Sentaurus™ Workbench Optimization Framework User Guide

Version T-2022.03, March 2022

Contents

About This Guide

This user guide describes the Optimization Framework that is part of Synopsys® Sentaurus™ Workbench Advanced.

For additional information, see:

- The TCAD Sentaurus release notes, available on the Synopsys SolvNetPlus support site (see Accessing SolvNetPlus on page 7)
- 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.
Menu > **Command**	Indicates a menu command, for example, **File** > **New** (from the **File** menu, choose **New**).

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 Optimization Framework

This chapter presents an overview of the Optimization Framework.

Functionality of the Optimization Framework

The Optimization Framework is a batch tool designed to extract efficiently general information about TCAD simulations. For example, it is used to determine a parameter set that satisfies given design specifications and to analyze how parameter variations affect device behavior.

It uses batch tools of Sentaurus Workbench to run simulations automatically in the context of TCAD simulation projects. It provides the following functionality:

- *Optimization* allows you to find an optimal parameter set for given target specifications. Multiple optimization targets can be combined, with weights specifying their relative importance. Several algorithms are available to solve optimization problems.
- *Screening analysis* displays model coefficients to illustrate the impact of different parameters on the simulation responses, helping you to determine the most relevant ones.
- *Sensitivity analysis* determines how small changes to a given parameter affect simulation responses.

The Optimization Framework provides a Python environment that allows you to define and use custom algorithms that address specific tasks. All the components of the Optimization Framework are available in this environment and can be combined to develop an optimization strategy. For example, a screening analysis can be combined with an optimization and, through the screening analysis, the important target parameters are found, and then only these are optimized.

The Optimization Framework interacts with batch tools of Sentaurus Workbench for setting up and running simulations (Sentaurus Workbench *experiments*). It takes advantage of the job scheduler of Sentaurus Workbench to speed up simulations using distributed, heterogeneous computing resources. The open architecture and the graphical user interface (GUI) of Sentaurus Workbench allow the Optimization Framework to be used for a wide range of purposes.

Basic Concepts and Terminology

This section describes some common terms relevant to understanding the Optimization Framework:

- A *parameter* is a scalar variable, defined in the Sentaurus Workbench project, that modifies the simulation flow, which allows you to define families of similar simulations. It has a finite value that is defined inside a certain domain. There are different parameter categories:
    - *Optimization parameters* are fitted during the optimization process.
    - *Split parameters* define an **initial** family of experiments (for example, conditions in which the target data was obtained experimentally), but these parameters are not optimized. Their values must not be modified while an optimization is running.
    - *State parameters* are other parameters for experiments that do not define an initial project split. Like split parameters, you must not modify the values of state parameters while an optimization is running, and they are not optimized. However, depending on their purpose, you can vary state parameters so the Optimization Framework can calculate specific targets. For example, you can change the drain voltage to calculate separate I_d–V_g curves.
- A *response* is a variable or simulation output that describes, for example, device or circuit behavior. A response could be the error of the threshold voltage for the target device with respect to the required value.
- An *experiment* or a *parameter setting* is a tuple with one value for each parameter of the Sentaurus Workbench project that defines a particular instance of the simulation flow.
- A *family of simulations* is a set of split simulations that have the same set of optimization parameters (but a different set of split parameters). An *evaluation* is defined as all the simulations required to compute a given family of simulations.
- A *scenario* is a particular subtree of experiments in a Sentaurus Workbench project. Scenarios can overlap, that is, a particular node or path can be a part of more than one scenario. When the Optimization Framework submits a new set of experiments, all of them are added to a new scenario or an existing scenario. In Sentaurus Workbench, scenarios can be run and edited independently.
- A *strategy* is a sequence of actions used to obtain information about the relationship between the parameters and responses under consideration.

- A *task* is a specific action that the Optimization Framework performs. For example, a task could be a screening analysis or a local optimization of a set of parameters. The execution of each task defines the end of a *stage* in an optimization strategy.

Running the Optimization Framework

You can run the Optimization Framework in different ways. The optimization Python script can be specified for:

- The entire Sentaurus Workbench project by choosing **Optimization** > **Edit Python Input** from the GUI of Sentaurus Workbench
- The Optimizer tool, which is a part of the project simulation flow, by right-clicking the Optimizer tool icon and choosing **Edit Input** > **Edit Python Input**

To run the Optimization Framework for a script external to a project, from the GUI of Sentaurus Workbench, choose **Optimization** > **Run**.

Alternatively, on the command line, enter:

```
genopt <options> <project_dir>
```

where `<project_dir>` is the path to the Sentaurus Workbench project to be executed. The Optimization Framework (`genopt`) uses the Python script `genopt.py` in the `<project_dir>` directory as the input file.

The Python script contains the description of the optimization process to be performed. It can be accessed from Sentaurus Workbench by choosing **Optimization** > **Edit Python Input**.

The following command-line options are available:

```
-h[elp]:    Displays this help message
-v[ersion]: Displays version information
-verbose:   Displays additional initialization and loading information
-q[ueue] "queue name": Submits all the jobs to a queue
-reset:     Resets optimization project to its initial task with no
            submission
-restart "scenario": Restarts optimization project from a saved point
                     (Restart_* scenario)
-inline:    Launches optimization as a standalone tool
```

If the optimization script is part of a project simulation flow, then the Optimizer tool is in the project and works like any other simulation tool such as Sentaurus Process, Sentaurus Device, and Sentaurus Visual. In this case, the optimization script is one of the input files to the tool and is accessed as usual.

In addition, there is a second input file that describes which tool instances and parameters must be optimized.

2

Using the Optimization Framework

This chapter describes how to use the Optimization Framework.

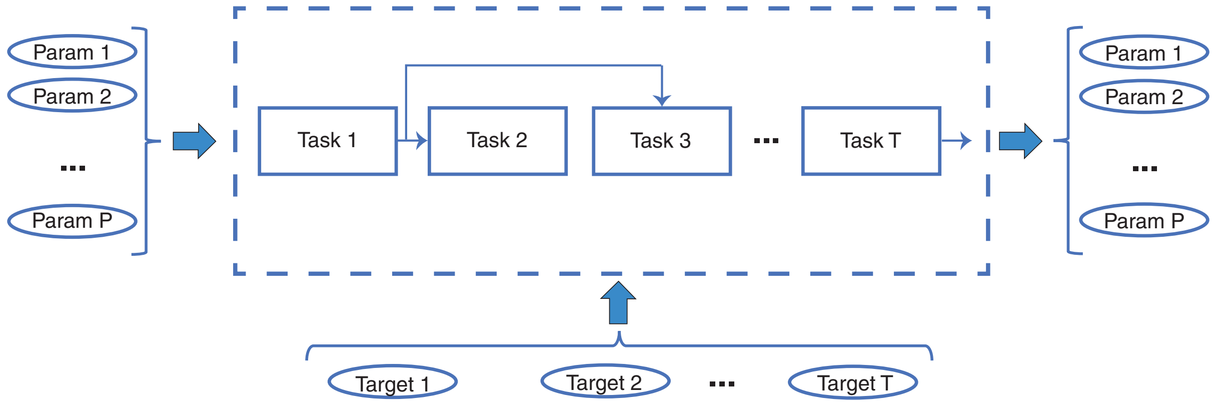
Structure of an Optimization

The main object that links all relevant elements in the Optimization Framework is the strategy. A strategy usually requires the evaluation of several combinations of different parameter sets and acts in accordance to the different response values. These evaluations are performed through an external simulation process invoked and coordinated by Sentaurus Workbench.

More than one task can be performed in a single strategy. Strategies use the evaluations and information obtained from different tasks (responses) and organize them to achieve the required result. For example, if a screening task selects some optimization parameters, then a strategy can instruct subsequent tasks to only use those. In addition, if a task requires the evaluation of a family of simulations that were already evaluated by a previous task, then the family is not reevaluated and the values are recovered.

Figure 1 shows a high-level block diagram of an optimization process: initial parameters and optimization targets are inputs to a strategy (dashed box), and the strategy uses them in a sequence of possibly dependent tasks and outputs the optimal parameter values.

Figure 1 Block diagram of the structure of an optimization



Input Files

The input file `genopt.py` defines an optimization strategy possibly composed of several tasks. This file is a Python script with some additional classes and objects *injected* into the environment related to the Optimization Framework, including the following:

- `optimizer` is the main object of the Optimization Framework. Internally, it drives the optimization process and manages the Sentaurus Workbench project. Optimization tasks are invoked using methods of this object.
- `OptParam` and `TOptParam` are classes that represent optimization parameters. They manage parameter value, bounds, and transforms (`TOptParam`).
- `DOptParam` is a class that represents discrete optimization parameters.
- `OptTarget` and `OptConstraint` are classes that define targets and constraints for the optimization process. In general, they need the response (simulation output, possibly postprocessed) and a target value for the simulation output. You can specify additional properties such as a weight.

Following the syntax of Python scripts, the input file of the Optimization Framework is case sensitive and indentation must be consistent. Comments start with the hash character (#) and can be appended to any line.

The script is interpreted sequentially, so every variable must be defined before it is used. Apart from this, the order in which parameters, targets, and strategies are defined is arbitrary.

Defining Optimization Parameters

Optimization parameters must be defined as Sentaurus Workbench parameters in a project. Their values are modified as the optimization progresses (see Optimization Parameters on page 77).

The syntax to define an optimization parameter is the following:

```
p = OptParam(name, value [, minval, maxval, scale])
```

The syntax to define an optimization parameter if a transform must be applied is the following:

```
p = TOptParam(name, value [, minval, maxval, transform])
```

Parameters are defined by a `name`, which must correspond to the name of a parameter as defined in the Sentaurus Workbench project (case sensitive), and their current value, specified with `value`. The range of values that it can take can be specified by a minimum value and a maximum value.

Finally, you can specify a scaling factor (`OptParam`) or a transform (`TOptParam`) to simplify the optimization problem. It is usually recommended to define a scaling factor so that the values are in the order of unity to facilitate the optimization process or, if you know some specific dependence of the target with the parameter, to specify a transform to simplify this dependence. For details on the available transforms, see Parameter Transforms on page 79.

Examples

Define an optimization parameter with the name `vsat`, an initial value of 1e7, and lower and upper bounds of 1e6 and 1e8, respectively:

```
p = OptParam("vsat", 1e7, minval=1e6, maxval=1e8)
```

As previously mentioned, it is usually a good idea to scale the value to be in the order of unity. In this case, you could add a scale of 1e7 to achieve this:

```
p = OptParam("vsat", 1e7, minval=1e6, maxval=1e8, scale=1e7)
```

Alternatively, you can apply a logarithmic transform to the optimization parameter:

```
p = TOptParam("vsat", 1e7, minval=1e6, maxval=1e8, transform=Log)
```

This example transforms a logarithmic dependence on the parameter `vsat` into a linear one, which would help in resolving features for smaller values of the parameter.

Defining Discrete Optimization Parameters

For discrete optimization parameters that take only a selected list of values, the syntax is as follows (see Discrete Optimization Parameters on page 78):

```
p = DOptParam(name, value, choices, sorted=False)
```

In this case, the parameter is also defined by a `name` corresponding to a Sentaurus Workbench parameter and the current `value`. However, the possible values that a discrete parameter can take are defined not by minimum and maximum values, but by the list `choices`, which should contain all the possible values the parameter can take.

Finally, `sorted` specifies whether to consider the listed values as ordered, that is, whether they can be sorted in a meaningful way. This is usually related to the way in which the discrete parameter affects the simulation.

Examples

Define a discrete parameter that can take the values `["a", "b", "c", "d"]`:

```
p = DOptParam("option", "c", choices=["a", "b", "c", "d"])
```

In this example, setting `"option"` to `"a"` or `"b"` represents an unrelated setting (for example, a change in methodology), so using `sorted=False` would be appropriate.

However, in some cases, the possible values will define an order. In this case, you would define the discrete parameter as:

```
p = DOptParam("option", "c", choices=["a", "b", "c", "d"], sorted=True)
```

This makes sense, for example, if setting `"option"="b"` reflects an increased value of some parameter with respect to `"option"="a"`.

Defining Optimization Targets

Like optimization parameters, optimization targets are separate objects that can be manipulated independently from the Optimization Framework. Since the Optimization Framework aims to be generic, it does not know how the specific simulation flow will produce the output data. Therefore, optimization targets are specified with different values:

- Response or simulation data

  The simulation output data must be contained in a Sentaurus Workbench variable. It can be either some raw simulation data (for example, an I_d–V_g curve produced by Sentaurus Device) or postprocessed data (for example, the threshold voltage of the device extracted by a Sentaurus Visual or Python tool instance). Therefore, the content of the specified node can be either a file (`.csv` or `.plt`) or a value.

- Target value (optional, defaults to 0)

  This is the value that the response should match. You can provide this value either as a file (`.csv` or `.plt`) or as a single value, and it must be the same way as the response was specified.

Examples

The following examples demonstrate the basic way to define optimization targets.

```
target = OptTarget("error_vt")
```

The simulation flow writes to the Sentaurus Workbench variable `error_vt`, which indicates the distance between the threshold voltage of the simulated device and a required threshold voltage. The target value of this variable in this simulation is zero.

```
target = OptTarget("vt", target=0.3)
```

The simulation flow writes to the Sentaurus Workbench variable `vt`, which indicates the threshold voltage of the simulated device. The target in this simulation is to match the threshold voltage to 0.3.

```
target = OptTarget("IdVg", target="target_IdVg.plt")
```

The simulation flow writes to the Sentaurus Workbench variable `IdVg` the name of a file containing the I_d–V_g characteristics of the simulated device. The target in this simulation is to match this curve to the I_d–V_g curve in the `target_IdVg.plt` file.

```
target = OptTarget("IdVg", target="target_IdVg")
```

The simulation flow writes to the Sentaurus Workbench variable `IdVg` the name of a file containing the I_d–V_g characteristics of the simulated device. The target in this simulation is to match this curve to the I_d–V_g curve in a file specified in the Sentaurus Workbench parameter or variable `target_IdVg`.

Target Weights

Each optimization target can have a weight that is applied when aggregating different targets. The weight is specified as an additional argument to `OptTarget`. For example:

```
target1 = OptTarget("vt_low", target=0.3)
target2 = OptTarget("vt_high", target=0.25, weight=0.5)
```

In this case, the error of `vt_low` will be twice as important as the error of `vt_high` during the optimization process.

Target Scaling

You can use scaling when the different optimization targets take values in different scales but they should be compared on equal grounds. For example, if `vt_low` is written in mV and `vt_high` is written in V, then you can use a scaling factor of 1e-3 for `vt_high` or 1e3 for `vt_low` to have their errors in the same scale. For example:

```
target1 = OptTarget("vt_low", target=30, scale=1e3)
target2 = OptTarget("vt_high", target=0.25, weight=0.5)
```

Note:

Scaling is applied to the data, but the weight is applied to the error contribution, so their magnitudes are not necessarily equivalent. For example, if a root mean square (RMS) error is used, then:

$$\text{Root Mean Square Error} = \sqrt{\sum_i \text{weight}_i \left[\left(\frac{\text{out}_i}{\text{scale}_i} - \frac{\text{target}_i}{\text{scale}_i}\right)\right]^2}$$

Target Constraints

In some situations, you might be interested in setting a target in a given range, but not with a specific value. In this case, you can supply a constraint to an optimization target. For example:

```
target2 = OptTarget("vt_high",
                    penalty=TargetPenalty([("lt", 0.4), ("gt", 0.2)]))
```

Here, the target range of the variable `vt_high` is between 0.2 and 0.4. If it is not satisfied, then a penalty is applied to the optimization target in the next optimization step. For details about defining constraints, see Constraints on page 83.

Target Errors

The errors e_i between the simulation output (out_i) and the target value ($target_i$) for each optimization target are used to guide the optimization process. The default way to measure this error is to simply calculate the difference between the simulation output and the target value:

$$e_i = out_i - target_i \tag{1}$$

You can change this by using the `error_method` argument with one of the values given in Table 1.

Table 1 Formulas used for different error_method options

Error method	Formula
`"abs"`	$e_i = \lvert out_i - target_i \rvert$
`"rel"`	$e_i = \left\lvert \frac{out_i - target_i}{target_i} \right\rvert$
`"log_abs"`	$e_i = \lvert \log_{10}\lvert out_i\rvert - \log_{10}\lvert target_i\rvert \rvert$
`"log_rel"`	$e_i = \left\lvert \frac{\log_{10}\lvert out_i\rvert - \log_{10}\lvert target_i\rvert}{\log_{10}\lvert target_i\rvert} \right\rvert$
`"square"`	$e_i = (out_i - target_i)^2$

For example, to define an optimization target that contributes a relative error with respect to the target value in a logarithmic scale, specify:

```
target1 = OptTarget("Ioff", target=1e-8, error_method="log_rel")
```

State Parameters

You can use state parameters to define a specific state for a given target. For example:

```
target1 = OptTarget("vt", target=0.3, conditions={"VDRAIN":0.05})
target2 = OptTarget("vt", target=0.2, conditions={"VDRAIN":0.7})
```

In a Sentaurus Workbench project in which the drain bias is parameterized with the parameter `VDRAIN` (but no initial split is present on it), these commands would define the following:

- `target1` as the error in the threshold voltage when using `VDRAIN=0.05`
- `target2` as the error in the threshold voltage when using `VDRAIN=0.7`

Using Files

To use a `.csv` or `.plt` file as simulation output data and target data, you must specify column names by using the arguments `simulation_variables` and `reference_variables`, respectively. They must be Python lists of length two: the first element is the name of the independent variable and the second element is the name of the dependent variable. For example:

```
target = OptTarget("simulation_data", target="target_data",
                   simulation_variables=["x", "y"],
                   reference_variables=["Vg [V]", "Id [A]"])
```

Here, the file name for the current simulation output is referenced by the Sentaurus Workbench node `simulation_data`. The file has columns named `x` and `y`, where `x` will be used as the independent variable and `y` as the dependent variable. The target file name is referenced by the Sentaurus Workbench node `target_data` and has columns named `Vg [V]` and `Id [A]`, where `Vg [V]` will be used as the independent variable and `Id [A]` as the dependent variable.

In addition, when using files, you can specify a range of interest for the independent and dependent variables of the target. These are specified with `x_range` and `y_range`. For example:

```
target = OptTarget("simulation_data", target="target_data",
                   simulation_variables=["x", "y"],
                   reference_variables=["Vg [V]", "Id [A]"],
                   x_range=[0.5, 0.75], y_range=[0, 5.])
```

Here, the error between the simulation and target curves is calculated only for target values with independent variables between 0.5 and 0.75, and with dependent variables between 0 and 5.

