SEMIDV

A Semiconductor Device Simulator

Chien-Ting Tung
cttung@berkeley.edu

Content

Introduction

SEMIDV is a 2D semiconductor device simulator that solves Poisson-Drift-Diffusion equations using finite difference method. It contains mobility models for Phonon scattering, Coulomb scattering, high-field velocity saturation, ballistic mobility, and length dependent saturation velocity model.

Installation

Use “**pip install semidv**” to install SEMIDV.
It will also install scipy, cython, and fdint.
Cython and fdint are required to compute Fermi-Dirac integral.
Please visit “https://github.com/scott-maddox/fdint” for more information about fdint.

“**import semidv**” in Python to start using it.

Physics

Most physics in SEMIDV are described in the paper “**SEMIDV: A Compact Semiconductor Device Simulator with Quantum Effects**”. This manual will only cover the models that are not included in that paper.

Material

Use **semidv.material()** to create a material object. For example, **parameters = semidv.material()**.
In this class, there are built-in materials with default values.

"Si": { "epsilon": 11.7, "Eg": 1.12, "xi": 4.05, "Nc": 2.8e19, "Nv": 1.04e19, "un": 1400, "up": 450, "vsat_n": 1e7, "beta_n": 2, "vsat_p": 1e7, "beta_p": 2, "vt_n": 1.2e7, "vt_p": 1.2e7, "lambda_n": 7.65e-9, "lambda_p": 7.65e-9, "ua_n": 0, "eu_n": 1, "ud_n": 0, "ucs_n": 1, "nref": 1e18, "ua_p": 0, "eu_p": 1, "ud_p": 0, "ucs_p": 1, "pref": 1e18, "tau_n": 1, "tau_p": 1, "etrap": 0, }

"SiO2": { "epsilon": 3.9, "Eg": 8.9, "xi": 0.95, "Nc": 1e18, "Nv": 1e18, "un": 1, "up": 1, "vsat_n": 1e7, "beta_n": 2, "vsat_p": 1e7, "beta_p": 2, "vt_n": 1.2e7, "vt_p": 1.2e7, "lambda_n": 7.65e-9, "lambda_p": 7.65e-9, "ua_n": 0, "eu_n": 1, "ud_n": 0, "ucs_n": 1, "nref": 1e18, "ua_p": 0, "eu_p": 1, "ud_p": 0, "ucs_p": 1, "pref": 1e18, "tau_n": 1, "tau_p": 1, "etrap": 0, }

