PScript Manual

Acknowledgments:

Dr.-Ing. Matthias Westhäuser, Dipl.-Inf. Nicolas Luck, B.Sc. Jan-Hendrik Menke, Dr.-Ing. Christian Remmersmann

Manual Creation Date:

August 15, 2013; PHOTOSS 5.92

Contents

1 Copyright Information

2 Acknowledgments

The author would like to thank Dipl.-Inf. Nicolas Luck, B.Sc. Jan-Hendrik Menke and Dr.-Ing. Christian Remmersmann for general comments and assistance with quality assurance concerning this document. Please open the PScript console and look at the menu entry *Help* ⇒ *About PScript* to get an overview of all the people who took part in developing PScript.

3 What is PScript?

PScript is an abbreviation for PHOTOSS Script. It is a script engine based on the ECMA-Standard. PScript can be used to embed a PHOTOSS simulation in an algorithmic context. Thus, it enables the user to control the PHOTOSS environment, using predefined, automated scripts which reside "outside" of the PHOTOSS simulation. PScript can help to drastically reduce the amount of "manual" work needed to create flexible, straightforward, and yet complex parameter variations.

New Possibilities arise by using PScript - a Short Overview:

- Generate automated scripts that require little or no user interaction to run.
- Use PScript to easily create dynamic parameter variations which are much more flexible than standard PHOTOSS parameter variations.
- Embed your simulations into an algorithmic context (for-loops, if-statements, etc.): Change component parameters depending on results of earlier simulation runs and parameter combinations, or extend and abort parameter combinations once certain conditions are met.
- Directly access all PHOTOSS results (without the use of file-writing mechanisms) and utilize these results to make decisions in your algorithmic context and parameter variation(s) or start a detailed analysis of your results by transferring them to a MATLAB® workspace environment.
- Use the algorithmic context to "propagate" knowledge of simulation parameters and results from one simulation run to another. This enables the use of powerful statistical methods such as importance sampling or Multicanonical Monte Carlo methods (MMC).

The remainder of this document is organized as follows:

- Section **4 - Version Changes** on page 7 is a list of all new features which have been added since the first release version of PScript. Browsing through this page will immediately make you familiar will all new functions and features.
- Section **5 - Basic Functionality of PScript** on page 10 gives a short overview of what the PScript engine is capable of.
- Section **6 - PScript Conventions** on page 12 will familiarize you with all important conventions and writing styles when making your own PScripts.
- Section **7 - PScript Demo Walkthrough** on page 15 will show you how easily you can make your own PScript s using a `"Hello World"`-example.
- Section **8 - Access to Simulations, Components, Parameters and Results (Quickstart Guide)** on page 16 will provide you with all the skills necessary to create a more complex PScript and you will learn how to handle simulations, components, parameters and results.
- Section **9 - PScript and MATLAB®** on page 19 will illustrate the powerful interaction between PScript, PHOTOSS and MATLAB®.
- Section **10 - PScript Console Main Window** on page 20 will familiarize you with the PScript main window and its functionality.
- Section **11 - PScript Console Menu Functionality** on page 21 gives a detailed overview of the PScript menu.
- Section **12 - PScript Settings Dialog** on page 22 describes in detail all available PScript Settings.
- Section **14 - PScript Syntax Highlighting and Code Assistant** on page 29 will introduce you to the comfortable Code Assistant and PScript Syntax Highlighting abilities.
- Section **15 - PScript Code Samples** on page 30 includes an overview of all available PScript Code Samples which are included in your PHOTOSS installation.
- Section **16 - PScript Function Reference** on page 31 includes detailed descriptions for all functions of PScript.

- Section **17 - Basic JavaScript Mathematical Function Reference** on page 60 is a short introduction to the basic mathematical functions which are automatically included in PScript and JavaScript.
- Section **18 - External References** on page 61 lists a few hyperlinks to Internet sources for further information.

4 Version Changes

In this section you will find all important changes which were made between new versions of PHOTOSS and PScript. If you are unsure which version of PHOTOSS and PScript you use, please open PScript and look at the menu entry *Help* ⇒ *About PScript*. There you will find the revision number of both PHOTOSS and PScript as well as the build date and several other information.

4.1 From PHOTOSS 5.0 (Rev. 3624) to 5.10 (Rev. 3929)

No changes at PScript.

4.2 From PHOTOSS 5.10 (Rev. 3929) to 5.10.3 (Rev. 5019257)

No changes at PScript.

4.3 From PHOTOSS 5.10.3 (Rev. 5019257) to 5.90.2 (Rev. fd68ed5)

This revision contains several major additions to the PScript functionality, including the creation, linking and deletion of components as well as the ability to use the new concept of simulation parameters.