Note:

If the target data for a given optimization target is available from the outset, then it is possible (and recommended) to use a data container from the `libfromage` Python library. A data container allows a variety of functions including input from and output to `.csv` files, `.plt` files, and the database, easy slicing of data based on index and targets, several error calculation methods, resampling, and interpolation.

The following code snippet shows the use of the `Data` class to load I_d–V_g curves from subband Boltzmann transport equation (subband-BTE) simulations using Sentaurus Device QTX, with the values for $V_g \geq 0.4$ as target data:

```
# Load Id-Vg data from subband-BTE simulations.
sbte_data = Data.from_plt("sbte_reference_IdVg.plt", ivar="V(gate)",
                          dvar="I(drain)")

target = OptTarget("simulation_data", target=sbte_data[0.4:],
                   simulation_variables=["x", "y"])
```

Note:

Even though the capability to use files as output data or target data is provided through this interface, the use of a specific tool instance to calculate errors is recommended if advanced data manipulation is required. This allows full access to simulation data and target data and their manipulation.

The following code snippet shows the code in a Python tool instance using the `Data` class to calculate the root mean square deviation (RMSD) between the I_d–V_g curves from subband-BTE and Garand VE simulations, and the difference between their threshold voltages, and to write them back as Sentaurus Workbench variables:

```
# Load data from subband-BTE simulations.
target = Data.from_plt("n@node|QTX_SBTE@_IdVg.plt", target=True,
                       ivar="V(gate)", dvar="I(drain)")

# Load data from Garand VE simulations with drain bias=@VDRAIN@
# (in the database).
datasets = dbi.get_dataset(project="@node|garandve@")
data = Data.from_db(dataset=datasets, dvar=f"idrain", ivar=f"vgate",
                    bias__drain=@VDRAIN@)

# Match target with data, calculate error for Vg > 0.4 V,
# and write the difference to Sentaurus Workbench.
# The matching uses interpolation if the target and data objects
# do not have the same independent variable values.
target.UpdateFit(data, error_method="abs", interp="slinear")
```

```
error = target[0.4:@VGATE@].CalculateSingleError("rmsd")

print(f"DOE: error_high {error}")

# Calculate threshold voltages and write the difference
# to Sentaurus Workbench.
target_vt = target.Vt()
data_vt = data.Vt()
error = np.abs(target_vt-data_vt)

print(f"DOE: error_vt {error}")
```

Interpolating Targets

When the simulation data and reference values are in a file, they are sometimes evaluated at different points. In this case, the simulation data is interpolated on to the locations of the target values. Depending on the data, different ways to interpolate might be more appropriate. You can use the parameter `interpolation_options` to modify the interpolation. The following keywords can be specified:

- `"clip"` limits the oscillations resulting from interpolation. If specified, then it must have two components to define the minimum and maximum values that the interpolated data can take.
- `"extra"` limits the value taken by the data when extrapolation is necessary. If specified, then it must have two components: one to limit the maximum value and one to limit the minimum value.
- `"log"` determines whether to use logarithmic interpolation.
- `"method"` determines the order of the interpolating spline.

For example, consider the following optimization target:

```
target = OptTarget("simulation_data", target="target_data",
                   simulation_variables=["x", "y"],
                   reference_variables=["Vg [V]", "Id [A]"],
                   interpolation_options={"log": True, "method": 3,
                                          "clip": [0, None],
                                          "extra": [0, 1e-3]})
```

In this example, interpolation is performed in log space using a cubic spline. If extrapolation is performed, then it is clipped to the range `[0, 1e-3]`, that is, no values above 1e-3 or below 0 are allowed. Finally, the clipping defined by `[0, None]` is applied to the entire domain: negative values are set to zero and no restriction is applied to the maximum value.

Note:

By default, interpolation uses a cubic spline. For some data, this might result in unwanted oscillations, thereby preventing a good optimization. If this is the case, then setting `"method"` to 1 can be used to avoid this effect.

Transforming Targets

Sometimes, the simulation output and reference data have to be transformed in more complex ways than a scaling factor. One way to apply these transforms is to add an additional tool stage to the experiment such as Sentaurus Visual. Another option is to use the `transform` parameter in `OptTarget`. In this case, the transform is applied after the scaling (if given), but before the interpolation is performed. Using the same example as in Target Scaling, the RMS would be:

$$\text{Root Mean Square Error} = \sqrt{\sum_i \text{weight}_i \left[\left(T\left(\frac{\text{out}_i}{\text{scale}_i}\right) - T\left(\frac{\text{target}_i}{\text{scale}_i}\right)\right)\right]^2}$$

The way in which the transform function must be defined depends on whether the target is scalar or file based. For scalar data, the function should be able to handle both scalars and a NumPy array of length one as input and to return the value as either of them. For example, consider the following transform:

```
def test_transform(x):
    x = np.atleast_1d(x)
    return 2*x[0]**2

target1 = OptTarget("err", 0.5, scale=2, transform=test_transform)
```

In this case, the contribution to the total error of target `"err"` is $2(err/2)^2 - 2(0.5/2)^2$.

For file-based data, the transform function receives the simulation and reference as a `pandas.Series` with the index containing the independent variable and the column containing the dependent variable. The transform must also return a `pandas.Series`. As an example of implementing the same transform as before:

```
def test_transform(x):
    return 2*x**2
```

You can use transforms to change the independent variable as well, which affects interpolation. For example, to square the independent variable, you could use the following transform:

```
def test_transform(x, *args, **kwargs):
    y_data = x.values
    x_data = x.index.values
    xhat = pd.Series(data=y_data, name=x.name, index=x_data**2)
    xhat.index.name = "sqx"
    return xhat
```

Defining Optimization Strategies

After the optimization parameters and the optimization targets are defined, you can define a strategy to find an optimal set of parameters.

In the simplest situation, you have a set of parameters that you know is a good initial guess for the optimization process. You can use gradient-based local optimization to find the best set of parameters by calling the `optimizer.optimize()` method, which takes a list of optimization parameters and a list of optimization targets as its first and second arguments, respectively. For example:

```
optimizer.optimize([param1, param2, ...], [target1, target2, ...])
```

Local optimization can be performed using different methods, which can be grouped into families according to their properties. Each family is called a *backend*. For details about available backends and the methods that can be used, see Backend Optimization on page 59.

Unfortunately, it is not always possible to obtain a good initial guess for the optimization beforehand. In this case, it might be beneficial to search the parameter space initially to find a good initial guess for further gradient-based optimization. This can be done by using the `initial_search` argument of the `optimizer.optimize()` method.

The following example runs the simulations in a 5×5 grid of the two parameters (`param1` and `param2`) and uses the best point as the starting point for the gradient-based optimization:

```
optimizer.optimize([param1, param2], [target1, target2],
          initial_search={"method":"grid", "options": {"Ns": (5, 5)}})
```

If the gradient-based optimization is not necessary, then you can perform the search only:

```
optimizer.search([param1, param2], [target1, target2], method="grid",
                 options={"Ns": (5, 5)})
```

These tasks can be combined in a strategy that is appropriate for the problem being optimized. For example, if you know that `param1` affects `target1` but not `target2`, and `param2` affects only `target2`, you could specify the following:

```
optimizer.optimize([param1], [target1])
optimizer.optimize([param2], [target2])
```

If both parameters are loosely coupled but both affect both targets, then you could iterate over the optimizations for both parameters in a Gummel-like (or block nonlinear Gauss–Seidel) iteration. For example:

```
for i in range(3):
   optimizer.optimize([param1], [target1, target2])
   optimizer.optimize([param2], [target1, target2])
```

More control over the stopping criterion is possible by using the error after a call. For example:

```
for i in range(3):
   optimizer.optimize([param1], [target1, target2])
   if optimizer.last_error < 1e-3:
      break
   optimizer.optimize([param2], [target1, target2])
```

```
    if optimizer.last_error < 1e-3:
       break
```

Cost Functions

For problems with more than one optimization target, their contributions might have to be aggregated into a single value to drive the optimization process. For example, this is the case when `optimizer.optimize()` is called after the method has been set to `"minimize"` or whenever `optimizer.search()` is used.

The Optimization Framework provides several options to combine the errors of individual targets, which can be selected using the `optimizer.set_cost_function()` method.

For all cost functions, the final value is calculated from the individual errors e_i of all optimization targets passed to the task. As previously mentioned, by default, these are the differences between the simulation outputs and their target values, but they can be modified using the arguments `transform`, `scale`, and `error_method` when creating the `OptTarget` object (see Target Errors on page 16). Table 2 summarizes the available cost calculation methods and their formulas.

Table 2 Different cost calculation methods and their formulas

Cost calculation method	Formula
`"arctan"`	$c = \sum_i \tan^{-1}((e_i)^2)$
`"cauchy"`	$c = \sum_i \log(1 + (e_i)^2)$
`"huber"`	$c_i = \begin{cases} (e_i)^2, & e_i \le 1 \\ 2 \times \lvert e_i \rvert - 1, & e_i > 1 \end{cases}$ $c = \sum_i c_i$
`"l1"`	$c = \sum_i \lvert e_i \rvert$

Table 2 Different cost calculation methods and their formulas (Continued)

Cost calculation method	Formula
`"rmsd"`	$c = \sqrt{\frac{1}{N}\sum_i (e_i)^2}$
`"soft_l1"`	$c = \sum_i 2 \times (\sqrt{1 + (e_i)^2} - 1)$
`"squared"`	$c = \sum_i (e_i)^2$
`"sum"`	$c = \sum_i e_i$

The default calculator `"rmsd"` provides the root mean square deviation of the simulation outputs with respect to their target values. The `"squared"` method is equivalent, that is, a minimizer of `"rmsd"` is a minimizer of `"squared"`, with the advantage of a smooth derivative at the minimum. On the other hand, its value can grow very fast due to the squared dependence on the individual errors.

The methods `"huber"`, `"l1"`, and `"soft_l1"` are similar for large values of the individual errors, which contribute to the cost with a linear function of their absolute value.

The methods `"arctan"` and `"cauchy"` can limit large individual error contributions severely, with a limit on the contribution for `"arctan"` and with a logarithmic value for `"cauchy"`.

The method `"sum"` simply adds the individual errors and, therefore, is not a positive function, that is, it can result in negative values. This means it is not appropriate as a measure of error, but it can be useful in situations where a simulation output must be minimized without reference to a target value, that is, you are interested only in the simulation target achieving the lowest (possibly negative) value.

Note:

The call to `optimizer.set_cost_function()` changes the cost function for all tasks in the script after the call up until a different method is set with another call to `optimizer.set_cost_function()`. To revert it to the default cost function, you should call the method again with `"rmsd"`.

Example

This example demonstrates how to use the `"l1"` cost function for a search and then to change it to `"squared"` for a local optimization of the parameters.

```
a = OptParam("a", 0, -1, 1)
b = OptParam("b", 0, -1, 1)

target1 = OptTarget("vt_low", target=0.3)
target2 = OptTarget("vt_high", target=0.25, weight=0.5)

optimizer.set_cost_function("l1")
optimizer.search([a, b], [target1, target2], method="grid",
                 options={"Ns": 3})

optimizer.set_method("minimize")
optimizer.set_cost_function("squared")
optimizer.optimize([a, b], [target1, target2])
```

Studying Optimization Problems

When defining an optimization problem, it is not always possible to know beforehand which parameters are important. To help with analyzing the dependence of a response on different optimization parameters, you can use the following methods:

- `optimizer.search()` – see Searching the Optimization Parameter Space
- `optimizer.screening_analysis()` – see Screening Analysis on page 28
- `optimizer.sensitivity_analysis()` – see Sensitivity Analysis on page 31

Searching the Optimization Parameter Space

You can use the `optimizer.search()` method to explore the input space and to have a better idea of how the response changes with the value of the parameters. This is also called *design-of-experiments* (DOE). There are several options that can be used for this:

- `"grid"` creates experiments with parameter values defined on a regular grid (see Grid Sampling on page 66).
- `"latin"` creates experiments with parameter values using Latin hypercube sampling (see Latin Hypercube Sampling on page 67).
- `"random"` creates experiments with random parameter values (see Random Sampling on page 67).
- `"sobol"` creates experiments with parameter values using a Sobol sequence (see Sampling Based on Sobol Sequences on page 67).

- `"bayesian_optimizer"` creates experiments *adaptively* starting from an initial set of experiments defined by Latin hypercube sampling. In each iteration, the algorithm selects some parameter sets that have the largest potential to minimize the response (see Bayesian Optimization on page 68).
- `"turbo"` also creates experiments adaptively and builds on top of Bayesian optimization by keeping separate models that can change the focus of their search as the algorithm progresses. This makes it more likely to converge to a better local minimum (see Trust Region Bayesian Optimization on page 71).
- *Sentaurus Workbench DOEs* create experiments according to the supplied Sentaurus Workbench DOE (see Sentaurus Workbench Design-of-Experiments on page 73).

Output Plots

If the option `do_plots=True` is passed to `optimizer.search()`, after the task is finished, the Optimization Framework creates a response surface model (RSM) using a Gaussian process that can be visualized by Sentaurus Visual to understand the problem better. Different plot types of this internal RSM are created:

- One-dimensional cutlines of the response surface and simulated points (PNG files
- Two-dimensional cutplanes of the response surface and simulated points (PNG files)
- Sentaurus Visual files of the cutplanes of the RSM

Figure 2 shows an example of a 1D cutline plot. The thick blue line is the mean value of the model along the cutlines, and the thin red lines are the mean value ±1 standard deviation.

Circles are the simulated values. If the number of optimization parameters is greater than one, then the color of the circle represents the distance to the cutline. Black circles are closer to the cutline and white circles are farther away. In addition, if there are more than one optimization parameters, then the cutline is taken going through the best solution found.

Figure 2 Example of 1D cutline plot generated by optimizer.search() method

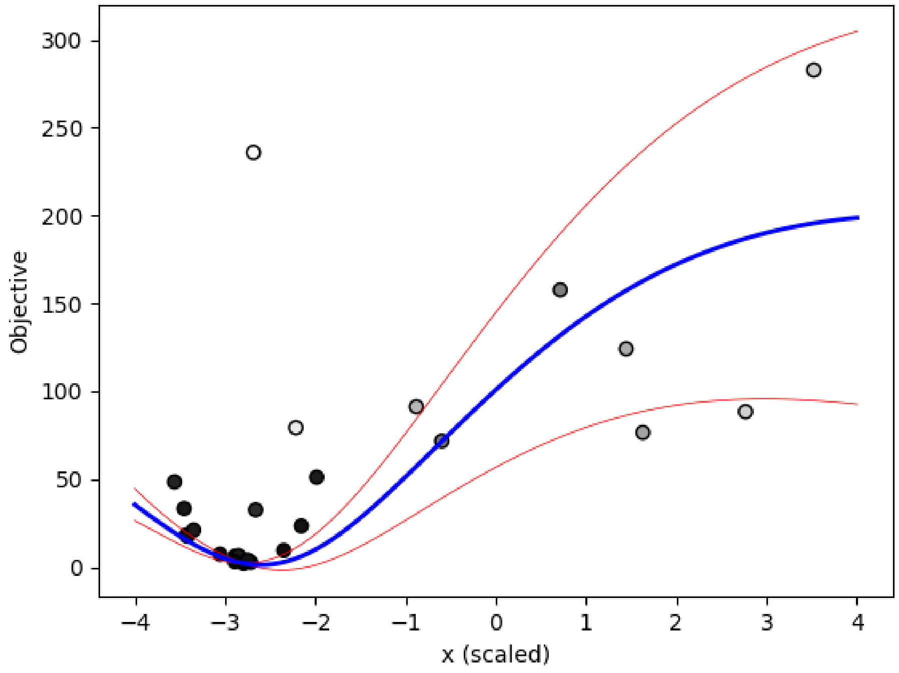


Figure 3 shows an example of a 2D cutplane plot. Like 1D cutline plots, circles are the points that have been simulated. If there are more than two optimization parameters in the problem, then the color of the circles represents the distance of the 2D cutplane to the actual simulation point. Closer to black means closer to the plane, and closer to transparent means farther from the plane. For example, consider a plot of an xy cutplane of a problem with variables `x`, `y`, and `z`. If the cutplane is at `z=0`, then the simulation point (1, 2, 0) would be black, and the simulation point (1, 2, 3) would be lighter in color. If there are more than two optimization parameters, then the cutplane is taken going through the best solution found.

The contour lines in the plot represent the mean value of the model fitted to the data. Background color crosses represent uncertainty in the model (1 standard deviation). Red means higher uncertainty, and blue means lower uncertainty.

