DEVSIM Manual

Release 2.4.0

DEVSIM LLC

Contents

List of Figures

List of Tables

Chapter 1

Front Matter

1.1 Contact

Table 1.1: Contact

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

1.2 Copyright

1.3 Citing this Work

Please cite this document using the instructions at https://doi.org/10.5281/zenodo.4583208. Instructions for citing the simulator are available from https://doi.org/10.5281/zenodo.1186952. A list of peer review publications concerning the simulator are available at https://devsim.org/introduction.html#publications.

1.4 Documentation License

1.5 Disclaimer

1.6 Trademark

Chapter 2

Release Notes

2.1 Introduction

DEVSIM download and installation instructions are located in *Installation* (page 75). 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 2.4.0

2.2.1 Determine Loaded Math Libraries

To determine the loaded math libraries, use

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

2.2.2 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{}](https://github.com/devsim/umfpack_lgpl)

Please note that this version uses a scheme to provide the needed math library functions when the library is loaded.

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

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

2.2.3 Direct Solver Callback

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

```
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.

2.2.4 Apple M1

On this platform, 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`. Removing this check allows more of the tests to run through to completion.

Bugs

Fix issue [#104](https://github.com/devsim/devsim/issues/104) where the 2D MOSFET example was not fully connected across region interfaces.

```
testing/mos_2d.py
testing/mos_2d_restart.py
testing/mos_2d_restart2.py
```

This was resulting in an FPE during testing on macOS M1.

2.3 Version 2.3.8

2.3.1 Bugs

[@ryan3141](https://github.com/ryan3141) fixed an issue where math functions added with `devsim.register_function` were not available in extended precision model evaluation. The `testing/testfunc_extended.py` test is added to validate the fix.

Update NOTICE with the license files from the various dependencies.

2.4 Version 2.3.7

2.4.1 Apple M1 Support

Intel MKL Pardiso not available, so using system BLAS/LAPACK or openblas by default. In addition, SuperLU, is used instead of the MKL Pardiso. This results in some test failures, based on the use of a different solver, and not the OS architecture.

Extended precision is enabled.

Enabled by running pip install.

The regression results are in this newly created repo:

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

2.4.2 Python Notebook Example With 3D Visualization

A plotting example using `pyvista` is presented in `examples/plotting/visualization.ipynb`. This example was provided by [@simbilod](https://github.com/simbilod).

2.4.3 Bugs

When instantiating a mesh from Gmsh, contact and interface related errors to dimensionality have an improved error message.

2.5 Version 2.3.6

On Windows the `DEVSIM_MATH_LIBS` now 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)

All platforms will search for the Intel MKL by trying several version numbers. When the Intel MKL is not available, the direct solver will switch from Intel MKL Pardiso to SuperLU.

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`

2.6 Release 2.3.1

2.6.1 Python PIP Package

DEVSIM is now available via `pip` for macOS, Linux, and Microsoft Windows. To install this package for your platform:

```
pip install devsim
```

Please see the `INSTALL.md` file in the distribution for more information. These files may be found in the `$CONDA_PREFIX/devsim_data` directory of your `Anaconda` environment. If you are using `venv`, it may be found in the `$VIRTUAL_ENV/devsim_data` of your virtual environment.

2.6.2 Remove Windows MSYS Build

The `msys` build is removed as an available binary package. Windows is still supported through the use of the Visual C++ compiler build.

2.6.3 Build Notes

The compiler for the Linux build are now upgraded to `devtoolset-10` and is now built on `manylinux2014`.

Boost is now added as a submodule, instead of using system libraries or Anaconda Python versions. The Linux build no longer requires Anaconda Python.

2.7 Release 2.2.0

2.7.1 Device and mesh deletion commands

The `devsim.delete_device()` (page 96) 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 96) 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 94) command.

2.7.2 Extended Precision

Extended precision is now available on Windows builds using the Visual Studio Compiler. Note that this precision is not as accurate as the float128 type used on other systems.

2.7.3 Direct Solver

SuperLU has been updated from version 4.3 to version 5.3. It is the solver used when the Intel MKL is not available.

2.7.4 Code Quality

Fixed defects found in Coverity scanning.

2.8 Release 2.1.0

2.8.1 Explicit math library loading

Since the Intel Math Kernel Library started versioning the names of their dynamic link libraries, it has been difficult to maintain a proper Anaconda Python environment when the version has been updated. With this release, it is possible to use any recent version of the Intel MKL. In addition, the user is able to load alternative BLAS/LAPACK math libraries.

2.8.1.1 Intel MKL

From DEVSIM Version 2.1.0 onward, a specific version is not required when loading the Intel MKL. If the Intel MKL is not found, the import of the `devsim` module will fail, and an error message will be printed. This method is the default, and should work when using an Anaconda Python environment with the `mkl` package installed.

When using a different Python distribution, or having an installation in a different place, it is possible to specify the location by modifying the `LD_LIBRARY_PATH` environment variable on `Linux`, or using `DYLD_LIBRARY_PATH` on `Apple macOS`. The explicit path may be set to the MKL math libraries may be set using the method in the next section.

2.8.1.2 Loading other math libraries

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 MKL 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
```

2.8.2 Direct solver selection

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

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

The following options are available:

- `mkl_pardiso` Intel MKL Pardiso
- `superlu` SuperLU 4.3

The default is `mkl_pardiso` when the Intel MKL is loaded. Otherwise, the default will switch to `superlu`.

2.8.3 Kahan summation in extended precision mode

The `kahan3` and `kahan4` functions are now using the Kahan summation algorithm for extended precision model evaluation. Previously, this algorithm was replaced with 128-bit floating point addition and subtraction in releases that support extended precision mode. 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` test has been added.

2.8.4 Visual Studio 2022

The Microsoft Windows `win64` release version is now built using the Visual Studio 2022 compiler. For users needing extended precision on the Windows platform, the `msys` build is recommended.

2.9 Release 2.0.1

2.9.1 Update documentation files

The following files were updated in the text documentation distributed with the software.

- `CONTRIBUTING.md`
- `INSTALL.md`
- `README.md`

This was done to create a version to coincide with this paper in the Journal of Open Source Software.

> Sanchez, J. E., (2022). DEVSIM: A TCAD Semiconductor Device Simulator. Journal of Open Source Software, 7(70), 3898, https://doi.org/10.21105/joss.03898.

2.9.2 Update MKL Version

The release version of this software is build against version 2 of the Intel MKL, which corresponds to the latest version of Anaconda Python.

2.10 Release 2.0.0

2.10.1 New Major Version

Based on the change in the sections *Contact and Interface Equation Commands* (page 9) and *Transient Simulation* (page 10). The major version of the software has been updated to 2. Existing scripts may need to be updated for this change.

2.10.2 Contact and Interface Equation Commands

The previously deprecated `variable_name` option is no longer accepted by the `devsim.contact_equation()` (page 83) and `devsim.interface_equation()` (page 86) commands. This has been updated in the documentation.

2.10.3 Documentation

2.10.3.1 Manual

- Fixed unit in description for example in *1D Capacitor* (page 117).
- Added DOI to referenced papers and updated bibliography style in biblography.
- Added reference to *Equation and Models* (page 37) with additional information about element assembly.
- Updated *Equation and Models* (page 37) and *1D Capacitor* (page 117) to remove `variable_name` option from `devsim.contact_equation()` (page 83) and `devsim.interface_equation()` (page 86).

2.10.3.2 Documentation Files

Some out of date files (e.g. RELEASE, INSTALL, ...) have been removed. The `README.md` has been updated and the `INSTALL.md` has been added.

2.10.4 Python Packages

The rampbias function in the devsim.python_packages.ramp module has been fixed to properly reduce the bias when there is a convergence failure.

Python 2.7 specific instructions in *User Interface* (page 61) has been removed.

2.10.5 Solver

2.10.5.1 Transient Simulation

- Fixed bug with transient_tr (trapezoidal) time integration method in the *devsim.solve()* (page 112) command where the wrong sign was used to integrate previous time steps.
- Fixed bug in the charge error calculation, which calculates the simulation result with that a forward difference projection.
- Added testing/transient_rc.py test which compares simulation with analytic result for RC circuit.
- Added *devsim.set_initial_condition()* (page 111) to set initial transient condition as alternative to using the transient_dc option to the *devsim.solve()* (page 112) command. Suitable options for this command may be provided from the get_matrix_and_rhs() command.

2.10.5.2 Convergence Tests

The maximum_error and maximum_divergence options where added to the *devsim.solve()* (page 112) 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.

2.10.5.3 Verbosity

During the *devsim.solve()* (page 112), circuit node and circuit solution information is no longer printed to the screen for the default verbosity level. In addition, the number of equations per device and region is no longer displayed at the start of the first iteration.

2.10.5.4 Intel Math Kernel Library

The Intel Math Kernel Library now uses versioned library names. Binary releases are now updated against the latest versioned dll names from MKL available in the Anaconda Python distribution.

2.10.5.5 SuperLU

The code now supports newer versions of `SuperLU`. The release version is still using SuperLU 4.3 for the iterative solution method, and the Intel MKL Pardiso for the direct solve method.

2.10.5.6 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.

2.10.6 Add Interface supporting Periodic Boundary Conditions

The *devsim.create_interface_from_nodes()* (page 95) command makes it possible to create an interface with non coincident nodes. This enables the use of periodic boundary conditions.

2.10.7 Build Scripts

The build scripts have been updated on all platforms to be less dependent on specific Python 3 versions.

An updated Fedora build script has been added. It uses the system installed `SuperLU` as the direct solver.

2.11 Release 1.6.0

2.11.1 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.

2.11.2 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`.

2.11.3 Maximum Divergence Count

If the Newton iteration errors keep increasing for 20 iterations in a row, then the simulator stops. This limit was previously 5. This gives a chance for a solution to be found, when there is a poor initial guess.

2.11.4 Mesh Visualization Element Orientation

Elements written to the `tecplot` format in 2d and 3d have node orderings compatible with the element connectivity in visualization formats. Specifying the `reorder=True` option in *devsim.get_element_node_list()* (page 87) will result in node ordering compatible with meshing and visualization software.

2.11.5 Figure annotation

Fig. 4.2 has been updated, showing the `EdgeNodeVolume`.

2.11.6 Citation

The *Citing this Work* (page 1) section has been added with information on how to cite the manual and the simulator.

2.11.7 Documentation License

The license terms have been changed in *Documentation License* (page 1) so that derivative works are allowed.

2.11.8 Online Forum

The online forum for discussion about the software has moved to https://forum.devsim.org. This has been updated in *Contact* (page 1).

2.12 Release 1.5.1

2.12.1 Installation Script

A new installation script is in the base directory of the package. It provides instructions of completing the installation to the `python` environment without having to set the `PYTHONPATH` environment variable. It notifies the user of missing components to finish the installation within an `Anaconda` or `Miniconda` environment.

To use the script, use the following command inside of the `devsim` directory.

```
python install.py
```

The install script will write a file named `lib/setup.py`, which can be used to complete the installation using `pip`. The script provides instructions for the installation and deinstallation of `devsim`.

```
INFO: Writing setup.py
INFO:
INFO: Please type the following command to install devsim:
INFO: pip install -e lib
INFO:
INFO: To remove the file, type:
INFO: pip uninstall devsim
```

2.12.2 Math Functions Table

The list of available math functions, Table 10.2, has been reformatted, and parts have been split into the tables referenced in the next few sections.

