DEVSIM Manual

Release 2.8.2

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://forum.devsim.org

1.2 Copyright

1.3 Citing this work

Please check `CITATION.md` in the distribution for suggestions on how to cite this work.

1.4 Contributing

Contributions to this project are welcome in the form of bug reporting, documentation, modeling, and feature implementation. Please see the `CONTRIBUTING.md` file in the source code distribution.

1.5 Documentation license

1.6 Documentation source

The documentation source code is available at https://github.com/devsim/devsim_documentation. Suggestions and contributions are welcome.

1.7 Software license

DEVSIM is available from https://devsim.org. The source code is available under the terms of the Apache License Version 2.0 [10]. Examples are released under the same license. Please see the NOTICE and LICENSE files in the distribution for more information.

1.8 Third party licenses

Please see the NOTICE file in the distribution, as well as *Third party libraries* (page 64).

1.9 Disclaimer

DEVSIM LLC MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

1.10 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 *Supported platforms* (page 7). The following sections list bug fixes and enhancements over time. Contact information is listed in *Contact* (page 1). A file named CHANGES.md is now distributed with DEVSIM, which can contain additional details concerning a new release.

2.2 Version 2.8.2

2.2.1 Documentation refactor

The release notes section has been shortened to the most recent releases. Important information from the release notes was placed in the appropriate sections of the manual. The manual has also been reorganized. The pdf formatting has been improved to reducte the number of empty pages.

For older release notes, please refer to the Version 2.8.1 manual located at https://doi.org/10.5281/zenodo.12211919. The latest version is available from https://doi.org/10.5281/zenodo.4583208.

2.3 Version 2.8.1

2.3.1 Help files

Updated instructions. Added additional documentation files.

Table 2.1: Added documentation files

File	Purpose
BUILD.md	Building from source
CODE_OF_CONDUCT.md	Code of conduct
TEST.md	Testing instructions

2.3.2 Database command removal

The material database has been removed.

- `devsim.create_db`
- `devsim.open_db`
- `devsim.close_db`
- `devsim.save_db`
- `devsim.add_db_entry`
- `devsim.get_db_entry`

This feature was only being used in the `bioapp1` examples, and those tests have been updated. This also removes the binary dependence on SQLite.

2.4 Version 2.8.0

2.4.1 `Python` scripts

Based on a contribution by [@simbilod](https://github.com/simbilod), all of the `Python` scripts have been reformatted. The build system was also updated to enforce `Python` script modifications are properly formatted when submitted to the project.

2.4.2 Data output

Reduction in data file sizes

Based on a contribution by [@simbilod](https://github.com/simbilod) *devsim.write_devices()* (page 95) now supports reducing the file size of data files by allowing users to specify a callback function to reduce data usage. In this example, only the `NetDoping` field is written to the Tecplot data file.

```
devsim.write_devices(
    file="mesh2d_reduced.tec",
    type="tecplot",
    include_test=lambda x: x in ("NetDoping",),
)
```

FLOOPS data file output

The `floops` option for *devsim.write_devices()* (page 95) has been removed.

2.4.3 Platform support

Windows build issue

During testing, it was found the Visual Studio 2022 builds were failing a test related to threading. This was found to be a problem with version `17.10`, but not version `17.9`. This affects the build automation, but should not affect the binary releases.

Centos 7 end of life

This is the last version to support Centos 7 before its end of life on June 30, 2024. After this date we will be moving to the AlmaLinux 8 based `manylinux_2_28`.

2.5 Previous releases

For older release notes, please refer to the Version 2.8.1 manual located at https://doi.org/10.5281/zenodo.12211919. The latest version is available from https://doi.org/10.5281/zenodo.4583208.

Chapter 3

Getting started

3.1 Introduction

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.

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 11)). 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 50)) 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.

3.2 Getting help

Please see *Contact* (page 1) for project contact information. The most responsive method is to contact the online forum at https://forum.devsim.org. Additional information, with links to documentation is available at https://devsim.org. Additional documentation files released with the simulator are presented at https://pypi.org/project/devsim/.

3.3 Supported platforms

DEVSIM is compiled and tested on the platforms in Table 3.1.

Table 3.1: Current platforms for DEVSIM

Platform	Architecture	OS Version
Microsoft Windows	x64	Microsoft Windows 10
Linux	x86_64, aarch64	Red Hat Enterprise Linux 7 (Centos 7 compatible)
Apple macOS	x86_64, arm64	macOS 12.5 (Monterey)

These are the minimum supported platforms, and also expected to work on newer versions of these operating systems. If you require a version on a different software platform, please contact us *Contact* (page 1).

3.4 Install Python

A Python version of 3.7 or higher is needed to run DEVSIM. This requirement is often met by the default installations of the above systems. In addition, it is possible to download other Python versions online. Popular distributions of Python are listed in Table 3.2.

Table 3.2: Python distributions

Vendor	Path	Website
Anaconda	$CONDA_PREFIX	https://www.anaconda.com
Python.org	$VIRTUAL_ENV	https://python.org

3.4.1 Create virtual environment

Creating a virtual environment is needed so DEVSIM may necessary math libraries, as discussed in *Default math search path* (page 47). The numpy package is also recommended to ensure that needed math libraries are available.

Anaconda

Using the conda package manager in an Anaconda, a virtual environment is created using.

```
conda create -n denv python numpy
conda activate denv
```

where denv is the name of the environment. If you are using a x64 or x86_64 based system, you may install the Intel Math Kernel Library with the Pardiso Solver.

```
conda install mkl
```

Using venv

For other Python distributions, the requisite packages may be installed by using a venv based virtual environment.

```
python3 -mvenv denv
source denv/bin/activate
pip install numpy
```

where denv is the name of directory containing the environment. If you are using a x64 or x86_64 based system, you may install the Intel Math Kernel Library with the Pardiso Solver.

```
pip install mkl
```

3.5 Install DEVSIM

3.5.1 Install

DEVSIM is available from PyPI (https://pypi.org/project/devsim/) using pip. To install this package for your platform:

```
pip install devsim
```

Please see the devsim_data/INSTALL.md file in the distribution for more information. This files may be found in the prefix directory for your chosen environment listed in Table 3.2.

3.5.2 Test

To ensure a proper installation, please type the following at a Python prompt.

```
>>> import devsim
Searching DEVSIM_MATH_LIBS="libopenblas.dylib:liblapack.dylib:libblas.dylib"
Loading "libopenblas.dylib": ALL BLAS/LAPACK LOADED
Skipping liblapack.dylib
Skipping libblas.dylib
loading UMFPACK 5.1 as direct solver
```

Note that there will be an error if no math libraries are available.

```
>>> import devsim
Searching DEVSIM_MATH_LIBS="libopenblas.so:liblapack.so:libblas.so"
Loading "libopenblas.so": MISSING DLL
```

(continues on next page)

(continued from previous page)

```
Loading "liblapack.so": MISSING DLL
Loading "libblas.so": MISSING DLL
Error loading math libraries.  Please install a suitable BLAS/LAPACK library and␣
↪set DEVSIM_MATH_LIBS.  Alternatively, install the Intel MKL.
libblas.so: cannot open shared object file: No such file or directory
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/user/venv/lib/python3.8/site-packages/devsim/__init__.py", line 8,␣
↪in <module>
    from .devsim_py3 import *
RuntimeError: Issues initializing DEVSIM.
```

3.5.3 Running DEVSIM

See *User interface* (page 11) for instructions on how to invoke `DEVSIM`.

3.6 Building from source

Building from source is possible, and is useful when you want to extend the simulator, use compiler optimizations, or port to a new platform. See the `BUILD.md` file in the project files for more information.

3.7 Install external software tools

3.7.1 Meshing

Gmsh

`Gmsh` [2] is available from https://gmsh.info. `DEVSIM` is able to import triangular or tetrahedral meshes from this application. More information is in *Gmsh* (page 39).

Other meshers

It is also possible to import other mesh formats by writing a converter in `Python`, as described in *Custom mesh loading using scripting* (page 40).

3.7.2 Visualization

See *Visualization software* (page 58) for a listing of available meshing tools, which are known to work with DEVSIM.

3.7.3 Math libraries

BLAS and LAPACK

These are the basic linear algebra routines used in DEVSIM and their selection is described in *BLAS/LAPACK library selection* (page 46).

Intel MKL Pardiso

This library may be installed and selected using the instructions in *Available libraries* (page 45).

Chapter 4

User interface

4.1 Starting DEVSIM

Refer to *Supported platforms* (page 7) for instructions on how to install DEVSIM. Once installed, DEVSIM may be invoked using the following command

devsim is loaded by calling

```
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
```

4.2 Directory structure

A DEVSIM directory is created with the following sub directories listed in *Directory structure for DEVSIM* (page 11).

Table 4.1: Directory structure for DEVSIM

devsim_data	contains project documentation files
devsim_data/doc	product documentation
devsim_data/examples	example scripts
devsim_data/testing	additional examples used for testing

This may be found using the virtual environment path specified in Table 3.2.

4.3 Python language

4.3.1 Introduction

Python is the scripting language employed as the text interface to DEVSIM. Documentation and tutorials for the language are available from [1]. A paper discussing the general benefits of using scripting languages may be found in [5].

4.3.2 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 *
```

4.3.3 Unicode support

Internally, DEVSIM uses UTF-8 encoding, and expects model equations and saved mesh files to be written using this encoding. Care should be taken when using non-ASCII characters in names for visualization using the tools in *Visualization and post processing* (page 58), as this character set may not be supported in these third-party tools.

4.4 Error handling

4.4.1 Exceptions

When a syntax error occurs in a Python script an exception may be thrown. If it is uncaught, then DEVSIM will terminate. An exception that is thrown by DEVSIM is of the type devsim.error. It may be caught, and a message may be extracted to determine the issue.

4.4.2 Fatal errors

When DEVSIM enters a state in which it may not recover. The interpreter will throw a devsim.error 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).

4.4.3 Floating point exceptions

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

4.4.4 Solver errors

When using the *devsim.solve()* (page 111), 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.

4.4.5 Example

More helpful exception information returned to Python if the error is considered fatal. This can be used to decide if the simulation can be restarted. Note that if this occurs during a solve, it is necessary for the user to restore the previous circuit and device solutions if a restart is desired. In addition, model evaluation is reset so that no false cyclic dependencies are reported after an error.

In this example code below, the previously DEVSIM FATAL error string will now provide the context that a floating point exception occurred and be handled in Python.

```
try:
    self.solve()
except error as msg:
    m = str(msg)
    if 'Convergence failure' in m:
        self.set_vapp(last_bias)
    elif'floating point exception' in m:
        self.set_vapp(last_bias)
        self.restore_callback(self.is_circuit)
    else:
        raise
```

4.5 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")
```

4.6 Command help

It is now possible to see the full list of DEVSIM commands by typing

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

4.7 Parallelization

4.7.1 Model evaluation

Routines for the evaluating of models are 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.

4.7.2 Long operations

While running long operations, DEVSIM, will yield to the Python to allow it to perform other operations.

4.7.3 External math libraries

Please see the documentation for external solvers, such as BLAS/LAPACK or the Intel MKL Pardiso, on how to control their threading behavior.

4.8 Reset simulator

The *devsim.reset_devsim()* (page 88) command will clear all simulator data, so that a program restart is not necessary.

4.9 Array type input and output

In most circumstances, the software now returns numerical data using the `Python array` class. This is more efficient than using standard lists, as it encapsulates a contiguous block of memory. More information about this class can be found at https://docs.python.org/3/library/array.html. The representation can be easily converted to lists and `numpy` arrays for efficient manipulation.

When accepting user input involving lists of homogenous data, such as *devsim.set_node_values()* (page 109) the user may enter data using either a list, string of bytes, or the `array` class. It may also be used to input `numpy` arrays or any other class with a `tobytes` method.

Chapter 5

Equation and models

5.1 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.

Additional detail concerning the discussion that follows is available in [7, 8].

In *Mesh elements in 2D* (page 17), 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 50). For special situations, custom matrix assembly is also available and is discussed in *Custom matrix assembly* (page 28).

