DEVSIM Manual

Release 1.0.0

DEVSIM LLC

Contents

List of Figures

List of Tables

Chapter 1

Front Matter

1.1 Contact

Table 1.1: Contact

Web:	https://devsim.com
Email:	info@devsim.com
Open Source Project:	https://devsim.org
Online Documentation:	https://devsim.net
Online Forum:	https://groups.google.com/d/forum/devsim

1.2 Copyright

1.3 Documentation License

1.4 Disclaimer

1.5 Trademark

DEVSIM is a registered trademark and SYMDIFF is a trademark of DEVSIM LLC. All other product or company names are trademarks of their respective owners.

Chapter 2

Release Notes

2.1 Introduction

DEVSIM download and installation instructions are located in *Installation* (page 53). The following sections list bug fixes and enhancements over time. Contact information is listed in *Contact* (page 1).

2.2 1.0.0 (December 18, 2018)

2.2.1 Documentation

The formatting of the PDF and online documentation has been improved. Also significant changes have been made to the way DEVSIM is called from Python.

2.2.2 Version

Due to the numerous changes in the Python API, the version number has been updated to having a major revision of 1. We adopt the semantic version numbering presented at https://semver.org. The version number can be accessed through the Python interface using the devsim.__version__ variable.

2.2.3 Operating Systems

The Microsoft Windows 32-bit operating system is now supported in addition to the platforms listed in *Supported platforms* (page 53).

2.2.4 Python Support

DEVSIM is now loaded as a shared library from any compatible Python interpreter. Previously, DEVSIM binaries contained an embedded Python interpreter. The following versions of Python are supported in this release

- 2.7
- 3.6
- 3.7

By first setting the PYTHONPATH variable to the lib directory in the DEVSIM distribution, devsim is loaded by using

```
import devsim
```

from Python. Previous releases of devsim used the ds module, the manual will be updated to reflect the change in module name.

Many of the examples in the distribution rely on the python_packages module, which is available by using:

```
import devsim.python_packages
```

The default version of Python for use in scripts is Python 3.7. Scripts written for earlier versions of Python 3 should work. Python 2.7 is deprecated for future development.

Anaconda Python 3.7 is the recommended distribution and is available from https://continuum.io. The Intel Math Kernel Library is required for the official DEVSIM releases. These may be installed in Anaconda using the following command:

```
conda install mkl
```

On the Microsoft Windows platform, the following packages should also be installed:

```
conda install sqlite zlib
```

Some of the examples and tests also use numpy, which is available using:

```
conda install numpy
```

Please see *User Interface* (page 41) and *Installation* (page 53) for more information.

2.2.5 GMSH Support

Gmsh has announced a new version of their mesh format 4.0. DEVSIM currently supports the previous version, 2.2. To load a file from Gmsh, it is now necessary to either:

- Save the file in the 2.2 format from Gmsh
- Parse the 4.0 file, and then use *Custom mesh loading using scripting* (page 38)

A future release of DEVSIM will provide this capability.

2.2.6 CGNS Support

Support for loading `CGNS` files is deprecated, and is no longer part of the official releases. Please see *Using an external mesher* (page 36) for more information about importing meshes from other tools.

2.3 July 20, 2018

2.3.1 Documentation

The documentation has a new license, which is described in *Copyright* (page 1). The source files are now available for download from: https://github.com/devsim/devsim_documentation.

2.3.2 Python 3 Support

Python 3 executable, `devsim_py3` is now supplied in addition to standard Python 2 executable, `devsim`.

2.3.3 Element Information

The `devsim.get_element_node_list()` retrieves a list of nodes for every element on a `region`, `contact`, or `interface`.

2.3.4 Interface Boundary Condition

The `type=hybrid` option is now available for the `devsim.interface_equation()` command. Please see *Interface equation assembly* (page 26) for information about boundary conditions.

2.3.5 Interace Equation Coupling

The `name0`, and `name1` options are now available for the `devsim.interface_equation()` command. They make it possible to couple dissimilar equation names across regions.

2.3.6 Interface and Contact Surface Area

Contact surface area is no longer included in `SurfaceArea` node model. It is now placed in `ContactSurfaceArea`. These are listed in Table 4.1.

2.3.7 Bug Fixes

- The `devsim.interface_equation()` command is fixed for `type=fluxterm` boundary conditions on the interface.
- The `devsim.get_material()`, and `devsim.set_material()` handle the `contact` option.

- Interface equation assembly skips nodes when an interface node is shared with a contact.

2.3.8 Extended Precision

The following new parameters are available:

- `extended_solver`, extended precision matrix for Newton and linear Solver
- `extended_model`, extended precision model evaluation
- `extended_equation`, extended precision equation assembly

When compiled with 128-bit extended precision support, these options enable calculations to be performed with higher precision. Default geometric models, are also calculated with extended precision.

```
devsim.set_parameter(name = "extended_solver", value=True)
devsim.set_parameter(name = "extended_model", value=True)
devsim.set_parameter(name = "extended_equation", value=True)
```

Currently, the `Linux` and gcc-based `Apple macOS` versions have extended precision support.

2.4 May 15, 2017

2.4.1 Platforms

- The `Ubuntu 16.04 (LTS)` platform is now supported.
- The `Ubuntu 12.04 (LTS)`, `Centos 5` (`Red Hat 5` compatible) platforms are no longer supported. These platforms are no longer supported by their vendors.
- `Apple macOS` compiled with `flat_namespace` to allow substitution of dynamically linked libraries.
- `Microsoft Windows 7` is compiled using Microsoft Visual Studio 2017.

2.4.2 Binary Releases

- Releases available from https://github.com/devsim/devsim/releases.
- `Centos 6` released is linked against the `Intel Math Kernel Library`.
- `Microsoft Windows 7` release is linked against the `Intel Math Kernel Library`
- `Apple macOS` can optionally use the `Intel Math Kernel Library`.
- `Anaconda Python 2.7` is the recommended distribution.
- Please see release notes for more information.

2.4.3 Bug Fixes

- 3D element edge derivatives were not being evaluated correctly
- 3D equation model evaluation for element edge models

2.4.4 Enhancements

- Build scripts are provided to build on various platforms.
- `DEVSIM` mesh format stores elements, instead of just nodes, for contact and interfaces
- The `devsim.create_gmsh_mesh()` command can be used to create a device from a provided list of elements.

2.4.5 Example Availability

- BJT simulation example available from https://github.com/devsim/devsim_bjt_example.

2.5 February 6, 2016

`DEVSIM` is now covered by the Apache License, Version 2.0 [ApacheSoftwareFoundation]. Please see the `NOTICE` and `LICENSE` file for more information.

2.6 November 24, 2015

2.6.1 Python Help

The `Python` interpreter now has documentation for each command, derived from the documentation in the manual. For example, help for the `devsim.solve()` can be found using:

```
help("devsim.solve")
```

2.6.2 Manual Updates

The manual has been updated so that commands are easier to find in the index. Every command now has a short description. Cross references have been fixed. The date has been added to the front page.

2.7 November 1, 2015

2.7.1 Convergence Info

The `devsim.solve()` now supports the `info` option. The solve command will then return convergence information.

2.7.2 Python Interpreter Changes

The way `DEVSIM` commands are loaded into the `devsim` module has been changed. It is now possible to see the full list of `DEVSIM` commands by typing

```
help('devsim')
```

in the `Python` interpreter.

2.7.3 Platform Improvements and Binary Availability

Many improvements have been made in the way binaries are generated for the `Linux`, `Apple macOS`, and `Microsoft Windows` platforms.

For `Linux` (see `linux.txt`):

- Create `Centos 5`, (`Red Hat Enterprise Linux 5` compatible) build
- Build uses `Intel Math Kernel Library` math libraries (community edition)
- Build uses any compatible `Python 2.7`, including `Anaconda`
- Build compatible with newer `Linux` distributions.

For `Apple macOS` (see `macos.txt`):

- Uses the system `Python 2.7` on `macOS 10.10 (Yosemite)`
- Provide instructions to use `Anaconda Python`

For `Microsoft Windows` (see `windows.txt`):

- Uses any compatible `Python 2.7`, including `Anaconda`
- Build uses `Intel Math Kernel Library` Community Edition

Binary releases are available for these platforms at https://devsim.org.

2.8 September 6, 2015

The `devsim.set_node_values()` takes a new option, `values`. It is a list containing values to set for all of the nodes in a region.

The following new commands have been added:

- devsim.get_equation_list()
- devsim.get_contact_equation_list()
- devsim.get_interface_equation_list()
- devsim.delete_equation()
- devsim.delete_contact_equation()
- devsim.delete_interface_equation()
- devsim.get_equation_command()
- devsim.get_contact_equation_command()
- devsim.get_interface_equation_command()

2.9 August 10, 2015

The devsim.create_contact_from_interface() may be used to create a contact at the location of an interface. This is useful when contact boundary conditions are needed for a region connected to the interface.