2.12.3 Error Functions

The following inverse functions and their derivatives are now available in the model interpreter, and also listed in Table 10.3.

- `erf_inv` Inverse Error Function
- `erfc_inv` Inverse Complimentary Error Function
- `derf_invdx` Derivative of Inverse Error Function
- `derfc_invdx` Derivative of Complimentary Inverse Error Function

2.12.4 Fermi Integral

The Joyce-Dixon approximation [4] for the Fermi integral and its inverse are now calculated with extended floating point precision, when extended precision is enabled. These functions are now listed in Table 10.4.

- `Fermi` Fermi integral
- `dFermidx` Derivative of Fermi integral
- `InvFermi` Inverse Fermi integral
- `dInvFermidx` Derivative of inverse Fermi Integral

The following examples are available:

- `testing/Fermi1.py` Fermi integral
- `testing/Fermi1_float128.py` Fermi integral in extended floating point precision

2.12.5 Gauss-Fermi Integral

The Gauss-Fermi Integral, using Paasch's equations [7] are now available, and are listed in Table 10.5..

- `gfi` Gauss-Fermi Integral
- `dgfidx` Derivative of Gauss-Fermi Integral
- `igfi` Inverse Gauss-Fermi Integral
- `digfidx` Derivative of Inverse Gauss-Fermi Integral

Each of these functions take two arguments, `zeta` and `s`. The derivatives with respect to the first argument are provided.

The following examples are available:

- `testing/GaussFermi.py` Gauss-Fermi integral
- `testing/GaussFermi.py` Gauss-Fermi integral with extended floating point precision

2.13 Release 1.5.0

The *`devsim.custom_equation()`* (page 83) command has been modified to require a third return value. This boolean value denotes whether the matrix entries should be row permutated or not. For the bulk equations this value should be `True`. For interface and contact boundary conditions, this value should be `False`. More information is available in *Custom Matrix assembly* (page 48).

It is now possible to replace an existing `custom_equation`.

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

The `EdgeNodeVolume` model is now available for the volume contained by an edge and is referenced in *Edge models* (page 42).

The *`devsim.equation()`* (page 84) command has removed support for the `volume_model` option. It has been replaced with:

- `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) now 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), described in *June 7, 2015* (page 28).

The integration parameters for edge_volume_model are set with

- edge_node0_volume_model (default EdgeNodeVolume *Edge models* (page 42))
- 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 43))
- element_node1_volume_model (default ElementNodeVolume)

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

2.14 Release 1.4.14

2.14.1 Platforms

Windows 32 bit is no longer supported. Binary releases of the Visual Studio 2019 MSYS2/Mingw-w64 64-bit builds are still available online for Microsoft Windows 10.

On Linux, the releases are now on Centos 7, as Centos 6 has reached its end of life on November 30, 2020.

Please see *Supported platforms* (page 75) for more information.

For future development, C++17 is now the recommended C++ compiler standard.

2.15 Release 1.4.13

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 112) 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.

2.16 Release 1.4.12

Element assembly for calculation of current and charges from the device into the circuit equation are fixed. These tests are added:

- testing/cap_2d_edge.py
- testing/cap_2d_element.py
- testing/cap_3d_edge.py
- testing/cap_3d_element.py

The edge variant is using standard edge based assembly, and the element variant is using element-based assembly.

2.17 Release 1.4.11

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

Fixed issue where command option names where not always shown in the documentation.

The platform specific notes now clarify that any version of Python 3 (3.6 or higher) is supported.

- linux.txt
- windows.txt
- macos.txt

2.18 Release 1.4.10

Fixed crash when evaluating element edge model in 3D.

Fixed potential error using *devsim.delete_node_model()* (page 99) and similar deletion commands.

2.19 Release 1.4.9

Support for loading mesh files containing element edge data.

2.20 Release 1.4.8

In transient mode, the convergence test was flawed so that the `charge_error` was the only convergence check required for convergence. The software now ensures all convergence criteria are met.

2.21 Release 1.4.7

2.21.1 Models

In the simple physics models, the sign for time-derivative terms was wrong for the electron and hole continuity equations. This affects small-signal and noise simulations. The example at `examples/diode/ssac_diode.py` was updated to reflect the change.

2.21.2 Platforms

Fix build script issue for `Apple macOS` on Travis CI, updated the compiler to `g++-9`.

Update `Centos 6` build from `devtoolset-6` to `devtoolset-8`.

2.22 Release 1.4.6

2.22.1 Version Information

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

```
python info.py
{'copyright': 'Copyright © 2009-2020 DEVSIM LLC', 'direct_solver': 'mkl_pardiso',
↪ 'extended_precision': True, 'license': 'Apache License, Version 2.0', 'version
↪': '1.4.6', 'website': 'https://devsim.org'}
```

2.22.2 Extended Precision

The example `examples/diode/gmsh_diode3d_float128.py` provides an example where extended precision is enabled.

2.22.3 Python Formatting

The `Python` scripts in the `examples` and `testing` directories have been reformatted to be more consistent with language standards.

2.22.4 Platforms

`Microsoft Windows 10` is supported and is now compiled using Microsoft Visual Studio 2019.

`Microsoft Windows 7` is no longer supported, as Microsoft has dropped support as of January 14, 2020.

2.22.5 External Meshing

Support for reading meshes from `Genius Device Simulator` has been completely removed from `DEVSIM`.

2.23 Release 1.4.5

An `MSYS2/Mingw-w64` build is available for 64-bit `Microsoft Windows`. This build, labeled `devsim_msys_v1.4.5`, enables the use of the 128-bit floating point precision already available on the `Apple macOS` and `Linux` platforms.

2.24 Release 1.4.4

2.24.1 Bug Fixes

An intermittent crash on `Microsoft Windows 10` was occuring at the end of the program. It is now fixed.

2.24.2 Documentation

A file named `CHANGES.md` is now distributed with `DEVSIM`, detailing changes to the program.

2.24.3 Internal changes

- Regression system script refactored to Python
- Refactored threading code using `C++11` function
- Refactored timing functions for verbose mode using `C++11` functions
- Refactored FPE detection code to `C++11` standard

2.25 Release 1.4.3

Fix failures with the following commands:

- *devsim.delete_edge_model()* (page 98)
- *devsim.delete_element_model()* (page 99)
- *devsim.delete_interface_model()* (page 99)
- *devsim.delete_node_model()* (page 99)

2.26 Release 1.4.2

In this release there are the following improvements.

- Errors due to floating point exceptions and failed matrix factorization are not fatal.
- The `Apple macOS` release fixes runtime issues with `macOS 10.13 (High Sierra)`.
- The provided binary releases utilize more libraries from `Anaconda`.

2.27 Release 1.4.1

2.27.1 Math Functions

The `cosh`, `sinh`, `tanh`, are now available math functions. Please see Table 10.2. In addition, all of the functions in the table, except for Fermi and inverse Fermi functions, are evaluate in extended precision mode. This mode may be enabled using the parameters discussed in *Extended Precision* (page 24).

2.27.2 Element Model Memory Leak

A large memory leak was occurring during the evaluation of element edge models created with *devsim.element_from_edge_model()* (page 101). It is now fixed and memory usage is now stable when these models are evaluated.

2.27.3 Python 3 API Memory Leak

A small memory leak could occur when *devsim* (page 81) functions where called, or when data was returned. These reference counting issues are now fixed.

2.28 Release 1.4.0

The *devsim.custom_equation()* (page 83) and *devsim.register_function()* (page 108) commands take Python functions, instead of the a string with the function name.

The following commands are available to store data on edges and element edges:

- *devsim.edge_solution()* (page 101)
- *devsim.set_edge_values()* (page 108)
- *devsim.element_solution()* (page 105)
- *devsim.set_element_values()* (page 109)

2.29 Release 1.3.0

2.29.1 Python 3 Examples

All of the Tcl regression tests in the testing directory have been converted to Python 3. These tests serve as examples for features that were previously only tested using Tcl scripting.

2.29.2 Tcl Support Deprecated

Tcl support is deprecated and will be removed in a future release of the software.

2.29.3 Binary Releases

2.29.3.1 Scripting Languages

`Python 3` is now the only scripting language in the releases available from:

https://github.com/devsim/devsim/releases

2.29.3.2 Math Library

The `Microsoft Windows` version now uses `Intel MKL Pardiso` for direct matrix factorization. Both `Linux` and `Apple macOS` have been using `Intel MKL Pardiso` since *November 1, 2015* (page 26). Binary releases for all operating systems use BLAS/LAPACK routines from `Intel Math Kernel Library`.

2.30 Release 1.2.0

`DEVSIM` releases have better support for `Python 3`. Using the stable ABI, the software is able to run newer `Python 3` releases, without rebuilding the software.

Support for `Python 2.7` has been removed.

The banner has been removed when the `DEVSIM` module is imported.

The `symdiff` python module is now part of the `DEVSIM` release. This module has additional features not available using the *devsim.symdiff()* (page 110) command from `DEVSIM`. By first setting the `PYTHONPATH` variable to the `lib` directory in the `DEVSIM` distribution, `symdiff` is loaded by using

```
import symdiff
```

Documentation is available in the `doc` directory of this distribution. Examples are available in the `examples/symdiff` directory.

2.31 Release 1.1.0

The Bernoulli function, $B(x)$,

$$B\left(x\right) = \frac{x}{\mathrm{e}^{x} - 1}$$

and its derivative,

$$dBdx\left(x\right) = \frac{\mathrm{e}^{x} - 1 - x\mathrm{e}^{x}}{\left(\mathrm{e}^{x} - 1\right)^{2}}$$

have been refactored. They are used to calculate electron and hole current densities using the Scharfetter-Gummel method [10].

The Bernoulli function has numerical issues when x approaches `0` and requires special evaluation. In this release, `DEVSIM`, takes advantage of `C++11` math library functions for evaluating the denominator.

In addition, these functions are evaluated with extended precision, when this mode is enabled in the simulator. This mode is described in *Extended Precision* (page 24) and controlling parameters are in *Parameters controlling program behavior.* (page 52).

Users should expect that simulation results should change in the number of solver iterations and small differences in simulation results. This and other functions are listed in *Predefined Functions* (page 67).

2.32 Release 1.0.0

2.32.1 Documentation

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

2.32.2 Version

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

2.32.3 Operating Systems

The `Microsoft Windows` 32-bit operating system is now supported in addition to the platforms listed in *Supported platforms* (page 75).

2.32.4 Python Support

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

- 2.7
- 3.6
- 3.7

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

```
import devsim
```

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

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

```
import devsim.python_packages
```

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

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

```
conda install mkl
```

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

```
conda install sqlite zlib
```

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

```
conda install numpy
```

Please see *User Interface* (page 61) and *Installation* (page 75) for more information.

2.32.5 GMSH Support

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

- Save the file in the 2.2 format from Gmsh
- Parse the 4.0 file, and then use *Custom mesh loading using scripting* (page 58)

A future release of DEVSIM will provide this capability.

2.32.6 CGNS Support

Support for loading CGNS files is deprecated, and is no longer part of the official releases. Please see *Using an external mesher* (page 57) for more information about importing meshes from other tools.

2.33 July 20, 2018

2.33.1 Documentation

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

2.33.2 Python 3 Support

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

2.33.3 Element Information

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

2.33.4 Interface Boundary Condition

The `type=hybrid` option is now available for the *devsim.interface_equation()* (page 86) command. Please see *Interface equation assembly* (page 46) for information about boundary conditions.