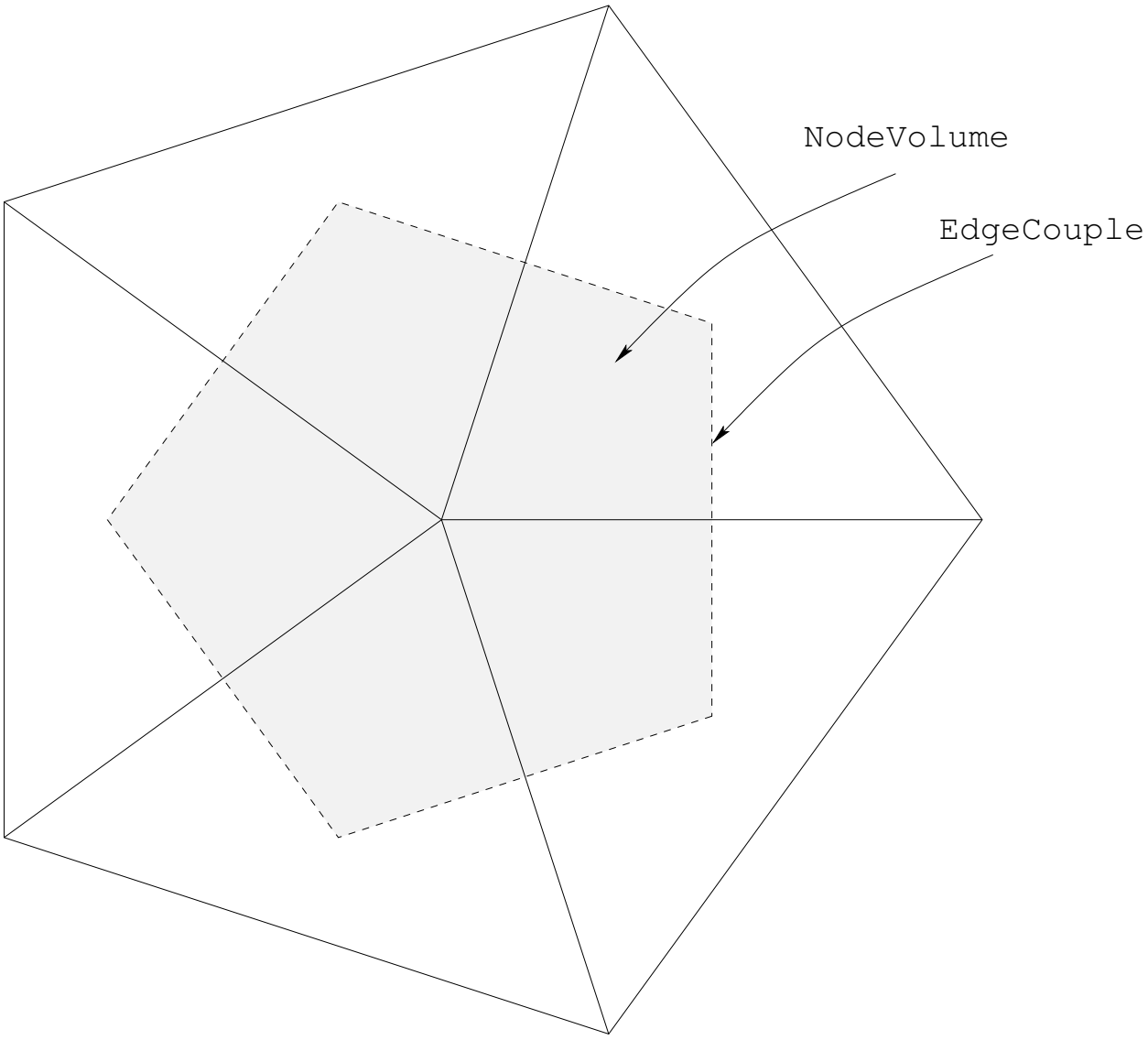

Fig. 5.1: Mesh elements in 2D

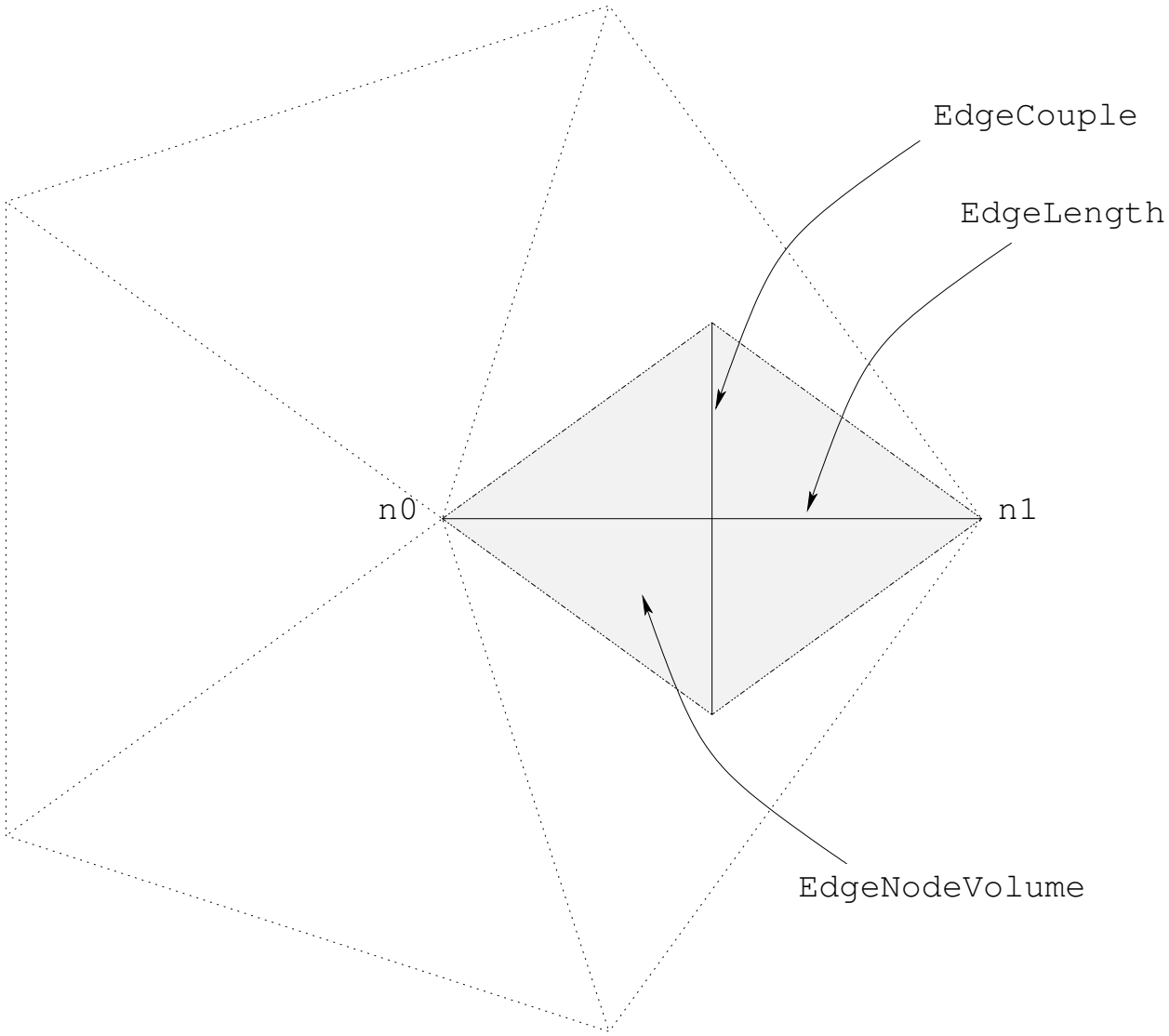

Fig. 5.2: Edge model constructs in 2D

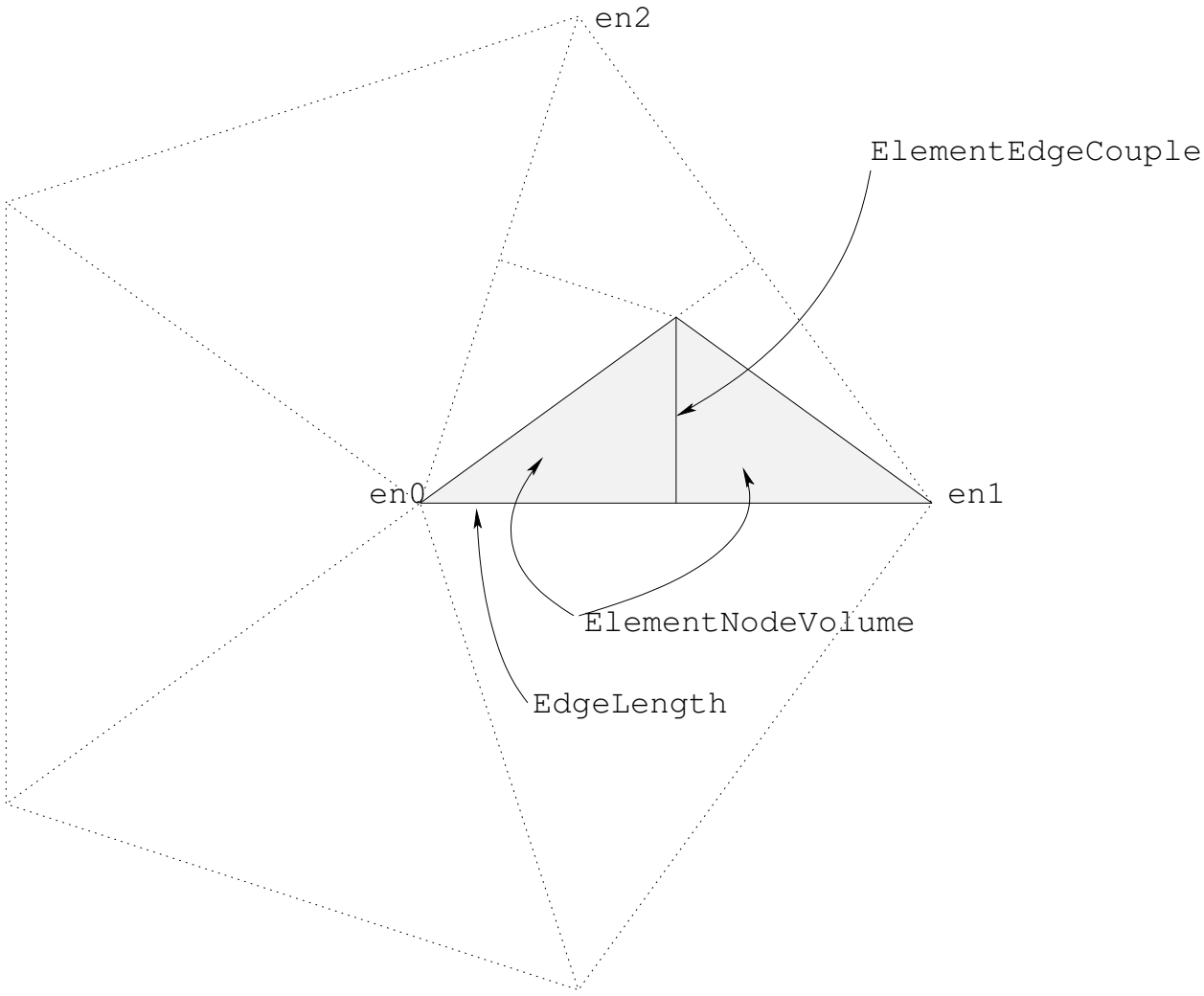

Fig. 5.3: Element edge model constructs in 2D

5.1.1 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.

5.2 Bulk models

5.2.1 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 *Node models defined on each region of a device* (page 21).

In this example, we present an implementation of Shockley Read Hall recombination [4].

```
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 5.1: 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

5.2.2 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 *Edge model constructs in 2D* (page 18), 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()* (page 99). In addition, the *devsim.edge_average_model()* (page 99) can be used to create edge models in terms of node model quantities.

Edge models automatically created for a region are listed in *Edge models defined on each region of a device* (page 22).

Table 5.2: 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.
`EdgeNodeVolume`	The volume for each node on an edge. Used to perform volume 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)

5.2.3 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 *Element edge model constructs in 2D* (page 19), 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 *Element edge models defined on each region of a device* (page 22).

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()* (page 84) for more information.

Table 5.3: 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.

5.2.4 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 *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* (page 23).

Table 5.4: 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)

5.2.5 Conversions between model types

The *devsim.edge_from_node_model()* (page 99) 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()* (page 99) 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()* (page 101) 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()* (page 102) is used to create element edge models referring to each node on the element of the element edge.

5.2.6 Equation assembly

Bulk equations are specified in terms of the node, edge, and element edge models using the *devsim.equation()* (page 84). 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_node0_model` and `volume_node1_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 5.5: 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`

5.3 Interface

5.3.1 Interface models

Interface constructs in 2D. Interface node pairs are located at each \bullet. The SurfaceArea model is used to integrate flux term models. (page 25) 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()* (page 106). 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

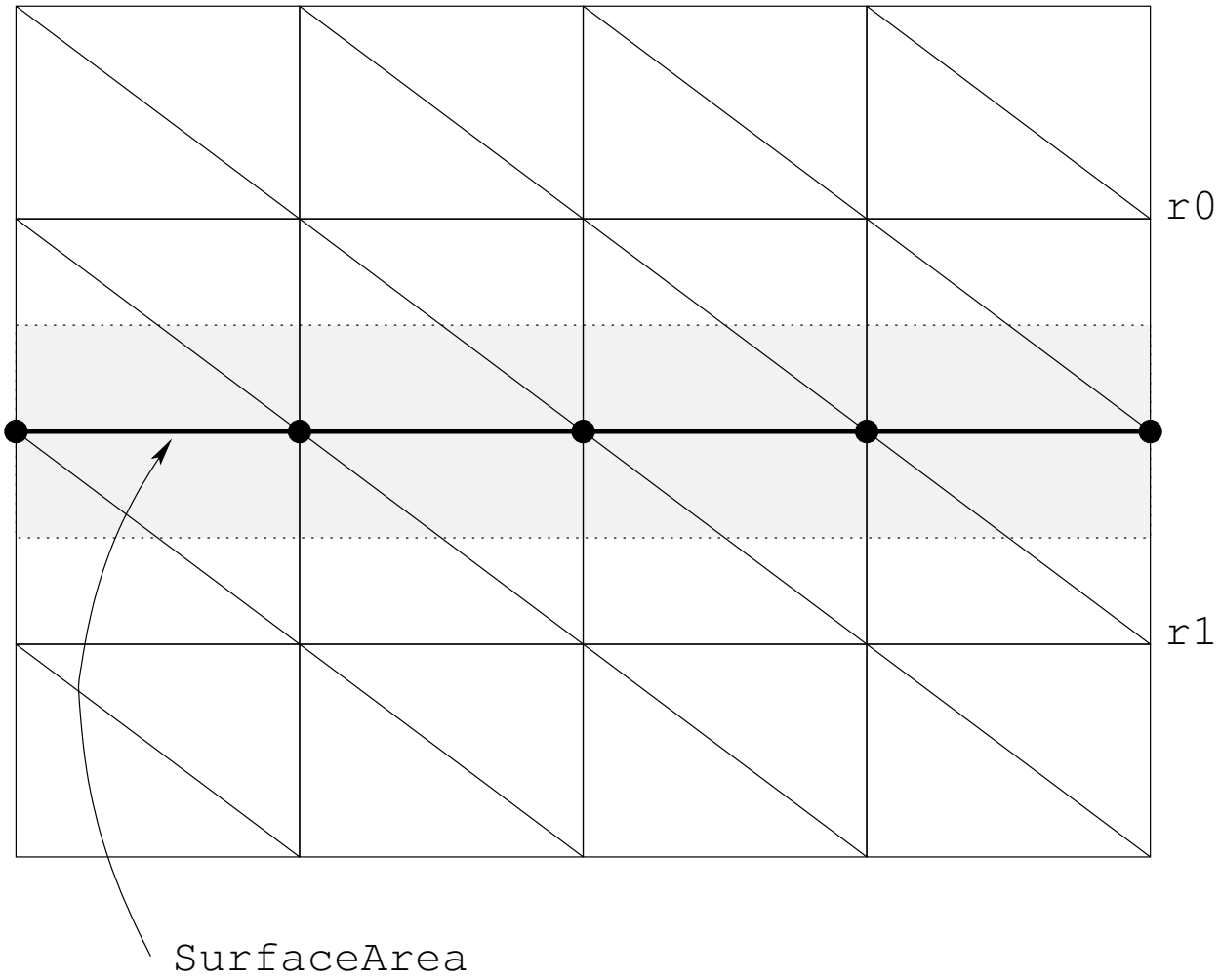

Fig. 5.4: Interface constructs in 2D. Interface node pairs are located at each •. The `SurfaceArea` model is used to integrate flux term models.

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 (page 24).

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

5.3.2 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")
```

5.3.3 Interface equation assembly

There are three types of interface equations considered in `DEVSIM`. They are both activated using the *devsim.interface_equation()* (page 87).

