

**Semiconductor Process and Device Fabrication**

**(ECE 6861)**

*TERM PROJECT REPORT*

by İbrahim Furkan TEZCAN

10.June.2024

# Example:

* Brief manual for my GUI and input conditions:

A screenshot of a computer

Description automatically generated

*Green area: Input panel (Impurity, T0, T1, Cb, t0, t1, Cth, xd) and progress bar*

*Red areas: Output screen (Concentration profiles and junction depths)*

* Output:

A screenshot of a computer

Description automatically generated

Junction depth for pre-deposition: 0.16µm

Junction depth for drive-in: 2.93µm

# Code:

You can access my code by clicking GitHub link or copying it from below.

## GUI.py

from PyQt6.QtWidgets import QApplication

import GUI\_background

import sys # Only needed for access to command line arguments

#######################

# You need one (and only one) QApplication instance per application.

# Pass in sys.argv to allow command line arguments for your app.

# If you know you won't use command line arguments QApplication([]) works too.

# Subclass QMainWindow to customize your application's main window

app = QApplication(sys.argv)

window = GUI\_background.MainWindow()

window.setWindowTitle("Numerical Simulation Tool for Diffusion by Ibrahim Furkan Tezcan | v1.1-stable | Release: 10.Jun.2024")

window.show()

app.exec()

## GUI\_background.py

from numeric\_sim import N\_simulation as nSim

from numeric\_sim import C\_profiles as cProf

from numeric\_sim import Impurity

import matplotlib

matplotlib.use('QtAgg')

from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg, NavigationToolbar2QT as NavigationToolbar

from matplotlib.figure import Figure

import numpy as np

from PyQt6.QtGui import QValidator

from PyQt6.QtCore import QSize, Qt

from PyQt6.QtWidgets import (

QComboBox,

QSpinBox,

QDoubleSpinBox,

QLabel,

QMainWindow,

QProgressBar,

QPushButton,

QVBoxLayout,

QHBoxLayout,

QGridLayout,

QFormLayout,

QWidget,

QFileDialog,

QMenuBar,

QLineEdit,

QApplication,

QSizePolicy,

QDialog,

QMessageBox

)

class ScientificDoubleSpinBox(QDoubleSpinBox):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(ScientificDoubleSpinBox, self).\_\_init\_\_(\*args, \*\*kwargs)

def textFromValue(self, value):

return "{:.3e}".format(value)

def validate(self, text, pos):

try:

if text == "" or text[-1] in 'e.-+':

return (QValidator.State.Intermediate, text, pos)

float(text)

return (QValidator.State.Acceptable, text, pos)

except ValueError:

return (QValidator.State.Invalid, text, pos)

class MainWindow(QMainWindow):

def \_\_init\_\_(self): # Initialize the main window

super().\_\_init\_\_()

# Define default parameters

global \_Cb\_default, \_Cth\_default, \_Dopant\_default, \_T0\_default, \_T1\_default, \_xL\_default, \_xL\_unit\_default, \_t0\_default, \_t1\_default, \_prgrss\_default, \_prgrss\_lgnd\_default, \_prgrss\_val\_default, \_xJun1\_default, \_xJun2\_default

\_Cb\_default = 0 # in atoms/cm^3

\_Cth\_default = 1e15 # in atoms/cm^3

\_Dopant\_default = 2 # Boron

\_T0\_default = 900 # in °C

\_T1\_default = 900 # in °C

\_xL\_default = 600 # in µm or nm -> should be converted to cm before simulation

\_xL\_unit\_default = 1 # 0: µm, 1: nm

\_t0\_default = 3000 # in s

\_t1\_default = 3000 # in s

\_prgrss\_default = 0

\_prgrss\_lgnd\_default = "Available"

\_prgrss\_val\_default = "..."

\_xJun1\_default = "..."

\_xJun2\_default = "..."

# Create a container widget

widget = QWidget()

self.setCentralWidget(widget)

#################################

## Label/Widget for extra part ##

#################################

# Create a butoons for "About", "Help", and "Exit" parts

aboutB = QPushButton("About")

helpB = QPushButton("Help")

exitB = QPushButton("Exit")

# Change the button colors

aboutB.setStyleSheet("color: #C0C0C0")

helpB.setStyleSheet("color: #C0C0C0")

exitB.setStyleSheet("color: #FF6347")

# "About" button functionality

aboutB.clicked.connect(lambda: self.show\_popup(dialog\_text = "Numerical Simulation Tool for Diffusion\n\n Author: Ibrahim Furkan Tezcan\n\n Version: 1.1-stable\n\n Release: 09.Jun.2024\n\n Last Update: 10.Jun.2024",

dialog\_title = "About"))

