Semiconductor Device Simulation Using DEVSIM

Juan E. Sanchez

Abstract DEVSIM is a technology computer aided design (TCAD) simulation software. It is released under an open source license. The software solves user defined partial differential equations (PDEs) on 1D, 2D, and 3D meshes. It is implemented in C++ using custom code and a collection of open source libraries. The Python scripting interface enables users to setup and control their simulations.

In this chapter, we present an overview of the tool. This is followed with a bipolar junction transistor (BJT) design and characterization example. A collection of open source tools were used to create a simulation mesh, and visualize results

Key words: BJT, DEVSIM, Open Source, Python, Semiconductor, Simulation, TCAD

1 Introduction

DEVSIM is a simulation software for technology computer aided design (TCAD) [13]. The software uses a generalized partial differential equation (PDE) approach to solving the semiconductor device equations [2, 1].

DEVSIM and the examples in this chapter are available from `http://www.devsim.org`. The source code is available under the terms of the Apache License, Version 2.0 [16]. The examples for this chapter are also released with this license. Contributions to this project are welcome in the form of bug reporting, documentation, modeling, and feature implementation.

Juan E. Sanchez
DEVSIM LLC, PO Box 50096, Austin, TX 78763 e-mail: `juan.sanchez@devsim.com`

From the Python [3] scripting interface, users input symbolic expressions for the device physics equations being solved. The SYMDIFF [4] symbolic differentiation software parses expressions and creates derivatives for the equations. The advantages of this system are:

- Model equations and derivatives are created symbolically and may be examined and optimized.
- New models are implemented without requiring advanced knowledge of C++.
- The scripting interface provides a wide array of open source and proprietary Python modules for performing analyses.

A key advantage of open source software is that the program code, data, and scripts to generate results can be disseminated in their entirety. Other members of the TCAD community are then able to replicate results, and extend the research.

DEVSIM leverages open source libraries for numerics, data structures, and other infrastructure [13]. The availability of these libraries under open source license has significantly decreased the development time and production costs of this software.

DEVSIM generates open data formats which can then be read into other software for analysis, visualization, and publication of simulation results. Open source tools used to generate, visualize, and document the simulation results for this chapter are listed in Table 1.

Table 1: Open source tools used in this chapter.

Name	Description	Website	License[a]
Gmsh	Mesh Generator	`http://geuz.org/gmsh`	GPL
LaTeX	Document Preparation System	`http://latex-project.org`	LPPL
matplotlib	Python 2D Plotting Library	`http://matplotlib.org`	matplotlib
NumPy	Python Scientific Computing	`http://numpy.org`	BSD
Python	Scripting Language	`http://python.org`	PSF
VisIt	Visualization Tool	`http://visit.llnl.gov`	BSD
Xfig	Interactive Figure Generation	`http://www.xfig.org`	MIT

[a] License information is available from `http://www.opensource.org`. Visit the official websites for specific license text.

In Sect. 2, an overview of DEVSIM is presented. In Sect. 3, semiconductor physics for bipolar junction transistor (BJT) simulation is presented. Sect. 4 presents the creation of a BJT structure and its meshing refinement. Simulation results are also presented. Sect. 5 provides the conclusion and discussion of future work.

In the following sections, several script examples for DEVSIM are presented. A basic knowledge of the Python scripting language is required to understand them. Documentation and tutorials for this language may be found in [3]. DEVSIM commands are documented in [13]. Additional commands are helper functions introduced through Python modules written specifically for the examples in this chapter.

2 DEVSIM Overview

2.1 Models

DEVSIM uses the finite volume method for assembling the PDEs on the simulation mesh [24]. it solves equations of the form:

$$\frac{\partial X}{\partial t} + \nabla \cdot \mathbf{Y} + Z = 0 \tag{1}$$

Internally, it transforms the PDEs into an integral form.

$$\int \frac{\partial X}{\partial t} \partial r + \int \mathbf{Y} \cdot \partial \mathbf{s} + \int Z \partial r = 0 \tag{2}$$

where equations involving the divergence operator are converted into surface integrals, scalar components are integrated over the device volume.

In Fig. 1, 2D mesh elements are depicted. Models integrated over the volume of each triangle vertex are referred to as node models. The shaded area around the center node is referred to as the node volume, and it is used for volume integration. The lines from the center node to other nodes are referred to as edges. Models integrated over the edges of triangles are referred to as edge models. The flux through the edge are integrated with respect to the perpendicular bisectors (dashed lines) crossing each triangle edge. Element edge models are like edge models, but account for variables at nodes off of the edge.

There are a default set of models created in each region upon initialization of a device, and are typically based on the geometrical attributes of the simulation mesh. Models required for describing the device behavior are created using the SYMDIFF [13, 14] expression parser.