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",
                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.

5.4 Contact

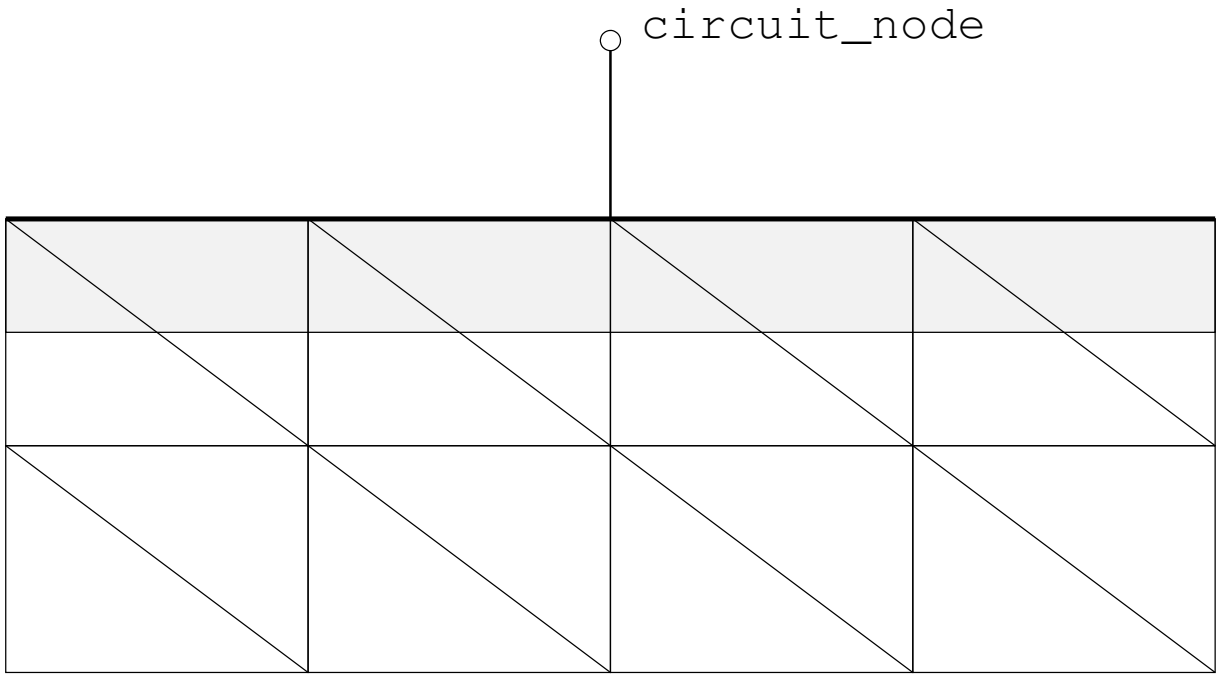

Fig. 5.5: Contact constructs in 2D.

5.4.1 Contact models

Contact constructs in 2D. (page 27) 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()* (page 96) and the *devsim.contact_edge_model()* (page 96), respectively. The key difference is that the models are only evaluated on the contact nodes for the contact specified.

5.4.2 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`.

5.4.3 Contact equation assembly

The *devsim.contact_equation()* (page 83) 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
↪",
  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.

5.5 Custom matrix assembly

The *devsim.custom_equation()* (page 84) command is used to register callbacks to be called during matrix and right hand side assembly. The `Python` procedure should expect to receive two arguments and return two lists and a boolean value. For example a procedure named `myassemble` registered with

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

expects two arguments

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

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 and a boolean value 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], True

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 boolean value denotes whether the matrix and right hand side entries should be row permutated. A value of `True` should be used for assembling bulk equations, and a value of `False` should be used for assembling contact and interface boundary conditions.

The *devsim.get_circuit_equation_number()* (page 82) may be used to get the equation numbers corresponding to circuit node names. The *devsim.get_equation_numbers()* (page 86) 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 as volume integration.

5.6 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()* (page 96)
- *devsim.cylindrical_node_volume()* (page 97)
- *devsim.cylindrical_surface_area()* (page 97)

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",
```

(continues on next page)

(continued from previous page)

```
  value="ElementCylindricalNodeVolume@en0")
set_parameter(name="element_node1_volume_model",
  value="ElementCylindricalNodeVolume@en1")
```

5.7 Notes

5.7.1 Interface

Interace equation coupling

The `name0`, and `name1` options are now available for the *devsim.interface_equation()* (page 87) command. They make it possible to couple dissimilar equation names across regions.

Interface and contact surface area

Interface surface area is stored in the `SurfaceArea` node model. Contact surface area is stored in the `ContactSurfaceArea` node model. These are listed in Table 5.1.

Skip nodes shared with contact

Interface equation assembly skips nodes when an interface node is shared with a contact. Best to be avoided. Share helper script.

5.7.2 Element assembly

The `EdgeNodeVolume` model is now available for the volume contained by an edge and is referenced in *Edge models* (page 21).

The *devsim.equation()* (page 84) supports these options:

- `volume_node0_model`
- `volume_node1_model`

This makes it possible to better integrate nodal quantities on the volumes of element edges. For example, a field dependent generation-recombination rate can be volume integrated separately for each node of an element edge.

The *devsim.contact_equation()* (page 83) supports the following options:

- `edge_volume_model`
- `volume_node0_model`
- `volume_node1_model`

This makes it possible to integrate edge and element edge quantities with respect to the volume on nodes of the edge at the contact. This is similar to *devsim.equation()* (page 84).

The integration parameters for `edge_volume_model` are set with

- `edge_node0_volume_model` (default `EdgeNodeVolume` *Edge models* (page 21))
- `edge_node1_volume_model` (default `EdgeNodeVolume`)

and for `volume_model` with:

- `element_node0_volume_model` (default `ElementNodeVolume` *Element edge models defined on each region of a device* (page 22))
- `element_node1_volume_model` (default `ElementNodeVolume`)

These parameters are applicable to both *devsim.equation()* (page 84) *devsim.contact_equation()* (page 83).

5.7.3 Edge volume model

The *devsim.equation()* (page 84) 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).

5.7.4 Element pair from edge model

The *devsim.element_pair_from_edge_model()* (page 103) command is available to calculate element edge components averaged onto each node of the element edge. This makes it possible to create an edge weighting scheme different from those used in *devsim.element_from_edge_model()* (page 101). The examples `examples/diode/laux2d.py` (2D) and `examples/diode/laux3d.py` (3D) compare the built-in implementations of these commands with equivalent implementations written in `Python`

Chapter 6

Parameters

6.1 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 *Parameters controlling program behavior* (page 33):

Table 6.1: Parameters controlling program behavior

Parameter	Description
debug_level	info, verbose *Example* (page 13)
edge_couple_model	*Cylindrical coordinate systems* (page 29)
edge_node0_volume_model	*Cylindrical coordinate systems* (page 29)
edge_node1_volume_model	*Cylindrical coordinate systems* (page 29)
element_edge_couple_model	*Cylindrical coordinate systems* (page 29)
element_node0_volume_model	*Cylindrical coordinate systems* (page 29)
element_node1_volume_model	*Cylindrical coordinate systems* (page 29)
extended_equation	value=False Extended precision equation evaluation
extended_model	value=False Extended precision model evaluation
extended_solver	value=False Extended precision matrix and RHS assembly and error evaluations. Linear solver and circuit assembly is still double precision``
info	*Determine loaded math libraries* (page 47)
node_volume_model	*Cylindrical coordinate systems* (page 29)
solver_callback	*Custom direct solver* (page 47)
surface_area_model	Model for integration of flux and hybrid interfaces.
threads_available	value=1, *Command help* (page 14)
threads_task_size	value=?, *Command help* (page 14)

6.2 Environment variables

Environment variables to control program behavior are listed in *Environment controlling program behavior* (page 33):. Please consult *Release notes* (page 3): for the most up to information concerning their usage.

Table 6.2: Environment controlling program behavior

Environment Variable	Description
DEVSIM_MATH_LIBS	List of BLAS/LAPACK libraries to load instead of system defaults
DEVSIM_NEW_SYMBOLIC	When set, do a new symbolic matrix factorization during direct solve iterations, *Symbolic factorization reuse* (page 48)
CONDA_PREFIX	For Anaconda Python installations, this is the location of the devsim_data directory.
VIRTUAL_ENV	For pip-based Python installations, this is the location of the devsim_data directory.

6.3 Notes

Parameters may be used in model expressions. If a parameter is not found, then `DEVSIM` will also look for a circuit node by the name used in the model expression.

Chapter 7

Circuits

7.1 Overview

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.

7.2 Circuit elements

Circuit elements are manipulated using the commands in *Circuit commands* (page 81). Using the *devsim.circuit_element()* (page 81) 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.

7.3 Connecting devices

For devices to contribute current to an external circuit, the *devsim.contact_equation()* (page 83) 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()* (page 82) 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.

7.4 Clearing circuit

The *devsim.delete_circuit()* (page 82) command may be used to remove the circuit completely.

Chapter 8

Meshing

8.1 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()* (page 93) 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()* (page 90) 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()* (page 90), *devsim.add_1d_interface()* (page 90) and *devsim.add_1d_region()* (page 90) are used to specify the contacts, interfaces and regions for the device.

Once the meshing commands have been completed, the *devsim.finalize_mesh()* (page 95) is called to create a mesh structure and then *devsim.create_device()* (page 93) is used to create a device using the mesh.

8.2 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()* (page 92) 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()* (page 91) or an interface using *devsim.add_2d_interface()* (page 91).

Once the meshing commands have been completed, the *devsim.finalize_mesh()* (page 95) is called to create a mesh structure and then *devsim.create_device()* (page 93) is used to create a device using the mesh.

8.3 Using an external mesher

8.3.1 Gmsh

The Gmsh meshing software (see *Gmsh* (page 9)) can be used to create a 1D, 2D, or 3D mesh suitable for use in DEVSIM. DEVSIM supports reading version 2.2 meshes from Gmsh. In order to write this format, it is necessary to specify the mesh format when writing out a mesh file. From the *gmsh* command line, use the `-format msh2` option.

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",
  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()* (page 95) is called to create a mesh structure and then *devsim.create_device()* (page 93) is used to create a device using the mesh.

8.3.2 Custom mesh loading using scripting

It is also possible to arbitrarily load a mesh from a `Python` using the *devsim.create_gmsh_mesh()* (page 94). This is explained in the `Notes` section of the command. In addition, please see the `testing/pythonmesh1d` script for a simple demonstration script. These meshes may only contain points, lines, triangles, and tetrahedra. Hybrid meshes or uniform meshes containing other elements are not supported at this time.

8.4 Loading and saving results

The *devsim.write_devices()* (page 95) 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()* (page 95) is then used to reload the device data for restarting the simulation.

8.5 Mesh processing

See *Examples* (page 60) for examples involving mesh processing.

8.6 Notes

8.6.1 Contacts

Contact material

Contacts requires 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.

Create contacts from interface

The *devsim.create_contact_from_interface()* (page 93) 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.

8.6.2 Device and mesh deletion commands

The *devsim.delete_device()* (page 95) command makes it possible to delete devices so they will no longer be solved in the simulation. Any parameters set on the device are also removed from the system.

The *devsim.delete_mesh()* (page 95) command makes it possible to delete meshes. Once a mesh has been deleted, it is no longer possible to create devices from it using the *devsim.create_device()* (page 93) command.

8.6.3 Periodic boundary conditions

The *devsim.create_interface_from_nodes()* (page 94) command makes it possible to create an interface with non coincident nodes. This enables the use of periodic boundary conditions.

Chapter 9

Solver and numerics

9.1 Overview

DEVSIM offers a range of simulation algorithms.

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.

9.2 Solution methods

DEVSIM uses Newton methods to solve the system of PDE's. All of the analyses are performed using the *devsim.solve()* (page 111).

9.2.1 DC analysis

A DC analysis is performed using the *devsim.solve()* (page 111).

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

9.2.2 AC analysis

An AC analysis is performed using the *devsim.solve()* (page 111). A circuit voltage source is required to set the AC source.

9.2.3 Noise and sensitivity analysis

An noise analysis is performed using the *devsim.solve()* (page 111) 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()* (page 111). 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)

9.2.4 Transient analysis

Transient analysis is performed using the *devsim.solve()* (page 111). `DEVSIM` supports time-integration of the device PDE's. The three methods are supported are:

- BDF1
- TRBDF
- BDF2

9.3 Extended precision

9.3.1 Platform dependence

Extended precision is available on all binaries. For `Linux x86_64`, this uses the 128-bit precision available with the `GCC` compilers. On other platforms, `x64`, `arm64`, `aarch64`, the `cpp_bin_float_quad` type is used from the `boost` libraries, and is similar to 128-bit precision.

9.3.2 How to control

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

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)
```

9.3.3 Kahan summation in extended precision mode

The `kahan3` and `kahan4` functions use the Kahan summation algorithm for extended precision model evaluation. With this change, better than 128-bit floating precision is available when extended precision is enabled.

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

The `testing/kahan_float128.py` script demonstrates this.

9.4 Floating point exceptions

9.4.1 FPE checking during external solve

On `arm64` and `aarch64` platforms, the software does not check for floating point exceptions (FPEs) during usage of the direct solver. During testing, it was discovered that FPEs were occuring during factorization for both the `SuperLU` and the `UMFPACK 5.1`. Removing this check allows more of the tests to run through to completion.

9.4.2 Additional Information

Please see *Floating point exceptions* (page 13).