- **Semantic change:** When invoking `s.run()` to execute a simulation, *parameter variations will no longer be run*. The simulation will only be run once, using the *current value* for each simulation parameter (and its placeholders). This is due to the fact that a complete parameter variation script can be automatically calculated and run in the PHOTOSS GUI (please refer to the PHOTOSS manual for more details). If you want to run a parameter variation directly from PScript, you may use the new function `getParameterVariationScript()` described on page 52.
- **Semantic change:** For addressing the simulation parameter "polPlanes", the valid values were changed from "`false`" or "`true`" to the *number of simulated polarization planes*. The new valid values may be "`1`" (one polarization axis, no polarization effects) or "`2`" (two polarization axes, polarization effects are included). Please adjust your scripts accordingly! A warning will automatically be generated each time you try to change the parameter "polPlanes" to indicate the semantic changes.
- **Semantic change:** PScript now has a *separate* random number generator which is no longer connected to a PHOTOSS simulation. It is thus possible to generate random numbers without having to open PHOTOSS simulations first. Please also refer to section 16.3 on page 34 for more details on random number generation. Changing the random generator settings for a simulation is, of course, also possible by adjusting the corresponding simulation parameters.
- New feature: PScript is now using an improved algorithm for syntax highlighting.
- New feature: PHOTOSS has undergone major changes concerning its simulation parameter and "global parameter" structures (please refer to the PHOTOSS manual for a more detailed explanation). Most importantly, the old concept of "global variables" has been replaced by a global "simulation parameter" concept which concerns both the built-in simulation parameters (e.g. *reference bitrate*, etc.) and custom simulation parameters (which were called "global parameters" before). Both custom and built-in simulation parameters may be used as place-holders for (e.g.) component parameters and may be used in a parameter variation. From a PScript point of view, the syntax for the simulation parameter handling has been extended and you can now distinguish between built-in and custom simulation parameters. **Additionally, creating or deleting (custom) simulation parameters is now possible.** Please refer to section 16.7 on page 48 for further details and an overview of all new functions!
- New feature: Due to the major changes in the simulation parameter class, a new feature has become available. You can now also access the so-called "derived" simulation parameters by using the new functions `listDerivedSimulationParameters`, `getDerivedSimulationParameterValue` and `getDerivedSimulationParameterUnit` described in section 16.7 on page 48. The derived simulation parameters are visible in the simulation parameter dialog and are automatically calculated from the standard set of built-in simulation parameters. Thus, these parameters are read-only; however, they can be used as placeholders in formulas, e.g. for component parameters.

- Continued support: Due to the changes to the simulation parameter concept explained above, the old concept of "global variables" has been abandoned. If, however, you still have legacy code using functions which operate on "global variables" this code still works and automatically calls the new simulation parameter counterparts. A warning will be issued to indicate this. For more details on which functions are now obsolete, please refer to section 16.9 on page 55.

- New feature: PScript is able to invoke command line calls. See the function `execute()` on page 32 for further details.

- New feature: New simulations can now be created using PScript by invoking the `newSimulation()` command described on page 38.

- New feature: You can check, whether any component is of a certain type (e.g. a "Pulse Generator" or an "Analytical Filter") by using the `isComponentOfType()` function described on page 46.

- New feature: You can access the type of any PHOTOSS component using the `getComponentType()` function described on page 44.

- New feature: You can now change the name of any component using the `setName()` function described on page 47.

- New feature: It is now possible to create components using the `createComponent()` functionality listed on page 43. Components can also be deleted by using the PScript function `deleteComponent()` and `deleteAllComponents()` which are described in detail on page 41. It is also possible to create a new component and automatically link it to a component which already exists using the PScript function `createAndConnectComponent()` (see page 44). The coordinates of a component can be accessed using the `getGridCoordinates()` function shown on page 44.

- New feature: A lot of new functions have been added to enable you to control component linking. Please refer to the new PScript code sample "Component Linking" to get a detailed example and description of all new features concerning component linking. The new functions are:
  You can now link components manually using the PScript function `linkComponents()`. It is described in detail on page 56. You can also remove the link on specified components by using the `unlinkComponents()` function of PScript which is described on page 57. Component ports are addressed zero-based (e.g. starting from 0 for the first port of a component). You can check whether a given In- or Outport of a component is linked by invoking the `isInPortLinked()`, `isOutPortLinked()` and `isLinked()` functionality which are described on page 56. You can check whether the component you specified is linked to another component by calling the `isLinkedTo()` function (see page 57). It is also possible to receive the port number(s) through which a given set of components is connected by calling the `getLinkPorts()` function described on page 57. The functions `getInPortLinks` (page 56) and `getOutPortLinks` (page 56) work the other way round: They return the component(s) which are connect to the given In- or Out-Ports of a given component.

- Objects and variables in the PScript workspace can now be deleted ("cleared") using the function `clear()`. You can find more details about the function on page 40.

- New feature: PHOTOSS compatible GPU devices and PHOTOSS GPU options can now be controlled in PScript by using the function `setPHOTOSSOptionValue()` described on 39.

- New feature: It is now possible to access components and component results by using the "asterisk-method". For example you can access all components which possess a name that starts with the phrase "Analytical" by using `myComponentVector = mySimObject.getComponents("Analytical*")`. If, for example, you want to access all result values which end with the phrase "penalty", you can use `myResultsValues = s.myComponent.getResultValues("*penalty")`. It is also possible to access all components (or results) which *contain* a given phrase. E.g. `myComponentVector = s.getComponents("*filter*")` would return all components which contain the phrase "filter" in their name.