# "Help" button functionality

helpB.clicked.connect(lambda: self.show\_popup(dialog\_text = "The simulation parameters are: the type of dopant, the temperature for predeposition, the temperature for drive-in, the spatial length, the predeposition time, and the drive-in time.\n\n The simulation results are the concentration profiles of the dopant in the silicon wafer and the junction depths for predeposition and drive-in. The concentration profiles are plotted in linear and logarithmic scales. The junction depths are given in the unit of the spatial length.\n\n The simulation is terminated by clicking the \"Simulate!\" button. The simulation can be terminated by clicking the \"Terminate!\" button during simulation. The parameters and plots can be reset by clicking the \"Reset!\" button.",

dialog\_title = "Help")

)

# Exit button functionality

exitB.clicked.connect(self.close)

######################################

## Label/Widget for plotting graphs ##

######################################

# Create two figures

self.figure\_lin = Figure()

self.figure\_log = Figure()

# Linear Scale Plot Labels/Canvas

linearPlotCanvasTitle = QLabel("Concentration Profile (Linear Scale)")

linearPlotCanvasTitle.setAlignment(Qt.AlignmentFlag.AlignCenter)

self.linearPlotCanvas = FigureCanvasQTAgg(self.figure\_lin)

# Log Scale Plot Labels/Canvas

logPlotCanvasTitle = QLabel("Concentration Profile (Logarithmic Scale)")

logPlotCanvasTitle.setAlignment(Qt.AlignmentFlag.AlignCenter)

self.logPlotCanvas = FigureCanvasQTAgg(self.figure\_log)

# Create a toolbar for the canvas

self.linPC\_tb = NavigationToolbar(self.linearPlotCanvas, self)

self.logPC\_tb = NavigationToolbar(self.logPlotCanvas, self)

################################################

## Label/Widget for numerical simulation tool ##

################################################

# Dopant selection panel

Dopant\_label = QLabel("Select impurity: ")

self.Dopant\_in = QComboBox()

self.Dopant\_in.addItem("Antimony (Sb)")

self.Dopant\_in.addItem("Arsenic (As)")

self.Dopant\_in.addItem("Boron (B)")

self.Dopant\_in.addItem("Phosphorus (P)")

self.Dopant\_in.setCurrentIndex(\_Dopant\_default)

self.dopantProfile\_list = self.createDopantProfile()

self.Dopant = self.dopantProfile\_list[self.Dopant\_in.currentIndex()]

# User input panel

Cb\_in = QLabel("Clipping conc. (Cb - Optional): ") # Previously: Cb\_in = QLabel("Choose minimum clipping concentration (Cb)")

self.Cb = ScientificDoubleSpinBox()

self.Cb.setSuffix(" atoms/cm^3")

self.Cb.setMinimum(0)

self.Cb.setMaximum(1e18)

self.Cb.setValue(\_Cb\_default)

# print(Cb)

Cth\_in = QLabel("Background conc. (Cth): ") # Previously: Cth\_in = QLabel("Choose threshold concentration for junction depth calculation (Cth)")

self.Cth = ScientificDoubleSpinBox()

self.Cth.setSuffix(" atoms/cm^3")

self.Cth.setMinimum(0)

self.Cth.setMaximum(1e18)

self.Cth.setValue(\_Cth\_default)

# print(Cth)

T0\_in = QLabel("Temperature for predep. (T0): ")

self.T0 = QSpinBox()

self.T0.setSuffix(" °C")

self.T0.setMinimum(900)

self.T0.setMaximum(1200)

self.T0.setValue(\_T0\_default)

# print(T0)

T1\_in = QLabel("Temperature for drive-in (T1): ")

self.T1 = QSpinBox()

self.T1.setSuffix(" °C")

self.T1.setMinimum(900)

self.T1.setMaximum(1200)

self.T1.setValue(\_T1\_default)

# print(T1)

xL\_in = QLabel("Spatial length (xd): ")

self.xL\_unit = QComboBox()

self.xL\_unit.addItem("µm")

self.xL\_unit.addItem("nm")

self.xL\_unit.setCurrentIndex(\_xL\_unit\_default)

self.xL\_inUnit = QSpinBox()

self.xL\_inUnit.setMinimum(0)

self.xL\_inUnit.setMaximum(1000)

self.xL\_inUnit.setValue(\_xL\_default)

self.xL = self.xL\_unitConverter(self.xL\_inUnit.value())

# print(xL)

t0\_in = QLabel("Predeposition time (t0): ")

self.t0 = QSpinBox()

self.t0.setSuffix(" s")

self.t0.setMinimum(0)

self.t0.setMaximum(86400) # 8.64e4 s = 24 hours

self.t0.setValue(\_t0\_default)

# print(t0)

t1\_in = QLabel("Drive-in time (t1): ")

self.t1 = QSpinBox()

self.t1.setSuffix(" s")

self.t1.setMinimum(0)

self.t1.setMaximum(86400) # 8.64e4 s = 24 hours

self.t1.setValue(\_t1\_default)

# print(t1)

# Clear button

Clear = QPushButton("Reset!")

