Mystic User Guide

Version T-2022.03, March 2022

SYNOPSYS®

Contents

About This Guide

This document describes the functionality of the Synopsys® Mystic tool.

For additional information, see:

- The TCAD Sentaurus™ release notes, available on the Synopsys SolvNetPlus support site (see Accessing SolvNetPlus)
- Documentation available on the SolvNetPlus support site

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
Bold text	Identifies a selectable icon, button, menu, or tab. It also indicates the name of a field or an option.
`Courier font`	Identifies text that is displayed on the screen or that the user must type. It identifies the names of files, directories, paths, parameters, keywords, and variables.
Italicized text	Used for emphasis, the titles of books and journals, and non-English words. It also identifies components of an equation or a formula, a placeholder, or an identifier.

Customer Support

Customer support is available through the Synopsys SolvNetPlus support site and by contacting the Synopsys support center.

Accessing SolvNetPlus

The SolvNetPlus support site includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The site also gives you access to a wide range of Synopsys online services, which include downloading software, viewing documentation, and entering a call to the Support Center.

To access the SolvNetPlus site:

1. Go to https://solvnetplus.synopsys.com.
2. Enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register.)

Contacting Synopsys Support

If you have problems, questions, or suggestions, you can contact Synopsys support in the following ways:

- Go to the Synopsys Global Support Centers site on www.synopsys.com. There you can find email addresses and telephone numbers for Synopsys support centers throughout the world.
- Go to either the Synopsys SolvNetPlus site or the Synopsys Global Support Centers site and open a case (Synopsys user name and password required).

Contacting Your Local TCAD Support Team Directly

Send an email message to:

- support-tcad-us@synopsys.com from within North America and South America
- support-tcad-eu@synopsys.com from within Europe
- support-tcad-ap@synopsys.com from within Asia Pacific (China, Taiwan, Singapore, Malaysia, India, Australia)
- support-tcad-kr@synopsys.com from Korea
- support-tcad-jp@synopsys.com from Japan

1

Introduction to the Mystic Tool

This chapter introduces the Mystic extraction tool.

Overview of Mystic Functionality

The Mystic tool is a Python-based framework designed to facilitate the extraction of SPICE compact models, using the Synopsys PrimeSim™ HSPICE® tool. The process of extracting a SPICE model manually involves adjusting the parameter values of a SPICE compact model, running simulations with a SPICE engine, and comparing these simulation results to a set of target data curves, usually coming from measurement or TCAD data. This process is repeated as many times as necessary to obtain an accurate fit with the target data and requires a strong understanding of the underlying equations of the SPICE model and experience in setting up and running SPICE simulations.

The Mystic tool aims to limit barriers to using SPICE modeling by providing a flexible framework to assist with all parts of the procedure, from parsing input SPICE models, to automatically setting up, running, and parsing outputs from SPICE simulations based on provided target data, to optimizing the values of the SPICE model parameters using a range of sophisticated optimization algorithms.

Mystic Tool Flow

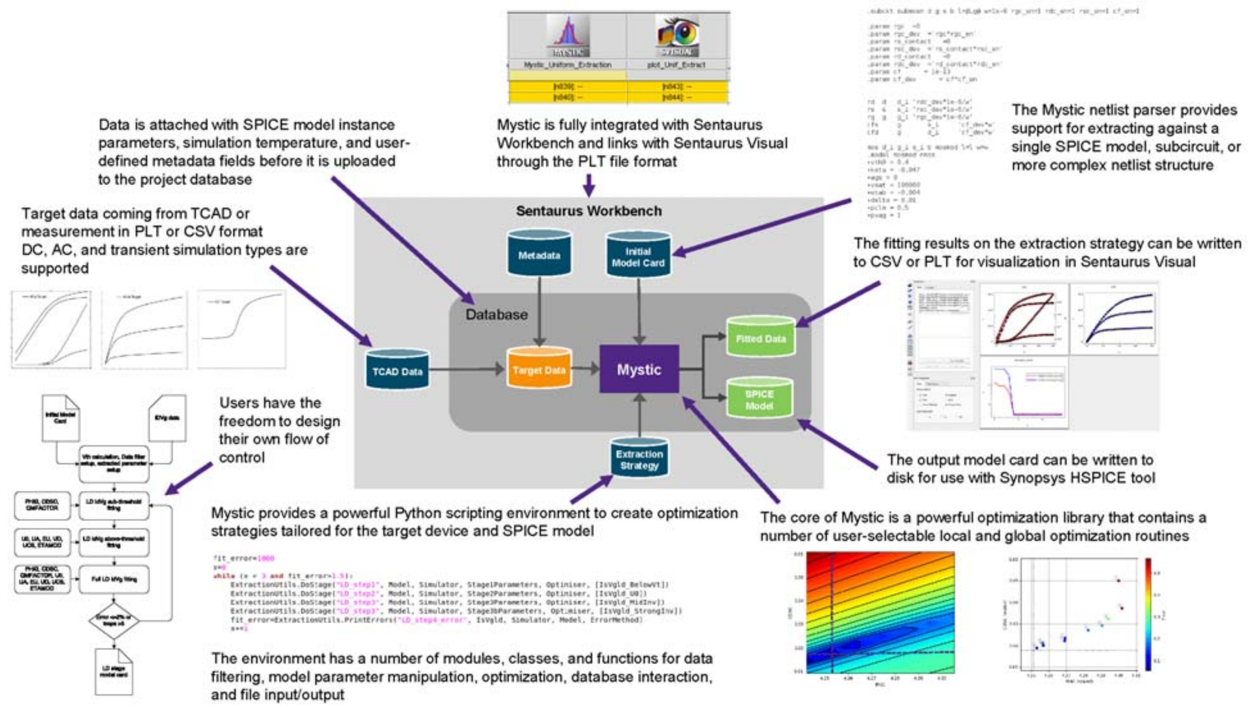
This section looks at the standard Mystic tool flow, including the input requirements, core components, and tool outputs. Figure 1 on page 9 shows a block-level overview of a standard Mystic tool flow.

The Mystic tool is contained within Sentaurus Workbench. This integration allows you to easily connect to TCAD Sentaurus projects that can provide target data for SPICE compact model extraction. It also helps with management of splits, such as different temperatures and process conditions, provides access to local compute clusters, and allows you to visualize SPICE model extraction results using Sentaurus Visual.

Underneath Sentaurus Workbench, the Mystic tool provides a local database for consolidating target data and simulation conditions, such as temperature and instance parameters, and transferring this information between sequential Mystic extraction stages. It can also be used to track SPICE model parameter values across stages and to integrate with the larger TCAD to SPICE tool flow.

This user guide describes each of the main components in the Mystic tool flow and gives you the tools to create your own SPICE model extraction flow in Mystic.

Figure 1 Overview of Mystic tool flow


Mystic is fully integrated with Sentaurus Workbench and links with Sentaurus Visual through the PLT file format
Data is attached with SPICE model instance parameters, simulation temperature, and user-defined metadata fields before it is uploaded to the project database
The Mystic netlist parser provides support for extracting against a single SPICE model, subcircuit, or more complex netlist structure
Target data coming from TCAD or measurement in PLT or CSV format DC, AC, and transient simulation types are supported
Sentaurus Workbench
Metadata
Initial Model Card
Database
TCAD Data
Target Data
Mystic
Fitted Data
SPICE Model
Extraction Strategy
The fitting results on the extraction strategy can be written to CSV or PLT for visualization in Sentaurus Visual
Users have the freedom to design their own flow of control
The output model card can be written to disk for use with Synopsys HSPICE tool
Mystic provides a powerful Python scripting environment to create optimization strategies tailored for the target device and SPICE model
The core of Mystic is a powerful optimization library that contains a number of user-selectable local and global optimization routines
The environment has a number of modules, classes, and functions for data filtering, model parameter manipulation, optimization, database interaction, and file input/output

```
.subckt submosn d g s b l=0.g0 w=1e-6 rgc_en=1 rdc_en=1 rsc_en=1 cf_en=1

.param rgc   =0
.param rgc_dev  = 'rgc*rgc_en'
.param rs_contact    =0
.param rsc_dev  = 'rs_contact*rsc_en'
.param rd_contact    =0
.param rdc_dev  = 'rd_contact*rdc_en'
.param cf        = 1e-13
.param cf_dev        = cf*cf_en

rd  d   d_i 'rdc_dev*1e-6/w'
rs  s   s_i 'rsc_dev*1e-6/w'
rg  g   g_i 'rgc_dev*1e-6/w'
cfs     g       s_i     'cf_dev*w'
cfd     g       d_i     'cf_dev*w'

mos d_i g_i s_i b mosmod l=l w=w
.model mosmod nmos
+vth0 = 0.4
+keta = -0.047
+ags = 0
+vsat = 100000
+etab = -0.004
+delta = 0.01
+pclm = 0.5
+pvag = 1
```

```
fit_error=1000
x=0
while (x < 3 and fit_error>1.5):
    ExtractionUtils.DoStage("LD_step1", Model, Simulator, Stage1Parameters, Optimiser, [IsVgld_BelowVt])
    ExtractionUtils.DoStage("LD_step2", Model, Simulator, Stage2Parameters, Optimiser, [IsVgld_U0])
    ExtractionUtils.DoStage("LD_step3", Model, Simulator, Stage3Parameters, Optimiser, [IsVgld_MidInv])
    ExtractionUtils.DoStage("LD_step3", Model, Simulator, Stage3bParameters, Optimiser, [IsVgld_StrongInv])
    fit_error=ExtractionUtils.PrintErrors("LD_step4_error", IsVgld, Simulator, Model, ErrorMethod)
    x+=1
```

Mystic API Documentation

The Mystic API documentation provides detailed information about all the Python commands and elements used in the Mystic tool.

You can access the API in the following location:

```
$STROOT/tcad/$STRELEASE/manuals/tcad_spice/mystic_api/index.html
```

Sentaurus Workbench API Documentation

From the graphical user interface of Sentaurus Workbench, you can access the Python API documentation by choosing **Help** > **Python API Documentation** > **SWB**.

TCAD Sentaurus Tutorial: Simulation Projects

The TCAD Sentaurus Tutorial provides various projects demonstrating the capabilities of TCAD to SPICE products.

To access the TCAD Sentaurus Tutorial:

1. Open Sentaurus Workbench by entering the following on the command line: `swb`
2. From the menu bar of Sentaurus Workbench, choose **Help** > **Training** or click T on the toolbar.

Alternatively, to access the TCAD Sentaurus Tutorial:

1. Go to the `$STROOT/tcad/current/Sentaurus_Training` directory.

   The `STROOT` environment variable indicates where the Synopsys TCAD distribution has been installed.
2. Open the `index.html` file in your browser.

Installing the Mystic Tool

For detailed setup and installation instructions, see the *TCAD Installation Notes*.

Running the Mystic Tool as a Standalone Tool

If you run the Mystic tool through Sentaurus Workbench, then the execution of the Mystic tool and the configuration of command-line options are handled automatically. For instructions on setting up the Mystic tool, refer to the *TCAD Installation Notes*.

After you have loaded the TCAD to SPICE tools into your environment, you can run Mystic as follows:

```
Mystic <filename> -d <string> -m <string> [other_options]
```

The mandatory file name specifies a Python file containing a structured set of commands that perform the SPICE model extraction. The `-d` option provides an optional path to the TCAD to SPICE database configuration file. If you want to use the database when running Mystic in standalone mode, then it must be configured separately. The `-m` option specifies a SPICE model compatible with PrimeSim HSPICE. For details, see SPICE Models on page 31.

Table 1 Command-line options for Mystic

Option (short form followed by long form)	**Description**
`<filename>`	Required. Mystic strategy to execute.
`-d <string>` `--db <string>`	Optional. Specifies the location of the configuration file of the TCAD to SPICE database.
`-ds <string>` `--dataset <string>`	Optional. Specifies the name of the dataset in the database in which to store the Mystic extraction results.
`-g` `--gui`	Optional. Starts the Mystic tool in interactive mode using the Mystic graphical user interface.
`-h` `--help`	Optional. Prints usage information and command-line options, and then exits.
`-l <string>` `--location <string>`	Optional. Specifies the full path of the PrimeSim HSPICE executable to use. This option overrides the version in the system path.
`-m <string>` `--model <string>`	Optional. Specifies the file name of the initial model to be loaded into the extraction strategy.
`-o <string>` `--optimizer <string>`	Optional. Specifies the optimization algorithm to use in the strategy (see Optimization Algorithms on page 36).
`-v` `--version`	Optional. Prints the Mystic version and then exits.
`-x <string>` `--spice-args <string>`	Optional. Specifies additional arguments to pass to the PrimeSim HSPICE backend simulation at runtime.

Extraction Environment

The Mystic extraction environment is a collection of specific Python modules, classes, and objects that are designed to help with SPICE compact model extraction (see Table 2). You have full freedom to assemble these in a strategy file to suit the specific needs of your extraction.

Table 2 Python modules, classes, and objects used in Mystic environment

Module, class, or object	**Description**
`ExtractionUtils`	This module provides utility functions to help with extraction. These functions usually combine `Model`, `Simulator`, `Optimizer`, and `SimData` objects.
`Corrector`	This module provides functionality for applying small analytic corrections on top of a SPICE model to perfectly match target data. This should be used when extreme accuracy is required.
`LinkedOptimizationParameter`	This class allows you to link one parameter value to another parameter value to ensure they remain consistent throughout the extraction.
`OptimizationParameter` or `OptParam`	This class defines SPICE model parameters to be extracted. These keep track of the current value of parameters throughout the extraction.
`mdb`	This object provides access to the TCAD to SPICE database where target data is stored. It can be used to load data into the environment at the start of the strategy and to store fitted models and data at the end.
`Model`	This object allows you to retrieve and set SPICE model parameter values inside the strategy.
`Optimizer`	This object provides an interface to the Mystic optimization library. It features a range of sophisticated algorithms for optimizing the values of SPICE model parameters.
`SimData`	This object provides a container format for target and simulation data. Usually, `SimData` objects can be loaded into the extraction strategy from the database, but they can also be initialized from scratch if necessary.
`Simulator`	This object represents the backend PrimeSim HSPICE simulator and can be used to set command-line options, global options, and the simulation temperature.

Objects that are passed into the environment are dependent on the command-line options passed to the Mystic tool. For example:

- If no initial model card is passed using the `-m` option, then the `Model` object in the environment will default to `None` and must be initialized within the strategy.
- If no database configuration file is passed using the `-d` option, then the `mdb` object will default to `None`.

- When running the Mystic tool through Sentaurus Workbench, the command-line options are handled automatically, so all objects in the extraction environment are configured correctly.

2

Uploading Target Data to Database

This chapter describes how to upload target data to the database.

Basics of Uploading Target Data

As mentioned in Mystic Tool Flow, the Mystic tool is most commonly run within Sentaurus Workbench as part of a larger TCAD to SPICE tool flow. These tool flows provide a local database for propagating information throughout the flow. The most common way of preparing target data for a Mystic extraction is to upload it to the database using a TCAD to SPICE Python tool instance located before the Mystic tool in the flow.

This methodology allows data to be preprocessed into the correct structure and tagged with useful metadata fields that the Mystic tool can utilize, such as simulation temperature and SPICE model instance parameters.

This chapter discusses the different file types that can be used as target data and how to upload these to the database using TCAD to SPICE Python. It also describes the required metadata fields that must be attached to the target data and how the Mystic tool uses these fields.

The Data Class

The `Data` class is a custom Python class that is inserted into the TCAD to SPICE Python environment at runtime. It provides functionality for reading and writing PLT and CSV files, as well as a link to the database that can be used to save and retrieve simulation data.

In the Mystic environment, the `Data` class is used to represent all target and simulation data. It provides functionality for filtering, resampling, applying transforms, and calculating errors between targets and simulations during optimization. The structure of `Data` objects is used to construct the backend PrimeSim HSPICE netlists that are created and simulated by the Mystic tool during the extraction process. Enough information must be provided through the `metadata` dictionary for the Mystic tool to replicate the simulation conditions of the target in the PrimeSim HSPICE tool. Figure 2 shows a representation of a standard I_d–V_g `Data` object.

Figure 2 Data format

```
bias : {'drain': 0.05, 'source': 0.0, 'substrate': 0.0}
instances : {'1': 2.5e-08, 'nfin': 1}
nodes : ['drain', 'gate', 'source', 'substrate']
temperature : 27.0
ftin : 8e-09
tspacer : 8e-09
workfn : 4.301
dtype : nMOS
eot : 1e-09
hfin : 4e-09


                    idrain  idrain-fit  errors
vgate
0.000000e+00 -4.654437e-09         nan     nan
2.500000e-02 -1.059546e-08         nan     nan
5.000000e-02 -2.393837e-08         nan     nan
7.500000e-02 -5.346387e-08         nan     nan
1.000000e-01 -1.171774e-07         nan     nan
```

The key–value pair dictionary at the top is the metadata. This contains static bias conditions, SPICE model instance parameters, an ordered list of SPICE model instance terminals, the simulation temperature, and some extra arbitrary fields that help to distinguish between `Data` objects when loading and filtering. These are particularly useful if you have a large design-of-experiments (DoE) of data at different process conditions.

Of these, the bias, temperature, nodes, and instances are required fields. Biases are extracted automatically from the PLT or CSV files if possible, but they can also be supplied explicitly if required (see Updating PLT Files on page 16).

The nodes list must match both the terminals of the model being extracted by the Mystic tool and the names of the biases in the `Data` object. In Figure 2, you can see that drain, source, and substrate are all present in the nodes list and `bias` dictionary (gate voltage does not have to be present in the `bias` dictionary as it is the independent variable being swept in the simulation). The simulation temperature should be entered into the `Data` class in kelvin. The Mystic tool converts this temperature to degree Celsius internally, as required by the SPICE model.

Entries in the instances dictionary must match the instance parameters in the SPICE model. This example comes from a BSIM-CMG SPICE model, which takes `l` and `nfin` instance parameters. This dictionary can remain empty if the default values should be used.

The core of the data is generally a single independent variable sweep (in this case, gate voltage) and a single dependent variable (in this case, drain current). Further dynamic input currents or voltages can be added as stimulus columns. This is mainly used for the transient simulation type and is discussed in Transient Simulation Extraction on page 45. The column headers determine the type of PrimeSim HSPICE simulation that the Mystic tool will produce for the data in question. Table 3 shows the different valid column notations.

Table 3 Valid column notations

Notation	As ivar | stimulus | bias	As dvar
`vcontact`	Voltage source, either stationary or a sweep, applied by using a `.data` statement in the netlist	A contact where a voltage response is measured
`icontact`	Current source, either stationary or a sweep, applied by using a `.data` statement in the netlist	A contact from which a current response is measured
`c(contact0,contact1)`	n/a	Capacitance, with a small signal applied to `contact0` and measured at `contact1`
`time`	Time values for transient simulations	n/a

When the `Data` object is created, two placeholder columns are added automatically for the PrimeSim HSPICE simulation data and a pointwise error calculation. Mystic populates these columns during the extraction.

For detailed information about the `Data` class, see the Mystic API documentation.

Updating PLT Files

The most common file format for Mystic extractions is the PLT file. PLT files are produced by Sentaurus Device simulations, which usually provide the target data for TCAD to SPICE flows. The following example demonstrates how to read in a Sentaurus Device PLT file and to upload its contents to the database for use with the Mystic tool:

```
ds = dbi.create_dataset(node_prj, "iv-data", clean=True)

metadata = {'temperature': 300,
            'instances': {'l': 27e-9, 'nfin':1},
            'nodes': ['drain','gate','source','substrate']}

Data.from_plt('idvgld.plt', metadata=metadata, upload_ds=ds)
```

First, a location in the database to store the data is created using `dbi.create_dataset`, where `dbi` is the Python interface to the TCAD to SPICE database. For more information, see the TCAD to SPICE Python API documentation.

Then, you create a metadata dictionary with the required information described in The Data Class on page 14. No bias dictionary is defined here because Sentaurus Device PLT files contain all of the static and dynamic biases for the simulation, so the bias dictionary is populated automatically when the file is read.

Finally, you use the `Data.from_plt` method to simultaneously read the information from the specified file and to upload it to the database in the created location. The PLT reader automatically recognizes some of the default Sentaurus Device notation and converts it to the format required by the Mystic tool. For example:

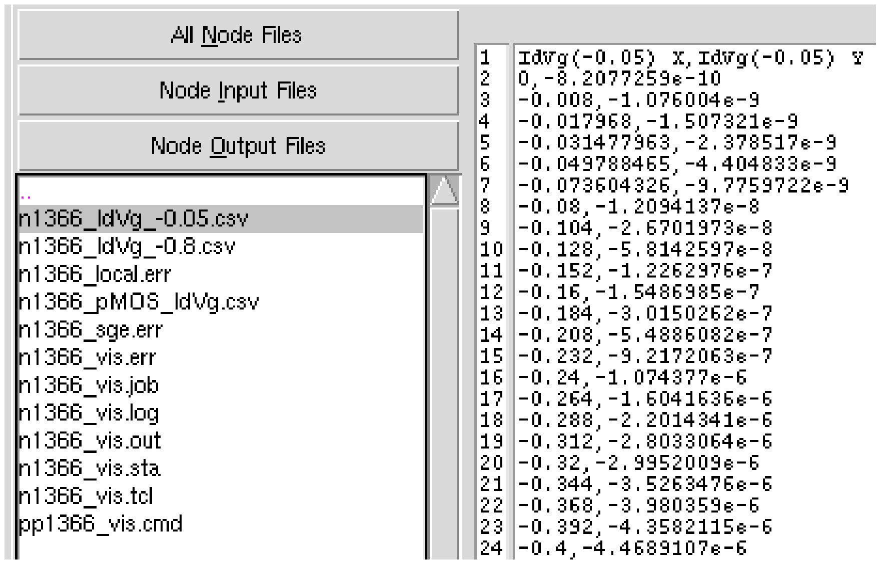
- `"contact OuterVoltage"` is converted to `"vcontact"`.
- `"contact TotalCurrent"` is converted to `"icontact"`.

Capacitances tend to be in the correct format already and do not need to be converted. If the column headers are not in the expected format, then they can also be converted explicitly by using a mapping (see Uploading CSV Files). The independent and dependent variables are detected automatically as the file is read, but they can also be specified explicitly with the ivar and dvar function arguments if required.

Uploading CSV Files

The CSV file is a very flexible format and allows a wider range of targets to be supported by the Mystic tool. When working with CSV files, you must provide some extra information to ensure the data labels are converted into the format required by the Mystic tool. Consider the example I_d–V_g CSV file shown in Figure 3.

Figure 3 Example of I_d–V_g CSV file



```
..
n1366_IdVg_-0.05.csv
n1366_IdVg_-0.8.csv
n1366_local.err
n1366_pMOS_IdVg.csv
n1366_sge.err
n1366_vis.err
n1366_vis.job
n1366_vis.log
n1366_vis.out
n1366_vis.sta
n1366_vis.tcl
pp1366_vis.cmd
```

```
IdVg(-0.05) X,IdVg(-0.05) Y
0,-8.2077259e-10
-0.008,-1.076004e-9
-0.017968,-1.507321e-9
-0.031477963,-2.378517e-9
-0.049788465,-4.404833e-9
-0.073604326,-9.7759722e-9
-0.08,-1.2094137e-8
-0.104,-2.6701973e-8
-0.128,-5.8142597e-8
-0.152,-1.2262976e-7
-0.16,-1.5486985e-7
-0.184,-3.0150262e-7
-0.208,-5.4886082e-7
-0.232,-9.2172063e-7
-0.24,-1.074377e-6
-0.264,-1.6041636e-6
-0.288,-2.2014341e-6
-0.312,-2.8033064e-6
-0.32,-2.9952009e-6
-0.344,-3.5263476e-6
-0.368,-3.980359e-6
-0.392,-4.3582115e-6
-0.4,-4.4689107e-6
```

In this case, the columns still contain the gate voltage and drain current, as in the previous example, but the column headers are not in a format that the Mystic tool recognizes. The following example shows how to upload this file:

```
ds = dbi.create_dataset(node_prj, "iv-data", clean=True)

metadata = {'temperature': 300,
            'instances': {'l': 27e-9, 'nfin':1},
            'nodes': ['drain','gate','source','substrate'],
            'bias': {'drain': -0.05,
                     'source': 0.0,
                     'substrate': 0.0}}

column_map = {'IdVg(-0.05) X': 'vgate',
              'IdVg(-0.05) Y': 'idrain'}

Data.from_csv('n1366_IdVg_-0.05.csv', metadata=metadata,
              column_map=column_map, upload_ds=ds)
```

In this example, note the following:

- There is now a `bias` dictionary in the metadata.
- This CSV file contains no static biases, so values for the drain, source, and substrate voltages must be added manually.
- A column mapping dictionary can be defined to instruct the `Data` class how to convert column headers into the correct format. This dictionary is passed into the `Data.from_csv` function as a `column_map` argument. After the data is stored in the database, it will have exactly the same formatting as PLT data and can be used by the Mystic tool to extract a SPICE model.

3

SPICE Model Extraction Strategy

This chapter describes aspects of the Mystic environment and demonstrates how it contributes to the construction of a SPICE model extraction strategy as a whole.

Database

The Mystic tool provides a link to the TCAD to SPICE database for consolidating target data and tracking the values of SPICE model parameters between sequential extraction stages. When running the Mystic tool through Sentaurus Workbench, the database connection is managed automatically with the path to the database configuration file being passed to the Mystic tool on the command line using the `-d` option.

If you run the Mystic tool standalone, then you must configure and manage the database connection manually. For information on how to do this, see Sentaurus Workbench API Documentation on page 9.

In the extraction environment, the `mdb` object is provided to store and retrieve information from the database. A standard Mystic extraction tool flow can contain up to five separate extraction stages, where each performs a subset of the fitting, allowing you to check successful parts of their extraction and to avoid continually rerunning the full strategy during development. The database ensures consistency of data and SPICE model parameter values between these stages as shown in Figure 4 on page 20.

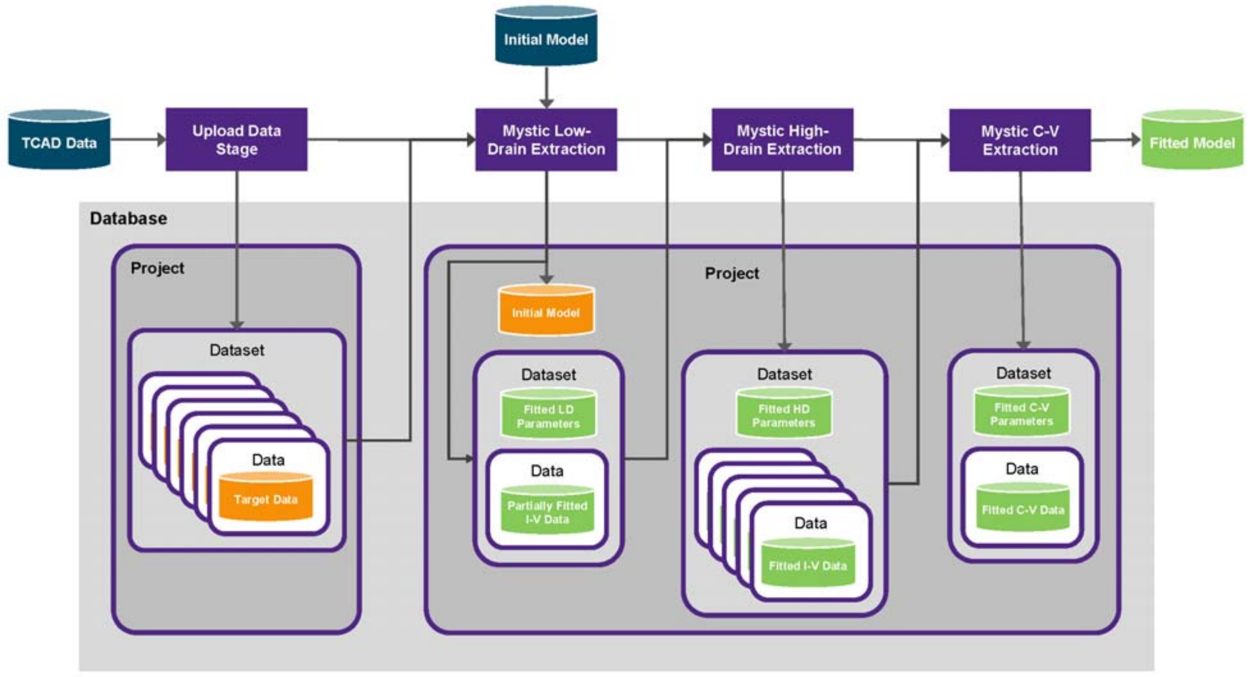
Figure 4 represents a three-stage SPICE model extraction for a MOSFET device with I–V and C–V target data. The upload data stage is a prerequisite for pushing target data into the database before the Mystic extraction. For details on this process, refer to the TCAD to SPICE module of the TCAD Sentaurus Tutorial (see TCAD Sentaurus Tutorial: Simulation Projects on page 10).

The database has a hierarchical structure with three levels: projects, datasets, and data objects. A project represents the full Mystic extraction strategy or one single node in Sentaurus Workbench. The main thing stored at this level is the initial model card, which can be added to the project at the start of the first extraction strategy as follows:

```
mdb.StoreInitialModel(Model)
```

where `Model` is the `Model` object passed in to the environment by the `-m` option. This model is then used as a point of reference in all downstream stages when setting up the `Model` object. Underneath the project is a list of datasets representing each individual extraction stage. Again, these are created by Sentaurus Workbench and passed to the Mystic tool using the `-ds` option.