Figure 3 Example of 2D cutplane plot generated by optimizer.search() method

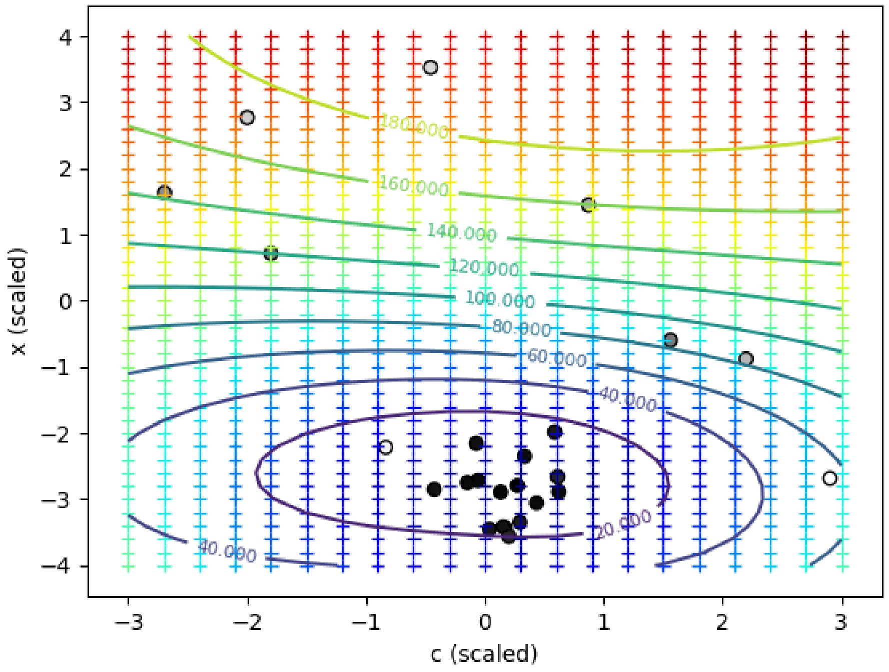


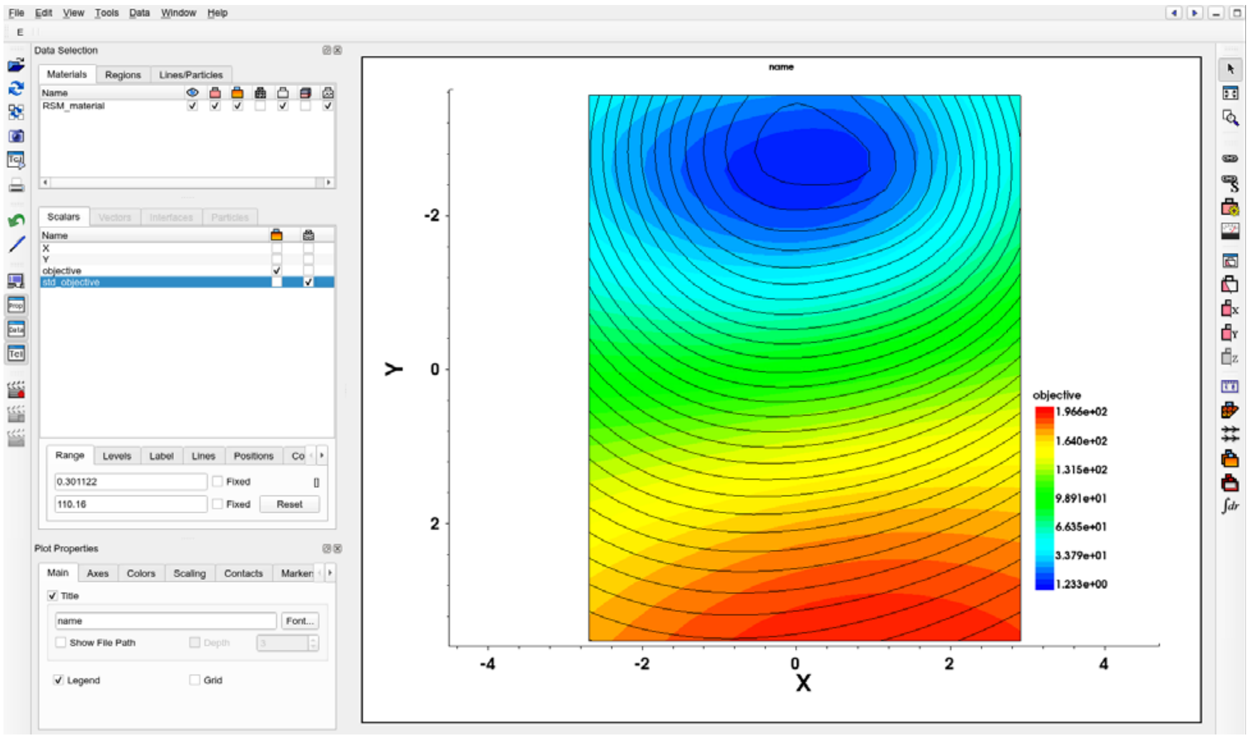
Figure 4 shows an example of the Sentaurus Visual 2D cutplane plot. The file includes the mean value (`objective` field) and standard deviation of the model prediction (`std_objective` field).

You can find the RSM plots in the Optimizer node folder, or you can access the RSM plots from the graphical user interface of Sentaurus Workbench by choosing **Nodes** > **Visualize** > **Optimization Plots**.

Note:

For this menu option to be available, you must highlight a real node in Sentaurus Workbench.

Figure 4 Example of Sentaurus Visual 2D cutplane plot generated by optimizer.search() method



Screening Analysis

For some problems, you might have many parameters, so you might be interested in selecting only a subset with the largest impact. You can use screening analysis to determine how much the different parameters affect the simulation responses. Parameters are then ranked according to their influence on each response, and this information can be used to select a subset of parameters.

Due to the possibly large number of parameters, this task uses a polynomial RSM built from a given DOE such as a Plackett–Burman design. These RSMs are used to obtain the effect of each parameter on the simulation responses. This is done using the normalized coefficients of the RSMs to rank parameters according to their effect on the responses.

To run a screening analysis, you use the `optimizer.screening_analysis()` method or, alternatively, the equivalent `optimizer.run_screening_analysis()` method (see optimizer.screening_analysis on page 55). For example:

```
sparams = optimizer.screening_analysis([a, b, c, d],
                                       [target1, target2],
                                       doe="boxBehnken",
                                       order=1,
                                       screening_criterion="local_strong",
                                       screen_best=3)
```

where a screening analysis is run with the following configuration:

- `doe` specifies the DOE (Box–Behnken in this case) to construct the RSM.
- `order` specifies the order of the polynomial model for the RSM.
- `screening_criterion` sets `local_strong`, which means to always use the most responsive value for each parameter from the RSMs of `target1` and `target2`.
- `screen_best` specifies to return only the top three parameters.

The method returns an object of type `ScreeningAnalysisResult` including the most influential parameters, influences, and RSMs used for the calculation of influences, and the evaluations used to fit the RSM. In addition, an analysis summary is printed to the screen unless `verbose=False` is passed.

The `ScreeningAnalysisResult` object has the following attributes:

- `evaluations`: A `pandas.DataFrame` object with the values of all evaluations made to calculate the sensitivities. Each row corresponds to a combination of parameters, and each column corresponds to a target.
- `influences`: A `pandas.DataFrame` object with the values of all influences. Each row contains a combination of parameters present in the polynomial RSM, and the columns are as follows:
    - Column *average* contains the influence by using `screening_criterion="average"`.
    - Column *local_strong* contains the influence by using `screening_criterion="local_strong"`.
    - Column *influential* lists whether the parameter combination is influential according to the selected criterion.
    - The remaining columns contain the individual influences for each target.
- `rsm`: A polynomial RSM fitted to the data used to calculate the influences.
- `screening_criterion`: A string describing the screening criterion used to aggregate multiple targets.
- `screening_type`: A string describing the screening decision method used to select the influential parameters.

- `selected_parameters`: A list with the parameters selected as influential.

  The list of parameters in `selected_parameters` can be used in any subsequent call to optimization routines. You could use the following example to optimize the value of the three most relevant parameters:

  ```
  optimizer.optimize(sparams.selected_parameters, [target1, target2])
  ```

Note:

The call to `optimizer.screening_analysis()` does not change the value of the optimization parameters (`a`, `b`, `c`, and `d`) in this example.

Example of Screening Analysis

Consider the following problem:

$$\text{err} = |a+b| + a^2 + b^2 + |ab| - c^2 d$$

$$\text{err2} = c^2 d$$

Parameters `a` and `b` have an impact on the variable `err` (`target1`) but not `err2` (`target2`). In addition, parameters `c` and `d` are not coupled to `a` and `b` in any of the targets. A model of order 2 built from a Box–Behnken design is used for the analysis. An example of a screening analysis report is:

```
Statistics for polynomial regressions (order 2):
+----------------+----------------+----------------+---------------+
|       r2       |     r2Adj      |      MSE       |      AICc     |
+----------------+----------------+----------------+---------------+
| 0.994647827018 | 0.987154784843 | 0.266666666667 | 122.520731325 |
| 0.971703452179 | 0.932088285229 | 0.266666666667 | 122.520731325 |
+----------------+----------------+----------------+---------------+

Screening Analysis - screening 3 most influential terms
+-------+------------------+----------+
|  Term |   Influence [%]  | Selected |
+-------+------------------+----------+
|   c   |   31.5789473684  |   True   |
| c x d |   31.5789473684  |   True   |
|   a   |   23.0769230769  |   True   |
|   b   |   23.0769230769  |  False   |
|   d   |   21.0526315789  |  False   |
| c x c |   15.7894736842  |  False   |
| a x a |   5.76923076923  |  False   |
| a x b |   5.76923076923  |  False   |
| b x b |   5.76923076923  |  False   |
| d x d | 1.92153985031e-14 |  False   |
| a x d |  1.0248212535e-14 |  False   |
| a x c | 8.96718596813e-15 |  False   |
| b x d | 7.36590275953e-15 |  False   |
```

```
| b x c | 3.84307970063e-15 |  False   |
+-------+-------------------+----------+
```

Table 1 reports the results of the polynomial fitting for each target. The report includes the coefficient of determination (r2) and the adjusted coefficient of determination (r2Adj) to quantify the quality of the fit, the mean square error (MSE) of the fit, and the corrected Akaike information criterion (AICc) to compare different models (for example, RSM of orders 1 and 2).

Table 2 includes a summary of the importance of each term in the RSM used. In this example, `screen_best=3` was used, so only the top three parameters have been selected. This would translate to parameters `a`, `c`, and `d` being returned by the call to `screening_analysis()`. In this case, there are no terms containing `d x d`, `a x c`, `a x d`, `b x c`, or `b x d` terms in the models, so the RSM correctly reports that they do not have any influence.

Sensitivity Analysis

Several sources of uncertainty affect the confidence in the results of a simulation tool. Two dominant sources are structural uncertainty and parametric uncertainty. Structural uncertainty comes from the inaccuracy of the physical models used and is not discussed here. Parametric uncertainty arises from incomplete knowledge of model parameters such as empirical quantities, defined constants, and stochastic parameters (for example, due to manufacturing variations).

Sensitivity analysis can be used to understand how the variability of different parameters affects the behavior of the system. This can help with various issues such as testing robustness to variations, identifying the most influential parameters with respect to different outputs for model simplification or improvement, and reducing dimensionality for more efficient optimization.

The Optimization Framework includes two main types of sensitivity analysis: one-factor-at-a-time and the method of elementary effects. One-factor-at-a-time sensitivity analysis is used mainly for a local study around a nominal point; whereas, the elementary effects method of sensitivity analysis is more useful when considering the global properties of the system.

Note:

A tool specific for parameter screening using polynomial models is described in Screening Analysis on page 28.

Local Sensitivity Analysis

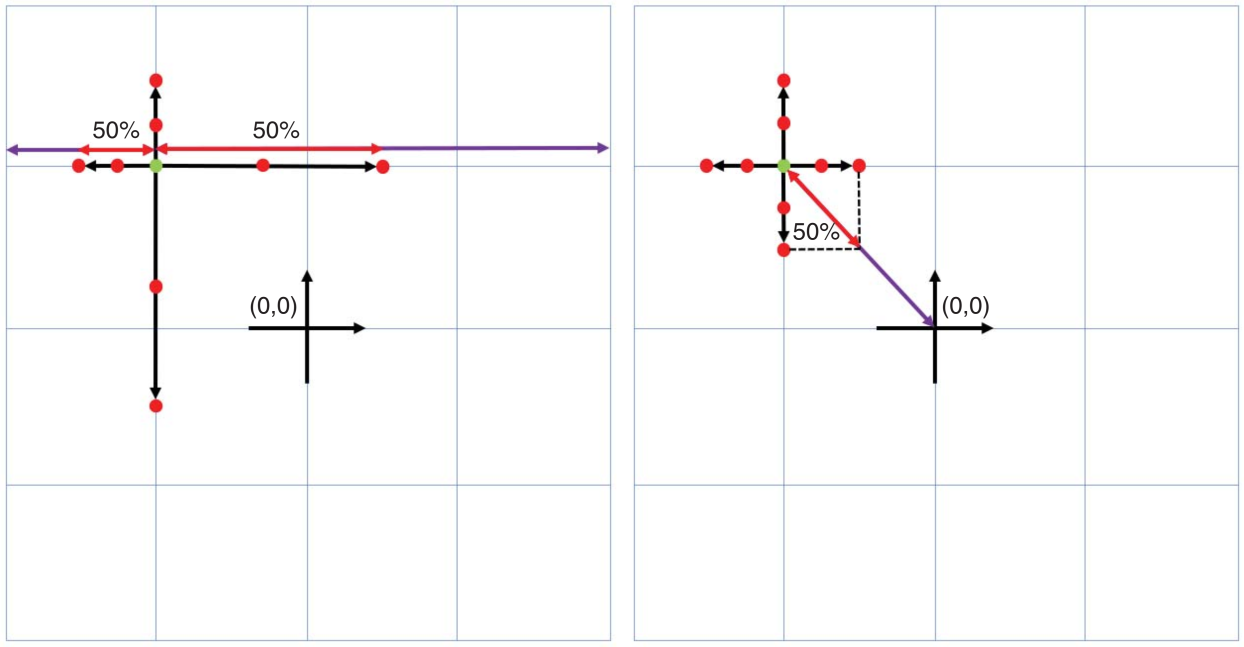
Local sensitivity analysis and, in particular, one-factor-at-a-time sensitivity analysis aim at understanding the response of the model as a function of small changes to a single parameter, with all other parameters fixed. Therefore, it reveals only the local gradient of the response surface of the model with respect to a given parameter.

There is some basic functionality to perform local sensitivity analysis using the `optimizer.sensitivity_analysis()` method (or, alternatively, the equivalent `optimizer.run_sensitivity_analysis()` method). For example:

```
s = optimizer.sensitivity_analysis([a, b, c, d], [target1, target2],
                                   percentage_range=10)
```

Here, `percentage_range` defines the percentage of the parameter range at which to evaluate each of the parameters to perform the sensitivity analysis. This is calculated based on the distance from the current nominal (that is, the values set for the input parameters such as `a`, `b`, `c`, and `d` in the previous example) and the upper and lower bounds. This means that the shifts to upper and lower values might differ if the nominal is not centered in the domain. Figure 5 (*left*) shows an example of such a situation. The nominal point is not centered in the domain and, therefore, the shifts left and up are smaller than the shifts right and bottom.

Figure 5 Example of designs with (left) npoints=5 and percentage range=50 and (right) percentage value=50. The nominal point of the design is indicated by a green circle and the rest by red circles. Red and purple arrows indicate the respective calculations of the 50%.



By default, only the experiments with parameter values at the boundary of the range and the midpoint are executed. More points in the range can be considered using the `npoints` parameter. For example:

```
s = optimizer.sensitivity_analysis([a, b, c, d], [target1, target2],
                                   percentage_range=10, npoints=7)
```

This example simulates a total of seven points: the midpoint, the boundaries, and two points in between on each side of the range.

As an alternative to defining the ranges for the design using `percentage_range`, you can specify `percentage_value`, which uses the given percentage of the current parameter values to define the maximum excursion of the parameters instead of using a percentage of their distance to the upper and lower bounds. For example:

```
s = optimizer.sensitivity_analysis([a, b, c, d], [target1, target2],
                                   percentage_value=10, npoints=7)
```

Figure 5 shows an example of the design using both `percentage_range` and `percentage_value` with five levels per parameter.

The call to `optimizer.sensitivity_analysis()` returns an object of type `SensitivityAnalysisResult` that includes sensitivities, nonlinearities, and evaluations for the given options. In addition, a summary of the analysis is printed to the screen unless `verbose=False` is set, and plots of the sensitivity are created in the optimizer node unless `do_plots=False` is set.

The sensitivity S_{ik} and the relative sensitivity RS_{ik} of target k to parameter i are defined as:

$$S_{ik} = \frac{p_r}{100} \frac{\sum_{j=1}^{nlevels} |T_{ik,j} - T_{ik,\text{center}}| / |P_{i,j} - P_{i,\text{center}}|}{\sum_{j=1}^{nlevels} 1 / |P_{i,j} - P_{i,\text{center}}|} \tag{2}$$

$$RS_{ik} = 100 \frac{S_{ik}}{|T_{ik,\text{center}}|} \tag{3}$$

where:

- p_r is `percentage_range` or `percentage_value`.
- *nlevels* is the number of levels in the sensitivity analysis, excluding the nominal point.
- $T_{ik,j}$ is the value of target k when setting parameter i to level j.
- $P_{i,j}$ is the value of parameter i at level j.

For example, in the case of two levels equidistant from the center point, the sensitivity is given by:

$$S_{ik} = \frac{p_r}{100}\frac{1}{2}(|T_{ik,\,\mathrm{low}} - T_{ik,\,\mathrm{center}}| + |T_{ik,\,\mathrm{high}} - T_{ik,\,\mathrm{center}}|) \tag{4}$$

Including `percentage_range` or `percentage_value` in the calculation allows for an easier comparison of the sensitivities for the same function with different values of the parameter. For example, for a target close to linearity, this results in a similar value of the sensitivity regardless of `percentage_range`.

The nonlinearity N_{ik} of the target k along the direction of parameter i is defined as the RMSD between the target value T_{ik} and a linear approximation to it, $\hat{T}_{ik}$, normalized by the range of the function in the domain of the design [1]:

$$N_{ik} = \frac{1}{\Delta T_{ik}\Big|_{P_{i,L}}^{P_{i,R}}}\sqrt{\frac{\int_{P_{i,L}}^{P_{i,R}}(T_{ik} - \hat{T}_{ik})^2\,dP}{P_{i,R} - P_{i,L}}} = \frac{\mathrm{RMSD}(T_{ik};\hat{T}_{ik})}{\Delta T_{ik}\Big|_{P_{i,L}}^{P_{i,R}}} \tag{5}$$

where:

- T_{ik} is the value of target k along the direction of parameter i.
- $P_{i,L}$ and $P_{i,R}$ are the values of parameter i at the leftmost and rightmost values.

Note:

The functions T and $\hat{T}$ are approximated using the sampling performed for the sensitivity analysis, so increasing the value of parameter `npoints` also increases the accuracy of the calculated nonlinearity.

Global Sensitivity Analysis

In contrast to local sensitivity analysis, global sensitivity analysis aims to summarize information about the entire domain, including interactions between parameters to some extent. The implemented methodology, the method of elementary effects [2][3] is based on aggregating data from multiple one-factor-at-a-time experimental designs.