# Clear button functionality

Clear.clicked.connect(lambda: self.clear()) # Clear the canvas

# "Simulate!" button

self.startSimulation = QPushButton("Simulate!")

self.startSimulation.setStyleSheet("color: #FFD700")

# "Simulate!" button functionality

self.startSimulation.clicked.connect(lambda: self.toggle\_simulation())

# Progress bar

self.progress = QProgressBar()

self.progress.setMinimum(0)

self.progress.setMaximum(100)

self.progress.setValue(0)

# Printed parts

xJunc\_1\_label = QLabel("Junction depth for predep.")

xJunc\_2\_label = QLabel("Junction depth for drive-in")

self.xJunc\_1 = QLabel("...")

self.xJunc\_2 = QLabel("...")

self.xJunc\_unit1 = QLabel(self.xL\_unit.currentText())

self.xJunc\_unit2 = QLabel(self.xL\_unit.currentText())

self.progress\_label = QLabel("Available")

self.progress\_val = QLabel("...")

""""""

#################################

## Place widgets on the window ##

#################################

# layout = QVBoxLayout()

layout = QGridLayout()

layout.addWidget(Dopant\_label, 0, 0, 1, 1)

layout.addWidget(self.Dopant\_in, 0, 1, 1, 2)

layout.addWidget(Cb\_in, 1, 0, 1, 1)

layout.addWidget(self.Cb, 1, 1, 1, 2)

layout.addWidget(Cth\_in, 2, 0, 1, 1)

layout.addWidget(self.Cth, 2, 1, 1, 2)

layout.addWidget(xL\_in, 3, 0, 1, 1)

layout.addWidget(self.xL\_inUnit, 3, 1, 1, 1)

layout.addWidget(self.xL\_unit, 3, 2, 1, 1)

layout.addWidget(T0\_in, 0, 3, 1, 1)

layout.addWidget(self.T0, 0, 4, 1, 1)

layout.addWidget(T1\_in, 0, 5, 1, 1)

layout.addWidget(self.T1, 0, 6, 1, 1)

layout.addWidget(t0\_in, 1, 3, 1, 1)

layout.addWidget(self.t0, 1, 4, 1, 1)

layout.addWidget(t1\_in, 1, 5, 1, 1)

layout.addWidget(self.t1, 1, 6, 1, 1)

layout.addWidget(self.startSimulation, 2, 3, 1, 2)

layout.addWidget(Clear, 2, 5, 1, 2)

layout.addWidget(self.progress, 3, 3, 1, 2)

layout.addWidget(self.progress\_val, 3, 5, 1, 1)

layout.addWidget(self.progress\_label, 3, 6, 1, 1)

layout.addWidget(xJunc\_1\_label, 0, 7, 1, 2)

layout.addWidget(self.xJunc\_1, 1, 7, 1, 1)

layout.addWidget(self.xJunc\_unit1, 1, 8, 1, 1)

layout.addWidget(xJunc\_2\_label, 2, 7, 1, 2)

layout.addWidget(self.xJunc\_2, 3, 7, 1, 1)

layout.addWidget(self.xJunc\_unit2, 3, 8, 1, 1)

layout.addWidget(aboutB, 0, 9, 1, 2)

layout.addWidget(helpB, 1, 9, 1, 2)

layout.addWidget(exitB, 2, 9, 2, 2)

layout.addWidget(linearPlotCanvasTitle, 4, 0, 1, 5)

layout.addWidget(self.linearPlotCanvas, 5, 0, 1, 5)

layout.addWidget(self.linPC\_tb, 6, 0, 1, 5)

layout.addWidget(logPlotCanvasTitle, 4, 5, 1, 6)

layout.addWidget(self.logPlotCanvas, 5, 5, 1, 6)

layout.addWidget(self.logPC\_tb, 6, 5, 1, 6)

# Create the menu bar

# self.\_createMenuBar(layout)

# Set the layout on the application's window

widget.setLayout(layout)

# Set the central widget of the Window. Widget will expand to take up all the space in the window by default.

self.setCentralWidget(widget)

def xL\_unitConverter(self, xL\_inUnit): # Convert the unit of xL to cm

if self.xL\_unit.currentIndex():

return xL\_inUnit\*1e-7 # convert nm to cm

else:

return xL\_inUnit\*1e-4 # convert µm to cm

def xL\_unitConverter\_inv(self, xL): # Convert the unit of xL to µm or nm

if self.xL\_unit.currentIndex():

return xL\*1e7 # convert cm to nm

else:

return xL\*1e4 # convert cm to µm

def plot(self, Cp\_1:cProf, Cp\_2:cProf, Cth\_profile:cProf, x\_ax:list, xJunc\_1:float, xJunc\_2:float): # Plot the results