Figure 4 Interaction with database in a standard Mystic model extraction



The datasets provide a container for extracted SPICE model parameters from each stage. Each dataset is linked to the previous one. In downstream stages, this makes it easy to retrieve the model parameters from the previous extraction stage using the `mdb.GetFitModel` method as follows:

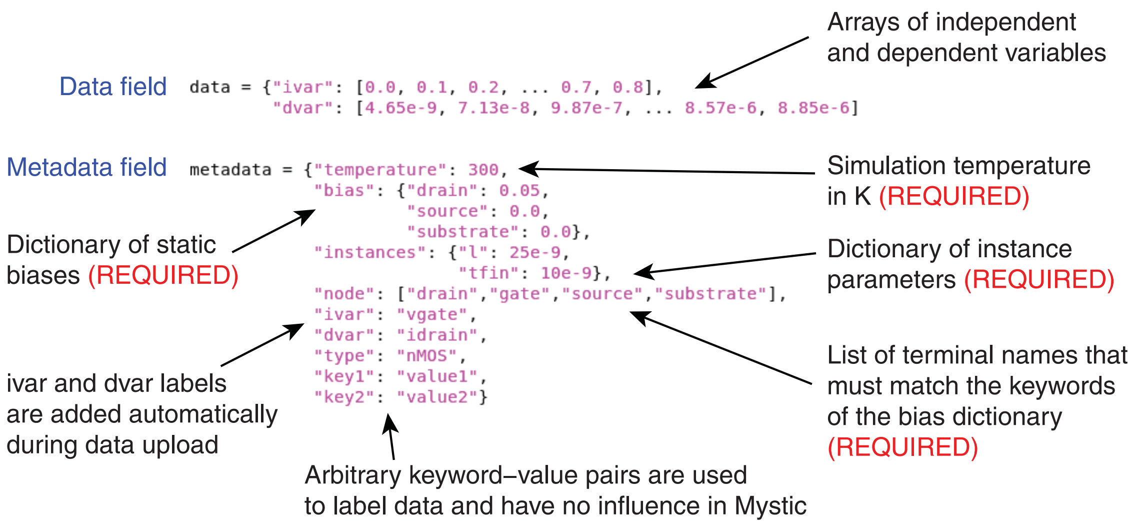
```
Model = mdb.GetFitModel()
```

Datasets also contain a list of data objects, where each object represents one piece of fitted target data. `Data` objects are split into two main fields:

- The *data field* is a dictionary of arrays containing independent and dependent variable data.
- The *metadata field* is a hierarchical dictionary of extra information, which can be either simulation conditions that will be used by the Mystic tool or simply unique tags that can be used to help query the data.

Figure 5 shows the standard Mystic metadata structure.

Figure 5 Structure of metadata in the database


Arrays of independent and dependent variables
Data field
Metadata field
Simulation temperature in K (REQUIRED)
Dictionary of static biases (REQUIRED)
Dictionary of instance parameters (REQUIRED)
ivar and dvar labels are added automatically during data upload
List of terminal names that must match the keywords of the bias dictionary (REQUIRED)
Arbitrary keyword–value pairs are used to label data and have no influence in Mystic

```
data = {"ivar": [0.0, 0.1, 0.2, ... 0.7, 0.8],
        "dvar": [4.65e-9, 7.13e-8, 9.87e-7, ... 8.57e-6, 8.85e-6]

metadata = {"temperature": 300,
            "bias": {"drain": 0.05,
                     "source": 0.0,
                     "substrate": 0.0},
            "instances": {"l": 25e-9,
                          "tfin": 10e-9},
            "node": ["drain","gate","source","substrate"],
            "ivar": "vgate",
            "dvar": "idrain",
            "type": "nMOS",
            "key1": "value1",
            "key2": "value2"}
```

It is important to understand the metadata requirements of the Mystic tool, so you can define it correctly when uploading data and know which fields are available when loading data back from the database.

In the Mystic strategy, data is retrieved from the database using the `mdb.Load` and `mdb.GetFitData` methods:

- `mdb.Load` is used to load target data from the upload data project in the first stage of an extraction strategy.
- `mdb.GetFitData` is usually used to load partially fitted data in downstream stages of the Mystic extraction.

Both methods take arbitrary filter arguments that select data from the database based on its project, data, and metadata fields. For example, the low-drain I_d–V_g data shown in Figure 5 could be loaded by using:

```
IdVgld = mdb.Load(temperature=300, bias__drain=0.05,
                  instances__l=25e-9, ivar="vgate", dvar="idrain")
```

Note:

To match hierarchical fields such as biases and instance parameters, a double underscore is used to separate the levels of hierarchy.

All data entries matching the filter criteria are included in the return. The `mdb.Load` and `mdb.GetFitData` methods return `SimData` objects. For more information about these, see Target Data on page 22.

Fitted data can be stored back in the database at the end of an extraction strategy using the `mdb.StoreFitData` method.

Target Data

The Mystic tool can accept target data from any PLT-like or CSV-like file format. As described in Chapter 2, Uploading Target Data to Database, this data is uploaded to the Mystic database in a separate upload data tool instance at the start of the TCAD to SPICE flow and is attached with metadata such as simulation temperature and instance parameters. This provides the Mystic tool with enough information to set up backend PrimeSim HSPICE simulations without further input from users.

Note:

Only I–V, C–V, transient, and RF scattering parameter data types are supported for extraction.

As previously mentioned, data can be retrieved from the database in the extraction strategy using the `mdb` object. The `mdb.Load` and `mdb.GetFitData` methods both return `SimData` objects, which are the data containers used in the Mystic tool. These consist of one or more datasets that usually have four columns:

- An independent variable
- A target-dependent variable (TCAD or measurement)
- A fitted-dependent variable (PrimeSim HSPICE simulation data)
- A pointwise error

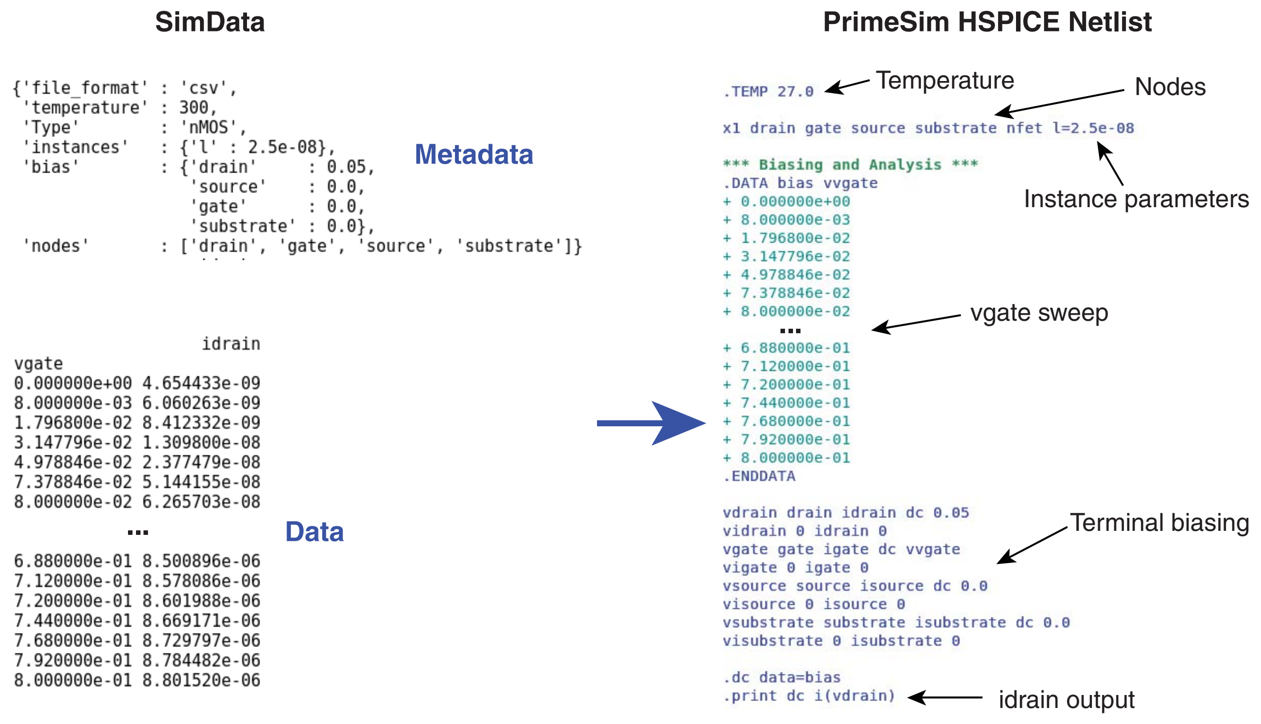
When the Mystic tool creates backend PrimeSim HSPICE simulations, `SimData` objects are used to automatically create biasing, stimulus, and output commands in the PrimeSim HSPICE netlists that match the simulation conditions of the targets as shown in Figure 6.

This ensures consistency between target data and fitted data when calculating errors and removes the burden from users of having to define and manage this information themselves. From an optimization perspective, the most important column of the `SimData` objects is the pointwise error column, which is passed back to the `Optimizer` object after every evaluation and dictates how it adjusts the parameters. Table 4 lists the options of the pointwise error method.

The default option is `rel`, but you can change it at either a global level or on an object-by-object basis as follows:

```
SimData.SetOptions(error_method="abs")
```

Figure 6 Example of how the Mystic tool transforms SimData objects into PrimeSim HSPICE netlist commands



```
{'file_format' : 'csv',
 'temperature' : 300,
 'Type'        : 'nMOS',
 'instances'   : {'l' : 2.5e-08},
 'bias'        : {'drain'     : 0.05,
                  'source'    : 0.0,
                  'gate'      : 0.0,
                  'substrate' : 0.0},
 'nodes'       : ['drain', 'gate', 'source', 'substrate']}
```

```
                   idrain
vgate
0.000000e+00 4.654433e-09
8.000000e-03 6.060263e-09
1.796800e-02 8.412332e-09
3.147796e-02 1.309800e-08
4.978846e-02 2.377479e-08
7.378846e-02 5.144155e-08
8.000000e-02 6.265703e-08
     ...
6.880000e-01 8.500896e-06
7.120000e-01 8.578086e-06
7.200000e-01 8.601988e-06
7.440000e-01 8.669171e-06
7.680000e-01 8.729797e-06
7.920000e-01 8.784482e-06
8.000000e-01 8.801520e-06
```

```
.TEMP 27.0

x1 drain gate source substrate nfet l=2.5e-08

*** Biasing and Analysis ***
.DATA bias vvgate
+ 0.000000e+00
+ 8.000000e-03
+ 1.796800e-02
+ 3.147796e-02
+ 4.978846e-02
+ 7.378846e-02
+ 8.000000e-02
      ...
+ 6.880000e-01
+ 7.120000e-01
+ 7.200000e-01
+ 7.440000e-01
+ 7.680000e-01
+ 7.920000e-01
+ 8.000000e-01
.ENDDATA

vdrain drain idrain dc 0.05
vidrain 0 idrain 0
vgate gate igate dc vvgate
vigate 0 igate 0
vsource source isource dc 0.0
visource 0 isource 0
vsubstrate substrate isubstrate dc 0.0
visubstrate 0 isubstrate 0

.dc data=bias
.print dc i(vdrain)
```

Table 4 Options of pointwise error method

Option	Description	Equation (f=fit, t=target)
`rel`	(Default) Relative	$\left\lvert\frac{f-t}{t}\right\rvert$
`abs`	Absolute	$\lvert f-t\rvert$
`log_abs`	Absolute to the log10 data	$\left\lvert\log_t\left(\frac{f}{t}\right)\right\rvert$
`log_rel`	Relative to the log10 data	$\left\lvert\log_{10}\left(\frac{f}{t}\right)\right\rvert$
`square`	Square	$(f-t)^2$

Multilevel Data Structures

When loading data from the database, multiple datasets of the same simulation type but varying metadata fields can be combined to significantly reduce the runtime of a Mystic extraction. This can be done by adding `combine=True` to any `mdb.Load` command. Consider the following example where there are two target I_d–V_g simulations with drain biases of 0.05 V and 0.8 V, respectively:

```
IdVg = mdb.Load(ivar="vgate", dvar="idrain",
                project=targetProjectName)
IdVg = mdb.Load(ivar="vgate", dvar="idrain",
                project=targetProjectName, combine=True)
```

The first line returns one `SimData` object with two underlying datasets separated by drain bias. The second line returns one `SimData` object containing one underlying dataset with a two-level index of drain bias and gate bias. At runtime, the combined dataset will produce only one backend PrimeSim HSPICE netlist per evaluation instead of two, which results in a large runtime reduction.

When the loaded data has more than one varying bias field, such as bulk device simulation data where there are multiple substrate biases and multiple drain biases, that data is combined based on the first varying field that is discovered. Alternatively, a combination field can be specified explicitly using the `extra_index` argument as follows:

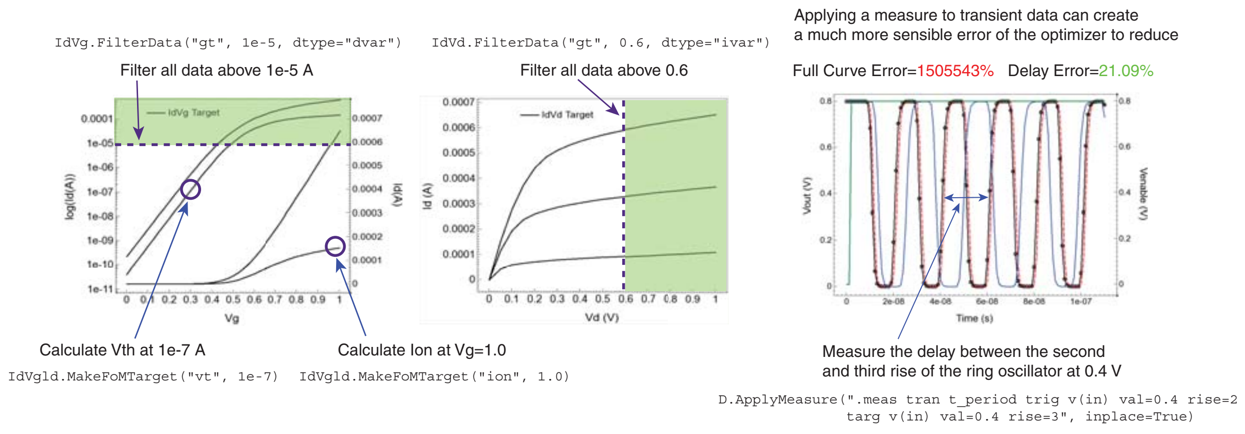
```
IdVg = mdb.Load(ivar="vgate", dvar="idrain",
                project=targetProjectName, combine=True,
                extra_index="vdrain")
IdVg = mdb.Load(ivar="vgate", dvar="idrain",
                project=targetProjectName, combine=True,
                extra_index="vsubstrate")
```

For the best performance increase, data should be combined based on the field that has the most varying values, that is, if there are two drain biases and three substrate biases, then specifying `extra_index="vsubstrate"` would be most reasonable.

Filtering

In SPICE model extraction, it is often necessary to target the fitting of a parameter or set of parameters to a specific region of the device-operating characteristics. For example, when fitting the workfunction, only the low-drain subthreshold characteristics are relevant. Fitting parameters against too broad a range of target data can overcomplicate the optimization problem and result in poor optimization performance. To help with this, `SimData` provides a range of Python methods for filtering, resampling, measuring, and calculating device figures of merit as shown in Figure 7.

Figure 7 Common data types supported by the Mystic tool



```
IdVg.FilterData("gt", 1e-5, dtype="dvar")
IdVd.FilterData("gt", 0.6, dtype="ivar")
IdVgld.MakeFoMTarget("vt", 1e-7)
IdVgld.MakeFoMTarget("ion", 1.0)
D.ApplyMeasure(".meas tran t_period trig v(in) val=0.4 rise=2
    targ v(in) val=0.4 rise=3", inplace=True)
```

Filters can be applied to any column of the data structure or any bias metadata value using the `SimData.FilterData` method. Any of the standard mathematical operators shown in Table 5 can be applied.

Table 5 Standard mathematical operators

Operator	Description
`ge`	Greater than or equal to
`gt`	Greater than
`lt`	Less than
`le`	Less than or equal to
`eq`	Equal to
`neq`	Not equal to

The following example illustrates how to filter data in a set of two I_d–V_g curves into manageable segments:

```
# Load data from database.
IdVg = mdb.Load(ivar="vgate", dvar="idrain", project="upload-data")

# Split the data by drain bias.
IdVgld = IdVg.FilterData("eq", 0.05, dtype="bias.drain")
IdVghd = IdVg.FilterData("eq", 0.05, dtype="bias.drain")

# Split the low-drain data into below and above Vt.
IdVgld_BelowVt = IdVgld.FilterData("le", 0.3, dtype="ivar")
```

```
IdVgld_AboveVt = IdVgld.FilterData("ge", 0.4, dtype="ivar")

# Split the high-drain data into below and above Vt.
IdVghd_BelowVt = IdVghd.FilterData("le", 0.3, dtype="ivar")
IdVghd_AboveVt = IdVghd.FilterData("ge", 0.4, dtype="ivar")
```

Figure of Merit Transforms

Each of these segments then has its own role in the extraction, matched to SPICE model parameters that influence the behavior in that region.

The `SimData` object also provides a set of figures of merit calculation methods specific to the behavior of MOSFET devices (see Table 6).

Table 6 Figures of merit calculation methods

Method	Description
`Derivative`	Derivative of the data.
`Gm`	Transconductance of the device at all bias points.
`GmMax`	Maximum transconductance of the device.
`Ioff`	Off-current of the device, which is calculated at $V_g = 0.0$ by default.
`Ion`	On-current of the device, which is calculated at the maximum V_g point of the data by default.
`Rout`	Output resistance of the data.
`SubSlope`	Subthreshold slope of the device measured in mV/decade. The default minimum current is the minimum current of the data. The default maximum current is 1 decade above the minimum current.
`Vt`	Threshold voltage of device, which should be calculated on low-drain I_d–V_g data. The default current criteria is 1e-7 A.
`VtGmMax`	Threshold voltage based on the maximum transconductance of the device. The calculated value is the x-intercept of the gradient of the maximum transconductance.

Each figure of merit (FoM) has its own method. For example:

```
Vt = IdVgld.VtGmmax()
```

In this case, `Vt` is a single float that can be used as a threshold for filtering. Figures of merit that require an argument, for example, V_t current criteria, are initialized with sensible defaults. These can be changed using the `SimData.SetFoMCriteria` method. It is also possible to create an extraction target from a figure of merit using the `SimData.MakeFoMTarget` method as follows:

```
Vt = IdVgld.MakeFoMTarget("vt", 1e-7)
```

In this case, `Vt` is a new `SimData` object that can be used as usual in extraction steps. The data of the root object (`IdVgld`) is used to create and run the PrimeSim HSPICE simulation, then the FoM transform is applied to both the target and fit columns as a postprocessing task. When used in an extraction step, the single error between the target and fitted FoM values will be used instead of the pointwise error between the curves.

In addition, you can define your own FoM transforms and apply them to any `SimData` object. The transform can be applied to all data points, such as a log transform or multiplier, or it can aggregate the data to a single value, such as threshold voltage.

The following example shows how to apply a log10 transform to all data:

```
import numpy as np

IdVg = mdb.Load(ivar="vgate", dvar="idrain")

def log10(data);
return np.log10(data)

IdVg = IdVg.MakeFoMTarget(log10)

IdVgld = IdVg.FilterData("eq", 0.05, "vdrain")
IdVgld_BelowVt = IdVgld("lt", 0.3, "ivar")
IdVgld_AboveVt = IdVgld("gt", 0.4, "ivar")

IdVghd = IdVg.FilterData("eq", 0.05, "vdrain")
IdVghd_BelowVt = IdVghd("lt", 0.3, "ivar")
IdVghd_AboveVt = IdVghd("gt", 0.4, "ivar")
```

Your function should accept a data argument that is the internal `Data` object to which the transform is applied. For a detailed description of this class, see Sentaurus Workbench API Documentation on page 9. As the transform is applied to the root I_d–V_g object, all objects derived from that using `SimData.FilterData` will also have the log transform applied to them.

The following example shows how to apply a mean aggregation transform:

```
IdVg = mdb.Load(ivar="vgate", dvar="idrain")

def mean(data, col="target"):
   if col == "target":
      return data._values.mean()
   elif col == "fit":
      return data._fit_values.mean()

IdVg_mean = IdVg.MakeFoMTarget(mean, add_as_fom=True)
```

The `add_as_fom` argument signals that the function is an aggregation that should be added to the `Data` object API. In this scenario, the function must accept a column argument that selects whether the aggregation is applied to the target data or fit data.

FoM transforms can also be chained together where appropriate. Consider this example that extracts against the log10 transform of the device transconductance:

```
IdVg = mdb.Load(ivar="vgate", dvar="idrain")

IdVgld = IdVg.FilterData("eq", 0.05, "vdrain")
Gmld = IdVgld.MakeFoMTarget("gm")

def log10(data):
   return np.log10(data)

Gmld_log = Gmld.MakeFoMTarget(log)
```

Data Masks

When filtering a `SimData` object in the Mystic tool using the `SimData.FilterData` method, a copy of the object is created with the filter criteria applied. The copy can be replaced with a mask that retains a reference to the root object. When a filtered object is used in a `ExtractionUtils.PrintErrors` or `ExtractionUtils.DoStage` call, the root object is simulated instead of the filtered object. This means that when two or more `SimData` objects with the same root object are passed in to a `ExtractionUtils.DoStage` optimization, only one PrimeSim HSPICE simulation is required per evaluation. This significantly reduces the runtime of a Mystic extraction.

Data masks can be activated globally through the `Data` class as follows:

```
from libfromage.data import Data
Data.set_options(data_mask=True)
```

They can also be activated on a case-by-case basis using the `mask` argument of `FilterData`:

```
IdVg = mdb.Load(ivar="vgate", dvar="idrain")

IdVgld = IdVg.FilterData("eq", 0.05, "vdrain", mask=True)
IdVgld_BelowVt = IdVgld.FilterData("lt", 0.3, "ivar", mask=True)
IdVgld_AboveVt = IdVgld.FilterData("gt", 0.4, "ivar", mask=True)

IdVghd = IdVg.FilterData("eq", 0.05, "vdrain", mask=True)
IdVghd_BelowVt = IdVghd.FilterData("lt", 0.3, "ivar", mask=True)
IdVghd_AboveVt = IdVghd.FilterData("gt", 0.4, "ivar", mask=True)
```

Masks can also be applied when creating FoM targets using `MakeFoMTarget`. As with filters, the root object is always simulated, meaning multiple FoMs can be evaluated in a single PrimeSim HSPICE simulation.

Consider the following example:

```
IdVg = mdb.Load(ivar="vgate", dvar="idrain")

IdVgld = IdVg.FilterData("eq", 0.05, "vdrain")
IdVgld_AboveVt = IdVgld.FilterData("gt", 0.4, "ivar", mask=True)

Ioff       = IdVgld.MakeFoMTarget("ioff")
VtGmMax    = IdVgld.MakeFoMTarget("vtgmmax")
Gm_AboveVt = IdVgld_AboveVt.MakeFoMTarget("gm")
Ion        = IdVgld.MakeFoMTarget("ion")

phig = OptParam('PHIG', 4.4,   4.2,   4.6,   1.0)
cdsc = OptParam('CDSC', 0.001, 0.0,   10.0,  1.0)
u0   = OptParam('U0',   0.02,  0.004, 1.0,   1.0)
ua   = OptParam('UA',   0.3,   0.0,   100.0, 1.0)
eu   = OptParam('EU',   2.5,   0.1,   10.0,  1.0)

params = [phig,cdsc,u0,ua,eu]

ExtractionUtils.DoStage("FOM", Model, Simulator, params, Optimizer,
                        [Ioff,VtGmMax,Gm_AboveVt,Ion])
```

This example is trying to extract against four FoMs that all come from the same root target data. Previously, four PrimeSim HSPICE simulations would have been required per evaluation. Using masks, only one simulation is required.

Generic PrimeSim HSPICE .MEASURE Commands

More generic transforms can also be applied to `SimData` objects using the PrimeSim HSPICE `.MEASURE` syntax. PrimeSim HSPICE measures can be used in the same way as figures of merit to calculate the electrical characteristics of a simulation, but they give you more freedom to define and apply your own transforms. In the Mystic tool, these can be applied using the `SimData.ApplyMeasure` method. When the data is used in a parameter optimization, the `.MEASURE` command is first post-applied to the target data and then added to each runtime netlist to obtain a fitted value from the PrimeSim HSPICE simulation. The root mean square (RMS) error of all measured values is then minimized by the optimizer. The following is an example of calculating the V_t and I_{on} of an I_d–V_g target using `.MEASURE` commands:

```
# Create a Vt target.
Vt = IdVg.ApplyMeasure('.MEASURE DC vt FIND v(g) when i(vd)=-1e-7')

# Create an Ion target.
Ion = IdVg.ApplyMeasure('.MEASURE DC ion MAX i(vd)')
```

As with `MakeFoMTarget`, `SimData` objects are returned from `SimData.ApplyMeasure` and can be used as usual throughout the extraction. At any point in the extraction, the contents of a `SimData` object can be written to disk as either a PLT or CSV file, using the `SimData.WritePLTFile` and `SimData.WriteCSVFile` methods. The created files can then be plotted using Sentaurus Visual.

All bias terminal names referenced in the `.MEASURE` command must match terminal names attached to the data. For example, if a data object has the terminal names `drain`, `gate`, `source`, and `substrate`, then the following `.MEASURE` command will fail at runtime because the terminal name `d` does not exist:

```
Ioff = IdVgld.ApplyMeasure(".MEAS DC ioff MIN i(vd)")
```

The following would correctly measure I_{off} for the low-drain I_d–V_g curve:

```
Ioff = IdVgld.ApplyMeasure(".MEAS DC ioff MIN i(vdrain)")
```

You can test `.MEASURE` commands before using them in an optimization by utilizing the `TestMeasure()` method as follows:

```
Ioff = IdVgld.ApplyMeasure(".MEAS DC ioff MIN i(vdrain)")
Ioff.TestMeasure(Simulator)
```

Multiple `.MEASURE` commands can also be applied to the same `SimData` object using the `inplace` argument. For example, I_{off} and V_t of a low-drain I_d–V_g simulation can be optimized simultaneously by specifying:

```
IoffVt = IdVgld.ApplyMeasure(".MEAS DC ioff MIN i(vdrain)")
IoffVt.ApplyMeasure(".MEAS DC vt FIND v(gate) WHEN i(vdrain)=1e-7",
                    inplace=True)
```

The error in both measured quantities will be passed to the optimizer during optimization. You can apply `.MEASURE` commands to DC, AC, or transient data, provided they are valid based on the simulation conditions of the data. For more information about how to construct a `.MEASURE` command, see the *PrimeSim™ Continuum Reference Manual: Commands and Control Options*.

Bias Extension

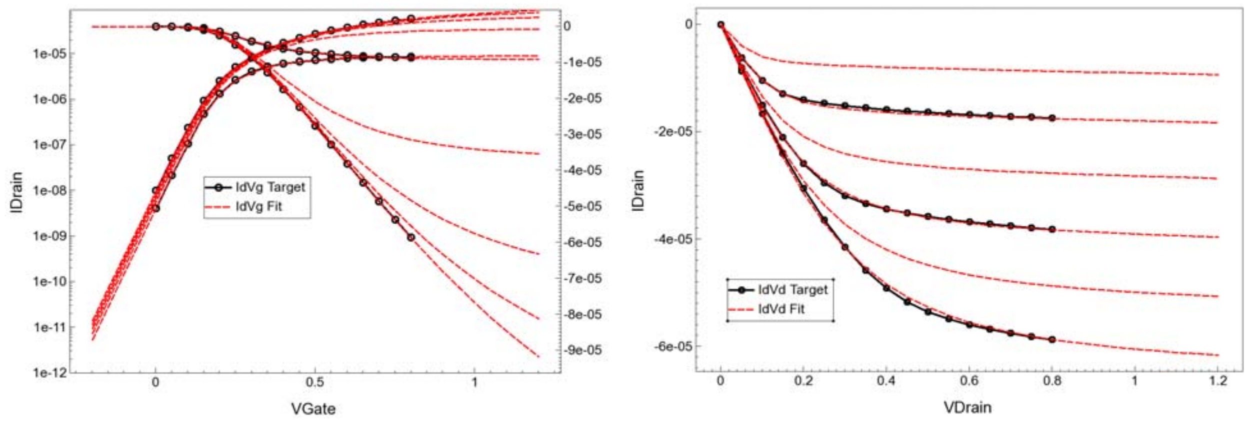
After model extraction, you might want to simulate your model at bias conditions outside of the extraction range. This can be done using `SimData.ExtendBias`. Consider an extraction where the target data consists of two I_d–V_g curves at V_d=0.05 V and 0.8 V, respectively, with a range of V_g=0.0 V – 0.8 V, and three I_d–V_d curves at V_g=0.4 V, 0.6 V, and 0.8 V, with a range of V_d=0.0 V – 0.8 V:

```
# Extend the bias ranges of the data
IdVg_ext = IdVg.ExtendBias(ivar_min=-0.2, ivar_max=1.2, spacing=0.05,
                           vdrain=[0.2,0.4,0.6])
```

```
IdVd_ext = IdVd.ExtendBias(ivar_max=1.2, spacing=0.05,
                           vgate=[0.3,0.5,0.7])

# Simulate the model at the extended ranges
ExtractionUtils.PrintErrors("idvg-ext", IdVg_ext, Simulator, Model,
                            "rmsd")
ExtractionUtils.PrintErrors("idvd-ext", IdVd_ext, Simulator, Model,
                            "rmsd")

# Write the extended data to a PLT file
IdVg_ext.WritePLTFile('idvg-ext')
IdVd_ext.WritePLTFile('idvg-ext')
```

As can be seen in Figure 8, the data range has extended to the bounds specified in `SimData.ExtendBias` and new data curves have been simulated for the intermediate drain and gate biases.