2.1.1 Node models

Node models may be specified in terms of other node models, and parameters on the device. The simplest model is the node solution, and it represents the solution variables being solved for. In addition to the built-in models, models are defined using symbolic expressions. Fig. 2 shows an implementation of the Shockley Read Hall recombination [21] model.

The first model specified, `USRH`, is the recombination model, and it is created with the `CreateNodeModel` command. The derivatives with respect to electrons and holes are `USRH:Electrons` and `USRH:Holes`, respectively, and are created with the `CreateNodeModelDerivative` command. In this particular example `NIE` is the intrinsic carrier density, and `Electrons` and `Holes` have already been declared as solution variables. The remaining variables in the equation are parameters which may be specified on the region or the whole device.

2.1.2 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 calculated on the ends of the edge. As depicted in Fig. 3, edge models are with respect to the two nodes on the edge, `n0` and `n1`.

To calculate the electric field and the displacement field on the mesh edges, the implementation in Fig. 4 may be used. The `edge_average_model` command for creates gradient models along a mesh edge. `Potential@n0` and `Potential@n1` are the `Potential` node values on the nodes on the ends of the edge. We assume the convention that the flux flows from `n0` to `n1`.

2.1.3 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. In 3D, element edge models are evaluated on each tetrahedron edge. As depicted in Fig. 6, 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. There is a value per triangle edge, which are averaged onto the edge using the weighting scheme in [19].

The `element_from_edge_model` is used to to average an edge model onto an element edge model. This is used to calculate vector components and the magnitude of the electric field on the elements and is demonstrated in Fig. 5.

These fields will be used for the mesh refinement steps in the BJT example presented in a later section. Element edge models are also useful for visualization of fields on the mesh and element edge quantities are averaged onto the center of the element when written to the Tecplot [5] or VTK [6] formats.

2.1.4 Model derivatives

To converge upon a solution, derivatives are required with respect to each of the solution variables on the regions of the device. Accurate derivatives are also required to perform a small-signal AC analysis. DEVSIM will use the derivatives when they are provided. For a model `model`, the derivatives with respect to solution variable `variable` are presented in Table 2. To improve efficiency of model evaluation, the software caches common subexpressions across all models. To prevent calculation errors, it detects models which refer to themselves through their dependencies. In addition, it will also report floating point errors if they occur when evaluating the model.

Table 2: 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)

2.1.5 Contact and Interface Models

Similar to the bulk simulation case, contact and interface boundary conditions take advantage of the symbolic expression parser. More detail is available in the manual [13] and in examples in the next section.

2.2 Solvers

DEVSIM has the following simulation capabilities:

- DC
- Small-signal AC
- Impedance field method (Noise Simulation)
- Transient

The tool currently uses Newton's method [24] for solving DC and transient. SuperLU [12] is used for direct factorization and solution of the simulation matrices. The `equation` command is used to specify the bulk equations, and the simulation variable associated with it. In addition, boundary conditions are specified using the `contact_equation` and `interface_equation` commands for boundary conditions at contacts and interfaces between materials, respectively.

3 BJT Physics

In this section, the semiconductor equations for BJT simulation are discussed. The device physical equations are set up using the Python scripts. For the purposes of this chapter, the following assumptions are made.

- Boltzmann statistics

- Silicon structure without band gap narrowing
- Drift-diffusion with constant temperature
- Doping dependent mobility with velocity saturation
- Ohmic contacts

3.1 Potential Only

The potential only simulation is used to identify regions for meshing refinement (Sect. 4.1). It also provides the initial guess for drift-diffusion in Sect. 3.2. Fig. 7 shows the implementation for the Poisson equation. In addition, ohmic boundary conditions are enforced at the contacts as shown in Fig. 8.

3.2 Drift Diffusion

For drift-diffusion simulation, the electron and hole continuity equations are solved in addition to the potential equation. The Scharfetter-Gummel [23] approach is used to model to discretize the electron and hole current density equations. The low-field current density is first calculated once with the Arora mobility model [8].

Velocity saturation is modeled with the Caughey-Thomas model [10] with parameters from [9]. Using the code in Fig. 9, velocity saturation is applied when the electric field and low-field current density are in the same direction. The velocity saturation limited mobility is then used to calculate the current density used in the simulation.

4 BJT Simulation

In this section, we present the meshing and simulation of a bipolar junction transistor (BJT).

4.1 Meshing and Mesh Refinement

DEVSIM solves equations on 1D, 2D, and 3D meshes and has support for generating 1D and 2D meshes. The software supports the import of meshes from both Gmsh [18], and the Genius Device Simulator [11]. For the example in this chapter, Gmsh is used to construct a mesh suitable for 2D drift-diffusion simulation.