2.10 July 16, 2015

The devsim.set_node_value() was not properly setting the value. This issue is now resolved.

2.11 June 7, 2015

The devsim.equation() now suppports the edge_volume_model. This makes it possible to integrate edge quantities properly so that it is integrated with respect to the volume on nodes of the edge. To set the node volumes for integration, it is necessary to define a model for the node volumes on both nodes of the edge. For example:

```
devsim.edge_model(device="device", region="region", name="EdgeNodeVolume",
  equation="0.5*EdgeCouple*EdgeLength")
set_parameter(name="edge_node0_volume_model", value="EdgeNodeVolume")
set_parameter(name="edge_node1_volume_model", value="EdgeNodeVolume")
```

For the cylindrical coordinate system in 2D, please see *Cylindrical Coordinate Systems* (page 29).

macOS 10.10 (Yosemite) is now supported. Regression results in the source distribution are for a 2014 Macbook Pro i7 running this operating system.

2.12 October 4, 2014

2.12.1 Platform Availability

The software is now supported on the `Microsoft Windows`. Please see *Supported platforms* (page 53) for more information.

2.13 December 25, 2013

2.13.1 Binary Availability

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

- `macOS 10.10 (Yosemite)`
- `Red Hat Enterprise Linux 6`
- `Ubuntu 12.04 (LTS)`

Please see *Installation* (page 53) for more information.

2.13.2 Platforms

`macOS 10.10 (Yosemite)` 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`. Building and running on these platforms will still be supported.

2.13.3 Source code improvements

The source code has been improved to compile on `macOS 10.10 (Yosemite)` 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.

2.14 September 8, 2013

2.14.1 Convergence

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

2.14.2 Bernoulli Function Derivative Evaluation

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

2.14.3 Default Edge Model

The `edge_index` is now a default edge models created on a region Table 4.2.

2.15 August 14, 2013

2.15.1 `SYMDIFF` functions

The `vec_max` and `vec_min` functions have been added to the `SYMDIFF` parser (Table 10.2). The `vec_sum` function replaces `sum`.

2.15.2 Default Node Models

The `coordinate_index` and `node_index` are now part of the default node models created on a region (Table 4.1}).

2.15.3 Set Node Value

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

2.15.4 Fix Edge Average Model

Fixed issue with `devsim.edge_average_model()` during serialization to the `DEVSIM` format.

2.16 July 29, 2013

2.16.1 `DEVSIM` is open source

`DEVSIM` is now an open source project and is available from https://github.com/devsim/devsim. License information may be found in *DEVSIM License* (page 55). If you would like to participate in this project or need support, please contact us using the information in *Contact* (page 1). Installation instructions may be found in *Installation* (page 53).

2.16.2 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.

2.16.3 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.

2.16.4 External Meshing

Please see *Using an external mesher* (page 36) 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 *Genius* (page 37).

`Gmsh` **Mesh Import** `DEVSIM` reads version `2.1` and `2.2` meshes from `Gmsh`. Version `2.0` is no longer supported. Please see *Gmsh* (page 37) for more information.

2.16.5 Math Functions

The `acosh`, `asinh`, `atanh`, are now available math functions. Please see Table 10.2.

2.16.6 Test directory structure

Platform specific results are stored in a hierarchical fashion.

Chapter 3

Introduction

3.1 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 https://devsim.org. The source code is available under the terms of the Apache License Version 2.0 [ApacheSoftwareFoundation]. Examples are released under the Apache License Version 2.0 [ApacheSoftwareFoundation]. Contributions to this project are welcome in the form of bug reporting, documentation, modeling, and feature implementation.

3.2 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 has a scripting language interface (*User Interface* (page 41)). This provides control structures and language syntax in a consistent and intuitive manner. 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 (*SYMDIFF* (page 45)) 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.

3.3 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 An `interface` connects two regions together. At the interfaces, equations are specified to account for how the flux in each device region crosses the region boundary.

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

3.4 Equation assembly

Equation assembly of models is discussed in *Equation and Models* (page 17).

3.5 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 *Parameters* (page 31).

3.6 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 *Circuits* (page 33).

3.7 Meshing

Meshing is discussed in *Meshing* (page 35).

3.8 Analysis

`DEVSIM` offers a range of simulation algorithms. They are discussed in more detail in *Solver* (page 39).

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.

3.9 Scripting interface

The scripting interface to DEVSIM is discussed in *User Interface* (page 41).

3.10 Expression parser

The expression parser is discussed in *SYMDIFF* (page 45).

3.11 Visualization and postprocessing

Visualization is discussed in *Visualization* (page 51).

3.12 Installation

Installation is discussed in *Installation* (page 53).

3.13 Additional information

Additional information is discussed in *Additional Information* (page 55).

3.14 Examples

Examples are discussed in the remaining chapters beginning with *Example Overview* (page 61).

Chapter 4

Equation and Models

4.1 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:

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

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

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

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

In Fig. 4.1, 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 *SYMDIFF* (page 45). For special situations, custom matrix assembly is also available and is discussed in *Custom matrix assembly* (page 28).

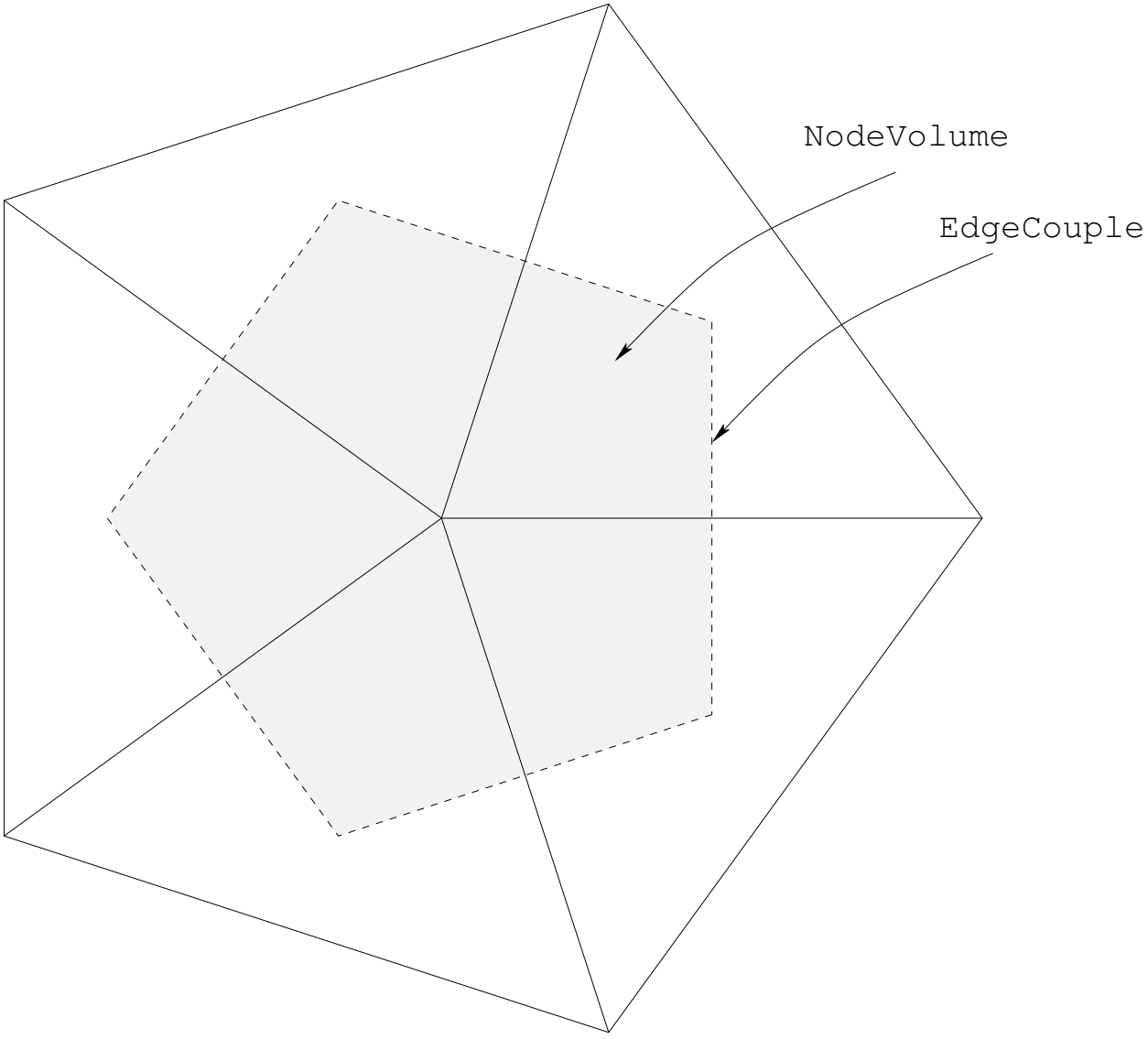

Fig. 4.1: Mesh elements in 2D.