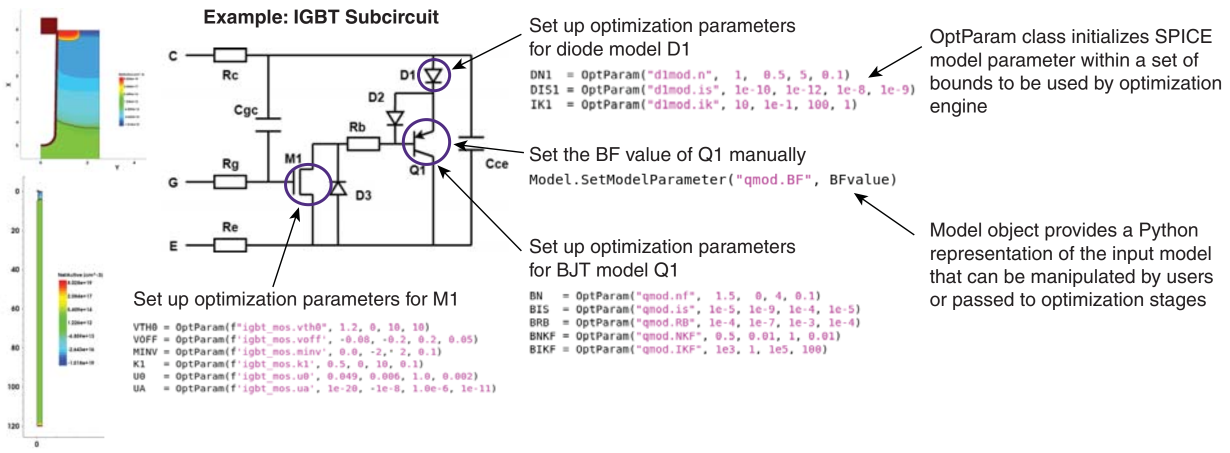
Figure 8 Target and fitted I–V data after the bias extension



SPICE Models

The Mystic tool supports any netlist that is compatible with the PrimeSim HSPICE tool for extraction. This can be anything from a single model card to a complex configuration of circuit components, including subcircuits and `.INCLUDE` commands. Figure 9 shows an example of a subcircuit model used for the extraction of an IGBT power device.

Figure 9 Example of an IGBT subcircuit model


Example: IGBT Subcircuit
C
Rc
Cgc
D1
D2
Rb
Q1
Cce
G
Rg
M1
D3
E
Re
Set up optimization parameters for diode model D1
OptParam class initializes SPICE model parameter within a set of bounds to be used by optimization engine
Set the BF value of Q1 manually
Model object provides a Python representation of the input model that can be manipulated by users or passed to optimization stages
Set up optimization parameters for BJT model Q1
Set up optimization parameters for M1

```
DN1  = OptParam("d1mod.n",  1,  0.5, 5, 0.1)
DIS1 = OptParam("d1mod.is", 1e-10, 1e-12, 1e-8, 1e-9)
IK1  = OptParam("d1mod.ik", 10, 1e-1, 100, 1)
```

```
Model.SetModelParameter("qmod.BF", BFvalue)
```

```
BN   = OptParam("qmod.nf",  1.5,  0, 4, 0.1)
BIS  = OptParam("qmod.is", 1e-5, 1e-9, 1e-4, 1e-5)
BRB  = OptParam("qmod.RB", 1e-4, 1e-7, 1e-3, 1e-4)
BNKF = OptParam("qmod.NKF", 0.5, 0.01, 1, 0.01)
BIKF = OptParam("qmod.IKF", 1e3, 1, 1e5, 100)
```

```
VTH0 = OptParam(f"igbt_mos.vth0", 1.2, 0, 10, 10)
VOFF = OptParam(f'igbt_mos.voff', -0.08, -0.2, 0.2, 0.05)
MINV = OptParam(f'igbt_mos.minv', 0.0, -2,' 2, 0.1)
K1   = OptParam(f'igbt_mos.k1', 0.5, 0, 10, 0.1)
U0   = OptParam(f'igbt_mos.u0', 0.049, 0.006, 1.0, 0.002)
UA   = OptParam(f'igbt_mos.ua', 1e-20, -1e-8, 1.0e-6, 1e-11)
```

When a model card is supplied on the command line using the `-m` option, it is parsed by the Mystic tool and injected into the extraction environment as a `Model` object. The `Model` object contains a hierarchical representation of the model definition in Python and has an API for manipulating the SPICE model parameter values at any level of the netlist structure using a dot-separated syntax. For example, in the netlist shown in Figure 9, if you want to set the Beta factor (BF) for the BJT model (QMOD) to 100, then you can add the following line to your Mystic strategy:

```
Model.SetModelParameter("QMOD.BF", 100)
```

Conversely, the current value of any model parameter can be retrieved from the netlist using:

```
BF = Model.GetModelParameter("QMOD.BF")
```

The primary objective of a Mystic extraction strategy is to find the set of SPICE model parameter values that ensure the best fit between the provided target data and the output of a PrimeSim HSPICE simulation under matching simulation conditions. To achieve this, the Mystic tool can control the model parameter values during optimization, which is done through the `OptParam` class. `OptParam` allows you to set an initial value (if the parameter does not already exist in the model) and the upper and lower bounds for the set of parameters you want to optimize. The naming conventions here are the same as for `Model.GetModelParameter`, with a dot separating the levels of hierarchy in the netlist. For example, to initialize the Beta factor for optimization, use:

```
# name, default, lower bound, upper bound, scale factor
BF = OptParam("QMOD.BF", 100.0, 1.0, 1000.0, 1.0)
```

For more information about `OptParam`, see Optimization Algorithms on page 36. At any point in the strategy, the current state of `Model` can be written to disk using `Model.write` or pushed to the database using `mdb.StoreFitParameters` or `mdb.StoreFitData`.

Simulator

The Mystic tool uses the PrimeSim HSPICE tool as a backend simulation engine. Before running the Mystic tool, you must ensure you have a valid PrimeSim HSPICE binary in your environment path or supply the full path to one using the `-l` option.

The Mystic `Simulator` object provides functionality for writing netlists based on the supplied `Model` and `SimData` objects, running PrimeSim HSPICE simulations, parsing the outputs of those simulations, and comparing the results to the target data. In a standard extraction strategy, these operations are all handled automatically by the Mystic tool inside the utility functions provided by `ExtractionUtils` so your only interaction with the simulator will be to set command-line options and to configure the global simulation options. The `Simulator` object is configured with the default PrimeSim HSPICE options listed in Table 7.

Table 7 PrimeSim HSPICE options and default values

Option	Default	Description
`ABSTOL`	1e-15	Absolute error tolerance for branch currents in DC and transient analysis.
`EXT_OP`	1	Allows additional information output in the PrimeSim HSPICE tool.
`GMINDC`	1e-18	Specifies minimum transconductance in parallel for p-n junctions and MOSFET nodes in DC analysis.
`INGOLD`	2	Controls whether the PrimeSim HSPICE tool prints output in exponential form. Default is exponential.
`MEASFORM`	1	Allows writing of measurement output files to Excel or HSIM formats as well as traditional PrimeSim HSPICE formats.
`NOMOD`	1	Suppresses the printing of model parameters in the output.
`NUMDGT`	8	Controls the listing printout accuracy.

These options are added to all backend netlists as `.OPTION` commands and can be altered using the `Simulator.set_options` method.

From a PrimeSim HSPICE licensing perspective, the Mystic tool can run simulations in different modes:

- In *normal mode*, a PrimeSim HSPICE license is checked out, utilized, and then checked back in for every simulation. Depending on the license checkout time and the number of simulations required in an extraction, this might be very inefficient.

- In *server mode*, the Mystic tool checks out and holds a PrimeSim HSPICE server license at the beginning of the extraction and then continually uses it throughout. To activate the PrimeSim HSPICE server mode, add a `-C` option to `Simulator.spiceargs` as follows:

  ```
  Simulator.spiceargs.append('-C')
  ```

A single PrimeSim HSPICE simulation can be run from any `SimData` object using the `SimData.GenerateData` method as follows:

```
IdVg.GenerateData(Simulator, Model)
```

The `Simulator` and `Model` are passed in as arguments to construct the netlist and run the backend PrimeSim HSPICE simulation. On completion of the simulation, the fit column of I_d–V_g is updated with the result of the simulation, and the pointwise error column is updated with the pointwise error between the target and fit curves. When running inside an optimization, this error vector is passed back to the optimization algorithm.

In a standard extraction strategy, the `Simulator` object is most commonly used as an argument to the utility functions `ExtractionUtils.PrintErrors` and `ExtractionUtils.DoStage`, and is rarely used standalone. For more information about these functions, see Full Strategy on page 40.

Parallel PrimeSim HSPICE Simulations

The Mystic tool extracts the values of SPICE model parameters by continually running PrimeSim HSPICE simulations and comparing their results with target data coming from TCAD or measurement. For targets that have more than one dataset, for example, low-drain and high-drain I_d–V_g data, multiple PrimeSim HSPICE simulations are required to complete one evaluation.

The Mystic PrimeSim HSPICE library allows these simulations to be executed in parallel. This can significantly reduce the runtime of complex extraction strategies.

The maximum number of parallel PrimeSim HSPICE simulations is controlled by the property `Simulator.max_sims` as follows:

```
Simulator.max_sims = 10
```

The default value for this property is 1, that is, serial simulation. The objective function of the `Optimizer` object has also been parallelized where possible, particularly in the Jacobian calculation of the commonly used bounded trust region algorithm. The Jacobian evaluation makes small perturbations in each of the parameters being optimized and uses the results to determine a gradient of best descent for the problem.

Consider an optimization problem with five parameters and 2 bits of target data. The number of PrimeSim HSPICE simulations required for a Jacobian evaluation is 5 parameters × 2 samples per parameter × 2 targets = 20 simulations. As these simulations have no dependence on each other, they can all be performed in parallel. To activate this feature, pass `parallel=True` to any call to `ExtractionUtils.DoStage` as follows:

```
ExtractionUtils.DoStage("BelowVt", Model, Simulator,
                        [PHIG,CDSC,QMFACTOR], Optimizer,
                        [IdVgld_BelowVt])
```

The number of parallel simulations in the `Optimizer` is still controlled by the `Simulator.max_sims` property.

Note:

Running PrimeSim HSPICE simulations in parallel requires multiple PrimeSim HSPICE licenses. If you are limited by PrimeSim HSPICE licenses, then you should run PrimeSim HSPICE in server mode to avoid license failures in Mystic. For example:

```
Simulator.spiceargs = ["-C"]
```

With the PrimeSim HSPICE server mode and parallel simulations activated, a queue of PrimeSim HSPICE server licenses is created dynamically when needed, then retained for the duration of the extraction. This is a more reliable way to limit the number of licenses required when running in parallel to the exact number specified by `Simulator.max_sims`.

Parameter Space Exploration

Before extracting a SPICE model parameter set in the Mystic tool, it is important to understand how each parameter in your chosen model affects the target data curves against which you are extracting. To help with this, Mystic provides the following utility functions to explore the parameter space.

The Mystic graphical user interface also provides functionality to explore the parameter space interactively. See Exploring Parameters Interactively on page 82 for details.

Sensitivity Analysis: ExtractionUtils.SensitivityAnalysis

One-factor-at-a-time sensitivity analysis aims at understanding the response of the underlying model as a function of small changes to a single parameter, with all other parameters fixed. It reveals only the local gradient of the response surface of the model with respect to a given parameter.

This local sensitivity analysis can be performed in the Mystic tool by using the `ExtractionUtils.SensitivityAnalysis` utility function as shown here:

```
IdVg_belowvt = IdVg.FilterData("le", 0.4)
IdVg_belowvt = IdVg.FilterData("gt", 0.4)
```

```
PHIG     = OptParam("PHIG", 4.5, 4.2, 4.7, 1.0)
CDSC     = OptParam("CDSC", 0.1, 0.0, 10.0, 1.0)
U0       = OptParam("U0",0.02,0.006,1.0,0.02)
UA       = OptParam("UA",0.3,0.0,100.0,1.0)

sense_df = SensitivityAnalysis([IdVg_belowvt, IdVg_abovevt],
                               Simulator, Model, [PHIG,CDSC,U0,UA])
```

The method can take any number of targets and parameters, and displays the sensitivity for each target individually and the combined root mean square deviation (RMSD) of the targets.

Parameter Sweep: ExtractionUtils.ParamSweep

You can perform a parameter sweep of a single parameter against a single target by using the `ExtractionUtils.ParamSweep` utility function, which is a simpler version of the utility function `ExtractionUtils.SensitivityAnalysis` and produces one PLT file for each evaluation of the target at a particular parameter value. The following example shows a workfunction parameter sweep using the BSIM-CMG parameter `PHIG`:

```
ExtractionUtils.ParamSweep("idvg-ld", IdVgld, Simulator, Model,
                           PHIG, "rmsd")
```

The default number of parameters in the sweep is 11, and the default upper and lower bounds are taken from the PHIG `OptParam` definition. These can be overwritten using the `num`, `pmin`, and `pmax` function arguments, respectively.

Optimization Algorithms

The core of the Mystic tool is a powerful optimization library with several algorithms to help optimize SPICE model parameter values to fit the target data provided. The `Optimizer` object is injected into the environment at runtime and provides an interface to the underlying library.

Table 8 Supported optimization algorithms

Algorithm	Description
`BOUNDED_TRUST_REGION`	Use Intel MKL bounded trust region least squares solver.
`TRUST_REGION`	Use Intel MKL trust region least squares solver.
`least_squares`	Use least squares solvers. Available internal methods are trust region, dogbox, and Levenberg–Marquardt.

Table 8 Supported optimization algorithms (Continued)

Algorithm	Description
`minimize`	Use minimizers. A cost function to aggregate the residuals must be given in this case (root mean square error by default). Available minimizers include trust region, BFGS (default), COBYLA, and L-BFGS-B.
`differential_evolution`	Use a differential evolution algorithm to find the global minimum of the multivariate function.
`doe`	Provides access to the Bayesian search-based optimization method.

The initial algorithm of the `Optimizer` is dictated by the `-o` option. If not supplied, then it defaults to `BOUNDED_TRUST_REGION`, which is used most frequently throughout Mystic extractions. Inside the strategy, the algorithm can be changed using `Optimizer.SetMethod` as follows:

```
Optimizer.SetMethod("least_squares")
```

Each algorithm has a set of parameters that can be used to adjust tolerances, the method, the initial conditions, and so on. A description of these parameters for the available algorithms can be found in the Optimizer/Optimization Parameter section of the API documentation. These parameters can be set using the `Optimizer.SetOptimizationParameter` method. The following code snippet illustrates some recommended settings for the `BOUNDED_TRUST_REGION` algorithm:

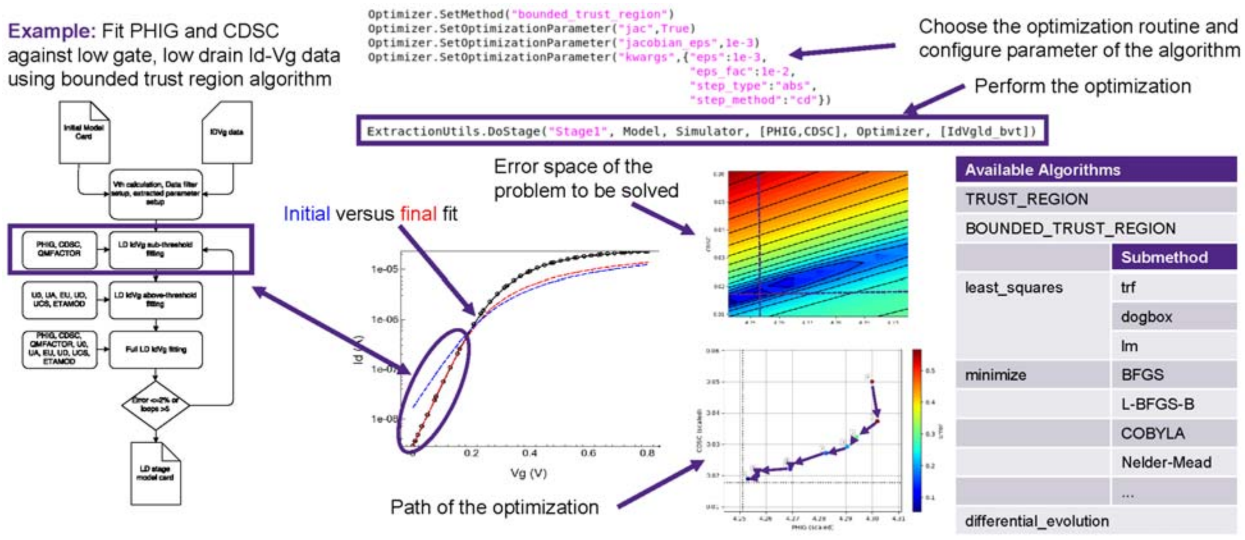
```
Optimizer.SetMethod("least_squares")
Optimizer.SetOptimizationParameter("eps1", 1e-3)
Optimizer.SetOptimizationParameter("eps2", 1e-3)
Optimizer.SetOptimizationParameter("eps3", 1e-3)
Optimizer.SetOptimizationParameter("eps4", 1e-3)
Optimizer.SetOptimizationParameter("eps5", 1e-3)
Optimizer.SetOptimizationParameter("eps6", 1e-3)
Optimizer.SetOptimizationParameter("jac", True)
Optimizer.SetOptimizationParameter("kwargs", {"eps": 1e-3,
                                              "step_type": "abs",
                                              "step_method": "cd",
                                              "eps_fac": 1e-6})
```

As mentioned in SPICE Models, `OptParam` objects are created for all parameters that will be optimized. How these are used varies depending on the optimization algorithm. For example, if `BOUNDED_TRUST_REGION` is used, then the parameter value will remain strictly within the bounds defined by the `OptParam` object during optimization. However, if `TRUST_REGION` is used, then these bounds will be ignored.

The last argument in the `OptParam` definition is a scale factor that can play a very important role in the convergence of an optimization, particularly with gradient descent–based methods. The idea of the scale factor is to balance the sensitivities of a set of parameters being used in the optimization to prevent one parameter dominating another parameter. A sensible default value for the scale factor is the initial parameter value. It can then be increased to add more sensitivity or decreased to reduce the sensitivity. To better understand the sensitivity of a set of parameters against an extraction target, you can use the utility function `ExtractionUtils.SensitivityAnalysis`.

As with the `Simulator`, the `Optimizer` has limited use in the extraction strategy as a standalone object beyond algorithm selection and parameter setup. It is most commonly used in combination with other objects in the `ExtractionUtils.DoStage` utility function, as shown in Figure 10.

Figure 10 Overview of Mystic optimization engine



```
Optimizer.SetMethod("bounded_trust_region")
Optimizer.SetOptimizationParameter("jac",True)
Optimizer.SetOptimizationParameter("jacobian_eps",1e-3)
Optimizer.SetOptimizationParameter("kwargs",{"eps":1e-3,
                                              "eps_fac":1e-2,
                                              "step_type":"abs",
                                              "step_method":"cd"})
```

```
ExtractionUtils.DoStage("Stage1", Model, Simulator, [PHIG,CDSC], Optimizer, [IdVgld_bvt])
```

Available Algorithms	
TRUST_REGION	
BOUNDED_TRUST_REGION	
	Submethod
least_squares	trf
	dogbox
	lm
minimize	BFGS
	L-BFGS-B
	COBYLA
	Nelder-Mead
	...
differential_evolution	

Example: Fit PHIG and CDSC against low gate, low drain Id-Vg data using bounded trust region algorithm
Choose the optimization routine and configure parameter of the algorithm
Perform the optimization
Initial Model Card
IDVg data
Vth calculation, Data filter setup, extracted parameter setup
PHIG, CDSC, QMFACTOR
LD IdVg sub-threshold fitting
U0, UA, EU, UD, UCS, ETAMOD
LD IdVg above-threshold fitting
PHIG, CDSC, QMFACTOR, U0, UA, EU, UD, UCS, ETAMOD
Full LD IdVg fitting
Error <=2% or loops >5
LD stage model card
Error space of the problem to be solved
Initial versus final fit
Id (A)
Vg (V)
Path of the optimization
PHIG (scaled)
CDSC (scaled)

The `ExtractionUtils.DoStage` utility function takes the `Model`, the `Simulator`, a list of `OptParam`, the `Optimizer`, and a list of `SimData` objects, and combines them to perform a single optimization step within a strategy. After a stage completes, the optimization result is written to screen or to the Mystic log file, the optimal parameter values are retained in the `Model` object, and the optimal simulation results are stored in the `SimData` objects.

Bayesian Optimization Algorithm

The Mystic optimization library provides a Bayesian optimization algorithm. This algorithm performs an adaptive search of the parameter space guided by current available knowledge from all the previously performed simulations. It starts with Latin hypercube sampling to construct an initial Gaussian process response surface model. In each iteration, an acquisition function (expected improvement) is evaluated, and new points are selected that could potentially improve on the current best solution.

To use the Bayesian optimization algorithm in Mystic, add the following to your extraction strategy:

```
Optimizer.set_method("doe")
Optimizer.set_optimization_parameters(method="bayesian_optimizer")
```

A list of other options for the Bayesian optimizer can be found in the Optimizer/Optimization Parameter section of the API documentation (see Mystic API Documentation on page 9).

The Engine Module

The `Engine` module is designed to be an object-oriented replacement for the `ExtractionUtils` module. It groups together the `Model`, `Simulator`, and `Optimizer` objects and provides functionality for defining the flow of an extraction strategy. A full description of the `Engine` module can be found in the Mystic API documentation.

A SPICE model extraction strategy is defined as a series of sequential optimization stages that fit sets of SPICE model parameters to a section of target data. The `Engine` module allows you to calculate parameter sensitivity, to define extraction steps, and to extract SPICE model parameters. Multiple stages can also be grouped together in a loop with a specified exit criterion.

The following code extract is from an example of a low-drain I_d–V_g strategy:

```
Engine.add_extraction_step(IsVgld_BelowVt, [PHIG,QMFACTOR,CDSC])
Engine.add_extraction_step(IsVgld_AboveVt, [U0,UA,EU,UD,UCS,ETAMOB])
Engine.add_extraction_step(IsVgld_BelowVt + IsVgld_AboveVt,
                           [PHIG,QMFACTOR,CDSC,U0
                            UA,EU,UD,ETAMOB,UCS])

fit_error = Engine.extract(error_threshold=2.0, max_loops=3)
```

Individual extraction stages are defined using the `add_extraction_step` method based on a list of SPICE model parameters and a subset of the target data.

The `extract` method is then used to perform the extraction. This method moves through the added stages sequentially, then calculates a single RMS error at the end across a combination of all the data in the stages. If the `error_threshold` is met, then the extraction concludes successfully, and it returns to the start of the loop and reruns the stages. If the exit criterion is not met after the maximum number of loops, then the extraction concludes in any case.

Full Strategy

This section describes how all of the individual components of a Mystic SPICE model extraction strategy work together to achieve an accurate SPICE model fitting of a template 14 nm FinFET device at low-drain bias conditions.

Note:

This section covers only the content of the extraction strategy. To run this example, see Chapter 6 on page 89.

In general, a standard extraction strategy can usually be split into the following sections:

- Configuring Settings
- Setting Up the Model
- Loading and Filtering Data
- Defining the Optimization Parameters
- Executing the Extraction Strategy
- Writing the Results

Configuring Settings

This section controls top-level options for the data, simulator, and optimization algorithm. The following code snippet shows the settings applied to the low-drain I_d–V_g extraction of a 14 nm FinFET device. In general, this section remains consistent across most extraction strategies.

```
tfin           = 8e-09        # Physical fin thickness [m]
hfin           = 4e-08        # Physical fin height [m]
wf0            = 4.301        # Workfunction

mystic_verbose = False
modstr         = "mosmod."
error_method   = "rmsd"

# Data settings
SimData.SetOptions(data_mask=True)

# Simulator settings
Simulator.spiceargs = ['-C']
Simulator.set_options(gmin=1e-18)

# Optimizer settings
eps=1e-3
Optimizer.SetMethod("BOUNDED_TRUST_REGION")
Optimizer.SetOptimizationParameter("jac", True)
Optimizer.SetOptimizationParameter("eps1",eps)
Optimizer.SetOptimizationParameter("eps2",eps)
Optimizer.SetOptimizationParameter("eps3",eps)
Optimizer.SetOptimizationParameter("eps4",eps)
Optimizer.SetOptimizationParameter("eps5",eps)
Optimizer.SetOptimizationParameter("eps6",eps)
Optimizer.SetOptimizationParameter("jacobian_eps",eps)
Optimizer.SetOptimizationParameter("kwargs",{"eps":eps,
                                             "step_type":"abs",
                                             "step_method":"cd"})
```

As the core of the Mystic tool is a Python interpreter, any number of custom variables can be defined through the strategy. The example defines some physical characteristics of the device:

- `mystic_verbose` sets the verbosity of extraction stages.
- `modstr` is the name of the model in the model card.
- `error_method` is the single error method to use when reporting the evaluation error back in verbose mode.

Target data options are configured through the `SimData.SetOptions` method. Here, `data_mask=True`. This argument handles data more efficiently when filtering and reduces extraction runtime. For details, see Data Masks on page 28.

Simulator settings can be grouped into two categories:

- Command-line arguments to pass to PrimeSim HSPICE, controlled with `spiceargs`
- Options to add to the backend netlists as `.OPTION` commands, controlled by `Simulator.set_options`

Here, you use the PrimeSim HSPICE server mode (see Simulator on page 33) and set the minimum conductance in PrimeSim HSPICE. The `Optimizer` object is configured with the `BOUNDED_TRUST_REGION` algorithm, which is the default. This can be changed using `Optimizer.SetMethod`.

The `eps1` to `eps6` arguments control the exit criteria of the algorithm. The value 1e-3 is usually a sensible value for these, although it might need to be reduced or increased for better accuracy or turnaround time.

`kwargs` is a dictionary of options to pass to the Jacobian calculation of the `BOUNDED_TRUST_REGION` algorithm (see Optimization Algorithms on page 36).

Setting Up the Model

Next, the initial `Model` object must be prepared for extraction by setting some default values. Here, you set the fin thickness, fin height, and workfunction based on the quantities defined earlier in the strategy. You also set the source and drain resistance of the model to zero to match the conditions of the target simulations. The specific parameter set used for a BSIM-CMG model extraction is beyond the scope of this user guide. For a more in-depth look at the model, refer to the TCAD to SPICE module of the TCAD Sentaurus Tutorial (see TCAD Sentaurus Tutorial: Simulation Projects on page 10).

```
Model.SetModelParameter(f'{modstr}TFIN',tfin)
Model.SetModelParameter(f'{modstr}HFIN',hfin)
Model.SetModelParameter(f'{modstr}PHIG',wf0)
Model.SetModelParameter('r_sc',0)
Model.SetModelParameter('r_dc',0)

Model.SetModelParameter(f'{modstr}ETA0', 0.0)
Model.SetModelParameter(f'{modstr}DVTP1',1.0)
Model.SetModelParameter(f'{modstr}CGSL', 1e-20)
```

Loading and Filtering Data

Now, you load in the target data. When running the Mystic tool through Sentaurus Workbench, data is usually loaded in from the database using `mdb.Load` and `mdb.GetFitData`. This example is standalone, meaning you do not have a database connection and all data is file based. Here, you load the data in from PLT files using the `SimData.FromPLT` method. The target PLT files come from a Sentaurus Device simulation of a 14 nm FinFET template device.

As can be seen in the following code snippet, the target data is attached with metadata containing the simulation temperature and a nodes list. The nodes list instructs the Mystic tool about the terminal order when it constructs netlists and must match the terminal naming in the PLT file.

```
metadata = {"temperature": 300,
            "nodes": ["drain","gate","source","substrate"]}

IdVgld = SimData.FromPLT("targets/IdVg_ld.plt", ivar="vgate",
                         dvar="idrain", metadata=metadata)

vt_gm_val = float(IdVgld.VtGmMaxFinal())

IdVgld         = IdVgld.FilterData('lt', -1e-13, dtype="dvar",
                                   mask=False)
IdVgld_BelowVt = IdVgld.FilterData('le', vt_gm_val-0.05, dtype="ivar")
IdVgld_AboveVt = IdVgld.FilterData('ge', vt_gm_val+0.05, dtype="ivar")
```

```
IdVgld._name = "IdVgld"
IdVgld.interactive = True
```

The `IdVgld SimData` object can then be filtered around `VtGmMax` in preparation for the extraction.