- New feature: PScript files can now be dropped on the PScript code area using the "Drag-And-Drop" method. Just drag you PScript file (from e.g. the Explorer) to the PScript code area and drop it there.

- New feature: The PScript Manual is now accessible through the PScript Console window by selecting *Help* ⇒ *PScript Manual*.

- New default value: The default value for the MATLAB® timeout has now been set to 10.000 seconds instead of 600.
- New default value: The default value for the Syntax Highlighting in PScript has now been set to true.
- Bugfixes: Issues with the PScript Syntax Highlighting (SHL) have been solved. Full undo functionality has been implemented; latency for displaying and editing larger PScript files has been greatly reduced.

4.4 From PHOTOSS 5.90.2 (Rev. fd68ed5) to 5.91

- New feature: You can get an overview of all the currently existing variables and objects in the PScript workspace by calling the `who()` functionality which is shown in more detail on page 33.
- New feature: Directories can now be created by using the `createDirectory()` function described on page 37.
- New feature: Directories can now be deleted by using the `deleteDirectory()` function described on page 37.
- New feature: The existence of a file can be checked by using the `doesFileExist()` function described on page 37.
- New feature: All files of a specified folder can be copied to a destination folder by using the `copyFromTo()` function described on page 37.
- New feature: `appendStringToFile()` function described on page 36.
- New feature: `readStringArrayFromFile()` function described on page 38.
- New feature: `writeStringArrayToFile()` function described on page 39.
- Bugfix: Calling the `clearWorkspace()` function no longer deletes the `execute()` functionality.

4.5 From PHOTOSS 5.91 to 5.92

- New feature: You can get a timestamp string in PScript by calling the `timestamp()` function described on page 39.
- New feature: You can save and load your current PScript workspace data similar to MATLAB® by using the functions `saveVariables()` (page 38) and `loadVariables()` (page 38).
- Misc: The addressable grid size of a network in PScript is now 1024x1024 instead of 128x128.

5 Basic Functionality of PScript

Please also refer to the section **15 - PScript Code Samples** on page **30** and **16 - PScript Function Reference** on page **31** for more details and explanations.

- PScript can be used to create, open, save and close any PHOTOSS simulation file.
- PScript can start, clear, end and abort a PHOTOSS simulation.
- You can access and modify any simulation parameter in an opened simulation file using PScript.
- You can create and delete custom simulation parameters using PScript.
- You can access and modify any component parameter in an opened simulation file using PScript.
- You can globally change parameters for all components of the same type in an opened PHOTOSS simulation file (for example you can set all analytical filter types to the type "Gauss"). You can also add conditions to such a statement (e.g. you can set all analytical filter types which work in the mode "optical" to the type "Gauss").
- After a simulation has finished, you can access any result by using PScript. The result does not have to be saved to a file or specifically activated in order to do this.
- You can access parameters and results of components which reside inside a network structure, sub-network structure, or an iterator component.
- Any formula used to calculate a simulation parameter / component parameter can be accessed and modified.
- You can access the PHOTOSS random generator and place a custom seed using PScript. You can also switch between the modes "deterministic" and "statistical".
- You can access the MATLAB® workspace at any time (except during a simulation run). PScript can copy double variables either from or to the MATLAB® workspace. You can also "send" *any* valid MATLAB® command to the MATLAB® workspace which will automatically be executed (creation of variables, execution of a MATLAB® script, etc.). Please refer to the chapter **9 - PScript and MATLAB®** on page 19 for further details.
- PScripts can be divided into "subscripts" by using the `include`-command.
- You can create and integrate your own function library for PScript.
- You can automatically execute any number of initialization scripts before starting your main PScript using the PScript "Auto Include" functionality.
- The active PScript directory can be accessed and changed to any valid path. Relative paths can be defined.
- You can use all data types and data structures currently supported by Qt Script / JavaScript - `double`, `int`, arrays, vectors, etc. Please see the reference [JavaScript] on page 61 for details.
- You can use all typical algorithmic constructs and operators supported by Qt Script - this of course includes loops (`for` / `while`), `if`-statements, etc. Please see the reference [JavaScript] on page 61 for details.
- You can use the (basic) mathematic functions included in Qt Script. This includes the use of (e.g.) `sin`/`cos`-functions, `exp`/`log`, etc. Please see section **17** on page 60 and the reference [MATH] on page 61 for details.
- You can use PScript to execute command line interface programs.
- PScript offers full access to all PHOTOSS console logs; you can predefine which logs will be shown on the PScript output console (errors, warnings, general output).
- You can `print` any valid `string` to both (or either) the PHOTOSS and the PScript console.

- PScript offers a script editor with basic syntax highlighting and automatic code completion abilities. The editor allows you to create your own PScripts and to view / modify the predefined example scripts. Of course, you can alternatively use any custom text file editor to create / modify your own scripts; any valid script can be loaded and run from the PScript main window.
- You are able to set all PHOTOSS options using PScript, including GPU options.