9.5 Solver and math library selection

9.5.1 Available libraries

Intel Math Kernel Library

A specific version is not required when loading the `Intel Math Kernel Library`. This method is the default for `x64` and `x86_64` systems. Instructions for installing in a `Python` virtual environment are given in *Create virtual environment* (page 7).

`UMFPACK 5.1` solver

The `UMFPACK 5.1` solver is now available as a shared library distributed with the software. It is licensed under the terms of the LGPL 2.1 and our version is hosted here:

https://github.com/devsim/umfpack_lgpl

Please note that this version uses a scheme to detect the BLAS/LAPACK libraries being used by `DEVSIM`, as described in *BLAS/LAPACK library selection* (page 46).

In order to use this library, a shim script is provided to load `UMFPACK 5.1` and set it as the solver. Please see this example:

```
python -mdevsim.umfpack.umfshim ssac_cap.py
```

Custom solver

Please see *Custom direct solver* (page 47) for more information.

`SuperLU`

`SuperLU` is no longer available as a solver in the binary distributions of `DEVSIM`. It is available for custom applications, which would require a custom build of the software.

9.5.2 Automatic direct solver selection

`UMFPACK 5.1` is the default when the `Intel Math Kernel Library` is not available, making this the default for the macOS arm64 platform.

The direct solver may be selected by using the `direct_solver` parameter.

```
devsim.set_parameter(name='direct_solver', value='mkl_pardiso')
```

All compatible platforms will search for the `Intel MKL Pardiso`. The default is `mkl_pardiso` when the `Intel Math Kernel Library` is loaded. The `umfshim` described in *UMFPACK 5.1 solver* (page 45) uses the `custom` option for this parameter to implement calls using the custom solver callback system described in *Custom direct solver* (page 47).

9.5.3 BLAS/LAPACK library selection

Reference versions are available from https://netlib.org. There are optimized versions available from other vendors. It is possible to load alternative implementations of the BLAS/LAPACK used by the software. The `DEVSIM_MATH_LIBS` environment variable may be used to set a `:` separated list of libraries. These names may be based on relative or absolute paths. The program will load the libraries in order, and stop when all of the necessary math symbols are supplied. If symbols for the `Intel Math Kernel Library` are detected, then the Pardiso direct solver will be enabled.

`Linux` example:

```
export DEVSIM_MATH_LIBS=libblas.so:liblapack.so
```

`Apple macOS` example:

```
export DEVSIM_MATH_LIBS=libblas.dylib:liblapack.dylib
```

On Windows the `DEVSIM_MATH_LIBS` uses the `;` as the path separator, while macOS and Linux still use `:`.

The math library search order is then:

- The math libraries listed in the `DEVSIM_MATH_LIBS` environment variable, with the appropriate separator.
- The Intel Math Kernel Library
- These dynamic libraries * OpenBLAS (e.g. libopenblas.so) * LAPACK (e.g. liblapack.so) * BLAS (e.g. libblas.so)

9.5.4 Default math search path

It is recommended to use the full path for all of the math solver libraries. On macOS and Linux, the RPATH has been modified to look in places relative to the *devsim* module, instead of using `CONDA_PREFIX` or `VIRTUAL_ENV`.

- `macOS : @loader_path;@loader_path/../lib;@loader_path/../../../../lib; @executable_path/../lib`
- `Linux : $ORIGIN:$ORIGIN/../lib:$ORIGIN/../../../../lib`

9.5.5 Determine loaded math libraries

To determine the loaded math libraries, use

```
devsim.get_parameter(name='info')['math_libraries']
```

9.6 Custom direct solver

It is call a custom direct solver. The direct solver is called from `Python` and the callback is implemented by setting these parameters:

parameter table

```
devsim.set_parameter(name="direct_solver", value="custom")
devsim.set_parameter(name="solver_callback", value=local_solver_callback)
```

Where the first parameter enables the use of the second parameter to set a callback function. Please see the `testing/umfpack_shim.py` for a sample implementation using `UMFPACK 5.1`.

9.7 Diagnostics

9.7.1 Problem node identification

The node indexes with the maximum error for each equation will be printed when `debug_level` is `verbose`.

```
devsim.set_parameter(name="debug_level", value="verbose")
```

These are printed as `RelErrorNode` and `AbsErrorNode`:

```
Region: "gate"      RelError: 5.21531e-14   AbsError: 4.91520e+04
  Equation: "ElectronContinuityEquation"    RelError: 4.91520e-16   AbsError: 4.
↪91520e+04
    RelErrorNode: 129       AbsErrorNode: 129
```

This information is also returned when using the `info=True` option on the *devsim.solve()* (page 111) command for each equation on each region of a device.

If the `info` flag is set to `True` on the `solve` command, the iteration information will be returned, and an exception for convergence will no longer be thrown. It is the responsibility of the caller to test the result of the `solve` command to see if the simulation converged. Other types of exceptions, such as floating point errors, will still result in a `Python` exception that needs to be caught.

9.7.2 Convergence information

The *devsim.solve()* (page 111) now supports the `info` option. The solve command will then return convergence information.

9.8 Symbolic factorization reuse

The `Intel Math Kernel Library` solver will now use reuse the symbolic factorization, if the simulation matrix sparse matrix pattern has not changed after the second nonlinear solver iteration. This reduces simulation time, but can result in numerical differences in the simulation result. Setting the environment variable, `DEVSIM_NEW_SYMBOLIC`, will do a new symbolic factorization for each iteration.

This behavior may be controlled by using this option in the *devsim.solve()* (page 111) command

```
solve(symbolic_iteration_limit = -1)
```

where setting the value to `-1` will create a new symbolic factorization for all nonlinear iterations. Setting the value to a number greater than `0` will mark all iterations afterwards for reusing the previous symbolic factorization.

9.9 Notes

9.9.1 Convergence tests

The `maximum_error` and `maximum_divergence` options were added to the *devsim.solve()* (page 111) command. If the absolute error of any iteration goes above `maximum_error`, the simulation stops with a convergence failure. The `maximum_divergence` is the maximum number of iterations that the simulator error may increase before stopping.

9.9.2 Simulation matrix

The *devsim.get_matrix_and_rhs()* (page 111) command was not properly accepting the `format` parameter, and was always returning the same type.

9.9.3 Get matrix and rhs for external use

The *devsim.get_matrix_and_rhs()* (page 111) command has been added to assemble the static and dynamic matrices, as well as their right hand sides, based on the current state of the device being simulated. The `format` option is used to specify the sparse matrix format, which may be either in the compressed column or compressed row formats, `csc` or `csr`.

9.9.4 Transient analysis

devsim.set_initial_condition() (page 111) to set initial transient condition as alternative to using the `transient_dc` option to the *devsim.solve()* (page 111) command. Suitable options for this command may be provided from the `get_matrix_and_rhs()` command.

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 *Basic expressions involving unary, binary, and logical operators* (page 51), 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 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
`cosh(exp1)`	Hyperbolic Cosine
`sinh(exp1)`	Hyperbolic Sine
`tanh(exp1)`	Hyperbolic Tangent
`B(exp1)`	Bernoulli Function
`dBdx(exp1)`	derivative of Bernoulli function
`dot2d(exp1x, exp1y, exp2x, exp2y)`	`exp1x*exp2x+exp1y*exp2y`
`exp(exp1)`	exponent
`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
`log(exp1)`	natural log
`max(exp1, exp2)`	maximum of the two arguments
`min(exp1, exp2)`	minimum of the two arguments
`pow(exp1, exp2)`	take exp1 to the power of exp2
`sgn(exp1)`	sign function
`step(exp1)`	unit step function
`kahan3(exp1, exp2, exp3)`	Extended precision addition of arguments
`kahan4(exp1, exp2, exp3, exp4)`	Extended precision addition of arguments
`vec_max`	maximum of all the values over the entire region or interface
`vec_min`	minimum of all the values over the entire region or interface
`vec_sum`	sum of all the values over the entire region or interface

Table 10.3: Error functions

Function	Description
`erfc(exp1)`	complementary error function
`derfcdx(exp1)`	derivative of complementary error function
`erfc_inv(exp1)`	inverse complementary error function
`derfc_invdx(exp1)`	derivative of inverse complementary error function
`erf(exp1)`	error function
`derfdx(exp1)`	derivative error function
`erf_inv(exp1)`	inverse error function
`derf_invdx(exp1)`	derivative of inverse error function

Table 10.4: Fermi Integral functions

Function	Description
`Fermi(exp1)`	Fermi Integral
`dFermidx(exp1)`	derivative of Fermi Integral
`InvFermi(exp1)`	inverse of the Fermi Integral
`dInvFermidx(exp1)`	derivative of InvFermi Integral

Table 10.5: Gauss-Fermi Integral functions

`gfi(exp1, exp2)`	Gauss-Fermi Integral
`dgfidx(exp1, exp2)`	Derivative of Gauss-Fermi Integral with respect to first argument
`igfi(exp1, exp2)`	Inverse Gauss-Fermi Integral
`digfidx(exp1, exp2)`	Derivative of Inverse Gauss-Fermi Integral with respect to first argument

In *Predefined functions* (page 52) 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. *Error functions* (page 53) lists the error functions, derivatives, and inverses. *Fermi Integral functions* (page 53) lists the Fermi functions, and are based on the Joyce-Dixon Approximation [3]. The Gauss-Fermi functions are listed in *Gauss-Fermi Integral functions* (page 53), based on [6].

10.2.4 Commands

Table 10.6: 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 *Commands* (page 54). 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 10.7: 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 *Commands for user functions* (page 54). 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()* (page 109) 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 96) 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()* (page 108) 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 96), `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 and post processing

11.1 Introduction

DEVSIM is able to create files for visualization tools. Information about acquiring these tools are presented in *Install external software tools* (page 9).

11.2 Visualization software

11.2.1 Overview

The tools in Table 11.1 can read the file Tecplot and VTK Formats.

Table 11.1: Open source visualization tools

ParaView	visualization tool available at https://paraview.org.
VisIt	visualization tool available from https://visit-dav.github.io/visit-website/

11.2.2 Using ParaView

The *devsim.write_devices()* (page 95) 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.

11.2.3 Using VisIt

VisIt supports reading the Tecplot and ParaView formats. When using the vtk option on the *devsim.write_devices()* (page 95), a file with a visit filename extension is created to load the files created for ParaView.

11.3 Reducing file sizes

Based on a contribution by [@simbilod](https://github.com/simbilod) *devsim.write_devices()* (page 95) now supports reducing the file size of data files by allowing users to specify a callback function to reduce data usage. In this example, only the NetDoping field is written to the Tecplot data file.

```
devsim.write_devices(
    file="mesh2d_reduced.tec",
    type="tecplot",
    include_test=lambda x: x in ("NetDoping",),
)
```

11.4 Post processing

DEVSIM has several commands for getting information on the mesh. Those related to post processing are described in *Model commands* (page 96) and *Geometry commands* (page 87).

See *Loading and saving results* (page 40) for information about loading and saving mesh information to a file.

11.4.1 Index information

The coordinate_index and node_index are default node models created on a region (Table 5.1}). The edge_index is a default edge models created on a region Table 5.2.

11.4.2 Element node list

The *devsim.get_element_node_list()* (page 87) retrieves a list of nodes for every element on a region, contact, or interface.

Chapter 12

Examples

12.1 Included examples

The following example directories are contained in the distribution. Some of them are described in *Simple Examples* (page 65).

Table 12.1: Examples Distributed with DEVSIM

Directory	Description
capacitance	These are 1D and 2D capacitor simulations, using the internal mesher. A description of these examples is presented in *Capacitor* (page 65).
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 75).
bioapp1	This is a biosensor application.
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. 12.1.
mobility	This is an advanced example using electric field dependendent mobility models.
plotting	Example using a Python notebook. There is 3D visualization using pyvista.

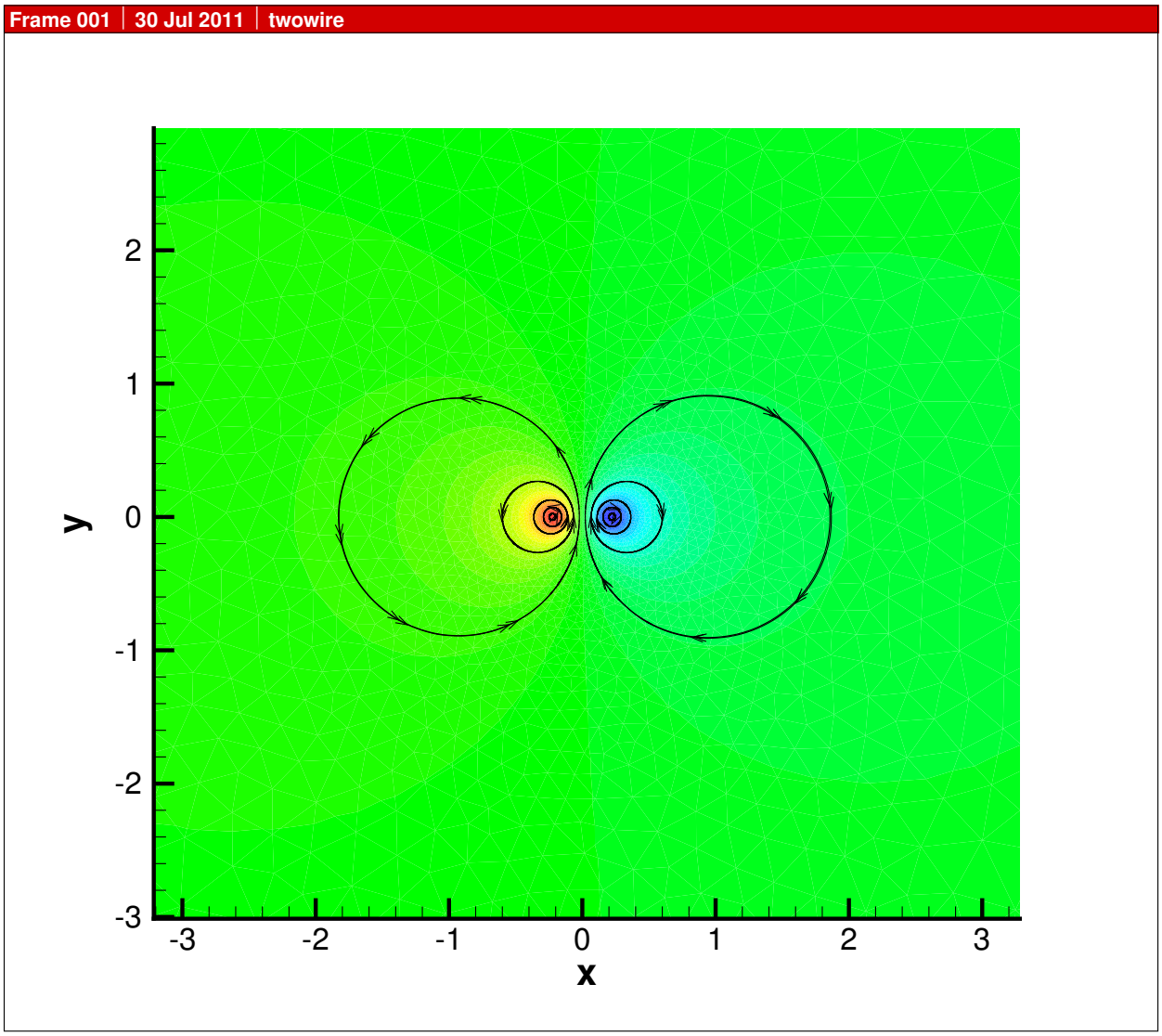

Fig. 12.1: Simulation result for solving for the magnetic potential and field. The coloring is by the Z component of the magnetic potential, and the stream traces are for components of magnetic field.