# Clear the previous plot (if any)

self.figure\_lin.clear()

self.figure\_log.clear()

# Get Cth from the Cth\_profile

Cth = Cth\_profile.arr[0]

# Convert the junction depths to specified unit

xJunc\_1\_inUnit = self.xL\_unitConverter\_inv(xJunc\_1)

xJunc\_2\_inUnit = self.xL\_unitConverter\_inv(xJunc\_2)

# Set the progress bar: 30% complete

self.updateProgress(40)

# Linear plot

ax1 = self.figure\_lin.add\_subplot()

ax1.set\_xlabel('Position ({})'.format(self.xL\_unit.currentText()))

ax1.set\_ylabel('Concentration (atoms/cm^3)')

ax1.plot(x\_ax, Cp\_1.arr, color='C0', label='Predep.', zorder=1)

ax1.plot(x\_ax, Cp\_2.arr, color='C1', label='Drive-in', zorder=0)

ax1.plot(x\_ax, Cth\_profile.arr, color='C2', label='Background Conc.', linestyle='dashed', zorder=2)

ax1.legend(loc='upper right', prop={'size': 6}) # Set the legend location

# Set the progress bar: 60% complete

self.updateProgress(70)

# Logarithmic plot

ax2 = self.figure\_log.add\_subplot()

ax2.set\_yscale('log') # Set the y-axis to log scale

ax2.set\_xlabel('Position ({})'.format(self.xL\_unit.currentText()))

ax2.set\_ylabel('Concentration (atoms/cm^3)')

ax2.plot(x\_ax, Cp\_1.arr, color='C0', label='Predep.', zorder=1)

ax2.plot(x\_ax, Cp\_2.arr, color='C1', label='Drive-in', zorder=0)

ax2.plot(x\_ax, Cth\_profile.arr, color='C2', label='Background Conc.', linestyle='dashed', zorder=2)

ax2.scatter(xJunc\_1\_inUnit, Cth, color='brown', marker='o', label='Junction Depth for Predep.', zorder=3)

ax2.scatter(xJunc\_2\_inUnit, Cth, color='black', marker='o', label='Junction Depth for Drive-in', zorder=3)

ax2.legend(loc='upper right', prop={'size': 6}) # Set the legend location

ax2.set\_ylim(bottom=1) # Set the y-axis limits

# Set the progress bar: 90% complete

self.updateProgress(90)

# Enable tight layout

self.figure\_lin.tight\_layout()

self.figure\_log.tight\_layout()

# Set the progress bar: 95% complete

self.updateProgress(95)

# Draw the plot

self.linearPlotCanvas.draw()

self.logPlotCanvas.draw()

def createDopantProfile(self): # Create a list of dopant profiles

imp\_Sb = Impurity(4.58, 3.88, 1e20)

imp\_As = Impurity(9.17, 3.99, 2e21)

imp\_B = Impurity(1.0, 3.5, 3e20)

imp\_P = Impurity(4.7, 3.68, 1e21)

self.dopantProfile\_list = [imp\_Sb, imp\_As, imp\_B, imp\_P]

return self.dopantProfile\_list

def updateDopantProfile(self, dopant\_idx:int): # Update the dopant profile to selected one

self.Dopant=self.dopantProfile\_list[dopant\_idx]

# self.Do, self.Ea, self.Co = self.Dopant.get\_attr()

def updateJuncDepth(self, label:QLabel, val:float, unit:QLabel): # Update the junction depths

val\_inUnit = self.xL\_unitConverter\_inv(val)

label.setText('{:.2f}'.format(val\_inUnit))

unit.setText(self.xL\_unit.currentText())

def updateProgressLabel(self, text:str): # Update the progress label and the GUI

self.progress\_label.setText(text)

QApplication.processEvents() # Process all pending events

def updateProgress(self, value:int, unit:str='%'): # Update the progress bar

self.progress.setValue(value)

self.progress\_val.setText("{}{}".format(value, unit))

QApplication.processEvents() # Process all pending events

def updateParameters(self): # Update input parameters

self.updateDopantProfile(self.Dopant\_in.currentIndex())

self.xL = self.xL\_unitConverter(self.xL\_inUnit.value())

kwarg = {

"Cb" : self.Cb.value(),

"Cth" : self.Cth.value(),

"Dopant" : self.Dopant,

"T0" : self.T0.value(),

"T1" : self.T1.value(),

"xL" : self.xL,

"t0" : self.t0.value(),

"t1" : self.t1.value(),

}

print("Updated parameters are: ", kwarg)

return kwarg

def kwargParser(self, \*\*kwargs): # Parse the keyword arguments

values = []

for \_, value in kwargs.items():

values.append(value)

return tuple(values)