The initial mesh was created from a `.geo` file containing the outline of the device and is shown in Fig. 10. Creating a mesh file in Gmsh is often an iterative process between using a text editor and its graphical interface. Points outline a rectangular box outlining the device and contact placement. Physical groups are specified in the file to specify the device and contact locations. This file was input into Gmsh to create an initial mesh.

The doping profile for this example is specified in DEVSIM using the code in Fig. 11. Since we are specifying analytic doping profiles, it is difficult to identify areas for refinement before simulation on a coarse mesh has been performed.

The initial mesh is read into DEVSIM and a potential-only equilibrium simulation is performed. Using the `Emag` model (see Fig. 5) and the `SurfaceArea` model (built-in model) as refinement criteria, a background mesh is created which identifies areas where there are high electric fields and where there are contacts present. The background mesh specifies smaller mesh spacing in critical areas.

The procedure is:

1. Generate coarse mesh in Gmsh.
2. Load mesh into DEVSIM and simulate on analytic doping profile to generate background mesh.
3. Generate refined mesh in Gmsh after merging original mesh specification and the background mesh.
4. Load refined mesh into DEVSIM and simulate on analytic doping profile to generate background mesh.
5. Repeat steps 3–4 until mesh has suitable number of elements.

The goal is to have a mesh with sufficient simulation accuracy without having so many elements that it negatively impacts the computation time. A mesh convergence study [22] should be performed to ensure that these goals are simultaneously met. Fig. 12 shows the simulation mesh before and after refinement based on the magnitude of the electric field. The initial mesh has 2975 nodes and the final mesh has 14773 nodes. It should be noted that the meshing algorithms in Gmsh are not necessarily deterministic and the actual number of nodes in the resulting mesh depend on the algorithms used and many other factors.

4.2 Simulation Results

In this section, DC and small-signal AC simulations are performed on the BJT transistor example. These types of simulations may be used to extract a compact model for the device [17]. The role of TCAD is to give access to compact models early in the device development process [7, 15]. In addition, TCAD simulation of useful for understanding device operation and model verification during the compact model development process [20, 25, 27].

The DC characterisitics were simulated in DEVSIM by ramping the bias on the BJT device. By keeping $V_{bc} = 0.0$ and sweeping V_{be} negative, the data for the Gummel plot in Fig. 13 was simulated.

For the data in Fig. 14, V_{be} was first ramped to 0.7 V. The data for I_c versus V_{ce} was then simulated.

Next, the small-signal behavior of the device was simulated. In DEVSIM, circuit boundary conditions with voltage sources are used to apply a perturbation at the base of the transistor, and the resulting current into the collector is simulated.

Fig. 15 shows $|\beta_{ac}|$ versus f for different values of V_{be} wihile $V_{bc} = 0.0$. The f_T for the transistor was calculated where the small-signal current gain was 1. Fig. 16 shows the resulting curves versus I_c.

5 Conclusion and Future Work

In this chapter, we presented an overview of the DEVSIM TCAD simulator. The physics, structure, and simulation results for a BJT transistor were developed.

DEVSIM is under active development. Like many TCAD simulation programs, there is ongoing work to be done in the development of new physical models. This software offers a platform where much of this work can be done without requiring implementation in a compiled languages, enabling rapid development. Additional work in development of the solvers, and model evaluation will improve the efficiency and types of analyses performed by the simulator.

This software uses a continuum approach, and would be well suited as part of a hierarchical simulation methodology [26]. For TCAD simulation to be physical, it is important that relevant phenomena is represented. For general simulation DEVSIM need to account for effects such as [7]:

- Avalanche effect
- Hydrodynamic Equations
- Self heating
- Surface Mobility

It is anticipated that the appropriate models may implemented primarily using the scripting approach used in Sect. 3, as well as additional C++ source code.

