mesh being generated
name=	name of the interface begin created
region0=	first region that the interface is attached to
region1=	second region that the interface is attached to

add_gmsh_region

```
ds.add_gmsh_region(mesh=STRING, gmsh_name=STRING,
                name=STRING, region=STRING, material=STRING)
```

gmsh_name=	physical group name in the Gmsh file
mesh=	name of the mesh being generated
region=	name of the region begin created
material=	material for the region being created

create_1d_mesh

```
ds.create_1d_mesh(mesh=STRING)
```

mesh=	name of the 1D mesh being created

finalize_mesh

```
ds.finalize_mesh(mesh=STRING)
```

`mesh=` Mesh to finalize

Finalize the 1D or 2D mesh so no additional mesh specifications can be added and devices can be created.

add_1d_mesh_line

```
ds.add_1d_mesh_line mesh=STRING, tag=STRING, pos=FLOAT,
                    ns=FLOAT, ps=FLOAT)
```

`mesh=` Mesh to add the line to
`tag=` Text label for the position
`pos=` Position for the mesh point
`ns=` Spacing from this point in the negative direction (defaults to ps value)
`ps=` Spacing from this point in the positive direction

add_1d_interface

```
ds.add_1d_mesh_interface mesh=STRING, tag=STRING, name=STRING)
```

`mesh=` Mesh to add the interface to
`tag=` Text label for the position to add the interface
`name=` Name for the interface being created

add_1d_contact

```
ds.add_1d_mesh_contact mesh=STRING, tag=STRING, name=STRING, material=STRING)
```

`material=` material for the contact being created
`mesh=` Mesh to add the contact to
`name=` Name for the contact being created
`tag=` Text label for the position to add the contact

add_1d_region

```
ds.add_1d_region mesh=STRING, tag1=STRING, tag2=STRING,
                     region=STRING, material=STRING)
```

`mesh=` Mesh to add the line to
`tag1=` Text label for the position bounding the region being added
`tag2=` Text label for the position bounding the region being added
`region=` Name for the region being created
`material=` Material for the region being created

create_2d_mesh

```
ds.create_2d_mesh mesh=STRING)
```

mesh=	name of the 2D mesh being created

add_2d_mesh_line

```
ds.add_2d_mesh_line mesh=STRING, pos=FLOAT,
                    ns=FLOAT, ps=FLOAT)
```

mesh=	Mesh to add the line to
pos=	Position for the mesh point
ns=	Spacing from this point in the negative direction (defaults to ps value)
ps=	Spacing from this point in the positive direction

add_2d_region

```
ds.add_2d_region mesh=STRING, region=STRING,
                 material=STRING,
                 xl=FLOAT, xh=FLOAT,
                 yl=FLOAT, yh=FLOAT,
                 bloat=FLOAT)
```

mesh=	Mesh to add the region to
region=	Name for the region being created
material=	Material for the region being created
xl=	x position for corner of bounding box (default -MAXDOUBLE)
xh=	x position for corner of bounding box (default +MAXDOUBLE)
yl=	y position for corner of bounding box (default -MAXDOUBLE)
yh=	y position for corner of bounding box (default +MAXDOUBLE)
bloat=	Extend bounding box by this amount when search for mesh to include in region (default 1e-10)

add_2d_interface

```
ds.add_2d_interface mesh=STRING, name=STRING,
                    region0=STRING, region1=STRING,
                    xl=FLOAT, xh=FLOAT,
                    yl=FLOAT, yh=FLOAT,
                    bloat=FLOAT)
```

mesh=	Mesh to add the interface to
interface=	Name for the interface being created
region0=	Name of the region included in the interface
region1=	Name of the region included in the interface
xl=	x position for corner of bounding box (default -MAXDOUBLE)
xh=	x position for corner of bounding box (default +MAXDOUBLE)
yl=	y position for corner of bounding box (default -MAXDOUBLE)
yh=	y position for corner of bounding box (default +MAXDOUBLE)
bloat=	Extend bounding box by this amount when search for mesh to include in region (default 1e-10)

add_2d_contact

```
ds.add_2d_contact(mesh=STRING, name=STRING,
              region=STRING, material=STRING,
              xl=FLOAT, xh=FLOAT,
              yl=FLOAT, yh=FLOAT,
              bloat=FLOAT)
```

contact=	Name for the contact being created
material=	material for the contact being created
mesh=	Mesh to add the contact to
region=	Name of the region included in the contact
xl=	x position for corner of bounding box (default -MAXDOUBLE)
xh=	x position for corner of bounding box (default +MAXDOUBLE)
yl=	y position for corner of bounding box (default -MAXDOUBLE)
yh=	y position for corner of bounding box (default +MAXDOUBLE)
bloat=	Extend bounding box by this amount when search for mesh to include in region (default 1e-10)

create_device

```
ds.create_device mesh=STRING, device=STRING)
```

mesh=	name of the mesh being used to create a device
device=	name of the device being created

load_devices

```
ds.load_devices(file=STRING)
```

file=	name of the file to load the meshes from

write_devices

```
ds.write_devices(file=STRING, device=STRING, type=OPTION)
```

file=	name of the file to write the meshes to
device=	name of the device to write (optional)
type=	format to use
"devsim"	DEVSIM format
"devsim_data"	DEVSIM output format with numerical data for all models
"floops"	Floops format (for visualization in Postmini)
"tecplot"	Tecplot format (for visualization in Tecplot)
"vtk"	VTK format (for visualization in ParaView and VisIt)

Chapter 7

Solver

7.1 Solver

DEVSIM uses Newton methods to solve the system of PDE's. All of the analyses are performed using the `solve` command (page 57).

7.2 DC analysis

A DC analysis is performed using the `solve` command (page 57).

```
solve(type="dc", absolute_error=1.0e10, relative_error=1e-7 maximum_iterations=30)
```

7.3 AC analysis

An AC analysis is performed using the `solve` command (page 57). A circuit voltage source is required to set the AC source.

7.4 Noise/Sensitivity analysis

An noise analysis is performed using the `solve` command. A circuit node is specified in order to find its sensitivity to changes in the bulk quantities of each device. If the circuit node is named `V1.I`. A noise simulation is performed using:

```
solve(type="noise", frequency=1e5, output_node="V1.I")
```

Noise and sensitivity analysis is performed using the `solve` command (page 57). If the equation begin solved is `PotentialEquation`, the names of the scalar impedance field is then:

- `V1.I_PotentialEquation_real`
- `V1.I_PotentialEquation_imag`

and the vector impedance fields evaluated on the nodes are

- `V1.I_PotentialEquation_real_gradx`
- `V1.I_PotentialEquation_imag_gradx`
- `V1.I_PotentialEquation_real_grady` (2D and 3D)
- `V1.I_PotentialEquation_imag_grady` (2D and 3D)
- `V1.I_PotentialEquation_real_gradz` (3D only)
- `V1.I_PotentialEquation_imag_gradz` (3D only)

7.5 Transient analysis

Transient analysis is performed using the `solve` command (page 57). DEVSIM supports time-integration of the device PDE's. The three methods are supported are:

- BDF1
- TRBDF
- BDF2

7.6 Solver commands

get_contact_current

```
ds.get_contact_current(device=STRING, contact=STRING, equation=STRING)
```

`device=`	Device on which to apply this command
`contact=`	Contact on which to apply this command
`equation=`	Name of the contact equation from which we are retrieving the current

get_contact_charge

```
ds.get_contact_charge(device=STRING, contact=STRING, equation=STRING)
```

`device=`	Device on which to apply this command
`contact=`	Contact on which to apply this command
`equation=`	Name of the contact equation from which we are retrieving the charge

solve

```
ds.solve(type=OPTION,
         solver_type=OPTION,
         absolute_error=FLOAT,
         relative_error=FLOAT,
         charge_error=FLOAT,
         maximum_iterations=INTEGER,
         frequency=FLOAT,
         output_node=STRING,
         gamma=FLOAT,
         tdelta=FLOAT)
```

type=	type of solve being performed
"dc"	DC steady state simulation
"ac"	Small-signal AC simulation
"noise"	Small-signal AC simulation
"transient_dc"	Perform DC steady state and keep calculate charge at initial time step
"transient_bdf1"	Perform transient simulation using backward difference integration
"transient_bdf2"	Perform transient simulation using backward difference integration
"transient_tr"	Perform transient simulation using trapezoidal integration
solver_type=	Linear solver type
"direct"	Use LU factorization (default)
"iterative"	Use iterative solver
absolute_error=	Required update norm in the solve
relative_error=	Required relative update in the solve
charge_error=	Relative error between projected and solved charge during transient simulation
gamma=	Scaling factor for transient time step (default 1.0)
tdelta=	time step
maximum_iterations=	Maximum number of iterations in the DC solve
frequency=	Frequency for small-signal AC simulation
output_node=	Output circuit node for noise simulation

A small-signal AC source is set with the circuit voltage source.