def resetParameters(self): # Reset parameters to default values

self.Dopant\_in.setCurrentIndex(\_Dopant\_default)

self.Cb.setValue(\_Cb\_default)

self.Cth.setValue(\_Cth\_default)

self.T0.setValue(\_T0\_default)

self.T1.setValue(\_T1\_default)

self.xL\_unit.setCurrentIndex(\_xL\_unit\_default)

self.xL\_inUnit.setValue(\_xL\_default)

self.t0.setValue(\_t0\_default)

self.t1.setValue(\_t1\_default)

self.resetProgress()

def resetProgress(self): # Reset the progress bar

self.progress.setValue(\_prgrss\_default)

self.progress\_label.setText(\_prgrss\_lgnd\_default)

self.progress\_val.setText(\_prgrss\_val\_default)

def resetJuncDepth(self): # Reset the junction depths

self.xJunc\_1.setText(\_xJun1\_default)

self.xJunc\_2.setText(\_xJun2\_default)

def clearCanvas(self): # Reset everything on the canvas

# Clear the previous plot (if any)

self.figure\_lin.clear()

self.figure\_log.clear()

# Draw the blank figure

self.linearPlotCanvas.draw()

self.logPlotCanvas.draw()

def clear(self): # Reset everything to default values

# Progress bar notificaiton

self.updateProgressLabel("Clearing...")

self.updateProgress(0)

# Clear the transformation parameters

self.resetParameters()

self.updateProgress(33)

# Clear the junction depths

self.resetJuncDepth()

self.updateProgress(66)

# Clear canvas

self.clearCanvas()

self.updateProgress(100)

# Clear the progress bar

self.updateProgressLabel("Done!")

self.resetProgress()

print("---------CLEARED---------")

def toggle\_simulation(self): # Toggle the Simulate! button

if self.startSimulation.text() == "Simulate!":

self.startSimulation.setText("Terminate!")

self.startSimulation.setStyleSheet("color: #DC143C")

self.simulate()

self.startSimulation.setText("Simulate!")

self.startSimulation.setStyleSheet("color: #FFD700")

else:

self.startSimulation.setText("Terminating...")

self.startSimulation.setStyleSheet("color: #800000")

self.startSimulation.setEnabled(False)

self.terminate\_simulation()

self.startSimulation.setText("Simulate!")

self.startSimulation.setStyleSheet("color: #FFD700")

self.startSimulation.setEnabled(True)

def simulate(self): # Run the simulation

self.clearCanvas()

self.resetJuncDepth()

self.resetProgress()

self.updateProgressLabel("Preparing...")

# Set initial parameters

param = self.updateParameters()

Cb, Cth, Dopant, T0, T1, xD, t0, t1 = self.kwargParser(\*\*param)

# Create instances of the N\_simulation class

self.preDep = nSim(Dopant, T0)

self.driveIn = nSim(Dopant, T1)

# Set the progress bar: 33% complete

self.updateProgress(33)

# Set the simulation parameters

x\_i = int(xD/self.preDep.x\_step)+1 #x\_step is set to either 1e-8 cm (1 Angstrom) or 1e-7 (1 nm) inside nSim - constant for both simulations

t\_j0 = int(t0/self.preDep.t\_step)+1 #Number of time iterations for predep.

t\_j1 = int(t1/self.driveIn.t\_step)+1 #Number of time iterations for drive-in

# print("# of iterations for predep.: ", t\_j0, "\n# of iterations for drive-in: ", t\_j1)

# Set the progress bar: 66% complete

self.updateProgress(66)

# Create an instance of the C\_profiles class

C\_1 = cProf(x\_i=x\_i, Cb=Cb)

C\_2 = cProf(x\_i=x\_i, Cb=Cb)

Cth\_profile = cProf().create\_empty\_profile(x\_i=x\_i, Cb=Cth)

# Set the progress bar: 100% complete

self.updateProgress(100)

self.updateProgressLabel("Done!")

# Set the progress bar: 0% complete

self.updateProgress(0)

self.updateProgressLabel("Running Simulation: 1/2...")

# Run the simulation for predep.

Cp\_1, xjunc\_1 = self.preDep.lumerical\_on\_budget(C\_1, Cb, Cth, t\_j=t\_j0,

progressPercentageOutput=self.updateProgress,

progressOutput=self.updateProgressLabel,

)

self.updateProgressLabel("Predep. sim. is completed. Saving profile...")

# Print the junction depths:

self.updateJuncDepth(self.xJunc\_1, xjunc\_1, self.xJunc\_unit1)

# Set the progress bar: 0% complete

self.updateProgress(0)

self.updateProgressLabel("Running Simulation: 2/2...")