2.33.5 Interace Equation Coupling

The `name0`, and `name1` options are now available for the *devsim.interface_equation()* (page 86) command. They make it possible to couple dissimilar equation names across regions.

2.33.6 Interface and Contact Surface Area

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

2.33.7 Bug Fixes

- The *devsim.interface_equation()* (page 86) command is fixed for `type=fluxterm` boundary conditions on the interface.
- The *devsim.get_material()* (page 89), and *devsim.set_material()* (page 90) handle the `contact` option.
- Interface equation assembly skips nodes when an interface node is shared with a contact.

2.33.8 Extended Precision

The following new parameters are available:

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

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

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

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

2.34 May 15, 2017

2.34.1 Platforms

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

2.34.2 Binary Releases

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

2.34.3 Bug Fixes

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

2.34.4 Enhancements

- Build scripts are provided to build on various platforms.
- `DEVSIM` mesh format stores elements, instead of just nodes, for contact and interfaces
- The *devsim.create_gmsh_mesh()* (page 94) command can be used to create a device from a provided list of elements.

2.34.5 Example Availability

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

2.35 February 6, 2016

DEVSIM is now covered by the Apache License, Version 2.0 [11]. Please see the NOTICE and LICENSE file for more information.

2.36 November 24, 2015

2.36.1 Python Help

The Python interpreter now has documentation for each command, derived from the documentation in the manual. For example, help for the *devsim.solve()* (page 112) can be found using:

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

2.36.2 Manual Updates

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

2.37 November 1, 2015

2.37.1 Convergence Info

The *devsim.solve()* (page 112) now supports the info option. The solve command will then return convergence information.

2.37.2 Python Interpreter Changes

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

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

in the Python interpreter.

2.37.3 Platform Improvements and Binary Availability

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

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

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

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

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

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

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

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

2.38 September 6, 2015

The *devsim.set_node_values()* (page 109) takes a new option, `values`. It is a list containing values to set for all of the nodes in a region.

The following new commands have been added:

- *devsim.get_equation_list()* (page 86)
- *devsim.get_contact_equation_list()* (page 85)
- *devsim.get_interface_equation_list()* (page 86)
- *devsim.delete_equation()* (page 84)
- *devsim.delete_contact_equation()* (page 84)
- *devsim.delete_interface_equation()* (page 84)
- *devsim.get_equation_command()* (page 86)
- *devsim.get_contact_equation_command()* (page 85)
- *devsim.get_interface_equation_command()* (page 86)

2.39 August 10, 2015

The *devsim.create_contact_from_interface()* (page 94) may be used to create a contact at the location of an interface. This is useful when contact boundary conditions are needed for a region connected to the interface.

2.40 July 16, 2015

The *devsim.set_node_value()* (page 109) was not properly setting the value. This issue is now resolved.

2.41 June 7, 2015

The *devsim.equation()* (page 84) now suppports the `edge_volume_model`. This makes it possible to integrate edge quantities properly so that it is integrated with respect to the volume on nodes of the edge. To set the node volumes for integration, it is necessary to define a model for the node volumes on both nodes of the edge. For example:

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

For the cylindrical coordinate system in 2D, please see *Cylindrical Coordinate Systems* (page 49).

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

2.42 October 4, 2014

2.42.1 Platform Availability

The software is now supported on the `Microsoft Windows`. Please see *Supported platforms* (page 75) for more information.

2.43 December 25, 2013

2.43.1 Binary Availability

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

- `macOS 10.10 (Yosemite)`

- Red Hat Enterprise Linux 6
- Ubuntu 12.04 (LTS)

Please see *Installation* (page 75) for more information.

2.43.2 Platforms

macOS 10.10 (Yosemite) is now supported. Support for 32 bit is no longer supported on this platform, since the operating system is only released as 64 bit.

Regression data will no longer be maintained in the source code repository for 32 bit versions of Ubuntu 12.04 (LTS) and Red Hat Enterprise Linux 6. Building and running on these platforms will still be supported.

2.43.3 Source code improvements

The source code has been improved to compile on macOS 10.10 (Yosemite) and to comply with C++11 language standards. Some of the structure of the project has been reorganized. These changes to the infrastructure will help to keep the program maintainable and useable into the future.

2.44 September 8, 2013

2.44.1 Convergence

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

2.44.2 Bernoulli Function Derivative Evaluation

The dBdx math function has been improved to reduce overflow.

2.44.3 Default Edge Model

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

2.45 August 14, 2013

2.45.1 SYMDIFF functions

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

2.45.2 Default Node Models

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

2.45.3 Set Node Value

It is now possible to use the *devsim.set_node_value()* (page 109) to set a uniform value or indexed value on a node model.

2.45.4 Fix Edge Average Model

Fixed issue with *devsim.edge_average_model()* (page 99) during serialization to the DEVSIM format.

2.46 July 29, 2013

2.46.1 DEVSIM is open source

DEVSIM is now an open source project and is available from https://github.com/devsim/devsim. License information may be found in *DEVSIM License* (page 77). If you would like to participate in this project or need support, please contact us using the information in *Contact* (page 1). Installation instructions may be found in *Installation* (page 75).

2.46.2 Build

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

2.46.3 Contact Material

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

2.46.4 External Meshing

Please see *Using an external mesher* (page 57) for more information about importing meshes from other tools.

Genius **Mesh Import** DEVSIM can now read meshes written from Genius Device Simulator. *Support is no longer available in recent versions releases.*

Gmsh **Mesh Import** DEVSIM reads version 2.1 and 2.2 meshes from Gmsh. Version 2.0 is no longer supported. Please see *Gmsh* (page 57) for more information.

2.46.5 Math Functions

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

2.46.6 Test directory structure

Platform specific results are stored in a hierarchical fashion.

Chapter 3

Introduction

3.1 Overview

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

> DEVSIM is available from https://devsim.org. The source code is available under the terms of the Apache License Version 2.0 [11]. Examples are released under the Apache License Version 2.0 [11]. Contributions to this project are welcome in the form of bug reporting, documentation, modeling, and feature implementation.

3.2 Goals

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

DEVSIM has a scripting language interface (*User Interface* (page 61)). 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 65)) is the symbolic expression parser used to allow the formulation of device equations in terms of models and parameters. Using symbolic differentiation, the required partial derivatives can be generated, or provided by the user. DEVSIM then assembles these equations over the mesh.

3.3 Structures

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

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

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

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

3.4 Equation assembly

Equation assembly of models is discussed in *Equation and Models* (page 37).

3.5 Parameters

Parameters may be specified globally, or for a specific device or region. Alternatively, parameters may be based on the material type of the regions. Usage is discussed in *Parameters* (page 51).

3.6 Circuits

Circuit boundary conditions allow multi-device simulation. They are also required for setting sources and their response for AC and noise analysis. Circuit elements, such as voltage sources, current sources, resistors, capacitors, and inductors may be specified. This is further discussed in *Circuits* (page 53).

3.7 Meshing

Meshing is discussed in *Meshing* (page 55).

3.8 Analysis

`DEVSIM` offers a range of simulation algorithms. They are discussed in more detail in *Solver* (page 59).

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

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

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

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

3.9 Scripting interface

The scripting interface to `DEVSIM` is discussed in *User Interface* (page 61).

3.10 Expression parser

The expression parser is discussed in *SYMDIFF* (page 65).

3.11 Visualization and postprocessing

Visualization is discussed in *Visualization* (page 73).

3.12 Installation

Installation is discussed in *Installation* (page 75).

3.13 Additional information

Additional information is discussed in *Additional Information* (page 77).

3.14 Examples

Examples are discussed in the remaining chapters beginning with *Example Overview* (page 113).

Chapter 4

Equation and Models

4.1 Overview

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

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

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

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

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

Additional detail concerning the discussion that follows is available in [8, 9].

In Fig. 4.1, 2D mesh elements are depicted. The shaded area around the center node is referred to as the node volume, and it is used for the volume integration. The lines from the center node to other nodes are referred to as edges. The flux through the edge are integrated with respect to the perpendicular bisectors (dashed lines) crossing each triangle edge.

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

There are a default set of models created in each region upon initialization of a device, and are typically based on the geometrical attributes. These are described in the following sections. Models required for describing the device behavior are created using the equation parser described in *SYMDIFF* (page 65). For special situations, custom matrix assembly is also available and is discussed in *Custom Matrix assembly* (page 48).

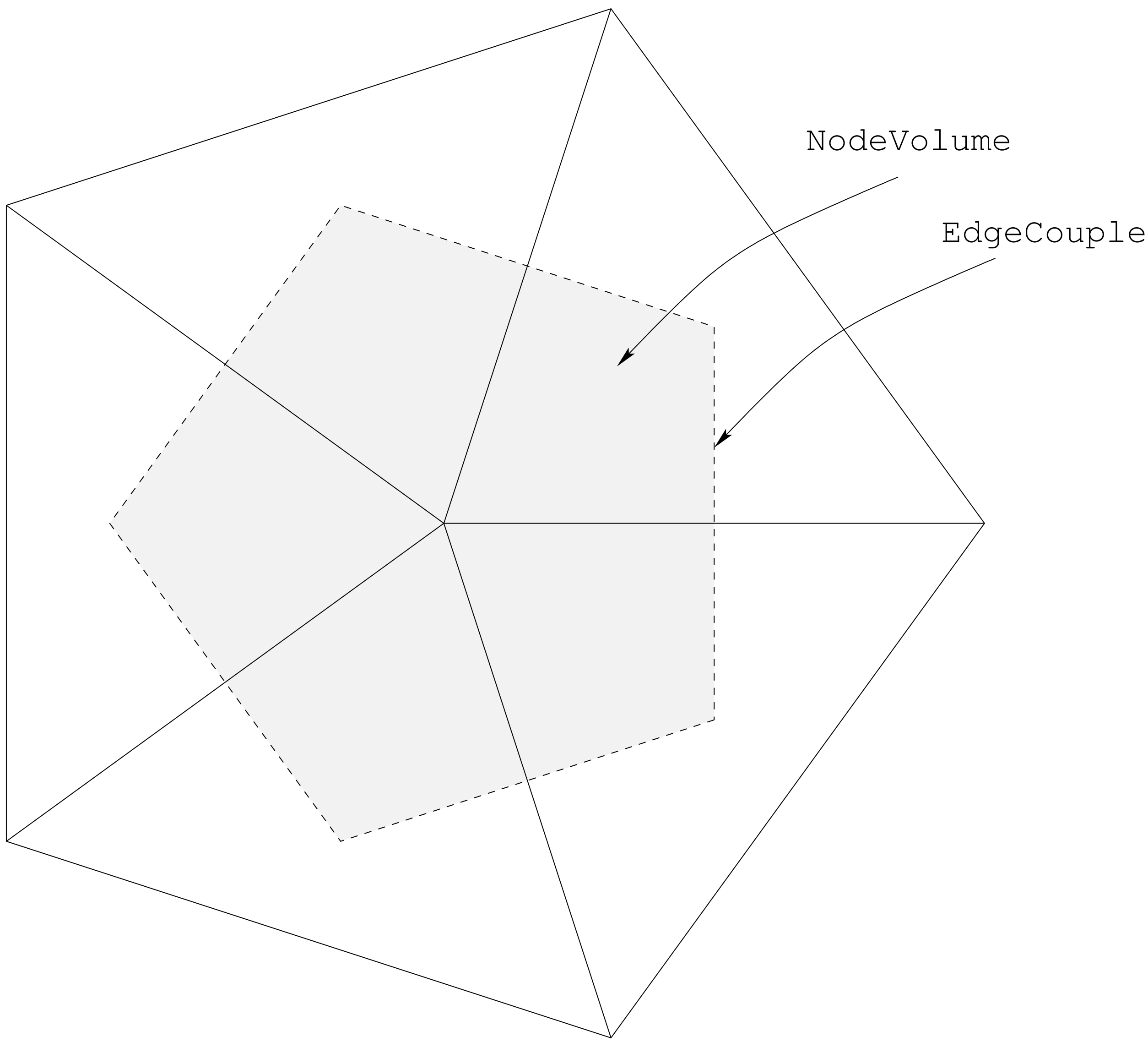

Fig. 4.1: Mesh elements in 2D.