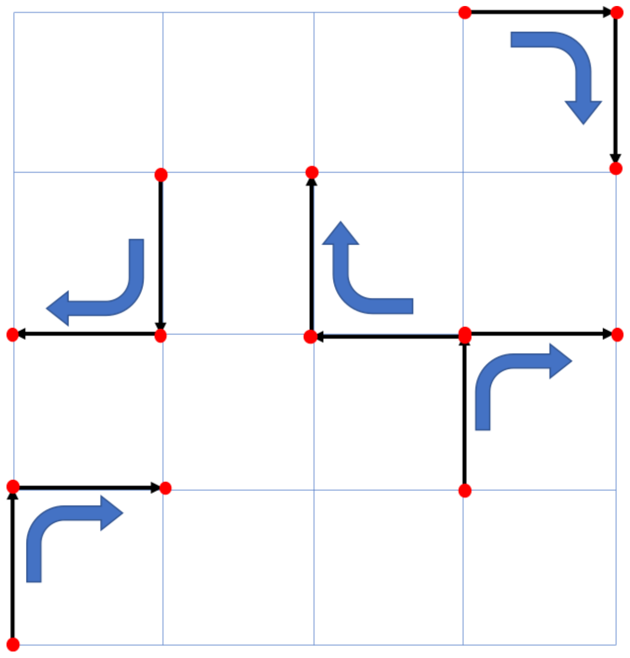
Two parameters are used to define the analysis. `nlevels` determines how many levels each parameter can take. For example, if there are four parameters in the analysis and `nlevels=5`, then the total number of possible parameter value combinations is 5^4 = 625 (these are *possible* values, not the actually simulated values). The other parameter needed to define the analysis is `ntrajectories`, which determines how many trajectories are executed.

Each of these trajectories is local in the sense that:

- Each trajectory has as many points as parameters in the simulation plus one. For example, for four parameters, each trajectory has 5 points.
- In each trajectory, the coordinates might differ from the previous point in the trajectory in only one dimension and only by one step in the grid defined by `nlevels`.
- In each trajectory, all dimensions must be changed exactly once.

Figure 6 shows an example of five trajectories in two dimensions using five levels for each parameter.

Figure 6 Five trajectories with five levels for each of two parameters



To use global sensitivity analysis, you must specify `method="global"`. For example, to run a global sensitivity analysis for targets `target1` and `target2` with respect to parameters `a`, `b`, `c`, and `d`, the command would be as follows:

```
s = optimizer.sensitivity_analysis([a, b, c, d], [target1, target2],
                                   method="global",
                                   nlevels=5, ntrajectories=10)
```

Note:

Global sensitivity analysis does not work with discrete parameters. If any discrete parameters are passed in the list of parameters, then they are ignored.

As in the case of local sensitivity analysis, the command returns an object of type `SensitivityAnalysisResult`, but in this case, only the sensitivity measure and evaluations are included. In addition, a summary of the analysis is printed to the screen unless `verbose=False` is set, and plots of the sensitivity are created in the optimizer node unless `do_plots=False` is set.

For each trajectory j, you can calculate the value of the elementary effect for target k in each dimension i:

$$EE_{ik}^{j} = \frac{T_k(P_{i+1}^{j}) - T_k(P_i^{j})}{\Delta} \tag{6}$$

where:

- P_i^j and P_{i+1}^j denote the initial point and the final point of the edge of trajectory j changing the value of parameter i.
- Δ is the step in the parameter.
- $T_k(P_i^j)$ denotes the value of target k at P_i^j.

The mean, the mean of the absolute, and the standard deviation of the values EE_{ik}^j can be used as sensitivity metrics to quantify the influence of each parameter:

$$\mu_{ik} = \frac{1}{N_t}\sum_j EE_{ik}^j \tag{7}$$

$$\mu_{ik}^* = \frac{1}{N_t}\sum_j \left|EE_{ik}^j\right| \tag{8}$$

$$\sigma_{ik} = \sqrt{\frac{1}{N_t - 1}\sum_j (EE_{ik}^j - \mu_{ik})^2} \tag{9}$$

In particular, μ* can be used as the main metric to determine the total (that is, including interactions) influence of each parameter. The standard deviation of the elementary effects σ_{ik} gives additional information regarding the variability of the dependence across the space. For example, if σ_{ik} is very small, it means that the dependence of target k on parameter i is uniform across the space. However, if σ_{ik} is large, then it indicates that the influence of target k on parameter i is highly dependent on the location in the parameter space.

SensitivityAnalysisResult Object

The `SensitivityAnalysisResult` object has the following attributes:

- `evaluations`: A `pandas.DataFrame` object with the values of all evaluations made to calculate the sensitivities. Each row corresponds to a combination of parameters, and each column corresponds to a target.
- `nonlinearity`: A `pandas.DataFrame` object with the nonlinearity measure for each parameter. Each row corresponds to a parameter. For the columns, the first-level indices are `percentage` and `RMSD`. Each one contains the nonlinearity for each target as either a percentage value or an RMSD value. If `method="global"`, then this field is set to `None`.
- `sensitivity`: A `pandas.DataFrame` object with the sensitivities for each parameter. Each row corresponds to a parameter. For the columns, the first-level indices are `absolute` and `relative`. Each one contains the sensitivity for each target as either an absolute value or a relative value.

Examples of Sensitivity Analysis

These examples cover local sensitivity analysis and global sensitivity analysis.

Local Sensitivity Analysis

Consider the following problem:

$$\text{err} = |a+b| + a^2 + b^2 + |ab| - c^2 d$$

$$\text{err2} = c^2 d$$

Parameters `a` and `b` affect variable `err` but not `err2`. Parameters `c` and `d` affect both targets.

You run the sensitivity analysis using the following command:

```
s = optimizer.sensitivity_analysis([a, b, c, d], [target1, target2],
                                    percentage_range=10, npoints=7)
```

where `target1` corresponds to `err` and `target2` corresponds to `err2`.

The corresponding sensitivity analysis report is:

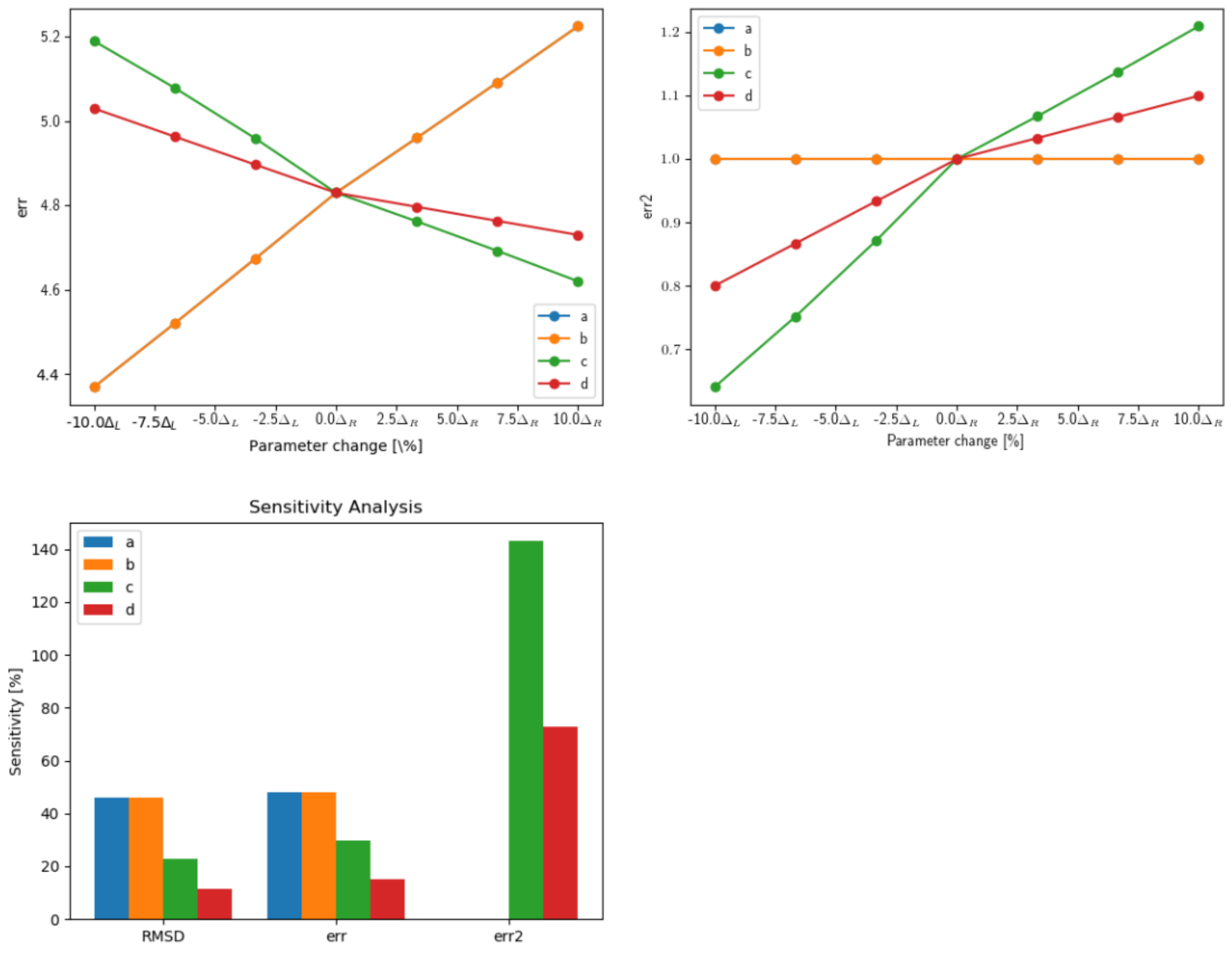
```
Sensitivity Analysis using custom parameter ranges:
    a: (1, 1.5)
    b: (0.8, 1.3)
    c: (-0.5, 1)
    d: (-0.5, 0.5)


Deviation from linearity for each target:
+-----------+----------------------+----------------------+------------------------+
| Parameter | Nonlinearity [%]: err | Nonlinearity [%]: err2 | Nonlinearity [%]: total |
+-----------+----------------------+----------------------+------------------------+
|     a     |         2.93         |         nan          |          2.93          |
|     b     |         1.47         |         nan          |          1.47          |
|     c     |        49.55         |        49.38         |         49.55          |
|     d     |         0.83         |        45.17         |          0.83          |
+-----------+----------------------+----------------------+------------------------+

Sensitivity Analysis per target:
+-----------+--------------------+---------------------+---------------------+----------------------+
| Parameter |  Sensitivity: err  | Sensitivity [%]: err |  Sensitivity: err2  | Sensitivity [%]: err2 |
+-----------+--------------------+---------------------+---------------------+----------------------+
|     a     | 10.956153333333354 |  17.913392564204045  |         0.0         |         0.0          |
|     b     | 14.974050000000005 |  24.482683635864213  |         0.0         |         0.0          |
|     c     | 4.224843749999991  |  6.9076510992155065  | 0.18749999999999997 |   62.499999999999986   |
|     d     | 4.814933333333343  |  7.8724519771055474  | 0.14666666666666672 |   48.88888888888891    |
+-----------+--------------------+---------------------+---------------------+----------------------+

Sensitivity Analysis on total rmsd of all targets:
+-----------+--------------------+-------------------+
| Parameter |    Sensitivity     |  Sensitivity [%]  |
+-----------+--------------------+-------------------+
|     a     | 7.747090529949982  | 17.91299258971263 |
|     b     | 10.588114461257515 | 24.48207041731311 |
|     c     | 2.986984260359146  | 6.906570500828662 |
|     d     | 3.404137954366324  | 7.871122418820782 |
+-----------+--------------------+-------------------+
```

The first table summarizes the nonlinearity for each target and for the RMSD of all targets and is only printed if `npoints` > 3. In this example, the nonlinearity for parameters `a` and `b` is not defined for `err2` since it does not depend on them.

The last two tables contain a summary of the sensitivity measures. The first table contains sensitivities for each individual target, and the second table contains an aggregate of all sensitivities for each parameter using the RMSD of all targets. In these tables, there are two columns for each target. The first column contains the sensitivity in the same units as the target, and the second column contains the value as a percentage with respect to the midpoint value.

The plots in Figure 7 are generated as the visual summary of the sensitivity analysis. The top two plots show the values of the target for all parameters and for each target as a function of the percentage range deviation from the center point. The bottom plot presents a visual overview of the relative sensitivity for all parameters and targets.

Figure 7 Local sensitivity analysis showing target evaluations and sensitivity for each parameter and target



Global Sensitivity Analysis

For the same problem, you run a global sensitivity analysis using the following command:

```
s = optimizer.sensitivity_analysis([a, b, c, d], [target1, target2],
                                   method="global",
                                   nlevels=5, ntrajectories=10)
```

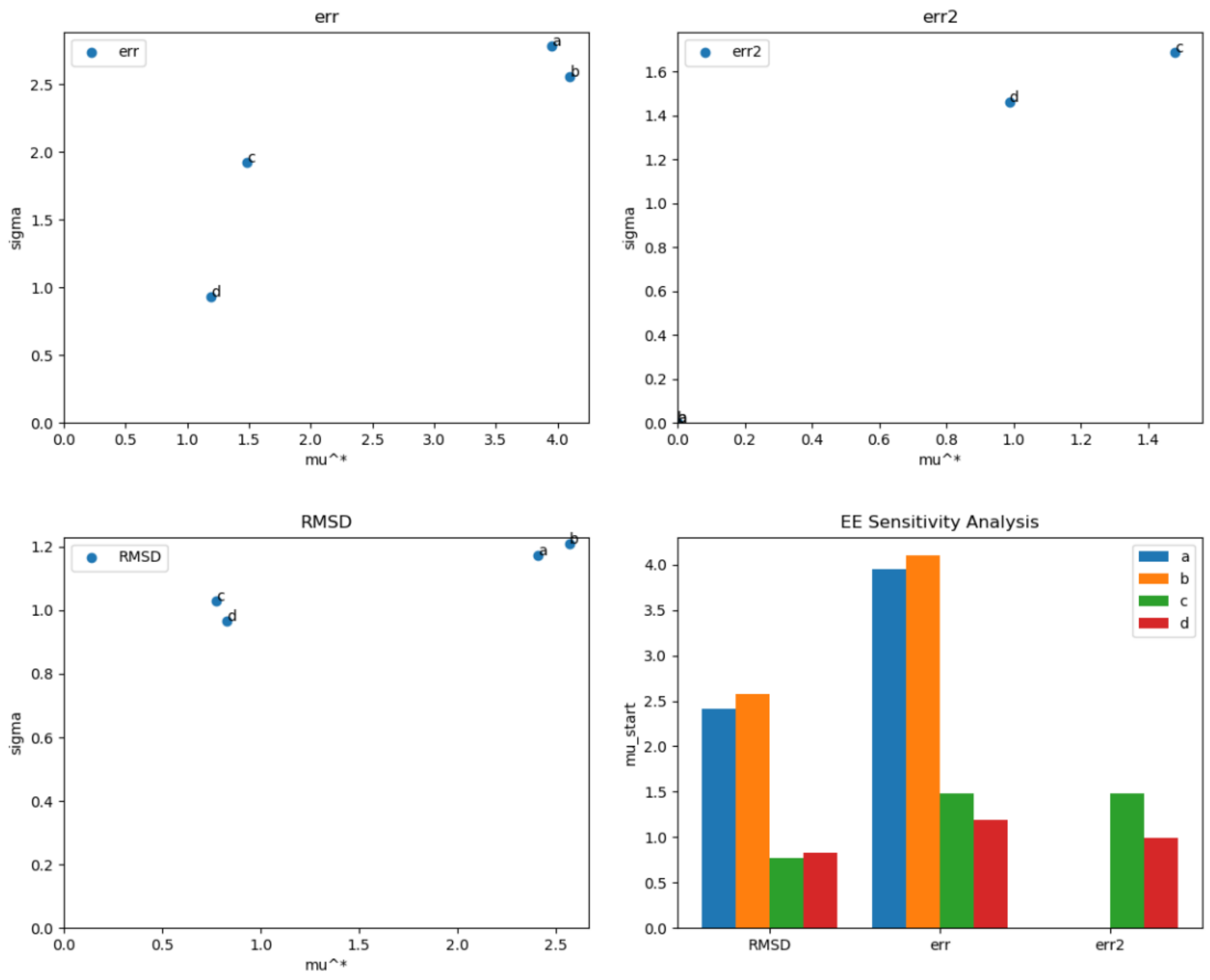
The corresponding sensitivity analysis report is:

```
Elementary effects per target:
+-----------+---------------------+--------------------+--------------------+----------+--------------------+------------+
| Parameter |       mu: err       |     sigma: err     |     mu^*: err      | mu: err2 |    sigma: err2     | mu^*: err2 |
+-----------+---------------------+--------------------+--------------------+----------+--------------------+------------+
|     a     |        2.95         | 2.7879203718901295 |        3.95        | 0.0      | 0.0                | 0.0        |
|     b     |         3.3         | 2.5612496949731396 |        4.1         | 0.0      | 0.0                | 0.0        |
|     c     |      -0.06875       | 1.921394757071019  |      1.48125       | 0.91875  | 1.688899882319849  | 1.48125    |
|     d     | -1.1958333333333333 | 0.9298857456698645 | 1.1958333333333333 | 0.35     | 1.4622506910011475 | 0.9875     |
+-----------+---------------------+--------------------+--------------------+----------+--------------------+------------+

Elementary effects on total rmsd of all targets:
+-----------+---------------------+--------------------+--------------------+
| Parameter |         mu          |       sigma        |        mu^*        |
+-----------+---------------------+--------------------+--------------------+
|     a     |  2.2818951237437948 | 1.1729038357527553 | 2.4107460620105203 |
|     b     |  2.450672527658676  | 1.2095949868082245 | 2.570706114987053  |
|     c     | 0.37647286639598193 | 1.0290805934827143 | 0.774537309552314  |
|     d     | -0.3972246971335995 | 0.9671499242854247 | 0.8304547273852373 |
+-----------+---------------------+--------------------+--------------------+
```

For global sensitivity analysis, there are two tables. The first table summarizes the sensitivity measures for each target, and the second table summarizes the RMSD of all targets. In each case, there are three columns for each target containing the previously defined measures μ, μ*, and σ.

The plots in Figure 8 are generated as the visual summary of the sensitivity analysis. The top two plots show the values for μ* and σ for all parameters and for each target. The location along the μ*-axis gives a measure of the influence of the parameter, while the location along the σ-axis gives a measure of the variability of the elementary effects across the complete space. The lower-left plot shows the same for the RMSD of all targets and the lower-right plot presents a visual overview of μ* for all parameters and targets.

Figure 8 Global sensitivity analysis showing target evaluations and sensitivity for each parameter and target



Optimization Postprocessing

All intermediate optimization results are available as a dictionary of Python objects at the end of the execution of each task indexed by the task number.