12.2 Test scripts

The scripts in the `testing` and `examples` directories are used for the regression tests whose platform dependent results are kept in the repositories listed in *Regression results* (page 63).

Some of them demonstrate the use of different features of the simulator.

- Extended precision

  `examples/diode/gmsh_diode3d_float128.py`

- Version and Solver Information

  Parameter `info` can be queried for getting version information. The file `testing/info.py` contains an example.

- Small signal simulation

  `examples/diode/ssac_diode.py`

- Custom Matrix Assembly

  `examples/diode/diode_1d_custom.py` demonstrates custom matrix assembly and can be directly compared to `examples/diode/diode_1d.py`.

- Custom External Meshing

  `testing/pythonmesh1d.py` demonstrates how to create a mesh via script, and how to get mesh information using *devsim.get_element_node_list()* (page 87).

- Transient Simulation
  - `examples/diode/tran_diode.py` demonstrates transient diode simulation.
  - `testing/transient_rc.py` test which compares simulation with analytic result for RC circuit.
- Fermi and Gauss-Fermi Statistics
  - `testing/Fermi1.py` Fermi integral
  - `testing/Fermi1_float128.py` Fermi integral in extended floating point precision
  - `testing/GaussFermi.py` Gauss-Fermi integral
  - `testing/GaussFermi.py` Gauss-Fermi integral with extended floating point precision

12.3 Related projects

12.3.1 Source code

- devsim (https://github.com/devsim/devsim)

  The simulator

- devsim_documentation (https://github.com/devsim/devsim_documentation)

Documentation for the project

- umfpack_lgpl (https://github.com/devsim/umfpack_lgpl)

  Fork of `UMFPACK 5.1`

12.3.2 Examples

- devsim_bjt_example (https://github.com/devsim/devsim_bjt_example)

  BJT Example.

- devsim_3dmos (https://github.com/devsim/devsim_3dmos)

  Simulation files use in publication. Processes mesh generated with `Cubit`.

- devsim_misc (https://github.com/devsim/devsim_misc)

  Miscellaneous Documentation Files. Contains mesh refinement scripts and `Gmsh` processing.

- devsim_density_gradient (https://github.com/devsim/devsim_density_gradient)

  Density Gradient Simulation

12.3.3 Regression results

- devsim_tests_linux_aarch64 (https://github.com/devsim/devsim_tests_linux_aarch64)

  Linux `arm64`

- devsim_tests_linux_x86_64 (https://github.com/devsim/devsim_tests_linux_x86_64)

  Linux `x86_64`

- devsim_tests_macos_arm64 (https://github.com/devsim/devsim_tests_macos_arm64)

  macOS `arm64`

- devsim_tests_win64 (https://github.com/devsim/devsim_tests_win64)

  Windows `x64`

12.4 Mobile app

- https://tcadapp

12.5 Third party libraries

- Eigen (https://gitlab.com/libeigen/eigen.git)
- Boost multiprecision (https://github.com/boostorg/multiprecision)
- Boost math (https://github.com/boostorg/math)
- |superlu| (https://github.com/xiaoyeli/superlu)

Chapter 13

Simple Examples

13.1 Capacitor

13.1.1 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 65) 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 69).

13.1.2 1D capacitor

Equations

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

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

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.

Creating the mesh

The following statements create a one-dimensional mesh which is 1 cm long with a 0.1 cm 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)
```

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)
```

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")
```

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,
             equation="1")

  contact_equation(device=device, contact=c, name="PotentialEquation",
             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.

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
↪")))
```

Running the simulation

We run the simulation and see the results.

```
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.

13.1.3 2D capacitor

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

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
```

(continues on next page)

(continued from previous page)

```
y1  =0.1
y2  =0.2
y3  =0.8
y4  =0.9
ymax=50.0

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

device=device
region="air"

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

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

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

Setting up the models

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

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

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

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

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

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

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

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

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

###
### Create the bulk equation
###
```

(continues on next page)

(continued from previous page)

```
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",
            node_model="%s_bc" % c, edge_charge_model="DField")

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



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


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

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
```

(continues on next page)

(continued from previous page)

```
↪"))

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

Running the simulation

```
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 *Capacitance simulation result. The coloring is by Potential, and the stream traces are for components of ElectricField.* (page 74).

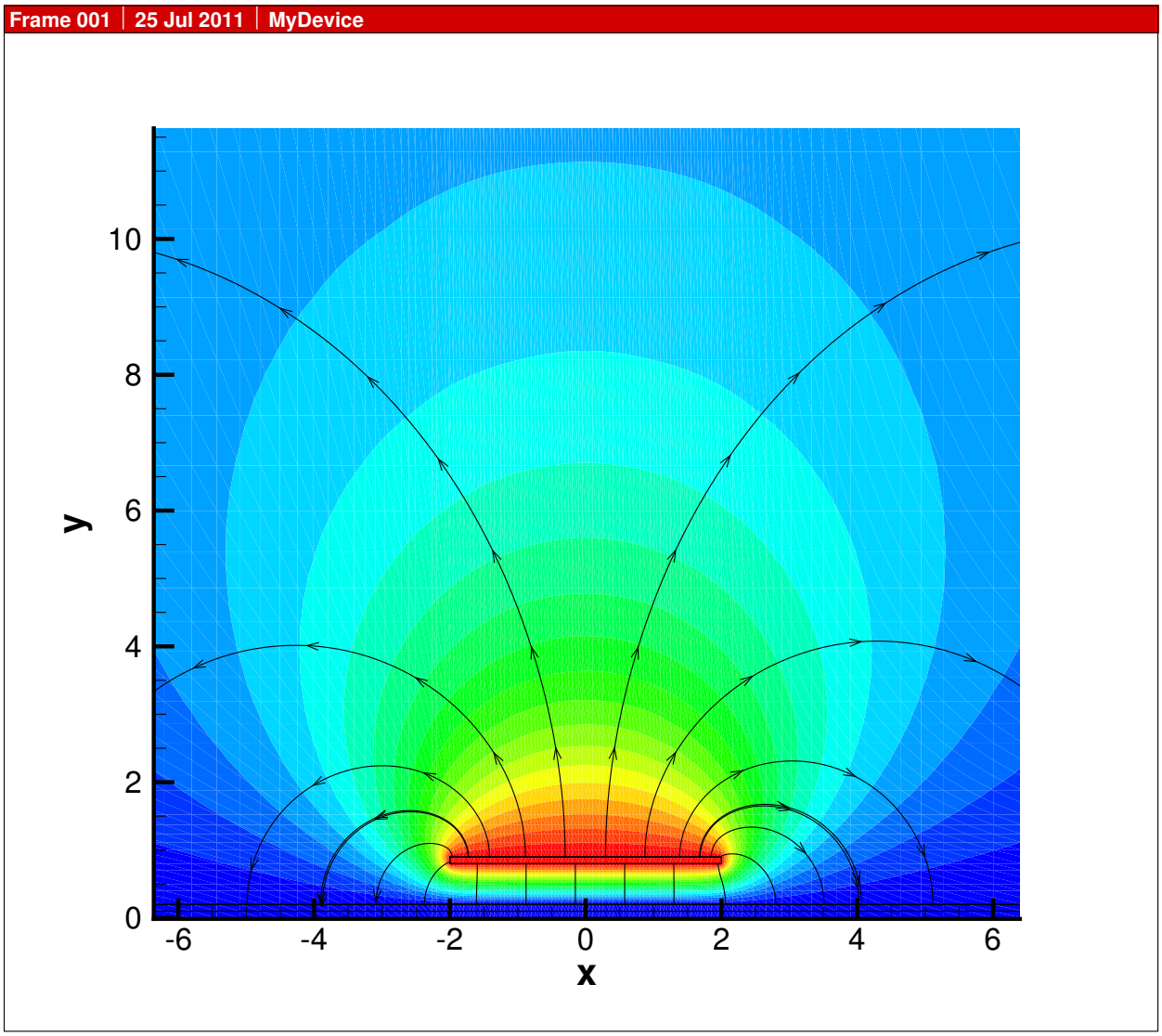

Fig. 13.1: Capacitance simulation result. The coloring is by `Potential`, and the stream traces are for components of `ElectricField`.

13.2 Diode

13.2.1 Overview

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 [9].

13.2.2 1D diode

Using the python packages

For these examples, python modules are provided to supply the appropriate model and parameter settings. A listing is shown in *Python package files* (page 75). 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 13.1: `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 *
```

Creating the mesh

This creates a mesh 10^{-5} 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")
```

(continues on next page)

(continued from previous page)

```
  create_device(mesh="dio", device=device)

device="MyDevice"
region="MyRegion"

createMesh(device, region)
```