6 PScript Conventions

6.1 General Conventions

- All PScript program code is case sensitive.
- A PScript program code line does not have to be terminated with a semicolon.
- Comments can be inserted in code lines by beginning the line with two slashes: `//`.
- Longer comments can be inserted by using `/* Long Comment */`
- When specifying a path or directory string, you can either use singles slashes or two backslashes: `application.setCurrentDirectory("C:/Test/")` or `application.setCurrentDirectory("C:\\Test\\SubDir")`
- For more syntax conventions please also refer to the PScript Code Samples in section 15, page 30.
- PScript function calls containing more than one word are written in camel case notation, e.g.: `.getParameterValue()`.
- The only exceptions from this rule are the phrases “PScript” and “PHOTOSS” which are always written with a capitalized “PS” or completely capitalized, respectively. Any other phrase which is also completely capitalized will also not be written in camel case notation, e.g.: `simObject.SMF.getParameterValue()`.

6.2 Conventions concerning PScript Objects

An overview of the objects of PScript is shown in figure 1. Objects which are depicted in black are always available in PScript. Objects with a different color must be created or opened first.

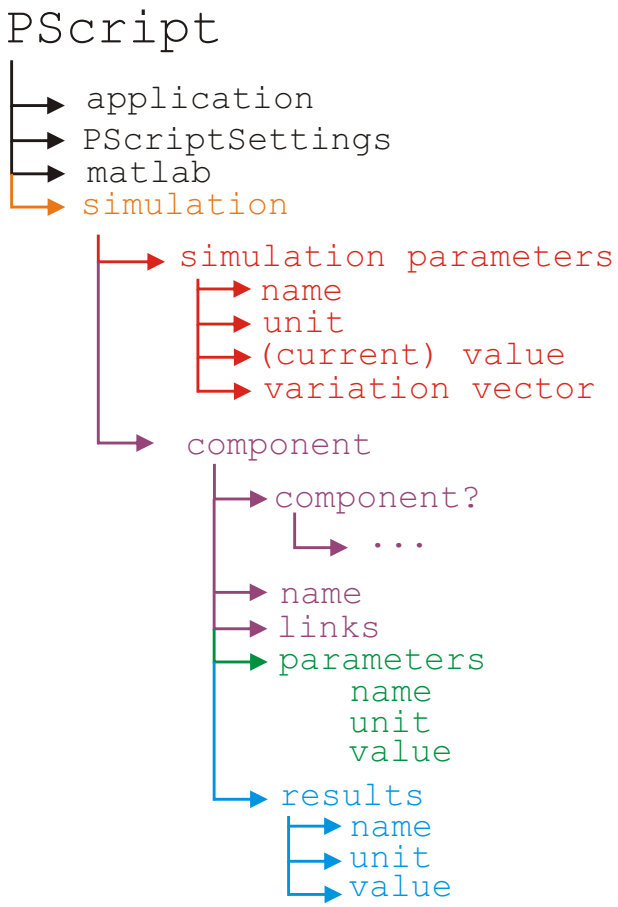

Figure 1. Overview of PScript objects. Objects shown in black are always available in PScript.

- Most PScript objects can be overwritten and copied (exceptions: the `application`-object and the `PScriptSettings`-object).
- Multiple PScript objects of the same type may coexist in PScript (exceptions: the `application`-object and the `PScriptSettings`-object).
- The basic object, you have access to once it is created or opened, is the simulation object:
  - A simulation object may hold several component objects (or none, if the simulation is empty).
  - A simulation object always holds the simulation parameters (both custom and built-in).
  - A simulation object can be assigned to one or more variables.
  - A simulation object can be created by opening a simulation or creating a new simulation.
  - A simulation object can be invalidated by closing a simulation.
  - A simulation object can be saved as a *.pho-file.
  - Invalidating a simulation object also invalidates all the component objects, simulation parameters, etc. inside the simulation.
  - Invalidating a simulation object does *not* invalidate copies of component objects, simulation parameters, etc. which have been assigned to a variable.
- The second object is the component object:

- A component object represents any valid PHOTOSS component (e.g. the Pulse Generator, a Numerical BERT or even a Network or Iterator component).
- A component object is a child object of a simulation object.
- A component object can be assigned to one or more variables.
- A component object may include parameters and results (Access to parameters and results of a component object is described in chapter **Access to Simulations, Components, Parameters and Results (Quickstart Guide)** on page 16).
- A parental component object may also include child components if the parent is a network or an iterator. Of course the child component may itself contain further child components; e.g. placing an iterator inside a network is allowed.
- A component object can only be accessed through the simulation object until it is copied to a variable. Thus, invalidating a simulation object automatically invalidates a component object, unless you have copied it to a variable before.

7 PScript Demo Walkthrough

In this chapter we will illustrate how easy it is to create and run a PScript. It is strongly recommended you take a closer look at the **Access to Simulations, Components, Parameters and Results (Quickstart Guide)** on page 16, the **PScript Code Samples** described in chapter 15 on page 30 and the **PScript Function Reference** on page 31.