Defining the Optimization Parameters

Now, you must initialize an `OptParam` of each SPICE model parameter to be extracted. For some parameters, such as the workfunction, it might be sensible to restrict the bounds based on its current value. In this case, the workfunction is obtained from the TCAD simulation, so you do not want it to vary significantly from that value.

```
phigval = Model.GetModelParameter(f'{modstr}PHIG')
PHIG    = OptParam(f'{modstr}PHIG',   phigval, phigval-0.5,
                   phigval+0.5, 1.0)
CDSC    = OptParam(f'{modstr}CDSC',   0.001,   0.0,   10.0,  1.0)
U0      = OptParam(f'{modstr}U0',     0.02,    0.006, 1.0,   0.02)
UA      = OptParam(f'{modstr}UA',     0.3,     0.0,   100.0, 1.0)
ETAMOB  = OptParam(f'{modstr}ETAMOB', 2.0,     0.0,   10.0,  2.0)
EU      = OptParam(f'{modstr}EU',     2.5,     0.1,   10.0,  2.0)
UD      = OptParam(f'{modstr}UD',     0.1,     0.0,   1.0,   0.1)
UCS     = OptParam(f'{modstr}UCS',    1.0,     0.0,   10.0,  1.0)

Stage1Parameters = [PHIG,CDSC]
Stage2Parameters = [U0,UA,EU,UD,UCS,ETAMOB]
Stage3Parameters = [PHIG,CDSC,U0,UA,EU,UD,ETAMOB,UCS]
```

The parameters are assembled into lists that will control which segments of data they are optimized against:

- `Stage1Parameters` will be fitted against below V_t data.
- `Stage2Parameters` will be fitted against above V_t data.
- `Stage3Parameters` will be fitted against all the low-drain I_d–V_g data.

Executing the Extraction Strategy

Now everything is set up to begin the extraction process. As the environment is Python, you can use loops and value comparisons to control the flow of the extraction. For low-drain I_d–V_g data, the extraction strategy is very simple: calculate the single-value fit error of the model against the `IdVgld` target, and continue executing the extraction loop until the error has dropped below 2% or you have reached the maximum of five loops.

In the core of the loop, first you fit the below-V_t data, then the above-V_t data, and then all the data together.

```
fit_error=ExtractionUtils.PrintErrors("Initial", IdVgld, Simulator,
                                       Model, error_method,
plot_style={"fit":{"colour":"#0000ff"}})
x=0

while (x < 5 and fit_error>2.0):
   ExtractionUtils.DoStage("Step1", Model, Simulator,
      Stage1Parameters, Optimizer, [IdVgld_BelowVt],
      verbose=mystic_verbose)
   ExtractionUtils.DoStage("Step2", Model, Simulator,
      Stage2Parameters, Optimizer, [IdVgld_AboveVt],
      verbose=mystic_verbose)
   ExtractionUtils.DoStage("Step3", Model, Simulator,
      Stage3Parameters, Optimizer, [IdVgld_BelowVt,IdVgld_AboveVt],
      [1,2], verbose=mystic_verbose)
   fit_error=ExtractionUtils.PrintErrors("IdVgld", IdVgld,
      Simulator, Model, error_method)
   x+=1

ExtractionUtils.PrintErrors("IdVgld", IdVgld, Simulator, Model,
                            error_method)
```

When using gradient descent–based algorithms such as bounded trust region, it is unlikely that fitting all the parameters against all the data, at the same time, will find a sensible minimum. The problem space is too big and the relationship between the SPICE model parameters is too nonlinear. It is possible to achieve a good result in a single extraction step by using a global optimization routine such as differential evolution, but this can be extremely time-consuming, taking thousands of iterations. Usually, an iterative approach as shown here can find a balance between the final error and the extraction time. After the extraction has finished, the final error is printed to screen.

Writing the Results

All that remains is to write out the results of the extraction. If you run the Mystic tool through Sentaurus Workbench, results will be stored in the database there. Data is written to a PLT file for visualization in Sentaurus Visual, and the `Model` is written as a `.mod` file. This will be used as the initial model for the next extraction strategy.

The evaluation history of the `Optimizer` can be written to a CSV file. This can be useful as it shows the single error and parameter value progression across the entire extraction strategy, which can help to identify problems or inefficiencies.

```
IdVgld.WritePLTFile("./results/IdVg-ld")
Model.write("./models/nmos-ld.mod")
Optimizer.write_parameter_csv("./results/IdVg-ld-parameters")
```

4

Special Use Cases

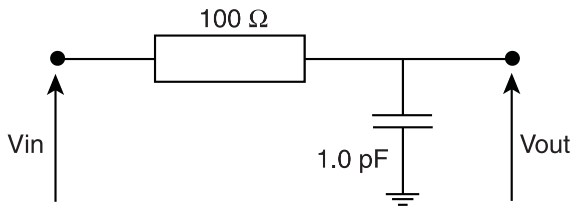
This chapter presents specific use cases in the Mystic tool.

Transient Simulation Extraction

The Mystic tool can handle and extract model parameters against transient simulation data coming from Sentaurus Device as any CSV-like data. This section describes how to upload transient simulation data and use it to extract SPICE parameters based on a very simple RC time constant example shown in Figure 11.

```
.subckt rc in out

r in out 100.0
c out 0 1e-12

.ends
```

Figure 11　　Simple RC netlist example schematic



The subcircuit `rc` will become the initial model for the extraction. Synthetic target data has been generated by PrimeSim HSPICE at R=100 and C=3e-12, with a square wave input stimulus. This data has been written to a CSV file as shown in Figure 12.

Before performing a parameter extraction, the target data must be uploaded to the TCAD to SPICE database as follows:

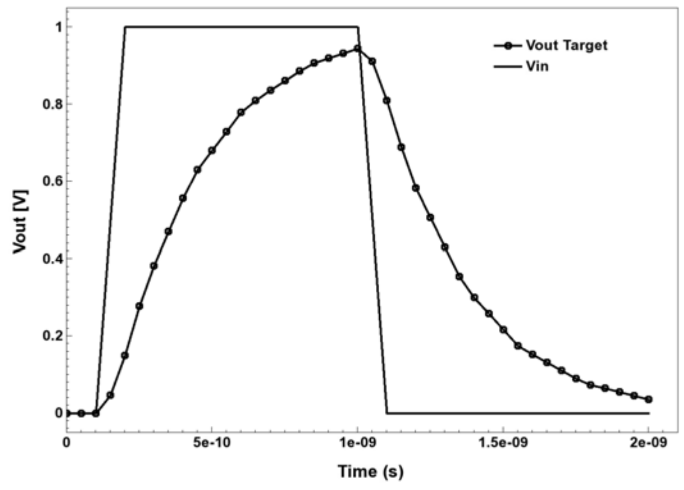
```
Data.from_csv("rc.csv", ivar="time", stimulus="vin",
              metadata={"nodes":["in","out"], "t":300},
              upload_ds="rc-target-data")
```

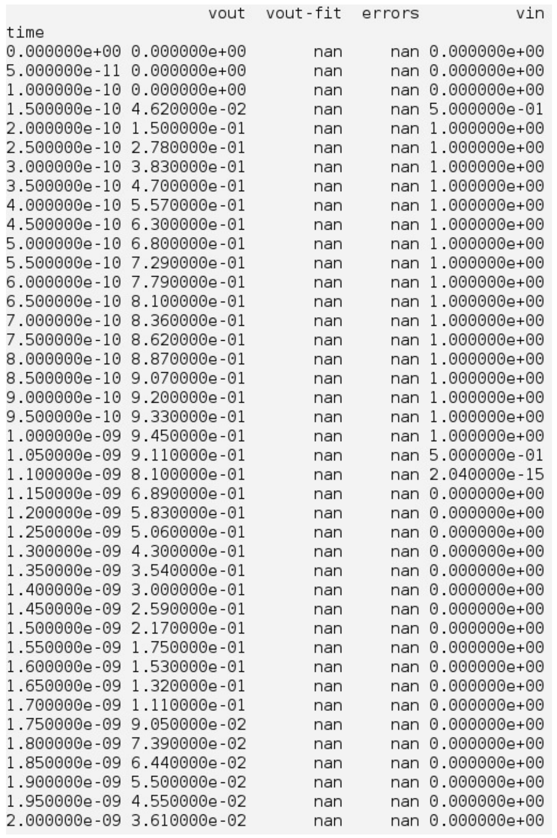
For more details on uploading target data, see Sentaurus Workbench API Documentation on page 9.

Note:

The `ivar` label for transient data must be `time` because the Mystic tool uses this in the backend to determine the simulation type and to set up the appropriate backend SPICE simulation.

Figure 12 Synthetic target I–V data generated by PrimeSim HSPICE tool





```
                       vout  vout-fit  errors          vin
time
0.000000e+00 0.000000e+00       nan     nan 0.000000e+00
5.000000e-11 0.000000e+00       nan     nan 0.000000e+00
1.000000e-10 0.000000e+00       nan     nan 0.000000e+00
1.500000e-10 4.620000e-02       nan     nan 5.000000e-01
2.000000e-10 1.500000e-01       nan     nan 1.000000e+00
2.500000e-10 2.780000e-01       nan     nan 1.000000e+00
3.000000e-10 3.830000e-01       nan     nan 1.000000e+00
3.500000e-10 4.700000e-01       nan     nan 1.000000e+00
4.000000e-10 5.570000e-01       nan     nan 1.000000e+00
4.500000e-10 6.300000e-01       nan     nan 1.000000e+00
5.000000e-10 6.800000e-01       nan     nan 1.000000e+00
5.500000e-10 7.290000e-01       nan     nan 1.000000e+00
6.000000e-10 7.790000e-01       nan     nan 1.000000e+00
6.500000e-10 8.100000e-01       nan     nan 1.000000e+00
7.000000e-10 8.360000e-01       nan     nan 1.000000e+00
7.500000e-10 8.620000e-01       nan     nan 1.000000e+00
8.000000e-10 8.870000e-01       nan     nan 1.000000e+00
8.500000e-10 9.070000e-01       nan     nan 1.000000e+00
9.000000e-10 9.200000e-01       nan     nan 1.000000e+00
9.500000e-10 9.330000e-01       nan     nan 1.000000e+00
1.000000e-09 9.450000e-01       nan     nan 1.000000e+00
1.050000e-09 9.110000e-01       nan     nan 5.000000e-01
1.100000e-09 8.100000e-01       nan     nan 2.040000e-15
1.150000e-09 6.890000e-01       nan     nan 0.000000e+00
1.200000e-09 5.830000e-01       nan     nan 0.000000e+00
1.250000e-09 5.060000e-01       nan     nan 0.000000e+00
1.300000e-09 4.300000e-01       nan     nan 0.000000e+00
1.350000e-09 3.540000e-01       nan     nan 0.000000e+00
1.400000e-09 3.000000e-01       nan     nan 0.000000e+00
1.450000e-09 2.590000e-01       nan     nan 0.000000e+00
1.500000e-09 2.170000e-01       nan     nan 0.000000e+00
1.550000e-09 1.750000e-01       nan     nan 0.000000e+00
1.600000e-09 1.530000e-01       nan     nan 0.000000e+00
1.650000e-09 1.320000e-01       nan     nan 0.000000e+00
1.700000e-09 1.110000e-01       nan     nan 0.000000e+00
1.750000e-09 9.050000e-02       nan     nan 0.000000e+00
1.800000e-09 7.390000e-02       nan     nan 0.000000e+00
1.850000e-09 6.440000e-02       nan     nan 0.000000e+00
1.900000e-09 5.500000e-02       nan     nan 0.000000e+00
1.950000e-09 4.550000e-02       nan     nan 0.000000e+00
2.000000e-09 3.610000e-02       nan     nan 0.000000e+00
```

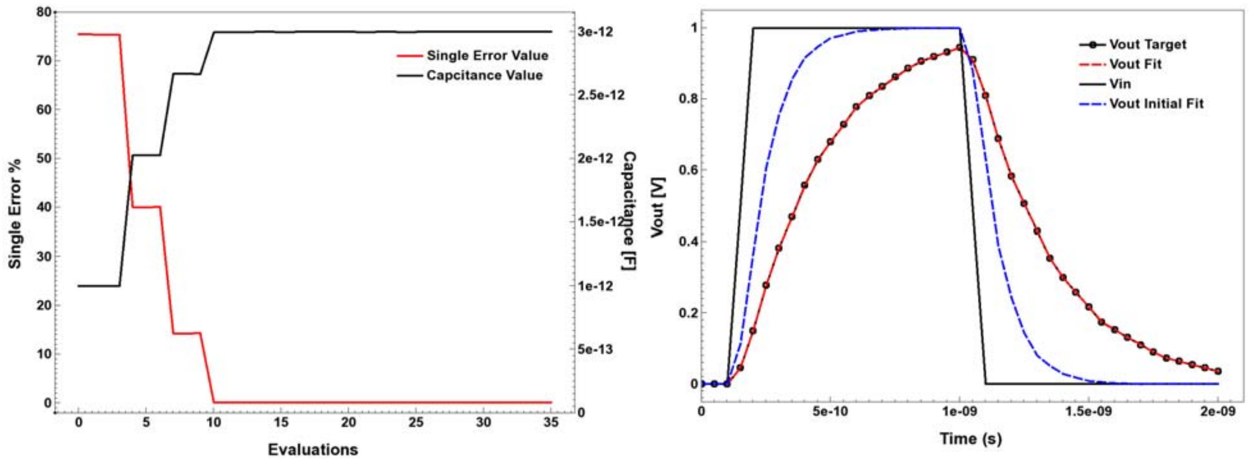
The Mystic strategy for extracting against transient data is essentially the same as a standard DC or AC extraction:

```
# Load the target data from the database
RC = mdb.Load(ivar="time", dvar="vout")

# Calculate an initial error and write a PLT file
ExtractionUtils.PrintErrors("RC-trans", D, Simulator, Model, "rmsd")
RC.WritePLTFile("RC-trans-init", bias_info=False)

# Extract the capacitance
C = OptParam("C", 1e-12, 1e-13, 1e-11, 1e-12)
```

```
ExtractionUtils.DoStage("RC-trans", Model, Simulator, [C],
                        Optimizer, [D])

# Write the final PLT file and the parameter history CSV file
RC.WritePLTFile("RC-trans", bias_info=False)
```

The stimulus columns of the target data (`vin` in this case) are converted into piecewise linear (PWL) functions in the backend SPICE netlist to ensure the simulation receives the same triggers used to generate the target data. Figure 13 shows the parameter value history of the `Optimizer` and the initial and final fits of the capacitance.

Figure 13 (Left) Parameter history and (right) final RC data fit for the capacitance extraction



MOSFET Reliability (MOSRA) Extraction

As CMOS technology scales down, reliability requirements become more challenging and more important. Hot-carrier injection and bias temperature instability can change the characteristics of a device over time. PrimeSim HSPICE MOSRA analysis can be used to model these effects, allowing circuit designers to predict the reliability of their design. Mystic provides the capability to extract PrimeSim HSPICE MOSRA parameters against ΔV_{th} and ΔI_{ds} targets.

The target data for a MOSRA extraction should be presented in a CSV file with the following layout:

```
temperature,stresstime,l,vdrain,vgate,delvth0,dids
85,3000,2.8e-08,1.1,1.1,0.0601,0.0206
85,3000,2.8e-08,1.3,1.1,0.0918,0.1460
85,3000,2.8e-08,1.5,1.1,0.1261,0.9991
85,3000,2.8e-08,1.1,1.3,0.1815,0.0165
85,3000,2.8e-08,1.3,1.3,0.2693,0.0742
85,3000,2.8e-08,1.5,1.3,0.4189,0.5421
```

```
85,3000,2.8e-08,1.1,1.5,0.5309,0.0071
85,3000,2.8e-08,1.3,1.5,0.7941,0.0466
85,3000,2.8e-08,1.5,1.5,1.1795,0.3020
```

The only required labels are `temperature`, `stresstime`, and at least one target, for example, `delvth0` or `dids`. The target labels must match the fields output to the `radeg0` stressing file produced by PrimeSim HSPICE. Other optional column labels are instance parameters, for example, `l`, and terminal biases, for example, `vdrain` and `vgate`.

Unlike conventional data that must be uploaded to the database, MOSRA target data is loaded directly into the strategy using the `CreateMOSRATarget()` method as shown here. In this case, you are only interested in the `delvth0` degradation type:

```
dvth = SimData.CreateMOSRATarget("mosra_target.csv",
                                  target=["delvth0"],
                                  instances=["l"],

metadata={"nodes":["drain","gate","source","substrate"],
          "mosra":{"bias":{"source":0.0,"substrate":0.0},
          "mosra_model":"nfet_ra",
          "base_model":"nfet"}})
```

Lists of instance parameters and target labels must be passed to the function to allow Mystic to identify the appropriate columns in the CSV file. Extra metadata can also be passed in during target creation to pass additional information that is not included in the CSV file such as static biases.

As with any standard Mystic target, the terminals present in the metadata `bias` dictionary must match those in the metadata nodes list. Note that the `mosra_model` and `base_model` keywords, which identify the MOSRA and base model definitions in the initial model card, are required entries in the `mosra` metadata dictionary if the built-in Mystic equations are used. These allow the Mystic tool to identify the MOSRA and base model definitions in the initial model card.

After the targets have been created, they can be treated as standard Mystic target data objects. When passed into an optimization stage, Mystic inherently knows how to set up the backend PrimeSim HSPICE MOSRA aging simulation and to parse the outputs accordingly.

When extracting against multiple MOSRA targets simultaneously, the Mystic tool must run one PrimeSim HSPICE simulation per target, which can cause a significant increase in extraction time. To combat this, the PrimeSim HSPICE MOSRA equations have been built in to the Mystic tool and can be evaluated directly without calling PrimeSim HSPICE. To select this mode of operation, set the `UseMOSRAEquation` attribute at the top of the Mystic extraction strategy as follows:

```
Simulator.UseMOSRAEquation = True
```

When running in this mode of operation, it is necessary to run a validation PrimeSim HSPICE simulation after the model parameters have been extracted to ensure the RMS error is consistent.

To correctly model physical reliability effects for a device, you might want to customize the model equations in PrimeSim HSPICE. Doing so will create a mismatch between the outputs of PrimeSim HSPICE and the outputs of the MOSRA equations built in to the Mystic tool. To prevent this, the Mystic tool allows you to change the built-in equations to match those that are being used in PrimeSim HSPICE using the `SwitchMOSRAEquation` method as shown here:

```
SwitchMOSRAEquation(new_equations, Model, delvth0,
                    error_threshold=1.0)
```

Here, `new_equations` is a Python callable with the function signature shown next, that is, it must accept model and data arguments. Inside the function, the `Model` and `Data` objects can be used to aid the construction of the equation:

```
def new_equations(model, data):
   # Metadata and model parameters can be retrieved from the model
   # and data objects are used within the custom equations
   L = data.metadata["mosra"]["instances"]["l"]
   hciap = model.get_model_parameter("nfet_ra.hciap")
   btiap = model.get_model_parameter("nfet_ra.btiap")

   # Define custom equations and perform calculations. Note that
   # the equations shown here are for illustration only
   dvth0 = 1e-3 * btiap * L
   dids = hciap * L

   # Return a dictionary of values calculated by the custom equations
   return {"delvth0":  dvth0, "dids": dids}
```

The return of the function is a Python dictionary, the keys of which must match the degradation labels from the target CSV file.

When the equations are changed, Mystic will run a validation simulation in PrimeSim HSPICE to ensure that the outputs of the new equations match the outputs of PrimeSim HSPICE within a tolerance dictated by the `error_threshold` argument. If this is not the case, then any parameters extracted against the Mystic equations will give incorrect or unexpected degradation effects when run in PrimeSim HSPICE.

After the extraction has concluded, data can be output to a CSV file by using the `WriteMOSRAFile()` method as shown here. The target and fit column headers will be appended with either `_target` or `_fit`, for example, `delvth0_target`, `delvth0_fit`.

```
delvth0.WriteMOSRAFile("mosra_output.csv")
```

Variability Extraction Using Principal Component Analysis

Adding variation to a SPICE compact model can be achieved in different ways. In TCAD to SPICE tool flows, variation models are created through a combination of variability TCAD simulations, Mystic SPICE model extractions, and RandomSpice simulations.

In these flows, the Mystic tool produces individual model cards without variation information, which are amalgamated into a variation library by the RandomSpice tool.

This library accurately captures the variability trends shown by the input TCAD data and can be used to run simple circuit benchmarks for the target technology. The Mystic tool can also add simple variations to SPICE models themselves, without the need for the RandomSpice tool. This section discusses how to do this using the `CorrelatedParameters` class.

Correlated Parameters

When running PrimeSim HSPICE Monte Carlo simulations, you can add a Gaussian distribution to a single parameter using the `agauss` statement. For example, the following model applies a Gaussian distribution to `PHIG`, causing a V_t shift:

```
.param phig_std = 0.001
.param pvar = agauss(0,1,1)

.model nmos1 nmos
+ level = 72
+ devtype = 1
+ phig = '4.3 + phig_std*pvar'
```

The arguments passed to the `agauss` statement are the mean, the standard deviation, and the number of standard deviations. It is common practice in SPICE modeling to create zero-centered distributions with a standard deviation of one, and then shift them afterwards. In this example, `PHIG` will be centered around 4.3 with a single standard deviation of 0.001.

The simplest form of variation SPICE model is to apply these single distributions to multiple parameters. In reality, the variation of SPICE model parameters is likely to be highly correlated and this oversimplified approach produces unrealistic output distribution. For example, as V_t decreases, the subthreshold slope also starts to decrease, so variation parameters for the workfunction and subthreshold slope are likely to be highly correlated. Applying two uncorrelated Gaussian distributions to these parameters is likely to underestimate the overall variation and give combinations of a high V_t and steep subthreshold slope that would likely not occur.

The `CorrelatedParameters` class can help to add correlations to variation parameters. It uses a principal component analysis (PCA) technique to automatically calculate variation terms for each parameter based on correlation and standard deviation information provided by users, and then it applies these terms to the model. Consider the following example:

```
PHIG = OptParam("PHIG", 4.5, 4.2, 4.7, 1.0)
CDSC = OptParam("CDSC", 0.1, 0.0, 10.0, 1.0)
U0 = OptParam("U0",0.02,0.006,1.0,1.0)
UA = OptParam("UA",0.3,0.0,100.0,1.0)

CP = CorrelatedParameters([PHIG,CDSC,U0,UA])
```

```
CP.setup_model(Model)
CP.update_model(Model)
print(Model)

# Set parameter standard deviations
CP.set_sigma("PHIG",0.01)
CP.set_sigma("CDSC",0.001)
CP.set_sigma("U0",0.005)
CP.set_sigma("UA",0.002)

# Set some correlation between parameters. These are for demonstration
# purposes only and do not represent anything real
CP.set_correlation("PHIG","CDSC", 0.7)
CP.set_correlation("U0","UA", 0.8)

CP.update_model(Model, components=2)
print(Model)
```

Here, you start from a BSIM-CMG model that has been prefitted to some nominal performance target data. The example does not consider variability target data as there is no need for optimization (which is covered in Variability Target Data on page 53). Four parameters are selected to apply variation to, covering the workfunction, subthreshold slope, and mobility. These parameters are first initialized as `OptParams` as in any other standard Mystic extraction and are used to create the `CorrelatedParameters` object. Before adding variation, the model is set up with the correct parameter structure using `CorrelatedParameters.setup_model` and `CorrelatedParameters.update_model`. The first printing of the `Model` object should look something like this:

```
.model nmos1 nmos
+DEVTYPE=1
+LEVEL=72
+PHIG='4.5 + PHIG_var'
+UA='0.001 + UA_var'
+CDSC='0.1 + CDSC_var'
+U0='0.01 + U0_var'

.param PHIG_var = '0.0*pca0 + 0.0*pca1'
.param CDSC_var = '0.0*pca0 + 0.0*pca1'
.param U0_var = '0.0*pca0 + 0.0*pca1'
.param UA_var = '0.0*pca0 + 0.0*pca1'
.param pca0_tmp = agauss(0,1,1)
.param pca0 = pca0_tmp
.param pca1_tmp = agauss(0,1,1)
.param pca1 = pca1_tmp
.param pca2_tmp = agauss(0,1,1)
.param pca2 = pca2_tmp
.param pca3_tmp = agauss(0,1,1)
.param pca3 = pca3_tmp
```

The parameter definitions in the base model definition have been replaced with `'<value> + <param>_var'`. The variation parameters are then defined below the model as a combination of multiple `agauss` statements.

Note:

Before you apply any standard deviations, all of the coefficients are zero, that is, there is no variation.

There are as many `agauss` statements as there are variation parameters. Each `agauss` statement represents one eigenvector of the multivariate parameter distribution, and they are ordered from largest influence to smallest influence, that is, the coefficients of `pca0` represent the largest contribution to the overall variation and `pca3`, the lowest. You can also see that only `pca0` and `pca1` are applied to the parameters. This is controlled by the `components` argument that is passed to `CorrelatedParameters.update_model` method. Depending on the complexity of the variation, the terms of least influence can be omitted without significantly changing the shape of the overall parameter distribution, that is, you can use the `components` argument to reduce the dimensionality of the variation.

Each parameter then has a standard deviation applied to it by using the method `CorrelatedParameters.set_sigma`, and some correlations are applied between parameters using `CorrelatedParameters.set_correlation`. (The standard deviations are solely for demonstration purposes and do not represent any real technology.) After variation has been applied, you can call `CorrelatedParameters.update_model` again to apply these changes to the model. The final model printout is shown here:

```
.model nmos1 nmos
+DEVTYPE=1
+LEVEL=72
+PHIG='4.5 + PHIG_var'
+UA='0.001 + UA_var'
+CDSC='0.1 + CDSC_var'
+U0='0.01 + U0_var'

.param PHIG_var = '-0.0050724244*pca0 + 0.0076987343*pca1'
.param CDSC_var = '-0.00050724244*pca0 + 0.00076987343*pca1'
.param U0_var = '-0.0041036776*pca0 + -0.0023790397*pca1'
.param UA_var = '-0.0016414710*pca0 + -0.00095161589*pca1'
.param pca0_tmp = agauss(0,1,1)
.param pca0 = pca0_tmp
.param pca1_tmp = agauss(0,1,1)
.param pca1 = pca1_tmp
.param pca2_tmp = agauss(0,1,1)
.param pca2 = pca2_tmp
.param pca3_tmp = agauss(0,1,1)
.param pca3 = pca3_tmp
```

The variation coefficients have now been updated to reflect the combined variation of all the parameters. The model can now written to a file and used in a PrimeSim HSPICE Monte Carlo simulation.

Variability Target Data

The variations produced by a Mystic PCA-based variability model can be compared against variability TCAD simulations or measurement target data. This section presents an example of this based on simulation data from the Synopsys statistical variability TCAD simulator, Garand VE. As the Garand VE tool is part of the TCAD to SPICE tool suite, all of its simulation results are uploaded to the TCAD to SPICE database, allowing the Mystic tool to seamlessly load and use the results as target data.

As with any other target in the Mystic tool, all of the information required to set up and run PrimeSim HSPICE simulations under the same conditions as the target is stored in the structure and the metadata of the target itself. Garand VE data can be loaded in to the Mystic environment as usual by using the `mdb.Load` method as follows:

```
IdVg = mdb.Load(ivar='vgate', dvar='idrain', combine=True)
```

In this scenario, the `combine` argument is essential to ensure that the Garand VE data is kept together in a single `Data` object. The second-level independent variable of the data is an increasing integer field labeled `ensemble_id`. The label here informs the Mystic tool that this is variation data and the Mystic tool should set up a PrimeSim HSPICE Monte Carlo netlist when running against this simulation data. It is assumed, at this point, that the `Model` object in the environment has already had some sensible variations applied to it. The variation data can then be passed as usual into `ExtractionUtils.PrintErrors`:

```
ExtractionUtils.PrintErrors('Variation', IdVg, Simulator, Model,
                            'rmsd')
```

When the simulation is finished, the fit column of the I_d–V_g `Data` object is populated with the result of the PrimeSim HSPICE Monte Carlo simulation.