Chapter 8

User Interface

8.1 Starting DEVSIM

Refer to *Chapter 11,* Installation *on page 73* for instructions on how to install DEVSIM. Once installed, DEVSIM may be invoked using the following command

```
devsim
```

for an interactive shell or

```
devsim filename.py
```

for batch mode where `filename.py` is the name of script being run. DEVSIM output is printed to the screen. To capture the output of the program, shell redirection commands may be used to direct the output to a file.

8.2 Python Language

8.2.1 Introduction

Python is the scripting language employed as the text interface to DEVSIM. Documentation and tutorials for the language are available from [4] A paper discussing the general benefits of using scripting languages may be found in [5].

8.2.2 DEVSIM commands

All of commands are in the `ds` namespace. In order to invoke a command, the command should be prefixed with `ds.`, or the following may be placed at the beginning of the script:

```
from ds import *
```

For details concerning error handling, please see *Section 8.4,* Error handling *on page 61*.

8.2.3 Other packages

DEVSIM is able to load Python packages. It is important to note that binary extensions loaded into DEVSIM must be compatible with the operating system which it was compiled for. To load an extension, it is first necessary to provide the path as an environment variable, or at program run time.

For example, if the Python packages on your system are available in "`/usr/share/tcltk`", it is necessary to set the environment variable in "`csh`" as

```
setenv PYTHONPATH /usr/share/tcltk
```

or in "`bash`"

```
export PYTHONPATH=/usr/share/tcltk
```

In the Python script, this may be done using using the appropriate paths for your system

```
import sys
sys.path.append("/usr/share/tcltk")
```

Please see *Python* on page 76 for more information on obtaining a copy of Python for your computer's operating system.

8.2.4 Advanced usage

In this manual, more advanced usage of the Python language may be used. The reader is encouraged to use a suitable reference to clarify the proper use of the scripting language constructs, such as control structures.

8.3 Unicode Support

Internally, DEVSIM uses UTF-8 encoding, and expects model equations and saved mesh files to be written using this encoding. Users are encouraged to use the standard ASCII character set if they do not wish to use this feature. When reading a Unicode encoded script, the built in Python interpreter should be made aware of the encoding of the source encoding using this on the first or second line of the script

```
# -*- coding: utf-8 -*-
```

This option is only required on systems, such as Microsoft Windows, which do not default to this encoding. Care should be taken when using Unicode names for visualization using the tools in *Chapter 10,* Visualization *on page 71*, as this character set may not be supported.

8.4 Error handling

8.4.1 Python errors

When a syntax error occurs in a Python script an exception may be thrown. If it is uncaught, then DEVSIM will terminate. More details may be found in an appropriate reference. An exception that is thrown by DEVSIM is of the type `ds.error`. It may be caught.

8.4.2 Fatal errors

When DEVSIM enters a state in which it may not recover. The interpreter should throw a Python exception with a message `DEVSIM FATAL`. At this point DEVSIM may enter an inconsistent state, so it is suggested not to attempt to continue script execution if this occurs.

In rare situations, the program may behave in an erratic manner, print a message, such as `UNEXPECTED` or terminate abruptly. Please report this to DEVSIM LLC using the contact information in the front cover of this manual.

8.4.3 Floating point exceptions

During model evaluation, DEVSIM will attempt to detect floating point issues and return an error with some diagnostic information printed to the screen, such as the symbolic expression being evaluated. Floating point errors may be characterized as invalid, division by zero, and numerical overflow. This is considered to be a fatal error.

8.4.4 Solver errors

When using the `solve` command (page 57), the solver may not converge and a message will be printed and an exception may be thrown. The solution will be restored to its previous value before the simulation began. This exception may be caught and the bias conditions may be changed so the simulation may be continued. For example:

```
try:
  solve(type="dc", absolute_error=abs_error,
    relative_error=rel_error, maximum_iterations=max_iter)
except ds.error as msg:
  if msg[0].find("Convergence failure") != 0:
    raise
  #### put code to modify step here.
```

8.4.5 Verbosity

The `set_parameter` command (page 38) may be used to set the verbosity globally, per device, or per region. Setting the `debug_level` parameter to `info` results in the default level of information to the screen. Setting this option to `verbose` or any other name results in more information to the screen which may be useful for debugging.

The following example sets the default level of debugging for the entire simulation, except that the gate region will have additional debugging information.

```
ds.set_parameter(name="debug_level", value="info")
ds.set_parameter(device="device" region="gate", name="debug_level", value="verbose")
```

8.5 Parallelization

Routines for the evaluating of models have been parallelized. In order to select the number of threads to use

```
ds.set_parameter(name="threads_available", value=2)
```

where the value specified is the number of threads to be used. By default, DEVSIM does not use threading. For regions with a small number of elements, the time for switching threads is more than the time to evaluate in a single thread. To set the minimum number of elements for a calculation, set the following parameter.

```
ds.set_parameter(name="threads_task_size", value=1024)
```

Chapter 9

SYMDIFF

9.1 Overview

SYMDIFF is a tool capable of evaluating derivatives of symbolic expressions. Using a natural syntax, it is possible to manipulate symbolic equations in order to aid derivation of equations for a variety of applications. It has been tailored for use within DEVSIM.

9.2 Syntax

9.2.1 Variables and numbers

Variables and numbers are the basic building blocks for expressions. A variable is defined as any sequence of characters beginning with a letter and followed by letters, integer digits, and the _ character. Note that the letters are case sensitive so that a and A are not the same variable. Any other characters are considered to be either mathematical operators or invalid, even if there is no space between the character and the rest of the variable name.

Examples of valid variable names are:

```
a, dog, var1, var_2
```

Numbers can be integer or floating point. Scientific notation is accepted as a valid syntax. For example:

```
1.0, 1.0e-2, 3.4E-4
```

Expression	Description
`(exp1)`	Parenthesis for changing precedence
`+exp1`	Unary Plus
`-exp1`	Unary Minus
`!exp1`	Logical Not
`exp1 ^ exp2`	Exponentiation
`exp1 * exp2`	Multiplication
`exp1 / exp2`	Division
`exp1 + exp2`	Addition
`exp1 - exp2`	Subtraction
`exp1 < exp2`	Test Less
`exp1 <= exp2`	Test Less Equal
`exp1 > exp2`	Test Greater
`exp1 >= exp2`	Test Greater Equal
`exp1 == exp2`	Test Equality
`exp1 != exp2`	Test Inequality
`exp1 && exp2`	Logical And
`exp1 || exp2`	Logical Or
`variable`	Independent Variable
`number`	Integer or decimal number

Table 9.1: Basic expressions involving unary, binary, and logical operators.

9.2.2 Basic expressions

In Table 9.1, the basic syntax for the language is presented. An expression may be composed of variables and numbers tied together with mathematical operations. Order of operations is from bottom to top in order of increasing precedence. Operators with the same level of precedence are contained within horizontal lines.

In the expression `a + b * c`, the multiplication will be performed before the addition. In order to override this precedence, parenthesis may be used. For example, in `(a + b) * c`, the addition operation is performed before the multiplication.

The logical operators are based on non zero values being true and zero values being false. The test operators are evaluate the numerical values and result in 0 for false and 1 for true.

It is important to note since values are based on double precision arithmetic, testing for equality with values other than 0.0 may yield unexpected results.

Function	Description
acosh(exp1)	Inverse Hyperbolic Cosine
asinh(exp1)	Inverse Hyperbolic Sine
atanh(exp1)	Inverse Hyperbolic Tangent
B(exp1)	Bernoulli Function
dBdx(exp1)	derivative of Bernoulli function
derfcdx(exp1)	derivative of complementary error function
derfdx(exp1)	derivative error function
dFermidx(exp1)	derivative of Fermi Integral
dInvFermidx(exp1)	derivative of InvFermi Integral
dot2d(exp1x, exp1y, exp2x, exp2y)	exp1x*exp2x+exp1y *exp2y
erfc(exp1)	complementary error function
erf(exp1)	error function
exp(exp1)	exponent
Fermi(exp1)	Fermi Integral
ifelse(test, exp1, exp2)	if test is true, then evaluate exp1, otherwise exp2
if(test, exp)	if test is true, then evaluate exp, otherwise 0
InvFermi(exp1)	inverse of the Fermi Integral
log(exp1)	natural log
max(exp1, exp2)	maximum of the two arguments
min(exp1, exp2)	minimum of the two arguments
pow(exp1, exp2)	take exp1 to the power of exp2
sgn(exp1)	sign function
step(exp1)	unit step function
kahan3(exp1, exp2, exp3)	Extended precision addition of arguments
kahan4(exp1, exp2, exp3, exp4)	Extended precision addition of arguments
vec_max	maximum of all the values over the entire region or interface
vec_min	minimum of all the values over the entire region or interface
vec_sum	sum of all the values over the entire region or interface

Table 9.2: Predefined Functions.