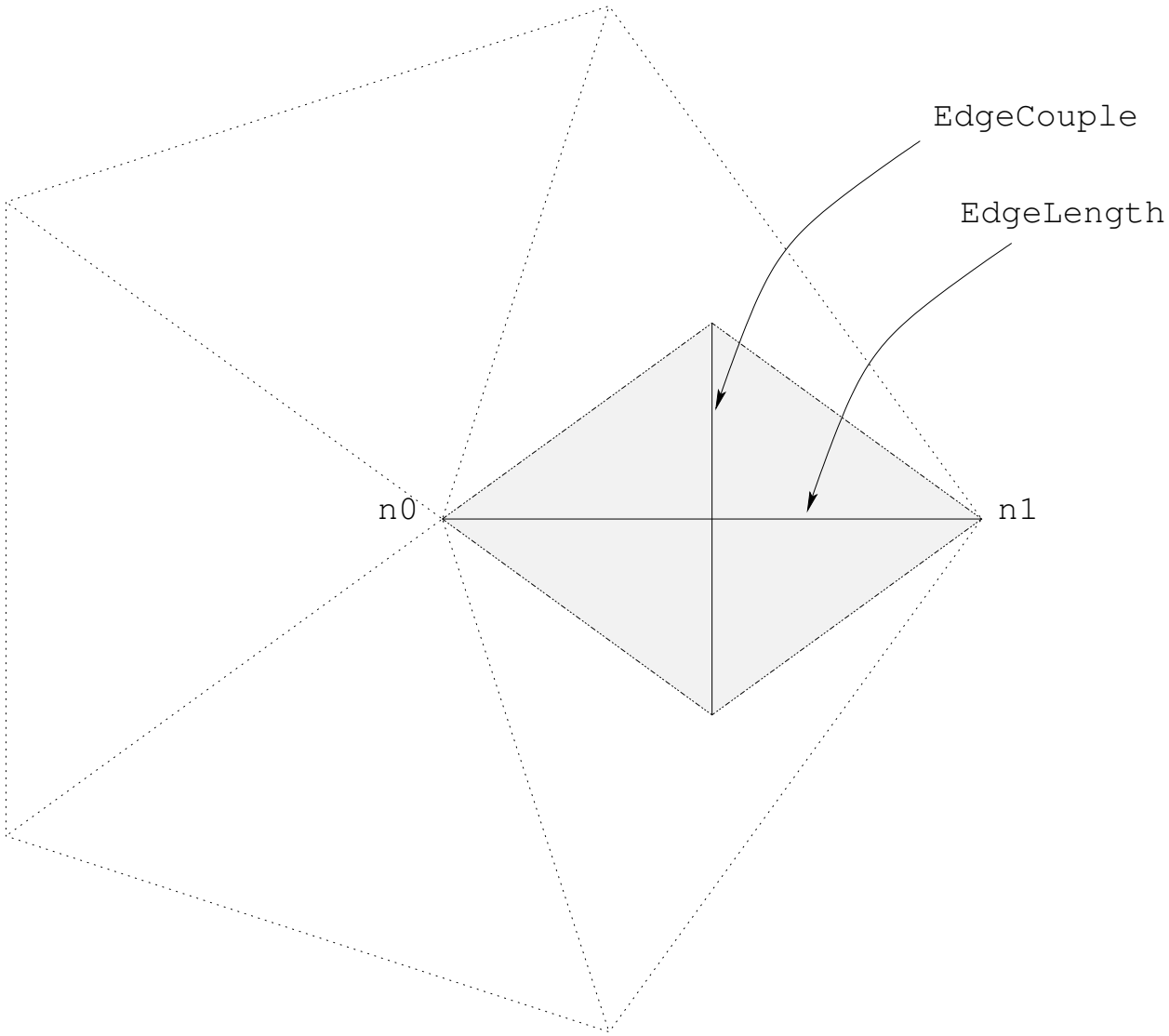

Fig. 4.2: Edge model constructs in 2D.

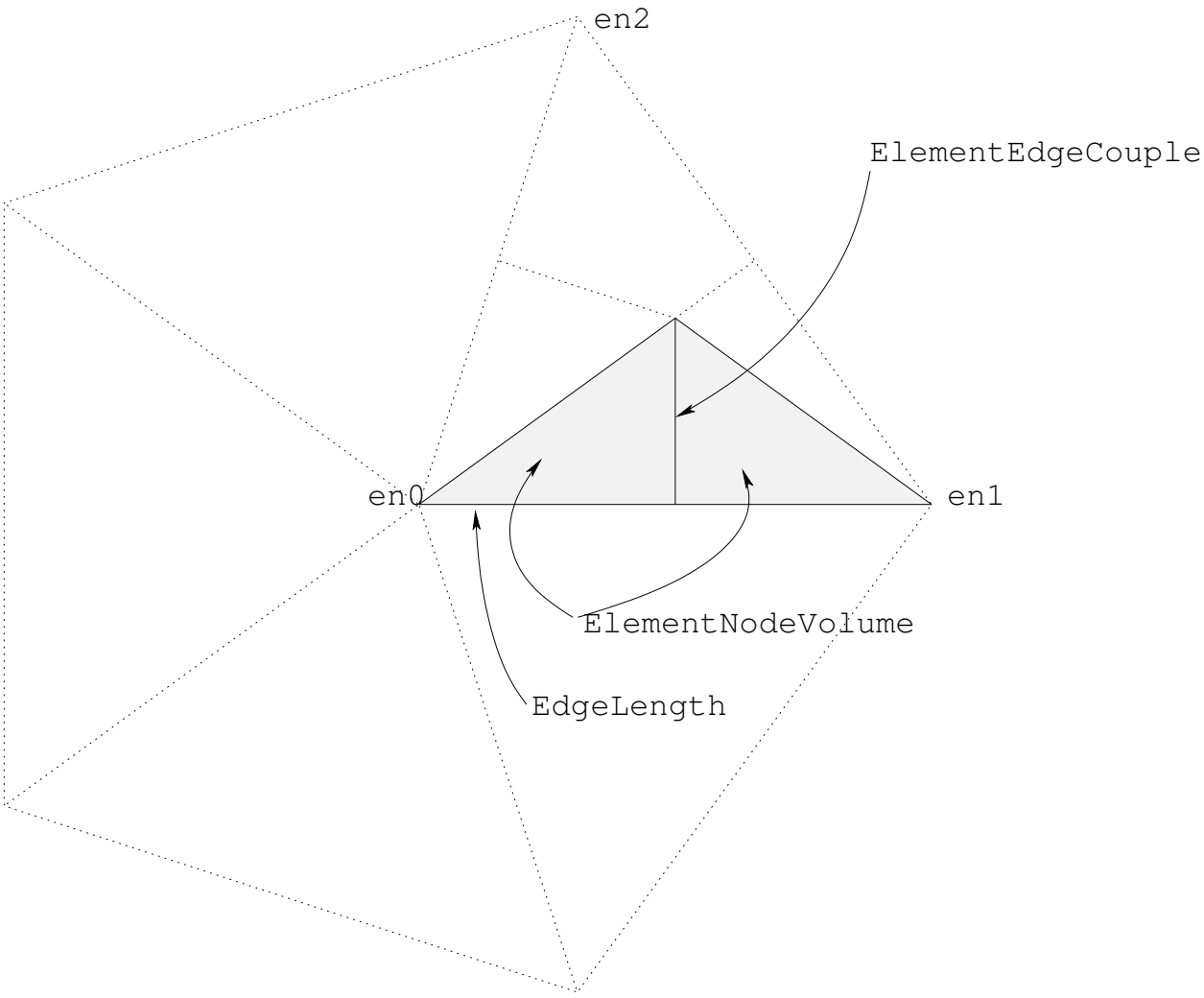

Fig. 4.3: Element edge model constructs in 2D.

4.2 Bulk models

4.2.1 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 4.1.

In this example, we present an implementation of Shockley Read Hall recombination [MKC02].

```
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
devsim.node_model(device='MyDevice', region='MyRegion',
  name="USRH", equation=USRH)
devsim.node_model(device='MyDevice', region='MyRegion',
  name="USRH:Electrons", equation=dUSRHdn)
devsim.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 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.

Table 4.1: Node models defined on each region of a device.

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 nodes, otherwise 0
`ContactSurfaceArea`	The surface area of a node on 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

4.2.2 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 Fig. 4.2, 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:

```
devsim.edge_model(device="device", region="region", name="ElectricField",
           equation="(Potential@n0 - Potential@n1)*EdgeInverseLength")
devsim.edge_model(device="device", region="region",
           name="ElectricField:Potential@n0", equation="EdgeInverseLength")
devsim.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 `devsim.edge_from_node_model()`. In addition, the `devsim.edge_average_model()` can be used to create edge models in terms of node model quantities.