At this point, the error values between the curves are meaningless because each dataset in the target and fit simulations has different random seeds that do not align. To calculate a more sensible error, the results can be transformed, first into single figures of merit and then into the moment of the figure of merit distributions as follows:

```
VtIon = IdVg.MakeFoMTarget(['vt','ion'], [5e-8,None])
MeanStd = VtIon.MakeFoMTarget(['mean','std'])
```

A printout of `MeanStd` would look something like this:

```
               mean          std      mean-fit      std-fit        errors
index
Vt    2.143298e-01 1.986727e-02  2.143137e-01 2.483426e-02  1.767829e-01
Ion  -9.023801e-07 2.105117e-08 -9.017812e-07 2.633861e-08  1.776053e-01
```

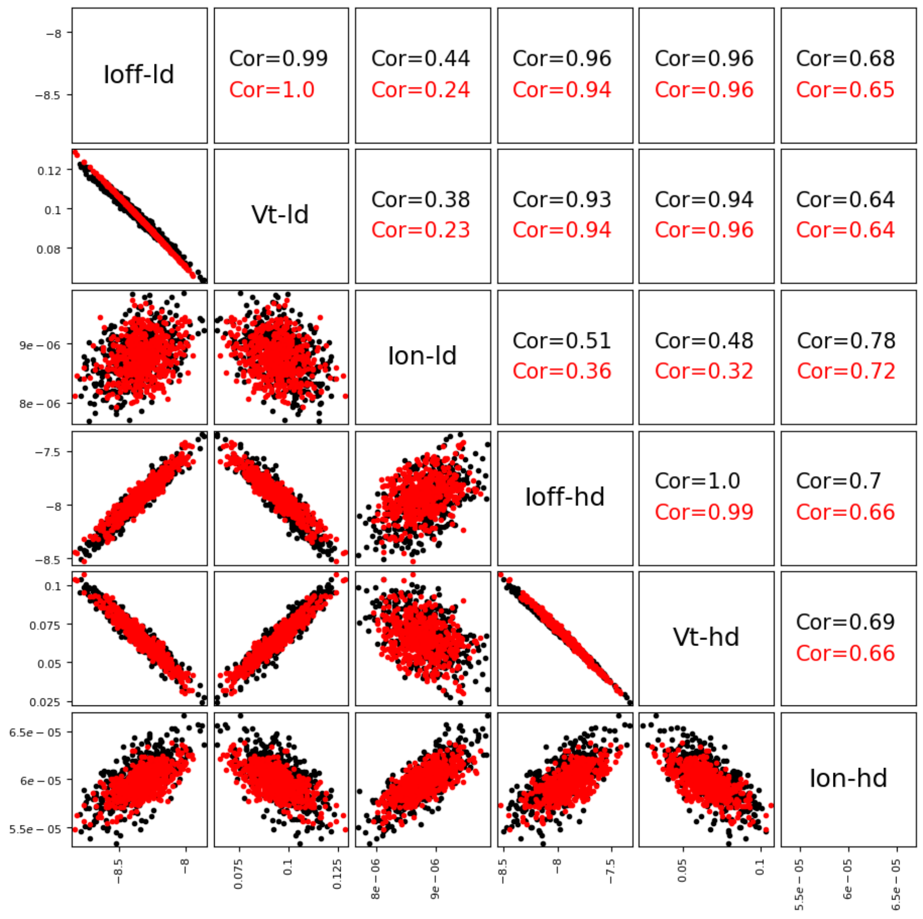
The error column of the transformed data should now contain much more manageable values. Any of these data objects can be written to file using `SimData.WritePLTFile` and plotted externally to the Mystic tool using Sentaurus Visual or the `matplotlib` Python plotting library provided with TCAD to SPICE Python.

Figure 14 shows an example of figure of merit distributions generated by the Garand VE tool being matched by a PCA-based variation SPICE model generated by the Mystic tool.

The transformed targets can also be used to optimize the parameters of the `CorrelatedParameters` class to give a variation model that better matches the Garand VE target distributions, but this is not covered in this document.

For more information as well as access to demonstration PCA projects, contact your local TCAD support team.

Figure 14 Statistical variability figure of merit distributions generated by the Garand VE tool (black) and PCA variation model created by the Mystic tool (red)



RF Extraction

The general procedure of SPICE model parameter extraction for RF applications is slightly more complex than for standard CMOS applications. The main challenges in RF SPICE modeling are:

- There is a very high accuracy requirement on the SPICE models, because the admittance (Y) and scattering (S) parameters of an RF device are determined by the derivatives of I–V characteristics (transconductance and output resistance). To achieve good Y- and S-parameter accuracy, at the device active region, the RMS error of the SPICE model I–V must be under 1%.
- For very high frequency applications required by an RF SPICE model, distributed parasitic networks, which might not be important for general CMOS logic applications, must be considered.

To help address these challenges, the Mystic tool supports the fitting of SPICE model parameters against Y- and S-parameter targets and the extraction of RF-related parasitic networks.

This section describes how to interact with RF-related target data within the Mystic framework but it does not describe in any detail the parameter extraction procedure. For a full demonstration flow of RF SPICE model extraction, contact your local TCAD support team.

Uploading Data

As with I–V, C–V, and transient targets, the Mystic tool expects scattering parameter data to be formatted in a certain way and to provide enough information about the simulation to allow the Mystic tool to reconstruct the simulation conditions in the backend PrimeSim HSPICE netlists it creates and runs.

This section describes how to upload PLT target data from a Sentaurus Device small-signal AC simulation to the TCAD to SPICE database for use in a Mystic extraction. Figure 15 shows the PLT file containing admittance data versus frequency for a two-port network representing the device being modeled. The admittance data can be extracted against directly in the Mystic tool or converted to an S-parameter matrix internally before extracting.

Figure 15 Sentaurus Device AC small-signal PLT output file

```
DF-ISE text

Info {
  version   = 1.0
  type      = xyplot
  datasets  = [
    "frequency"
    "v(1)" "v(2)" "a(1,1)" "c(1,1)" "a(1,2)"
    "c(1,2)" "a(2,1)" "c(2,1)" "a(2,2)" "c(2,2)" ]
  functions = [
    Frequency
    Voltage Voltage Admittance Capacitance Admittance
    Capacitance Admittance Capacitance Admittance Capacitance ]
}

Data {
      7.00000000000000E+09
      4.00000000000000E-01   1.00000000000000E+00   2.05746555054810E-05   2.97051216276209E-14  -5.35215470262388E-06
     -8.78627105607206E-15   5.60590721073175E-02  -3.60398335248718E-14   2.66178221353005E-03   1.68118975683240E-14
      1.40000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   8.22337630452501E-05   2.96824687541098E-14  -2.13926407327530E-05
     -8.78061866298883E-15   5.59809725014359E-02  -3.60108012253218E-14   2.68220851905252E-03   1.68047545845361E-14
      2.10000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   1.84783263306495E-04   2.96447939893905E-14  -4.80736504747715E-05
     -8.77121762734341E-15   5.58510841968250E-02  -3.59625173262611E-14   2.71618445011621E-03   1.67928748365051E-14
      2.80000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   3.27901441671843E-04   2.95922166995640E-14  -8.53159247504598E-05
     -8.75809721449307E-15   5.56698214209221E-02  -3.58951358603892E-14   2.76360870954171E-03   1.67762959513041E-14
      3.50000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   5.11141477823370E-04   2.95249026662063E-14  -1.33009376531383E-04
     -8.74129807073404E-15   5.54377594668781E-02  -3.58088707536249E-14   2.82434060985314E-03   1.67550701858389E-14
      4.20000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   7.33934962097791E-04   2.94430627791226E-14  -1.91013951199770E-04
     -8.72087190386197E-15   5.51556301508956E-02  -3.57039941339137E-14   2.89820118265606E-03   1.67292640153693E-14
      4.90000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   9.95596327434367E-04   2.93469513890680E-14  -2.59160710773220E-04
     -8.69688108056986E-15   5.48243160860411E-02  -3.55808342043607E-14   2.98497457757609E-03   1.66989576135024E-14
      5.60000000000000E+10
      4.00000000000000E-01   1.00000000000000E+00   1.29532813201212E-03   2.92368643435425E-14  -3.37253126445924E-04
     -8.66939814654730E-15   5.44448438591123E-02  -3.54397727002367E-14   3.08440972763483E-03   1.66642442323733E-14
}
```

As shown in Updating PLT Files, `Data.from_plt` can be used to simultaneously read in and upload the PLT data to the TCAD to SPICE database. For admittance data, some extra options and metadata fields are required. The following example demonstrates an admittance file upload:

```
ds = dbi.create_dataset(node_prj, "rf-data", clean=True)

metadata = {"temperature": 298,
            "instances": {"l":3e-8, "w":5e-5},
            "ports": {"1": "gate", "2": "drain"},
              "nodes: ["drain","gate","source","substrate"],
              "bias": {"source": 0.0, "substrate": 0.0}}

Data.from_plt("RFQ_ac_des.plot", ivar="frequency",
              metadata=metadata, upload_ds=ds,
              rf_data=True, combine=True)
```

The `metadata` dictionary contains the same temperature, instances, nodes, and bias fields that are required for DC target data. There is an extra `ports` field that maps port numbers in the network to terminals of the SPICE model. In general, for standard MOSFETs represented as two-port networks, this is always the same: port 1 is mapped to the gate contact and port 2 is mapped to the drain contact.

For admittance data, two extra arguments must be passed to `Data.from_plt`. The `rf_data` argument instructs the `Data` class to produce an `RF_Data` object. This is a derivative of the `Data` class described in The Data Class with some extra functionality for matrix conversion and extracting the real and imaginary parts of the individual data components. The `combine` argument indicates that all of the individual admittance components should be combined horizontally into a single structure as shown here:

```
                   (11)re       y(11)i       y(12)re        y(12)i ...
frequency                                                          ...
2.795000e+10 1.784044e-04 3.542094e-03 -3.196667e-05 -9.527018e-04 ...
2.797500e+10 1.787211e-04 3.545245e-03 -3.202406e-05 -9.535513e-04 ...
2.800000e+10 1.790381e-04 3.548396e-03 -3.208150e-05 -9.544008e-04 ...
2.802500e+10 1.793553e-04 3.551547e-03 -3.213899e-05 -9.552503e-04 ...
2.805000e+10 1.796729e-04 3.554698e-03 -3.219653e-05 -9.560998e-04 ...
```

As with I–V and C–V data, the Mystic tool expects the column headers to be in a specific format that it uses to infer the simulation type when constructing backend PrimeSim HSPICE netlists. For Y- and S-parameter data, Table 9 shows the required header formatting.

Table 9 Header notation

Notation	As ivar	As dvar
`y(<port1><port2>)<real>`	n/a	The admittance between port 1 and port 2 of the network, where `real` denotes whether it is the real or imaginary part
`s(<port1><port2>)<real>`	n/a	The scattering parameter between port 1 and port 2 of the network
`frequency`	AC frequency sweep. This is the required independent variable for admittance or S-parameter data.	n/a

As mentioned in Updating PLT Files, there is a standard mapping to convert Sentaurus Device columns to the expected Mystic format. The default mapping for small-signal AC data is as follows:

```
"a(<port>,<port>)" is converted to "y(<port><port>)re"
```

```
"c(<port>,<port>)" is converted to "y(<port><port>)i"
```

If the data is not coming from the output of a Sentaurus Device simulation, then you can use `column_map` to convert and upload any CSV-like data format. See Uploading CSV Files on page 17.

Loading and Manipulating Data

When the Y-parameter target data has been uploaded to the TCAD to SPICE database, it can be loaded into the Mystic extraction strategy as usual by using the `mdb.Load` method (see Chapter 2 on page 14). The following example shows how to do this for RF-specific data:

```
Y_data = mdb.Load(ivar='frequency',
                  dvar=['y(11)re','y(11)i',
                  'y(12)re','y(12)i',
                  'y(21)re','y(21)i',
                  'y(22)re','y(22)i'])
```

Here, you load the reflection and gain from both ports, giving you the full matrix. This is required if the data should be converted to S-parameter data as the conversion function requires all components to be present. After the data is in the Mystic environment, it can be filtered and resampled like any other data object:

```
Y_f28 = Y_data.FilterData('eq', 28e9, dtype='ivar')

Y22 = Y_f28.FilterData("eq", ["y(22)re","y(22)i"],
                       dtype="column", mask=True)
Y12 = Y_f28.FilterData("eq", ["y(12)re","y(12)i"],
                       dtype="column", mask=True)
Y21 = Y_f28.FilterData("eq", ["y(21)re","y(21)i"],
                       dtype="column", mask=True)
```

The first filter selects the first harmonic of the frequency sweep, and the subsequent filters use `dtype='column'` to isolate individual Y-components, so they can be extracted separately.

Note:

In these filters, `mask=True`. This is essential so a reference to all components is retained in order to construct the PrimeSim HSPICE simulation netlist. It also helps to speed up the extraction when multiple components are being extracted against simultaneously as only a single simulation of the root object is required. Only the RMS error will be calculated on the filtered components.

The `RF_Data` objects have a specific function for performing matrix conversions between Y-parameters and S-parameters. The following example shows how to create an S-parameter target from Y-parameter data using `SimData.MakeFoMTarget`, which is described in Figure of Merit Transforms on page 26:

```
S_data = Y_data.MakeFoMTarget("convert", "s")
```

SPICE model parameters can be extracted against either Y-parameters or S-parameters depending on the required effect.

Extracting Parameters

In general, RF data can be treated in the same as any other type of data when it is used to extract SPICE model parameters.

Individual data objects can be passed to `ExtractionUtils.PrintErrors` to calculate single RMSD errors and to `ExtractionUtils.DoStage` to optimize parameter values. The following example shows a two-parameter fringing capacitance extraction against some Y-parameter target data:

```
cfs = OptParam("submosn.cfringes", 1e-16, 1e-17, 1e-14, 1e-18)
cfd = OptParam("submosn.cfringed", 1e-16, 1e-17, 1e-14, 1e-17)

#Y12_data1 : vg=0.4 vd=1.0
#Y12_data2 : vg=0.4 vd=1.2
#Y12_data3 : vg=0.6 vd=1.2

ExtractionUtils.DoStage("Step1", Model, Simulator,
                        [cfd,cds], Optimizer,
                        [Y12_data1,Y12_data2,Y12_data3],
                        [2,2,1])
```

The target here is the Y12 gain component of the matrix. One of the complexities of RF parameter extraction is that the device often must perform over a range of frequencies and biases. In this example, the full set of Y-parameter data includes three biases denoted in the commented lines.

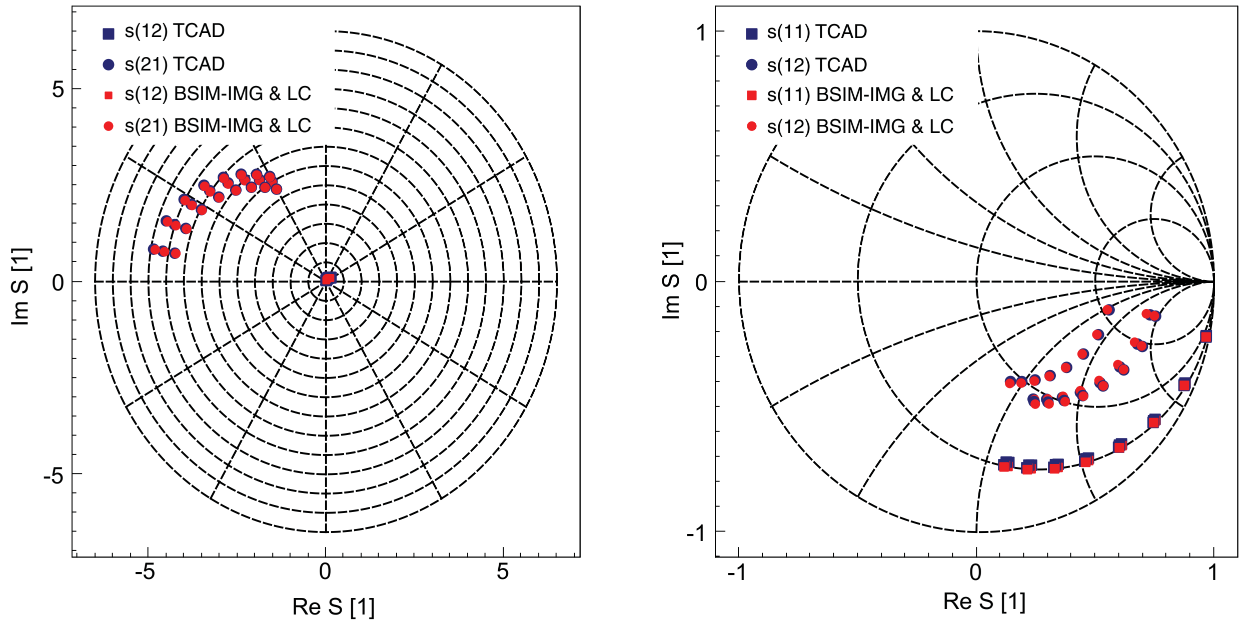
While you want the fit to be as accurate as possible across all data, it sometimes makes sense to apply weighting factors to a section of the data to prioritize accuracy in that area. In this case, you double the influence of the `vg=0.4` targets and deprioritize the `vg=0.6` target. This gives extra flexibility where extreme accuracy cannot be achieved across the full performance range. Note that this is only a single extraction step in a complex multi-stage strategy.

Data Output

RF simulation targets can be written to disk using the `WritePLTFile` or `WriteCSVFile` method. For details about these methods, see the Mystic API documentation.

After the data is written, it can be plotted externally using Sentaurus Visual (see Figure 16). The plotted data covers a range of bias conditions and frequencies, and shows an accurate fitting for all components between the BSIM-IMG SPICE model and TCAD data.

Figure 16 Scattering parameter fitting for BSIM-IMG SPICE model versus TCAD



Lookup Table Modeling

Lookup table (LUT) models are analytic behavioral models that are used to perform fast analysis of a new technology before a reliable SPICE model or extraction strategy has been established. The Mystic tool can generate LUT models based on grids of target I–V and Q–V information using the Synopsys PrimeSim™ tool.

Target data for an LUT extraction should be uploaded to the TCAD to SPICE database in the usual way as described in Chapter 2, Uploading Target Data to Database. The target data should include biases and currents for all the terminals of the device, that is, drain, gate, source, and substrate voltages and currents. It should also have length (L), width (W), and temperature metadata fields. These are required by the PrimeSim LUT model.

After the data has been loaded from the database into the environment using `mdb.Load`, it should be resampled using the `SimData.resampleNd` method to give an evenly spaced, dense grid of points as shown here:

```
IV = mdb.Load(project=targetDataProject, ivar='vgate',
              combine=True, extra_index='all')

IVr = IV.resampleNd(points={'vdrain':51, 'vgate':51, 'vsubstrate':11},
                    overwrite=False)
```

Note:

A denser target grid gives a smoother model response in the PrimeSim tool but results in slower simulation times. You should evaluate some different options and choose a reasonable grid density for your application.

To allow grid resampling, the `combine=True` and `extra_index='all'` parameters should be passed to the `mdb.Load` method to condense all the data into a single object. The `resampleNd` method should be passed as either a `points` or a `step` dictionary that determines the density of the data for all terminal biases.

After the dense data has been created, the LUT can be written using the `Simulator.write_LUT` method as shown here:

```
column_map = {'vdrain':     'Vds',
              'vgate':      'Vgs',
              'vsubstrate': 'Vbs',
              'idrain':     'Id',
              'igate':      'Ig',
              'isource':    'Is',
              'qdrain':     'Qd',
              'qgate':      'Qg',
              'qsource':    'Qs'}

Simulator.write_LUT(IVr[0], 'nmosmod', path='LUT_MODELS',
                    column_map=column_map)
```

The PrimeSim LUT model requires specific formatting of voltages, currents, and charges. Therefore, you must supply a `column_map` dictionary to convert the naming conventions in the target data to the expected format of the PrimeSim tool. The LUT model is written to the directory specified by the `path` argument using the `'nmosmod'` model name.

In the previous example, the output model will be written to `LUT_MODELS/nmosmod_W1u_L32n_T27.dat`, where the device width is 1 µm, the gate length is 32 nm, and the temperature is 27 K. The PrimeSim tool uses this model file when running SPICE simulations.

When the LUT model has been written, you can run a validation simulation using the `ExtractionUtils.PrintErrors` utility as follows:

```
Simulator.executable = 'primesim'

ExtractionUtils.PrintErrors("IV", IVr, Simulator, Model, "rmsd")
```

Before running the simulation, ensure a PrimeSim binary is available is your `$PATH` and switch to using the PrimeSim tool in the Mystic tool by setting `Simulator.executable`.

If the LUT model is operating correctly, then the error printed to screen should be close to zero as the data generated by the PrimeSim simulation should match the target data perfectly.

5

Mystic Graphical User Interface

This chapter describes the graphical user interface of the Mystic tool.

Introduction to the Graphical User Interface

You can use the Mystic graphical user interface (Mystic GUI) to run Mystic SPICE model extraction strategies interactively. It allows you to move through existing extraction strategies step by step, to analyze the outcomes of each individual command, and to generate plots to monitor the progress of the SPICE model fitting against the target data. The built-in IPython console also allows you to build up new strategies step by step, to explore parameter spaces, and to make manual adjustments to the values of SPICE model parameters.

This chapter provides an overview of the Mystic GUI and examples of how the Mystic GUI can be used to enhance the SPICE model extraction process. For more detailed information, consult the Mystic API documentation.

For instructions on how to launch the Mystic GUI, see Running the Mystic Tool as a Standalone Tool on page 10.

Features of the Mystic GUI

The Mystic GUI comprises multiple dockable panels that can be configured to suit your preferences. Figure 17 shows the default layout of panels in the main window of the Mystic GUI.

The default layout splits the Mystic GUI into four main panels. Each of the elements in these panels is described in this section. The elements link together and interactive with each other to provide an environment for the interactive extraction of SPICE compact models.

All panels can be shown or hidden by choosing **View** > **Panels** > *panel name*.

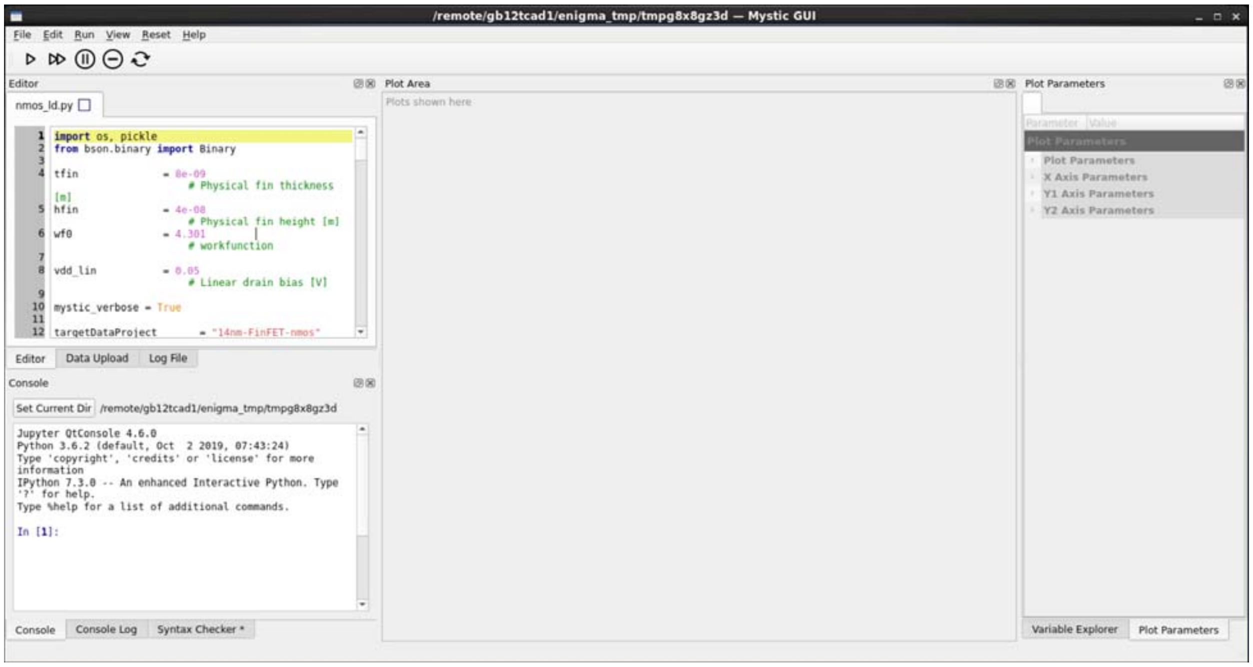
After you have arranged the layout of the main window to your preferences, you can save your layout as follows:

1. Choose **Edit** > **Preferences**.

   The Preferences dialog box opens.

2. In the left pane, click **Layout**.
3. In the **Save Current Layout As** field, enter a name for the layout.
4. Click **OK** or **Apply**.

Figure 17 Default layout of panels in main window of Mystic GUI


/remote/gb12tcad1/enigma_tmp/tmpg8x8gz3d — Mystic GUI
File Edit Run View Reset Help
Editor
nmos_ld.py
Plot Area
Plots shown here
Plot Parameters
Parameter Value
Plot Parameters
Plot Parameters
X Axis Parameters
Y1 Axis Parameters
Y2 Axis Parameters
Editor Data Upload Log File
Console
Set Current Dir /remote/gb12tcad1/enigma_tmp/tmpg8x8gz3d
Console Console Log Syntax Checker *
Variable Explorer Plot Parameters

```
import os, pickle
from bson.binary import Binary

tfin                = 8e-09
                        # Physical fin thickness
[m]
hfin                = 4e-08
                        # Physical fin height [m]
wf0                 = 4.301
                        # workfunction

vdd_lin             = 0.05
                        # Linear drain bias [V]

mystic_verbose = True

targetDataProject          = "14nm-FinFET-nmos"
```

```
Jupyter QtConsole 4.6.0
Python 3.6.2 (default, Oct  2 2019, 07:43:24)
Type 'copyright', 'credits' or 'license' for more
information
IPython 7.3.0 -- An enhanced Interactive Python. Type
'?' for help.
Type %help for a list of additional commands.

In [1]:
```

Menus and Run Toolbar

Table 10 describes the available menus.

Table 10 Menus of Mystic GUI

Menu	Description
File	Commands for opening and saving files, and exiting the Mystic GUI. Shortcuts for these operations are also provided.

Table 10 Menus of Mystic GUI (Continued)

Menu	Description
Edit	Utilities for copying commands, finding patterns in the open file, and accessing the GUI preferences.
Run	Commands for running the strategy file in the Console panel (see Console Panel on page 68).
View	Shows or hides panels, as well as elements visible in your layout.
Reset	Resets the Editor, Console, and Plot Area panels, or resets the entire session.
Help	Provides links to the Mystic API documentation.

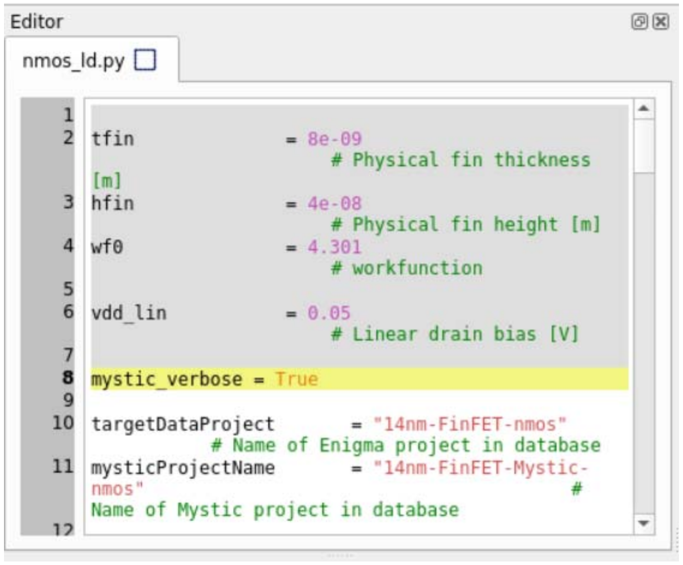
The Run toolbar controls the execution of the strategy (see Controlling the Execution of the Strategy on page 69).

Editor Panel

The strategy file itself opens in the Editor panel. When first opened, the entire strategy is editable. You can make alterations and save these to disk by using the **File** menu.

When an extraction is running, the contents of the file are progressively blocked and made read-only as each line is executed. The line highlighted by a yellow background is the next to be executed. For longer-running commands such as extraction stages, the line that is currently being executed has a green background. Executed lines have a gray background and become read-only (see Figure 18).

Figure 18 Strategy file: executed lines have gray background and are read-only



```
tfin                 = 8e-09
                            # Physical fin thickness
[m]
hfin                 = 4e-08
                            # Physical fin height [m]
wf0                  = 4.301
                            # workfunction

vdd_lin              = 0.05
                            # Linear drain bias [V]

mystic_verbose = True

targetDataProject         = "14nm-FinFET-nmos"
                      # Name of Enigma project in database
mysticProjectName         = "14nm-FinFET-Mystic-
nmos"                                                  #
Name of Mystic project in database
```

Inserting Breakpoints in Strategy File

You can insert breakpoints at any line below the one that is next to be executed.

To insert a breakpoint, either click the line number on the left side or move the cursor to the required line and then click the **Breakpoint** button on the Run toolbar (see Figure 22 on page 69).

After a breakpoint has been set, script execution stops at the breakpoint when you click the **Run/Continue** button on the Run toolbar (see Figure 22).

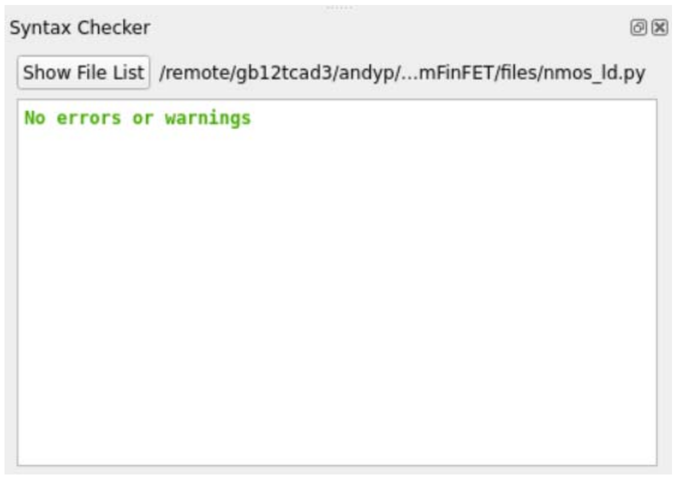
See also Controlling the Execution of the Strategy on page 69.