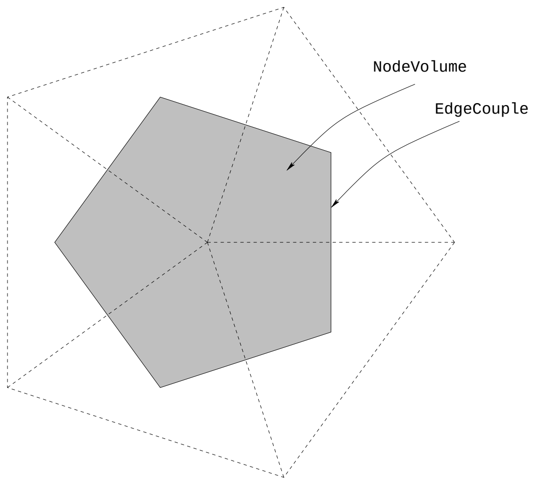


Fig. 1: Mesh elements in 2D.

```
USRH='''(Electrons*Holes - NIE^2)
       /(taup*(Electrons + n1) + taun*(Holes + p1))'''
Gn = "-q * USRH"
Gp = "+q * USRH"
CreateNodeModel(device, region, "USRH", USRH)
CreateNodeModel(device, region, "ElectronGeneration", Gn)
CreateNodeModel(device, region, "HoleGeneration", Gp)
for i in ("Electrons", "Holes", "T"):
  if i in variables:
    CreateNodeModelDerivative(device, region, "USRH", USRH, i)
    CreateNodeModelDerivative(device, region,
      "ElectronGeneration", Gn, i)
    CreateNodeModelDerivative(device, region,
      "HoleGeneration", Gp, i)
```

Fig. 2: Shockley Read Hall recombination model implementation.

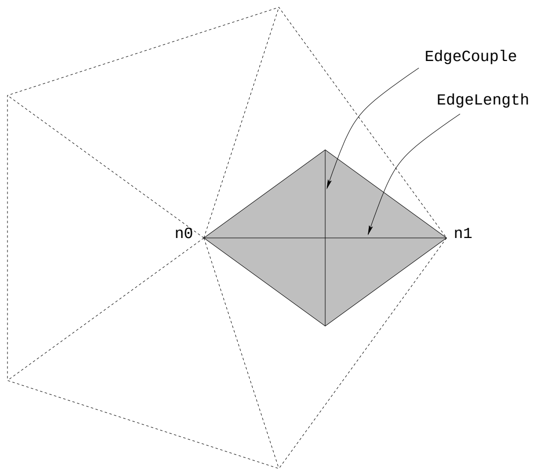


Fig. 3: Edge model constructs in 2D.

```
edge_average_model(device=device, region=region, node_model="Potential",
  edge_model="EField", average_type="negative_gradient")
edge_average_model(device=device, region=region, node_model="Potential",
  edge_model="EField", average_type="negative_gradient",
  derivative="Potential")
CreateEdgeModel(device, region, "DField", "Permittivity * EField")
CreateEdgeModel(device, region,
  "DField:Potential@n0", "Permittivity * EField:Potential@n0")
CreateEdgeModel(device, region,
  "DField:Potential@n1", "Permittivity * EField:Potential@n1")
```

Fig. 4: Edge models for electric and displacement field.

```
element_from_edge_model(edge_model="EField",
  device=device, region=region)
element_model(device=device, region=region,
  name="Emag", equation="(EField_x^2 + EField_y^2)^(0.5)")
```

Fig. 5: Element models for electric field components and magnitude.

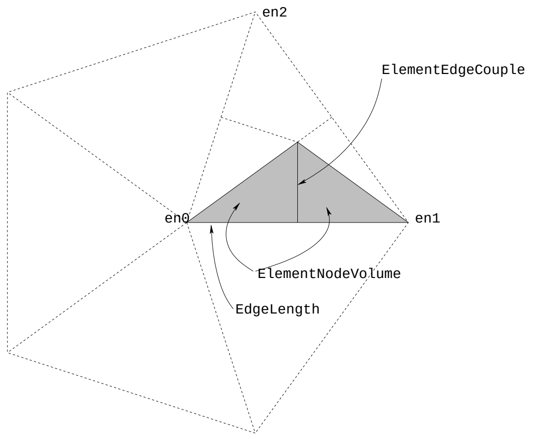


Fig. 6: Element edge model constructs in 2D.

```
def CreateSiliconPotentialOnly(device, region):
  '''
    Creates the physical models for a Silicon region
    for equilibrium simulation.
  '''

  variables = ("Potential",)
  CreateVT(device, region, variables)
  CreateDensityOfStates(device, region, variables)

  SetSiliconParameters(device, region)

  # require NetDoping
  for i in (
       ("IntrinsicElectrons",       "NIE*exp(Potential/V_t)"),
       ("IntrinsicHoles",           "NIE^2/IntrinsicElectrons"),
       ("IntrinsicCharge",
        "kahan3(IntrinsicHoles, -IntrinsicElectrons, NetDoping)"),
       ("PotentialIntrinsicCharge", "-q * IntrinsicCharge")
      ):
    n = i[0]
    e = i[1]
    CreateNodeModel(device, region, n, e)
    CreateNodeModelDerivative(device, region, n, e, 'Potential')

  CreateQuasiFermiLevels(device, region,
    'IntrinsicElectrons', 'IntrinsicHoles', variables)

  CreateEField(device, region)
  CreateDField(device, region)

  equation(device=device, region=region, name="PotentialEquation",
      variable_name="Potential", node_model="PotentialIntrinsicCharge",
      edge_model="DField", variable_update="log_damp")
```