Edge models automatically created for a region are listed in Table 4.2.

Table 4.2: Edge 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)

4.2.3 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 Fig. 4.3, 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 4.3.

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

Table 4.3: Element edge models defined on each region of a device.

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.

4.2.4 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 4.4.

Table 4.4: 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.

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)

4.2.5 Conversions between model types

The `devsim.edge_from_node_model()` 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 `devsim.edge_average_model()` 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 `devsim.element_from_edge_model()` 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 `devsim.element_from_node_model()` is used to create element edge models referring to each node on the element of the element edge.

4.2.6 Equation assembly

Bulk equations are specified in terms of the node, edge, and element edge models using the `devsim.equation()`. 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.

```
devsim.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.

Table 4.5: 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.

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`

4.3 Interface

4.3.1 Interface models

Fig. 4.4 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 `devsim.interface_model()`. Interface models may 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 4.5.

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

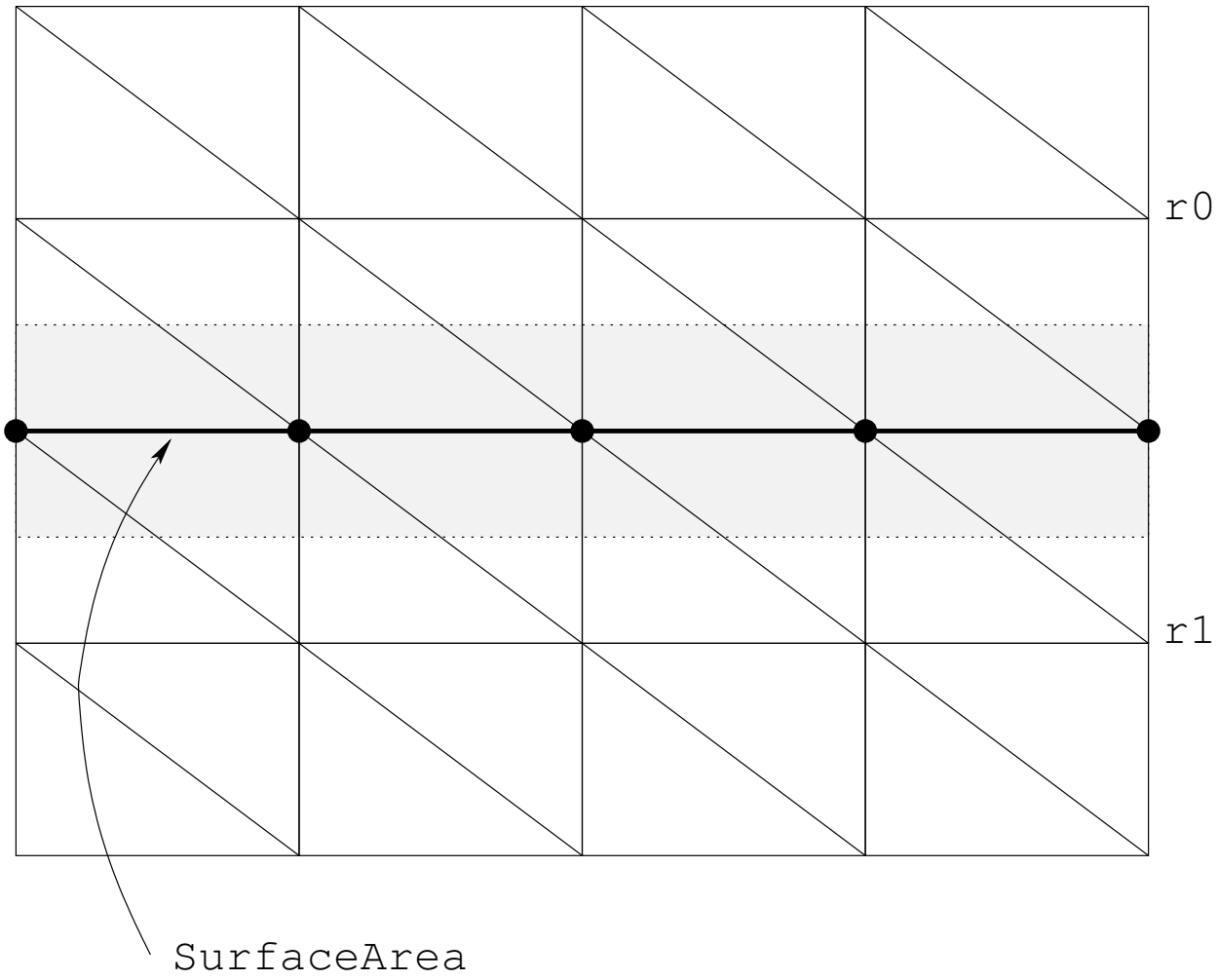

Fig. 4.4: Interface constructs in 2D. Interface node pairs are located at each •. The `SurfaceArea` model is used to integrate flux term models.

4.3.2 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`

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

4.3.3 Interface equation assembly

There are three types of interface equations considered in `DEVSIM`. They are both activated using the `devsim.interface_equation()`.

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.

```
devsim.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.

In the third form, `hybrid`, equations for nodes on both sides of the interface are added into the equation for the node in the first region. The equation for the node on the second interface is integrated in the second region, and the fluxterm is subracted in the second region.

4.4 Contact

4.4.1 Contact models

Fig. 4.5 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 `devsim.contact_node_model()` and the `devsim.contact_edge_model()`, respectively. The key difference is that the models are only evaluated on the contact nodes for the contact specified.

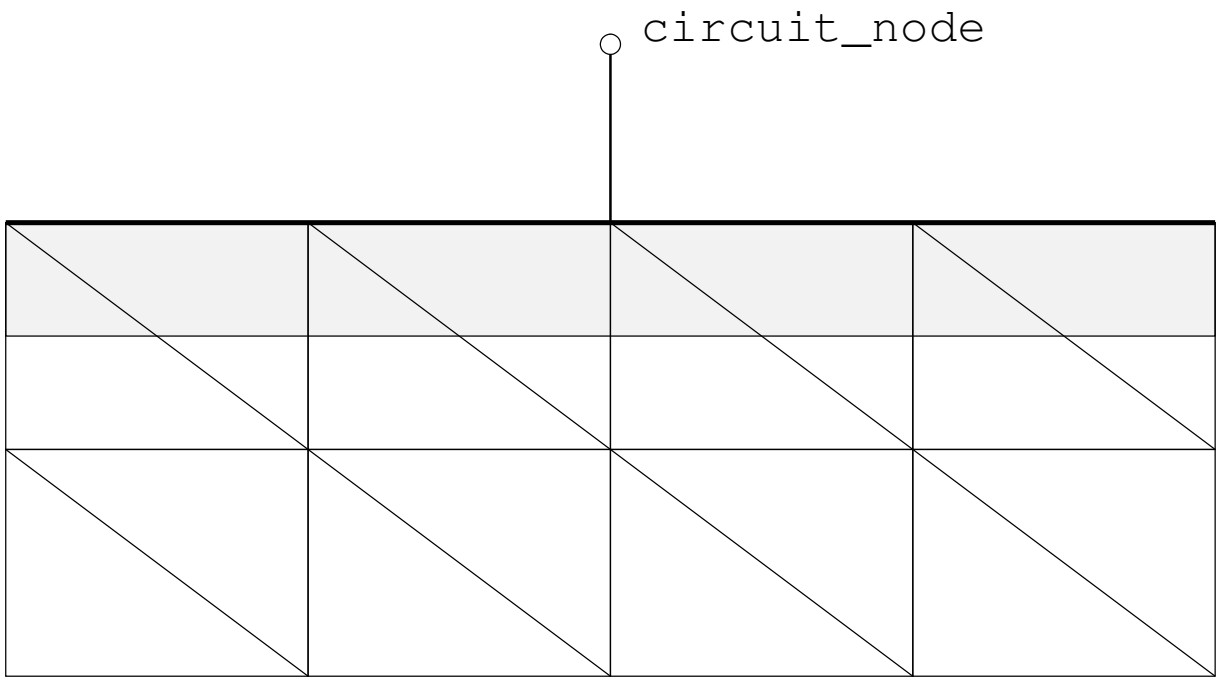

Fig. 4.5: Contact constructs in 2D.

4.4.2 Contact model derivatives

The derivatives are equivalent to the discussion in *Model derivatives* (page 23). 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`.