9.2.3 Functions

In Table 9.2 are the built in functions of SYMDIFF. Note that the `pow` function uses the , operator to separate arguments. In addition an expression like `pow(a,b+y)` is equivalent to an expression like `a^(b+y)`. Both `exp` and `log` are provided since many derivative expressions can be expressed in terms of these two functions. It is possible to nest expressions within functions and vice-versa.

Command	Description
`diff(obj1, var)`	Take derivative of `obj1` with respect to variable `var`
`expand(obj)`	Expand out all multiplications into a sum of products
`help`	Print description of commands
`scale(obj)`	Get constant factor
`sign(obj)`	Get sign as 1 or -1
`simplify(obj)`	Simplify as much as possible
`subst(obj1,obj2,obj3)`	substitute obj3 for obj2 into obj1
`unscaledval(obj)`	Get value without constant scaling
`unsignedval(obj)`	Get unsigned value

Table 9.3: Commands.

9.2.4 Commands

Commands are shown in Table 9.3. While they appear to have the same form as functions, they are special in the sense that they manipulate expressions and are never present in the expression which results. For example, note the result of the following command

```
> diff(a*b, b)
a
```

Command	Description
`clear(name)`	Clears the name of a user function
`declare(name(arg1, arg2, ...))`	declare function name taking dummy arguments arg1, arg2, ... Derivatives assumed to be 0
`define(name(arg1, arg2, ...), obj1, obj2, ...)`	declare function name taking arguments arg1, arg2, . . . having corresponding derivatives obj1, obj2, ...

Table 9.4: Commands for user functions.

9.2.5 User functions

Commands for specifying and manipulating user functions are listed in Table 9.4. They are used in order to define new user function, as well as the derivatives of the functions with respect to the user variables. For example, the following expression defines a function named `f` which takes one argument.

```
> define(f(x), 0.5*x)
```

The list after the function protoype is used to define the derivatives with respect to each of the independent variables. Once defined, the function may be used in any other expression. In additions the any expression can be used as an arguments. For example:

```
> diff(f(x*y),x)
((0.5 * (x * y)) * y)
> simplify((0.5 * (x * y)) * y)
(0.5 * x * (y^2))
```

The chain rule is applied to ensure that the derivative is correct. This can be expressed as

$$\frac{\partial}{\partial x} f(u,v,\ldots) = \frac{\partial u}{\partial x} \cdot \frac{\partial}{\partial u} f(u,v,\ldots) + \frac{\partial v}{\partial x} \cdot \frac{\partial}{\partial v} f(u,v,\ldots)$$

The `declare` command is required when the derivatives of two user functions are based on one another. For example:

```
> declare(cos(x))
cos(x)
> define(sin(x),cos(x))
sin(x)
> define(cos(x),-sin(x))
cos(x)
```

When declared, a functions derivatives are set to 0, unless specified with a define command. It is now possible to use these expressions as desired.

```
> diff(sin(cos(x)),x)
(cos(cos(x)) * (-sin(x)))
> simplify(cos(cos(x)) * (-sin(x)))
(-cos(cos(x)) * sin(x))
```

9.2.6 Macro assignment

The use of macro assignment allows the substitution of expressions into new expressions. Every time a command is successfully used, the resulting expression is assigned to a special macro definition, `$_`.

In this example, the result of the each command is substituted into the next.

```
> a+b
(a + b)
> $_-b
((a + b) - b)
> simplify($_)
a
```

In addition to the default macro definition, it is possible to specify a variable identifier by using the `$` character followed by an alphanumeric string beginning with a letter. In addition to letters and numbers, a `_` character may be used as well. A macro which has not previously assigned will implicitly use `0` as its value.

This example demonstrates the use of macro assignment.

```
> $a1 = a + b
(a + b)
> $a2 = a - b
(a - b)
> simplify($a1+$a2)
(2 * a)
```

9.3 Invoking SYMDIFF from DEVSIM

9.3.1 Equation parser

The `symdiff` command (page 33) should be used when defining new functions to the parser. Since you do not specify regions or interfaces, it considers all strings as being independent variables, as opposed to models. *Section 3.8,* Model commands *on page 23* presents commands which have the concepts of models. A "`;`" should be used to separate each statement.

This is a sample invocation from DEVSIM

```
% symdiff(expr="subst(dog * cat, dog, bear)")
(bear * cat)
```

9.3.2 Evaluating external math

The `register_function` command (page 32) is used to evaluate functions declared or defined within SYMDIFF. A Python procedure may then be used taking the same number of arguments. For example:

```
from math import cos
from math import sin
symdiff(expr="declare(sin(x))")
symdiff(expr="define(cos(x), -sin(x))")
symdiff(expr="define(sin(x),  cos(x))")
register_function(name=cos, nargs=1)
register_function(name=sin, nargs=1)
```

The `cos` and `sin` function may then be used for model evaluation. For improved efficiency, it is possible to create procedures written in C or C++ and load them into Python.

9.3.3 Models

When used withing the model commands discussed in *Section 3.8,* Model commands *on page 23*, DEVSIM has been extended to recognize model names in the expressions. In this situation, the derivative of a model named, `model`, with respect to another model, `variable`, is then `model:variable`.

During the element assembly process, DEVSIM evaluates all models of an equation together. While the expressions in models and their derivatives are independent, the software uses a caching scheme to ensure that redundant calculations are not performed. It is recommended, however, that users developing their own models investigate creating intermediate models in order to improve their understanding of the equations that they wish to be assembled.

Chapter 10

Visualization

10.1 Introduction

DEVSIM is able to create files for visualization tools. Information about acquiring these tools are presented in *Chapter 12.3,* External Software Tools *on page 75*.

10.2 Using Tecplot

The `write_devices` command (page 54) is used to create an ASCII file suitable for use in Tecplot. Edge quantities are interpolated onto the node positions in the resulting structure. Element edge quantities are interpolated onto the centers of each triangle or tetrahedron in the mesh.

```
write_devices(file="mos_2d_dd.dat", type="tecplot")
```

10.3 Using Postmini

The `write_devices` command (page 54) is used to create an ASCII file suitable for use in Postmini. Edge and element edge quantities are interpolated onto the node positions in the resulting structure.

```
write_devices(file="mos_2d_dd.flps", type="floops")
```

10.4 Using ParaView

The `write_devices` command (page 54) is used to create an ASCII file suitable for use in ParaView. Edge quantities are interpolated onto the node positions in the resulting structure. Element edge quantities are interpolated onto the centers of each triangle or tetrahedron in the mesh.

```
write_devices(file="mos_2d_dd", type="vtk")
```

One `vtu` file per device region will be created, as well as a `vtm` file which may be used to load all of the device regions into ParaView.

10.5 Using VisIt

VisIt supports reading the Tecplot and ParaView formats. When using the `vtk` option on the `write_devices` command (page 54), a file with a `visit` filename extension is created to load the files created for ParaView.

10.6 DEVSIM

DEVSIM has several commands for getting information on the mesh. Those related to post processing are described in *Section 3.8,* Model commands *on page 23* and *Section 3.9,* Geometry commands *on page 34*.

See *Section 6.4,* Loading and saving results *on page 48* for information about loading and saving mesh information to a file.

Chapter 11

Installation

11.1 Availability

Information about the open source version of DEVSIM is available from http://www.devsim.org. This site contains up-to-date information about where to obtain compiled and source code versions of this software. It also contains information about how to get support and participate in the development of this project.

11.1.1 Supported platforms

DEVSIM is compiled and tested on the platforms in Table 11.1. If you require a version on a different software platform, please contact us.

Platform	Bits	OS Version
Linux	32, 64 64	Ubuntu 12.04 (LTS) Red Hat Enterprise Linux 6.5 (Centos 6.5 compatible)
Apple Mac OS X	64	Mac OS X 10.9 (Mavericks)

Table 11.1: Current platforms for DEVSIM.