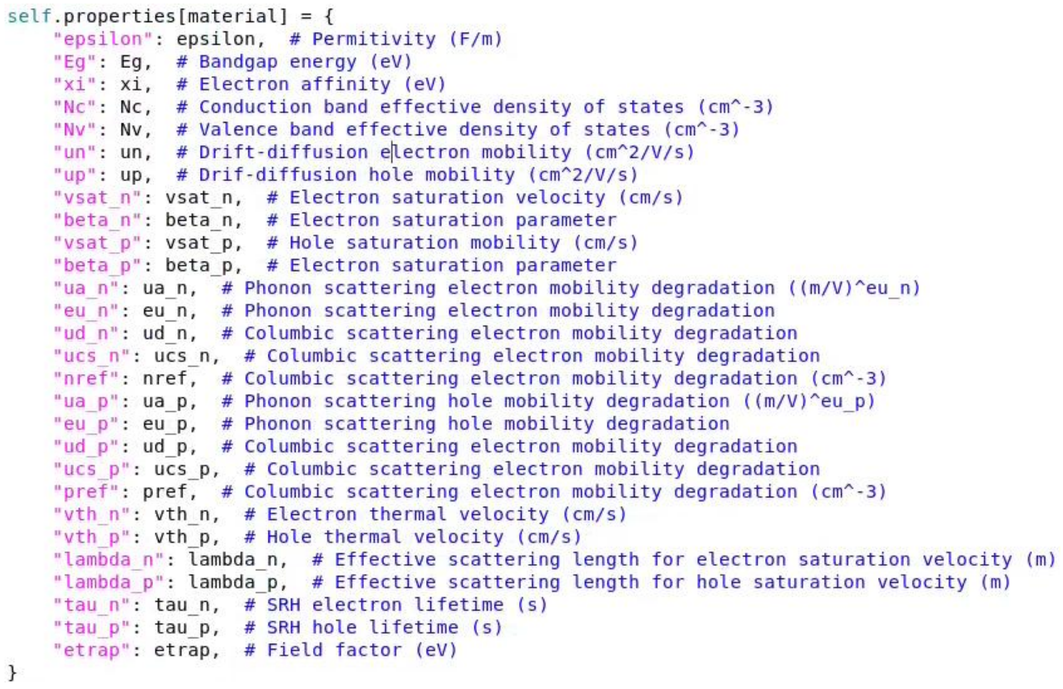
"Si3N4": { "epsilon": 3, "Eg": 5.3, "xi": 2.15, "Nc": 1e18, "Nv": 1e18, "un": 1, "up": 1, "vsat_n": 1e7, "beta_n": 2, "vsat_p": 1e7, "beta_p": 2, "vt_n": 1.2e7, "vt_p": 1.2e7, "lambda_n": 7.65e-9, "lambda_p": 7.65e-9, "ua_n": 0, "eu_n": 1, "ud_n": 0, "ucs_n": 1, "nref": 1e18, "ua_p": 0, "eu_p": 1, "ud_p": 0, "ucs_p": 1, "pref": 1e18, "tau_n": 1, "tau_p": 1, "etrap": 0, }

"HfO2": { "epsilon": 25, "Eg": 5.7, "xi": 1.2, "Nc": 1e18, "Nv": 1e18, "un": 1, "up": 1, "vsat_n": 1e7, "beta_n": 2, "vsat_p": 1e7, "beta_p": 2, "vt_n": 1.2e7, "vt_p": 1.2e7, "lambda_n": 7.65e-9, "lambda_p": 7.65e-9, "ua_n": 0, "eu_n": 1, "ud_n": 0, "ucs_n": 1, "nref": 1e18, "ua_p": 0, "eu_p": 1, "ud_p": 0, "ucs_p": 1, "pref": 1e18, "tau_n": 1, "tau_p": 1, "etrap": 0, }

"Metal": { "epsilon": 1, "Eg": 0, "xi": 4.5, "Nc": 1e25, "Nv": 1e25, "un": 1, "up": 1, "vsat_n": 1e7, "beta_n": 2, "vsat_p": 1e7, "beta_p": 2, "vt_n": 1.2e7, "vt_p": 1.2e7, "lambda_n": 7.65e-9, "lambda_p": 7.65e-9, "ua_n": 0, "eu_n": 1, "ud_n": 0, "ucs_n": 1, "nref": 1e18, "ua_p": 0, "eu_p": 1, "ud_p": 0, "ucs_p": 1, "pref": 1e18, "tau_n": 1, "tau_p": 1, "etrap": 0, }

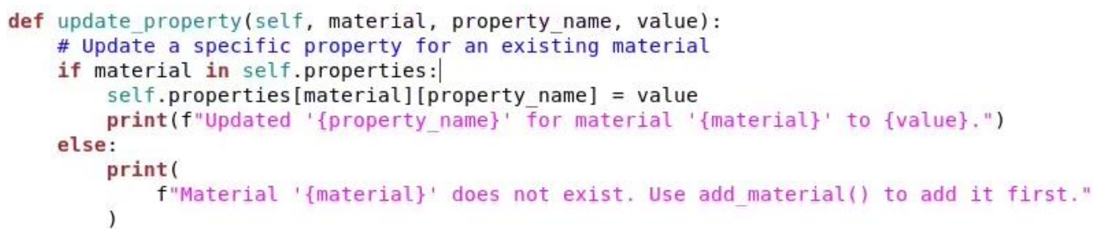
"Ge": { "epsilon": 16.0, "Eg": 0.66, "xi": 4.0, "Nc": 1.2e19, "Nv": 0.6e19, "un": 3900, "up": 1900, "vsat_n": 6e6, "beta_n": 2, "vsat_p": 6e6, "beta_p": 2, "vt_n": 3.1e7, "vt_p": 1.9e7, "lambda_n": 7.65e-9, "lambda_p": 7.65e-9, "ua_n": 0, "eu_n": 1, "ud_n": 0, "ucs_n": 1, "nref": 1e18, "ua_p": 0, "eu_p": 1, "ud_p": 0, "ucs_p": 1, "pref": 1e18, "tau_n": 1, "tau_p": 1, "etrap": 0, }