Physical models and parameters

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

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

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

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

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


####
#### Initial DC solution
```

(continues on next page)

(continued from previous page)

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

Plotting the result

A plot showing the doping profile and carrier densities are shown in *Carrier density versus position in 1D diode.* (page 78). The potential and electric field distribution is shown in *Potential and electric field versus position in 1D diode.* (page 79). The current distributions are shown in *Electron and hole current and recombination.* (page 80).

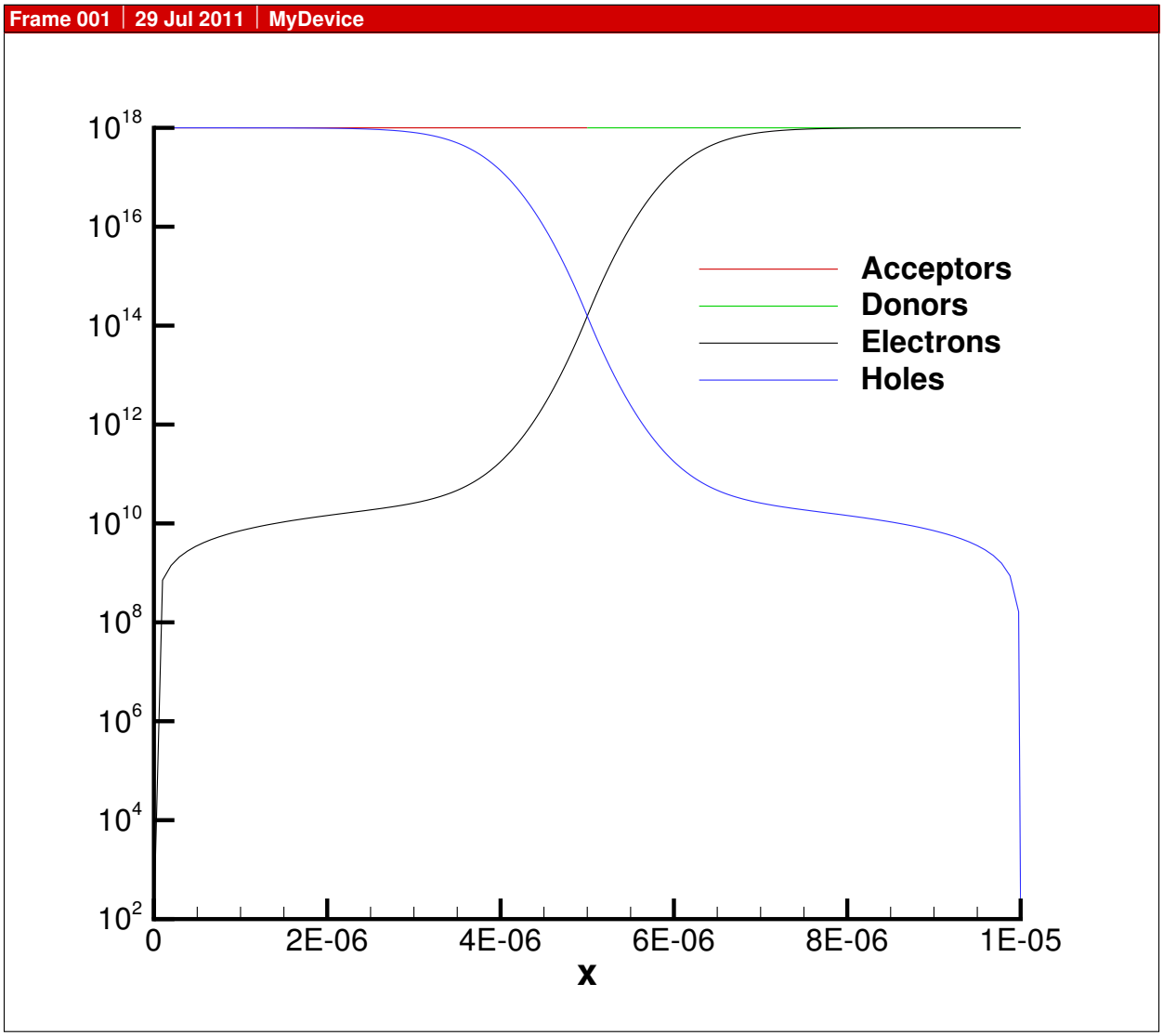

Fig. 13.2: Carrier density versus position in 1D diode.

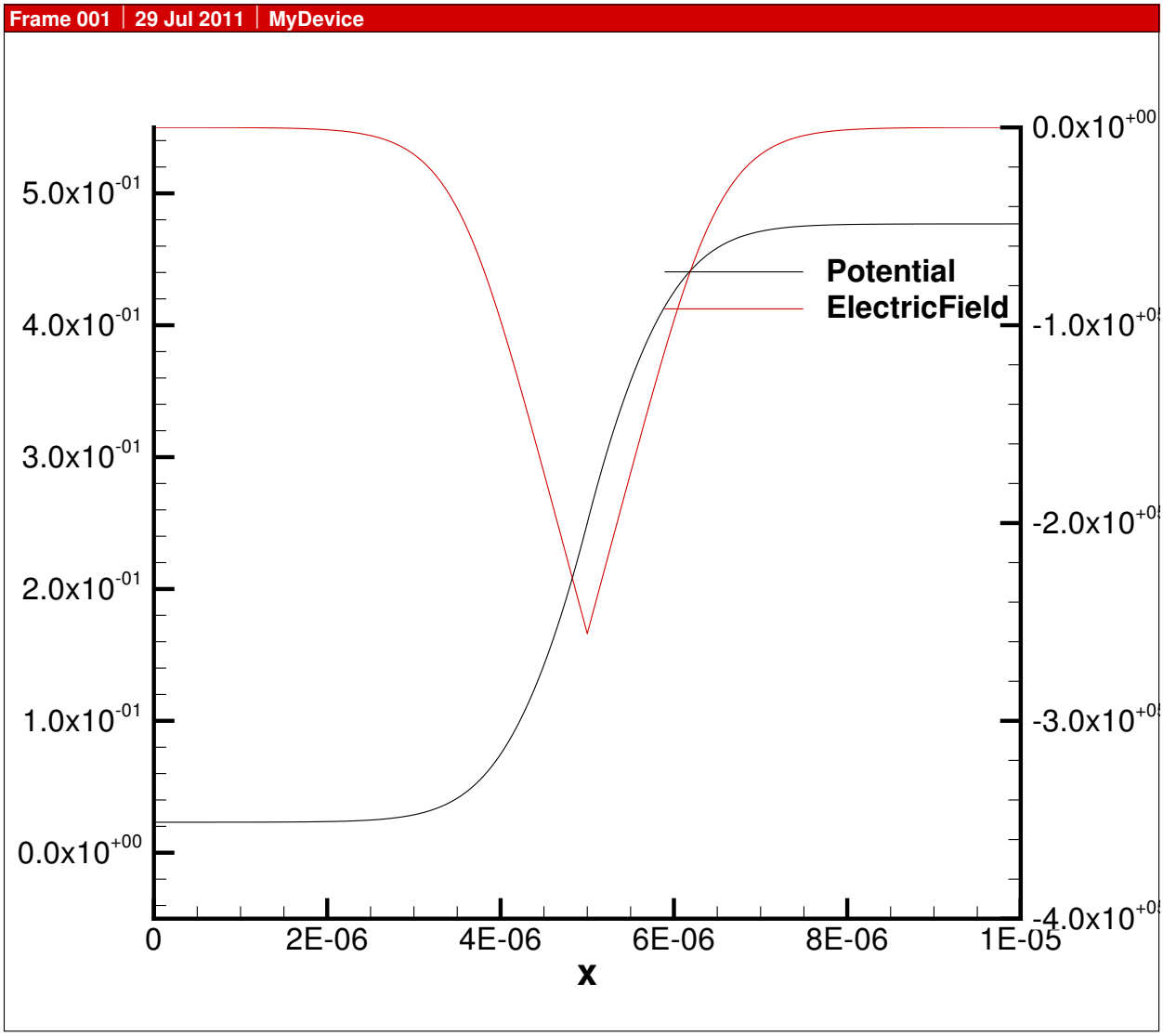

Fig. 13.3: Potential and electric field versus position in 1D diode.

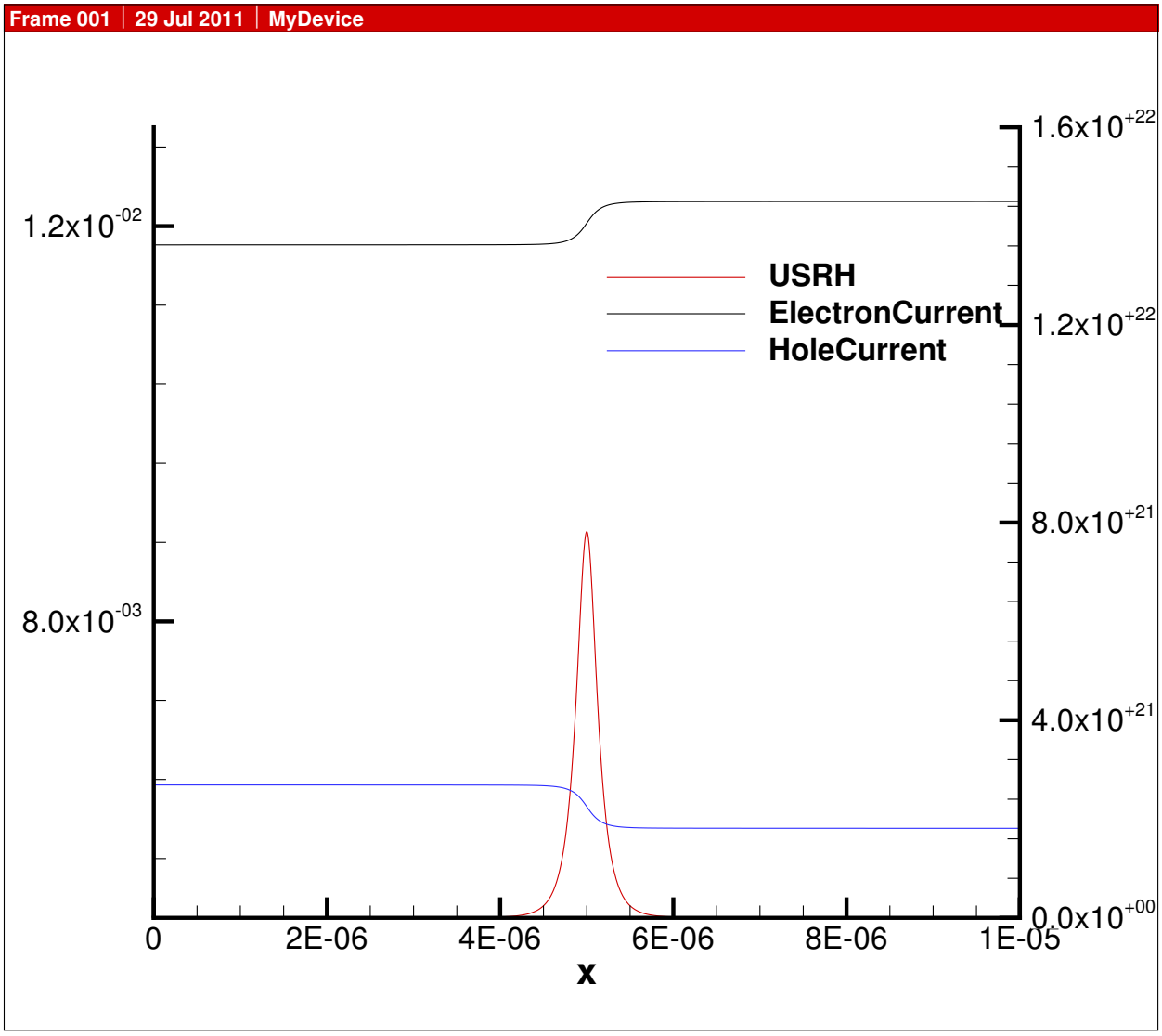

Fig. 13.4: Electron and hole current and recombination.

Chapter 14

Command Reference

14.1 Circuit commands

Commands are for adding circuit elements to the simulation.

devsim.add_circuit_node(*name*, *value*, *variable_update*)

Adds a circuit node for use in circuit or multi-device simulation

Parameters

- name (*str*) – Name of the circuit node being created
- value (*Float , optional*) – initial value (default 0.0)
- variable_update (*{'default', 'log_damp', 'positive'}*) – update type for circuit variable

devsim.circuit_alter(*name*, *param*, *value*)

Alter the value of a circuit element parameter

Parameters

- name (*str*) – Name of the circuit node being created
- param (*str , optional*) – parameter being modified (default 'value')
- value (*Float*) – value for the parameter

devsim.circuit_element(*name*, *value*, *n1*, *n2*, *acreal*, *acimag*)

Adds a circuit element external to the devices

Parameters

- name (*str*) – Name of the circuit element being created. A prefix of 'V' is for voltage source, 'I' for current source, 'R' for resistor, 'L' for inductor, and 'C' for capacitor.
- value (*Float , optional*) – value for the default parameter of the circuit element (default 0.0)

- n1 (*str*) – circuit node
- n2 (*str*) – circuit node
- acreal (*Float , optional*) – real part of AC source for voltage (default 0.0)
- acimag (*Float , optional*) – imag part of AC source for voltage (default 0.0)

devsim.circuit_node_alias(*node*, *alias*)

Create an alias for a circuit node

Parameters

- node (*str*) – circuit node being aliased
- alias (*str*) – alias for the circuit node

devsim.delete_circuit()

Deletes any present circuit and its solutions.

devsim.get_circuit_equation_number(*node*)

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

Parameters

node (*str*) – circuit node

devsim.get_circuit_node_list()

Gets the list of the nodes in the circuit.

devsim.get_circuit_node_value(*solution*, *node*)

Gets the value of a circuit node for a given solution type.

Parameters

- solution (*str , optional*) – name of the solution. 'dcop' is the name for the DC solution (default 'dcop')
- node (*str*) – circuit node of interest

devsim.get_circuit_solution_list()

Gets the list of available circuit solutions.

devsim.set_circuit_node_value(*solution*, *node*, *value*)

Sets the value of a circuit node for a given solution type.

Parameters

- solution (*str , optional*) – name of the solution. 'dcop' is the name for the DC solution (default 'dcop')
- node (*str*) – circuit node of interest
- value (*Float , optional*) – new value (default 0.0)

14.2 Equation commands

Commands for manipulating equations on contacts, interface, and regions

devsim.contact_equation(*device*, *contact*, *name*, *circuit_node*, *edge_charge_model*, *edge_current_model*, *edge_model*, *edge_volume_model*, *element_charge_model*, *element_current_model*, *element_model*, *volume_node0_model*, *volume_node1_model*, *node_charge_model*, *node_current_model*, *node_model*)

Create a contact equation on a device

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command
- name (*str*) – Name of the contact equation being created
- circuit_node (*str, optional*) – Name of the circuit we integrate the flux into
- edge_charge_model (*str, optional*) – Name of the edge model used to determine the charge at this contact
- edge_current_model (*str, optional*) – Name of the edge model used to determine the current flowing out of this contact
- edge_model (*str, optional*) – Name of the edge model being integrated at each edge at this contact
- edge_volume_model (*str, optional*) – Name of the edge model being integrated over the volume of each edge on the contact
- element_charge_model (*str, optional*) – Name of the element edge model used to determine the charge at this contact
- element_current_model (*str, optional*) – Name of the element edge model used to determine the current flowing out of this contact
- element_model (*str, optional*) – Name of the element edge model being integrated at each edge at this contact
- volume_node0_model (*str, optional*) – Name of the element model being integrated over the volume of node 0 of each edge on the contact
- volume_node1_model (*str, optional*) – Name of the element model being integrated over the volume of node 1 of each edge on the contact
- node_charge_model (*str, optional*) – Name of the node model used to determine the charge at this contact
- node_current_model (*str, optional*) – Name of the node model used to determine the current flowing out of this contact

- node_model (*str, optional*) – Name of the node model being integrated at each node at this contact

devsim.custom_equation(*name*, *procedure*)

Custom equation assembly. See *Custom matrix assembly* (page 28) for a description of how the function should be structured.

Parameters

- name (*str*) – Name of the custom equation being created
- procedure (*str*) – The procedure to be called

devsim.delete_contact_equation(*device*, *contact*, *name*)

This command deletes an equation from a contact.

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command
- name (*str*) – Name of the contact equation being deleted

devsim.delete_equation(*device*, *region*, *name*)

This command deletes an equation from a region.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the equation being deleted

devsim.delete_interface_equation(*device*, *interface*, *name*)

This command deletes an equation from an interface.

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command
- name (*str*) – Name of the interface equation being deleted

devsim.equation(*device*, *region*, *name*, *variable_name*, *node_model*, *edge_model*, *edge_volume_model*, *time_node_model*, *element_model*, *volume_node0_model*, *volume_node1_model*, *variable_update*)

Specify an equation to solve on a device

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the equation being created

- `variable_name` (*str*) – Name of the node solution being solved
- `node_model` (*str, optional*) – Name of the node model being integrated at each node in the device volume
- `edge_model` (*str, optional*) – Name of the edge model being integrated over each edge in the device volume
- `edge_volume_model` (*str, optional*) – Name of the edge model being integrated over the volume of each edge in the device volume
- `time_node_model` (*str, optional*) – Name of the time dependent node_model being integrated at each node in the device volume
- `element_model` (*str, optional*) – Name of the element model being integrated over each edge in the device volume
- `volume_node0_model` (*str, optional*) – Name of the element model being integrated over the volume of node 0 of each edge on the contact
- `volume_node1_model` (*str, optional*) – Name of the element model being integrated over the volume of node 1 of each edge on the contact
- `variable_update` (*{'default', 'log_damp', 'positive'}*) – update type for circuit variable

Notes

The integration variables can be changed in 2D for cylindrical coordinate systems by setting the appropriate parameters as described in *Cylindrical coordinate systems* (page 29).

In order to set the node volumes for integration of the `edge_volume_model`, it is possible to do something like this:

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

`devsim.get_contact_equation_command(`*device*, *contact*, *name*`)`

This command gets the options used when creating this contact equation.

Parameters

- `device` (*str*) – The selected device
- `contact` (*str*) – Contact on which to apply this command
- `name` (*str*) – Name of the contact equation being command options returned

`devsim.get_contact_equation_list(`*device*, *contact*`)`

This command gets a list of equations on the specified contact.

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command

devsim.get_equation_command(*device*, *region*, *name*)

This command gets the options used when creating this equation.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the equation being command options returned

devsim.get_equation_list(*device*, *region*)

This command gets a list of equations on the specified region.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region

devsim.get_equation_numbers(*device*, *region*, *equation*, *variable*)

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

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- equation (*str, optional*) – Name of the equation
- variable (*str, optional*) – Name of the variable

devsim.get_interface_equation_command(*device*, *interface*, *name*)

This command gets the options used when creating this interface equation.

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command
- name (*str*) – Name of the interface equation being command options returned

devsim.get_interface_equation_list(*device*, *interface*)

This command gets a list of equations on the specified interface.

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command

devsim.interface_equation(*device*, *interface*, *name*, *name0*, *name1*, *interface_model*, *type*)

Command to specify an equation at an interface

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command
- name (*str*) – Name of the interface equation being created
- name0 (*str, optional*) – Name of the equation coupling in region 0 being created (default 'name')
- name1 (*str, optional*) – Name of the equation coupling in region 1 being created (default 'name')
- interface_model (*str*) – When specified, the bulk equations on both sides of the interface are integrated together. This model is then used to specify how nodal quantities on both sides of the interface are balanced
- type (*{'continuous', 'fluxterm', 'hybrid'} required*) – Specifies the type of boundary condition

14.3 Geometry commands

Commands for getting information about the device structure.

devsim.get_contact_list(*device*)

Gets a list of contacts on a device.

Parameters

device (*str*) – The selected device

devsim.get_device_list()

Gets a list of devices on the simulation.

devsim.get_element_node_list(*device*, *region*, *contact*, *interface*, *reorder*)

Gets a list of nodes for each element on a device, region, contact, or interface.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- contact (*str, optional*) – If specified, gets the element nodes for the contact on the specified region
- interface (*str, optional*) – If specified, gets the element nodes for the interface on the specified region
- reorder (*bool, optional*) – If specified, reorders the element nodes in a manner compatible in meshing software (default False)

devsim.get_interface_list(*device*)

Gets a list of interfaces on a device.

Parameters

device (*str*) – The selected device

devsim.get_region_list(*device*, *contact*, *interface*)

Gets a list of regions on a device, contact, or interface.

Parameters

- device (*str*) – The selected device
- contact (*str, optional*) – If specified, gets the name of the region belonging to this contact on the device
- interface (*str, optional*) – If specified, gets the name of the regions belonging to this interface on the device

devsim.reset_devsim()

Resets all data for clean restart.

14.4 Material commands

Commands for manipulating parameters and material properties

devsim.get_dimension(*device*)

Get the dimension of the device

Parameters

device (*str, optional*) – The selected device

devsim.get_material(*device*, *region*, *contact*)

Returns the material for the specified region

Parameters

- device (*str, optional*) – The selected device
- region (*str, optional*) – The selected region
- contact (*str, optional*) – Contact on which to apply this command

devsim.get_parameter(*device*, *region*, *name*)

Get a parameter on a region, device, or globally.

Parameters

- device (*str, optional*) – The selected device
- region (*str, optional*) – The selected region
- name (*str*) – Name of the parameter name being retrieved

Notes

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

`devsim.get_parameter_list(`*device*, *region*`)`

Get list of parameter names on region, device, or globally

Parameters

- `device` (*str, optional*) – The selected device
- `region` (*str, optional*) – The selected region

Notes

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

`devsim.set_material(`*device*, *region*, *contact*, *material*`)`

Sets the new material for a region

Parameters

- `device` (*str, optional*) – The selected device
- `region` (*str, optional*) – The selected region
- `contact` (*str, optional*) – Contact on which to apply this command
- `material` (*str*) – New material name

`devsim.set_parameter(`*device*, *region*, *name*, *value*`)`

Set a parameter on region, device, or globally

Parameters

- `device` (*str, optional*) – The selected device
- `region` (*str, optional*) – The selected region
- `name` (*str*) – Name of the parameter name being retrieved
- `value` (*any*) – value to set for the parameter

Notes

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

14.5 Meshing commands

Commands for reading and writing meshes

devsim.add_1d_contact(*material*, *mesh*, *name*, *tag*)

Add a contact to a 1D mesh

Parameters

- material (*str*) – material for the contact being created
- mesh (*str*) – Mesh to add the contact to