- Open a PHOTOSS instance.
- Click on the Menu "Tools" ⇒ "PScript Console" to open the PScript console or use the "PS"-Icon in the icon bar.
- Type the following code into the Code Window: `print("Hello PScript User.")`
- Save your PScript by using "File" ⇒ "Save as" and assign the name "Hello PScript User.pscript" to it. Store your PScript in a directory in which you have write permission.
- Click on "Run" ⇒ "Run" to start your PScript (or press "F").
- The text "Hello PScript User!" should be returned in the Console Window. If the text does not appear, check your **PScript Settings Dialog** (page 22), choose "Restore Defaults" and rerun the script.
- Congratulations, you have just created and executed your first PScript!
- For more complex scripts, please refer to the **Access to Simulations, Components, Parameters and Results (Quickstart Guide)** on page 16.

A typical PScript of a more complex simulation might include some of the following, more general steps:

- Initialize PScript parameters / variables
- Create arrays to store the results of your parameter variation
- Open your PHOTOSS simulation
- Setup a main `for`-loop which iterates over all parameters in your parameter variation
- Adjust component / simulation parameters according to your parameter variation
- Run the PHOTOSS simulation
- Obtain your results / observables
- Clear the simulation
- Define an abort criterion for the parameter variation
- Continue the main for loop until the parameter variation is complete
- Close the simulation
- Examine your result space

8 Access to Simulations, Components, Parameters and Results (Quickstart Guide)

Aside from the PScript settings and the application object (see chapters below), the most important feature of PScript is the access to the components of a simulation and their parameters and results. In this chapter, we will learn that just a few function calls will enable us to access the parameters or results of any component. Please also refer to the **PScript Code Samples** on page 30 and the **PScript Function Reference** on page 31 for more detailed descriptions.

Changing the working directory of PScript:

Before we can open a simulation, we need to switch to the directory in which it resides. We can do this by either switching to an absolute or relative path:

```
application.setCurrentDirectory("C:/PScripts/")//Switch to absolute path.
```

Opening a simulation:

We can open a simulation and assign the simulation object to a variable by using:

```
1 simulationObject=application.openSimulation("My Simulation Name.pho")
```

Making a backup of a simulation:

```
1 simulationObject.saveAs(application.getUserHomeDirectory()+"/mybackup.pho") //store
      backup in the user's  home directory \bb
```

Accessing a Component Object:

Before we can actually access a component object, it might be useful to get an overview off all components in the simulation objects:

```
1  simulationObject.listComponents() // print a list of all components to the console
```

Now, accessing a component object through a simulation object can be done by using the "." operator. In many cases, it us useful to assign the component object to a variable. You can both use the camel case notation and the same notation as in the PHOTOSS GUI as long as the name of the component does not contain spaces. If it does, you can use the camel case notation only. In short, the camel case notation is always valid. So, if for example the name of the desired component is "AnalyticalFilter1" we could write:

```
1 componentObject = simulationObject.AnalyticalFilter1; // access a component object using
      the GUI notation, OR - equivalently:
2 componentObject = simulationObject.analyticalFilter1; // access a component object using
      the camel case notation
```

If, instead, the name would be "Analytical Filter 1" you would have to use:

```
1 componentObject = simulationObject.analyticalFilter1; // access a component object using
      the camel case notation
```

There is also an equivalent which is useful when component objects shall be directly addressed using their string name *regardless* whether it contains spaces or not:

```
myFilterObject = simulationObject.getComponent("Analytical Filter 1") // access the component object  using ‘‘natural’’ notation
```

Accessing parameters of a Component Object:

Most PHOTOSS components have more than one parameter which can be edited in the PHOTOSS GUI. To get an overview, which parameters are available for your desired component, type:

```
componentObject.listParameters() //print a list of all parameters of the current component to the console \index{listParameters()}
```

Accessing a single parameter in PScript works the following way:

```
parameterValue = componentObject.getParameterValue("Parameter Name")
```

It is also possible to directly access a parameter value. The same rules as for the component names apply here as well:

```
parameterValue = simulationObject.componentObject.parameterName // directly access the parameter value using (e.g.) camel case notation.
```

If we are interested in the unit of the parameter, we would use:

```
parameterUnit = componentObject.getParameterUnit("Parameter Name")
```

Setting parameters works in a similar manner:

```
componentObject.setParameterValue("Parameter Name", parameterValue)
```

Sometimes, you want to access more than one parameter at a time. In these cases, the following methods are helpful:

```
parameterNames = componentObject.getParameterNames() //read all parameter names into a string vector
parameterValues = componentObject.getParameterValues()  // read all parameter values into a string vector
parameterUnits = componentObject.getParameterUnits() //read all parameter units (if they exist) into a string vector
```

Accessing results of a Component Object:

Many PHOTOSS components have results which become available at the end of a simulation. So before we obtain the results, we have to start the simulation:

```
simulationObject.run()
```

First, an overview of which results are available for our desired component might be useful. Type:

```
componentObject.listResults()
// or alternatively:
simulationObject.componentObject.listResults()
```

Accessing a single result value is also easy:

```
resultValue = componentObject.getResultValue("Result Name")
//or alternatively:
resultValue = simulationObject.componentObject.getResultValue("Result Name")
```