To add new material, please use **add_material()** to input material properties.



```
self.properties[material] = {
    "epsilon": epsilon,  # Permitivity (F/m)
    "Eg": Eg,  # Bandgap energy (eV)
    "xi": xi,  # Electron affinity (eV)
    "Nc": Nc,  # Conduction band effective density of states (cm^-3)
    "Nv": Nv,  # Valence band effective density of states (cm^-3)
    "un": un,  # Drift-diffusion electron mobility (cm^2/V/s)
    "up": up,  # Drif-diffusion hole mobility (cm^2/V/s)
    "vsat_n": vsat_n,  # Electron saturation velocity (cm/s)
    "beta_n": beta_n,  # Electron saturation parameter
    "vsat_p": vsat_p,  # Hole saturation mobility (cm/s)
    "beta_p": beta_p,  # Electron saturation parameter
    "ua_n": ua_n,  # Phonon scattering electron mobility degradation ((m/V)^eu_n)
    "eu_n": eu_n,  # Phonon scattering electron mobility degradation
    "ud_n": ud_n,  # Columbic scattering electron mobility degradation
    "ucs_n": ucs_n,  # Columbic scattering electron mobility degradation
    "nref": nref,  # Columbic scattering electron mobility degradation (cm^-3)
    "ua_p": ua_p,  # Phonon scattering hole mobility degradation ((m/V)^eu_p)
    "eu_p": eu_p,  # Phonon scattering hole mobility degradation
    "ud_p": ud_p,  # Columbic scattering electron mobility degradation
    "ucs_p": ucs_p,  # Columbic scattering electron mobility degradation
    "pref": pref,  # Columbic scattering electron mobility degradation (cm^-3)
    "vth_n": vth_n,  # Electron thermal velocity (cm/s)
    "vth_p": vth_p,  # Hole thermal velocity (cm/s)
    "lambda_n": lambda_n,  # Effective scattering length for electron saturation velocity (m)
    "lambda_p": lambda_p,  # Effective scattering length for hole saturation velocity (m)
    "tau_n": tau_n,  # SRH electron lifetime (s)
    "tau_p": tau_p,  # SRH hole lifetime (s)
    "etrap": etrap,  # Field factor (eV)
}
```

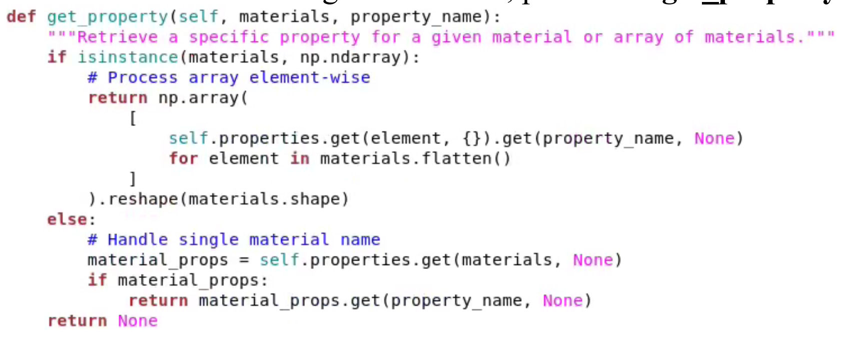
To change the property of existing materials, please use **update_property()**.



```
def update_property(self, material, property_name, value):
    # Update a specific property for an existing material
    if material in self.properties:
        self.properties[material][property_name] = value
        print(f"Updated '{property_name}' for material '{material}' to {value}.")
    else:
        print(
            f"Material '{material}' does not exist. Use add_material() to add it first."
        )
```

For example, **parameters.update_property("Si", "un", 310)**.