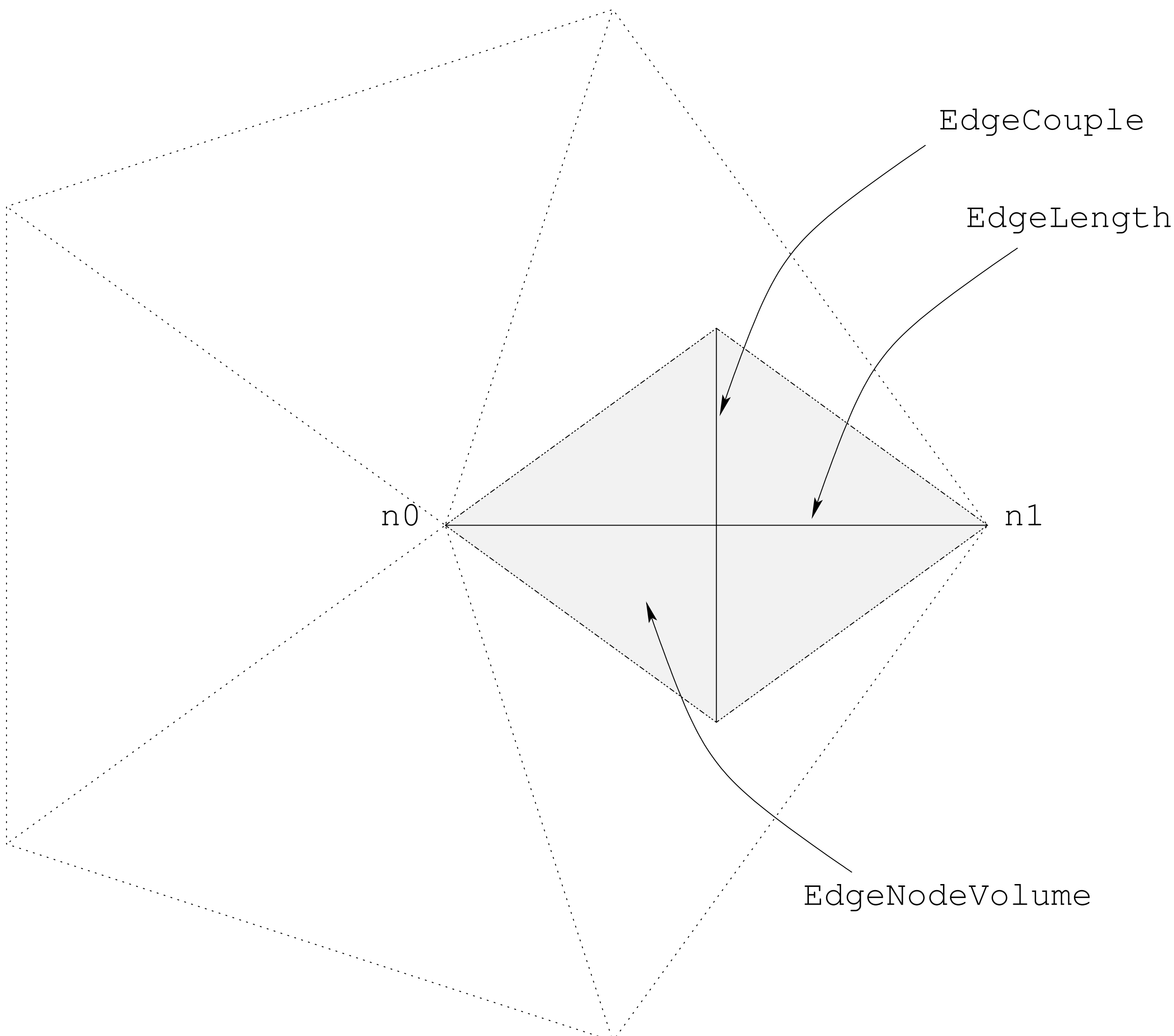

Fig. 4.2: Edge model constructs in 2D.

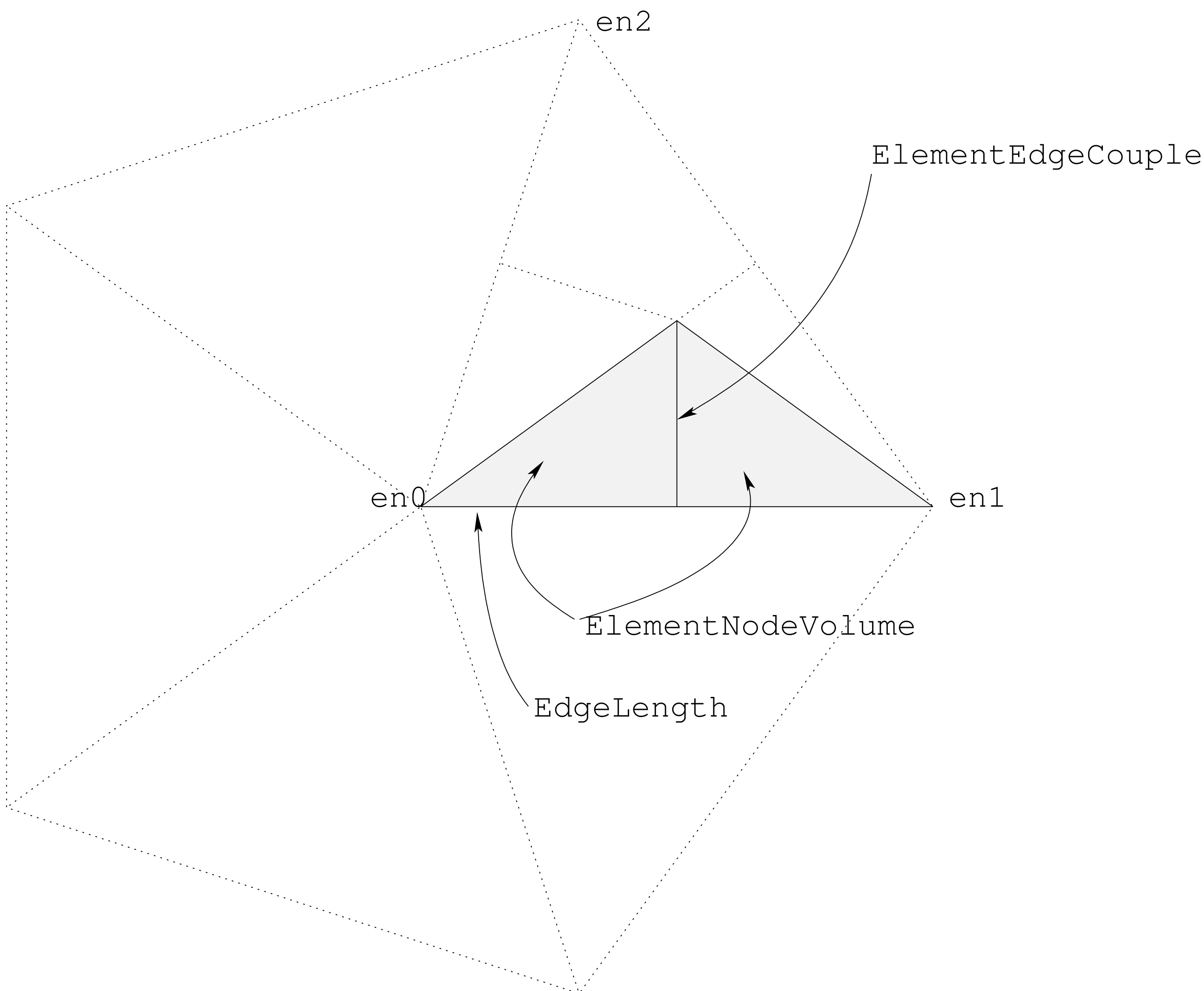

Fig. 4.3: Element edge model constructs in 2D.

4.2 Bulk models

4.2.1 Node models

Node models may be specified in terms of other node models, mathematical functions, and parameters on the device. The simplest model is the node solution, and it represents the solution variables being solved for. Node models automatically created for a region are listed in Table 4.1.

In this example, we present an implementation of Shockley Read Hall recombination [5].

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

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

The `diff` function tells the equation parser to take the derivative of the original expression, with respect to the variable specified as the second argument. During equation assembly, these derivatives are required in order to converge upon a solution. The `simplify` function tells the expression parser to attempt to simplify the expression as much as possible.

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

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

4.2.2 Edge models

Edge models may be specified in terms of other edge models, mathematical functions, and parameters on the device. In addition, edge models may reference node models defined on the ends of the edge. As depicted in Fig. 4.2, edge models are with respect to the two nodes on the edge, `n0` and `n1`.

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

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

In this example, `EdgeInverseLength` is a built-in model for the inverse length between nodes on an edge. `Potential@n0` and `Potential@n1` is the `Potential` node solution on the nodes at the end of the edge. These edge quantities are created using the *`devsim.edge_from_node_model()`* (page 100). 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 Table 4.2.

Table 4.2: Edge models defined on each region of a device.

Edge Model	Description
`EdgeCouple`	The length of the perpendicular bisector of an element edge. Used to perform surface integration of edge models on edges in mesh.
`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)

4.2.3 Element edge models

Element edge models are used when the edge quantitites cannot be specified entirely in terms of the quantities on both nodes of the edge, such as when the carrier mobility is dependent on the normal electric field. In 2D, element edge models are evaluated on each triangle edge. As depicted in Fig. 4.3, edge models are with respect to the three nodes on each triangle edge and are denoted as `en0`, `en1`, and `en2`. Derivatives are with respect to each node on the triangle.

In 3D, element edge models are evaluated on each tetrahedron edge. Derivatives are with respect to the nodes on both triangles on the tetrahedron edge. Element edge models automatically created for a region are listed in Table 4.3.

As an alternative to treating integrating the element edge model with respect to `ElementEdgeCouple`, the integration may be performed with respect to `ElementNodeVolume`. See *devsim.equation()* (page 84) for more information.

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

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

4.2.4 Model derivatives

To converge upon the solution, derivatives are required with respect to each of the solution variables in the system. `DEVSIM` will look for the required derivatives. For a model `model`, the derivatives with respect to solution variable `variable` are presented in Table 4.4.

Table 4.4: Required derivatives for equation assembly. `model` is the name of the model being evaluated, and `variable` is one of the solution variables being solved at each node.

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

4.2.5 Conversions between model types

The *devsim.edge_from_node_model()* (page 100) 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.