Syntax Checker Panel

Prior to running a strategy, it is important to check for potential errors in the syntax. Identifying issues can save time, particularly, with long-running strategies.

You can use the Syntax Checker panel to check for errors in the general Python syntax. If no errors are detected, then you will see the message as shown in Figure 19.

Figure 19 Syntax Checker panel with message indicating no identified issues in the strategy

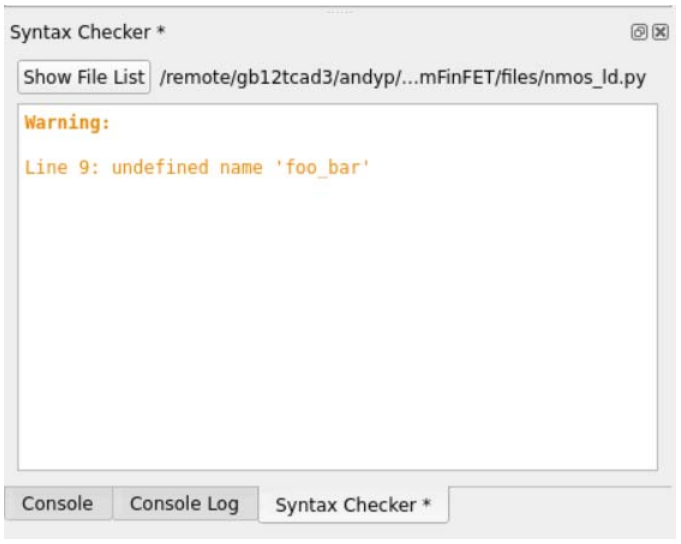


/remote/gb12tcad3/andyp/...mFinFET/files/nmos_ld.py

```
No errors or warnings
```

If there are errors in the strategy file, for example with Python variable definitions, then an asterisk appears on the **Syntax Checker** tab as shown in Figure 20.

Figure 20 Syntax Checker panel indicating an error by inclusion of asterisk next to its name



/remote/gb12tcad3/andyp/...mFinFET/files/nmos_ld.py

```
Warning:

Line 9: undefined name 'foo_bar'
```

Potential issues are split into two categories:

- *Errors* arise from problems that will definitely cause the extraction to fail, for example, Python syntax errors.
- *Warnings* arise from problems that might cause failures but might be harmless.

Note:

The Syntax Checker panel is designed to identify syntax-related problems only and cannot predict failures in the extraction procedure.

Console Panel

The Mystic GUI runs commands by using a built-in IPython console built in to the Console panel. The Mystic namespace, described in Extraction Environment on page 11, is injected automatically into the Console panel when the Mystic GUI is initialized. The Console panel receives commands from the Editor panel and executes them in the context of the Mystic extraction environment.

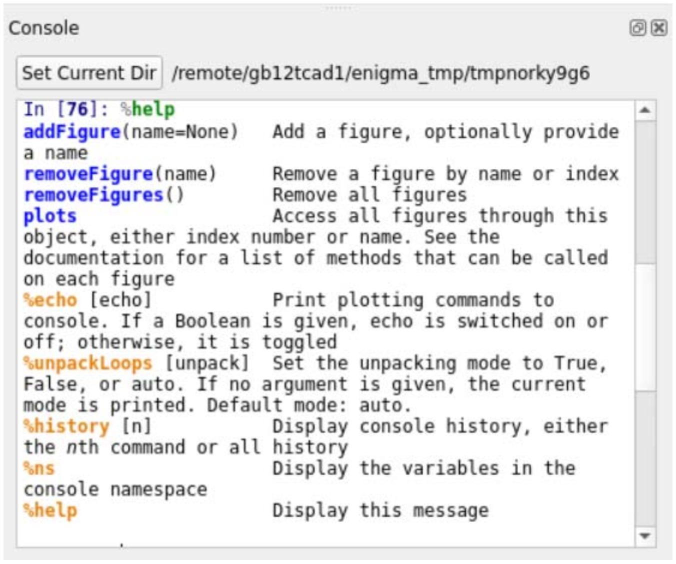
You can also enter commands into the Console panel directly. This can be useful for plotting results at the end of a strategy, making manual adjustments to the model, or debugging failures in the strategy.

As well as the Mystic namespace, the Console panel provides specific commands for creating, modifying, and updating graphics in the Plot Area panel. These commands are described further in Plot Area Panel on page 72. For a complete list of plotting commands, consult the Mystic API documentation.

The IPython console has a set of *magic* commands that provide extra functionality to the Console panel. Those commands specific to the Mystic GUI are described in the following table. IPython also provides a set of built-in magic commands [1].

Command	Description
`%echo`	If this command is set, then making alterations to plots using the Mystic GUI will result in the equivalent command being printed to the Console panel.
`%help`	Prints a general help message, with an overview of some plotting commands and other commands (see Figure 21).
`%history`	Displays all the commands previously executed in the Console panel.
`%ns`	Lists the variables in the namespace.
`%unpackLoops`	When this command is set, *for* and *while* loops and *if* statements are unpacked when executing, so they can be stepped through line-by-line rather than being run as a single block.

Figure 21 Output of the %help command

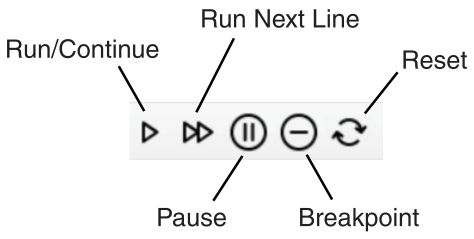


```
In [76]: %help
addFigure(name=None)   Add a figure, optionally provide
a name
removeFigure(name)     Remove a figure by name or index
removeFigures()        Remove all figures
plots                  Access all figures through this
object, either index number or name. See the
documentation for a list of methods that can be called
on each figure
%echo [echo]           Print plotting commands to
console. If a Boolean is given, echo is switched on or
off; otherwise, it is toggled
%unpackLoops [unpack]  Set the unpacking mode to True,
False, or auto. If no argument is given, the current
mode is printed. Default mode: auto.
%history [n]           Display console history, either
the nth command or all history
%ns                    Display the variables in the
console namespace
%help                  Display this message
```

Controlling the Execution of the Strategy

The Run toolbar controls the execution of the strategy in the Console panel.

Figure 22 Run toolbar



The **Run/Continue** button continuously executes commands until either a breakpoint or the end of the file is reached. This execution can be paused at any time by clicking the **Pause** button.

Note:

The line that is currently being executed will run until completion first.

By default, indented code blocks such as *if* statements and *for* and *while* loops are treated as a single block of Python code in continuous run mode. This means that the entire block is parsed and executed as a single entity.

The **Run Next Line** button executes the next line of the strategy that has a yellow background in the Console panel. If the next line is the start of an indented block, then the block is unpacked, and the cursor and next line highlighting will move inside it. This means you can loop round a *for* or *while* loop multiple times, line by line, as you would in a debugger.

You can adjust the rules for unpacking blocks by using the `%unpackLoops` command.

Otherwise, to adjust the rules for unpacking blocks:

1. Choose **Edit** > **Preferences**.

   The Preferences dialog box opens.

2. In the left pane, click **Run**.
3. Under Loop Unpacking, select the required option.
4. Click **OK** or **Apply**.

The **Breakpoint** button inserts a breakpoint at the current line where the cursor is positioned in the Editor panel. A red dot next to the line number indicates that the line has a breakpoint. When breakpoints are set, the **Run/Continue** button will execute only as far as the next breakpoint. You can also create breakpoints by clicking the line number in the Editor panel.

The **Reset** button resets the entire console namespace, removing all plots and variables created or modified throughout the strategy, including the model. Click this button if the strategy is not converging as you expected, and you want to start again.

Caution:

Any information saved to the database is not reset, and content written to disk is not removed.

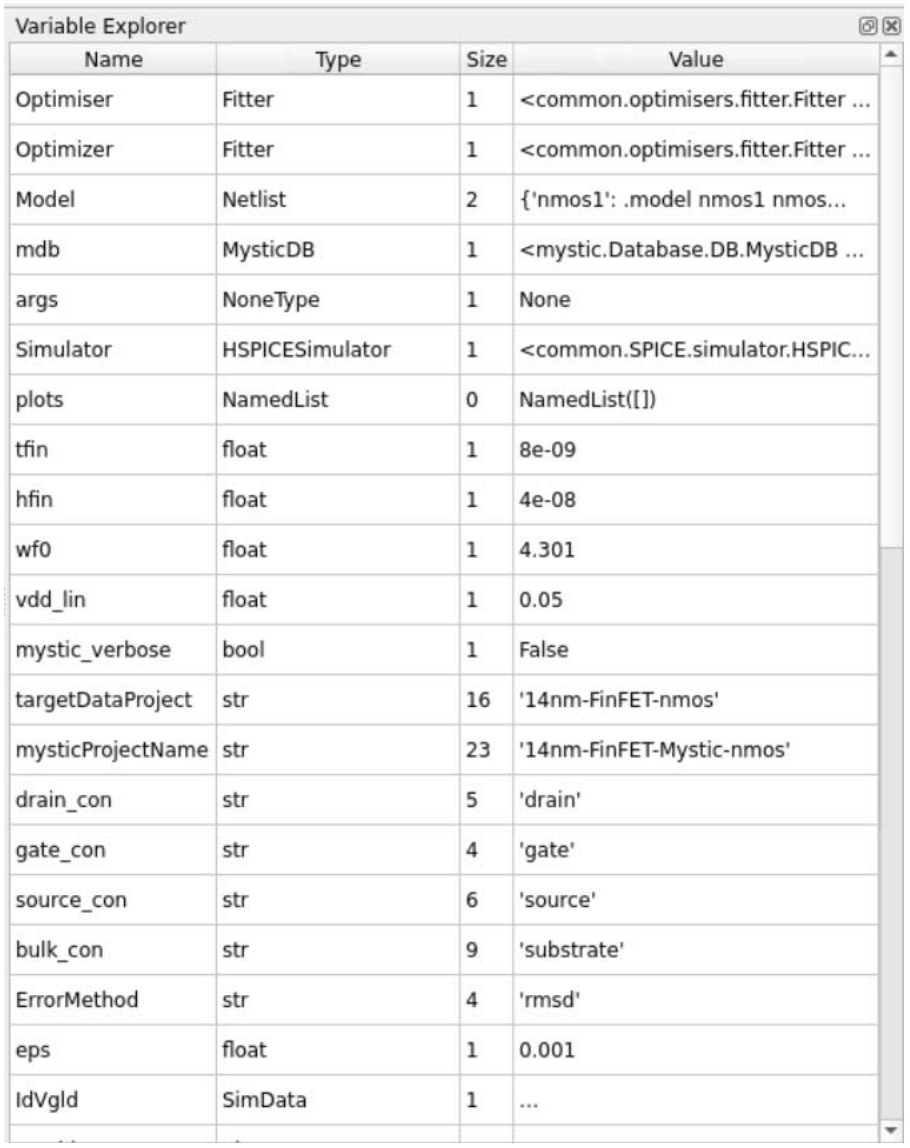
Variable Explorer Panel

You can use the Variable Explorer panel to browse the Python variables currently defined in the Console panel. The variable name, type, size, and value are displayed in a table (see Figure 23). The list of variables grows as the strategy runs and new variables are defined.

The following variable types can be explored further by double-clicking the Value entry of the variable: `list`, `dict`, `np.array`, `pd.DataFrame`, `Data`, `SimData`, and `Netlist`.

For example, double-clicking the `Model` object (a `Netlist` instance) displays a table of all the SPICE model parameters and their current values. Parameters that have been modified during the extraction strategy are highlighted in green as shown in Figure 24. The `Model` can be saved to a file at any time by clicking the **Save** button in the upper-right corner.

Figure 23 Variable Explorer showing main table



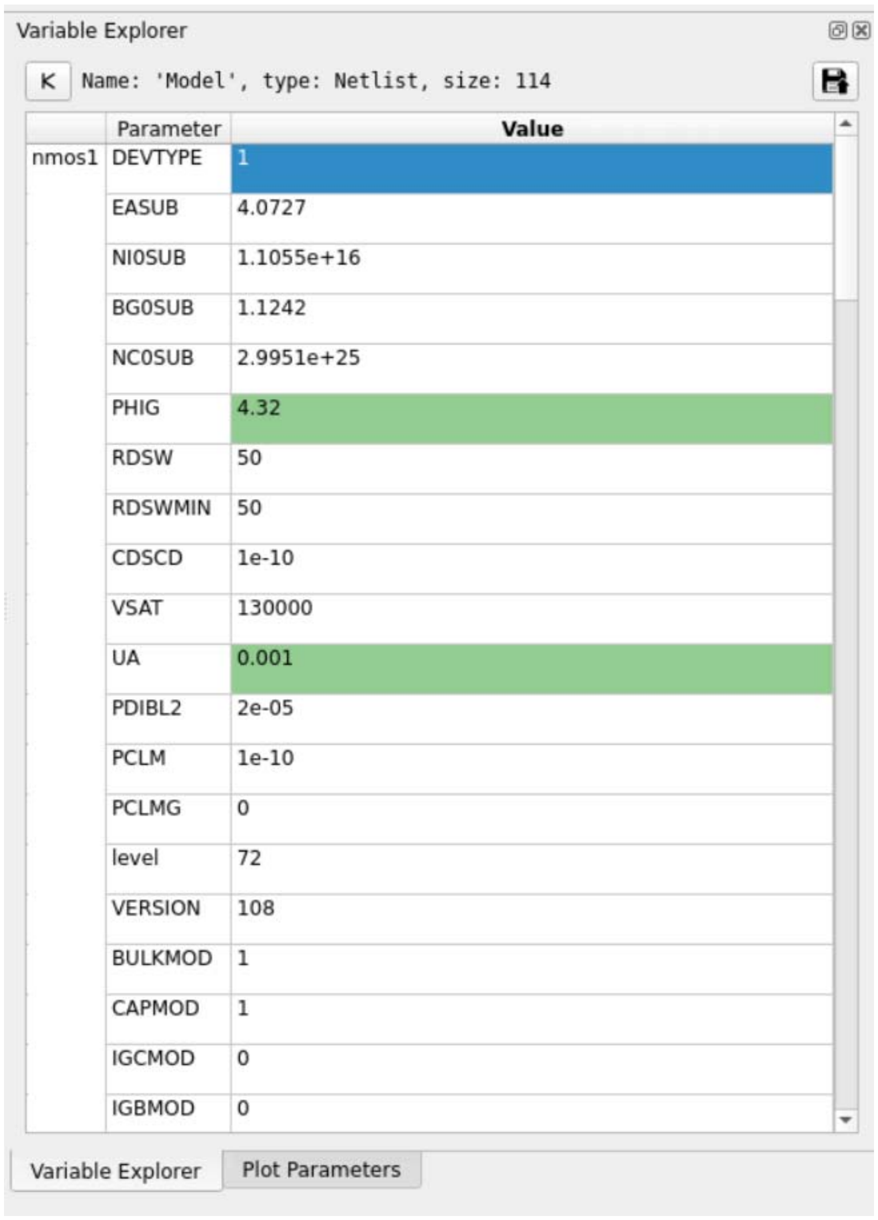
Variable Explorer

Name	Type	Size	Value
Optimiser	Fitter	1	<common.optimisers.fitter.Fitter ...
Optimizer	Fitter	1	<common.optimisers.fitter.Fitter ...
Model	Netlist	2	{'nmos1': .model nmos1 nmos...
mdb	MysticDB	1	<mystic.Database.DB.MysticDB ...
args	NoneType	1	None
Simulator	HSPICESimulator	1	<common.SPICE.simulator.HSPIC...
plots	NamedList	0	NamedList([])
tfin	float	1	8e-09
hfin	float	1	4e-08
wf0	float	1	4.301
vdd_lin	float	1	0.05
mystic_verbose	bool	1	False
targetDataProject	str	16	'14nm-FinFET-nmos'
mysticProjectName	str	23	'14nm-FinFET-Mystic-nmos'
drain_con	str	5	'drain'
gate_con	str	4	'gate'
source_con	str	6	'source'
bulk_con	str	9	'substrate'
ErrorMethod	str	4	'rmsd'
eps	float	1	0.001
IdVgld	SimData	1	...

Double-clicking `SimData` objects shows the internal `Data` objects contained within the `SimData` object. The contents of the object can be plotted by clicking the button in the upper-right corner of the Variable Explorer panel.

This is one of many ways of creating plots in the Mystic GUI. For more information, see Plot Area Panel on page 72.

Figure 24 Netlist explorer view in Variable Explorer, where modified parameters are highlighted in green cells



Variable Explorer

```
Name: 'Model', type: Netlist, size: 114
```

	Parameter	Value
nmos1	DEVTYPE	1
	EASUB	4.0727
	NI0SUB	1.1055e+16
	BG0SUB	1.1242
	NC0SUB	2.9951e+25
	PHIG	4.32
	RDSW	50
	RDSWMIN	50
	CDSCD	1e-10
	VSAT	130000
	UA	0.001
	PDIBL2	2e-05
	PCLM	1e-10
	PCLMG	0
	level	72
	VERSION	108
	BULKMOD	1
	CAPMOD	1
	IGCMOD	0
	IGBMOD	0

Plot Area Panel

The Plot Area panel is an interactive canvas that displays the plots created during the execution of the strategy. There are different ways to generate plots in the Mystic GUI (such as using the Variable Explorer panel) and different types of plot can be created.

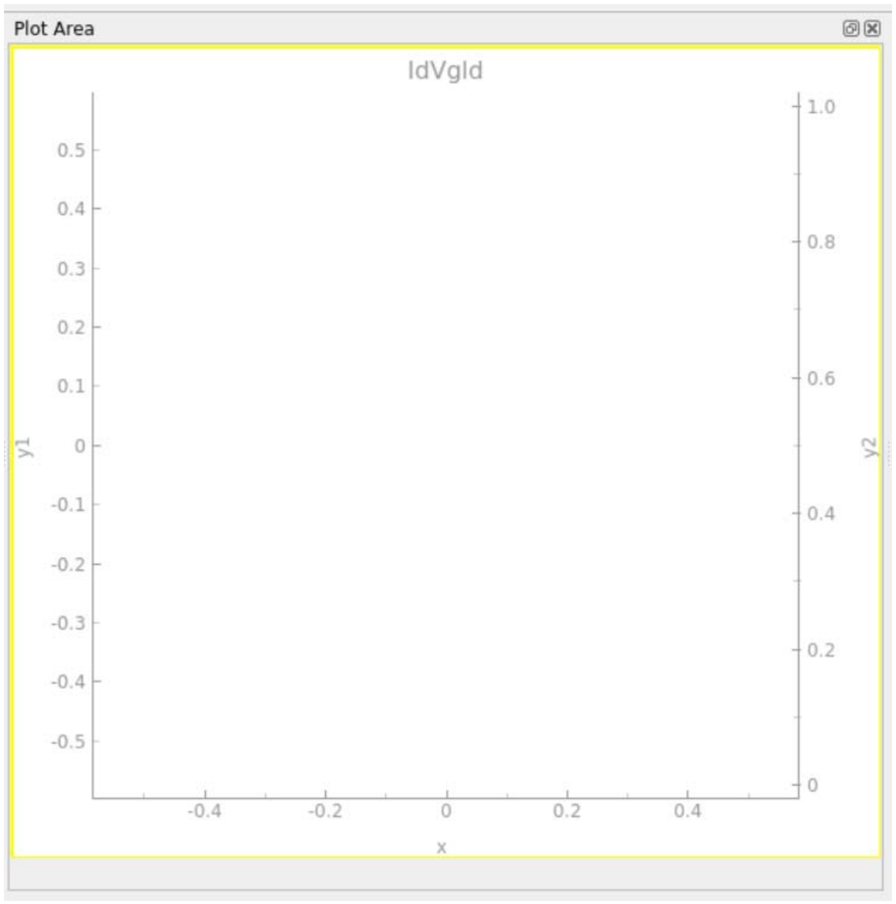
This section describes how to create simple line plots in the Console panel using the built-in plotting commands and Mystic `SimData` objects. Plots can also be generated automatically and updated by the Mystic tool as it runs an extraction (see Running a Strategy Interactively on page 79). For more information about different plot types, consult the Mystic GUI section in the Mystic API documentation.

You can create a plot by entering the `addFigure` command in the Console panel as follows:

```
fig = addFigure('IdVgld')
```

This command creates an empty plot in the Plot Area panel (see Figure 25) and returns a `FigureWidget` object that can be used to populate the plot with data.

Figure 25 Plot Area panel showing empty plot



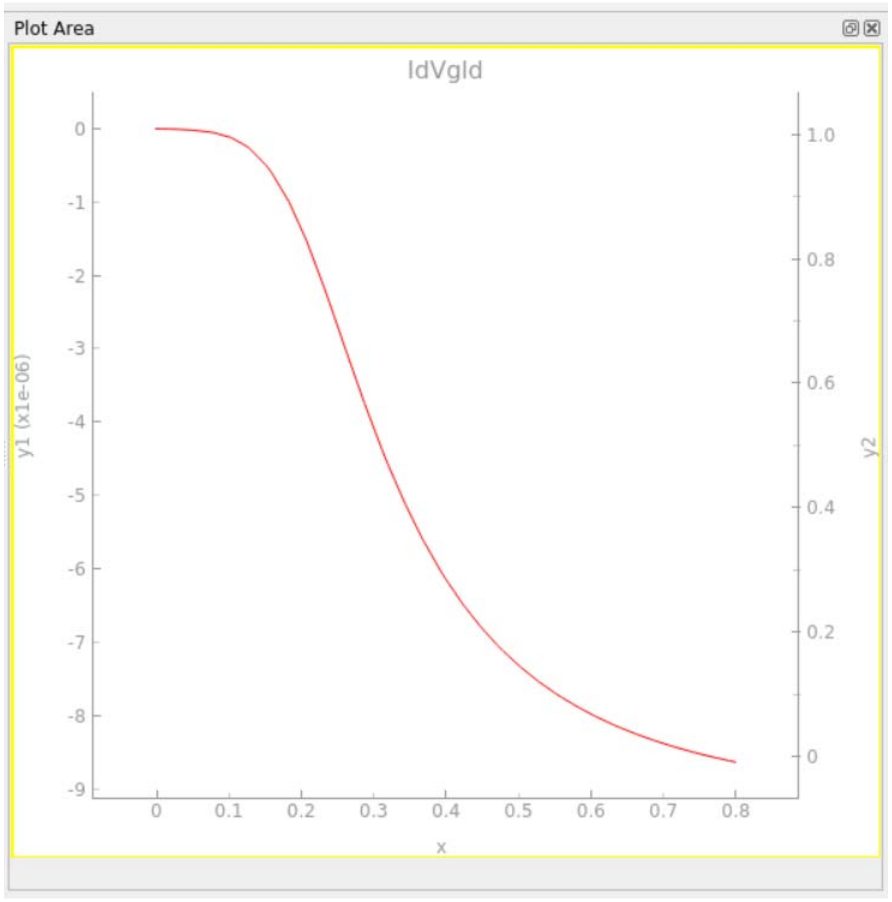
The empty plot has a left and right y-axis allowing you to display two different data series on the same plot or to apply different scales to each axis, for example, a log10 scale. For more information about the axis properties, see Plot Parameters Panel on page 76.

Before adding data, first, you must decide which data you want to visualize. For this example, assume that a low-drain I_d–V_g `SimData` object has been loaded into the environment already using the `mdb.Load` command. This data can be plotted directly using the `FigureWidget.plot` command as follows:

```
series = fig.plot(IdVgld)
```

The contents of the `SimData` object are plotted as shown in Figure 26.

Figure 26 Simple low-drain I_d–V_g line plot



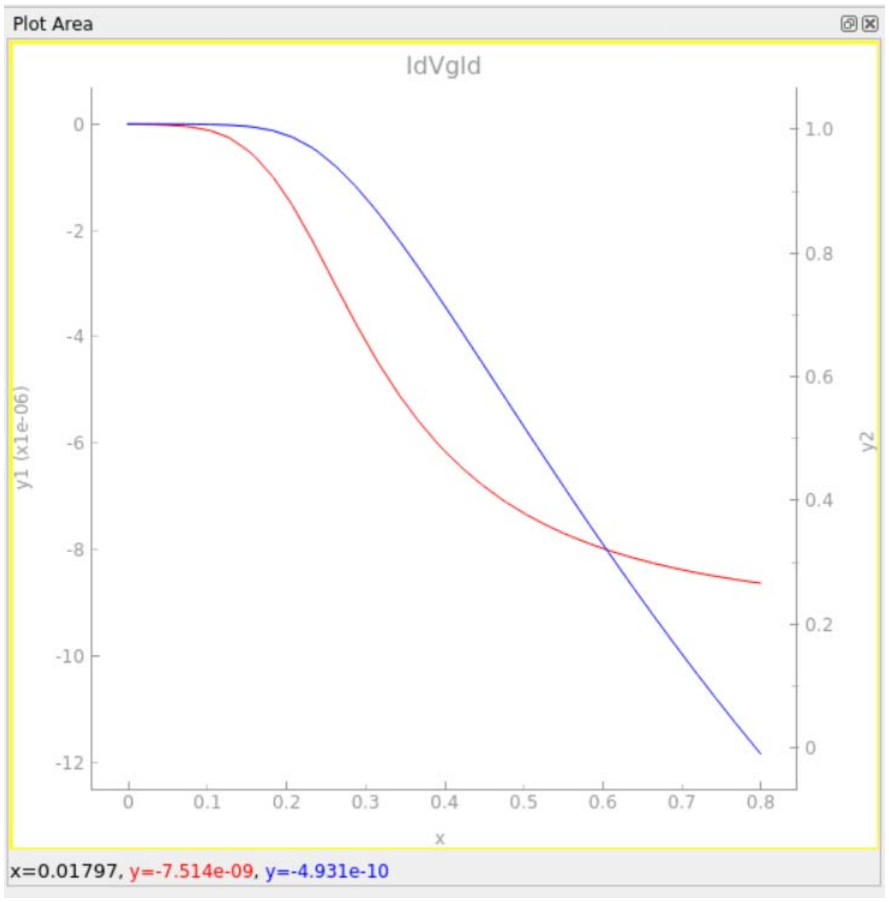
The `FigureWidget.plot` command returns a `PlotSeries` object, which can be used to update the curve properties. You can also do this by using the Plot Parameters panel (see Plot Parameters Panel on page 76).

`SimData` objects should contain both target data and fit data, and so two series, `IdVgld0` and `IdVgld1`, have been created in the Plot Parameters panel, but only one curve has been displayed on the plot. In this example, the `IdVgld` object has been freshly loaded from the database, so no fit data is present. You can calculate an initial error and update the data on the plot as follows:

```
ExtractionUtils.PrintErrors("IdVgld", IdVgld, Simulator,
                                Model, ErrorMethod)

series.updateData(IdVgld)
```

The fit data should now appear on the plot as shown in Figure 27.

Figure 27 Line plot for low-drain I_d–V_g data showing both target data and fit data



x=0.01797, y=-7.514e-09, y=-4.931e-10

You can select a series on the plot by clicking it, which highlights both the series and its associated menu in the Plot Parameters panel.

There are several zoom and pan operations you can apply to a plot:

- To uniformly zoom in to or out of a plot, use the mouse wheel.
- For a free zoom that affects each axis independently, right-click and drag in any direction.

- Moving left and right zooms in to the x-axis, and moving up and down zooms in to the y-axes.
- To zoom only one axis, move the cursor on to that axis, and then hold the right mouse button and drag along the direction of the axis.
- To pan around the data freely, hold the left mouse button and drag on the plot.
- To pan a single axis, hold the left mouse button on the axis.
- At any point, you can auto-scale the plot and return to the initial zoom by clicking the **A** button in the lower-left corner of the plot.

To remove a series from a plot, use the `FigureWidget.removeSeries` command. Plots can be removed from the Plot Area panel by using the `removeFigure` or `removeFigures` commands as follows:

```
removeFigure('IdVgld')
```

For a more comprehensive description of all the plotting commands, consult the Mystic GUI section in the Mystic API documentation.

When multiple plots are added to the Plot Area panel, they are arranged in a grid fashion, which you can scroll through. By default, a 2×2 grid of plots is visible. If more plots are added, then a scroll bar appears, allowing you to move between plots.

To adjust the number of plots that are shown without scrolling:

1. Choose **Edit** > **Preferences**.

   The Preferences dialog box opens.

2. In the left pane, click **Plots**.
3. Under Plot Grid, select **Auto Adjust**.

   With this option selected, the number of visible plots changes automatically as the size of the main window changes, in order to show the plots with the best aspect ratio.