To check the value for a given material, please use **get_property()**.



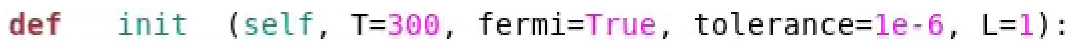
```
def get_property(self, materials, property_name):
    """Retrieve a specific property for a given material or array of materials."""
    if isinstance(materials, np.ndarray):
        # Process array element-wise
        return np.array(
            [
                self.properties.get(element, {}).get(property_name, None)
                for element in materials.flatten()
            ]
        ).reshape(materials.shape)
    else:
        # Handle single material name
        material_props = self.properties.get(materials, None)
        if material_props:
            return material_props.get(property_name, None)
    return None
```

Device Structure

Create Device Object

To set up a device for simulation, first, create a device object with **semidv.device()**.

For example, **device = semidv.device(T=300, fermi=True, tolerance=1e-6, L=12e-9)**.



```
def   init  (self, T=300, fermi=True, tolerance=1e-6, L=1):
```

T is the ambient temperature for device simulation. The default is 300K. In the current version, there is no self-heating, so T is also the device temperature.

tolerance is the convergence criteria. The default is 1e-6 V.

L is a parameter for ballistic mobility and saturation velocity calculation. The default is 1.

Create Device Structure

```
"""
Build a 2D array representing the device structure with separate spatial steps for x and y.

Parameters:
- domain_size: tuple (width, height) in meters, e.g., (1e-6, 1e-6)
- spatial_steps: tuple (x_step, y_step) in meters, e.g., (1e-9, 2e-9)
- materials: list of dictionaries with material and range, e.g.,
  [{"material": "Si", "x": (1e-9, 2e-9), "y": (2e-9, 3e-9)}]
- doping_profiles: list of dictionaries with doping type, concentration (cm^-3), and range, e.g.,
  [{"type": "n-type", "concentration": 1e18, "x": (1e-9, 2e-9), "y": (2e-9, 3e-9)}]

Returns:
- A dictionary containing:
  - "structure": A 2D numpy array representing the device structure materials.
  - "doping": A 2D numpy array representing the doping concentrations (n-type positive, p-type negative).
"""
```

Users need to specify the simulation range in x, y axis, and the step size.

For example,
domain = (Thickness, Length)
steps = (dx, dy)

The device structure is defined by a list specifying the materials for different regions.

The default material for all regions is Silicon. Define the material by specifying the start and end point in x, y axis.

For example,
materials = [
{"material": "Metal", "x": (0, Tsp-Tox-Thk), "y": (Lsd+Lsp+Thk, Length-Lsd-Lsp-Thk)},
{"material": "Metal", "x": (Thickness-Tsp+Tox+Thk, Thickness), "y": (Lsd+Lsp+Thk, Length-Lsd-Lsp-Thk)},
{"material": "HfO2", "x": (Tsp-Tox-Thk, Tsp-Tox), "y": (Lsd+Lsp, Length-Lsd-Lsp)},
{"material": "HfO2", "x": (Thickness-Tsp+Tox, Thickness-Tsp+Tox+Thk), "y": (Lsd+Lsp, Length-Lsd-Lsp)},
{"material": "HfO2", "x": (0, Tsp-Tox-Thk), "y": (Lsd+Lsp, Lsd+Lsp+Thk)},
{"material": "HfO2", "x": (0, Tsp-Tox-Thk), "y": (Length-Lsd-Lsp-Thk, Length-Lsd-Lsp)},
{"material": "HfO2", "x": (Thickness-Tsp+Tox+Thk, Thickness), "y": (Lsd+Lsp, Lsd+Lsp+Thk)},
{"material": "HfO2", "x": (Thickness-Tsp+Tox+Thk, Thickness), "y": (Length-Lsd-Lsp-Thk, Length-Lsd-Lsp)},
{"material": "SiO2", "x": (Tsp-Tox, Tsp), "y": (Lsd+Lsp, Length-Lsd-Lsp)},
{"material": "SiO2", "x": (Thickness-Tsp, Thickness-Tsp+Tox), "y": (Lsd+Lsp, Length-Lsd-Lsp)},
{"material": "Si3N4", "x": (0, Tsp), "y": (Lsd, Lsd+Lsp)},
{"material": "Si3N4", "x": (0, Tsp), "y": (Length-Lsd-Lsp, Length-Lsd)},
{"material": "Si3N4", "x": (Thickness-Tsp, Thickness), "y": (Lsd, Lsd+Lsp)},
{"material": "Si3N4", "x": (Thickness-Tsp, Thickness), "y": (Length-Lsd-Lsp, Length-Lsd)}
]