Fig. 7: Bulk equation for potential only simulation.

```
def CreateSiliconPotentialOnlyContact(device, region, contact,
  is_circuit=False):
  '''
    Creates the potential equation at the contact
    if is_circuit is true, than use node given by
    GetContactBiasName
  '''
  if not InNodeModelList(device, region, "contactcharge_node"):
    CreateNodeModel(device, region, "contactcharge_node", "q*IntrinsicCharge")

  celec_model = \
    "(1e-10 + 0.5*abs(NetDoping+(NetDoping^2 + 4 * NIE^2)^(0.5)))"
  chole_model = \
    "(1e-10 + 0.5*abs(-NetDoping+(NetDoping^2 + 4 * NIE^2)^(0.5)))"
  contact_model = "Potential -{0} + ifelse(NetDoping > 0, \
    -V_t*log({1}/NIE), \
    V_t*log({2}/NIE))".format(GetContactBiasName(contact),
    celec_model, chole_model)

  contact_model_name = GetContactNodeModelName(contact)
  CreateContactNodeModel(device, contact,
    contact_model_name, contact_model)
  CreateContactNodeModel(device, contact,
    "{0}:{1}".format(contact_model_name,"Potential"), "1")
  if is_circuit:
    CreateContactNodeModel(device, contact,
    "{0}:{1}".format(contact_model_name,GetContactBiasName(contact)), "-1")

  if is_circuit:
    contact_equation(device=device, contact=contact,
      name="PotentialEquation", variable_name="Potential",
      node_model=contact_model_name, edge_model="",
      node_charge_model="contactcharge_node",
      edge_charge_model="DField",
      node_current_model="", edge_current_model="",
      circuit_node=GetContactBiasName(contact))
  else:
    contact_equation(device=device, contact=contact,
      name="PotentialEquation", variable_name="Potential",
      node_model=contact_model_name, edge_model="",
      node_charge_model="contactcharge_node", edge_charge_model="DField",
      node_current_model="", edge_current_model="")
```

Fig. 8: Contact equation for potential only simulation.

```
def CreateHFMobility(device, region, mu_n, mu_p, Jn, Jp):
  '''
    Add T derivatives when debugged
    use parameters to set model flags
    Caughey Thomas
  '''

  tdict = {
    'Jn' : Jn,
    'mu_n' : mu_n,
    'Jp' : Jp,
    'mu_p' : mu_p
  }
  tlist = (
    ("vsat_n", "VSATN0 * pow(T, VSATNE)" % tdict, ('T')),
    ("beta_n", "BETAN0 * pow(T, BETANE)" % tdict, ('T')),
    ("Epar_n",
    "ifelse((%(Jn)s * EField) > 0, abs(EField), 1e-15)" \
       % tdict, ('Potential')),
    ("mu_n", "%(mu_n)s * pow(1 + pow((%(mu_n)s*Epar_n/vsat_n), beta_n), \
      -1/beta_n)"
      % tdict, ('Electrons', 'Holes', 'Potential', 'T')),
    ("vsat_p", "VSATP0 * pow(T, VSATPE)" % tdict, ('T')),
    ("beta_p", "BETAP0 * pow(T, BETAPE)" % tdict, ('T')),
    ("Epar_p",
    "ifelse((%(Jp)s * EField) > 0, abs(EField), 1e-15)" \
      % tdict, ('Potential')),
    ("mu_p", "%(mu_p)s * pow(1 + pow(%(mu_p)s*Epar_p/vsat_p, beta_p), \
      -1/beta_p)"
   % tdict, ('Electrons', 'Holes', 'Potential', 'T')),
  )

  variable_list = ('Electrons', 'Holes', 'Potential')
  for (model, equation, variables) in tlist:
    CreateEdgeModel(device, region, model, equation)
    for v in variable_list:
      if v in variables:
        CreateEdgeModelDerivatives(device, region, model, equation, v)

  # This create derivatives automatically
  CreateElectronCurrent(device, region, mu_n='mu_n',
    Potential="Potential", sign=-1, ElectronCurrent="Jn", V_t="V_t_edge")
  CreateHoleCurrent(    device, region, mu_p='mu_p',
    Potential="Potential", sign=-1, HoleCurrent="Jp", V_t="V_t_edge")
  return {
    'mu_n' : 'mu_n',
    'mu_p' : 'mu_p',
    'Jn'   : 'Jn',
    'Jp'   : 'Jp',
  }
```