4.4.3 Contact equation assembly

The `devsim.contact_equation()` 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

```
devsim.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

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

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

4.5 Custom matrix assembly

The `devsim.custom_equation()` 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

```
devsim.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 `devsim.get_circuit_equation_number()` may be used to get the equation numbers corresponding to circuit node names. The `devsim.get_equation_numbers()` 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.

4.6 Cylindrical Coordinate Systems

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

- `devsim.cylindrical_edge_couple()`
- `devsim.cylindrical_node_volume()`
- `devsim.cylindrical_surface_area()`

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="edge_node0_volume_model",
  value="CylindricalEdgeNodeVolume@n0")
set_parameter(name="edge_node1_volume_model",
  value="CylindricalEdgeNodeVolume@n1")
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")
```

Chapter 5

Parameters

Parameters can be set using the commands in *Material Commands* (page 59). There are two complementary formalisms for doing this.

5.1 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.

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

They may also be used to control program behavior, as listed in Table 5.1:

Table 5.1: Parameters controlling program behavior.

Parameter	Description
debug_level	info, verbose Section 9.3.5
threads_available	value=1, Section 9.3.6
threads_task_size	value=?, Section 9.3.6
node_volume_model	Section 4.6
edge_couple_model	Section 4.6
edge_node0_volume_model	Section 4.6
edge_node1_volume_model	Section 4.6
element_edge_couple_model	Section 4.6
element_node0_volume_model	Section 4.6
element_node1_volume_model	Section 4.6
extended_solver	value=False Extended precision matrix and RHS assembly and error evaluations. Linear solver and circuit assembly is still double precision‘‘
extended_model	value=False Extended precision model evaluation
extended_equation	value=False Extended precision equation evaluation

5.2 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.

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

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

5.3 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.

Chapter 6

Circuits

6.1 Circuit elements

Circuit elements are manipulated using the commands in *Circuit Commands* (page 59). Using the `devsim.circuit_element()` to add a circuit element will implicitly create the nodes being references.

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

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

6.2 Connecting devices

For devices to contribute current to an external circuit, the `devsim.contact_equation()` should use the `circuitnode` 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 `devsim.circuit_node_alias()` 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.

Chapter 7

Meshing

7.1 1D mesher

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

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

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

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

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

7.2 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 `devsim.add_2d_region()` 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.

```
devsim.create_2d_mesh(mesh="cap")
devsim.add_2d_mesh_line(mesh="cap", dir="y", pos=-0.001, ps=0.001)
devsim.add_2d_mesh_line(mesh="cap", dir="x", pos=xmin, ps=0.1)
devsim.add_2d_mesh_line(mesh="cap", dir="x", pos=xmax, ps=0.1)
devsim.add_2d_mesh_line(mesh="cap", dir="y", pos=ymin, ps=0.1)
devsim.add_2d_mesh_line(mesh="cap", dir="y", pos=ymax, ps=0.1)
devsim.add_2d_mesh_line(mesh="cap", dir="y", pos=+1.001, ps=0.001)
devsim.add_2d_region(mesh="cap", material="gas", region="gas1", yl=-.001,␣
↪yh=0.0)
devsim.add_2d_region(mesh="cap", material="gas", region="gas2", yl=1.0, yh=1.
↪001)
devsim.add_2d_region(mesh="cap", material="Oxide", region="r0", xl=xmin,␣
↪xh=xmax,
  yl=ymid1, yh=ymin)
devsim.add_2d_region(mesh="cap", material="Silicon", region="r1", xl=xmin,␣
↪xh=xmax,
  yl=ymid2, yh=ymid1)
devsim.add_2d_region(mesh="cap", material="Silicon", region="r2", xl=xmin,␣
↪xh=xmax,
  yl=ymid2, yh=ymax)

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

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

7.3 Using an external mesher

`DEVSIM` supports reading meshes from `Gmsh`. Support for `Genius Device Simulator` is deprecated and will be removed from a future release. In addition, meshes may be input directly using the `Python` interface. These meshes may only contain points, lines, triangles, and tetrahedra. Hybrid meshes or uniform meshes containing other elements are not supported at this time.

7.3.1 Genius

Meshes from the Genius Device Simulator software (see *Genius* (page 55)) can be imported using the CGNS format. In this example, devsim.create_genius_mesh() 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 *genius* (page 61).

7.3.2 Gmsh

The Gmsh meshing software (see *Gmsh* (page 55)) 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:

```
devsim.create_gmsh_mesh(file="gmsh_mos2d.msh", mesh="mos2d")
devsim.add_gmsh_region(mesh="mos2d" gmsh_name="bulk", region="bulk",
  material="Silicon")
devsim.add_gmsh_region(mesh="mos2d" gmsh_name="oxide", region="oxide",
```

(continues on next page)

(continued from previous page)

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

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

7.3.3 Custom mesh loading using scripting

It is also possible to arbitrarily load a mesh from a `Python` using the `devsim.create_gmsh_mesh()`. This is explained in the `Notes` section of the command.

7.4 Loading and saving results

The `devsim.write_devices()` 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 `devsim_data` format is used for storing numerical information for use in other programs for analysis. The `devsim.load_devices()` is then used to reload the device data for restarting the simulation.

Chapter 8

Solver

8.1 Solver

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

8.2 DC analysis

A DC analysis is performed using the devsim.solve().

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

8.3 AC analysis

An AC analysis is performed using the devsim.solve(). A circuit voltage source is required to set the AC source.

8.4 Noise/Sensitivity analysis

An noise analysis is performed using the devsim.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 devsim.solve(). 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)

8.5 Transient analysis

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

- BDF1
- TRBDF
- BDF2

Chapter 9

User Interface

9.1 Starting DEVSIM

Refer to *Installation* (page 53) for instructions on how to install DEVSIM. Once installed, DEVSIM may be invoked using the following command

By first setting the PYTHONPATH variable to the lib directory in the DEVSIM distribution, devsim is loaded by using

```
import devsim
```

from Python.

Many of the examples in the distribution rely on the python_packages module, which is available by using:

```
import devsim.python_packages
```

The default version of Python for use in scripts is 3.7, however scripts written for earlier versions of Python 3 should work. Python 2.7 is deprecated for future development.

9.2 Python Language

9.2.1 Introduction

Python is the scripting language employed as the text interface to DEVSIM. Documentation and tutorials for the language are available from [pyt]. A paper discussing the general benefits of using scripting languages may be found in [Ous98].

9.2.2 DEVSIM commands

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

```
from devsim import *
```

For details concerning error handling, please see *Error handling* (page 42).

9.2.3 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.

9.2.4 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. Python 3 interpreters handle UTF-8 encoding well. For the deprecated Python 2 interpreter, it is necessary to put the following line at the beginning of the python script.

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

On some systems, such as Microsoft Windows, it may be necessary to set the following environment variable before running a script containing UTF-8 characters.

```
SET PYTHONIOENCODING=utf-8
```

Care should be taken when using UTF-8 characters in names for visualization using the tools in *Visualization* (page 51), as this character set may not be supported.

9.3 Error handling

9.3.1 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 devsim.error. It may be caught.

9.3.2 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 using the contact information in *Contact* (page 1).

9.3.3 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.

9.3.4 Solver errors

When using the devsim.solve(), 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 devsim.error as msg:
  if msg[0].find("Convergence failure") != 0:
    raise
  #### put code to modify step here.
```

9.3.5 Verbosity

The set_parameter() 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.

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

9.3.6 Parallelization

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

```
devsim.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.

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

The Intel Math Kernel Library is parallelized, the number of thread may be controlled by setting the MKL_NUM_THREADS environment variable.

Chapter 10

SYMDIFF

10.1 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.

10.2 Syntax

10.2.1 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
```

10.2.2 Basic expressions

Table 10.1: Basic expressions involving unary, binary, and logical operators.

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

In Table 10.1, 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.*

10.2.3 Functions

Table 10.2: Predefined Functions.

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

In Table 10.2 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.

10.2.4 Commands

Table 10.3: Commands.

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

Commands are shown in Table 10.3. 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
```

10.2.5 User functions

Table 10.4: Commands for user functions.

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`, ...