- name (*str*) – Name for the contact being created
- tag (*str*) – Text label for the position to add the contact

devsim.add_1d_interface(*mesh*, *tag*, *name*)

Add an interface to a 1D mesh

Parameters

- mesh (*str*) – Mesh to add the interface to
- tag (*str*) – Text label for the position to add the interface
- name (*str*) – Name for the interface being created

devsim.add_1d_mesh_line(*mesh*, *tag*, *pos*, *ns*, *ps*)

Add a mesh line to a 1D mesh

Parameters

- mesh (*str*) – Mesh to add the line to
- tag (*str, optional*) – Text label for the position
- pos (*str*) – Position for the mesh point
- ns (*Float, optional*) – Spacing from this point in the negative direction (default ps value)
- ps (*Float*) – Spacing from this point in the positive direction

devsim.add_1d_region(*mesh*, *tag1*, *tag2*, *region*, *material*)

Add a region to a 1D mesh

Parameters

- mesh (*str*) – Mesh to add the line to

- tag1 (*str*) – Text label for the position bounding the region being added
- tag2 (*str*) – Text label for the position bounding the region being added
- region (*str*) – Name for the region being created
- material (*str*) – Material for the region being created

devsim.add_2d_contact(*name*, *material*, *mesh*, *region*, *xl*, *xh*, *yl*, *yh*, *bloat*)

Add an interface to a 2D mesh

Parameters

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

devsim.add_2d_interface(*mesh*, *name*, *region0*, *region1*, *xl*, *xh*, *yl*, *yh*, *bloat*)

Add an interface to a 2D mesh

Parameters

- mesh (*str*) – Mesh to add the interface to
- name (*str*) – Name for the interface being created
- region0 (*str*) – Name of the region included in the interface
- region1 (*str*) – Name of the region included in the interface
- xl (*Float , optional*) – x position for corner of bounding box (default -MAXDOUBLE)
- xh (*Float , optional*) – x position for corner of bounding box (default +MAXDOUBLE)
- yl (*Float , optional*) – y position for corner of bounding box (default -MAXDOUBLE)

- yh (*Float , optional*) – y position for corner of bounding box (default +MAXDOUBLE)
- bloat (*Float , optional*) – Extend bounding box by this amount when search for mesh to include in region (default 1e-10)

devsim.add_2d_mesh_line(*mesh*, *pos*, *ns*, *ps*)

Add a mesh line to a 2D mesh

Parameters

- mesh (*str*) – Mesh to add the line to
- pos (*str*) – Position for the mesh point
- ns (*Float*) – Spacing from this point in the negative direction
- ps (*Float*) – Spacing from this point in the positive direction

devsim.add_2d_region(*mesh*, *region*, *material*, *xl*, *xh*, *yl*, *yh*, *bloat*)

Add a region to a 2D mesh

Parameters

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

devsim.add_gmsh_contact(*gmsh_name*, *material*, *mesh*, *name*, *region*)

Create a mesh to import a Gmsh mesh

Parameters

- gmsh_name (*str*) – physical group name in the Gmsh file
- material (*str*) – material for the contact being created
- mesh (*str*) – name of the mesh being generated
- name (*str*) – name of the contact begin created
- region (*str*) – region that the contact is attached to

devsim.add_gmsh_interface(*gmsh_name*, *mesh*, *name*, *region0*, *region1*)

Create an interface for an imported Gmsh mesh

Parameters

- gmsh_name (*str*) – physical group name in the Gmsh file
- mesh (*str*) – name of the mesh being generated
- name (*str*) – name of the interface begin created
- region0 (*str*) – first region that the interface is attached to
- region1 (*str*) – second region that the interface is attached to

devsim.add_gmsh_region(*gmsh_name*, *mesh*, *region*, *material*)

Create a region for an imported Gmsh mesh

Parameters

- gmsh_name (*str*) – physical group name in the Gmsh file
- mesh (*str*) – name of the mesh being generated
- region (*str*) – name of the region begin created
- material (*str*) – material for the region being created

devsim.create_1d_mesh(*mesh*)

Create a mesh to create a 1D device

Parameters

mesh (*str*) – name of the 1D mesh being created

devsim.create_2d_mesh(*mesh*)

Create a mesh to create a 2D device

Parameters

mesh (*str*) – name of the 2D mesh being created

devsim.create_contact_from_interface(*device*, *region*, *interface*, *material*, *name*)

Creates a contact on a device from an existing interface

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- interface (*str*) – Interface on which to apply this command
- material (*str*) – material for the contact being created
- name (*str*) – name of the contact begin created

devsim.create_device(*mesh*, *device*)

Create a device from a mesh

Parameters

- `mesh` (*str*) – name of the mesh being used to create a device
- `device` (*str*) – name of the device being created

`devsim.create_gmsh_mesh`(*mesh*, *file*, *coordinates*, *elements*, *physical_names*)

Create a mesh to import a Gmsh mesh

Parameters

- `mesh` (*str*) – name of the mesh being generated
- `file` (*str, optional*) – name of the Gmsh mesh file being read into DEVSIM
- `coordinates` (*list, optional*) – List of coordinate positions on mesh.
- `elements` (*list, optional*) – List of elements on the mesh.
- `physical_names` (*list, optional*) – List of names for each contact, interface, and region on mesh.

Notes

This file will import a Gmsh format mesh from a file. Alternatively, the mesh structure may be passed in as as arguments:

`coordinates` is a float list of positions in the mesh. Each coordinate adds an x, y, and z position so that the coordinate list length is 3 times the number of coordinates.

`physical_names` is a list of contact, interface, and region names. It is referenced by index by the `elements` list.

`elements` is a list of elements. Each element adds

- Element Type (float)
  - 0 node
  - 1 edge
  - 2 triangle
  - 3 tetrahedron
- Physical Index
  - This indexes into the `physical_names` list.
- Nodes
  - Each node of the element indexes into the coordinates list.

`devsim.create_interface_from_nodes`(*device*, *name*, *region0*, *region1*, *nodes0*, *nodes1*)

Creates an interface from lists of nodes

Parameters

- `device` (*str*) – The selected device

- name (*str*) – name of the interface begin created
- region0 (*str*) – first region that the interface is attached to
- region1 (*str*) – second region that the interface is attached to
- nodes0 (*str*) – list of nodes for the interface in the first region
- nodes1 (*str*) – list of nodes for the interface in the second region

devsim.delete_device(*device*)

Delete a device and its parameters

Parameters

device (*str*) – name of the device being deleted

devsim.delete_mesh(*mesh*)

Delete a mesh so devices can no longer be instantiated from it.

Parameters

mesh (*str*) – Mesh to delete

devsim.finalize_mesh(*mesh*)

Finalize a mesh so no additional mesh specifications can be added and devices can be created.

Parameters

mesh (*str*) – Mesh to finalize

devsim.get_mesh_list()

Get list of meshes

devsim.load_devices(*file*)

Load devices from a DEVSIM file

Parameters

file (*str*) – name of the file to load the meshes from

devsim.write_devices(*file*, *device*, *type*, *include_test*)

Write a device to a file for visualization or restart

Parameters

- file (*str*) – name of the file to write the meshes to
- device (*str, optional*) – name of the device to write
- type (*{'devsim', 'devsim_data', 'tecplot', 'vtk'}*) – format to use
- include_test (*str*) – Callback function which tests whether a model should be written to the tecplot or vtk format

14.6 Model commands

Commands for defining and evaluating models

devsim.contact_edge_model(*device*, *contact*, *name*, *equation*, *display_type*)

Create an edge model evaluated at a contact

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command
- name (*str*) – Name of the contact edge model being created
- equation (*str*) – Equation used to describe the contact edge model being created
- display_type (*{'vector', 'nodisplay', 'scalar'}*) – Option for output display in graphical viewer

devsim.contact_node_model(*device*, *contact*, *name*, *equation*, *display_type*)

Create an node model evaluated at a contact

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command
- name (*str*) – Name of the contact node model being created
- equation (*str*) – Equation used to describe the contact node model being created
- display_type (*{'scalar', 'nodisplay'}*) – Option for output display in graphical viewer

devsim.cylindrical_edge_couple(*device*, *region*)

This command creates the EdgeCouple model for 2D cylindrical simulation

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region

Notes

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

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

The *devsim.set_parameter()* (page 89) must be used to set

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

`devsim.cylindrical_node_volume(`*device*, *region*`)`

This command creates the `NodeVolume` model for 2D cylindrical simulation

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region

Notes

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

- `ElementCylindricalNodeVolume@en0` (Element Edge Model)
- `ElementCylindricalNodeVolume@en1` (Element Edge Model)
- `CylindricalEdgeNodeVolume@n0` (Edge Model)
- `CylindricalEdgeNodeVolume@n1` (Edge Model)
- `CylindricalNodeVolume` (Node Model)

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

The *devsim.set_parameter()* (page 89) must be used to set

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

`devsim.cylindrical_surface_area(`*device*, *region*`)`

This command creates the `SurfaceArea` model for 2D cylindrical simulation

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region

Notes

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

- `CylindricalSurfaceArea` (Node Model)

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

The *devsim.set_parameter()* (page 89) must be used to set

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

`devsim.debug_triangle_models(`*device*, *region*`)`

Debugging command used in the development of DEVSIM and used in regressions.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region

`devsim.delete_edge_model(`*device*, *region*, *name*`)`

Deletes an edge model from a region

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the edge model being deleted

`devsim.delete_element_model(`*device*, *region*, *name*`)`

Deletes a element model from a region

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the node model being deleted

`devsim.delete_interface_model(`*device*, *interface*, *name*`)`

Deletes an interface model from an interface

Parameters

- `device` (*str*) – The selected device
- `interface` (*str*) – Interface on which to apply this command
- `name` (*str*) – Name of the interface model being deleted

devsim.delete_node_model(*device*, *region*, *name*)

Deletes a node model from a region

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the node model being deleted

devsim.edge_average_model(*device*, *region*, *node_model*, *edge_model*, *derivative*, *average_type*)

Creates an edge model based on the node model values

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- node_model (*str*) – The node model from which we are creating the edge model. If derivative is specified, the edge model is created from nodeModel:derivativeModel
- edge_model (*str*) – The edge model name being created. If derivative is specified, the edge models created are edgeModel:derivativeModel@n0 edgeModel:derivativeModel@n1, which are the derivatives with respect to the derivative model on each side of the edge
- derivative (*str, optional*) – The node model of the variable for which the derivative is being taken. The node model nodeModel:derivativeModel is used to create the resulting edge models.
- average_type (*{'arithmetic', 'geometric', 'gradient', 'negative_gradient'}*) – The node models on both sides of the edge are averaged together to create one of the following types of averages.

Notes

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

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

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

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

devsim.edge_from_node_model(*device*, *region*, *node_model*)

For a node model, creates an 2 edge models referring to the node model value at both ends of the edge.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- node_model (*str*) – The node model from which we are creating the edge model

Notes

For example, to calculate electric field:

devsim.edge_from_node_model(device=device, region=region, node_model="Potential")

devsim.edge_model(*device*, *region*, *name*, *equation*, *display_type*)

Creates an edge model based on an equation

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the edge model being created
- equation (*str*) – Equation used to describe the edge model being created
- display_type (*{'scalar', 'nodisplay', 'vector'}*) – Option for output display in graphical viewer

Notes

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

- model_x_onNode
- model_y_onNode
- model_z_onNode (3D)

This averaging scheme does not produce accurate results, and it is recommended to use the *devsim.element_from_edge_model()* (page 101) to create components better suited for visualization. See *Visualization and post processing* (page 58) for more information about creating data files for external visualization programs.

devsim.edge_solution(*device*, *region*, *name*)

Create node model whose values are set.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the solution being created

devsim.element_from_edge_model(*device*, *region*, *edge_model*, *derivative*)

Creates element edge models from an edge model

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- edge_model (*str*) – The edge model from which we are creating the element model
- derivative (*str, optional*) – The variable we are taking with respect to edge_model

Notes

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

- emodel_x
- emodel_y

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

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

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

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

- emodel_y:variable@en1
- emodel_z:variable@en1
- emodel_x:variable@en2
- emodel_y:variable@en2