# Set the initial profile for the next simulation

C\_2.Cold = Cp\_1

# Run the simulation for drive-in

Cp\_2, xjunc\_2 = self.driveIn.lumerical\_on\_budget(C\_2, Cb, Cth, t\_j=t\_j1,

process=1,

progressPercentageOutput=self.updateProgress,

progressOutput=self.updateProgressLabel,

)

self.updateProgressLabel("Drive-in simulation completed. Saving profile...")

# Print the junction depths:

self.updateJuncDepth(self.xJunc\_2, xjunc\_2, self.xJunc\_unit2)

# Set the progress bar: 0% complete

self.updateProgress(0)

self.updateProgressLabel("Plotting the profiles...")

# Cut the first element of the profile to avoid the initial condition

Cp\_1.cut\_initial()

Cp\_2.cut\_initial()

Cth\_profile.cut\_initial()

# Construct the x-axis array

x\_step\_inUnit = self.xL\_unitConverter\_inv(self.preDep.x\_step) # Convert the x\_step to the specified unit

range\_arr = np.array(range(Cth\_profile.size()))

x\_ax = range\_arr\*x\_step\_inUnit # Construct the x-axis array in specified unit

# Set the progress bar: 10% complete

self.updateProgress(20)

# Plot the results

self.plot(Cp\_1, Cp\_2, Cth\_profile, x\_ax, xjunc\_1, xjunc\_2)

# Set the progress bar: 100% complete

self.updateProgress(100)

self.updateProgressLabel("Done!")

# # Clear the junk

# del self.preDep, self.driveIn

# del C\_1, C\_2, Cp\_1, Cp\_2, Cth\_profile

def terminate\_simulation(self): # Terminate the simulation - Not working properly

self.preDep.terminate()

self.driveIn.terminate()

def show\_popup(self, dialog\_text:str="You've found me!", dialog\_title:str="Easter Egg"): # Show a popup window

msg = QMessageBox()

msg.setWindowTitle(dialog\_title)

msg.setText(dialog\_text)

msg.setIcon(QMessageBox.Icon.Information)

msg.setStandardButtons(QMessageBox.StandardButton.Ok)

msg.exec()

# TODO: (Optional) Try to handle the boundary conditions in a more elegant way (indx 0 and negative indexes) for both processes

## numeric\_sim.py

###########################################

## DEVELOPER: Ibrahim Furkan Tezcan ##

## Version: 1.0 ##

## Date: 06.Jun.2024 ##

###########################################

import sys

import numpy as np

import matplotlib.pyplot as plt

# import argparse

# from numpy.lib.stride\_tricks import as\_strided as ast

class Impurity: # Dopant properties - TODO: CHECK "Co". It may depend on temperature.

"""

This class contains attributes and properties of an impurity object.

"""

def \_\_init\_\_(self, Do:float, Ea:float, Co:float):

self.Do = Do

self.Ea = Ea

self.Co = Co

def get\_attr(self) -> tuple:

return (self.Do, self.Ea, self.Co)

class \_C\_profile: #Concentration profile

"""

This class contains attributes and properties of a concentration profile object.\n

Extends C\_profiles class.

"""

def \_\_init\_\_(self, x\_i:int=1, Cb:float=0):

self.arr = np.ones(x\_i)\*Cb # Creates a blank concentration profile when initialized

def size(self) -> int:

return self.arr.size

def get\_val(self, i:int) -> float:

return self.arr[i]

def set\_val(self, val, i:int):

self.arr[i] = val

def get\_profile(self) -> np.array:

return self.arr

def cut\_initial(self):

self.arr = self.arr[1:]

class C\_profiles: # Conc. profiles at two different times

"""

This object contains two concentration profiles at two different times.\n

Concentration profile is a 1D array that contains the concentration values of a dopant at sampled positions.

"""

def \_\_init\_\_(self, x\_i:int=1, Cb:float=0):

self.Cnew = \_C\_profile(x\_i, Cb) # Creates a blank concentration profile for t\_j when initialized

self.Cold = \_C\_profile(x\_i, Cb) # Creates a blank concentration profile for t\_j-1 when initialized

def create\_empty\_profile(self, x\_i:int=1, Cb:float=0) -> \_C\_profile:

return \_C\_profile(x\_i, Cb)

def make\_profiles(self, Cnew:\_C\_profile, Cold:\_C\_profile):

self.Cnew = Cnew

self.Cold = Cold

def get\_profiles(self) -> tuple:

return (self.Cnew, self.Cold)

def set\_profile(self, C:\_C\_profile, i:int):

if i == 0:

self.Cnew = C

elif i == 1:

self.Cold = C

else:

print("Invalid index.")

def size(self) -> int:

return self.Cnew.size()

def update\_profiles(self, Cij:\_C\_profile):

if self.size() == Cij.size():

self.Cold = self.Cnew

self.Cnew = Cij

else:

print("Size mismatch. Profiles not updated.")

class N\_simulation: # Simulation class

"""

This class performs numerical simulations using difference equation derived from diffusion equation.