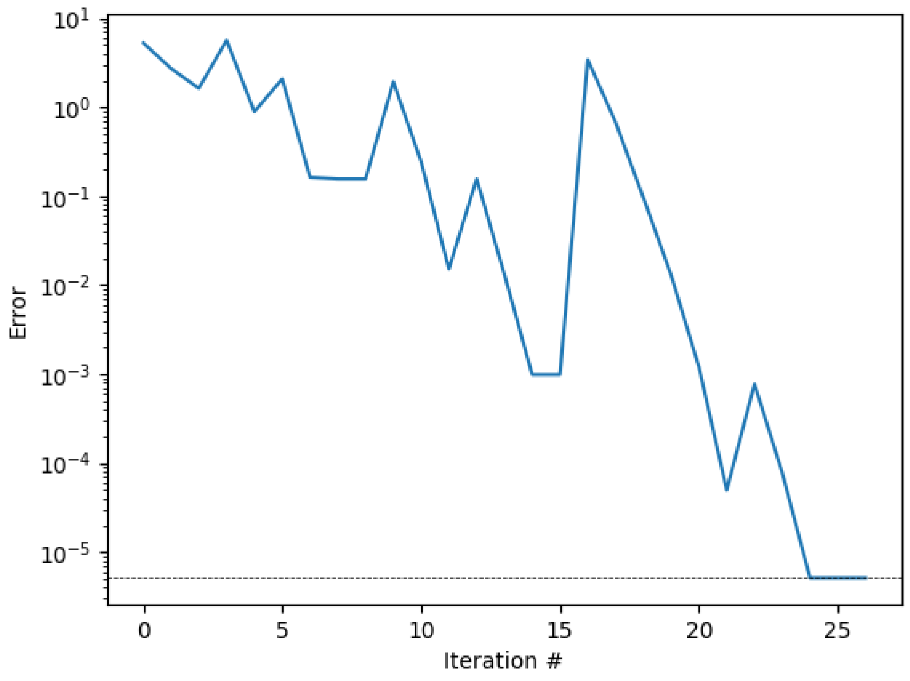
You can manipulate the information in the optimization script by using standard Python libraries. For example, to create a plot of error versus iteration number for each optimization call, use the following:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
for i, h in optimizer.history.items():
```

```
plt.figure()
iterations = h.index.values
errors = h["errors"]

plt.plot(iterations, errors)
plt.axhline(np.min(np.abs(errors))*np.sign(np.min(errors)),
            linestyle='--', linewidth=0.5, color='k')
plt.xlabel("Iteration #")
plt.ylabel("Error")
plt.savefig(f"error_history_{i}.png")
```

Figure 9 shows the plot created by this code.

Figure 9 Convergence plot generated by postprocessing of optimization results



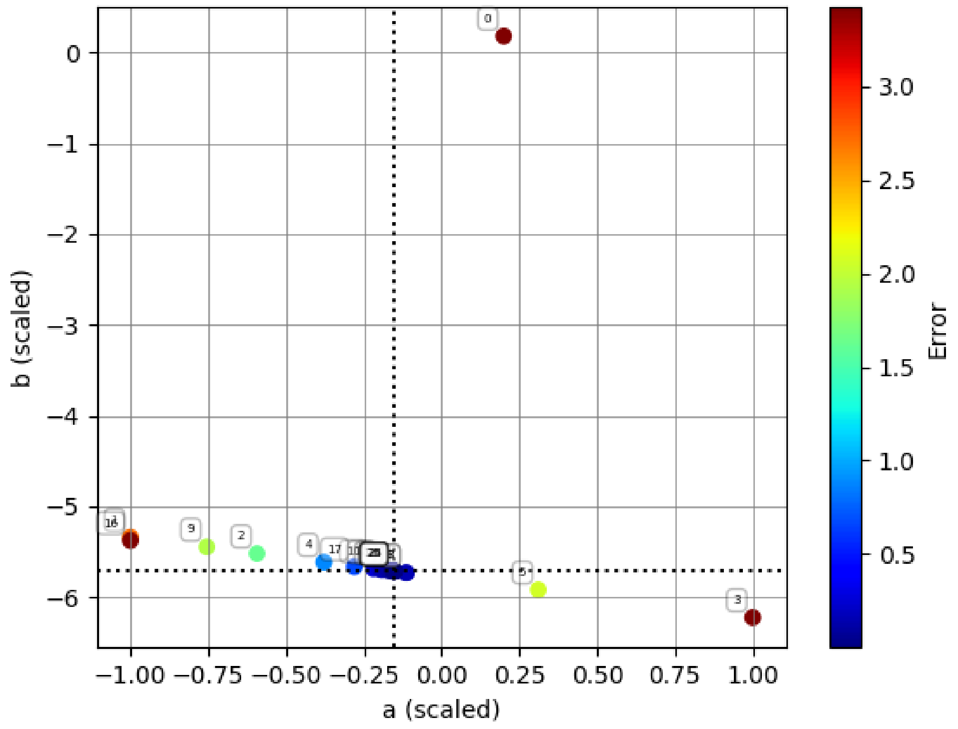
As a more complex example, a plot of the trajectory of the optimization in the space defined by `param1` and `param2` in the first optimization can be created as follows:

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import matplotlib.cm as cm
history = optimizer.history[1]
param1 = "a"
param2 = "b"
fig = plt.figure()
x = history[param1].values[:-1]
y = history[param2].values[:-1]
errors = history["errors"].values[:-1]
```

```
sc = plt.scatter(x, y, c=errors, cmap=cm.jet,
                 vmax=np.percentile(errors[np.isfinite(errors)], 95,
                 interpolation='lower'))
cbar = plt.colorbar(sc)
cbar.set_label("Error")

plt.axvline(history[param1].values[-1], ls=':', c='k')
plt.axhline(history[param2].values[-1], ls=':', c='k')

plt.grid(which='major', color='0.5', linestyle='-', linewidth=0.5)
plt.grid(which='minor', color='0.5', linestyle=':', linewidth=0.25)
for label, xi, yi in zip(range(len(x)), x, y):
   colour = "white"
   plt.annotate(
      label,
      xy=(xi, yi), xytext=(-5, 5), fontsize=5,
      textcoords='offset points', ha='right', va='bottom',
      bbox=dict(boxstyle='round,pad=0.5', fc=colour, alpha=0.25))

plt.xlabel(f"{param1} (scaled)")
plt.ylabel(f"{param2} (scaled)")

plt.savefig(f"trajectory.png")
```

Figure 10 shows the plot created by this code.

Figure 10 Parameter trajectory plot generated by postprocessing of optimization results



After the execution of each task, the history of the parameters is also written to a CSV file so that it can be visualized or further analyzed externally. CSV files are located in the folder `genopt_data` in the `optimizer` node folder, which is `results/nodes/optimizer` if the project has hierarchical organization or `noptimizer_` if it has traditional organization.

The format of these files is as follows. The first column is the evaluation number. This column is followed by one column per optimization parameter. After these, there are two columns per target: one with the name of the target containing its value and one with the name of the target with `-error` appended, which contains the distance to its target. The last column (errors) contains the aggregated error for the evaluation (for example, RMS error).

The values for the targets (and their errors) are enclosed in brackets because they might be lists instead of single values, and this should be taken into account when reading them for analysis. The following table presents an example of the contents of a CSV file from an optimization with parameters `a` and `c` and simulation output `output1` with target value 0:

evaluations	a	c	output1	output-error	errors
0	0.5	0.5	[0.825]	[0.825]	0.825
1	-2.5	0.01	[7.15011]	[7.15011]	7.15011
2	-0.19578	0.201805	[0.36196074]	[0.36196074]	0.361961
3	-0.19578	0.201805	[0.36196074]	[0.36196074]	0.361961
4	0.5	0.019218	[0.55040626]	[0.55040626]	0.550406
5	0.05798	0.0856	[0.28675802]	[0.28675802]	0.286758
6	0.05798	0.0856	[0.28675802]	[0.28675802]	0.286758
7	0.01849	0.082824	[0.2829218]	[0.2829218]	0.282922
8	0.01849	0.082824	[0.2829218]	[0.2829218]	0.282922

Sentaurus Workbench Project After Optimization

The Sentaurus Workbench project is updated and refreshed as the optimization progresses, so all intermediate results produced by the simulation tools are available.

Note:

Since all experiments are added to the project, using hierarchical project organization is recommended, since it allows Sentaurus Workbench to manage experiments faster.

All experiments are added to at least one of the following scenarios:

- `objective_evaluation`: Experiments in the main optimization trajectory
- `jacobian_evaluation`: Experiments used to evaluate the numeric approximation to the Jacobian used to drive the (gradient-based) optimizations

You can hide the `jacobian_evaluation` scenario to see the optimization trajectory more clearly.

There are other special scenarios created by the Optimization Framework:

- `initial` contains the experiments of the initial input file.
- `optimal_*` contains all experiments executed using the optimal set of parameters after each call to `optimizer.optimize()`.
- `restart_*` contains a copy of the `initial` scenario in which the optimization parameters have been replaced by the optimal ones after each call to `optimizer.optimize()`. This can be used for restarting purposes. The difference with `optimal_*` scenarios is that they do not include any modification done to the state parameters during the optimization, and they include experiments that were locked or omitted during an optimization task and, therefore, were not executed.
- `stage_*` contains all experiments executed in a given call to `optimizer.optimize()` or `optimizer.search()`, excluding Jacobian evaluations.

Using the Optimization Framework in a Project Simulation Flow

For some applications of the Optimization Framework, it is of interest to continue the flow using the optimized Sentaurus Workbench parameters. For this purpose, the Optimization Framework can be included as part of the flow. In this case, the optimization will run like any other tool and not through the Sentaurus Workbench **Optimization** menu. The Optimizer tool can be included by choosing **Tool** > **Add** and selecting **genopt** in the **Name** field.

When executed, the Optimization Framework in the flow creates a separate Sentaurus Workbench project that includes only the tool instances that are necessary for the optimization, adding all other tool instances to a Bridge tool instance. This newly created *child project* then runs as previously described. After the child project is finished, it communicates the final optimal parameters to the original Sentaurus Workbench project.

The Optimizer tool instance has two input files:

- The first is a Python script describing the optimization analogous to the one described in previous sections. You can access this file by right-clicking the tool icon and choosing **Edit Input** > **Commands**. The only difference with it is that now it can contain Sentaurus Workbench preprocessor directives.

In addition to standard methods, you can use two methods for the `optimizer` object only when the object is part of the simulation flow:

- `optimizer.link_files_to_parent` (see optimizer.link_files_to_parent on page 52)
- `optimizer.link_optimizer_node` (see optimizer.link_optimizer_node on page 52)

You can use these methods to access specific files and folders of a child project from the Node Explorer of Sentaurus Workbench in the original project, without having to open the child project.

- The second defines which tool instances must be optimized and also which parameters must be updated. You can access this file by right-clicking the tool icon and choosing **Edit Input** > **Configuration**.

  This file is a Python script in which the following variables can be defined:

  - `files_to_copy`: List of files of the parent project to be copied to the child project (for example, files with target data).
  - `folders_to_copy`: List of folders of the parent project to be copied to the child project (for example, a folder containing files with target data).
  - `opt_folder`: Path to a directory in which all child projects will be created and executed. Since Sentaurus Workbench does not recognize projects inside projects, this directory must *not* be an existing Sentaurus Workbench project directory. The default folder is `../optimizations`, that is, a folder named `optimizations` at the same hierarchy level as the current project.
  - `opt_params`: List of parameters that will be updated in the project. If it is not defined, then all optimization parameters are updated. If its value is an empty list, then no parameters are updated in the project.
  - `opt_pattern`: Pattern used to name the child projects generated by the Optimizer tool instance. By default, `<parent_project_name>_<node>_<date>_<time>` is the naming format.
  - `opt_queue`: String of the name of the queue to which simulations of the child project will be submitted. If not given, then the queue used will be the same specified when submitting the node.
  - `opt_tools`: List of tool instance names to be included in the optimization. All parameters from tool instances before the first of these will be fixed to the current value. If it is not defined, then all tools before the current Optimizer tool instance are included unless a previous Optimizer tool instance exists. In this case, all tools between the Optimizer tool instances are included.
  - `restart`: Boolean specifying whether to restart the last run child project. It can also be a full path to the child project to restart (default is `True`). The location of the last run child project can be found in the `.path` file in the node folder.

Note:

Dependency tracking is manual. Therefore, all tool instances that must be run for a particular optimization must be added to the input file.

Child projects are not deleted by default when cleaning up nodes.

References

[1] K. Emancipator and M. H. Kroll, “A Quantitative Measure of Nonlinearity,” *Clinical Chemistry*, vol. 39, no. 5, pp. 766–772, 1993.

[2] M. D. Morris, “Factorial Sampling Plans for Preliminary Computational Experiments,” *Technometrics*, vol. 33, no. 2, pp. 161–174, 1991.

[3] F. Campolongo, J. Cariboni, and A. Saltelli, “An effective screening design for sensitivity analysis of large models,” *Environmental Modelling & Software*, vol. 22, no. 10, pp. 1509–1518, 2007.

3

Optimization Methodology

This chapter presents reference material including all the commands and file formats, and the methodology pertinent to the Optimization Framework.

The optimizer Object

This object is the main driver to create optimization strategies and provides functionality through a series of attributes and methods.

Attributes of optimizer Object

The following attributes can be used for optimizations:

- `optimizer.backend`: Backend optimization object, which is used to perform the optimization in the call to `optimizer.optimize()` or `optimizer.search()`. See Backend Optimization on page 59.
- `optimizer.errors`: Dictionary with the error for the optimal parameters after each call to `optimizer.optimize()` or `optimizer.search()`
- `optimizer.history`: Dictionary with the optimization history for each call to `optimizer` methods
- `optimizer.last_error`: Value of the error from the last call to `optimizer.optimize()` or `optimizer.search()`
- `optimizer.optimization_calls`: Number of calls to `optimizer.optimize()` or `optimizer.search()` that have been performed so far
- `optimizer.parent`: For child optimization projects, dictionary with information about the parent project and the Optimizer node from which the current project was launched
- `optimizer.rsms`: Dictionary with the response surface models (RSMs) created in each call to `optimizer.search()`
- `optimizer.verbosity`: Integer representing the level of output verbosity

Methods of optimizer Object

The following methods are available for the `optimizer` object.

optimizer.evaluate

```
optimizer.evaluate(params, targets, verbosity=None, scenario=None,
                   **kwargs)
```

Creates and runs the experiments defined by the given options, and returns the value of the given targets in a `pandas.DataFrame`. Parameters are:

- `params`: List of optimization parameters.
- `targets`: List of optimization targets.
- `verbosity [None]`: Verbosity level used in the call to `optimize.evaluate()`. If not specified, then the global verbosity level in `optimizer.verbosity()` is used.
- `scenario [None]`: Scenario to which the new experiments are added.
- `**kwargs`: Additional optional parameters for the evaluation:
  - `conditions [{}]`: Dictionary with Sentaurus Workbench parameters to set during the search.
  - `include_points [[]]`: List of dictionaries defining points to be included in the search. It can also be the name of a Sentaurus Workbench scenario whose points are to be included.
  - `method`: A method used to define a design for the evaluation. The method can be `"grid"`, `"latin"`, `"random"`, `"sobol"`, or a Sentaurus Workbench design-of-experiments (DOE) name.
  - `options`: Dictionary with specific options for the method.

optimizer.get_experiment_variables

```
optimizer.get_experiment_variables(variables, filters, scenario)
```

Returns the variable values for experiments in the Sentaurus Workbench project defined by the given filters. Parameters are:

- `variables [None]`: List with names of variables to return.
- `filters [None]`: A dictionary whose keys are parameter names and values tuples with an operator and a value. If the value is not a tuple, then equality is assumed. The operator must be `'lt'`, `'le'`, `'eq'`, `'neq'`, `'ge'`, or `'gt'`.
- `scenario ["all"]`: Name of the scenario to look for experiments.

Returns: A dictionary with variable names and values for each leaf node representing the experiments that verify the given filters

optimizer.get_experiments

```
optimizer.get_experiments(filters, scenario="all")
```

Returns the leaf nodes of experiments in the Sentaurus Workbench project defined by the given filters and scenario. Parameters are:

- `filters [None]`: A dictionary whose keys are parameter names and values tuples with an operator and a value. If the value is not a tuple, equality is assumed. The operator must be `'lt'`, `'le'`, `'eq'`, `'neq'`, `'ge'`, or `'gt'`.
- `scenario ["all"]`: Name of the scenario to look for experiments.

Returns: A list of leaf nodes representing the experiments that verify the given filters

optimizer.get_last_initial

```
optimizer.get_last_initial()
```

Returns the parameter and target values evaluated at the initial point of the last executed task.

Returns: A `pandas.DataFrame` object with the parameter values, target values, and errors for the initial experiment of the last task

optimizer.get_last_optimal

```
optimizer.get_last_optimal(target)
```

Returns the parameter and target values evaluated at the optimal point of the last executed task. The way in which the optimal experiment is selected can be chosen using the `target` argument:

- `target ["errors"]`: Name of the target used for the selection of the optimal experiment. If `"errors"` is selected, then the aggregate of all target errors is used.

Returns: A `pandas.DataFrame` object with parameter values, target values, and errors for the optimal experiment of the last task

optimizer.get_project_as_dataframe

```
optimizer.get_project_as_dataframe()
```

Returns the Sentaurus Workbench project as a `pandas.DataFrame` object for further analysis. It can also be used for exporting to a file from the script.