Commands for specifying and manipulating user functions are listed in Table 10.4. 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

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

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))
```

10.2.6 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)
```

10.3 Invoking SYMDIFF from DEVSIM

10.3.1 Equation parser

The `devsim.symdiff()` 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. *Model Commands* (page 59) 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)
```

10.3.2 Evaluating external math

The `devsim.register_function()` 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`.

10.3.3 Models

When used withing the model commands discussed in *Model Commands* (page 59), `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 11

Visualization

11.1 Introduction

DEVSIM is able to create files for visualization tools. Information about acquiring these tools are presented in *External Software Tools* (page 55).

11.2 Using Tecplot

The devsim.write_devices() 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")
```

11.3 Using Postmini

The devsim.write_devices() 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")
```

11.4 Using Paraview

The devsim.write_devices() 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.

11.5 Using VisIt

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

11.6 DEVSIM

DEVSIM has several commands for getting information on the mesh. Those related to post processing are described in *Model Commands* (page 59) and *Geometry Commands* (page 59).

See *Loading and saving results* (page 38) for information about loading and saving mesh information to a file.

Chapter 12

Installation

12.1 Availability

Information about the open source version of DEVSIM is available from https://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.

12.2 Supported platforms

DEVSIM is compiled and tested on the platforms in Table 12.1. If you require a version on a different software platform, please contact us.

Table 12.1: Current platforms for DEVSIM.

Platform	Bits	OS Version
Microsoft Windows	32, 64	Microsoft Windows 7, Microsoft Windows 10
Linux	64	Ubuntu 14.04 (LTS), Ubuntu 16.04 (LTS), Red Hat Enterprise Linux 6 (Centos 6 compatible)
Apple macOS	64	macOS 10.13 (High Sierra)

12.3 Binary availability

Compiled packages for the the platforms in Table 12.1 are currently available from https://github.com/devsim/devsim/releases. The prerequisites on each platform are described in the linux.txt, macos.txt, and windows.txt.

12.4 Source code availability

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

12.5 Directory Structure

A DEVSIM directory is created with the following sub directories listed in Table 12.2.

Table 12.2: Directory structure for DEVSIM.

bin	contains the devsim tcl binary
lib/devsim	contains the devsim interpreter modules
lib/devsim/python_packages	contains runtime libraries
doc	contains product documentation
examples	contains example scripts
testing	contains additional examples used for testing

12.6 Running DEVSIM

See *User Interface* (page 41) for instructions on how to invoke DEVSIM.

Chapter 13

Additional Information

13.1 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 Apache License, Version 2.0 [ApacheSoftwareFoundation]. Please see the NOTICE and LICENSE file for more information.

13.2 SYMDIFF

SYMDIFF is available from https://symdiff.org under the terms of the Apache License, Version 2.0 [ApacheSoftwareFoundation].

13.3 External Software Tools

13.3.1 Genius

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

13.3.2 Gmsh

Gmsh [GR09] is available from http://gmsh.info.

13.3.3 Paraview

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

13.3.4 Tecplot

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

13.3.5 VisIt

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

13.4 Library Availablilty

The following tools are used to build DEVSIM.

13.4.1 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.

13.4.2 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.

13.4.3 Python

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

13.4.4 SQlite3

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

13.4.5 SuperLU

SuperLU [DEG+99] is used within DEVSIM and and is available from http://crd-legacy.lbl.gov/~xiaoye/SuperLU:

> Copyright (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy)
>
> All rights reserved.

13.4.6 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.

13.4.7 zlib

`zlib` is an open source compression library available from https://zlib.net.

Chapter 14

Command Reference

14.1 Circuit Commands

Commands are for adding circuit elements to the simulation.

14.2 Equation Commands

Commands for manipulating equations on contacts, interface, and regions

14.3 Geometry Commands

Commands for getting information about the device structure.

14.4 Material Commands

Commands for manipulating parameters and material properties

14.5 Meshing Commands

Commands for reading and writing meshes

14.6 Model Commands

Commands for defining and evaluating models

14.7 Solver Commands

Commands for simulation

Chapter 15

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 *Directory structure for DEVSIM.* (page 54).

The following example directories are contained in the distribution.

15.1 capacitance

These are 1D and 2D capacitor simulations, using the internal mesher. A description of these examples is presented in *Capacitor* (page 63).

15.2 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 *Diode* (page 73).

15.3 bioapp1

This is a biosensor application.

15.4 genius

This directory has examples importing meshes from Genius Device Simulator.

15.5 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 Fig. 15.1.

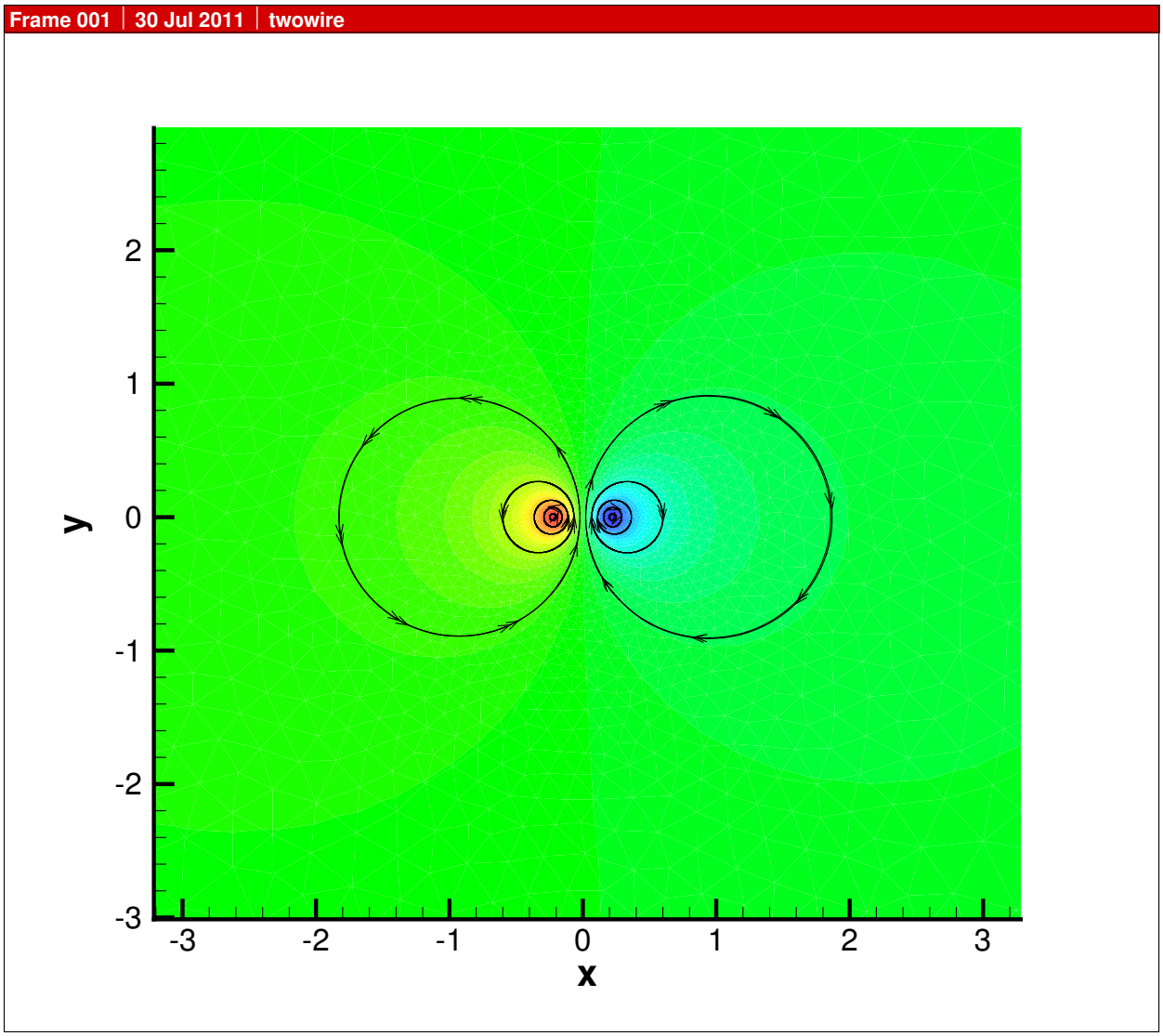

Fig. 15.1: Simulation result for solving for the magnetic potential and field. The coloring is by the Z component of the magnetic potential, and the stream traces are for components of magnetic field.

15.6 mobility

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

Chapter 16

Capacitor

16.1 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 *1D Capacitor* (page 63) 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 *2D Capacitor* (page 67).

16.2 1D Capacitor

16.2.1 Equations

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

$$\epsilon \nabla^2 \psi = 0$$

and the contact boundary equations are

$$\psi_i - V_c = 0$$

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

16.2.2 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 devsim 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)
```

16.3 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)
```

16.3.1 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")
```

16.3.2 Contact boundary conditions

We then create the contact models and equations. We use the `Python for` loop 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,
```

(continues on next page)

(continued from previous page)

```
            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.

16.3.3 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")))
```

16.3.4 Running the simulation

We run the simulation and see the results.