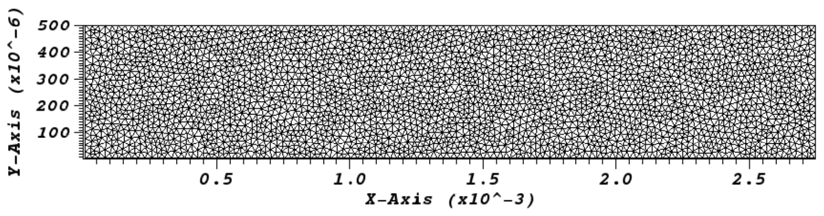
Fig. 9: Current density calculation using high field mobility saturation.

```
/* we are using cm with um scale length*/
/* don't extend from boundary points */
Mesh.CharacteristicLengthExtendFromBoundary=0;
Mesh.Algorithm=5; /*Delaunay*/
Mesh.RandomFactor=1e-7; /*perturbation*/

sf = 1.0e-4;
Mesh.CharacteristicLengthMax = 2.5e-5; /*maximum characteristic length */
/* characterisitic lengths for meshing */
/* results in coarse mesh */
cl1 = 2.5e-5;
cl2 = 2.5e-5;


/* all in microns, final output in cm */
/* we are simulating the intrinsic device*/
device_depth = 5 * sf;
left_space = 5 * sf;
right_space = 5 * sf;
contact_space = 7.5 * sf;
base_contact_width = 5 * sf;
emitter_contact_width = 5 * sf;
device_width = (left_space + base_contact_width + contact_space
  + emitter_contact_width + right_space);
collector_contact_width = device_width;

xb1 = left_space;
xb2 = xb1 + base_contact_width;
xe1 = xb2 + contact_space;
xe2 = xe1 + emitter_contact_width;


/* positive y is in the depth direction */
/* base contact */
Point(1) = {0, 0, 0, cl1};
Point(2) = {xb1, 0, 0, cl2};
Point(3) = {xb2, 0, 0, cl2};
/* emitter contact */
Point(4) = {xe1, 0, 0, cl2};
Point(5) = {xe2, 0, 0, cl2};
Point(6) = {device_width, 0, 0, cl1};

/* collector/bottom */
Point(7) = {0, device_depth, 0, cl1};
Point(8) = {device_width, device_depth, 0, cl1};

Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 5};
Line(5) = {5, 6};
Line(6) = {6, 8};
Line(7) = {7, 8};
Line(8) = {7, 1};
Physical Line("base") = {2};
Physical Line("emitter") = {4};
Physical Line("collector") = {7};
Line Loop(12) = {7, -6, -5, -4, -3, -2, -1, -8};
Plane Surface(13) = {12};
Physical Surface("bjt") = {13};
```

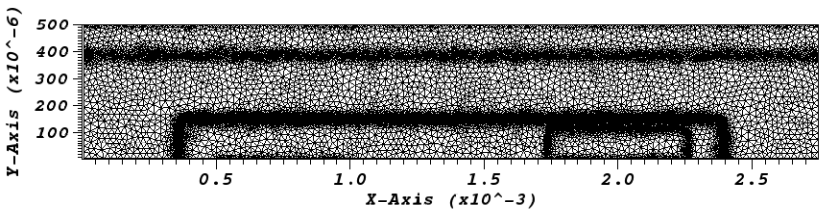
Fig. 10: Gmsh file for the BJT.

```
node_model(device=device, region=region, name="Acceptors", equation='''
  base_doping
  * erfc((y-base_depth)/base_vdiff)
  * erfc(-(x + 0.5*base_width-base_center)/base_hdiff)
  * erfc((x - 0.5*base_width-base_center)/base_hdiff)
''')
node_model(device=device, region=region, name="Donors", equation='''
  emitter_doping
  * erfc((y-emitter_depth)/emitter_vdiff)
  * erfc(-(x + 0.5*emitter_width-emitter_center)/emitter_hdiff)
  * erfc((x - 0.5*emitter_width-emitter_center)/emitter_hdiff)
  + collector_doping
  + sub_collector_doping
  * erfc(-(y-sub_collector_depth)/sub_collector_vdiff)
  * erfc(-(x + 0.5*sub_collector_width-sub_collector_center)
      /sub_collector_hdiff)
  * erfc((x - 0.5*sub_collector_width-sub_collector_center)
      /sub_collector_hdiff)
''')
node_model(device=device, region=region,
  name="NetDoping", equation="Donors-Acceptors;")
```

Fig. 11: Doping profile.



(a)



(b)

Fig. 12: Unrefined and refined meshes. Physical dimensions are in cm.