Returns: A `pandas.DataFrame` object with Sentaurus Workbench parameters as a MultiIndex and variables as columns

optimizer.get_RSM

```
optimizer.get_RSM(params, targets, doe, rsm_type, doe_options={},
                  rsm_options={}, verbosity=None, **kwargs)
```

Returns a response surface model (RSM) for the given parameter and targets. Parameters are:

- `params`: A list of optimization parameters
- `targets`: A list of optimization targets
- `doe`: A method used to select the experiments used to build the RSM, which can be `"grid"`, `"latin"`, `"random"`, `"sobol"`, or a Sentaurus Workbench DOE name
- `rsm_type`: Type of RSM to create, which can be `"gaussian_process"` or `"polynomial"`
- `doe_options [{}]`: Specific options to build the DOE (see Searching the Parameter Space on page 66)
- `rsm_options [{}]`: Specific options to build the RSM (see Response Surface Models on page 75)
- `verbosity [None]`: Verbosity level used in the call to `optimizer.get_RSM`. If not specified, then the global verbosity level in `optimizer.verbosity` is used.
- `**kwargs`: Additional optional parameters for the execution of the experiments:
  - `conditions [{}]`: Dictionary with Sentaurus Workbench parameters to set during the execution of the experiments

Returns: A list of RSMs for the given targets

optimizer.get_targets

```
optimizer.get_targets(callno, iterations=None, targets=None)
```

Returns the target values for the specified task number, iterations, and targets. Parameters are:

- `callno`: Integer representing the task from which to retrieve targets
- `iterations`: A list of iteration numbers whose targets are to be retrieved
- `targets`: A list of optimization targets

Returns: A `pandas.DataFrame` object for the targets in the given task and iterations

optimizer.link_files_to_parent

```
optimizer.link_files_to_parent(file_list, path=".", prefix=None)
```

When using the Optimization Framework in a flow, this method creates a link in the optimizer tool node to the files and folders specified in `file_list` (see Using the Optimization Framework in a Project Simulation Flow on page 45). Parameters are:

- `file_list`: List of files and folders to be linked, given as paths relative to the child project
- `path ["."]`: Path relative to the Optimization Framework node where the links will be created
- `prefix [None]`: Optional prefix to add to the linked file names

Returns: None

optimizer.link_optimizer_node

```
optimizer.link_optimizer_node(prefix=None)
```

When using the Optimization Framework in a flow, this method creates a link in the Optimizer tool node to the child project Optimizer node folder (see Using the Optimization Framework in a Project Simulation Flow on page 45). Parameters are:

- `prefix [None]`: Optional prefix to add to the name of the linked folder

Returns: None

optimizer.load_parameter_history

```
optimizer.load_parameter_history(stage, parameters)
```

Loads the values of the given parameters at the given task of the optimization. Parameters are:

- `stage`: Optimization task at which to recover the parameter values
- `parameters`: A list of parameters whose values you want to recover

Returns: A list of dictionaries with the values for the specified parameters

optimizer.load_targets

```
optimizer.load_targets(callno, iterations=None, targets=None)
```

Returns the target values for the specified task number, iterations, and targets of a previously executed optimization. The values will be loaded from the history files in the folder `genopt_data` inside the optimizer node folder. Parameters are:

- `callno`: Integer representing the task from which to retrieve targets
- `iterations`: A list of iteration numbers whose targets are to be retrieved
- `targets`: A list of optimization targets

Returns: A `pandas.DataFrame` object for the targets in the given task and iterations

optimizer.lock_experiments

```
optimizer.lock_experiments(filters)
```

Locks experiments in the Sentaurus Workbench project so that they are not used in the optimization. Parameters are:

- `filters`: A dictionary containing values for Sentaurus Workbench parameters. All experiments with values matching those in the dictionary are locked.

The following example locks all experiments whose value for `"Type"` is `"nMOS"`:

```
optimizer.lock_experiments({"Type":"nMOS"})
```

See optimizer.unlock_experiments on page 59.

Returns: None

optimizer.optimize

```
optimizer.optimize(params, targets, verbosity=None, **kwargs)
```

Calls the optimization backend for the given parameters and targets. For details about specific backends, see Backend Optimization on page 59. Parameters are:

- `params`: A list of optimization parameters.
- `targets`: A list of optimization targets.
- `verbosity [None]`: Verbosity level used in the call to `optimize.get_RSM`. If not specified, then the global verbosity level in `optimizer.verbosity` is used.

- `**kwargs`: Additional optional parameters for the optimization:
    - `plotting [{"convergence":True, "parameters":"all"}]`: A dictionary with the following options:

      `"convergence"` specifies whether to create a plot of the convergence.

      `"parameters"` is a list with one or two tuples that define which parameters to plot by their dimension number. For example, `"parameters": [(0, 1), (1,)]` creates a 2D plot with parameters `params[0]` and `params[1]`, and a 1D plot with the parameter `params[1]`. If `"parameters":"all"` is passed, then all 1D and 2D plots are created.
    - `print_sensitivity [False]`: Specifies whether to print a summary of the sensitivities calculated from the evaluations of the Jacobian during the optimization.
    - `initial_search [{}]`: A dictionary that defines an initial search in the parameter space before selecting a starting point for the optimization backend. The dictionary has the following keys:

      `method` specifies which method to use for the search. It can be `"bayesian_optimizer"`, `"grid"`, `"latin"`, `"random"`, or a Sentaurus Workbench DOE name.

      `polish` specifies whether to run the optimization backend using the best result as the starting point. If `polish` is set to `False`, then the local minimization is not executed. Therefore, in this case, the behavior of `optimizer.optimize()` is equivalent that of `optimizer.search()`.

      `options` lists specific options for `method` (see Searching the Parameter Space on page 66).

Returns: A dictionary with the optimal value of the optimization parameters

optimizer.print_summary

```
optimizer.print_summary(stage="all")
```

Prints a summary of the results of the given task. Parameters are:

- `stage ["all"]`: Task number whose summary you want to print

Returns: None

optimizer.run_experiment

```
optimizer.run_experiment(params, targets, verbosity=None, **kwargs)
```

Creates and runs an experiment defined by the given values of the parameters, and returns the value of the given targets. Parameters are:

- `params`: A list of optimization parameters that define the experiment
- `targets`: A list of optimization targets whose values are calculated
- `verbosity [None]`: Verbosity level used in the call to `optimize.get_RSM`. If not specified, then the global verbosity level in `optimizer.verbosity` is used.
- `**kwargs`: Additional optional parameters:
    - `conditions [{}]`: Dictionary with Sentaurus Workbench parameters to set in the experiment
    - `scenario [None]`: Scenario to which the new experiment will be added

Returns: A list with the values of the given targets

optimizer.run_screening_analysis

```
optimizer.run_screening_analysis(params, targets, verbosity=None,
                                 **kwargs)
```

This is an alias to the `optimizer.screening_analysis()` method (see optimizer.screening_analysis).

optimizer.run_sensitivity_analysis

```
optimizer.run_sensitivity_analysis(params, targets, verbosity=None,
                                   **kwargs)
```

This is an alias to the `optimizer.sensitivity_analysis()` method (see optimizer.sensitivity_analysis on page 57).

optimizer.screening_analysis

```
optimizer.screening_analysis(params, targets, verbosity=None, **kwargs)
```

Screens the most influential parameters based on a polynomial RSM of the objective function. Parameters are:

- `params`: A list of optimization parameters for which the sensitivity is calculated
- `targets`: A list of optimization targets whose sensitivity is calculated

- `verbosity [None]`: Verbosity level used in the call to `optimize.get_RSM`. If not specified, then the global verbosity level in `optimizer.verbosity` is used.
- `**kwargs`: Additional optional parameters for screening analysis:
    - `doe ["plackettBurman"]`: DOE used to create the polynomial RSM.
    - `doe_params [{}]`: Parameters for the creation of the DOE if required.
    - `order [1]`: Order of the polynomial RSM.
    - `cost [None]`: Cost function used to aggregate several outputs of the target function; if it is not given, the method creates one RSM per output.
    - `screening_criterion ["average"]`: Specifies how to aggregate multiple outputs: `"average"` uses the mean of the coefficients for a given RSM term, and `"local_strong"` uses the maximum of the coefficients for a given RSM term.
    - `screen_range [10]`: Influence threshold to select RSM terms.
    - `screen_best [None]`: If given, the number of terms from the RSM to select; it ignores the value given in `screen_range`.
    - `normalise_responses [False]`: Specifies whether to normalize the outputs of the target function to [-1, 1].
    - `weight_responses [None]`: If given, a list with weights for each output of the target function.

Returns: An object of type `ScreeningAnalysisResult` including the most influential parameters, influences, and RSMs used for the calculation of influences, and the evaluations used to fit the RSM (see Screening Analysis on page 28)

optimizer.search

```
optimizer.search(params, targets, method, options, verbosity=None,
                 **kwargs)
```

Performs a search in the optimization parameter space (see Searching the Parameter Space on page 66). Parameters are:

- `params`: A list of optimization parameters
- `targets`: A list of optimization targets
- `method`: A method used to perform the search, which can be `"bayesian_optimizer"`, `"grid"`, `"latin"`, `"random"`, `"sobol"`, `"turbo"`, or a Sentaurus Workbench DOE name
- `options`: Dictionary with specific options for the method

- `verbosity [None]`: Verbosity level used in the call to `optimize.get_RSM`. If not specified, then the global verbosity level in `optimizer.verbosity` is used.
- `**kwargs`: Additional optional parameters for the search:
  - `conditions [{}]`: Dictionary with Sentaurus Workbench parameters to set during the search.
  - `run_predicted_minimum [False]`: Specifies whether to run a final simulation at the minimum of an RSM of the data.
  - `do_plots [True]`: Specifies whether to create plots of slices of an RSM of the data.
  - `include_points [[]]`: List of dictionaries defining points to be included in the search. It can also be the name of a Sentaurus Workbench scenario whose points are to be included.
  - `include_nominal [False]`: Specifies whether to add the nominal point (that is, the current value of the optimization parameters) to the selected design.

Returns: A dictionary with the best value of the optimization parameters found in the search

optimizer.sensitivity_analysis

```
optimizer.sensitivity_analysis(params, targets, verbosity=None,
                               **kwargs)
```

Calculates the sensitivity of the optimization parameters `params` on the optimization targets `targets`. The sensitivity is given in the unit of the optimization target and as a percentage of the center target value.

Parameters passed to the method are:

- `params`: A list of optimization parameters for which the sensitivity is calculated
- `targets`: A list of optimization targets whose sensitivity is calculated
- `verbosity [None]`: Verbosity level used in the call to `optimize.get_RSM`. If not specified, then the global verbosity level in `optimizer.verbosity` is used.
- `**kwargs`: Additional optional parameters for sensitivity analysis:
  - `bounds`: Defines the domain for the sensitivity analysis explicitly. It must be a list with as many elements as parameters, with each element being a list containing the required lower and upper bounds of the parameter. The current value is still used as the midpoint to calculate the sensitivity analysis measures.
  - `direction ["both"]`: For local sensitivity analysis, specifies the directions for the increment of each parameter. It can be `"both"`, `"backward"`, or `"forward"`. If you specify `"both"`, then `npoints` must be an odd number.

- `do_plots [True]`: Specifies whether to create plots from the sensitivity analysis
- `method ["local"]`: Specifies whether to use local (`"local"`) or global (`"global"`) sensitivity analysis.
- `nlevels [None]`: If you set `method="global"`, then this parameter defines the number of possible levels each parameter can take.
- `npoints [3]`: Number of points per parameter to simulate within the given range
- `ntrajectories [None]`: If you set `method="global"`, then this parameter defines the number of different trajectories used for the analysis.
- `percentage_range [10]`: Range of the domain to be used for the sensitivity analysis
- `percentage_value`: Range of the domain to be used for the sensitivity analysis defined as a given percentage of the current parameter values.

Returns: An object of type `SensitivityAnalysisResult` that includes sensitivities, nonlinearities, and evaluations for the given options (see Sensitivity Analysis on page 31)

optimizer.set_cost_function

```
optimizer.set_cost_function(name)
```

Sets the cost calculator used to drive optimizations. Parameters are:

- `name`: A name of the cost calculator, which must be `"arctan"`, `"cauchy"`, `"huber"`, `"l1"`, `"rmsd"`, `"soft_l1"`, `"squared"`, or `"sum"`

Returns: None

optimizer.set_from_swb

```
optimizer.set_from_swb(parameters)
```

Sets the values of the optimization parameters to the current value in the Sentaurus Workbench project. Parameters are:

- `parameters`: A list of optimization parameters whose value should be changed to the value in the Sentaurus Workbench project

Returns: None

optimizer.set_reference_experiments

```
optimizer.set_reference_experiments(nodes)
```

Sets the reference experiments for the optimization. It can be used, for example, to optimize some experiments from an initial split sequentially or to restart an optimization from a given point. Parameters are:

- `nodes`: A list of leaf nodes defining the experiments to use as the base for the optimization

Note:

This function does NOT redefine the *initial* scenario experiments, so the nodes will be removed if the optimization project is cleaned.

Returns: None

optimizer.unlock_experiments

```
optimizer.unlock_experiments(filters)
```

Unlocks experiments in the Sentaurus Workbench project so that they are used in the optimization. Parameters are:

- `filters`: A dictionary containing values for Sentaurus Workbench parameters. All experiments with values matching those in the dictionary are unlocked.

The following example unlocks all experiments whose value for `"Type"` is `"nMOS"`:

```
optimizer.unlock_experiments({"Type":"nMOS"})
```

See optimizer.lock_experiments on page 53.

Returns: None

Backend Optimization

The optimization process can be driven by one of the available backend optimizers:

- `"least_squares"` interfaces optimization functions using the same interface as that of the SciPy `least_squares` backend.
- `"minimize"` uses the SciPy `minimize` backend.
- `"mkl_nls"` uses the Intel MKL bounded trust region implementation for least squares problems.
- `"shgo"` uses the simplicial homology global optimization (SHGO) based on the SciPy implementation of the algorithm.

Backend options can be set using methods of the `optimizer.backend` attribute.

Setting the Backend Optimizer

To set the backend optimizer, specify:

```
optimizer.backend.set_method(backend_name)
```

where `backend_name` can be one of the following:

- `"least_squares"` accesses least squares algorithms, so it naturally works on multiple outputs, minimizing the squared residual between curves. It is a good choice for curve fitting.
- `"minimize"` accesses gradient-based and nongradient-based minimizers for single output functions, for example, root mean square (RMS) error between two curves.
- `"mkl_nls"` accesses the Intel MKL implementation of nonlinear least squares methods using a bounded trust region method.
- `"shgo"` accesses a parallel implementation of SHGO. This method runs an initial search and local optimizations starting from selected points of the search. Therefore, do not use this backend optimizer in combination with the `initial_search` argument of the `optimizer.optimize` method.

Setting Common and Backend-Specific Optimization Parameters

To set common and backend-specific optimization parameters, specify:

```
optimizer.backend.set_optimization_parameters(jacobian_options,
                                              method, options)
```

where:

- `jacobian_options` is a dictionary with keyword arguments passed to the Jacobian evaluation. Options are:
  - `"eps"` sets the step size.
  - `"step_type"` specifies how this step is applied. It can be `"abs"` (default, absolute value of the step regardless of the value of the parameter) or `"rel"` (relative step to the current value of the parameter).
- `method` selects the optimization method used by the backend. Options are:
  - `"least_squares"`: Options are `"trf"` (trust region, default), `"dogbox"`, or `"lm"` (Levenberg–Marquardt, unbounded only).

- `"minimize"`: Derivative-based options are `"L-BFGS-B"` (default), `"BFGS"`, `"CG"`, `"SLSQP"`, or `"TNC"`.

  Derivative-free options are `"COBYLA"`, `"Nelder-Mead"`, or `"Powell"`.

  The `"Nelder-Mead"` method is unbounded, that is, it does not respect the bounds set in the optimization parameter definition.
- `"mkl_nls"`: Options are `"bounded_trust_region"` (default) for bounded problems or `"trust_region"` for unbounded problems.
- `"shgo"`: Options are `"COBYLA"` (derivative free) or `"SLSQP"` (derivative based).

Examples

Use the `"minimize"` backend with the `"L-BFGS-B"` solver and a relative step of 1e-3 for the derivatives:

```
optimizer.backend.set_optimization_parameters(
                              kwargs={"eps":1.e-3, "step_type": "rel"},
                              method='L-BFGS-B')
```

Specific options for the method are passed through the `"options"` dictionary. For example, to use a stopping tolerance for the gradient and the objective of 1e-3, specify:

```
optimizer.backend.set_optimization_parameters(kwargs={"eps":1.e-3},
                              method='L-BFGS-B',
                              options={"gtol": 1e-3, "ftol": 1e-3})
```

Use the `"least_squares"` backend with the default method (trust region), an absolute step for the Jacobian of 1e-3, and a maximum of 10 iterations:

```
optimizer.backend.set_optimization_parameters(kwargs={"eps":1.e-3},
                                              maxiter=10)
```

Note:

You can change the backend at any point during the optimization (for example, between two successive calls to `optimizer.optimize()`, you can change the backend from `"minimize"` to `"least_squares"`). However, you must update the corresponding backend optimization parameters to ensure they are appropriate for the new backend optimizer.

Method-Specific Optimization Options

This section describes the method-specific optimization options.

least_squares

All methods for the `least_squares` backend have the same options.

Table 3 Options for least_squares backend

Option	Default	Description
`ftol`	`1e-08`	Stopping criterion by the *relative* change of the cost function.
`gtol`	`1e-08`	Stopping criterion based the norm of the gradient.
`xtol`	`1e-08`	Stopping criterion by the *relative* change of the independent variable.
`loss`	`'linear'`	Loss function, which is one of the following: • `'arctan'`: Limits the maximum loss on any single residual. • `'cauchy'`: Severely weakens influence of outliers, but might cause difficulties in optimization process. • `'huber'`: Similar to `'soft_l1'`. • `'linear'`: Gives a standard least-squares problem. • `'soft_l1'`: Smooth approximation of l1 (absolute value) loss. Usually a good choice for robust least squares. Method `'lm'` supports only `'linear'` loss.
`max_nfev`	`None`	Maximum number of function evaluations excluding calls to the Jacobian.
`verbose`	`0`	Verbosity level, which can be from `0` (silent) to `2` (display information every iteration).

minimize

The variety of methods for the `minimize` backend means that options vary from method to method.

Note:

The symbol `inf` used in the options for the BFGS and CG solvers is defined in the `numpy` library module, so it can be accessed as `np.inf`. For example, to use the BFGS solver with `norm -inf`, specify:

```
optimizer.backend.set_optimization_parameters(kwargs={"eps":1.e-3},
                        method="BFGS", options={"norm": -np.inf})
```

Table 4 Options for minimize backend

Option	Default	Description
Common to all methods		
`disp`	`0`	Verbosity level.
`maxiter`	`None`	Maximum number of iterations.
BFGS		
`gtol`	`1e-5`	Stopping criterion based on the norm of the gradient.
`norm`	`inf`	Order of the norm: • `inf` takes the largest element. • `-inf` takes the smallest element.
CG		
`gtol`	`1e-5`	Stopping criterion based the norm of the gradient.
`norm`	`inf`	Order of the norm: • `inf` takes the largest element. • `-inf` takes the smallest element.
COBYLA		
`catol`	`2e-4`	Tolerance (absolute) for constraint violations.
`rhobeg`	`1.0`	Reasonable initial changes to the variables.
`tol`	`1e-4`	Final accuracy in the optimization (not precisely guaranteed).
L-BGFS-B		
`ftol`	`2.2e-9`	Stopping criterion by the *relative* change of the objective function.
`gtol`	`1e-5`	Stopping criterion based on the norm of the gradient.
`maxcor`	`10`	Maximum number of variable metric corrections used to define the limited memory matrix.
`maxfun`	`15000`	Maximum number of function evaluations.