```
capacitance> python cap1d.py
-----------------------------------------

 DEVSIM

 Version: Beta 0.01

 Copyright 2009-2013 Devsim LLC

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

(continues on next page)

(continued from previous page)

```
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.

16.4 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.

16.5 Defining the mesh

```
from devsim 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
```

(continues on next page)

(continued from previous page)

```
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)
```

16.6 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")
```

(continues on next page)

(continued from previous page)

```
###
### 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")
```

(continues on next page)

(continued from previous page)

```
###
### 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")
```

16.7 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")
```

16.8 Running the simulation

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

 DEVSIM

 Version: Beta 0.01

 Copyright 2009-2013 Devsim LLC

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

```

(continues on next page)

(continued from previous page)

```
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 Fig. 16.1.

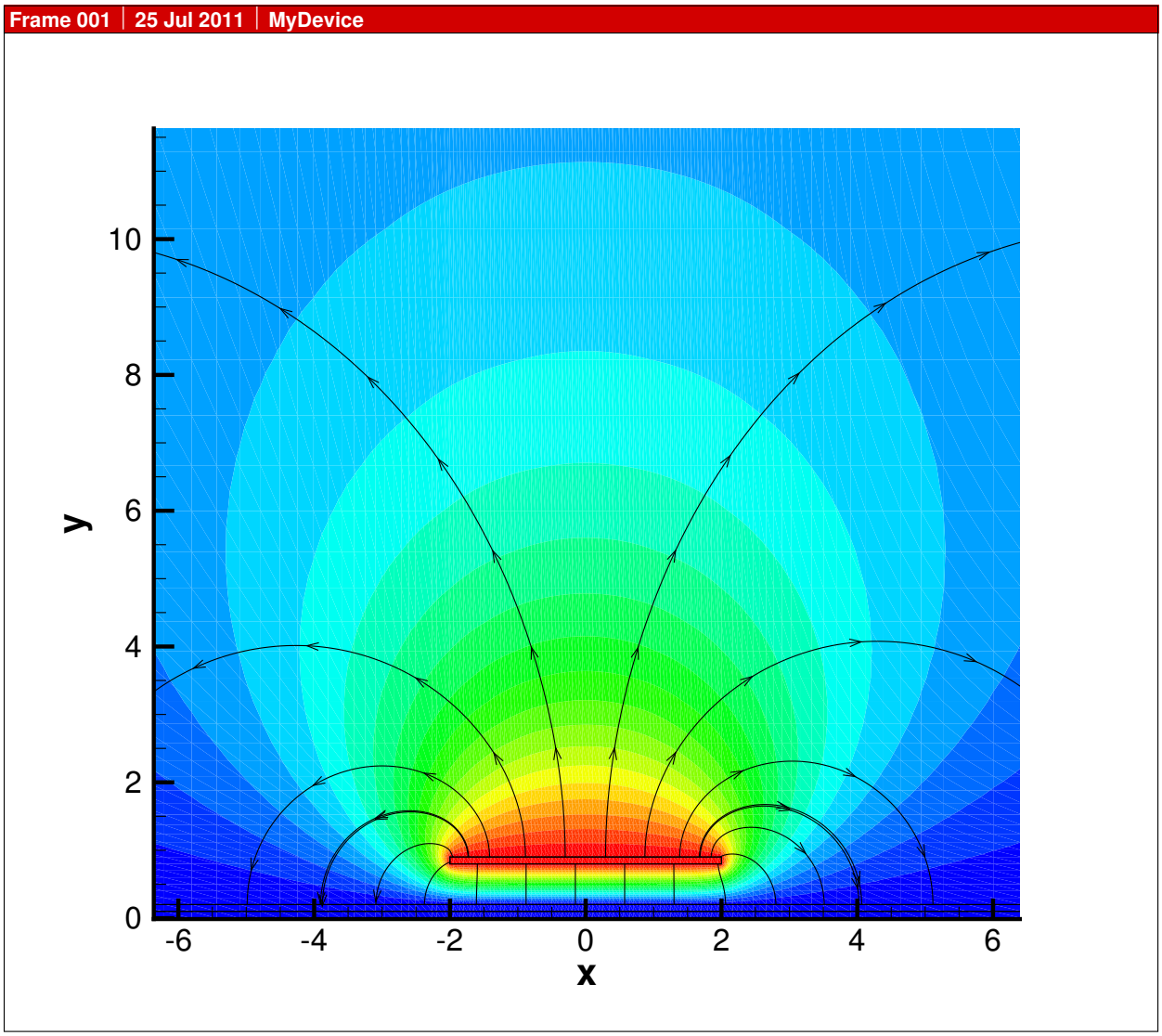

Fig. 16.1: Capacitance simulation result. The coloring is by `Potential`, and the stream traces are for components of `ElectricField`.

Chapter 17

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 [SG69].

17.1 1D diode

17.1.1 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 17.1. The `devsim.python_packages` module is part of the distribution. The example files in the `DEVSIM` distribution set the path properly when loading modules.

Table 17.1: Python package files.

`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

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

```
from devsim import *
from simple_physics import *
```

17.1.2 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)

device="MyDevice"
region="MyRegion"

createMesh(device, region)
```

17.2 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)
```

(continues on next page)

(continued from previous page)