4.2.6 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 4.5: Required derivatives for interface equation assembly. The node model name `nodemodel` and its derivatives `nodemodel:variable` are suffixed with `@r0` and `@r1` to denote which region on the interface is being referred to.

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

4.3 Interface

4.3.1 Interface models

Fig. 4.4 depicts an interface in `DEVSIM`. It is a collection of overlapping nodes existing in two regions, `r0` and `r1`.

Interface models are node models specific to the interface being considered. They are unique from bulk node models, in the sense that they may refer to node models on both sides of the interface. They are specified using the *devsim.interface_model()* (page 107). Interface models may refer to node models or parameters on either side of the interface using the syntax `nodemodel@r0` and `nodemodel@r1` to refer to the node model in the first and second regions of the interface. The naming convention for node models, interface node models, and their derivatives are shown in Table 4.5.

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

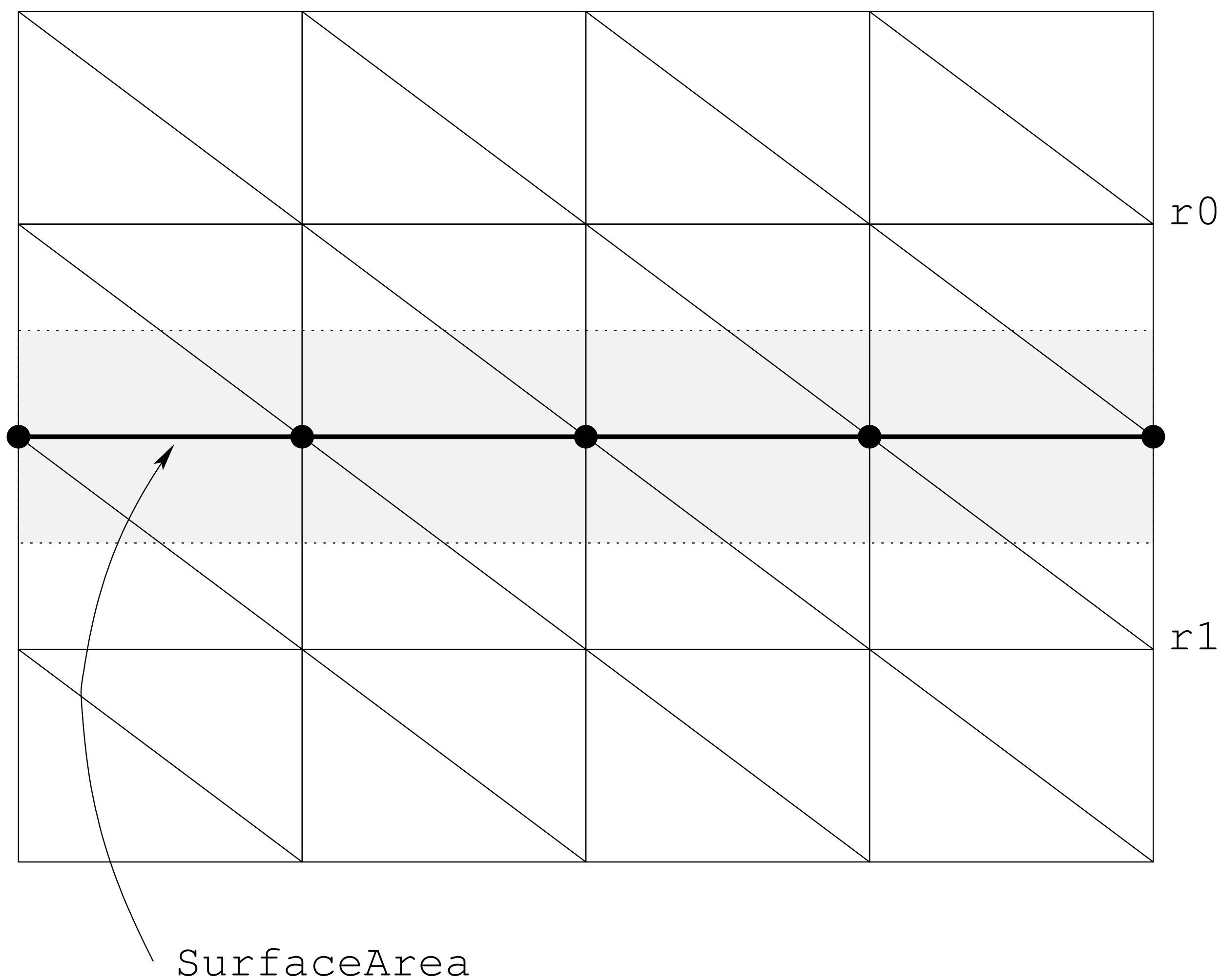

Fig. 4.4: Interface constructs in 2D. Interface node pairs are located at each •. The `SurfaceArea` model is used to integrate flux term models.

4.3.2 Interface model derivatives

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

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

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

4.3.3 Interface equation assembly

There are three types of interface equations considered in DEVSIM. They are both activated using the *devsim.interface_equation()* (page 86).

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.

4.4 Contact

4.4.1 Contact models

Fig. 4.5 depicts how a contact is treated in a simulation. It is a collection of nodes on a region. During assembly, the specified models form an equation, which replaces the equation applied to these nodes for a bulk node.

Contact models are equivalent to node and edge models, and are specified using the *devsim.contact_node_model()* (page 97) 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.

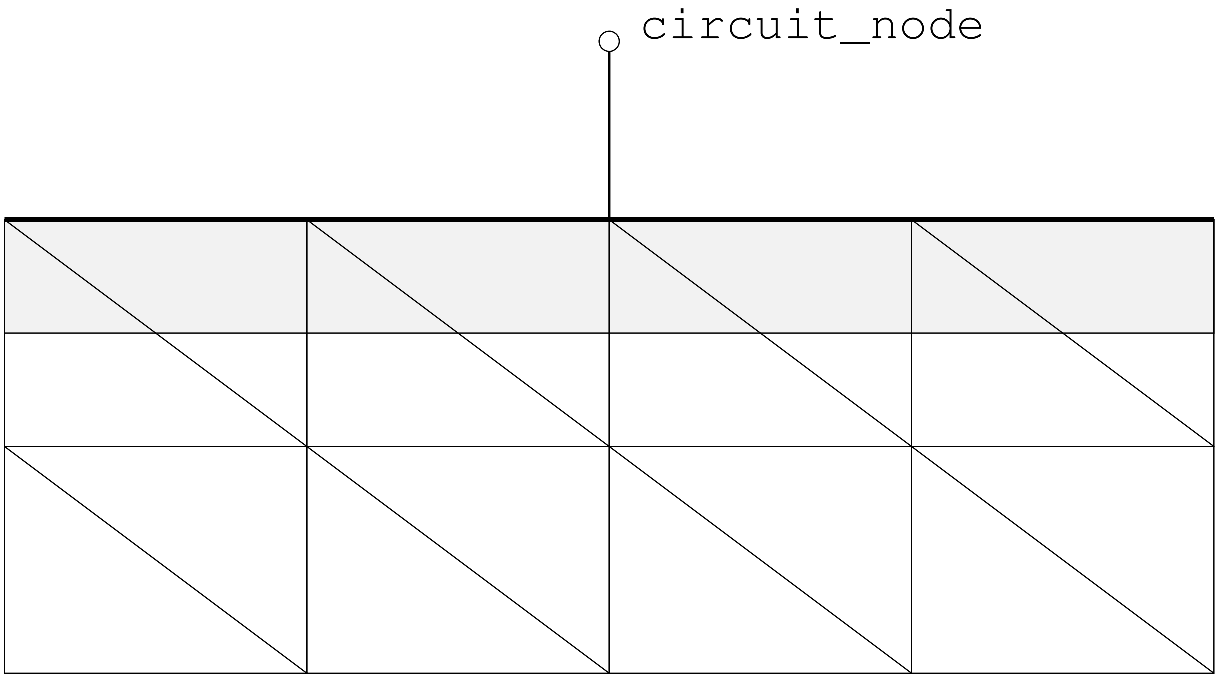

Fig. 4.5: Contact constructs in 2D.

4.4.2 Contact model derivatives

The derivatives are equivalent to the discussion in *Model derivatives* (page 43). If external circuit boundary conditions are being used, the model `model` derivative with respect to the circuit node `node` name should be specified as `model:node`.

4.4.3 Contact equation assembly

The *`devsim.contact_equation()`* (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.

4.5 Custom Matrix assembly

The *devsim.custom_equation()* (page 83) 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.

4.6 Cylindrical Coordinate Systems

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

- *devsim.cylindrical_edge_couple()* (page 97)
- *devsim.cylindrical_node_volume()* (page 97)
- *devsim.cylindrical_surface_area()* (page 98)

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

```
set_parameter(name="node_volume_model",
  value="CylindricalNodeVolume")
set_parameter(name="edge_couple_model",
  value="CylindricalEdgeCouple")
set_parameter(name="edge_node0_volume_model",
  value="CylindricalEdgeNodeVolume@n0")
set_parameter(name="edge_node1_volume_model",
  value="CylindricalEdgeNodeVolume@n1")
set_parameter(name="element_edge_couple_model",
  value="ElementCylindricalEdgeCouple")
set_parameter(name="element_node0_volume_model",
  value="ElementCylindricalNodeVolume@en0")
set_parameter(name="element_node1_volume_model",
  value="ElementCylindricalNodeVolume@en1")
```

Chapter 5

Parameters

Parameters can be set using the commands in *Material Commands* (page 88). There are two complementary formalisms for doing this.

5.1 Parameters

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

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

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

Table 5.1: Parameters controlling program behavior.

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

5.2 Material database entries

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

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

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

5.3 Discussion

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

Chapter 6

Circuits

6.1 Circuit elements

Circuit elements are manipulated using the commands in *Circuit Commands* (page 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.

6.2 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.

Chapter 7

Meshing

7.1 1D mesher

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

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

The *devsim.create_1d_mesh()* (page 94) 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 91) 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 91), *devsim.add_1d_interface()* (page 91) and *devsim.add_1d_region()* (page 91) are used to specify the contacts, interfaces and regions for the device.

Once the meshing commands have been completed, the *devsim.finalize_mesh()* (page 96) is called to create a mesh structure and then *devsim.create_device()* (page 94) is used to create a device using the mesh.

7.2 2D mesher

Similar to the 1D mesher, the 2D mesher uses a sequence of non-terminating mesh lines are specified in both the x and y directions to specify a mesh structure. As opposed to using tags, the regions are specified using *devsim.add_2d_region()* (page 93) 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 92).

Once the meshing commands have been completed, the *devsim.finalize_mesh()* (page 96) is called to create a mesh structure and then *devsim.create_device()* (page 94) is used to create a device using the mesh.

7.3 Using an external mesher

DEVSIM supports reading meshes from Gmsh. In addition, meshes may be input directly using the Python interface. These meshes may only contain points, lines, triangles, and tetrahedra. Hybrid meshes or uniform meshes containing other elements are not supported at this time.