11.1.2 Binary availability

Compiled packages for the the platforms in Table 11.1 are currently available from http://sourceforge.net/projects/devsim. The prerequisites on each platform are the default Python and Tcl packages for your operating system.

11.1.3 Source code availability

DEVSIM is also available in source code form from http://www.github.com/devsim/devsim.

11.2 Directory Structure

A `devsim` directory is created with the following sub directories.

`bin` contains the devsim binary

`doc` contains product documentation

`examples` contains example scripts

`python_packages` contains runtime libraries

`testing` contains additional examples used for testing

11.3 Running DEVSIM

See *Chapter 8,* User Interface *on page 59* for instructions on how to invoke DEVSIM.

Chapter 12

Additional Information

12.1 DEVSIM License

Individual files are covered by the license terms contained in the comments at the top of the file. Contributions to this project are subject to the license terms of their authors. In general, DEVSIM is covered by the following licenses:

Apache 2.0 License This applies to scripts implementing physics, tests, and examples. Please see the `NOTICE` and `LICENSE` file for more information.

GNU Lesser General Public License Version 3 This applies to the source code. Please see the `COPYING` file for more information.

12.2 SYMDIFF

SYMDIFF is available from http://www.symdiff.org and is covered by the terms of the Apache 2.0 License.

12.3 External Software Tools

12.3.1 Genius

Genius is available in commercial and open source versions from http://www.cogenda.com.

12.3.2 Gmsh

Gmsh [6] is available from http://geuz.org/gmsh/.

12.3.3 ParaView

ParaView is an open source visualization tool available at http://www.paraview.org.

12.3.4 Postmini

Postmini is available from http://home.comcast.net/~john.faricelli/tcad.htm.

12.3.5 Tecplot

Tecplot is a commercial visualization tool available from http://www.tecplot.com.

12.3.6 VisIt

VisIt is an open source visualization tool available from https://wci.llnl.gov/codes/visit/.

12.4 Library Availablilty

The following tools are used to build DEVSIM.

12.4.1 BLAS and LAPACK

These are the basic linear algebra routines used directly by DEVSIM and by SuperLU. Reference versions are available from http://www.netlib.org. There are optimized versions available from other vendors.

12.4.2 CGNS

CGNS (CFD Generalized Notation System) is an open source library, which implements the storage format used to read Genius Device Simulator meshes. It is available from http://www.cgns.org.

12.4.3 Python

A Python distribution is required for using DEVSIM and is distributed with many operating system. Additional information is available at http://www.python.org. It should be stressed that binary packages must be compatible with the Python distribution used by DEVSIM.

12.4.4 SQLite3

SQLite3 is an open source database engine used for the material database and is available from http://www.sqlite.com.

12.4.5 SuperLU

SuperLU [7] is used within DEVSIM and and is available from http://crd-legacy.lbl.gov/~xiaoye/SuperLU:

12.4.6 Tcl

Tcl is the original parser for DEVSIM and is superseded by Python. It is still used for some of the tests. Tcl is available from http://www.tcl.tk.

12.4.7 zlib

zlib is an open source compression library available from http://www.zlib.net.

Part II

Examples

Chapter 13

Example Overview

In the following chapters, examples are presented for the use of DEVSIM to solve some simulation problems. Examples are also located in the DEVSIM distribution and their location is mentioned in *Section 11.2,* Directory Structure *on page 74*.

The following example directories are contained in the distribution.

capacitance These are 1D and 2D capacitor simulations, using the internal mesher. A description of these examples is presented in *Chapter 14,* Capacitor *on page 83*.

diode This is a collection of 1D, 2D, and 3D diode structures using the internal mesher, as well as Gmsh. These examples are discussed in *Chapter 15,* Diode *on page 93*.

bioapp1 This is a biosensor application.

genius This directory has examples importing meshes from Genius Device Simulator.

vector_potential This is a 2d magnetic field simulation solving for the magnetic potential. The simulation script is `vector_potential/twowire.py` A simulation result for two wires conducting current is shown in Figure 13.1.

mobility This is an advanced example using electric field dependendent mobility models.

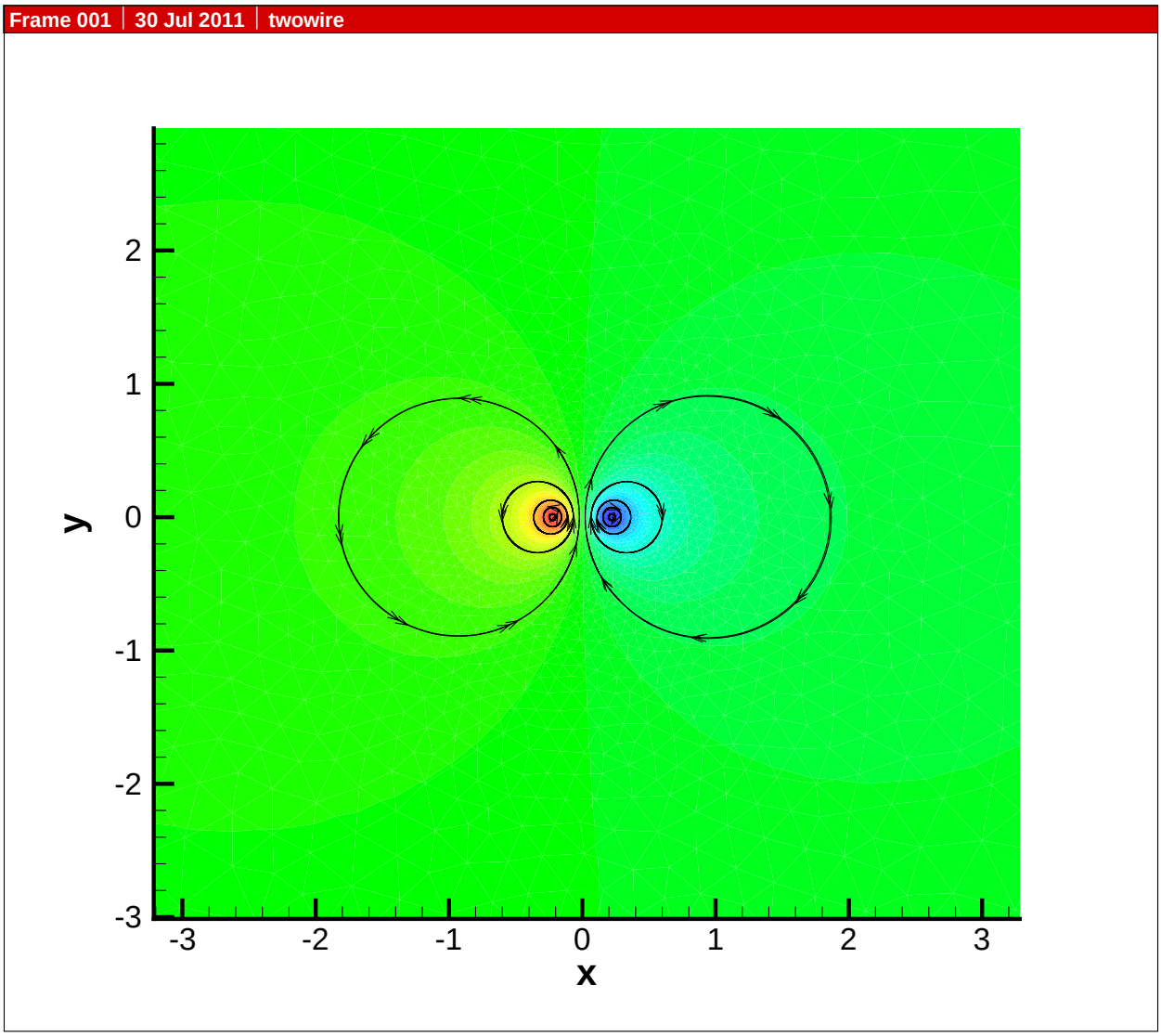

Figure 13.1: Simulation result for solving for the magnetic potential and field. The coloring is by the Z component of the magnetic potential, and the stream traces are for components of magnetic field.

Chapter 14

Capacitor

14.1 Overview

In this chapter, we present a capacitance simulations. The purpose is to demonstrate the use of DEVSIM with a rather simple example. The first example in *Section 14.2,* 1D Capacitor *on page 83* is called `cap1d.py` and is located in the `examples/capacitance` directory distributed with DEVSIM. In this example, we have manually taken the derivative expressions. In other examples, we will show use of SYMDIFF to create the derivatives in an automatic fashion. The second example is in *Section 14.3,* 2D Capacitor *on page 87*.