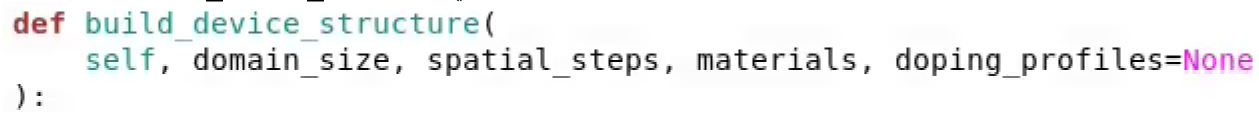
Users can define doping profiles by creating a doping list.

For example,
doping = [
{"type": "n-type", "concentration": 2e20, "x": (0, Thickness), "y": (0, Lsd)},
{"type": "n-type", "concentration": 2e20, "x": (0, Thickness), "y": (Length-Lsd, Length)},
{"type": "n-type", "concentration": "2e20*10(-2*((y-"+str(Lsd)+")/"+str(Lsp)+")**2)", "x": (Tsp, Tsp+Tch), "y": (Lsd, Length)},**
{"type": "n-type", "concentration": "2e20*10(-2*((y-"+str(Length-Lsd)+")/"+str(Lsp)+")**2)", "x": (Tsp, Tsp+Tch), "y": (0, Length-Lsd)},**
{"type": "p-type", "concentration": 1e15, "x": (Tsp, Tsp+Tch), "y": (Lsd+Lsp, Length-Lsd-Lsp)}
]

Doping can be constant.
{"type": "n-type", "concentration": **2e20**, "x": (0, Thickness), "y": (0, Lsd)}

It can also be a function by a string.
{"type": "n-type", "concentration": **"2e20*10**(-2*((y-"+str(Lsd)+")/"+str(Lsp)+")**2)"**, "x": (Tsp, Tsp+Tch), "y": (Lsd, Length)}

Then, use **build_device_structure()** to build the device.



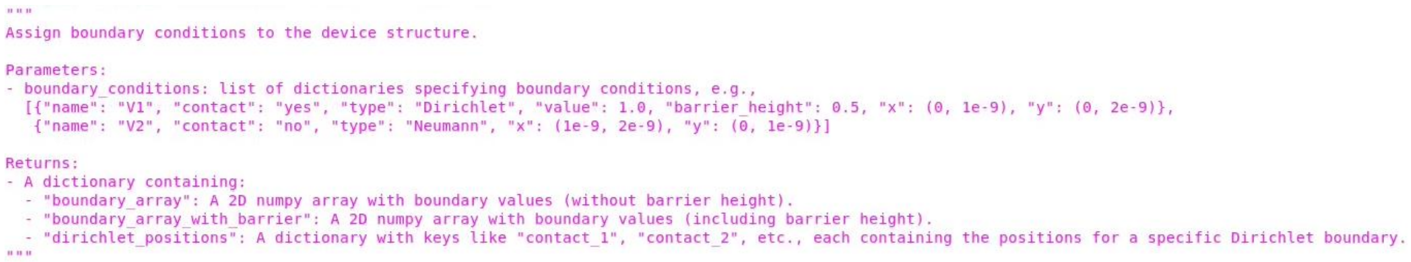
```
def build_device_structure(
        self, domain_size, spatial_steps, materials, doping_profiles=None
):
```

For example, **device.build_device_structure(domain, steps, materials, doping)**.