Accessing a result unit (if available) works similarly:

```
resultUnit = componentObject.getResultUnit("Result Name")
// or alternatively:
resultUnit = simulationObject.componentObject.getResultUnit()
```

Sometimes, it is useful to address more than one result at a time. The following functions can be used to realize this:

```
resultValues = simulationObject.componentObject.getResultValues()  // returns a string
    vector holding all the result values of a component
resultNames = simulationObject.componentObject.getResultNames()  // returns a string
    vector holding all the result names of a component
resultUnits = simulationObject.componentObject.getResultUnits()   //returns a string
    vector holding all the result units (if they exist) of a component.
```

Accessing Simulation Parameters:

The simulation parameters can always be accessed through the simulation object as these two objects are closely connected. First, an overview over all simulation parameters variables might be useful:

```
simulationObject.listSimulationParameters()
```

Now, let us read a simulation parameter:

```
myParamName = simulationObject.getSimulationParameterValue("Parameter Name")
```

Setting a parameter works in a similar manner:

```
simulationObject.setSimulationParameterValue("Parameter Name", parameterValue)
```

Congratulations! You have already mastered all basic skills necessary to create PScripts. For more details please refer to the **PScript Code Samples** in chapter 15, page 30, or the **PScript Function Reference** in chapter 16, on page 31.

9 PScript and MATLAB®

This chapter offers a basic overview of PScript and MATLAB® interaction. Please refer to the PScript MATLAB® code example named `"1 Matlab Interface Interaction.pscript"` in chapter PScript Code Samples on page 30 for a detailed explanation of every topic listed below:

- PScript can automatically determine whether a version of MATLAB® which is currently supported by PHOTOSS is available.
- PScript can assign any (complex) double, float, int or bool value (or a corresponding matrix) to any existing MATLAB® variable using the command:
  ```
  matlab.setValue()
  ```
- PScript can currently read any MATLAB® variable which is either a double or a (complex) matrix into a PScript variable using the command:
  ```
  matlab.getValue()
  ```
- PScript can pass any string containing any valid MATLAB® code to the MATLAB® workspace which will automatically be evaluated afterwards by using the command:
  ```
  matlab.eval("a=10+11")
  ```
- PScript can define whether to check for or ignore errors that occur while executing MATLAB® code (this applies for any MATLAB® code run in the MATLAB® workspace). Please refer to the section PScript Settings Dialog on page 22 for more information.
- PScript can create either a common or a separate MATLAB® workspace for both the PScript MATLAB® environment and the PHOTOSS MATLAB® Component environment. The default is that a single, common workspace is used. This can be useful when passing information to or getting information from MATLAB® components in the simulation as desired by the user. Please refer to the section PScript Settings Dialog on page 22 for more information.

10 PScript Console Main Window

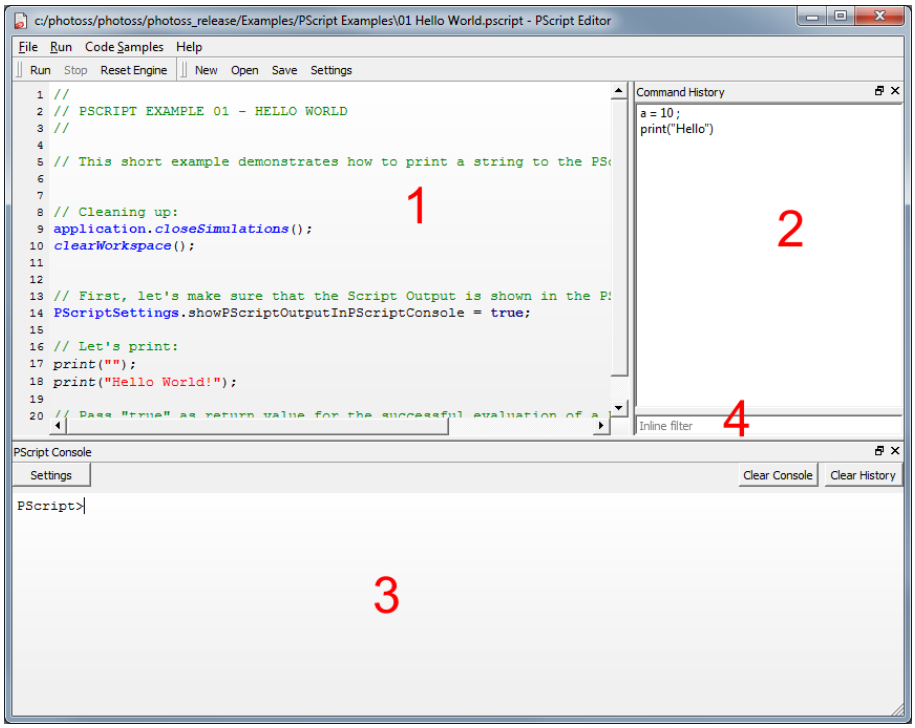

Figure 2. The PScript Console Main Window is devided into the Code Area (1), the Command History (2) and the Console Area (3). An inline filter (4) can be used to search through the commands in the command history.