14.2 1D Capacitor

14.2.1 Equations

In this example, we are solving Poisson's equation. In differential operator form, the equation to be solved over the structure is:

$$\varepsilon \nabla^2 \psi = 0 \tag{14.1}$$

and the contact boundary equations are

$$\psi_i - V_c = 0 \tag{14.2}$$

where ψ_i is the potential at the contact node and V_c is the applied voltage.

14.2.2 Creating the mesh

The following statements create a one-dimensional mesh which is 1 m long with a 0.1 m spacing. A contact is placed at 0 and 1 and are named `contact1` and `contact2` respectively.

```
from ds import *
device="MyDevice"
region="MyRegion"

###
```

```
### Create a 1D mesh
###
create_1d_mesh (mesh="mesh1")
add_1d_mesh_line (mesh="mesh1", pos=0.0, ps=0.1, tag="contact1")
add_1d_mesh_line (mesh="mesh1", pos=1.0, ps=0.1, tag="contact2")
add_1d_contact   (mesh="mesh1", name="contact1", tag="contact1", material="metal")
add_1d_contact   (mesh="mesh1", name="contact2", tag="contact2", material="metal")
add_1d_region    (mesh="mesh1", material="Si", region=region, tag1="contact1", tag2="contact2")
finalize_mesh (mesh="mesh1")
create_device (mesh="mesh1", device=device)
```

14.2.3 Setting device parameters

In this section, we set the value of the permittivity to that of SiO_2.

```
###
### Set parameters on the region
###
set_parameter(device=device, region=region, name="Permittivity", value=3.9*8.85e-14)
```

14.2.4 Creating the models

Solving for the Potential, ψ, we first create the solution variable.

```
###
### Create the Potential solution variable
###
node_solution(device=device, region=region, name="Potential")
```

In order to create the edge models, we need to be able to refer to `Potential` on the nodes on each edge.

```
###
### Creates the Potential@n0 and Potential@n1 edge model
###
edge_from_node_model(device=device, region=region, node_model="Potential")
```

We then create the `ElectricField` model with knowledge of `Potential` on each node, as well as the `EdgeInverseLength` of each edge. We also manually calculate the derivative of `ElectricField` with `Potential` on each node and name them `ElectricField:Potential@n0` and `ElectricField:Potential@n1`.

```
###
### Electric field on each edge, as well as its derivatives with respect to
### the potential at each node
###
edge_model(device=device, region=region, name="ElectricField",
                equation="(Potential@n0 - Potential@n1)*EdgeInverseLength")

edge_model(device=device, region=region, name="ElectricField:Potential@n0",
```

```
                equation="EdgeInverseLength")

edge_model(device=device, region=region, name="ElectricField:Potential@n1",
                equation="-EdgeInverseLength")
```

We create `DField` to account for the electric displacement field.

```
###
### Model the D Field
###
edge_model(device=device, region=region, name="DField",
           equation="Permittivity*ElectricField")

edge_model(device=device, region=region, name="DField:Potential@n0",
           equation="diff(Permittivity*ElectricField, Potential@n0)")

edge_model(device=device, region=region, name="DField:Potential@n1",
           equation="-DField:Potential@n0")
```

The bulk equation is now created for the structure using the models and parameters we have previously defined.

```
###
### Create the bulk equation
###
equation(device=device, region=region, name="PotentialEquation", variable_name="Potential",
    edge_model="DField", variable_update="default")
```

14.2.5 Contact boundary conditions

We then create the contact models and equations. We use the Python `foreach` looping construct and variable substitutions to create a unique model for each contact, `contact1_bc` and `contact2_bc`.

```
###
### Contact models and equations
###
for c in ("contact1", "contact2"):
  contact_node_model(device=device, contact=c, name="%s_bc" % c,
           equation="Potential - %s_bias" % c)

  contact_node_model(device=device, contact=c, name="%s_bc:Potential" % c,
            equation="1")

  contact_equation(device=device, contact=c, name="PotentialEquation",
            variable_name="Potential",
            node_model="%s_bc" % c, edge_charge_model="DField")
```

In this example, the contact bias is applied through parameters named `contact1_bias` and `contact2_bias`. When applying the boundary conditions through circuit nodes, models with respect to their names and their derivatives would be required.

14.2.6 Setting the boundary conditions

```
###
### Set the contact
###
set_parameter(device=device, region=region, name="contact1_bias", value=1.0e-0)
set_parameter(device=device, region=region, name="contact2_bias", value=0.0)

###
### Solve
###
solve(type="dc", absolute_error=1.0, relative_error=1e-10, maximum_iterations=30)

###
### Print the charge on the contacts
###
for c in ("contact1", "contact2"):
  print("contact: %s charge: %1.5e"
    % (c, get_contact_charge(device=device, contact=c, equation="PotentialEquation")))
```

14.2.7 Running the simulation

We run the simulation and see the results.

```
 capacitance> devsim cap1d.py
----------------------------------------

 DEVSIM

 Version: Beta 0.01

 Copyright 2009-2013 Devsim LLC

----------------------------------------


contact2
 (region: MyRegion)
 (contact: contact1)
 (contact: contact2)
Region "MyRegion" on device "MyDevice" has equations 0:10
Device "MyDevice" has equations 0:10
number of equations 11
Iteration: 0
  Device: "MyDevice" RelError: 1.00000e+00 AbsError: 1.00000e+00
    Region: "MyRegion" RelError: 1.00000e+00 AbsError: 1.00000e+00
      Equation: "PotentialEquation" RelError: 1.00000e+00 AbsError: 1.00000e+00
Iteration: 1
  Device: "MyDevice" RelError: 2.77924e-16 AbsError: 1.12632e-16
    Region: "MyRegion" RelError: 2.77924e-16 AbsError: 1.12632e-16
      Equation: "PotentialEquation" RelError: 2.77924e-16 AbsError: 1.12632e-16
contact: contact1 charge: 3.45150e-13
contact: contact2 charge: -3.45150e-13
```

Which corresponds to our expected result of 3.451510^{-13} F/cm^2 for a homogenous capacitor.

14.3 2D Capacitor

This example is called `cap2d.py` and is located in the `examples/capacitance` directory distributed with DEVSIM. This file uses the same physics as the 1d example, but with a 2d structure. The mesh is built using the DEVSIM internal mesher. An air region exists with two electrodes in the simulation domain.

14.4 Defining the mesh

```
from ds import *
device="MyDevice"
region="MyRegion"

xmin=-25
x1  =-24.975
x2  =-2
x3  =2
x4  =24.975
xmax=25.0

ymin=0.0
y1  =0.1
y2  =0.2
y3  =0.8
y4  =0.9
ymax=50.0

create_2d_mesh(mesh=device)
add_2d_mesh_line(mesh=device, dir="y", pos=ymin, ps=0.1)
add_2d_mesh_line(mesh=device, dir="y", pos=y1  , ps=0.1)
add_2d_mesh_line(mesh=device, dir="y", pos=y2  , ps=0.1)
add_2d_mesh_line(mesh=device, dir="y", pos=y3  , ps=0.1)
add_2d_mesh_line(mesh=device, dir="y", pos=y4  , ps=0.1)
add_2d_mesh_line(mesh=device, dir="y", pos=ymax, ps=5.0)

device=device
region="air"

add_2d_mesh_line(mesh=device, dir="x", pos=xmin, ps=5)
add_2d_mesh_line(mesh=device, dir="x", pos=x1  , ps=2)
add_2d_mesh_line(mesh=device, dir="x", pos=x2  , ps=0.05)
add_2d_mesh_line(mesh=device, dir="x", pos=x3  , ps=0.05)
add_2d_mesh_line(mesh=device, dir="x", pos=x4  , ps=2)
add_2d_mesh_line(mesh=device, dir="x", pos=xmax, ps=5)

add_2d_region(mesh=device, material="gas"  , region="air", yl=ymin, yh=ymax, xl=xmin, xh=xmax)
add_2d_region(mesh=device, material="metal", region="m1" , yl=y1  , yh=y2  , xl=x1  , xh=x4)
add_2d_region(mesh=device, material="metal", region="m2" , yl=y3  , yh=y4  , xl=x2  , xh=x3)
```

```
# must be air since contacts don't have any equations
add_2d_contact(mesh=device, name="bot", region="air", material="metal", yl=y1, yh=y2, xl=x1, xh=x4)
add_2d_contact(mesh=device, name="top", region="air", material="metal", yl=y3, yh=y4, xl=x2, xh=x3)
finalize_mesh(mesh=device)
create_device(mesh=device, device=device)
```