Then, input the material parameters by **device.materialproperties()**.

For example, **device.materialproperties(parameters)**.

Contacts and Boundary Conditions



```
"""
Assign boundary conditions to the device structure.

Parameters:
- boundary_conditions: list of dictionaries specifying boundary conditions, e.g.,
  [{"name": "V1", "contact": "yes", "type": "Dirichlet", "value": 1.0, "barrier_height": 0.5, "x": (0, 1e-9), "y": (0, 2e-9)},
   {"name": "V2", "contact": "no", "type": "Neumann", "x": (1e-9, 2e-9), "y": (0, 1e-9)}]

Returns:
- A dictionary containing:
  - "boundary_array": A 2D numpy array with boundary values (without barrier height).
  - "boundary_array_with_barrier": A 2D numpy array with boundary values (including barrier height).
  - "dirichlet_positions": A dictionary with keys like "contact_1", "contact_2", etc., each containing the positions for a specific Dirichlet boundary.
"""
```

If contact is “yes”, it means the boundary is connected to electrode. It can be either **Dirichlet** or **Neumann** boundary. For **Dirichlet boundary**, a value for contact barrier in eV is required. For **Neumann boundary**, the “value” is ignored.

If contact is “no”, it will automatically be **Neumann boundary**.

Please create a boundary list and use **assign_boundary_conditions()** to assign boundaries.

For example,

```
boundary = [
        {"name": "gate", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height": 0,
"x": (0, 0), "y": (Lsd+Lsp+Thk, Length-Lsd-Lsp-Thk)},
        {"name": "gate", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height": 0,
"x": (Thickness, Thickness), "y": (Lsd+Lsp+Thk, Length-Lsd-Lsp-Thk)},
        {"name": "source", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height":
0, "x": (Thickness, Thickness), "y": (0, Lsd)},
        {"name": "drain", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height": 0,
"x": (Thickness, Thickness), "y": (Length-Lsd, Length)}
   ]
```

Charge Integration

SEMIDV calculates charges by specifying the region to integrate charges. Define a directory and use **add_charge_region()** to specify the charge integration region.

For example,

```
channel = {"name": "channel", "x": (Tsp, Tsp+Tch), "y": (Lsd+Lsp, Length-Lsd-Lsp)}
device.add_charge_region(channel)
```

Device Simulation

Initialize

First, initialize device with equilibrium condition using **Initialize()**

For example,

```
Device.Initialize()
```

Solve Poisson-Drift-Diffusion

Use **getmatrix()** to get a iteration-independent Poisson finite difference matrix. Users only need to do this once.

Then, use **Poisson(damping=1)** to solve the Poisson equation.
damping can control how fast the simulator updates the solution. Smaller damping value can improve convergence but increase the iteration steps.
To call the error in the Poisson equation, use call the error variable in the device object.
For example, **device.error**. It will return an error in volts.
After solving the Poisson equation, E_C and E_V will be updated in the object.

Use **DriftDiffusion(recombination=False)** to solve Drift-Diffusion equations.
Users can turn on the recombination module. The default is False.
SEMIDV only has Shockley-Read-Hall (SRH) recombination model now.
SRH = (n * p - ni^2) / (tau_p* n+ ni * exp(etrap / kbT)) + tau_n * (p + ni * exp(-etrap / kbT)))
Drift-Diffusion equations will update Efn and Efp in the object.

Users need to create a loop to do Gummel iteration till simulation converges.