```
####
#### 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")
```

17.2.1 Plotting the result

A plot showing the doping profile and carrier densities are shown in Fig. 17.1. The potential and electric field distribution is shown in Fig. 17.2. The current distributions are shown in Fig. 17.3.

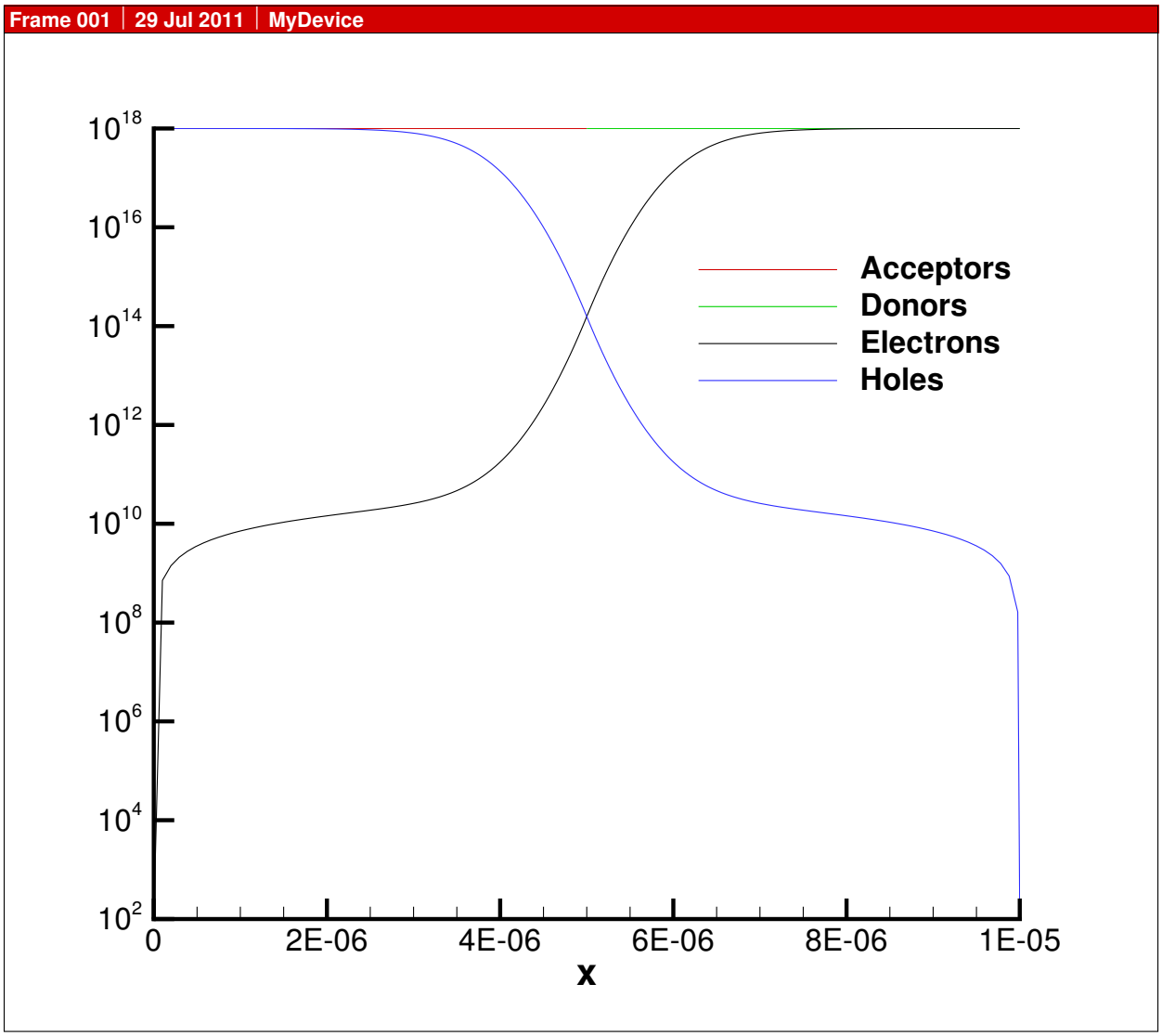

Fig. 17.1: Carrier density versus position in 1D diode.

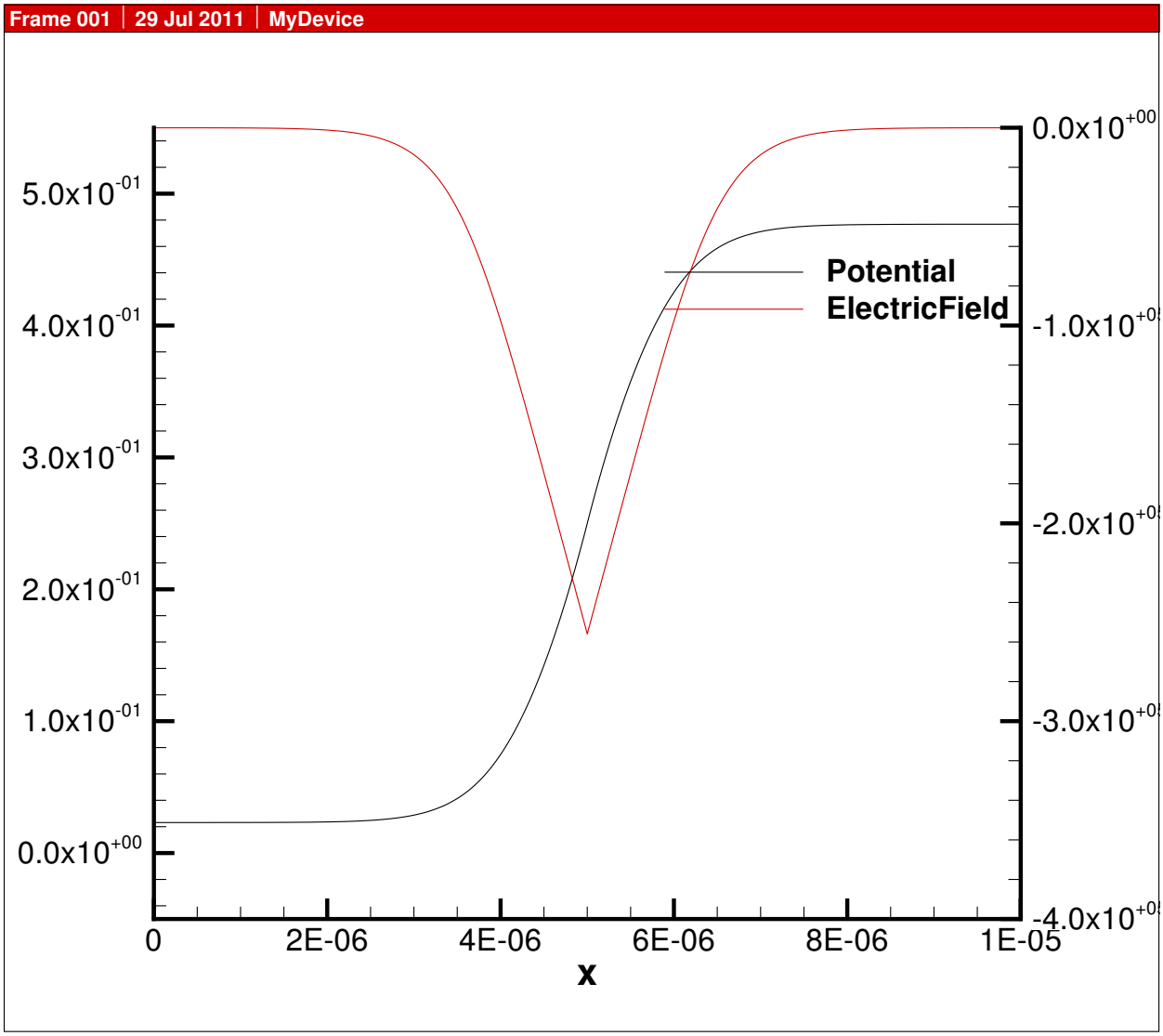

Fig. 17.2: Potential and electric field versus position in 1D diode.

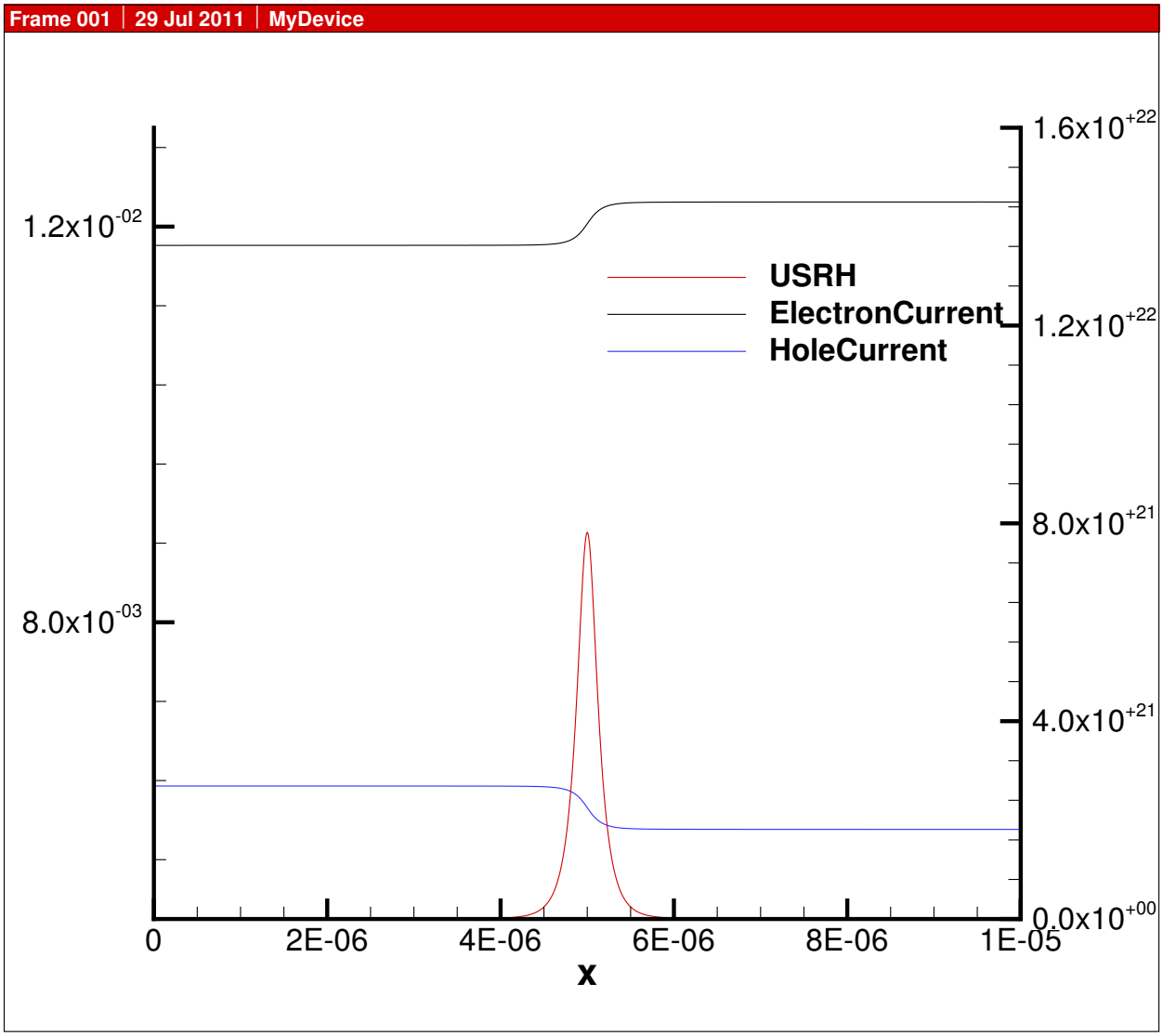

Fig. 17.3: Electron and hole current and recombination.

Bibliography

[pyt] Python Programming Language –- Official Website. http://www.python.org.

[DEG+99] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[GR09] 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.

[MKC02] Richard S. Muller, Theodore I. Kamins, and Mansun Chan. *Device Electronics for Integrated Circuits*. John Wiley & Sons, 3 edition, 2002.

[Ous98] John K. Ousterhout. Scripting: higher level programming for the 21st century. *IEEE Computer*, 31:23–30, 1998.

[SG69] D. L. Scharfetter and H. K. Gummel. Large-signal analysis of a silicon Read diode oscillator. *IEEE Trans. Electron Devices*, ED-16(1):64–77, January 1969.

[ApacheSoftwareFoundation] Apache Software Foundation. Apache License, Version 2.0. URL: http://www.apache.org/licenses/LICENSE-2.0.html.