Table 4 Options for minimize backend (Continued)

Option	Default	Description
`maxls`	`20`	Maximum number of line search steps (per iteration).
Nelder-Mead		
`fatol`	`1e-4`	Stopping criterion based on the *absolute* change in the objective between iterations.
`initial_simplex`	`None`	Initial simplex given as an array with the coordinates of each vertex in different rows.
`maxfev`	`None`	Maximum number of function evaluations.
`xatol`	`1e-4`	Stopping criterion based on the *absolute* change of x between iterations.
Powell		
`direc`	`None`	Initial set of direction vectors.
`ftol`	`1e-4`	Stopping criterion based on the *relative* change in the objective between iterations.
`maxfev`	`None`	Maximum number of function evaluations.
`xtol`	`1e-4`	Stopping criterion based on the *relative* change of x between iterations.
SLSQP		
`ftol`	`1e-6`	Precision goal for the value of *f* in the stopping criterion.
TNC		
`eta`	`-1`	Severity of the line search. If `eta` < 0 or `eta` > 1, set to 0.25.
`ftol`	`0`	Stopping criterion by the *relative* change of the objective function.
`gtol`	`-1`	Stopping criterion based on the norm of the projected gradient.
`maxCGit`	`-1`	Maximum number of Hessian-vector product evaluations per main iteration.

Table 4 Options for minimize backend (Continued)

Option	Default	Description
`minfev`	`0`	Minimum function value estimate.
`offset`	`None`	Value to subtract from each variable. If `None`, the offsets are (lower + upper)/2 for bounded parameters and x for unbounded ones.
`rescale`	`-1`	Scaling factor (in log10) used to trigger *f* value rescaling: • If `0`, rescales at each iteration. • If a large value, never rescales. • If `rescale` < 0, `rescale` is set to 1.3.
`scale`	`None`	Scaling factors to apply to each variable. If `None`, the factors are (upper – lower) for bounded variables and 1+|x| for unbounded ones.
`stepmx`	`0`	Maximum step for the line search.
`xtol`	`sqrt(machine precision)`	Stopping criterion based on the *relative* change of x between iterations.

mkl_nls

All methods for the `mkl_nls` backend have the same options.

Table 5 Options for mkl_nls backend

Option	Default	Description
`atol`	`1e-08`	Absolute tolerance for stopping. Applied as default to all stopping criteria.
`delta`	`1`	Initial size of the trust region.
`ftol`	`1e-08`	Stopping criterion for the 2-norm of the residuals and change of the 2-norm of the residuals after a trial step.
`gtol`	`1e-08`	Stopping criterion for the 2-norm of the Jacobian matrix.
`max_nfev`	`None`	Maximum number of function evaluations excluding calls to the Jacobian.
`trtol`	`1e-08`	Stopping criterion for the trust region radius.

Table 5 Options for mkl_nls backend (Continued)

Option	Default	Description
`verbose`	`0`	Verbosity level, which can be from `0` (silent) to `2` (display information every iteration).
`xtol`	`1e-08`	Stopping criterion for the 2-norm of the trial step.

shgo

All methods for the `shgo` backend have the same options.

Table 6 Options for shgo backend

Option	Default	Description
`iters`	`None`	Number of iterations used to construct the simplicial complex.
`minimizer_kwargs`	`{}`	Dictionary with options for the local minimization method. Same options as defined in Table 4 on page 63 for `"COBYLA"` and `"SLSQP"`.
`n`	`None`	Number of points used to construct the simplicial complex.
`options`	`{}`	Dictionary with options for the solver. Options are: • `f_min`: Minimum of the target if known • `f_tol`: Precision in the function value • `maxfev`: Limits the maximum number of function evaluations • `maxtime`: Maximum-allowed running time

Searching the Parameter Space

The `optimizer.search` method provides options for searching the parameter space.

Grid Sampling

Grid sampling creates a tensor-product grid from 1D uniform grids in each parameter dimension. Since the number of samples grows exponentially with the number of parameters, it is usually reserved for low-dimensional problems. The result of grid sampling is also known as a full factorial design.

Options for grid sampling are:

- `boundary [False]`: Specifies whether the design includes the boundary.
- `location ["equidist"]`: Selects whether points in each dimension are chosen at equidistant (`"equidist"`) or random (`"random"`) locations.
- `midpoint [False]`: Specifies whether the design includes the midpoint.
- `Ns`: Number of samples per dimension. If an integer is given, then it is used for all dimensions. A tuple with a number of samples per dimension can also be given.

Latin Hypercube Sampling

Latin hypercube sampling generates near-random samples in the parameter space. A grid is constructed in the space, and samples are generated such that they are the only ones in the axis-aligned hyperplane containing them. Options for Latin hypercube sampling are:

- `Ns`: Number of samples.

Random Sampling

Random sampling creates samples distributed randomly in the parameter space. Options for random sampling are:

- `Ns`: Number of samples.

Sampling Based on Sobol Sequences

Sobol sequences can be used to generate a sampling with a lower discrepancy than purely random sampling, that is, the samples obtained from a Sobol sequence cover the space more uniformly.

However, this property only holds if sequences of size 2^n are used (for example, 32 or 128). If this is not possible or convenient for your application, then another sampling method might be more appropriate.

Options for Sobol sampling are:

- `Ns`: Number of samples.
- `scramble [False]`: Specifies whether to apply Owen (nested) binary scrambling to the design.
- `skip [1]`: Number of points of the sequence to omit. It can be used to generate points incrementally.

If `scramble=False`, then setting `skip=0` generates a point at the lower bound of all parameters.

Bayesian Optimization

Bayesian optimization performs an adaptive search in the parameter space guided by the current available knowledge from all the previously performed experiments. It starts from a Latin hypercube sampling of the space to construct the initial Gaussian process RSM. In each iteration, an acquisition function (*expected improvement*) is evaluated, and new points are selected that could potentially improve on the current best solution.

Options for the Bayesian optimization are:

- `acquisitions [None]`: List of acquisition functions used to generate candidate solutions in each iteration.
- `data_noise [1e-5]`: Expected noise of the simulation results.
- `do_plots [False]`: Specifies whether to plot the Gaussian process RSM (or 1D and 2D slices if there are three or more parameters).
- `hyperparam_mode ["fixed"]`: Selects a mode of setting hyperparameters for the kernel. It can be one of the following:
    - `"fixed"` retains the initial value throughout.
    - `"ml-ii"` selects the values that maximize the likelihood after each iteration.
- `include_points [[]]`: A list of dictionaries containing the value for optimization parameters that should be included in the initial search in addition to the points generated using Latin hypercube sampling.
- `kernel [Matern32]`: Selects a kernel to use in the construction of the Gaussian process RSM (see Kernels).
- `length_scale [domain diameter/5]`: Length scale for kernels that use it (including the default one).
- `max_iter [6]`: Maximum number of iterations.
- `n_acquisitions [2]`: Number of acquisition function evaluations per iteration.
- `Ns`: Number of samples for the initial Latin hypercube sampling search.
- `tag [""]`: String to append to the names of plots created.
- `transform [None]`: Transform to apply to the *optimization target*.

Kernels

To select a kernel other than the default, the option `"kernel"` must be passed with a kernel instance.

Table 7 lists available kernels. By default, a new kernel instance applies to all dimensions, but you can apply kernels to specific dimensions only and then compose the kernels.

For example, to create a kernel that applies to dimensions 0 and 2, and another kernel that applies to dimension 1, and then combine them into a kernel that applies to dimensions 0, 1, and 2, specify:

```
kernel02 = Matern32(l_scale=[1, 1], dimensions=[0,2])
kernel1 = Matern52(l_scale=[1], dimensions=[1])
sum_kernel = kernel02 + kernel1
```

The kernel instance `sum_kernel` can then be passed to the option `"kernel"` for the Bayesian optimization.

Table 7 Available kernel classes and their hyperparameters

Kernel class [hyperparameters]	**Description**
`Matern [nu, l_scale]`	Matern kernel of order `nu` with length scale `l_scale`. When selecting a Matern kernel and choosing to optimize the hyperparameters, the order `nu` of the Matern kernel will also be optimized. Example: `"kernel": Matern(nu=0.5, l_scale=[1, 1, 1])`
`Matern32 [l_scale]`	Matern kernel of order 3/2 with length scale `l_scale`. If `l_scale` is given as a scalar, an isotropic kernel will be maintained during the hyperparameter optimization. Example: `"kernel": Matern32(l_scale=[1, 1, 1])`
`Matern52 [l_scale]`	Matern kernel of order 5/2 with length scale `l_scale`. If `l_scale` is given as a scalar, an isotropic kernel will be maintained during the hyperparameter optimization. Example: `"kernel": Matern52(l_scale=[1, 1, 1])`
`NeuralNetwork [sigma0, sigma]`	Neural network kernel using the error function (erf) as the transfer function. `sigma0` controls the variance of the bias, and `sigma` is the variance of the weights of the neural network. Example: `"kernel": NeuralNetwork(sigma0=1, sigma=[1, 1, 1])`
`SquaredExponential [l_scale]`	Squared exponential kernel with length scale `l_scale`. If `l_scale` is given as a scalar, an isotropic kernel will be maintained during the hyperparameter optimization. Example: `"kernel": SquaredExponential(l_scale=[1, 1, 1])`

Acquisition Functions

To select an acquisition function other than the default (`ExpectedImprovement`), you can use the `acquisitions` parameter, which takes a list of the acquisition functions to be used to generate the candidate solutions in every iteration. If the acquisition functions are passed explicitly, then do not use the `n_acquisitions` parameter.

Table 8 Acquisition functions and options required

Acquisition function	Option	Description
`ExpectedImprovement`	`xi`	Expected improvement acquisition function. The parameter `xi` sets the bias to modify the current best in the calculation of the expected improvement.
`ExpectedImprovementConstrained`	`xi`	Variation of the expected improvement acquisition function than can handle the use of `OptConstraint` objects. The parameter `xi` sets the bias to modify the current best in the calculation of the expected improvement.
`ProbabilityOfImprovement`	`xi`	Probability of improvement acquisition function. The parameter `xi` sets the bias to modify the current best in the calculation of the probability of improvement.
`UpperConfidenceBound`	`ci`	Upper confidence bound acquisition function. The parameter `ci` sets the confidence interval level at which to evaluate the upper confidence bound.

Before using acquisition functions in an Optimization Framework script, the classes must be imported. For example:

```
from common.optimizers.acquisition_function
import ExpectedImprovement, UpperConfidenceBound,
ProbabilityOfImprovement
```

For example, the following command can be used to generate three candidates, each from the upper confidence bound acquisition function, with parameter values of 75, 80, and 90:

```
opt_params = optimizer.search(params, targets,
               method="bayesian_optimizer",
               options={"Ns": 10, "max_iter": 3,
                        "acquisitions": [UpperConfidenceBound(75),
                                         UpperConfidenceBound(80),
                                         UpperConfidenceBound(90)]})
```

You can mix different families of acquisition functions. For example, to generate one candidate from each family, specify:

```
opt_params = optimizer.search(params, targets,
               method="bayesian_optimizer",
               options={"Ns": 10, "max_iter": 3,
                        "acquisitions": [UpperConfidenceBound(75),
                                         ExpectedImprovement(0),
                                         ProbabilityOfImprovement(0)]})
```

Note:

For constrained problems, only the `ExpectedImprovementConstrained` acquisition function can be used.

Trust Region Bayesian Optimization

Trust region Bayesian optimization (TuRBO) [1] performs an adaptive search in the parameter space guided by the current available knowledge from all of the previously performed experiments. It starts from several independent Latin hypercube samplings of the space to construct the initial Gaussian process models. In each iteration, a sample of each model is evaluated and minimized, and the best points are selected for evaluation (Thompson sampling).

The new evaluation affects only the model that led to it, so the different models compete for the new evaluations and only the ones producing the best minima are updated. At the same time, if a model does not improve its solution over several iterations, then its search space increases to provide an opportunity to find a better region. On the other hand, if a model improves its best solution, then the search space shrinks around its best point to further refine it.

Because several independent models can refine their search space, the convergence to a better local minimum at the end of the simulation is more likely, and it can help simplify more complex optimization strategies required using other algorithms. However, because several models are used for the search space, the execution times of the algorithm can be high as the number of experiments grows.

Note:

Even if the number of trust regions is set to one, the algorithm still differs from Bayesian optimization in the following ways:

- The single trust region in which the search occurs can change size during the optimization.
- By default, the selection of candidates is performed by using Thompson sampling, even though this can be changed to another acquisition function as used by Bayesian optimization.

Options for TuRBO are:

- `acqs_options [None]`: Dictionary with options to define the acquisition functions to generate candidate samples.
- `acquisition [Thompson]`: Acquisition function to generate new candidate points. Valid options are `ExpectedImprovement`, `ProbabilityOfImprovement`, `Thompson`, and `UpperConfidenceBound`.
- `batch_size [2]`: Number of new experiments executed per iteration
- `disp [0]`: Verbosity level during execution of the algorithm. Values can be 0, 1, or 2
- `do_plots [False]`: Specifies whether to plot the Gaussian process RSMs (or 1D and 2D slices if there are three or more parameters). These plots are stored in the `gp_results` folder in the root directory of the project and are generated per trust region. They are independent of the ones created by the `do_plots` option of `optimizer.search()`.
- `max_iter [6]`: Maximum number of iterations.
- `Ns`: Number of samples for the initial Latin hypercube sampling search for each trust region. The total number of initial experiments is `Ns` × `n_tr`.
- `tag [""]`: String to append to the names of the plots created.
- `transform [None]`: Transform to apply to the optimization target.
- `trust_regions [1]`: Number of trust regions used by the algorithm.

Acquisition Functions

When using TuRBO, the default way to generate new candidates is by using Thompson sampling. However, it is possible to generate new candidates using one of the acquisition functions used for standard Bayesian optimization. In this case, all candidate points must be generated from the same family of acquisition functions.

Therefore, to change the default, both the `acquisition` and `acq_options` options should be used together. Table 8 on page 70 summarizes the available functions and the options required.

For example, the following command can be used to generate three candidates, each from the `ExpectedImprovement` acquisition function, with parameter values of 0, 0.5, and 1:

```
opt_params = optimizer.search(params, targets, method="turbo",
                options={"Ns": 10, "trust_regions": 1,
                         "batch_size": 3, "max_iter": 3,
                         "acquisition": ExpectedImprovement,
                         "acqs_options": {"xi": [0, 0.5, 1.0]}} )
```

Note:

The options of the acquisition function must have at least as many elements as the requested batch size. As in the case of Bayesian optimization, only the default (Thompson sampling) and the `ExpectedImprovement` acquisition function support constrained optimizations.

Sentaurus Workbench Design-of-Experiments

The `optimizer.search()` method provides a simple interface to construct DOEs using Sentaurus Workbench. See the *Sentaurus™ Workbench User Guide*.

Note:

Taguchi, Latin square, and Greco-Latin square designs cannot be used from the Optimization Framework. You must set up these designs from the graphical user interface of Sentaurus Workbench.

The following DOEs do not require options:

- Box–Behnken design
- Central composite designs:
  - `circumscribedCentralComposite`
  - `facedCentralComposite`
  - `inscribedCentralComposite`
  - `midPointDesign`
  - `orthogonalCentralComposite`
  - `smallCentralComposite`
- Factorial designs:
  - `exFactorial`
  - `fullFacTwoLev`
  - `halfFactorialMinus`
  - `halfFactorialPlus`
  - `octFactorial`
  - `quartFactorial`

- Plackett–Burman designs:
  - `foldedPlackettBurman`
  - `plackettBurman`

Options to Sentaurus Workbench DOEs are passed using as the keyword the position in which they should be passed to the Sentaurus Workbench DOE generator. Sentaurus Workbench DOEs that accept options are:

- `customCentralComposite`:
  - `"0"`: Alpha, distance of the *star points* from the center of the design.
- `customDesign` (constructs the D-optimal design for the given model and number of experiments):
  - `"0"`: Number of experiments.
  - `"1"`: Model given as a list of coefficients (Tcl format). This is therefore a list of lists, where each sublist has as many terms as parameters.
- `diagonalDesign`:
  - `"0"`: Number of experiments.
- `fractFactorial2` [3–11 parameters]:
  - `"0"`: Specifies the design, which can be `"fraction"`, `"numExper"`, or `"resolution"`.
  - `"1"`: Value represents the way in which the designed is specified.
- `fullFacNLev`:
  - `"0"`: Specifies whether points in each dimension are chosen equidistant (`"equidist"`) or at random (`"random"`).
  - `"1"`: Specifies whether the design includes the boundaries (`0` or `1`).
  - `"2"`: Specifies whether the design includes the midpoint (`0` or `1`).
  - `"3"`: Number of levels of the design.

  Note:
  The Optimization Framework can use only the same options for every parameter. However, full control over each parameter can be recovered using the `"grid"` search method, which is also a full factorial design.
- `hybrid` [3, 4, or 6 parameters]:
  - `"0"`: Design name, which can be `"D310"`, `"D311"`, `"D416"`, or `"D628"`, and can be used for three, three, four, or six parameters, respectively.

- `randomDesign`:
  - `"0"`: Number of experiments.

Examples

Search for the D-optimal design (`customDesign`) with seven experiments for three parameters and a quadratic model:

```
optimizer.search([a, b, c], target, method="customDesign",
          options={"0": 7, "1": "{{0 0 0} {1 0 0} {0 1 0} {0 0 1}
                 {1 1 0} {1 0 1} {0 1 1} {2 0 0} {0 2 0} {0 0 2}}"})
```

Search for a two-level full factorial design:

```
optimizer.search([a, b, c], target, method="fullFacTwoLev")
```

Search for a D311 hybrid design:

```
optimizer.search([a, b, c], target, method="hybrid",
                options={"0": "D311"})
```

Response Surface Models

The `optimizer.get_RSM` method provides options for creating a response surface model (RSM) from a given DOE in the parameter space. DOE methods and options are the same as those available for `optimizer.search`. The types of supported RSM are polynomial and Gaussian process.