"""

def \_\_init\_\_(self, dopant:Impurity=None, T:int=900): #DONE!

# Termination flag

self.terminateFlag = False

# Impurity parameters

self.Ea = dopant.Ea #eV

self.D0 = dopant.Do #cm^2/s

self.C0 = dopant.Co #cm^-3

# Constants

self.Boltzmann = 8.617e-5 #eV/K

# self.x\_step = 1e-8 #cm (1 Angstrom)

self.x\_step = 1e-7 #cm (1 nm)

print("Position step: ", self.x\_step)

# Calculate diffusivity based on dopant

self.T = T + 273.15 #convert °C to K

self.D = self.diffusivity(self.T)

# Calculate time step for convergence

self.t\_step = (self.x\_step\*\*2) / (2\*self.D) #seconds

print("Time step: ", self.t\_step)

def diffusivity(self, T=900) -> float: #DONE!

"""

diffusivity(T=900)

Calculates diffusivity given the parameters.

Parameters:

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

T - Temperature for process (degree C) : int

"""

return (self.D0 \* np.exp(-self.Ea/(self.Boltzmann) \* (1/T)))

def lumerical\_on\_budget(self, C:C\_profiles, Cb:float=0, Cth:float=1e15, t\_j:int=1, process:bool=0, progressPercentageOutput=print, progressOutput=print) -> \_C\_profile: #DONE!

"""

lumerical\_on\_budget(C, Cb=0, Cth=100, t\_j=1, process=0, progressPercentageOutput=print, progressOutput=print)

Numerically calculates the concentration profile of given dopant.\n

Cb must be smaller than Cth.\n

If process is 0 (set by default), calculation will be done for predeposition.\n

If process is 1, calculation will be done for drive-in.

Parameters:

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

C - Concentration profile provided : \_C\_profile

Cb - Bottom concentration clip (atoms/cm^3) : float

Cth - Threshold (backgrnd) concentration (atoms/cm^3) : float

t\_j - Time iteration for process (seconds) : int

process - Selected process (predep./drive-in) : bool

progressPercentageOutput - Function to print the progress percentage : function

progressOutput - Function to print the progress : function

"""

xjunc=0

coef = self.D\*self.t\_step/(self.x\_step\*\*2)

if process == 0:

# j-1 iteration of time

for j in range(1, t\_j):

C.Cold.set\_val(self.C0, 0) #set initial condition and boundary condition

# progressOutput(100\*j/t\_j, "%", "completed.", end="\r")

progressPercentageOutput(int(100\*j/t\_j))

for i in range(1, C.Cold.size()-1):

Cij=C.Cold.get\_val(i) + coef \* (C.Cold.get\_val(i+1) - 2\*C.Cold.get\_val(i) + C.Cold.get\_val(i-1))

C.Cnew.set\_val(Cij, i)

C.update\_profiles(C.Cnew)

if self.terminateFlag:

progressOutput("Simulation is terminated.")

break

# Find junction depth

xjunc\_idx=0

min\_diff = float('inf')

for i in range(1, C.Cnew.size()-1):

current\_diff = abs(C.Cnew.get\_val(i) - Cth)

if current\_diff < min\_diff:

min\_diff = current\_diff

xjunc\_idx = i

xjunc = xjunc\_idx\*self.x\_step

elif process == 1:

xjunc\_idx=0

min\_diff = float('inf')

# j-1 iteration of time

for j in range(1, t\_j):

C.Cold.set\_val(Cb, -1) #set initial condition and boundary condition

# progressOutput(100\*j/t\_j, "%", "completed.", end="\r")

progressPercentageOutput(int(100\*j/t\_j))

for i in range(1, C.size()-1):

Cij=C.Cold.get\_val(i) + coef \* (C.Cold.get\_val(i+1) - 2\*C.Cold.get\_val(i) + C.Cold.get\_val(i-1))

C.Cnew.set\_val(Cij, i)

C.update\_profiles(C.Cnew)

if self.terminateFlag:

progressOutput("Simulation is terminated.")

break

# Find junction depth

xjunc\_idx=0

min\_diff = float('inf')

for i in range(1, C.Cnew.size()-1):

current\_diff = abs(C.Cnew.get\_val(i) - Cth)

if current\_diff < min\_diff:

min\_diff = current\_diff

xjunc\_idx = i

xjunc = xjunc\_idx\*self.x\_step

else:

progressOutput("Process not selected properly. Returning given profile.")

Cn = C.get\_profiles()[0]

return Cn, xjunc

def terminate(self):

self.terminateFlag = True

class plot: #TESTING...

"""

This class contains methods to plot the data.