14.5 Setting up the models

```
###
### Set parameters on the region
###
set_parameter(device=device, region=region, name="Permittivity", value=3.9*8.85e-14)

###
### Create the Potential solution variable
###
node_solution(device=device, region=region, name="Potential")

###
### Creates the Potential@n0 and Potential@n1 edge model
###
edge_from_node_model(device=device, region=region, node_model="Potential")

###
### Electric field on each edge, as well as its derivatives with respect to
### the potential at each node
###
edge_model(device=device, region=region, name="ElectricField",
                equation="(Potential@n0 - Potential@n1)*EdgeInverseLength")

edge_model(device=device, region=region, name="ElectricField:Potential@n0",
                equation="EdgeInverseLength")

edge_model(device=device, region=region, name="ElectricField:Potential@n1",
                equation="-EdgeInverseLength")

###
### Model the D Field
###
edge_model(device=device, region=region, name="DField",
           equation="Permittivity*ElectricField")

edge_model(device=device, region=region, name="DField:Potential@n0",
           equation="diff(Permittivity*ElectricField, Potential@n0)")

edge_model(device=device, region=region, name="DField:Potential@n1",
           equation="-DField:Potential@n0")
```

```
###
### Create the bulk equation
###
equation(device=device, region=region, name="PotentialEquation", variable_name="Potential",
    edge_model="DField", variable_update="default")


###
### Contact models and equations
###
for c in ("top", "bot"):
  contact_node_model(device=device, contact=c, name="%s_bc" % c,
           equation="Potential - %s_bias" % c)

  contact_node_model(device=device, contact=c, name="%s_bc:Potential" % c,
            equation="1")

  contact_equation(device=device, contact=c, name="PotentialEquation",
            variable_name="Potential",
            node_model="%s_bc" % c, edge_charge_model="DField")

###
### Set the contact
###
set_parameter(device=device, name="top_bias", value=1.0e-0)
set_parameter(device=device, name="bot_bias", value=0.0)



edge_model(device=device, region="m1", name="ElectricField", equation="0")
edge_model(device=device, region="m2", name="ElectricField", equation="0")
node_model(device=device, region="m1", name="Potential", equation="bot_bias;")
node_model(device=device, region="m2", name="Potential", equation="top_bias;")


solve(type="dc", absolute_error=1.0, relative_error=1e-10, maximum_iterations=30,
  solver_type="direct")
```

14.6 Fields for visualization

Before writing the mesh out for visualization, the `element_from_edge_model` is used to calculate the electric field at each triangle center in the mesh. The components are the `ElectricField_x` and `ElectricField_y`.

```
element_from_edge_model(edge_model="ElectricField", device=device, region=region)
print(get_contact_charge(device=device, contact="top", equation="PotentialEquation"))
print(get_contact_charge(device=device, contact="bot", equation="PotentialEquation"))
```

```
write_devices(file="cap2d.msh", type="devsim")
write_devices(file="cap2d.dat", type="tecplot")
```

14.7 Running the simulation

```
----------------------------------------

 DEVSIM

 Version: Beta 0.01

 Copyright 2009-2013 Devsim LLC

----------------------------------------


Creating Region air
Creating Region m1
Creating Region m2
Adding 8281 nodes
Adding 23918 edges with 22990 duplicates removed
Adding 15636 triangles  with 0 duplicate removed
Adding 334 nodes
Adding 665 edges with 331 duplicates removed
Adding 332 triangles  with 0 duplicate removed
Adding 162 nodes
Adding 321 edges with 159 duplicates removed
Adding 160 triangles  with 0 duplicate removed
Contact bot in region air with 334 nodes
Contact top in region air with 162 nodes
Region "air" on device "MyDevice" has equations 0:8280
Region "m1" on device "MyDevice" has no equations.
Region "m2" on device "MyDevice" has no equations.
Device "MyDevice" has equations 0:8280
number of equations 8281
Iteration: 0
  Device: "MyDevice" RelError: 1.00000e+00 AbsError: 1.00000e+00
    Region: "air" RelError: 1.00000e+00 AbsError: 1.00000e+00
      Equation: "PotentialEquation" RelError: 1.00000e+00 AbsError: 1.00000e+00
Iteration: 1
  Device: "MyDevice" RelError: 1.25144e-12 AbsError: 1.73395e-13
    Region: "air" RelError: 1.25144e-12 AbsError: 1.73395e-13
      Equation: "PotentialEquation" RelError: 1.25144e-12 AbsError: 1.73395e-13
3.35017166004e-12
-3.35017166004e-12
```

A visualization of the results is shown in Figure 14.1.

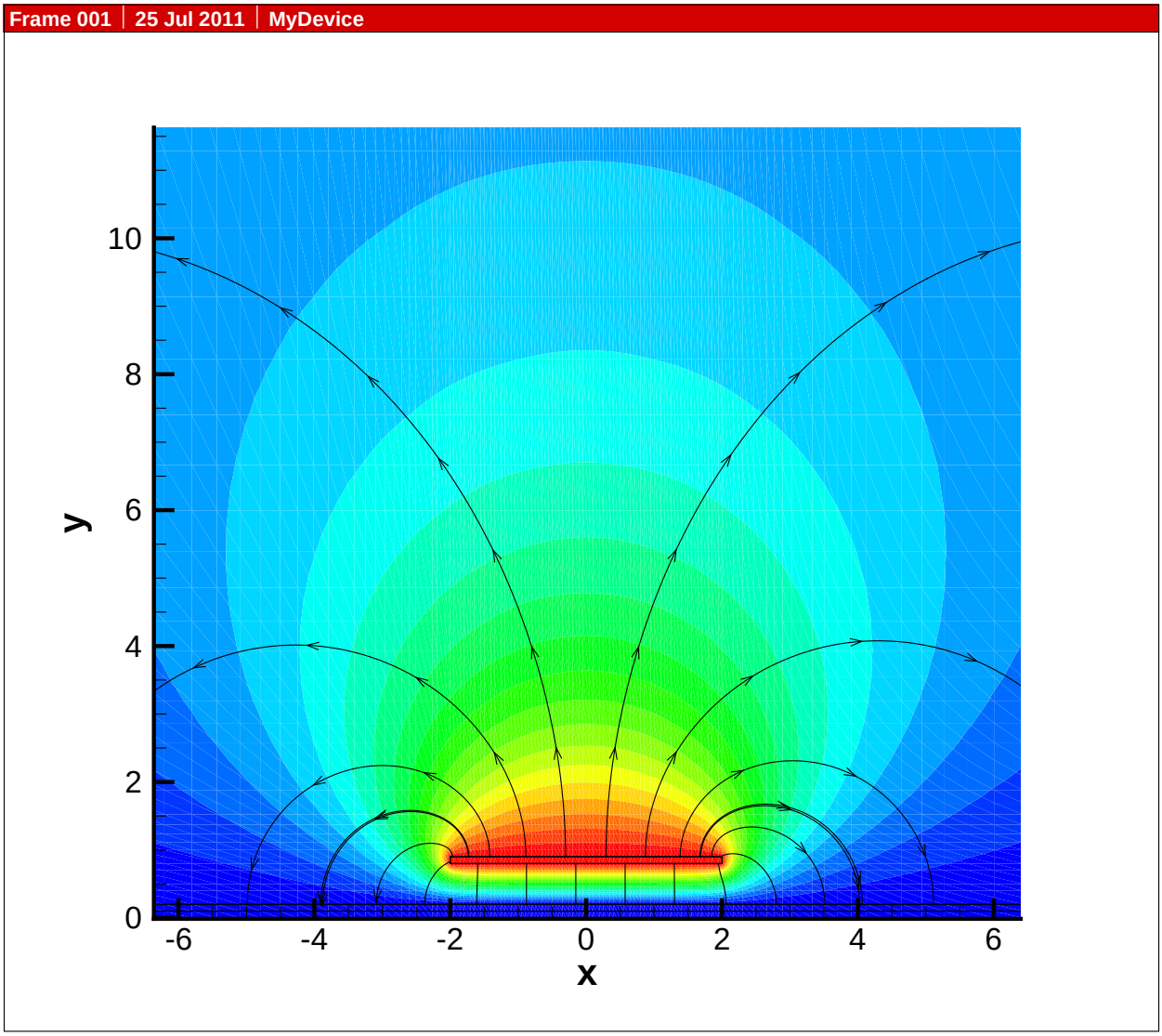

Figure 14.1: Capacitance simulation result. The coloring is by `Potential`, and the stream traces are for components of `ElectricField`.

Chapter 15

Diode

The diode examples are located in the `examples/diode`. They demonstrate the use of packages located in the `python_packages` directory to simulate drift-diffusion using the Scharfetter-Gummel method [8].