For example,

```
device.getmatrix()
error = 1
while error > device.tolerance:
    device.Poisson()
    device.DriftDiffusion()
    error = device.error
```

Solve Quantum-Poisson-Drift-Diffusion

To include quantum correction from the Localization Landscape Theory, users need to use **quantum_getmatrix()** to obtain the iteration-independent part of the Hamiltonian first.

Then, users can choose to use either electron or hole to calculate quantum potential.
To use electron, use **equantum(Kq=0)**.
To use hole, use **hquantum(Kq=0)**.
Only one of them can be used not both.
Kq is a parameter for the strength of quantum correction. The default is 0. The larger the Kq, the weaker the quantum effect is.
Quantum will update quantum potential corrections QCn and QCp in the object.

To solve the device with quantum correction, users need to add **device.quantum_getmatrix()** into the iteration.

For example,

```
device.getmatrix()
device.quantum_getmatrix()
error = 1
```

```
while error > device.tolerance:
        device.Poisson()
        device.equantum()
        device.DriftDiffusion()
        error = device.error
```

Currents and Charges

After simulation, users can use **getcurrent()** and **getcharge()** to calculate current at each contact, and charge in given integration region.
The currents and charges are stored in **current** and **charge**. Both of them are lists. Users can obtain the current or charge at different place with the "contact name" or "charge region name".

For example,
device.current[“drain”]
device.charge[“gate”]

Initial Guess

Users can use **update_initial_guess(Ec, Efn, Efp)** to assign Ec, Efn, and Efp at the beginning of iterations.

Built-in Solver

SEMIDV also provides a solver with predefined loops for easy use.
This solver can do double voltage sweeps and extrapolate previous solutions as initial guess.
Users need to define voltages, boundaries, and use **semidv.solve().IVsweep()** do the sweep**.**

```
def   init  (self, device, boundary, damping=1, recombination=False, quantum=False, Kq=0):
IVsweep(self, V1 contact name, V1 sweep, V2 contact name, V2 sweep, results):
```

This solver can recognize the name of each contact and update the electrode voltage based on the contact name.

For example,

```
Vs=0
Vd_sweep = np.array([0.05,0.65])
Vg_sweep = np.linspace(0.0,0.7,15)

boundary = [
        {"name": "gate", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height": 0,
"x": (0, 0), "y": (Lsd+Lsp+Thk, Length-Lsd-Lsp-Thk)},
        {"name": "gate", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height": 0,
"x": (Thickness, Thickness), "y": (Lsd+Lsp+Thk, Length-Lsd-Lsp-Thk)},
        {"name": "source", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height":
```

0, "x": (Thickness, Thickness), "y": (0, Lsd)},
{"name": "drain", "contact": "yes", "type": "Dirichlet", "value": Vs, "barrier_height": 0, "x": (Thickness, Thickness), "y": (Length-Lsd, Length)}
]

results={}
results = semidv.solve(device, boundary, damping=1, recombination=False, quantum=False).IVsweep("gate", Vg_sweep, "drain", Vd_sweep, results)

Results will be stored in a directory using voltages as keys. It contains objects, currents, and charges for each bias.

```
results[(V1, V2)] = {
    "model": deepcopy(self.device),
    "current": self.device.current.copy(),
    "charge": self.device.charge.copy(),
}
```

Visualization

SEMIDV provides a visualizer that can generate device plots for different quantities.
First, users need to create a visual object using **semidv.visual()** by input a device object.
Each plot function has a save option. The default is False. If enter True, it will save the figure into “figure.png”. Users can enter a figure name, and it will save the figure with that name.