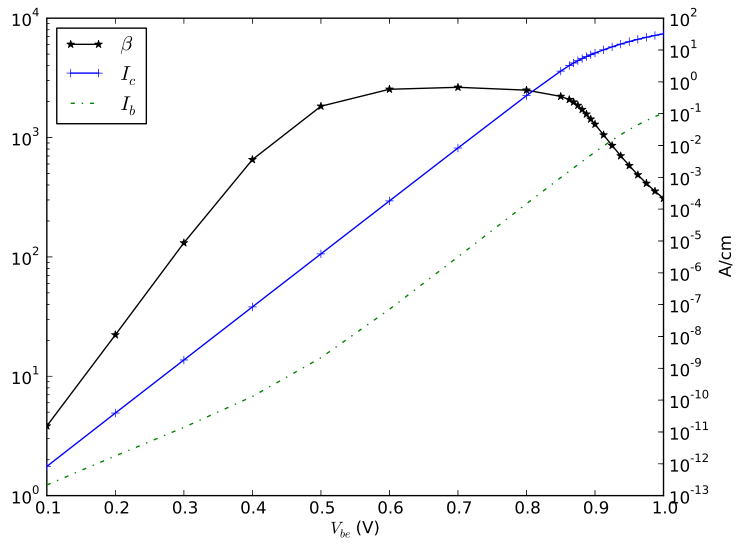


Fig. 13: Gummel plot for $V_{bc} = 0.0$.

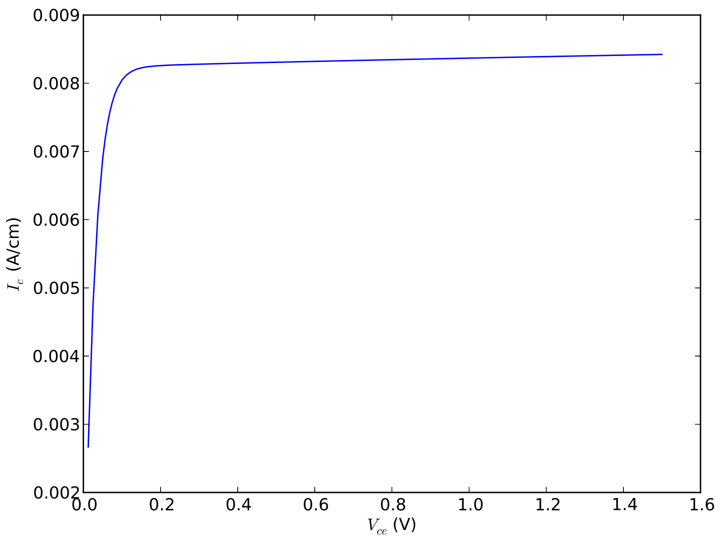


Fig. 14: I_c versus V_{ce} for $V_{be} = 0.7$.

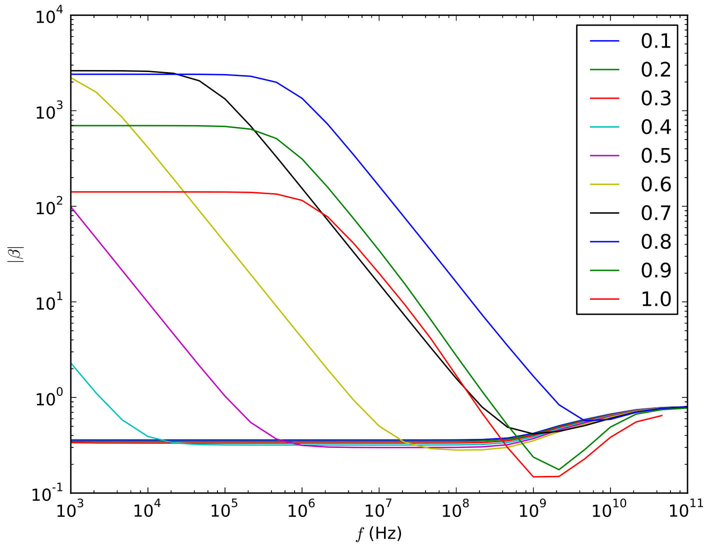


Fig. 15: $|\beta_{ac}|$ versus f for $V_{bc} = 0.0$.