"""

def \_\_init\_\_(self, Cp\_1:\_C\_profile=None, Cp\_2:\_C\_profile=None, Cth\_profile:\_C\_profile=None, xJunc\_1:float=0, xJunc\_2:float=0):

fig, (ax1, ax2) = plt.subplots(2)

fig.suptitle('Concentration Profile of the Dopant')

# Get Cth from the Cth\_profile

Cth = Cth\_profile.arr[0]

# Linear plot

ax1.plot(0,0)

ax1.set\_title('Linear Plot')

ax1.set\_xlabel('Position ({})'.format('cm'))

ax1.set\_ylabel('Concentration (atoms/cm^3)')

ax1.plot(Cp\_1.arr, color='blue', label='Predep.')

ax1.plot(Cp\_2.arr, color='red', label='Drive-in')

ax1.plot(Cth\_profile.arr, color='green', label='Background Conc.', linestyle='dashed')

ax1.legend(loc='upper right', prop={'size': 5}) # Set the legend location

# Logarithmic plot

ax2.plot(0,1)

ax2.set\_yscale('log')

ax2.set\_title('Logarithmic Plot') # Set the y-axis to log scale

ax2.set\_xlabel('Position ({})'.format('cm'))

ax2.set\_ylabel('Concentration (atoms/cm^3)')

ax2.plot(Cp\_1.arr, color='blue', label='Predep.')

ax2.plot(Cp\_2.arr, color='red', label='Drive-in')

ax2.plot(Cth\_profile.arr, color='green', label='Background Conc.', linestyle='dashed')

ax2.scatter(xJunc\_1, Cth, color='cyan', marker='o', label='Junction Depth for Predep.')

ax2.scatter(xJunc\_2, Cth, color='magenta', marker='o', label='Junction Depth for Drive-in')

ax2.legend(loc='upper right', prop={'size': 5}) # Set the legend location

ax2.set\_ylim(bottom=1)

def plot\_all(self, C:\_C\_profile=None):

plt.tight\_layout()

plt.show()

def createDopantProfile(dopant\_idx:int): # DONE!

imp\_Sb = Impurity(4.58, 3.88, 1e20)

imp\_As = Impurity(9.17, 3.99, 2e21)

imp\_B = Impurity(1.0, 3.5, 3e20)

imp\_P = Impurity(4.7, 3.68, 1e21)

dopantProfile\_list = [imp\_Sb, imp\_As, imp\_B, imp\_P]

return dopantProfile\_list[dopant\_idx]

def main(Cb, Cth, T0, T1, x, t0, t1): #TESTING...

# Create an instance of the Impurity class

impurity = createDopantProfile(1)

# Create instances of the N\_simulation class

preDep = N\_simulation(impurity, T0)

driveIn = N\_simulation(impurity, T1)

# Set the simulation parameters

x\_i = int(x/preDep.x\_step)+1 #x\_step is 1e-8 cm (1 Angstrom) - constant for both simulations

t\_j0 = int(t0/preDep.t\_step)+1 #Number of time iterations for predep.

t\_j1 = int(t1/driveIn.t\_step)+1 #Number of time iterations for drive-in

print("# of iterations for predep.: ", t\_j0, "\n# of iterations for drive-in: ", t\_j1)

# Create an instance of the C\_profiles class

C\_1 = C\_profiles(x\_i=x\_i, Cb=Cb)

C\_2 = C\_profiles(x\_i=x\_i, Cb=Cb)

Cth\_profile = C\_profiles().create\_empty\_profile(x\_i, Cth)

# Run the simulation

print("Running predep simulation...")

Cp\_1,xjunc\_1 = preDep.lumerical\_on\_budget(C\_1, Cb, Cth, t\_j=t\_j0) #predep.

print("Junction depth for predep.: ", xjunc\_1)

print("Predep simulation completed. Saving profile...")

C\_2.Cold = Cp\_1 #set the initial profile for the next simulation

print("Running drive-in simulation...")

Cp\_2,xjunc\_2 = driveIn.lumerical\_on\_budget(C\_2, Cb, Cth, t\_j=t\_j1, process=1) #drive-in

print("Drive-in simulation completed. Saving profile...")

print("Junction depth for drive-in: ", xjunc\_2)

# Cut the first element of the profile to avoid the initial condition

Cp\_1.cut\_initial()

Cp\_2.cut\_initial()

# Create an instance of the plot class

plotter = plot(Cp\_1=Cp\_1, Cp\_2=Cp\_2, Cth\_profile=Cth\_profile, xJunc\_1=xjunc\_1, xJunc\_2=xjunc\_2)

# Plot the results

plotter.plot\_all()

# Return the result

print("Exiting...")

# return Cp\_1, Cp\_2

if \_\_name\_\_ == "\_\_main\_\_": #RUN BOY RUN!

main(0, 1e15, 1000, 1150, 3e-4, 600, 30)

sys.exit()

# TODO: (Optional) Try to handle the boundary conditions in a more elegant way (indx 0) for both processes