7.3.1 Gmsh

The Gmsh meshing software (see *Gmsh* (page 77)) can be used to create a 1D, 2D, or 3D mesh suitable for use in DEVSIM. When creating the mesh file using the software, use physical group names to map the difference entities in the resulting mesh file to a group name. In this example, a MOS structure is read in:

```
devsim.create_gmsh_mesh(file="gmsh_mos2d.msh", mesh="mos2d")
devsim.add_gmsh_region(mesh="mos2d" gmsh_name="bulk", region="bulk",
  material="Silicon")
devsim.add_gmsh_region(mesh="mos2d" gmsh_name="oxide", region="oxide",
  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 96) is called to create a mesh structure and then *devsim.create_device()* (page 94) is used to create a device using the mesh.

7.3.2 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.

7.4 Loading and saving results

The *devsim.write_devices()* (page 96) 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 96) is then used to reload the device data for restarting the simulation.

Chapter 8

Solver

8.1 Solver

DEVSIM uses Newton methods to solve the system of PDE's. All of the analyses are performed using the *devsim.solve()* (page 112).

8.2 DC analysis

A DC analysis is performed using the *devsim.solve()* (page 112).

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

8.3 AC analysis

An AC analysis is performed using the *devsim.solve()* (page 112). A circuit voltage source is required to set the AC source.

8.4 Noise/Sensitivity analysis

An noise analysis is performed using the *devsim.solve()* (page 112) 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 112). If the equation begin solved is PotentialEquation, the names of the scalar impedance field is then:

- V1.I_PotentialEquation_real

- `V1.I_PotentialEquation_imag`

and the vector impedance fields evaluated on the nodes are

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

8.5 Transient analysis

Transient analysis is performed using the `devsim.solve()` (page 112). `DEVSIM` supports time-integration of the device PDE's. The three methods are supported are:

- BDF1
- TRBDF
- BDF2

Chapter 9

User Interface

9.1 Starting DEVSIM

Refer to *Installation* (page 75) for instructions on how to install DEVSIM. Once installed, DEVSIM may be invoked using the following command

It is necessary to first PYTHONPATH variable to the lib directory in the DEVSIM distribution. As an alternative, an experimental installation script is available to make the process easier. Please see *Installation Script* (page 12) for more information.

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
```

The supported versions of Python for use in scripts is 3.6 or higher.

9.2 Python Language

9.2.1 Introduction

Python is the scripting language employed as the text interface to DEVSIM. Documentation and tutorials for the language are available from [1]. A paper discussing the general benefits of using scripting languages may be found in [6].

9.2.2 DEVSIM commands

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

```
from devsim import *
```

For details concerning error handling, please see *Error handling* (page 62).

9.2.3 Advanced usage

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

9.2.4 Unicode Support

Internally, `DEVSIM` uses UTF-8 encoding, and expects model equations and saved mesh files to be written using this encoding. Users are encouraged to use the standard ASCII character set if they do not wish to use this feature. Python 3 interpreters handle UTF-8 encoding well.

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

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

Care should be taken when using UTF-8 characters in names for visualization using the tools in *Visualization* (page 73), as this character set may not be supported.

9.3 Error handling

9.3.1 Python errors

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

9.3.2 Fatal errors

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

In rare situations, the program may behave in an erratic manner, print a message, such as `UNEXPECTED` or terminate abruptly. Please report this using the contact information in *Contact* (page 1).

9.3.3 Floating point exceptions

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

9.3.4 Solver errors

When using the *devsim.solve()* (page 112), the solver may not converge and a message will be printed and an exception may be thrown. The solution will be restored to its previous value before the simulation began. This exception may be caught and the bias conditions may be changed so the simulation may be continued. For example:

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

9.3.5 Verbosity

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

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

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

9.3.6 Parallelization

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

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

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

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

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

Chapter 10

SYMDIFF

10.1 Overview

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

10.2 Syntax

10.2.1 Variables and numbers

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

Examples of valid variable names are:

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

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

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

10.2.2 Basic expressions

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

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

In Table 10.1, the basic syntax for the language is presented. An expression may be composed of variables and numbers tied together with mathematical operations. Order of operations is from bottom to top in order of increasing precedence. Operators with the same level of precedence are contained within horizontal lines.

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

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

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

10.2.3 Functions

Table 10.2: Predefined Functions

Function	Description
`acosh(exp1)`	Inverse Hyperbolic Cosine
`asinh(exp1)`	Inverse Hyperbolic Sine
`atanh(exp1)`	Inverse Hyperbolic Tangent
`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 Table 10.2 are the built in functions of `SYMDIFF`. Note that the `pow` function uses the `,` operator to separate arguments. In addition an expression like `pow(a,b+y)` is equivalent to an expression like `a^(b+y)`. Both `exp` and `log` are provided since many derivative expressions can be expressed in terms of these two functions. It is possible to nest expressions within functions and vice-versa. Table 10.3 lists the error functions, derivatives, and inverses. Table 10.4 lists the Fermi functions, and are based on the Joyce-Dixon Approximation [4]. The Gauss-Fermi functions are listed in Table 10.5, based on [7].

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 Table 10.6. 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 Table 10.7. 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

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

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 110) 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

11.1 Introduction

DEVSIM is able to create files for visualization tools. Information about acquiring these tools are presented in *External Software Tools* (page 77).

11.2 Using Tecplot

The *devsim.write_devices()* (page 96) is used to create an ASCII file suitable for use in Tecplot. Edge quantities are interpolated onto the node positions in the resulting structure. Element edge quantities are interpolated onto the centers of each triangle or tetrahedron in the mesh.

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

11.3 Using Postmini

The *devsim.write_devices()* (page 96) is used to create an ASCII file suitable for use in Postmini. Edge and element edge quantities are interpolated onto the node positions in the resulting structure.

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

11.4 Using Paraview

The *devsim.write_devices()* (page 96) is used to create an ASCII file suitable for use in ParaView. Edge quantities are interpolated onto the node positions in the resulting structure. Element edge quantities are interpolated onto the centers of each triangle or tetrahedron in the mesh.

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

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

11.5 Using VisIt

VisIt supports reading the Tecplot and ParaView formats. When using the vtk option on the *devsim.write_devices()* (page 96), a file with a visit filename extension is created to load the files created for ParaView.

11.6 DEVSIM

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

See *Loading and saving results* (page 58) for information about loading and saving mesh information to a file.

Chapter 12

Installation

12.1 Availability

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

12.2 Supported platforms

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

Table 12.1: Current platforms for DEVSIM.

Platform	Bits	OS Version
Microsoft Windows	64	Microsoft Windows 10
Linux	64	Red Hat Enterprise Linux 7 (Centos 7 compatible)
Apple macOS	64	macOS 10.13 (High Sierra)

12.3 Binary availability

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

12.4 Source code availability

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

12.5 Directory Structure

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

Table 12.2: Directory structure for DEVSIM.

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

12.6 Running DEVSIM

See *User Interface* (page 61) for instructions on how to invoke DEVSIM.

Chapter 13

Additional Information

13.1 DEVSIM License

Individual files are covered by the license terms contained in the comments at the top of the file. Contributions to this project are subject to the license terms of their authors. In general, DEVSIM is covered by the Apache License, Version 2.0 [11]. Please see the NOTICE and LICENSE file for more information.

13.2 SYMDIFF

SYMDIFF is available from https://symdiff.org under the terms of the Apache License, Version 2.0 [11].

13.3 External Software Tools

13.3.1 Gmsh

Gmsh [3] is available from http://gmsh.info.

13.3.2 Paraview

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

13.3.3 Tecplot

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

13.3.4 VisIt

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

13.4 Library Availablilty

The following tools are used to build DEVSIM.

13.4.1 BLAS and LAPACK

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

13.4.2 Python

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

13.4.3 SQlite3

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

13.4.4 SuperLU

SuperLU [2] is used within DEVSIM and and is available from http://crd-legacy.lbl.gov/~xiaoye/SuperLU:

> Copyright (c) 2003, The Regents of the University of California, through Lawrence Berkeley National Laboratory (subject to receipt of any required approvals from U.S. Dept. of Energy)
>
> All rights reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or

other materials provided with the distribution. (3) Neither the name of Lawrence Berkeley National Laboratory, U.S. Dept. of Energy nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

13.4.5 zlib

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

Chapter 14

Command Reference

14.1 Circuit Commands

Commands are for adding circuit elements to the simulation.

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.**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 48) 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 49).

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

14.4 Material Commands

Commands for manipulating parameters and material properties

`devsim.add_db_entry(material, parameter, value, unit, description)`

Adds an entry to the database

Parameters

- **material** (*str*) – Material name requested. `global` refers to all regions whose material does not have the parameter name specified
- **parameter** (*str*) – Parameter name
- **value** (*str*) – Value assigned for the parameter
- **unit** (*str*) – String describing the units for this parameter name
- **description** (*str*) – Description of the parameter for this material type.

Notes

The *devsim.save_db()* (page 90) command is used to commit these added entries permanently to the database.

`devsim.close_db()`

Closes the database so that its entries are no longer available

`devsim.create_db(filename)`

Create a database to store material properties

Parameters

filename (*str*) – filename to create for the db

`devsim.get_db_entry(material, parameter)`

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

Parameters

- **material** (*str*) – Material name
- **parameter** (*str*) – Parameter name

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.open_db`(*filename*, *permissions*)

Open a database storing material properties

Parameters

- **filename** (*str*) – filename to create for the db
- **permissions** (*{'readonly', 'readwrite'}*) – permissions on the db

`devsim.save_db`()

Saves any new or modified db entries to the database file

`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.**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*)

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', 'floops', 'tecplot', 'vtk'}*) – format to use

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 90) 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 90) 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 90) 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* (page 73) 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*)

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)

Chapter 15

Example Overview

In the following chapters, examples are presented for the use of `DEVSIM` to solve some simulation problems. Examples are also located in the `DEVSIM` distribution and their location is mentioned in *Directory structure for DEVSIM.* (page 76).

Additional examples are available online, and listed here:

- https://devsim.org/introduction.html#examples

The following example directories are contained in the distribution.

15.1 capacitance

These are 1D and 2D capacitor simulations, using the internal mesher. A description of these examples is presented in *Capacitor* (page 117).

15.2 diode

This is a collection of 1D, 2D, and 3D diode structures using the internal mesher, as well as `Gmsh`. These examples are discussed in *Diode* (page 129).

15.3 bioapp1

This is a biosensor application.

15.4 vector_potential

This is a 2D magnetic field simulation solving for the magnetic potential. The simulation script is `vector_potential/twowire.py` A simulation result for two wires conducting current is shown in Fig. 15.1.