The PScript Console Main Window consists of four main areas which can individually be configured to your liking:

1. The Code Area contains the program code of your currently active PScript; any loaded PScript will be displayed here. This is also the area where you can edit your PScripts.
2. The Command History Area contains previously (manually) entered PScript console commands for easy reuse. It can be cleared by using the PScript Settings menu (see chapter PScript Settings Dialog on page 22)
3. The PScript Console Area also shows the PScript output and (if you have enabled the option in the PScript Settings) the date and time information. The output may contain errors, warnings, and simple information. You can enter and execute any valid PScript or valid Qt Script command. You can also decide which parts of the PHOTOSS output you want to see in your PScript console (see *Code Example 04* for further details on this topic).
4. An inline filter can be used to search through all commands included in the command history. If, for example, you want to find all recently used commands containing the phrase “print” , you can simply type “print” in the inline filter.

11 PScript Console Menu Functionality

The following operations can be accessed via the PScript console menu:

- *File ⇒ New*
  Creates a new PScript code file. You will be reminded, if you have any active PScript file which has not yet been saved.
- *File ⇒ Open*
  Open any existing PScript code file. If you have already have opened an active PScript file, you will be asked to either discard or save the changes before a new file will be opened.
- *File ⇒Save*
  Save the active PScript code file using its current filename.
- *File ⇒ Save as*
  Save the active PScript code file and change / select the filename.
- *File ⇒ Close the PScript Console*
  Closes the console. You will be asked to save any changes on your active PScript file if you have not already done so.
- *Run ⇒ Run*
  Runs the active PScript file. If auto includes have been defined, they will be run first (see chapter PScript Settings Dialog on page 22 for more details.
- *Run ⇒ Clear PScript Workspace*
  The PScript workspace is cleared. All variables and objects will be removed. Clear PScript Workspace will also automatically be called when you close the PScript Main Window; it will not be called if you minimize the PScript Window to your task bar.
- *Run ⇒ Stop*
  Stops a running PScript file. Warning: Currently running MATLAB®calculations will *not* be terminated!
- *Run ⇒ Clear Console*
  Removes any output from the PScript console. *Clear Console* will not remove the variables and objects currently residing in the PScript workspace. Please use *Clear PScript Workspace* instead.
- *Code Samples ⇒ ...*
  Opens a PScript Example, containing a wide number of explanations and documented PScript features. See the Chapter PScript Code Samples for further information.
- *Settings ⇒ ...*
  Opens a dialog where the user can choose the PScript options. Please refer to the chapter PScript Settings Dialog on page 22 for more details on the specific options.
- *Help ⇒ About PScript*
  Opens a dialog window with copyright information and release dates of the current PScript version.

12 PScript Settings Dialog

The PScript Settings Dialog can be accessed through the GUI interface. All PScript settings can also be set manually by invoking the appropriate function.
Many functions include a set- and a get-method. The get-methods can be used to check the according PScript Setting.

set / get All
Code set: `PScriptSettings.setAll(PScriptSettingsObjekt)`
Code get: `mySettingsObject = PScriptSettings.getAll()`
Default: *NA*
Description: The get-method reads all PScript Settings into a PScript Settings object which can be copied. The set-method sets all PScript Settings to the values specified in the given PScript Settings Object.

Restore Defaults
Code: `PScriptSettings.restoreDefaults()`
Default: *NA*
Description: All PScript Settings will be reverted to their original defaults. Invoking the function has the same effect as clicking on the button "Restore Defaults" in the PScript Settings menu.

12.1 Console Settings

Show Time and Date at Prompt
Code set: `PScriptSettings.setShowTimeAndDateAtPrompt(bool setBool)`
Code get: `myBool = PScriptSettings.getShowTimeAndDateAtPrompt()`
Default: `true`
Description: If set to `true`, the complete date is shown at the PScript prompt. It includes the day, month, year and the time in format hours : minutes : seconds. When PScript finishes the execution of a script, both date and time will be displayed automatically.

Show PHOTOSS Output in PScript Console
Code set: `PScriptSettings.setShowPHOTOSSOutputInPScriptConsole(bool setBool)`
Code get: `myBool=PScriptSettings.getShowPHOTOSSOutputInPScriptConsole()`
Default: `false`
Description: If set to `true`, all general output of the PHOTOSS console will also be shown in the PScript console output. This does *not* include PHOTOSS warnings or errors.

Show PHOTOSS Errors in PScript Console
Code set: `PScriptSettings.setShowPHOTOSSErrorsInPScriptConsole(bool setBool)`
Code get: `myBool = PScriptSettings.getShowPHOTOSSErrorsInPScriptConsole()`
Default: `true`
Description: If set to `true` and an error occurs during a PHOTOSS simulation, the error will also be displayed in the PScriptconsole. Disabling this option is *not* recommended.

Show PHOTOSS Warnings in PScript Console
Code set: `PScriptSettings.setShowPHOTOSSWarningsInPScriptConsole(bool setBool)`
Code get: `myBool = PScriptSettings.getShowPHOTOSSWarningsInPScriptConsole()`
Default: `false`
Description: If set to `true` all warnings which occur during or prior to the execution of a PHOTOSS simulation will also be displayed in the PScript console.