Since any number of targets can be provided, the `optimizer.get_RSM` method returns an `RSMList` object, which provides the methods described in Table 9.

Table 9 Methods of the RSMList object

Method	Arguments	Description
`evaluate`	`x [, return_std=False, return_cov=False]`	Returns a list with the evaluation of the RSM for each target at points `x`. For Gaussian process RSMs, you can obtain the standard deviation of the prediction at each point in `x` by setting `return_std=True`, or the full covariance matrix of the predictions at points `x` by setting `return_cov=True`.
`update`	`x,y`	Updates the RSMs with evaluations in points `x` with target values `y`.

Table 9 Methods of the RSMList object (Continued)

Method	Arguments	Description
`write_to_csv`	`fname [, nx=100, dim=0, reference_point=None]`	Writes a `.csv` file with a 1D sample of `nx` points along dimension `dim`. The value of the remaining fixed dimensions can be specified by setting `reference_point` as a point that the 1D line goes through.
`write_to_rsm`	`fname [, nx=30, ny=30, dims=(0, 1), reference_point=None]`	Writes an `.rsm` file with a 2D sample of `nx`,`ny` points in dimensions given by `dims`. The value of the remaining fixed dimensions can be specified by setting `reference_point` as a point that the 2D surface goes through.

Polynomial Response Surface Models

Polynomial RSMs fit an *N*-dimensional polynomial to each of the provided targets. Options to create polynomial RSMs are:

- `order`: Order of the polynomial to be fitted to the data

Gaussian Process Response Surface Models

Gaussian process RSMs fit an *N*-dimensional Gaussian process to each of the provided targets. Options to create Gaussian process RSMs are:

- `kernel [Matern32]`: Selects a kernel to use in the construction of the RSM (see Kernels on page 69)
- `hyperparam_mode ["ml-ii"]`: Selects the mode of setting hyperparameters for the kernel, which can be one of the following:
  - `"fixed"` retains the initial value of the hyperparameters throughout.
  - `"ml-ii"` selects the values of the hyperparameters that maximize the likelihood function, after each iteration.

Gaussian process RSMs have additional uncertainty information that can be returned using the `return_std` or `return_cov` optional arguments when evaluating the RSM.

Examples

Build an RSM for targets `t1` and `t2` from a 3×3×3 grid sampling of the input space:

```
rsm = optimizer.get_RSM([p1, p2, p3], [t1, t2], doe="grid",
                        rsm="gaussian_process",
                        doe_options={"Ns": (3, 3, 3)})
```

Write a CSV file with a 1D cutline along dimension 1 going through the point (0.5, 0.5, 0.5):

```
rsm.write_to_csv("cut.csv", dim=1, reference_point=(0.5, 0.5, 0.5))
```

Calculate a Monte Carlo estimation of the integral of the targets `t1` and `t2`:

```
xs = np.random.uniform(0, 1, size=(1000, 3))
res_t1, res_t2 = rsm(xs)
integ_t1 = np.sum(res_t1)/len(res_t1)
integ_t2 = np.sum(res_t2)/len(res_t2)
```

Optimization Parameters

Optimization parameters are Python objects of the classes `OptParam` or `TOptParam`. The signature of the construction is:

```
p = OptParam(name, value [, minval, maxval, scale, precision_rel,
                         precision_abs])

p = TOptParam(name, value [, minval, maxval, transform, precision_rel,
                          precision_abs])
```

where:

- `name` is the name of the parameter as defined in the Sentaurus Workbench project (case sensitive).
- `value` is the value of the parameter.
- `minval [None]` is the minimum value the parameter can accept.
- `maxval [None]` is the maximum value the parameter can accept.
- `scale [1.0]` is the scaling factor applied to the value.
- `transform [None]` is the transform applied to the value.
- `precision_rel [None]` is the relative precision given as a number of significant digits.
- `precision_abs [None]` is the absolute precision given as a number of digits after the decimal point (or before the decimal point if it is negative).

See Defining Optimization Parameters on page 12.

The `OptParam` class has the following attributes:

- `name`: Name of the parameter as defined in the Sentaurus Workbench project (case sensitive)
- `value`: Current value of the parameter
- `min`: Minimum value the parameter can accept
- `max`: Maximum value the parameter can accept
- `bounds`: Tuple with (`min`, `max`)
- `scale`: If `OptParam`, scaling factor applied to the value
- `scaled_value`: Current value of the parameter after applying scaling or transform
- `scaled_min`: Minimum value the parameter can accept after applying scaling or transform
- `scaled_max`: Maximum value the parameter can accept after applying scaling or transform
- `scaled_bounds`: Tuple with (`scaled_min`, `scaled_max`)
- `transform`: If `TOptParam`, transform function applied to the value

Discrete Optimization Parameters

Discrete optimization parameters can take only a limited number of values and have the following syntax:

```
p = DOptParam(name, value, choices, sorted=False)
```

where:

- `name` is the name of the parameter as defined in the Sentaurus Workbench project (case sensitive).
- `value` is the current value of the parameter.
- `choices` is a list of the possible values the parameter can take.
- `sorted` specifies whether to consider the listed values as ordered.

See Defining Discrete Optimization Parameters on page 13.

Note:

Not all optimization methods are appropriate for discrete parameters. In particular, gradient-based methods do not work with discrete parameters. Therefore, the only recommended way to perform an optimization with discrete parameters is to use the `optimizer.search` method (see optimizer.search on page 56).

Parameter Transforms

In an optimization, the range of parameters and the dependence of the targets on them can be complicated. When you have enough information about this behavior, using parameter transforms can simplify the dependence of the target on the parameters and, therefore, simplify the optimization.

For example, Figure 11 (*left*) shows the dependence of the following function on parameter r:

$$\text{target} = \left(\frac{0.1}{r}\right)^6 - e^{-r/0.2} \tag{10}$$

The basin of the minimum is very narrow, so unless the initial value is close to the final solution, it will be difficult to find it. However, after applying a logarithmic transform to the parameter, the basin of the minimum widens for the same bounds, as shown in Figure 11 (*right*), which makes it easier for the optimizer to find the minimum.

Figure 11 Dependence of the example function on (left) parameter r and (right) its logarithm

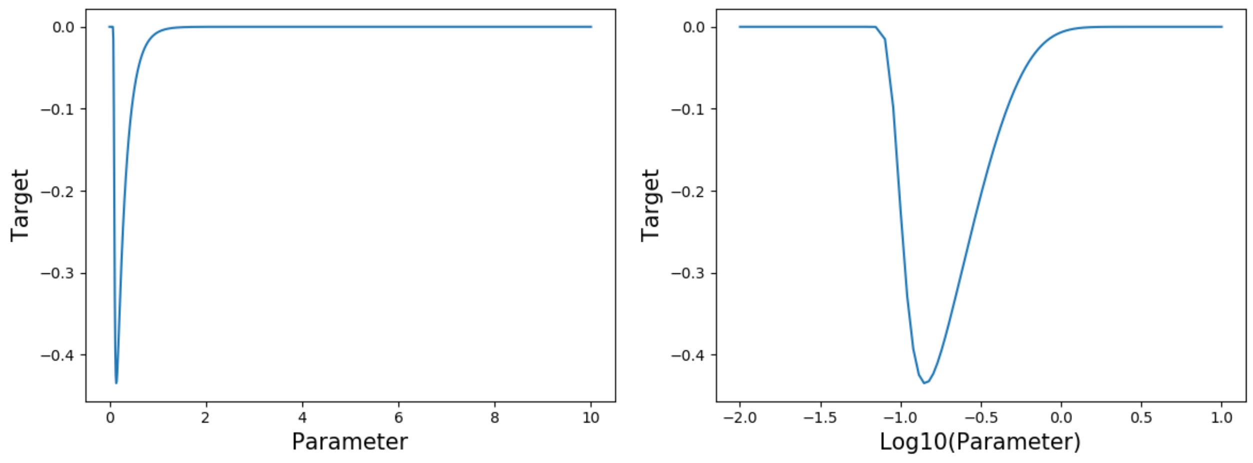


Table 10 Available parameter transforms

Transform	Arguments	Description
`Linear`	`scale`	Applies a linear scaling, $x_t = \text{scale} \cdot x$
`Log`		Applies a natural logarithm to the parameter, $x_t = \log(x)$
`Log10`		Applies a base 10 logarithm to the parameter, $x_t = \log_{10}(x)$
`Normalize`	`lower, upper`	Normalizes values from the range (`lower`, `upper`) to the range (–1, 1)
`Reciprocal`		Applies the reciprocal operator, $x_t = 1/x$

For example, to use an optimization parameter with a decimal logarithmic transform, specify:

```
p=TOptParam(name, 1, 1e-2, 1, Log10)
```

Parameter transforms can be combined by passing a list or tuple of transforms, and they are applied from right to left. For example, to apply a decimal logarithmic transform to a parameter and then normalize its range to (–1, 1), you can use the following syntax:

```
p=TOptParam(name, 1, 1e-2, 1, (Normalize, Log10))
```

Alternatively, parameter transforms can be combined using the * operator, and they are also applied from right to left. In this case, the transform must be instantiated with the parameters defined in Table 10. For example, to define the previous transform that calculates the logarithm of a parameter and then to normalize it, you could specify:

```
mytransform = Normalize(-2, 0)*Log10()
p=TOptParam(name, 1, 1e-2, 1, mytransform)
```

When using this syntax, the arguments for `Normalize` do not need to be the lower and upper bounds of the parameter. For example, if instead of wanting the transformed space of the parameter to be (–1,1) you want it to be (0, 1), you could specify the `lower` argument of `Normalize` to be `-4`:

```
mytransform = Normalize(-4, 0)*Log10()
p=TOptParam(name, 1, 1e-2, 1, mytransform)
```

Optimization Targets

Optimization targets are defined using the `OptTarget` class. The signature of the constructor is:

```
OptTarget(simulation_response, value=0.0,
          conditions={}, error_method=None, weight=1.0,
          scale=1.0, penalty=None, transform=None,
          reference_variables=(), simulation_variables=(),
          reference_options={}, simulation_options={},
          x_range=None, y_range=None,
          x_scale=None, y_scale=None,
          interpolation_options={"log": False, "method": 3},
          reference_range={}, simulation_range={},
          reference_scale={}, simulation_scale={})
```

where:

- `simulation_response`: Name of the Sentaurus Workbench parameter or variable in which the simulation response will be available (either as a value or as a file name).
- `value [0.0]`: Value to be matched by the simulation response. It can be either a value or a file name, or the name of a Sentaurus Workbench parameter or variable containing the value or file name.
- `conditions [{}]`: Additional state parameters that must be set to calculate the optimization targets.
- `error_method [None]`: If the simulation response is in a file, then this is the method used to calculate the pointwise error between the simulation response and the target value (that is, the residual). It can be `"abs"`, `"log_rel"`, `"rel"`, or `"square"`.
- `weight [1.0]`: Weight to be applied to the optimization target.
- `scale [1.0]`: Scaling factor to be applied to the target.
- `penalty [None]`: Penalty to apply to the target in case of violation of the constraints.
- `transform [None]`: Optional transform function applied to the target.
- `reference_variables [()]`: If the target value is in a file, then this is a tuple with the name of the independent and dependent variables (in that order).
- `simulation_variables [()]`: If the simulation response is in a file, then this is a tuple with the name of the independent and dependent variables (in that order).
- `reference_options [{}]`: If the target value is in a file, then these are additional options needed to read the values.

- `simulation_options [{}]`: If the simulation response is in a file, then these are additional options needed to read the values (for example, for a `.csv` file with comments starting with a hash character (#), `simulation_options={comment: '#'}`).
- `x_range [None]`: If the target value is in a file, then this is the range of interest for the independent variable.
- `y_range [None]`: If the target value is in a file, then this is the range of interest for the dependent variable.
- `x_scale [None]`: If the target value is in a file, then this is the scaling factor for the independent variable.
- `y_scale [None]`: If the target value is in a file, then this is the scaling factor for the dependent variable.
- `interpolation_options [{"log": False, "method": 3}]`: Interpolation options for the simulation data. If the simulation data and reference values are in a file but evaluated at different points, then the simulation data is interpolated on to the locations of the target values using the given options: `"log"` determines whether to use logarithmic interpolation, and `"method"` determines the order of the interpolating spline. You also can use `"clip"` and `"extra"` to limit unwanted behavior in the interpolation. For details, see Interpolating Targets on page 19.
- `reference_range [{}]`: If the target value is in a file, then you can use this dictionary to restrict the data to the given range. The keywords are `"x"`, `"y"`, or both, and their values are a tuple of two elements to denote their minimum-allowed and maximum-allowed values. The effect is the same as specifying `x_range` and `y_range`.
- `simulation_range [{}]`: If the simulation data is in a file, then you can use this dictionary to restrict the data to the given range. The keywords are `"x"`, `"y"`, or both, and their values are a tuple of two elements to denote their minimum-allowed and maximum-allowed values.
- `reference_scale [{}]`: If the target value is in a file, then you can use this dictionary to scale the data. The keywords are `"x"`, `"y"`, or both, and their values define by how much to scale the x- and y-values, respectively. The effect is the same as specifying `x_scale` and `y_scale`.
- `simulation_scale [{}]`: If the simulation data is in a file, then you can use this dictionary to scale the data. The keywords are `"x"`, `"y"`, or both, and their values define by how much to scale the x- and y-values, respectively.

Note:

If the simulation response is in a file, the target must also be given in a file. The contents of both files must have overlapping independent ranges (after possible target scaling).

You can access some of the options set at construction time, in instances of the `OptTarget` class, through the following attributes:

- `name`: Access the target name set with `simulation_response`
- `value`: Access the target value set with `"value"`
- `error_method`: Access the error calculation method set with `error_method`
- `penalty`: Access the penalty objects set with `penalty`
- `scale`: Access the target scale factor set with `scale`
- `transform`: Access the target transform set with `transform`
- `weight`: Access the target weight set with `weight`
- `x_range`: Access the independent variable range set with `x_range`
- `x_scale`: Access the independent variable scale set with `x_scale`
- `y_range`: Access the dependent variable range set with `y_range`
- `y_scale`: Access the dependent variable scale set with `y_scale`

Constraints

This section discusses how to apply constraints.

Applying Nonlinear Constraints

Nonlinear constraints are defined in a similar way to optimization targets, but their value is used only to determine whether the constraint is satisfied. You can use the `OptConstraint` class to define constraints, and the signature for its constructor is:

```
OptConstraint(simulation_response, **conditions)
```

where:

- `simulation_response`: Name of the Sentaurus Workbench parameter or variable in which the simulation response will be available as a value
- `conditions [{"gt": 0}]`: Conditions defining the constraints on the simulation response, which can be given as `"ge"`, `"gt"`, `"le"`, or `"lt"`

Constraints are then added to the call to `optimizer.optimize` or `optimizer.search`, together with the optimization targets.

Note:

Not all solvers support nonlinear constraints. In particular, only the `minimize` backend with the `"COBYLA"` or `"SLSQP"` method can be used.

You can access the name of the simulation response and the constraints applied to it from the instance of an `OptConstraint` object by using the following attributes:

- `name`: Access the constraint name that was set with `simulation_response`
- `inequalities`: Access the list of constraints applied to the output `simulation_response`

Examples

Set a constraint for the Sentaurus Workbench variable `"vt"` to be greater than 0.5:

```
c1 = OptConstraint("vt", gt=0.5)
optimizer.optimize([p1, p2], [t1, c1])
```

Set a constraint for the Sentaurus Workbench variable `"vt"` to be greater than 0.5 and less than 1.0:

```
c1 = OptConstraint("vt", gt=0.5)
c2 = OptConstraint("vt", lt=1.0)

optimizer.optimize([p1, p2], [t1, c1, c2])
```

Alternatively:

```
c1 = OptConstraint("vt", gt=0.5, lt=1.0)
optimizer.optimize([p1, p2], [t1, c1])
```

The advantage of this alternative approach that specifies both constraints for the same output is that the constraints will be checked for consistency at the time of constraint creation. For example, if you specify `lt=0.5, gt=1.0` accidentally, then an error will be raised because both conditions are mutually exclusive.

Applying Constraints Using a Penalty Function

Since hard constraints, as defined in Applying Nonlinear Constraints, can be used only with two solvers, you can apply constraints through a penalty function. In this case, when the constraint is violated, a penalty is added to the target, directing the optimizer away from the value. Penalties are defined using the `TargetPenalty` object. The signature of the constructor is:

```
p = TargetPenalty(conditions, penalty [, ignore_value])
```

where:

- `conditions`: A list of conditions defining the constraint for the target
- `penalty`: A function that calculates a penalty, given the difference between the target and the threshold of the constraint
- `ignore_value [False]`: Specifies whether to ignore the actual value of the target, keeping only the penalty

This can be passed to any optimization target by using the keyword `constraint`. For example:

```
def quad_penalty(delta):
    return delta**2
p = TargetPenalty([("lt", 1.0), ("gt", 0.5)], quad_penalty)
target2 = OptTarget("vt", penalty=p)
```

Here, the target range of the variable `vt` is between 0.5 and 1.0. If it is not satisfied, then a penalty is applied to the optimization target in the next optimization step.

The condition given in `TargetPenalty` refers to the same target given in the `OptTarget` in which it is used. For example, consider a case where you want to optimize two values of a threshold voltage, which are output in the Sentaurus Workbench variables `vt_low` and `vt_high`:

```
def quad_penalty(delta):
   return delta**2
p = TargetPenalty([("lt", 0.2), ("gt", -0.1)], quad_penalty)
target1 = OptTarget("vt_low", target=0.25, penalty=p)
target2 = OptTarget("vt_high", target=0.3, penalty=p)
```

This penalty definition will penalize simulation output values if they are greater than 0.2 or less than –0.1 with *respect to their target value*. In other words, if the value of `vt_low` is above 0.45 (0.25 + 0.2) or below 0.15 (0.25 – 0.1), then the penalty will be applied to it. Similarly, if the value of `vt_high` is above 0.5 (0.3 + 0.2) or below 0.2 (0.3 – 0.1), then the penalty will be applied to it.

References

[1] D. Eriksson *et al*., “Scalable Global Optimization via Local Bayesian Optimization” in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, Vancouver, BC, Canada, pp. 5473–5484, December 2019.

Index