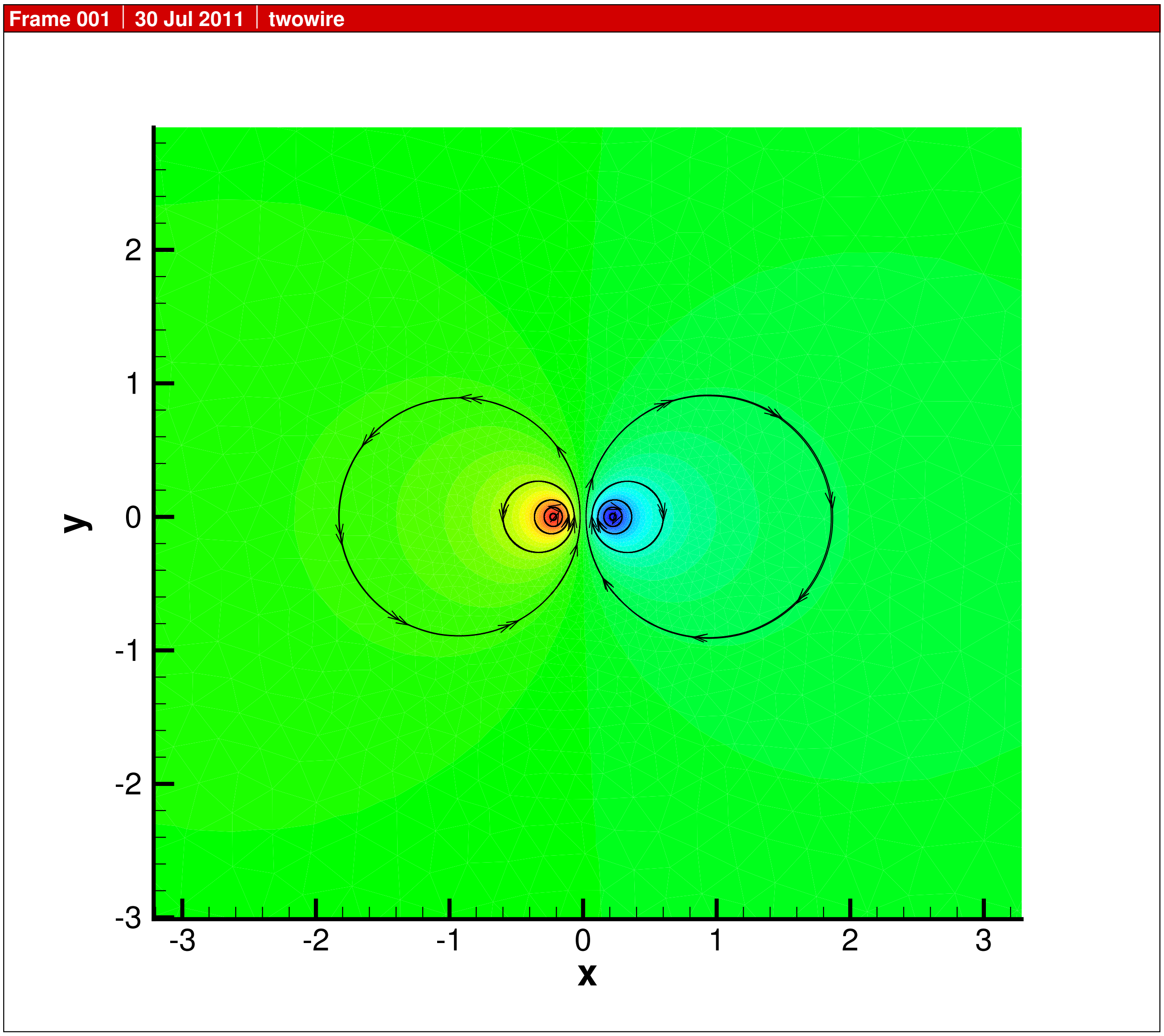

Fig. 15.1: Simulation result for solving for the magnetic potential and field. The coloring is by the Z component of the magnetic potential, and the stream traces are for components of magnetic field.

15.5 mobility

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

Chapter 16

Capacitor

16.1 Overview

In this chapter, we present a capacitance simulations. The purpose is to demonstrate the use of `DEVSIM` with a rather simple example. The first example in *1D Capacitor* (page 117) 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 121).

16.2 1D Capacitor

16.2.1 Equations

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

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

and the contact boundary equations are

$$\psi_i - V_c = 0$$

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

16.2.2 Creating the mesh

The following statements create a one-dimensional mesh which is 1 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)
```

16.3 Setting device parameters

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

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

16.3.1 Creating the models

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

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

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

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

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

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

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

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

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

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

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

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

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

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

16.3.2 Contact boundary conditions

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

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

  contact_node_model(device=device, contact=c, name="%s_bc:Potential" % c,
             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.

16.3.3 Setting the boundary conditions

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

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

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

16.3.4 Running the simulation

We run the simulation and see the results.

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


 DEVSIM

 Version: Beta 0.01

 Copyright 2009-2013 Devsim LLC

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



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 $3.451510^{-13}\ \mathrm{F/cm^2}$ for a homogenous capacitor.

16.4 2D Capacitor

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

16.5 Defining the mesh

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

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

ymin=0.0
y1  =0.1
y2  =0.2
y3  =0.8
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
```

(continues on next page)

(continued from previous page)

```
add_2d_contact(mesh=device, name="bot", region="air", material="metal", yl=y1,␣
↪yh=y2, xl=x1, xh=x4)
add_2d_contact(mesh=device, name="top", region="air", material="metal", yl=y3,␣
↪yh=y4, xl=x2, xh=x3)
finalize_mesh(mesh=device)
create_device(mesh=device, device=device)
```

16.6 Setting up the models

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

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

###
### 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")
```

(continues on next page)

(continued from previous page)

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

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

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


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

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

  contact_equation(device=device, contact=c, name="PotentialEquation",
             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")
```

16.7 Fields for visualization

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

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

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

16.8 Running the simulation

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

 DEVSIM

 Version: Beta 0.01

 Copyright 2009-2013 Devsim LLC

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


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.
```

(continues on next page)

(continued from previous page)

```
Device "MyDevice" has equations 0:8280
number of equations 8281
Iteration: 0
  Device: "MyDevice"  RelError: 1.00000e+00   AbsError: 1.00000e+00
    Region: "air"     RelError: 1.00000e+00   AbsError: 1.00000e+00
      Equation: "PotentialEquation"   RelError: 1.00000e+00   AbsError: 1.
↪00000e+00
Iteration: 1
  Device: "MyDevice"  RelError: 1.25144e-12   AbsError: 1.73395e-13
    Region: "air"     RelError: 1.25144e-12   AbsError: 1.73395e-13
      Equation: "PotentialEquation"   RelError: 1.25144e-12   AbsError: 1.73395e-
↪13
3.35017166004e-12
-3.35017166004e-12
```

A visualization of the results is shown in Fig. 16.1.

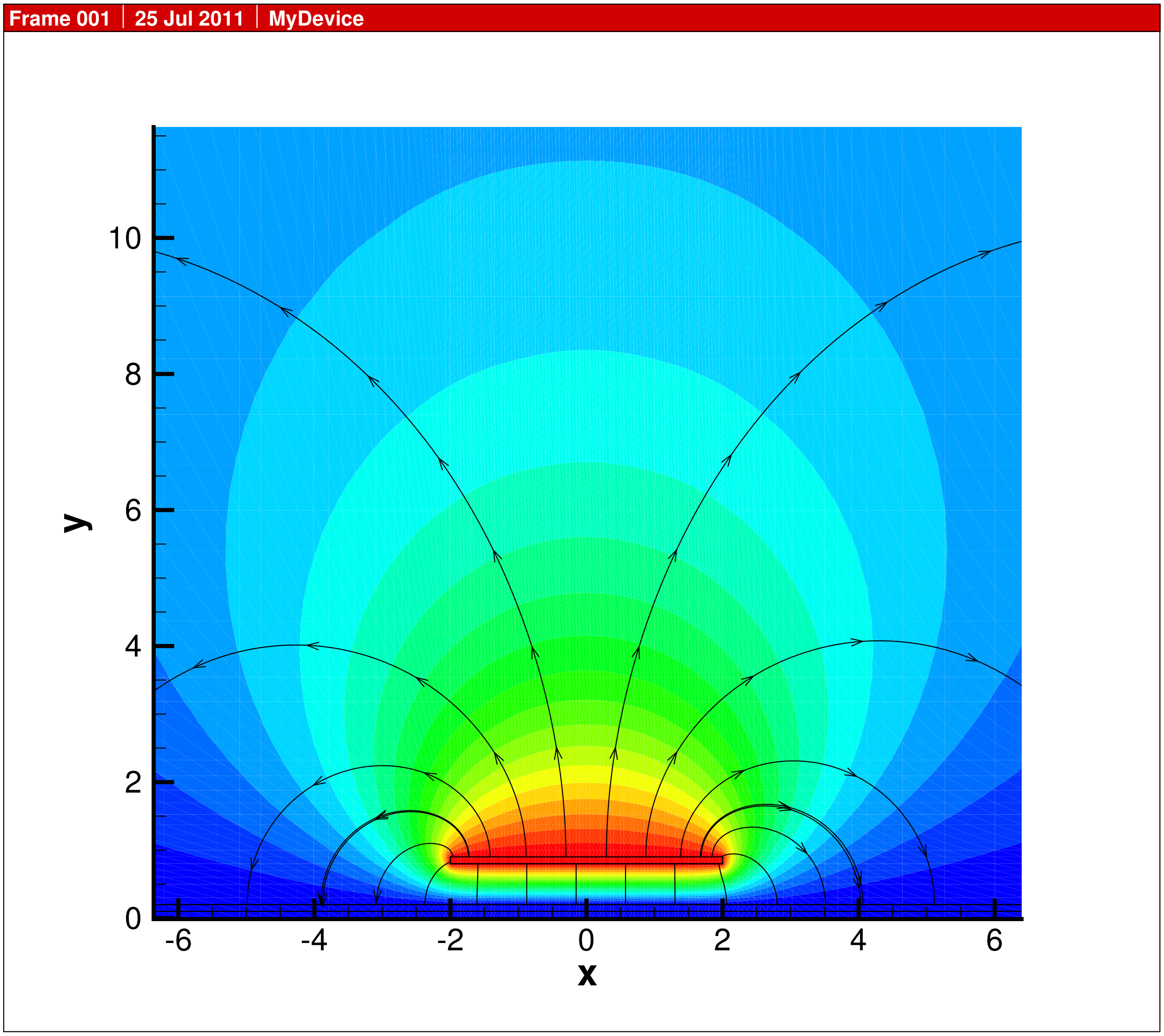

Fig. 16.1: Capacitance simulation result. The coloring is by `Potential`, and the stream traces are for components of `ElectricField`.

Chapter 17

Diode

The diode examples are located in the `examples/diode`. They demonstrate the use of packages located in the `python_packages` directory to simulate drift-diffusion using the Scharfetter-Gummel method [10].

17.1 1D diode

17.1.1 Using the python packages

For these examples, python modules are provided to supply the appropriate model and parameter settings. A listing is shown in Table 17.1. The `devsim.python_packages` module is part of the distribution. The example files in the `DEVSIM` distribution set the path properly when loading modules.

Table 17.1: Python package files.

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

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

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

17.1.2 Creating the mesh

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

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

device="MyDevice"
region="MyRegion"

createMesh(device, region)
```

17.2 Physical Models and Parameters

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

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

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

####
```

(continues on next page)

(continued from previous page)

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

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


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

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

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

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

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

####
#### Ramp the bias to 0.5 Volts
```

(continues on next page)