4. Click **OK** or **Apply**.

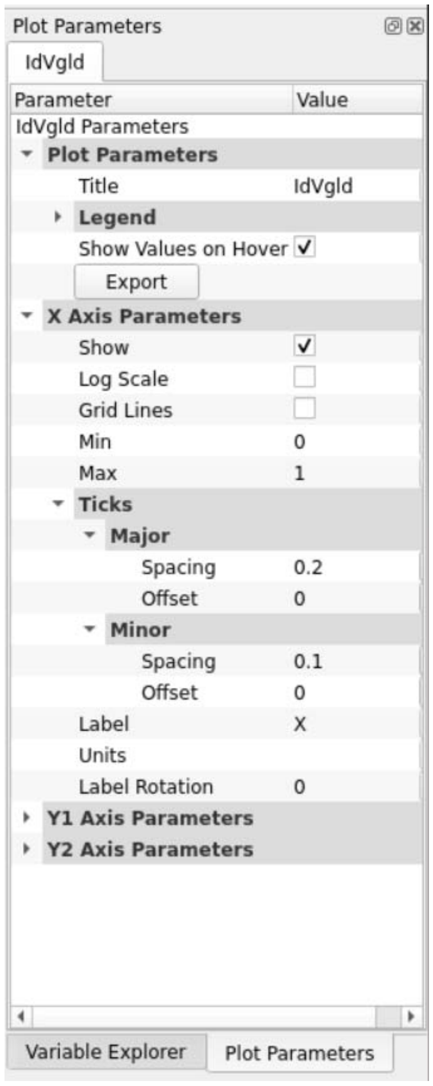
You can make a plot full screened by clicking the four arrows in the lower-left corner of the Plot Area panel. When the Mystic GUI is full screened, you can cycle through the plots using the tabs on the Plot Parameters panel.

Plot Parameters Panel

The Plot Parameters panel allows you to update the properties of all plots in the Plot Area panel. One tab on the Plot Parameters panel is created per plot.

When a plot is first created, a new tab is created in the Plot Parameters panel titled with the name of the plot. Before a curve is created, the tab contains menus for plot parameters and axis parameters as shown in Figure 28.

Figure 28 Plot Parameters panel showing a tab for an empty plot

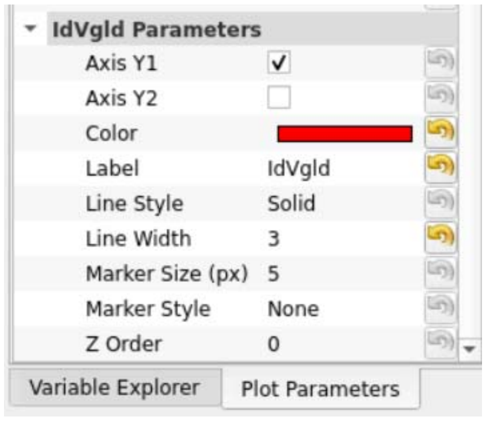


Parameter	Value
IdVgld Parameters	
▾ **Plot Parameters**	
Title	IdVgld
▸ **Legend**	
Show Values on Hover	✓
▾ **X Axis Parameters**	
Show	✓
Log Scale	☐
Grid Lines	☐
Min	0
Max	1
▾ **Ticks**	
▾ **Major**	
Spacing	0.2
Offset	0
▾ **Minor**	
Spacing	0.1
Offset	0
Label	X
Units	
Label Rotation	0
▸ **Y1 Axis Parameters**	
▸ **Y2 Axis Parameters**	

When the tab for a plot is selected, the corresponding plot in the Plot Area panel has a yellow border as shown in Figure 25, Figure 26, and Figure 27.

When a series is added to the plot, a new menu is created for that series. The contents of the menu depend of the type of series that has been plotted. Figure 29 shows the parameter structure for a standard scatter/line series.

Figure 29   Plot Parameters panel showing menu for a scatter/line series



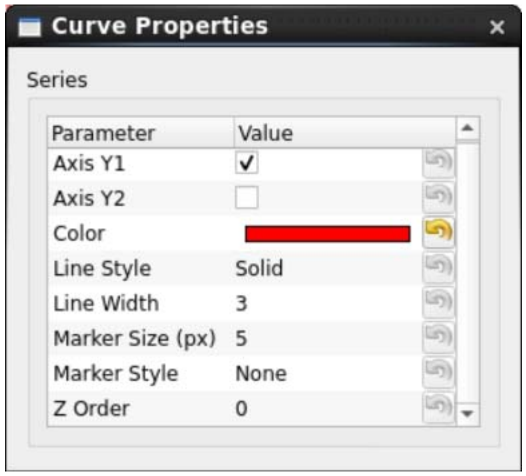
IdVgld Parameters

Axis Y1	✓
Axis Y2	☐
Color	🟥
Label	IdVgld
Line Style	Solid
Line Width	3
Marker Size (px)	5
Marker Style	None
Z Order	0

Changing any of these fields updates the series on the plot immediately. The menu for a series opens automatically when you click a series on the plot.

You can edit the properties of multiple curves simultaneously by highlighting multiple menus, and then right-clicking and choosing **Curve Properties**. Any properties changed in the Curve Properties dialog box apply to all of the selected curves (see Figure 30).

Figure 30   Curve Properties dialog box for editing the properties of multiple curves simultaneously



Parameter	Value
Axis Y1	✓
Axis Y2	☐
Color	🟥
Line Style	Solid
Line Width	3
Marker Size (px)	5
Marker Style	None
Z Order	0

Usage Examples

This section describes some common uses of the Mystic GUI in realistic Mystic examples, which focus specifically on plotting rather than the strategy itself. For more information and access to demonstration projects, contact your local TCAD support team.

Running a Strategy Interactively

When running a strategy in the Mystic GUI, you can monitor the quality of the fit of the model against the target data in real time as the strategy progresses. You can do this by making some minor updates to an existing strategy file.

First, you must identify the target data you want to monitor throughout the strategy. This is usually the target against which the best overall fit is required by the end of the strategy. The identified target or targets can then be tagged as interactive as follows:

```
IdVg = mdb.Load(ivar="vgate", dvar="idrain",
                project=targetProjectName)
IdVgld = IdVg.FilterData('eq', 0.05, 'bias.drain')

IdVgld.interactive = True
IdVgld.name = 'IdVgld'
```

One plot is created in the Plot Area panel for each piece of interactive data, using the given name containing both the target data and fitted data. For example, if I_d–V_g and I_d–V_d data are both present and tagged as interactive, then two separate plots are created that are both updated in real time. In this case, you are working on a low-drain I_d–V_g strategy, so you have only one target.

As the strategy runs, the interactive targets are updated at the end of every optimization stage (`ExtractionUtils.DoStage`), even if the interactive object is not the target for that stage.

A snapshot of an interactive target can be taken at any point throughout the extraction by calling `ExtractionUtils.PrintErrors` with that target. The label passed to `ExtractionUtils.PrintErrors` dictates how the snapshot is displayed. If the label passed *matches* the name of the object, that is:

```
ExtractionUtils.PrintErrors("IdVgld", IdVgld, Simulator,
                            Model, ErrorMethod)
```

then the existing fit data is updated. The next time that `ExtractionUtils.DoStage` is called, this snapshot is overwritten by new results.

If the label *does not match* the name of the object, that is:

```
ExtractionUtils.PrintErrors("Initial", IdVgld, Simulator,
                            Model, ErrorMethod)
```

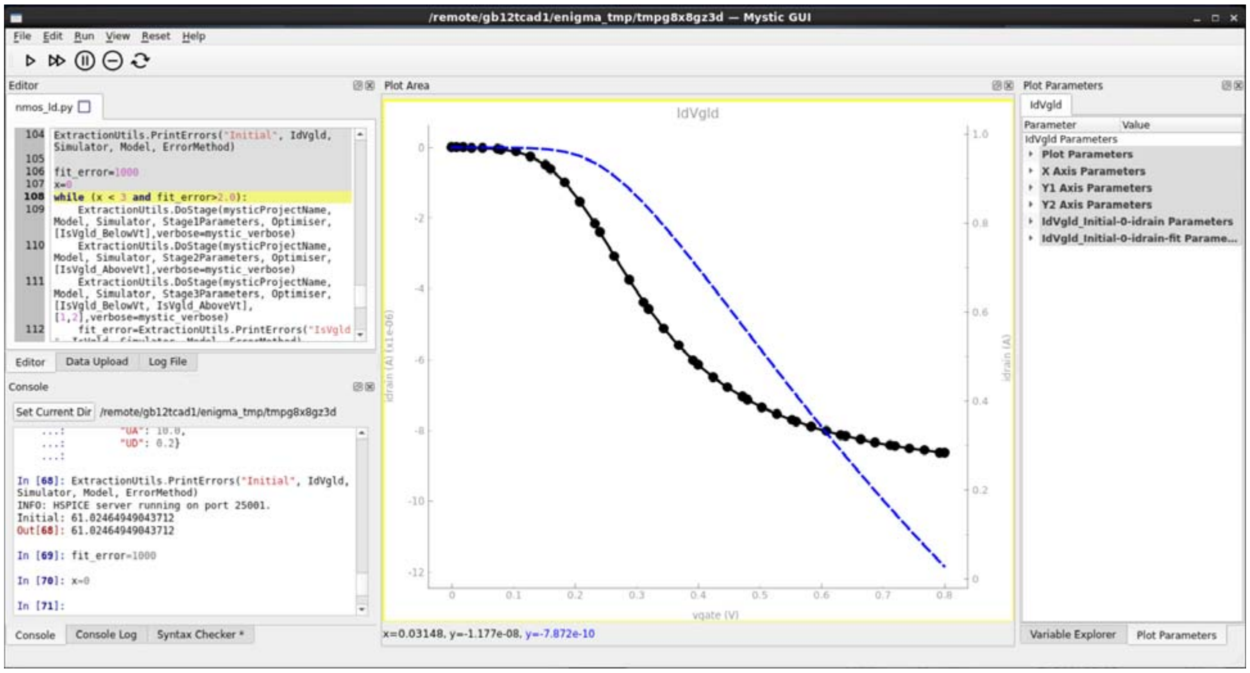
then a new series is created on the plot with the specified label. This series remains unchanged for the duration of the extraction unless it is updated again using `ExtractionUtils.PrintErrors`. This can be used to create snapshots of certain conditions, such as the initial fit or the best overall fit that has been achieved so far.

Providing a visual reference point for the remainder of the extraction, consider the following low-drain I_d–V_g example strategy:

```
ExtractionUtils.PrintErrors("Initial", IdVgld, Simulator,
                            Model, ErrorMethod)

fit_error=1000
x=0
while (x < 3 and fit_error>2.0):
   ExtractionUtils.DoStage(mysticProjectName, Model, Simulator,
                           Stage1Parameters, Optimizer,
                           [IsVgld_BelowVt])
   ExtractionUtils.DoStage(mysticProjectName, Model, Simulator,
                           Stage2Parameters, Optimizer,
                           [IsVgld_AboveVt])
   ExtractionUtils.DoStage(mysticProjectName, Model, Simulator,
                           Stage3Parameters, Optimizer,
                           [IsVgld])
   fit_error=ExtractionUtils.PrintErrors("IsVgld", IsVgld, Simulator,
                                         Model, ErrorMethod)
   x+=1
```

In this example, an initial snapshot labeled `"Initial"` is created to retain a visual reference to the starting point of the extraction. Going through the example, line by line, you can see what is produced. Figure 31 shows the plot after the first call to `ExtractionUtils.PrintErrors`.

Figure 31   Snapshot of initial conditions of the SPICE model fitting



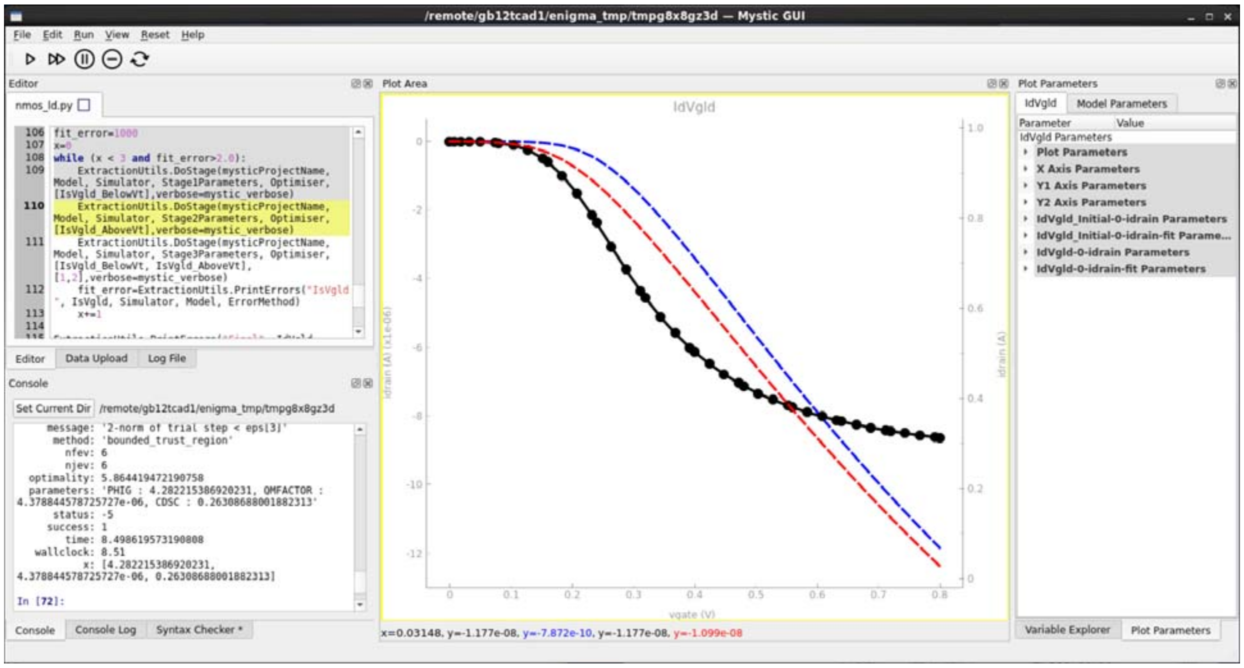
```
ExtractionUtils.PrintErrors("Initial", IdVgld,
Simulator, Model, ErrorMethod)

fit_error=1000
x=0
while (x < 3 and fit_error>2.0):
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage1Parameters, Optimiser,
[IsVgld_BelowVt],verbose=mystic_verbose)
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage2Parameters, Optimiser,
[IsVgld_AboveVt],verbose=mystic_verbose)
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage3Parameters, Optimiser,
[IsVgld_BelowVt, IsVgld_AboveVt],
[1,2],verbose=mystic_verbose)
    fit_error=ExtractionUtils.PrintErrors("IsVgld
```

```
   ...:         "UA": 10.0,
   ...:         "UD": 0.2}
   ...:

In [68]: ExtractionUtils.PrintErrors("Initial", IdVgld,
Simulator, Model, ErrorMethod)
INFO: HSPICE server running on port 25001.
Initial: 61.02464949043712
Out[68]: 61.02464949043712

In [69]: fit_error=1000

In [70]: x=0

In [71]:
```

```
x=0.03148, y=-1.177e-08, y=-7.872e-10
```

The initial conditions are represented by the blue curve. Moving passed the first call to `ExtractionUtils.DoStage`, you see the plot shown in Figure 32.

Figure 32 Real-time fit of the model after the first call to ExtractionUtils.DoStage; initial conditions are shown by blue curve, and current fit is shown by red curve


/remote/gb12tcad1/enigma_tmp/tmpg8x8gz3d — Mystic GUI
File Edit Run View Reset Help
Editor
nmos_ld.py
Editor Data Upload Log File
Console
Set Current Dir /remote/gb12tcad1/enigma_tmp/tmpg8x8gz3d
Console Console Log Syntax Checker *
Plot Area
IdVgld
vgate (V)
idrain (A) (x1e-06)
idrain (A)
x=0.03148, y=-1.177e-08, y=-7.872e-10, y=-1.177e-08, y=-1.099e-08
Plot Parameters
IdVgld Model Parameters
Parameter Value
IdVgld Parameters
Plot Parameters
X Axis Parameters
Y1 Axis Parameters
Y2 Axis Parameters
IdVgld_Initial-0-idrain Parameters
IdVgld_Initial-0-idrain-fit Parame...
IdVgld-0-idrain Parameters
IdVgld-0-idrain-fit Parameters
Variable Explorer Plot Parameters

```
fit_error=1000
x=0
while (x < 3 and fit_error>2.0):
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage1Parameters, Optimiser,
[IsVgld_BelowVt],verbose=mystic_verbose)
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage2Parameters, Optimiser,
[IsVgld_AboveVt],verbose=mystic_verbose)
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage3Parameters, Optimiser,
[IsVgld_BelowVt, IsVgld_AboveVt],
[1,2],verbose=mystic_verbose)
    fit_error=ExtractionUtils.PrintErrors("IsVgld
", IsVgld, Simulator, Model, ErrorMethod)
    x+=1
```

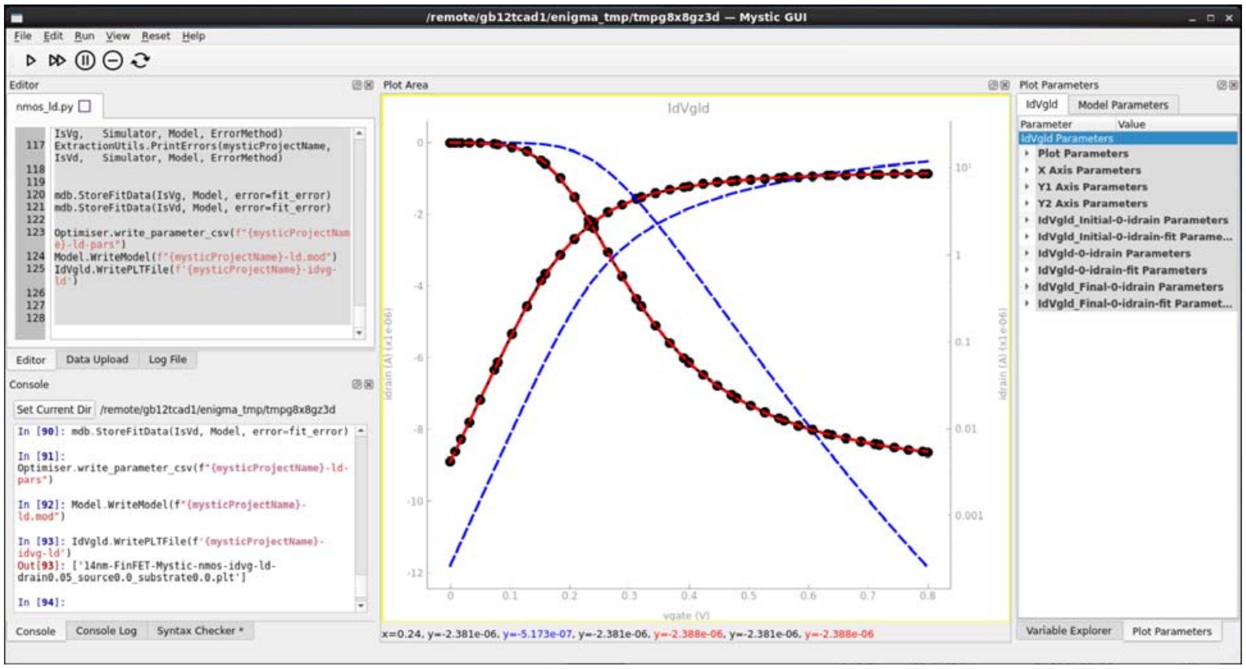
```
      message: '2-norm of trial step < eps[3]'
       method: 'bounded_trust_region'
         nfev: 6
         njev: 6
   optimality: 5.864419472190758
   parameters: 'PHIG : 4.282215386920231, QMFACTOR :
4.378844578725727e-06, CDSC : 0.2630868800188231З'
       status: -5
      success: 1
         time: 8.498619573190808
    wallclock: 8.51
            x: [4.282215386920231,
4.378844578725727e-06, 0.26308688001882313]

In [72]:
```

Now a red curve has been created showing the current status of the fit. The blue curve of the initial conditions remains on the plot. Now, you can run to the end and watch the extraction progress. Figure 33 shows the final result of the strategy.

You can see that the red fit curve has been updated to reflect the current fit of the model.

Now that the strategy has completed, the IPython console remains open for you to further analyze the results, rerun stages, or manually adjust SPICE model parameter values.

Figure 33 Final result of low-drain I_d–V_g strategy


/remote/gb12tcad1/enigma_tmp/tmpg8x8gz3d — Mystic GUI
File Edit Run View Reset Help
Editor
nmos_ld.py
Editor Data Upload Log File
Console
Set Current Dir /remote/gb12tcad1/enigma_tmp/tmpg8x8gz3d
Console Console Log Syntax Checker *
Plot Area
IdVgld
idrain (A) (x1e-06)
vgate (V)
Plot Parameters
IdVgld Model Parameters
Parameter Value
IdVgld Parameters
Plot Parameters
X Axis Parameters
Y1 Axis Parameters
Y2 Axis Parameters
IdVgld_Initial-0-idrain Parameters
IdVgld_Initial-0-idrain-fit Parame...
IdVgld-0-idrain Parameters
IdVgld-0-idrain-fit Parameters
IdVgld_Final-0-idrain Parameters
IdVgld_Final-0-idrain-fit Paramet...
Variable Explorer Plot Parameters

```
IsVg,   Simulator, Model, ErrorMethod)
ExtractionUtils.PrintErrors(mysticProjectName,
IsVd,   Simulator, Model, ErrorMethod)


mdb.StoreFitData(IsVg, Model, error=fit_error)
mdb.StoreFitData(IsVd, Model, error=fit_error)

Optimiser.write_parameter_csv(f"{mysticProjectNam
e}-ld-pars")
Model.WriteModel(f"{mysticProjectName}-ld.mod")
IdVgld.WritePLTFile(f'{mysticProjectName}-idvg-
ld')
```

```
In [90]: mdb.StoreFitData(IsVd, Model, error=fit_error)

In [91]:
Optimiser.write_parameter_csv(f"{mysticProjectName}-ld-
pars")

In [92]: Model.WriteModel(f"{mysticProjectName}-
ld.mod")

In [93]: IdVgld.WritePLTFile(f'{mysticProjectName}-
idvg-ld')
Out[93]: ['14nm-FinFET-Mystic-nmos-idvg-ld-
drain0.05_source0.0_substrate0.0.plt']

In [94]:
```

```
x=0.24, y=-2.381e-06, y=-5.173e-07, y=-2.381e-06, y=-2.388e-06, y=-2.381e-06, y=-2.388e-06
```

Exploring Parameters Interactively

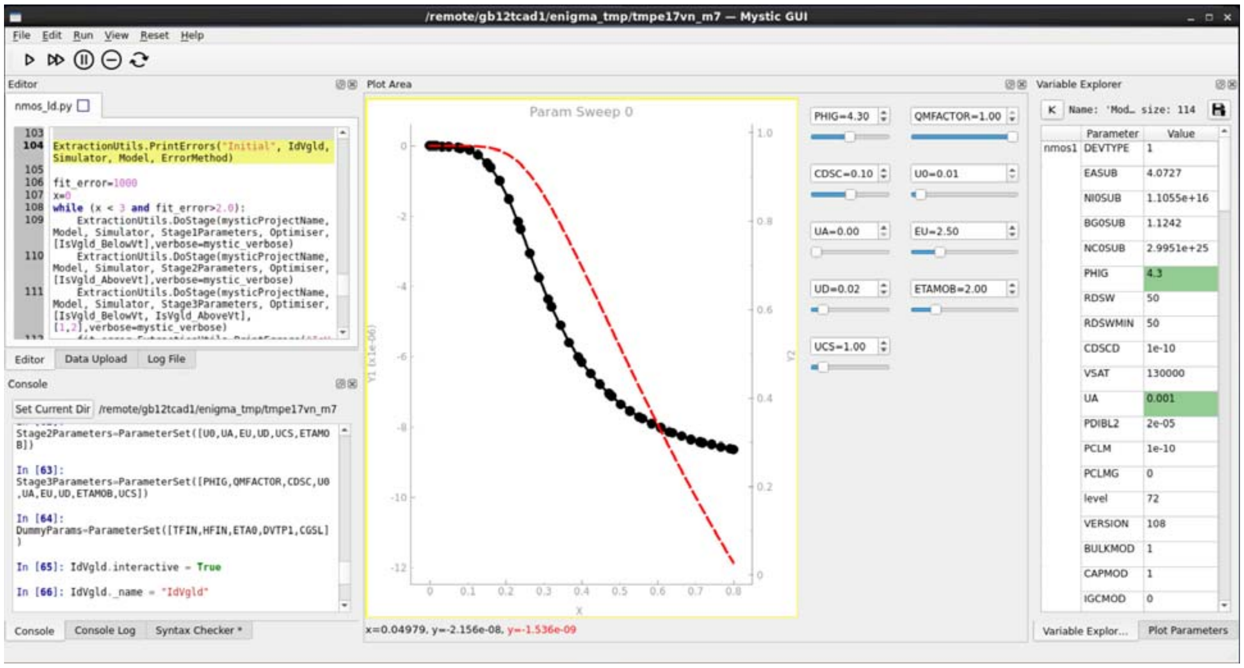
When developing a strategy from the beginning, you can use the Mystic GUI to investigate parameter sensitivity interactively. The `ExtractionUtils.InteractiveParamSweep` utility takes a single `SimData` target and a list of SPICE model parameters and creates an interactive plot where you can adjust parameter values using sliders and watch the response of the model in real time on the plot.

The following example shows an interactive parameter sweep for low-drain I_d–V_g data:

```
pmax = {"QMFACTOR": 1.0,
        "CDSC": 0.2,
        "U0": 0.1,
        "UA": 10.0,
        "UD": 0.2}

ExtractionUtils.InteractiveParamSweep(IdVgld, Simulator, Model,
                [PHIG,QMFACTOR,CDSC,U0,UA,EU,UD,ETAMOB,UCS],
                pmax=pmax, num=101)
```

The optional `pmax` dictionary provides the upper limit of the slider for the parameters. If this is not provided, or certain parameter names do not exist in the dictionary, then the maximum bound supplied in the `OptParam` definition is used. Minimum bounds can also be supplied using an optional `pmin` dictionary. The number of levels between the minimum and maximum bounds of the parameters is determined by the `num` argument. Figure 34 shows the interactive plot produced by `ExtractionUtils.InteractiveParamSweep`.

Figure 34 Interactive parameter plot for low-drain I_d–V_g data


/remote/gb12tcad1/enigma_tmp/tmpe17vn_m7 — Mystic GUI
File Edit Run View Reset Help
Editor
nmos_ld.py
Editor Data Upload Log File
Console
Set Current Dir /remote/gb12tcad1/enigma_tmp/tmpe17vn_m7
Console Console Log Syntax Checker *
Plot Area
Param Sweep 0
PHIG=4.30
QMFACTOR=1.00
CDSC=0.10
U0=0.01
UA=0.00
EU=2.50
UD=0.02
ETAMOB=2.00
UCS=1.00
x=0.04979, y=-2.156e-08, y=-1.536e-09
Variable Explorer
K Name: 'Mod… size: 114
Variable Explor... Plot Parameters

```
ExtractionUtils.PrintErrors("Initial", IdVgld,
Simulator, Model, ErrorMethod)

fit_error=1000
x=0
while (x < 3 and fit_error>2.0):
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage1Parameters, Optimiser,
[IsVgld_BelowVt],verbose=mystic_verbose)
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage2Parameters, Optimiser,
[IsVgld_AboveVt],verbose=mystic_verbose)
    ExtractionUtils.DoStage(mysticProjectName,
Model, Simulator, Stage3Parameters, Optimiser,
[IsVgld_BelowVt, IsVgld_AboveVt],
[1,2],verbose=mystic_verbose)
```

```
Stage2Parameters=ParameterSet([U0,UA,EU,UD,UCS,ETAMO
B])

In [63]:
Stage3Parameters=ParameterSet([PHIG,QMFACTOR,CDSC,U0
,UA,EU,UD,ETAMOB,UCS])

In [64]:
DummyParams=ParameterSet([TFIN,HFIN,ETA0,DVTP1,CGSL]
)

In [65]: IdVgld.interactive = True

In [66]: IdVgld._name = "IdVgld"
```

	Parameter	Value
nmos1	DEVTYPE	1
	EASUB	4.0727
	NIOSUB	1.1055e+16
	BG0SUB	1.1242
	NC0SUB	2.9951e+25
	PHIG	4.3
	RDSW	50
	RDSWMIN	50
	CDSCD	1e-10
	VSAT	130000
	UA	0.001
	PDIBL2	2e-05
	PCLM	1e-10
	PCLMG	0
	level	72
	VERSION	108
	BULKMOD	1
	CAPMOD	1
	IGCMOD	0

When you move each slider, the Mystic tool runs a backend simulation to evaluate the new set of parameters for the given target conditions and updates the plot accordingly. The interactive parameter plot can be used to manually fit a set of parameters for the model or to provide guidance in assembling an extraction strategy for the model based on the sensitivity of parameters to certain sections of the target data.

As can be seen to the right of the plot in Figure 34, each slider has its own menu. This can be used to show or hide the slider, adjust its minimum value, maximum value, and step, or to set the value directly.

When you are satisfied with the quality of the fit, or you want to take a snapshot of the current model parameter values, you can use the model explorer in the Variable Explorer panel to save the model to disk.

Parallel Coordinate Plots

When using the Mystic tool to extract geometry- or temperature-dependent models, the number of data points you are fitting across increases significantly, making it difficult to assess the quality of the model fit across all conditions using standard line plots. You can use parallel coordinate plots to effectively visualize and compare higher-dimensional datasets.

This example demonstrates how to use the Mystic GUI to visualize a SPICE model fitted for a 14 nm FinFET device across multiple gate lengths and fin widths. It assumes the model has been prefitted to a good standard and the example focuses on plotting.

To learn more about fitting across multiple geometries or temperatures using the Mystic tool, contact your local TCAD support team.