15.1 1D diode

15.1.1 Using the python packages

For these examples, python modules are provided to supply the appropriate model and parameter settings. A listing is shown in Table 15.1. The `python_packages` must be in your path, and may be set using the methods described in *Section 8.2.3,* Other packages *on page 60*. The example files in the DEVSIM distribution set the path properly when loading modules.

For this example, `diode_1d.py`, the following line is used to import the relevant physics.

```
from ds import *
from simple_physics import *
```

15.1.2 Creating the mesh

This creates a mesh 10^{-6} cm long with a junction located at the midpoint. The name of the device is `MyDevice` with a single region names `MyRegion`. The contacts on either end are called `top` and `bot`.

```
def createMesh(device, region):
  create_1d_mesh(mesh="dio")
  add_1d_mesh_line(mesh="dio", pos=0, ps=1e-7, tag="top")
  add_1d_mesh_line(mesh="dio", pos=0.5e-5, ps=1e-9, tag="mid")
  add_1d_mesh_line(mesh="dio", pos=1e-5, ps=1e-7, tag="bot")
  add_1d_contact  (mesh="dio", name="top", tag="top", material="metal")
  add_1d_contact  (mesh="dio", name="bot", tag="bot", material="metal")
  add_1d_region   (mesh="dio", material="Si", region=region, tag1="top", tag2="bot")
  finalize_mesh(mesh="dio")
  create_device(mesh="dio", device=device)
```

model_create	Creation of models and their derivatives
ramp	Ramping bias and automatic stepping
simple_dd	Functions for calculating bulk electron and hole current
simple_physics	Functions for setting up device physics

Table 15.1: Python package files.

```
device="MyDevice"
region="MyRegion"

createMesh(device, region)
```

15.2 Physical Models and Parameters

```
####
#### Set parameters for 300 K
####
SetSiliconParameters(device, region, 300)
set_parameter(device=device, region=region, name="taun", value=1e-8)
set_parameter(device=device, region=region, name="taup", value=1e-8)

####
#### NetDoping
####
CreateNodeModel(device, region, "Acceptors", "1.0e18*step(0.5e-5-x)")
CreateNodeModel(device, region, "Donors",    "1.0e18*step(x-0.5e-5)")
CreateNodeModel(device, region, "NetDoping", "Donors-Acceptors")
print_node_values(device=device, region=region, name="NetDoping")

####
#### Create Potential, Potential@n0, Potential@n1
####
CreateSolution(device, region, "Potential")

####
#### Create potential only physical models
####
CreateSiliconPotentialOnly(device, region)

####
#### Set up the contacts applying a bias
####
for i in get_contact_list(device=device):
  set_parameter(device=device, name=GetContactBiasName(i), value=0.0)
  CreateSiliconPotentialOnlyContact(device, region, i)
```

```
####
#### Initial DC solution
####
solve(type="dc", absolute_error=1.0, relative_error=1e-12, maximum_iterations=30)

####
#### drift diffusion solution variables
####
CreateSolution(device, region, "Electrons")
CreateSolution(device, region, "Holes")

####
#### create initial guess from dc only solution
####
set_node_values(device=device, region=region, name="Electrons", init_from="IntrinsicElectrons")
set_node_values(device=device, region=region, name="Holes",     init_from="IntrinsicHoles")

###
### Set up equations
###
CreateSiliconDriftDiffusion(device, region)
for i in get_contact_list(device=device):
  CreateSiliconDriftDiffusionAtContact(device, region, i)

###
### Drift diffusion simulation at equilibrium
###
solve(type="dc", absolute_error=1e10, relative_error=1e-10, maximum_iterations=30)

####
#### Ramp the bias to 0.5 Volts
####
v = 0.0
while v < 0.51:
  set_parameter(device=device, name=GetContactBiasName("top"), value=v)
  solve(type="dc", absolute_error=1e10, relative_error=1e-10, maximum_iterations=30)
  PrintCurrents(device, "top")
  PrintCurrents(device, "bot")
  v += 0.1

####
#### Write out the result
####
write_devices(file="diode_1d.dat", type="tecplot")
```

15.2.1 Plotting the result

A plot showing the doping profile and carrier densities are shown in Figure 15.1. The potential and electric field distribution is shown in Figure 15.2. The current distributions are shown in Figure 15.3.

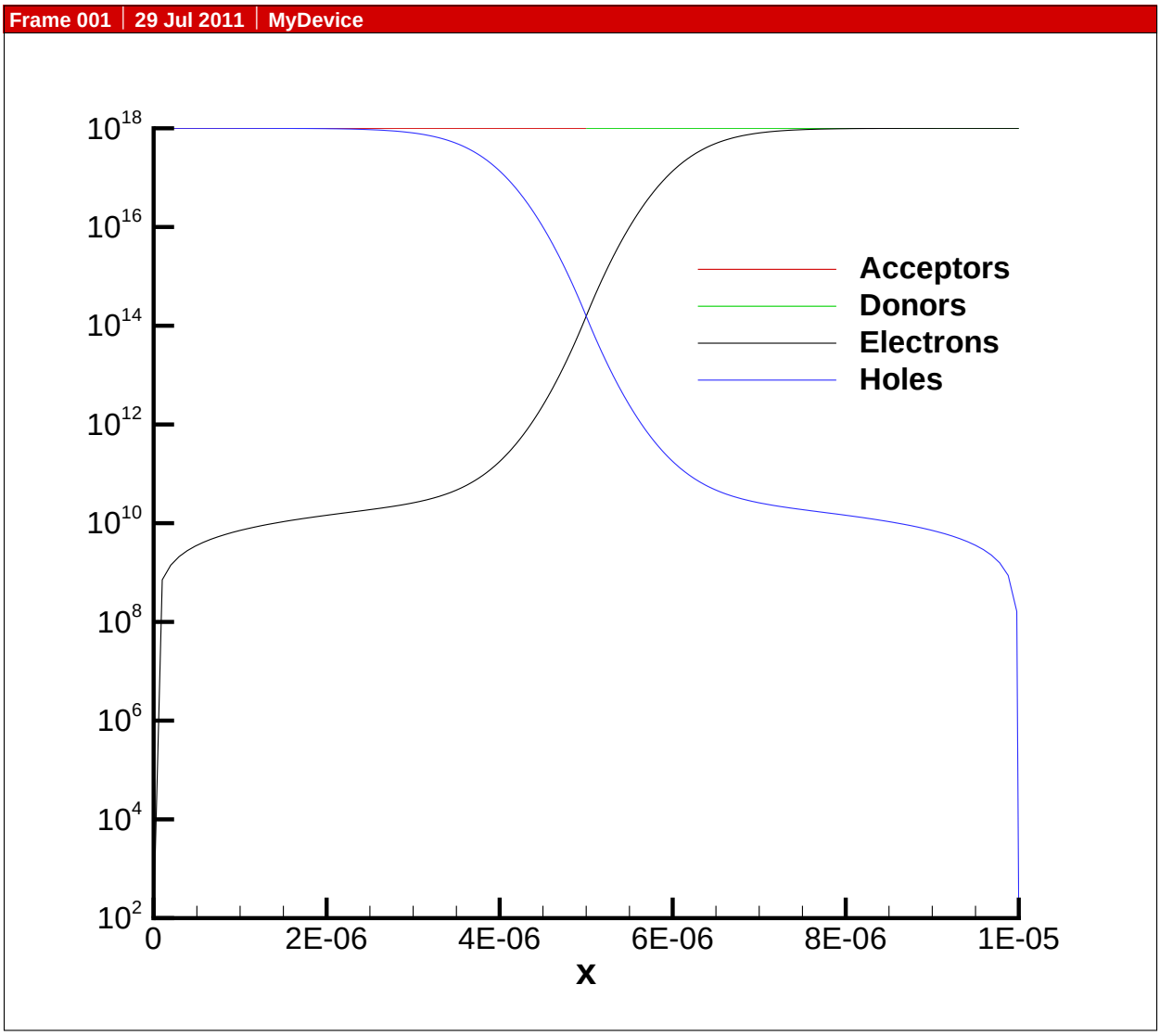

Figure 15.1: Carrier density versus position in 1D diode.