(continued from previous page)

```
####
v = 0.0
while v < 0.51:
  set_parameter(device=device, name=GetContactBiasName("top"), value=v)
  solve(type="dc", absolute_error=1e10, relative_error=1e-10, maximum_
↪iterations=30)
  PrintCurrents(device, "top")
  PrintCurrents(device, "bot")
  v += 0.1

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

17.2.1 Plotting the result

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

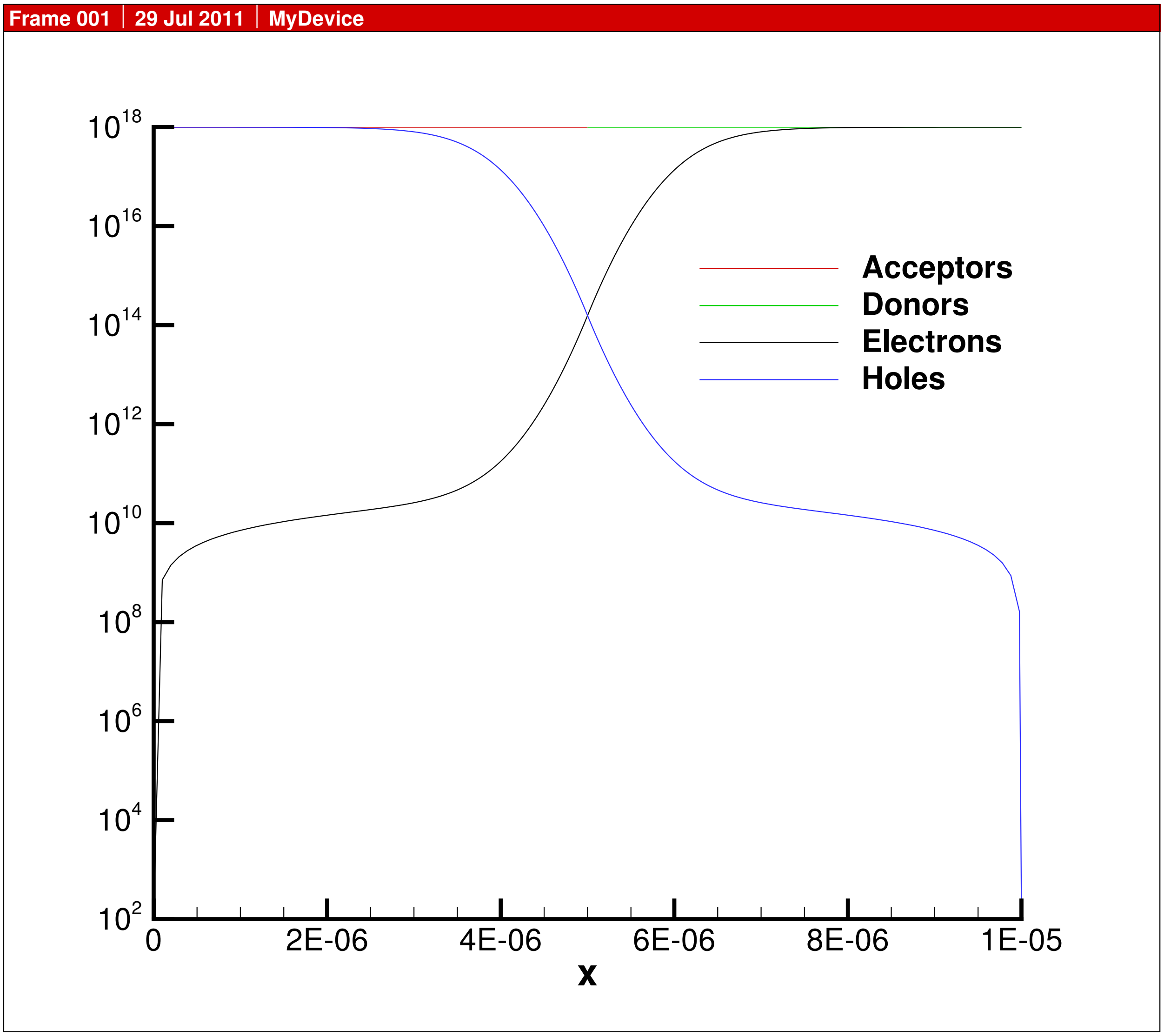

Fig. 17.1: Carrier density versus position in 1D diode.

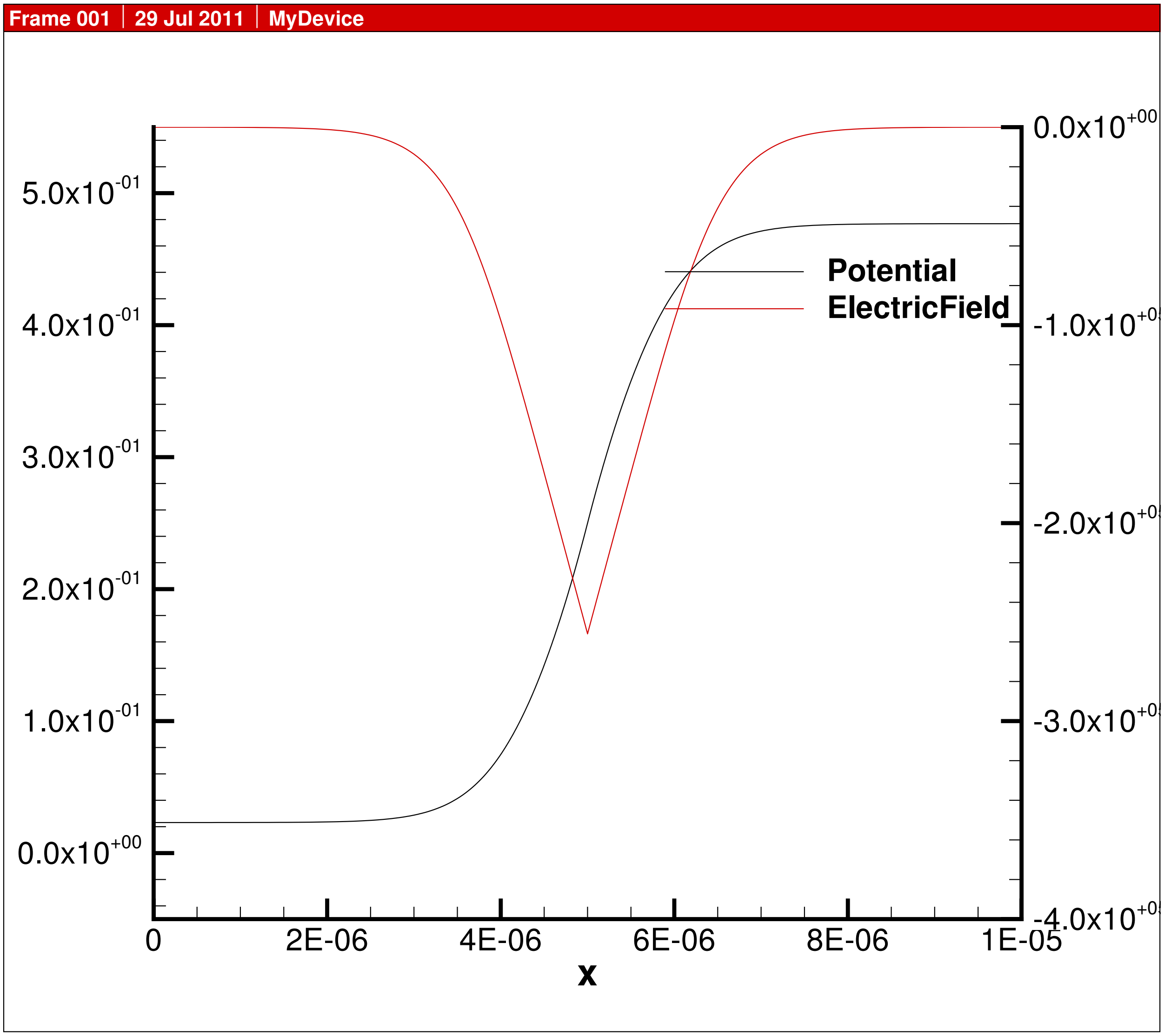

Fig. 17.2: Potential and electric field versus position in 1D diode.

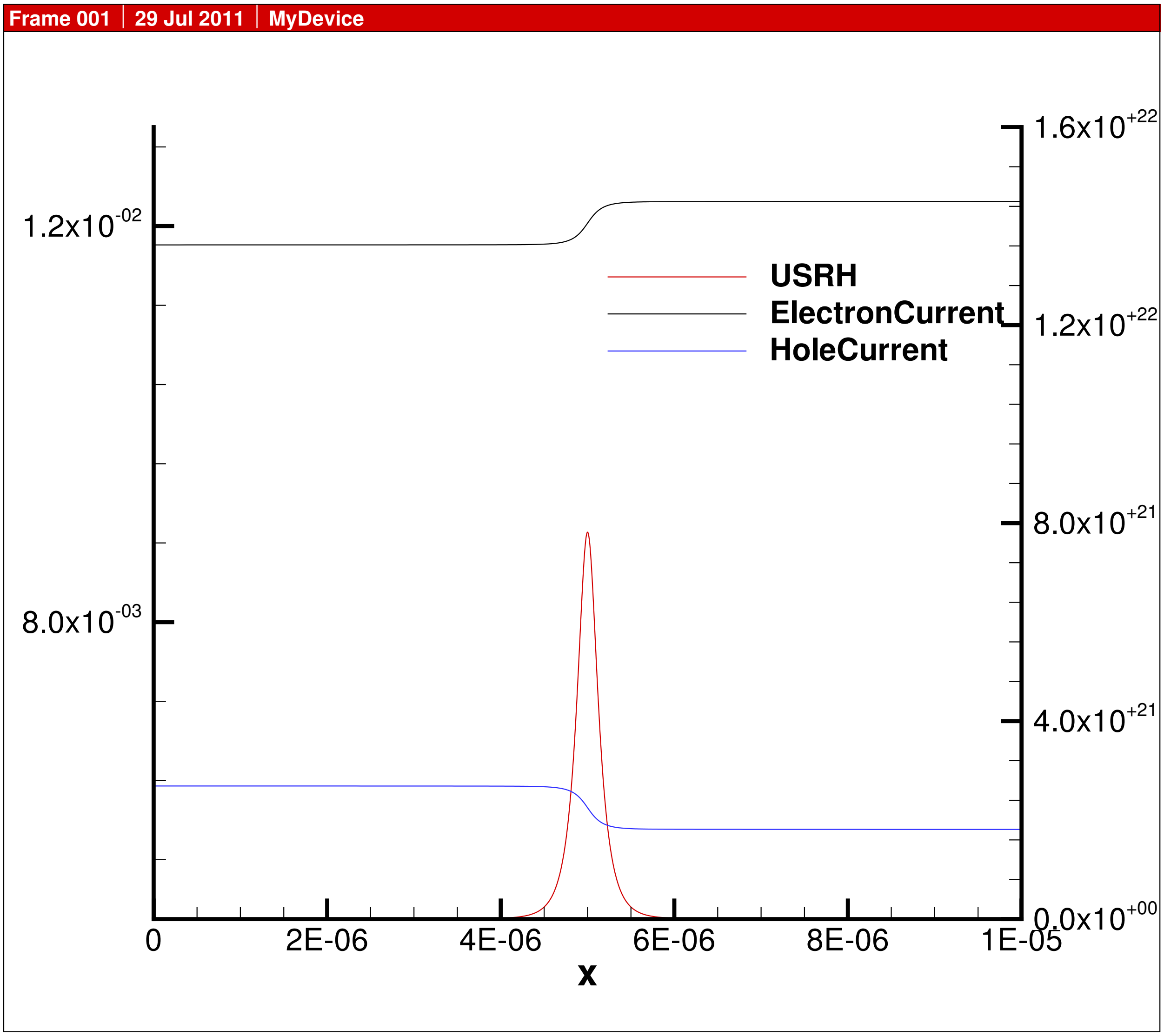

Fig. 17.3: Electron and hole current and recombination.

Bibliography

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

[2] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999. doi:10.1137/S0895479895291765 (https://doi.org/10.1137/S0895479895291765).

[3] 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).

[4] 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).

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

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] 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).

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

Index

I

L

M

N

O

P

R

S

V

W