The starting point of this example is two `SimData` objects, `IdVgld` and `IdVghd`, representing low-drain and high-drain geometry-dependent data, respectively, for the 14 nm FinFET device. As the model is prefitted, both target data and fit data are already present.

A parallel coordinate plot requires a set of features that can be represented as single points for each geometry. For this FinFET device, you want to evaluate the fit at these metrics: Ioff-lin, Vt-lin, Ion-lin, Ioff-sat, Vt-sat, and Ion-sat. To do this, you must first calculate them using a figure of merit (FoM) transform as follows:

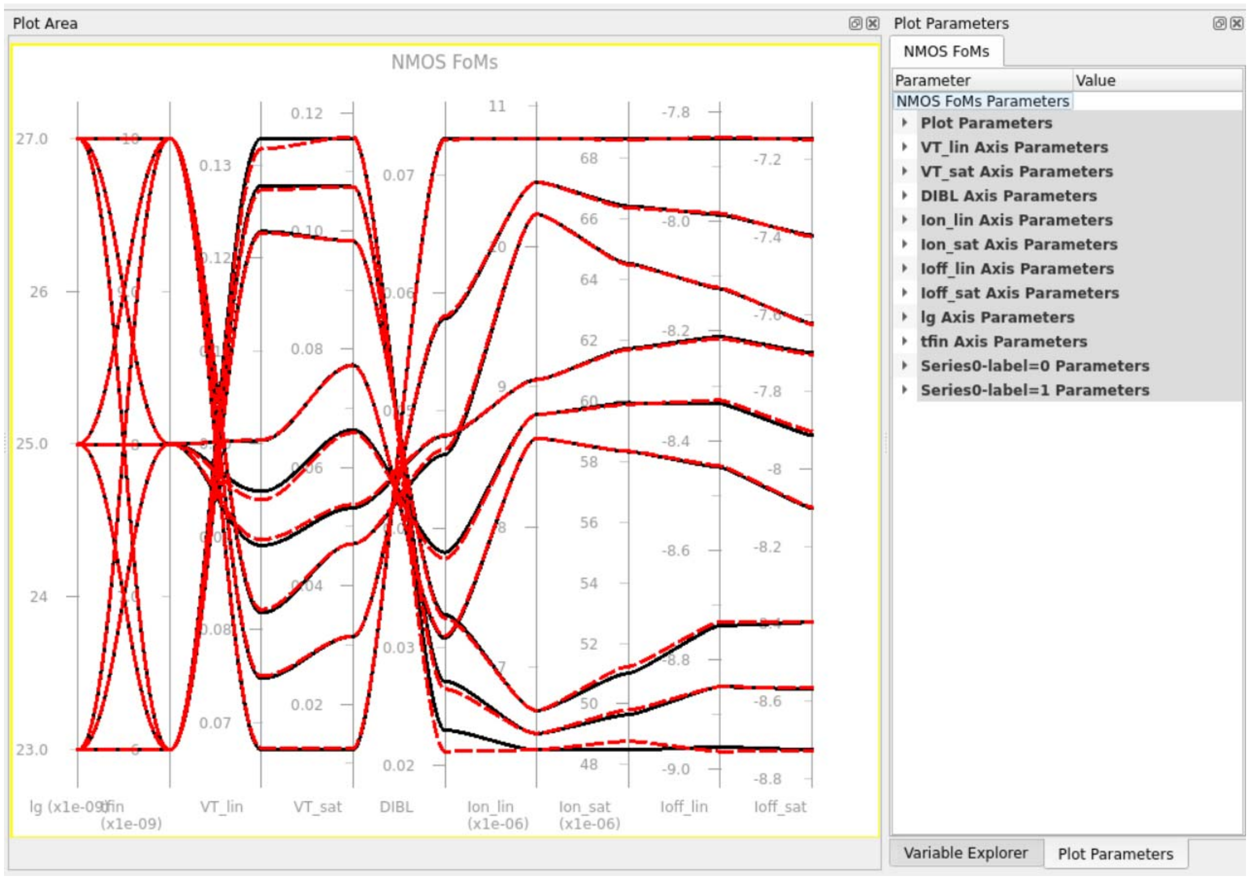
```
ld_foms = IdVgld.MakeFoMTarget(['ioff','vt','ion'])
hd_foms = IdVghd.MakeFoMTarget(['ioff','vt','ion'])
```

Now, you can create the plot and pass the FoM data to the parallel coordinates plotting function:

```
fig = addFigure('NMOS FoMs')

series = fig.plotParallelCoordinates([ld_foms, hd_foms],
                                     suffix=['_lin','_sat'])
```

Figure 35 shows the resulting parallel coordinate plot.

Figure 35 Parallel coordinate plot of geometry-dependent fitting for a 14 nm FinFET device: black=target data and red=fit data



One series is created for the target data (black) and one is created for the fit data (red). You can see from Figure 35 that the model fits well across most metrics.

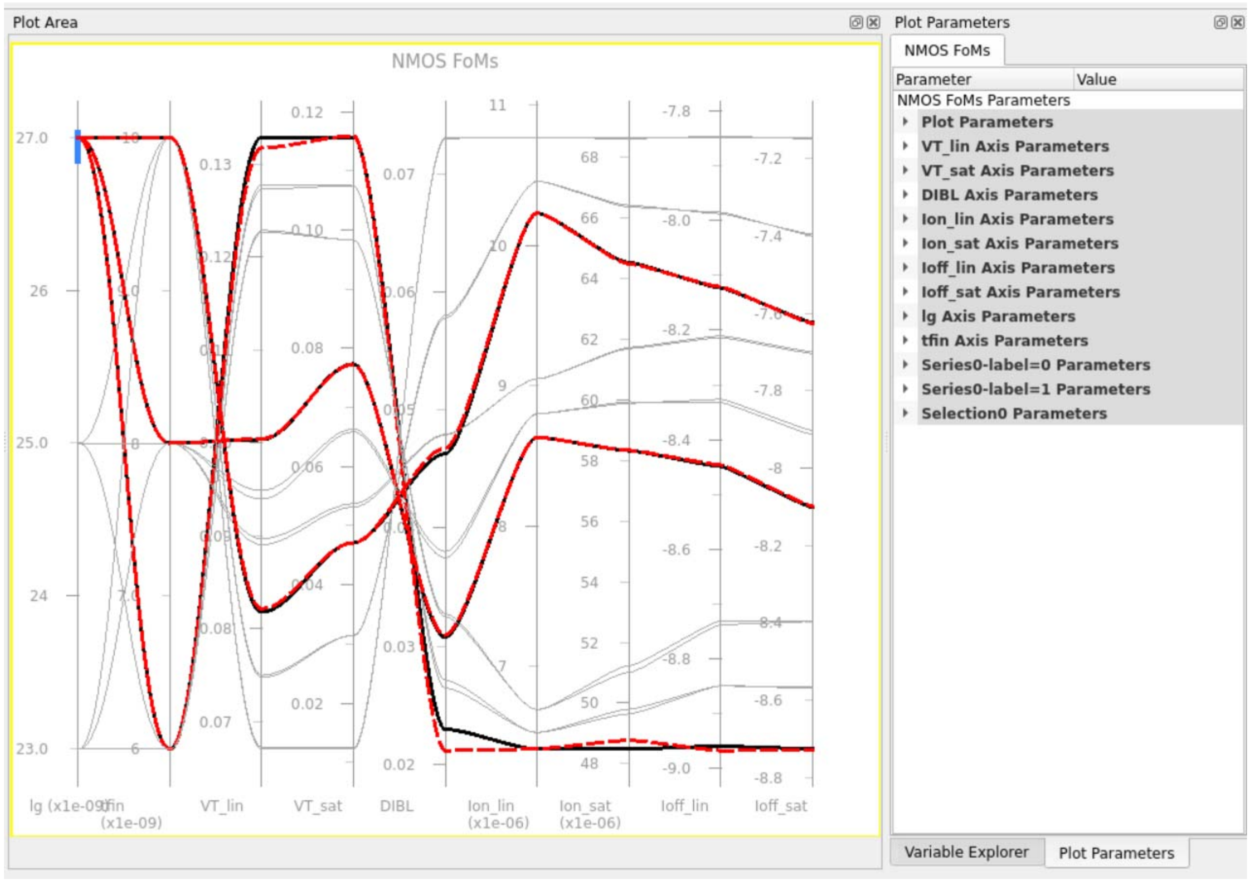
One of the main strengths of parallel coordinate plots is their ability to relate trends in the output metrics to changes in the inputs. When a large amount of data is being analyzed on the plot, it can quickly become difficult to track these patterns in the data. In the Mystic GUI, the visible data on the plot can be filtered, based on selection criteria on each axis.

To add a selection, hold the Shift key and drag one point of an axis to a new point. This filters out the data outside of the selection region.

Figure 36 shows a selection filter placed at 27 nm on the Lg axis. The data at all other gate lengths is gray. The selection can be extended or moved at any time by holding the Shift key while dragging the selection line or by using the selection menu that has been added to the Plot Parameters panel.

Multiple selections can be applied to any of the axes to further filter the data based on as many selection criteria as you want.

Figure 36 Parallel coordinates plot with a selection added at Lg=27 nm



Parameter	Value
NMOS FoMs Parameters	
Plot Parameters	
VT_lin Axis Parameters	
VT_sat Axis Parameters	
DIBL Axis Parameters	
Ion_lin Axis Parameters	
Ion_sat Axis Parameters	
Ioff_lin Axis Parameters	
Ioff_sat Axis Parameters	
lg Axis Parameters	
tfin Axis Parameters	
Series0-label=0 Parameters	
Series0-label=1 Parameters	
Selection0 Parameters	

RF Smith Charts

As described in RF Extraction on page 55, RF devices require the scattering parameters (S-parameters) to be captured accurately. S-parameters come in the form of a 2×2 matrix with both real and imaginary parts and are most commonly plotted on a Smith chart.

You can create Smith charts to compare target and fit S-parameter data for an RF device extraction. This example focuses on how to plot S-parameter data that has been prefitted. For a full demonstration flow of an RF SPICE model extraction, contact your local TCAD support team.

The starting point for this example is a `SimData` object named `S_data` that contains a set of S-parameters at frequencies ranging from 27.7 GHz to 28.2 GHz as shown in Figure 37.

Figure 37 S-parameter data for the example RF device

```
                   s(11)re        s(11)i   ...      s(22)i-fit       errors
frequency                                  ...
2.772000e+10 4.686165e-01 -7.190487e-01   ...   -3.525271e-01 2.539432e-02
2.772600e+10 4.672343e-01 -7.194720e-01   ...   -3.527785e-01 2.540475e-02
2.783200e+10 4.658533e-01 -7.198920e-01   ...   -3.530283e-01 2.541532e-02
2.788800e+10 4.644735e-01 -7.203088e-01   ...   -3.532768e-01 2.542601e-02
2.794400e+10 4.630949e-01 -7.207224e-01   ...   -3.535237e-01 2.543684e-02
2.800000e+10 4.617175e-01 -7.211327e-01   ...   -3.537692e-01 2.544779e-02
2.805600e+10 4.603414e-01 -7.215398e-01   ...   -3.540132e-01 2.545888e-02
2.811200e+10 4.589664e-01 -7.219437e-01   ...   -3.542558e-01 2.547011e-02
2.816800e+10 4.575927e-01 -7.223444e-01   ...   -3.544970e-01 2.548147e-02
2.822400e+10 4.532203e-01 -7.227419e-01   ...   -3.547367e-01 2.549297e-02
2.828000e+10 4.548490e-01 -7.231362e-01   ...   -3.549750e-01 2.550460e-02
```

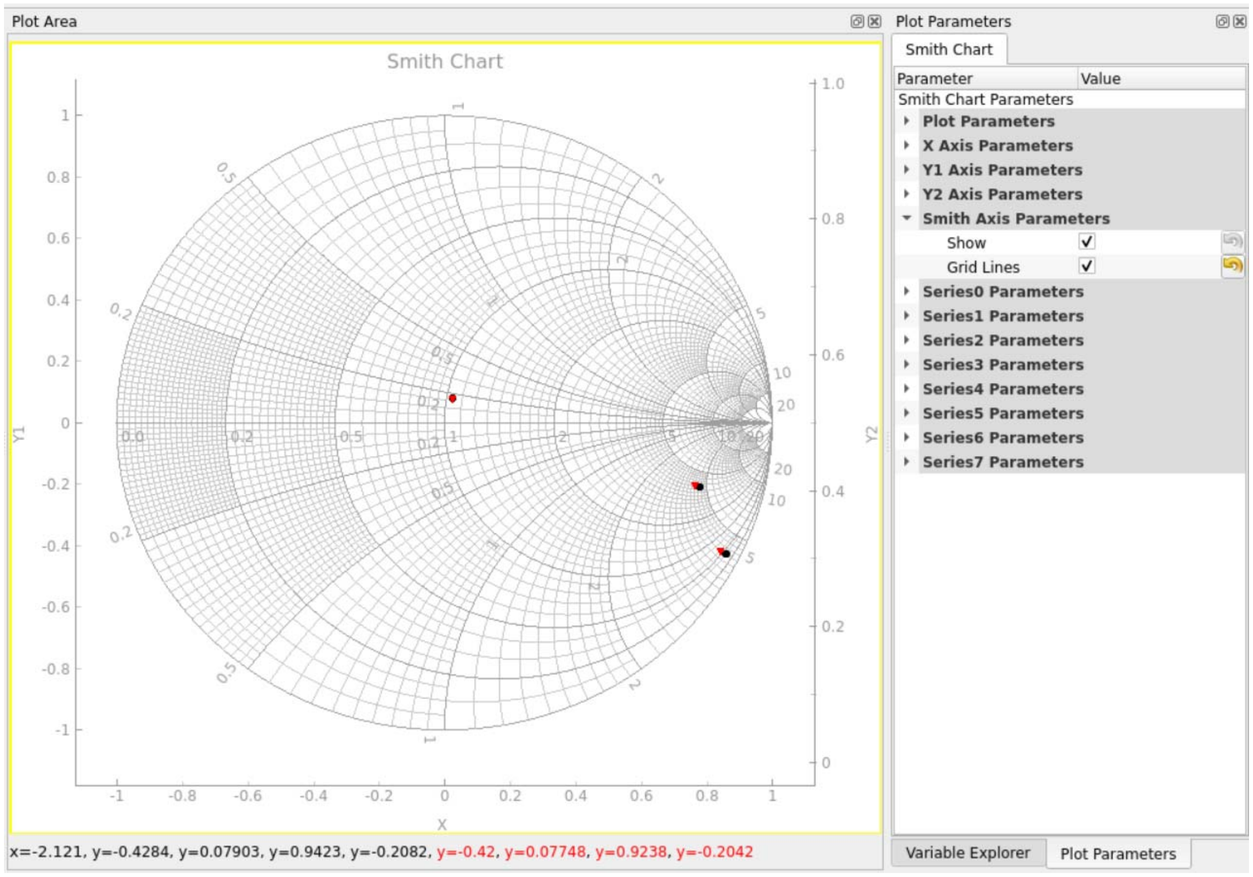
You can create a Smith chart from this data using the following commands:

```
fig = addFigure('Smith Chart')

target = fig.plot(S_data, col='target', mode="smith",
                  colour="#000000", markerStyle='o')

fit = fig.plot(S_data, col=fit, mode="smith",
               colour="#FF0000", markerStyle='Triangle')
```

Figure 38 shows generated plot.

For Smith charts, an extra Smith axis parameter menu is added to the Plot Parameters panel. This menu controls the properties of the circular Smith axis.

Figure 38 Smith chart for example RF device showing target data (black) and fit data (red)



x=-2.121, y=-0.4284, y=0.07903, y=0.9423, y=-0.2082, y=-0.42, y=0.07748, y=0.9238, y=-0.2042

Parameter	Value
Smith Chart Parameters	
Plot Parameters	
X Axis Parameters	
Y1 Axis Parameters	
Y2 Axis Parameters	
Smith Axis Parameters	
Show	✓
Grid Lines	✓
Series0 Parameters	
Series1 Parameters	
Series2 Parameters	
Series3 Parameters	
Series4 Parameters	
Series5 Parameters	
Series6 Parameters	
Series7 Parameters	

References

[1] For more information, go to https://ipython.readthedocs.io/en/stable/interactive/magics.html.

6

Mystic Examples

This chapter presents a step by step guide through an example packaged with the Mystic tool.

Location of Examples

You can find all of the examples in the following directory, in your installation directory:

```
$STROOT/tcad/$STRELEASE/manuals/tcad_spice/mystic_api/examples
```

Note:

Before running any example, ensure that both the Mystic tool and the PrimeSim HSPICE tool are available in your environment and licenses for both tools are available through the `SNPSLMD_LICENSE_FILE` environment variable.

To check whether both tools are available and that you have the required licences, enter the following on the command line:

```
:$ Mystic --version
========================================================================
                          TCAD to SPICE - Mystic

                      Version T-2022.03 for linux64

                  Copyright (c) 2010-2022 Synopsys, Inc.
       This Synopsys software and all associated documentation are
       proprietary to Synopsys, Inc. and may only be used pursuant
     to the terms and conditions of a written license agreement with
       Synopsys, Inc. All other use, reproduction, modification, or
         distribution of the Synopsys software or the associated
                    documentation is strictly prohibited.
========================================================================
Mystic T-2022.03 (9c5934d)
```

For these examples, you will run the Mystic tool on the command line and then use Sentaurus Visual to view the results.

These examples assume you have some familiarity with the Mystic extraction environment and how to structure an extraction strategy. For more information, see Full Strategy on page 40.

14 nm FinFET Example

This example is located in your installation directory as follows:

```
$STROOT/tcad/$STRELEASE/manuals/tcad_spice/mystic_api/examples/
14nmFinFET.zip
```

When you unzip the file in a new directory, you should have the following files and directories:

- `models/nmos.mod`: The initial model card. This is a BSIM-CMG subcircuit model set up to represent a 14 nm FinFET technology. Some default parameter values have been set. For details about the default parameters, refer to the TCAD to SPICE module of the TCAD Sentaurus Tutorial (see TCAD Sentaurus Tutorial: Simulation Projects on page 10).
- `results`: An empty directory where SPICE model extraction results will be written.
- `strategies`: A directory containing three extraction strategies to be run sequentially: a low-drain I_d–V_g strategy, a high-drain I_d–V_g and I_d–V_d strategy, and a C–V strategy. It also contains Sentaurus Visual plotting scripts.
- `targets`: A directory containing the I–V and C–V target data for the extraction.

The extraction is separated into three separate stages: low-drain I_d–V_g, high-drain I_d–V_g and I_d–V_d, and C–V. Splitting up the extraction helps to check the strategy and retain partial results while working on downstream stages.

First Stage: Low-Drain Extraction

The first extraction stage fits the low-drain I_d–V_g target data. To execute this stage, navigate to the unzipped `14nmFinFET` file and run:

```
Mystic strategies/ld_extraction.py --model=models/nmos.mod
```

Mystic should start executing in the terminal as follows:

```
========================================================================
                        TCAD to SPICE - Mystic

                    Version T-2022.03 for linux64

                Copyright (c) 2010-2022 Synopsys, Inc.
     This Synopsys software and all associated documentation are
```

```
          proprietary to Synopsys, Inc. and may only be used pursuant to
           the terms and conditions of a written license agreement with
           Synopsys, Inc. All other use, reproduction, modification, or
             distribution of the Synopsys software or the associated
                       documentation is strictly prohibited.
========================================================================
  Loading:
          Extraction Strategy:     /<path>/strategies/ld_extraction.py
          Initial Model:           /<path>/models/nmos.mod
          Database:                No database loaded.
          SPICE Backend:           hspice
          Optimizer:               BOUNDED_TRUST_REGION

  INFO: HSPICE server running on port 25001.
  Initial: 60.37694266243185

  Fitting Result Data
  -------------------
  parameters = mosmod.PHIG : 4.228635892175941, mosmod.CDSC :
               0.1191443780274296
  iterations = 11
  fit error  = 0.020096712548626192
  converged  = 1
  method     = BOUNDED_TRUST_REGION
  wallclock  = 9.02s
```

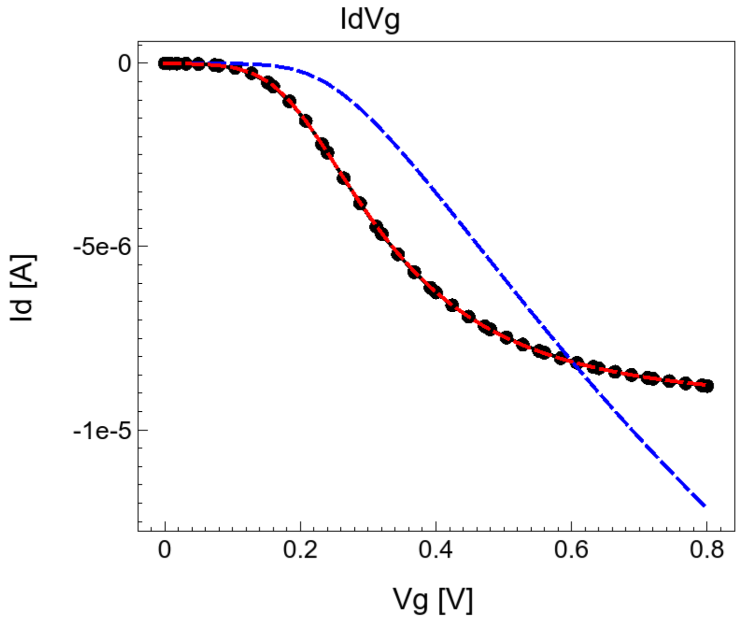
When the strategy has been completed, the results of the low-drain I_d–V_g extraction is written to the `results` directory. The results can be visualized using Sentaurus Visual with the low-drain plotting script as follows:

```
svisual strategies/plot_ld.tcl
```

The final model card is also written to the `results` directory. This will be used as the starting point for the high-drain I_d–V_g and I_d–V_d extraction strategy (see Second Stage: High-Drain and I_d–V_d Extraction on page 92).

Figure 39 shows the results in Sentaurus Visual.

Figure 39 Low-drain I_d–V_g extraction strategy results in Sentaurus Visual



Second Stage: High-Drain and I_d–V_d Extraction

The high-drain I_d–V_g and I_d–V_d extraction strategy uses the output model card from the low-drain I_d–V_g extraction strategy as a starting point. To run this strategy, use the following command:

```
Mystic strategies/hd_extraction.py --model=models/nmos-ld.mod
```

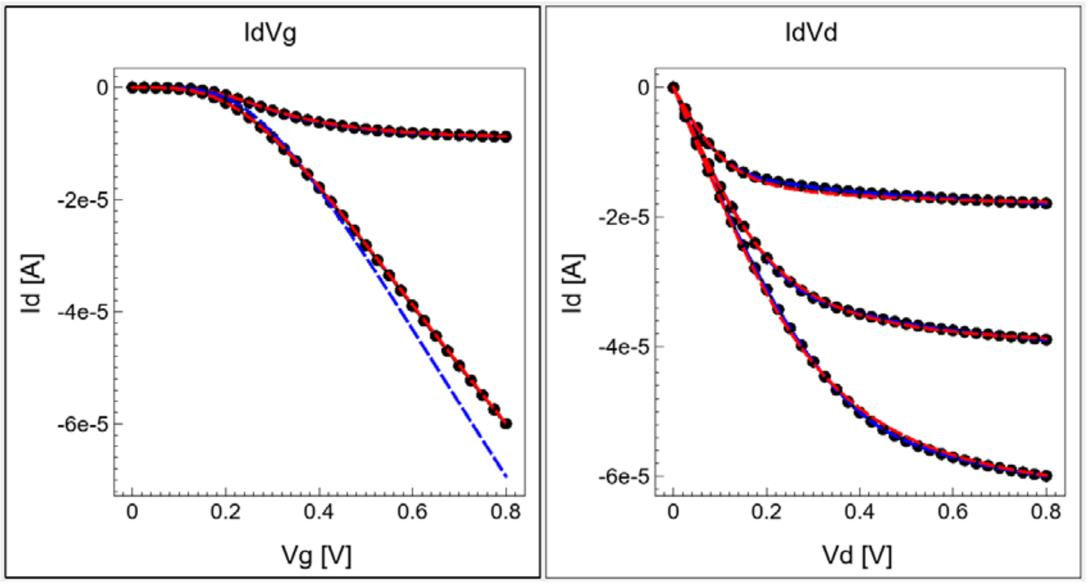
Again, the Mystic tool will start running in the terminal. This extraction strategy is much more complex than the low-drain I_d–V_g extraction strategy, fitting a larger set of parameters to the full set of I–V curves. It will take longer to execute.

When the extraction has completed, you can visualize the results in Sentaurus Visual using the high-drain plotting script as follows:

```
svisual strategies/plot_hd.tcl
```

There will be two plots, for I_d–V_g and I_d–V_d data, as shown in Figure 40.

Figure 40 Initial conditions of high-drain strategy in Sentaurus Visual



You can see from the initial conditions that the low-drain I_d–V_g data is well fitted from the low-drain extraction strategy. The results of this extraction stage are again stored in the `results` directory. The fitted model will supply the initial conditions for the third and final extraction stage (see Third Stage: C–V Extraction).

Third Stage: C–V Extraction

The C–V extraction strategy uses the output model card from the high-drain I_d–V_g and I_d–V_d strategy as a starting point.

To run this strategy, use the following command:

```
Mystic strategies/cv_extraction.py --model=models/nmos-hd.mod
```

The C–V extraction strategy has the same basic structure as the two I–V strategies but operates on gate capacitance target data. The C–V targets require no special treatment, from a strategy perspective, as the Mystic tool inherently knows how to handle them on the backend.

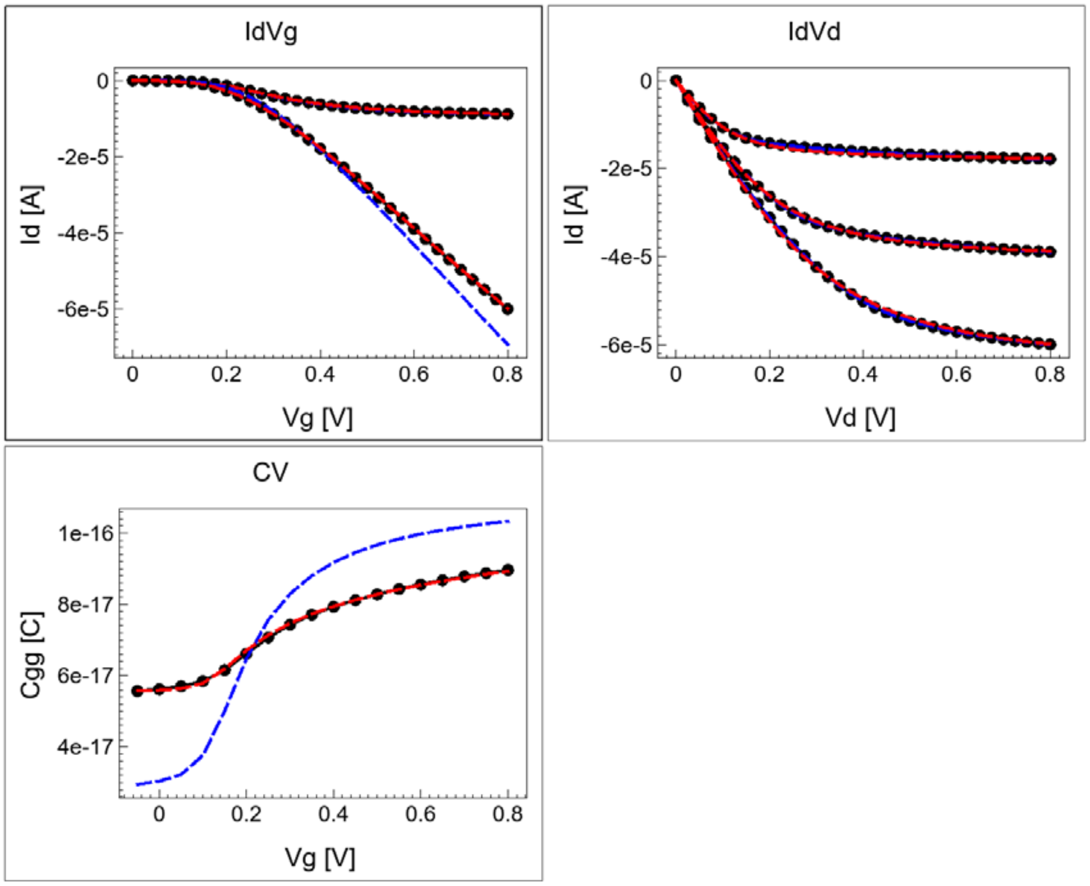
When the strategy has completed, you can visualize the results as follows:

```
svisual strategies/plot_cv.tcl
```

Now, all three targets used throughout the extraction procedure are displayed in Sentaurus Visual as shown in Figure 41.

The final fitted NMOS model card has been written to `models/nmos-final.mod`. This can now be used in any PrimeSim HSPICE circuit simulation.

Figure 41 Results of capacitance extraction strategy in Sentaurus Visual



Glossary

DoE
Design-of-experiments.

FoM
Figure of merit.

LUT
Lookup table.

MOSRA
MOSFET reliability analysis.

PCA
Principal component analysis.

PWL
Piecewise linear.

RMS
Root mean square.

RMSD
Root mean square deviation.

Index