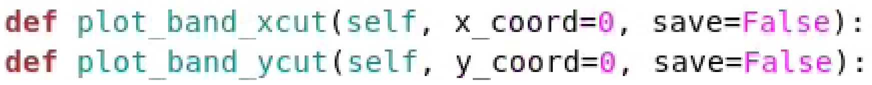
```
def __init__(self, device):
def plot_structure(self, save=False):
```

For example,
deviceplot = semidv.visual(device)
deviceplot.plot_structure(save="structure.tif")

SEMIDV visual provides several 2D contour plots.

plot_structure(): plot the device structure with materials
plot_potential(): plot 2D electrostatic potential
plot_efermi(): plot 2D electron quasi-fermi level Efn
plot_hfermi(): plot 2D hole quasi-fermi level Efp
plot_doping(): plot 2D doping distribution
plot_n(): plot 2D electron distribution
plot_p(): plot 2D hole distribution
plot_Efield(): plot 2D vector plot of electric field
plot_Jn(): plot 2D vector plot of electron current
plot_Jp(): plot 2D vector plot of hole current

SEMIDV visual also provides 1D cut line plot for several quantities. Users will need to specify x or y coordinate for each plot.



```
def plot_band_xcut(self, x_coord=0, save=False):
def plot_band_ycut(self, y_coord=0, save=False):
```

For example,
deviceplot.plot_band_xcut(6e-9, save="band.tif")

Here are the available 1D plots.
plot_band_xcut(): Band diagram along the y axis
plot_band_ycut(): Band diagram along the x axis
plot_efield_xcut(): Electric field along the y axis
plot_efield_ycut(): Electric field along the x axis
plot_q_xcut(): n and p along the y axis
plot_q_ycut(): n and p along the x axis
plot_J_xcut(): Jn and Jp along the y axis
plot_J_ycut(): Jn and Jp along the x axis
plot_vn_xcut(): Electron velocity along the y axis
plot_vn_ycut(): Electron velocity along the x axis
plot_vp_xcut(): Hole velocity along the y axis
plot_vp_ycut(): Hole velocity along the x axis

Physical Quantities

This section lists the accessible physical variables in the device object.

Ec: Conduction band energy
Ev: Valence band energy
Efn: Electron quasi-fermi level
Efp Hole quasi-fermi level
n: Electron concentration
p: Hole concentration
NB: Net doping concentration
QCn: Electron quantum potential
QCp: Hole quantum potential
vnx: Component of electron velocity in the x-direction
vny: Component of electron velocity in the y-direction
vpx: Component of hole velocity in the x-direction
vpy: Component of hole velocity in the y-direction
Jnx: Component of electron current density in the x-direction
Jny: Component of electron current density in the y-direction
Jpx: Component of hole current density in the x-direction
Jpy: Component of hole current density in the y-direction

current: A directory of currents
charge: A directory of charges
T: Ambient temperature
Nc: Conduction band effective density of states
Nv: Valence band effective density of states
me: Electron effective mass (self.h**2 / 2 / np.pi / self.kbT * (self.Nc / 2.0) ** (2.0 / 3.0))
mh: Hole effective mass (self.h**2 / 2 / np.pi / self.kbT * (self.Nv / 2.0) ** (2.0 / 3.0))
epsilon: Permittivity
Eg: Bandgap
xi: Electron affinity
un: low-field drift-diffusion electron mobility
up: low-field drift-diffusion hole mobility
lambda_n: Effective scattering length for electron saturation velocity
lambda_p: Effective scattering length for hole saturation velocity
vsat_n: Electron saturation velocity
vsat_p: Hole saturation velocity
vt_n: Electron thermal velocity
vt_p: Hole thermal velocity
ub_n: Electron ballistic mobility
ub_p: Hole ballistic mobility
beta_n: Electron velocity saturation parameter
beta_p: Hole velocity saturation parameter
ua_n: Electron Phonon scattering parameter
ua_p: Hole Phonon scattering parameter
eu_n: Electron Phonon scattering parameter
eu_p: Hole Phonon scattering parameter
ud_n: Electron Coulomb scattering parameter
ud_p: Hole Coulomb scattering parameter
ucs_n: Electron Coulomb scattering parameter
ucs_p: Hole Coulomb scattering parameter
nref: Electron Coulomb scattering parameter
pref: Hole Coulomb scattering parameter
ni: Intrinsic carrier concentration
tau_n: Electron SRH recombination lifetime
tau_p: Hole SRH recombination lifetime
etrap: SRH Trap energy