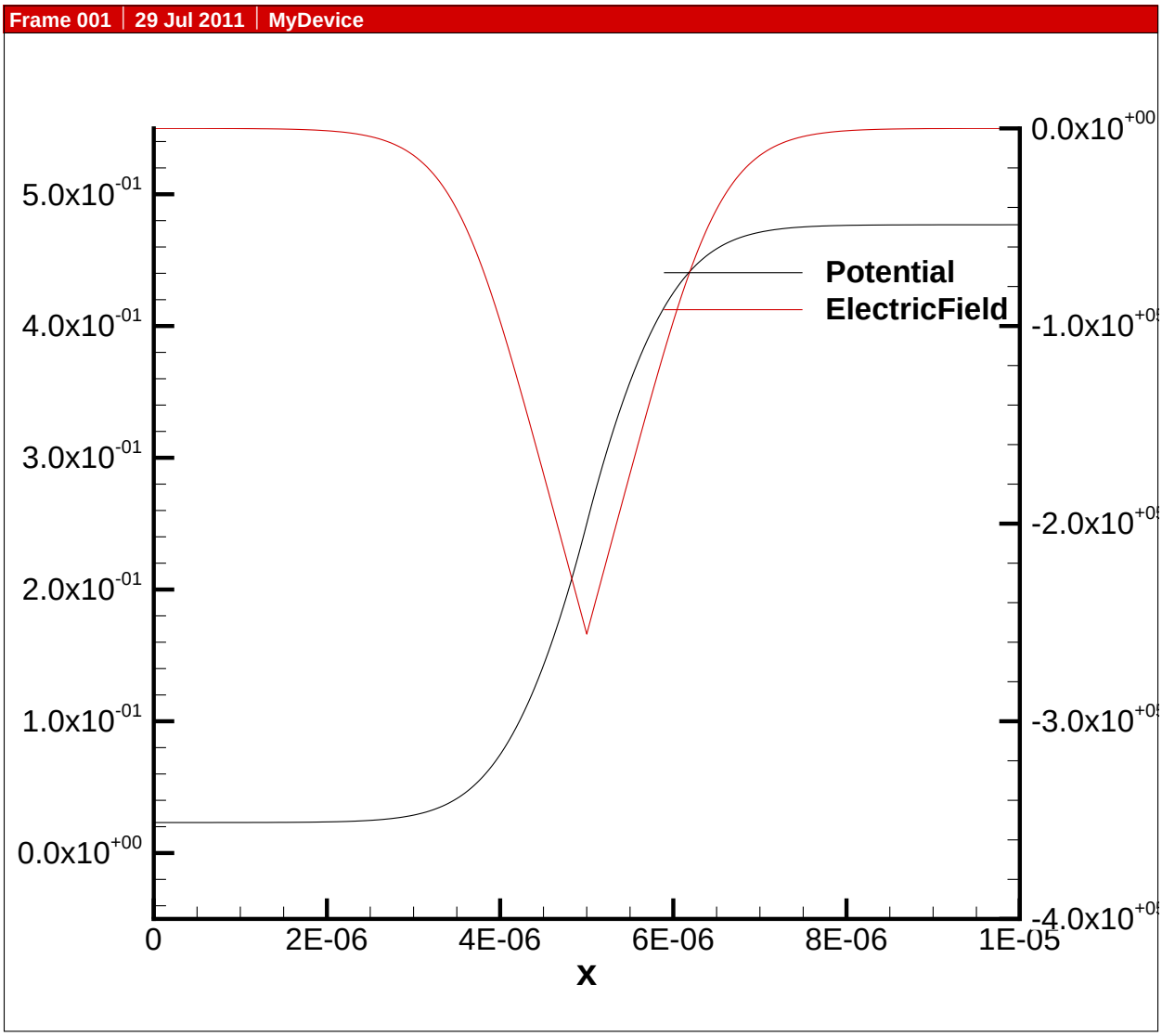

Figure 15.2: Potential and electric field versus position in 1D diode.

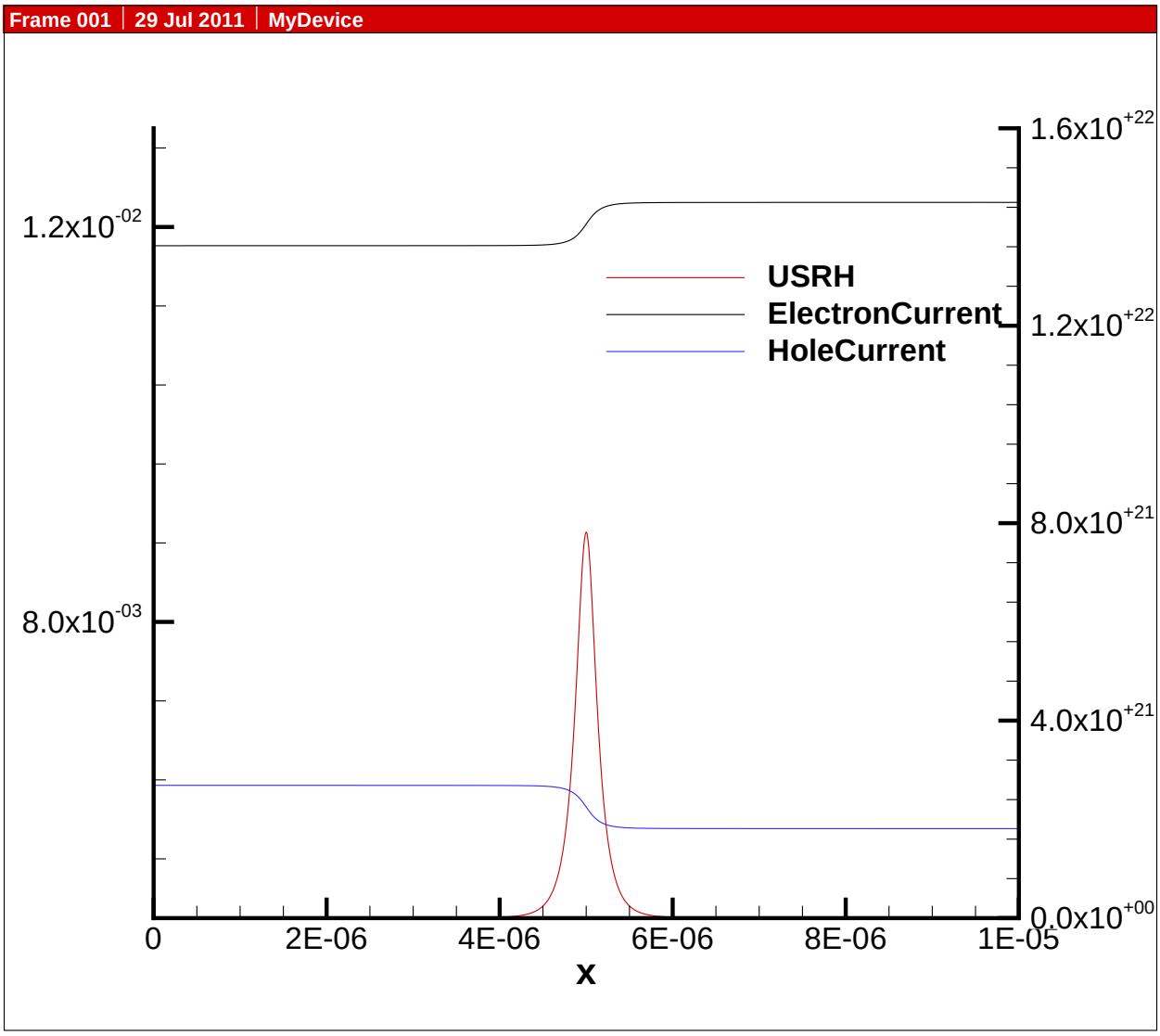

Figure 15.3: Electron and hole current and recombination.

Bibliography

[1] Free Software Foundation, “GNU Lesser General Public License Version 3.” 7

[2] Apache Software Foundation, “Apache License, Version 2.0.” 7

[3] R. S. Muller, T. I. Kamins, and M. Chan, *Device Electronics for Integrated Circuits*. John Wiley & Sons, 3 ed., 2002. 12

[4] “Python programming language — official website.” `http://www.python.org`. 59

[5] J. K. Ousterhout, “Scripting: Higher level programming for the 21st century,” vol. 31, pp. 23–30, 1998. 59

[6] C. Geuzaine and J.-F. Remacle, “Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities,” *International Journal for Numerical Methods in Engineering*, 2009. 75

[7] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, “A supernodal approach to sparse partial pivoting,” *SIAM J. Matrix Analysis and Applications*, vol. 20, no. 3, pp. 720–755, 1999. 77

[8] D. L. Scharfetter and H. K. Gummel, “Large-signal analysis of a silicon Read diode oscillator,” vol. ED-16, pp. 64–77, Jan. 1969. 93

Index