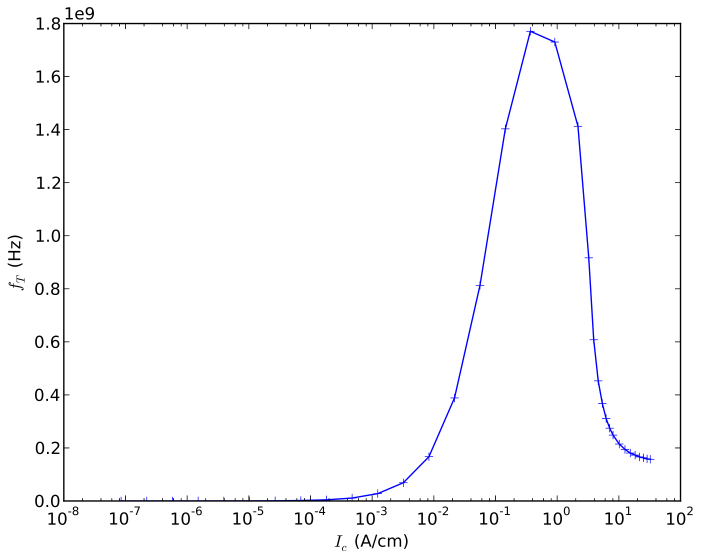


Fig. 16: f_T versus I_c for $V_{bc} = 0.0$.

References

1. FLOOPS/FLOODS home page. URL `http://www.flooxs.tec.ufl.edu`
2. PROPHET. URL `http://www-tcad.stanford.edu/~prophet`
3. Python. Available: `http://www.python.org`
4. SYMDIFF. Available: `http://www.symdiff.org`
5. Tecplot - CFD post processing to visualize simulation data. URL `http://www.tecplot.com`
6. VTK the visualization toolkit. URL `http://www.vtk.org`
7. Armstrong, G., Maiti, C., of Engineering, I., Technology: TCAD for Si, SiGe and GaAs Integrated Circuits. Institution of Engineering and Technology (2007)
8. Arora, N., Hauser, J.R., Roulston, D.: Electron and hole mobilities in silicon as a function of concentration and temperature. IEEE Trans. Electron Devices **29**(2), 292–295 (1982)
9. Canali, C., Majni, G., Minder, R., Ottaviani, G.: Electron and hole drift velocity measurements in silicon and their empirical relation to electric field and temperature. IEEE Trans. Electron Devices **22**(11), 1045–1047 (1975)
10. Caughey, D., Thomas, R.: Carrier mobilities in silicon empirically related to doping and field. Proc. IEEE **55**(12), 2192–2193 (1967)
11. Cogenda: Genius Device Simulator. `http://www.cogenda.com`
12. Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S., Liu, J.W.H.: A supernodal approach to sparse partial pivoting. SIAM J. Matrix Analysis and Applications **20**(3), 720–755 (1999)
13. DEVSIM LLC: DEVSIM User Guide. Available: `http://www.devsim.org`
14. DEVSIM LLC: SYMDIFF User Guide. Available: `http://www.symdiff.org`
15. Duane, M.: The role of TCAD in compact modeling. In: Technical Proceedings of the 2002 International Conference on Modeling and Simulation of Microsystems, pp. 719–721 (2002)
16. Apache Software Foundation: Apache License, Version 2.0. URL `http://www.apache.org/licenses/LICENSE-2.0.html`
17. Getreu, I.: Modeling the Bipolar Transistor. Ian Getreu (2009). URL `http://stores.lulu.com/iangetreu`
18. Geuzaine, C., Remacle, J.F.: Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering **79**, 1309–1331 (2009)
19. Laux, S.E., Byrnes, R.G.: Semiconductor device simulation using generalized mobility models. IBM J. Res. Dev. **29**(3), 289–301 (1985)
20. McAndrew, C.: Predictive technology characterization, missing links between TCAD and compact modeling. In: IEEE SISPAD, pp. 12–17 (2000)
21. Muller, R.S., Kamins, T.I., Chan, M.: Device Electronics for Integrated Circuits, 3 edn. John Wiley & Sons (2002)
22. Roache, P.J.: Fundamentals of Verification and Validation. Hermosa, NM (2009)
23. Scharfetter, D.L., Gummel, H.K.: Large-signal analysis of a silicon Read diode oscillator. IEEE Trans. Electron Devices **ED-16**(1), 64–77 (1969)
24. Selberherr, S.: Analysis and simulation of semiconductor devices. Springer-Verlag, NY (1984)
25. Steigerwald, J., Humphries, P.: TCAD assisted reflection on parameter extraction for compact modeling. In: Proc. IEEE BCTM, pp. 245–252 (2010)
26. Wu, J., Diaz, C.: Expanding role of predictive TCAD in advanced technology development. In: IEEE SISPAD, pp. 167–171 (2013)
27. Yao, W., Gildenblat, G., McAndrew, C., Cassagnes, A.: SP-HV: a scalable surface-potential-based compact model for LDMOS transistors. IEEE Trans. Electron Devices **59**(3), 542–550 (2012)

Index