- emodel_z:variable@en2
- emodel_x:variable@en3
- emodel_y:variable@en3
- emodel_z:variable@en3

in 3D for each node on a tetrahedral element.

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

devsim.element_from_node_model(*device*, *region*, *node_model*)

Creates element edge models from a node model

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- node_model (*str*) – The node model from which we are creating the edge model

Notes

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

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

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

devsim.element_model(*device*, *region*, *name*, *equation*, *display_type*)

Create a model evaluated on element edges.

Parameters

- device (*str*) – The selected device

- `region` (*str*) – The selected region
- `name` (*str*) – Name of the element edge model being created
- `equation` (*str*) – Equation used to describe the element edge model being created
- `display_type` (*{'scalar', 'nodisplay'}*) – Option for output display in graphical viewer

`devsim.element_pair_from_edge_model(`*device*, *region*, *edge_model*, *derivative*`)`

Creates element edge models from an edge model

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `edge_model` (*str*) – The edge model from which we are creating the element model
- `derivative` (*str, optional*) – The variable we are taking with respect to edge_model

Notes

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

- `emodel_node0_x`
- `emodel_node0_y`
- `emodel_node1_x`
- `emodel_node1_y`

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

- `emodel_node0_x:variable@en0`
- `emodel_node0_y:variable@en0`
- `emodel_node0_x:variable@en1`
- `emodel_node0_y:variable@en1`
- `emodel_node0_x:variable@en2`
- `emodel_node0_y:variable@en2`
- `emodel_node1_x:variable@en0`
- `emodel_node1_y:variable@en0`
- `emodel_node1_x:variable@en1`

- `emodel_node1_y:variable@en1`
- `emodel_node1_x:variable@en2`
- `emodel_node1_y:variable@en2`

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

- `emodel_node0_x:variable@en0`
- `emodel_node0_y:variable@en0`
- `emodel_node0_z:variable@en0`
- `emodel_node0_x:variable@en1`
- `emodel_node0_y:variable@en1`
- `emodel_node0_z:variable@en1`
- `emodel_node0_x:variable@en2`
- `emodel_node0_y:variable@en2`
- `emodel_node0_z:variable@en2`
- `emodel_node0_x:variable@en3`
- `emodel_node0_y:variable@en3`
- `emodel_node0_z:variable@en3`
- `emodel_node1_x:variable@en0`
- `emodel_node1_y:variable@en0`
- `emodel_node1_z:variable@en0`
- `emodel_node1_x:variable@en1`
- `emodel_node1_y:variable@en1`
- `emodel_node1_z:variable@en1`
- `emodel_node1_x:variable@en2`
- `emodel_node1_y:variable@en2`
- `emodel_node1_z:variable@en2`
- `emodel_node1_x:variable@en3`
- `emodel_node1_y:variable@en3`
- `emodel_node1_z:variable@en3`

in 3D for each node on a tetrahedral element.

The label `node0` and `node1` refer to the node on the edge for which the element field average was performed. For example, `node0` signifies that all edges connected to `node0` where used to calculate the element field.

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

`devsim.element_solution(`*device*, *region*, *name*`)`

Create node model whose values are set.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the solution being created

`devsim.get_edge_model_list(`*device*, *region*`)`

Returns a list of the edge models on the device region

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region

`devsim.get_edge_model_values(`*device*, *region*, *name*`)`

Get the edge model values calculated at each edge.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the edge model values being returned as a list

`devsim.get_element_model_list(`*device*, *region*`)`

Returns a list of the element edge models on the device region

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region

`devsim.get_element_model_values(`*device*, *region*, *name*`)`

Get element model values at each element edge

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the element edge model values being returned as a list

devsim.get_interface_model_list(*device*, *interface*)

Returns a list of the interface models on the interface

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command

devsim.get_interface_model_values(*device*, *interface*, *name*)

Gets interface model values evaluated at each interface node.

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command
- name (*str*) – Name of the interface model values being returned as a list

devsim.get_node_model_list(*device*, *region*)

Returns a list of the node models on the device region

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region

devsim.get_node_model_values(*device*, *region*, *name*)

Get node model values evaluated at each node in a region.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the node model values being returned as a list

devsim.interface_model(*device*, *interface*, *equation*)

Create an interface model from an equation.

Parameters

- device (*str*) – The selected device
- interface (*str*) – Interface on which to apply this command
- equation (*str*) – Equation used to describe the interface node model being created

devsim.interface_normal_model(*device*, *region*, *interface*)

Creates edge models whose components are based on direction and distance to an interface

Parameters

- device (*str*) – The selected device

- `region` (*str*) – The selected region
- `interface` (*str*) – Interface on which to apply this command

Notes

This model creates the following edge models:

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

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

`devsim.node_model(`*device*, *region*, *name*, *equation*, *display_type*`)`

Create a node model from an equation.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the node model being created
- `equation` (*str*) – Equation used to describe the node model being created
- `display_type` (*{'scalar', 'nodisplay'}*) – Option for output display in graphical viewer

`devsim.node_solution(`*device*, *region*, *name*`)`

Create node model whose values are set.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the solution being created

`devsim.print_edge_values(`*device*, *region*, *name*`)`

Print edge values for debugging.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the edge model values being printed to the screen

devsim.print_element_values(*device*, *region*, *name*)

Print element values for debugging.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the element edge model values being printed to the screen

devsim.print_node_values(*device*, *region*, *name*)

Print node values for debugging.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the node model values being printed to the screen

devsim.register_function(*name*, *nargs*, *procedure*)

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

Parameters

- name (*str*) – Name of the function
- nargs (*str*) – Number of arguments to the function
- procedure (*str*) – The procedure to be called

devsim.set_edge_values(*device*, *region*, *name*, *init_from*, *values*)

Set edge model values from another edge model, or a list of values.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the edge model being initialized
- init_from (*str, optional*) – Node model we are using to initialize the edge solution
- values (*list, optional*) – List of values for each edge in the region.

devsim.set_element_values(*device*, *region*, *name*, *init_from*, *values*)

Set element model values from another element model, or a list of values.

Parameters

- device (*str*) – The selected device
- region (*str*) – The selected region
- name (*str*) – Name of the element model being initialized

- `init_from` (*str , optional*) – Node model we are using to initialize the element solution
- `values` (*list , optional*) – List of values for each element in the region.

`devsim.set_node_value(`*device*, *region*, *name*, *index*, *value*`)`

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

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the node model being whose value is being set
- `index` (*int*) – Index of node being set
- `value` (*Float*) – Value of node being set

`devsim.set_node_values(`*device*, *region*, *name*, *init_from*, *values*`)`

Set node model values from another node model, or a list of values.

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `name` (*str*) – Name of the node model being initialized
- `init_from` (*str , optional*) – Node model we are using to initialize the node solution
- `values` (*list , optional*) – List of values for each node in the region.

`devsim.symdiff(`*expr*`)`

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

Parameters

`expr` (*str*) – Expression to send to SYMDIFF

`devsim.vector_element_model(`*device*, *region*, *element_model*`)`

Create vector components from an element edge model

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `element_model` (*str*) – The element model for which we are calculating the vector components

Notes

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

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

The primary use of these components are for visualization.

`devsim.vector_gradient(`*device*, *region*, *node_model*, *calc_type*`)`

Creates the vector gradient for noise analysis

Parameters

- `device` (*str*) – The selected device
- `region` (*str*) – The selected region
- `node_model` (*str*) – The node model from which we are creating the edge model
- `calc_type` (*{'default', 'avoidzero'}*) – The node model from which we are creating the edge model

Notes

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

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

It is important not to use these models for simulation, since DEVSIM, does not have a way of evaluating the derivatives of these models. The models can be used for integrating the impedance field, and other postprocessing. The *devsim.element_from_edge_model()* (page 101) command can be used to create gradients for use in a simulation.

14.7 Solver commands

Commands for simulation

devsim.get_contact_charge(*device*, *contact*, *equation*)

Get charge at the contact

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command
- equation (*str*) – Name of the contact equation from which we are retrieving the charge

devsim.get_contact_current(*device*, *contact*, *equation*)

Get current at the contact

Parameters

- device (*str*) – The selected device
- contact (*str*) – Contact on which to apply this command
- equation (*str*) – Name of the contact equation from which we are retrieving the current

devsim.get_matrix_and_rhs(*format*)

Returns matrices and rhs vectors.

Parameters

format (*{'csc', 'csr'} required*) – Option for returned matrix format.

devsim.set_initial_condition(*static_rhs*, *dynamic_rhs*)

Sets the initial condition for subsequent transient solver steps.

Parameters

- static_rhs (*list, optional*) – List of double values for non time-displacement terms in right hand side.
- dynamic_rhs (*list, optional*) – List of double values for time-displacement terms in right hand side.

devsim.solve(*type*, *solver_type*, *absolute_error*, *relative_error*, *maximum_error*, *charge_error*, *gamma*, *tdelta*, *maximum_iterations*, *maximum_divergence*, *frequency*, *output_node*, *info*, *symbolic_iteration_limit*)

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

Parameters

- type (*{'dc', 'ac', 'noise', 'transient_dc', 'transient_bdf1', 'transient_bdf2', 'transient_tr'} required*) – type of solve being performed

- solver_type (*{'direct', 'iterative'} required*) – Linear solver type
- absolute_error (*Float, optional*) – Required update norm in the solve (default 0.0)
- relative_error (*Float, optional*) – Required relative update in the solve (default 0.0)
- maximum_error (*Float, optional*) – Maximum absolute error before solve stops (default MAXDOUBLE)
- charge_error (*Float, optional*) – Relative error between projected and solved charge during transient simulation (default 0.0)
- gamma (*Float, optional*) – Scaling factor for transient time step (default 1.0)
- tdelta (*Float, optional*) – time step (default 0.0)
- maximum_iterations (*int, optional*) – Maximum number of iterations in the DC solve (default 20)
- maximum_divergence (*int, optional*) – Maximum number of diverging iterations during solve (default 20)
- frequency (*Float, optional*) – Frequency for small-signal AC simulation (default 0.0)
- output_node (*str, optional*) – Output circuit node for noise simulation
- info (*bool, optional*) – Solve command return convergence information (default False)
- symbolic_iteration_limit (*int, optional*) – Reuse symbolic matrix factorization after this number of iterations (default 1)

Bibliography

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

[2] 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. doi:10.1002/nme.2579 (https://doi.org/10.1002/nme.2579).

[3] W. B. Joyce and R. W. Dixon. Analytic approximations for the Fermi energy of an ideal Fermi gas. *Applied Physics Letters*, 31(5):354–356, 1977. doi:10.1063/1.89697 (https://doi.org/10.1063/1.89697).

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

[5] John K. Ousterhout. Scripting: higher level programming for the 21st century. *IEEE Computer*, 31:23–30, 1998. doi:10.1109/2.660187 (https://doi.org/10.1109/2.660187).

[6] Gernot Paasch and Susanne Scheinert. Charge carrier density of organics with Gaussian density of states: analytical approximation for the Gauss–Fermi integral. *Journal of Applied Physics*, 107(10):104501, 2010. doi:10.1063/1.3374475 (https://doi.org/10.1063/1.3374475).

[7] Juan Sanchez and Qiusong Chen. Element Edge Based Discretization for TCAD Device Simulation. *TechRxiv*, 7 2021. doi:10.36227/techrxiv.14129081 (https://doi.org/10.36227/techrxiv.14129081).

[8] Juan E. Sanchez and Qiusong Chen. Element edge based discretization for TCAD device simulation. *IEEE Transactions on Electron Devices*, 68(11):5414–5420, 2021. doi:10.1109/TED.2021.3094776 (https://doi.org/10.1109/TED.2021.3094776).

[9] 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. doi:10.1109/T-ED.1969.16566 (https://doi.org/10.1109/T-ED.1969.16566).

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

Index

I

L

M

N

P

R

S

V

W