Show Script Output in PScript Console
Code set: `PScriptSettings.setShowPScriptOutputInPScriptConsole(bool setBool)`
Code get: `myBool= PScriptSettings.getShowPScriptOutputInPScriptConsole()`
Default: `true`
Description: If set to `true`, information which is included in a print command in a PScript context (e.g. `print("Hello World")`) will be displayed in the PScript output console. Please keep in mind that this

includes error and warning messages which are issued by PScript - if a PScript error occurs during the execution of your PScript and the option is set to `false`, you will not receive a notification.
Disabling this option is *not* recommended.

Show Printed Output in PHOTOSS Console
Code set: `PScriptSettings.setShowPrintedOutputInPHOTOSSConsole(bool setBool)`
Code get: `PScriptSettings.getShowPrintedOutputInPHOTOSSConsole()`
Default: `false`
Description: If set to `true`, information which is included in a print command in a PScript context (e.g. `print("Hello World")`) will be displayed in the PHOTOSS output console. This does *not* include PScript warnings or errors.

Show PScript Errors in PHOTOSS Console
Code set: `PScriptSettings.setShowPScriptErrorsInPHOTOSSConsole(bool setBool)`
Code get: `myBool = PScriptSettings.getShowPScriptErrorsInPHOTOSSConsole()`
Default: `true`
Description: If set to `true`, PScript errors will appear in the PHOTOSS error-output console (and the standard output console). Disabling this option is *not* recommended.

Show PScript Warnings in PHOTOSS Console
Code set: `PScriptSettings.setShowPScriptWarningsInPHOTOSSConsole(bool setBool)`
Code get: `myBool = PScriptSettings.getShowPScriptWarningsInPHOTOSSConsole()`
Default: `false`
Description: If set to `true`, PScript warnings will appear in the PHOTOSS warnings-output console (and the standard output console).

12.2 Editor Settings

Show Code Line Numbers
Code set: `PScriptSettings.setShowCodeLineNumbers(bool setBool)`
Code get: `myBool = PScriptSettings.getShowCodeLineNumbers()`
Default: `true`
Description: If set to `true`, the code line numbers will be displayed on the left of the PScript Code Area.

Use Syntax Highlighting
Code set: `PScriptSettings.setUseSyntaxHighlighting(bool setBool)`
Code get: `myBool = PScriptSettings.getUseSyntaxHighlighting()`
Default: `true`
Description: If set to `true`, syntax highlighting is enabled in the PScript Code Area. Please refer to the chapter PScript Syntax Highlighting and Code Assistant for more details.

Use Syntax Highlighting in Console
Code set: `PScriptSettings.setUseSyntaxHighlightingInConsole(bool setBool)`
Code get: `myBool = PScriptSettings.getUseSyntaxHighlightingInConsole()`
Default: `false`
Description: If set to true, syntax highlighting will be enabled in the PScript console. Please refer to the chapter PScript Syntax Highlighting and Code Assistant for more details.

Use Code Assistant
Code set: `PScriptSettings.setUseCodeAssistant(bool setBool)`
Code get: `PScriptSettings.getUseCodeAssistant()`
Default: `true`
Description: If set to `true`, the code assistant will be enabled in the PScript Code Area. Please refer to the chapter PScript Syntax Highlighting and Code Assistant for more details.

12.3 Log File Settings

Use Log File
Code set: `PScriptSettings.setUseLogFile(bool setBool)`
Code get: `mybool = PScriptSettings.getUseLogFile()`
Default: `false`
Description: If set to `true`, PScript will create a log file which contains various information depending on the following log file settings.

Include Time and Date in Log File
Code set: `PScriptSettings.setIncludeTimeAndDateInLogFile(bool setBool)`
Code get: `myBool = PScriptSettings.getIncludeTimeAndDateInLogFile()`
Default: `true`
Description: If set to `true`, the time and date information at the beginning of the execution of the PScript and at after the execution of the PScript will be included in the log file.

Include Detailed Information in Log File
Code set: `PScriptSettings.setIncludeDetailedInformationInLogFile(bool setBool)`
Code get: `mybool = PScriptSettings.getIncludeDetailedInformationInLogFile()`
Default: `true`
Description: If set to `true`, information which is included in a print command in a PScript context (e.g. `print("Hello World")`!) will be included in the log file.

Include Errors in Log File
Code set: `PScriptSettings.setIncludeErrorsInLogFile(bool setBool)`
Code get: `myBool = PScriptSettings.getIncludeErrorsInLogFile()`
Default: `true`
Description: If set to `true`, PHOTOSS errors will be included in the log file.

Include Warnings in Log File
Code set: `PScriptSettings.setIncludeWarningsInLogFile(bool setBool)`
Code get: `myBool = PScriptSettings.getIncludeWarningsInLogFile()`
Default: `true`
Description: If set to `true`, PHOTOSS warnings will be included in the log file.

Use Custom Log Save Path
Code set: `PScriptSettings.setLogFileUseCustomSavePath(bool setBool)`
Code get: `myBool = PScriptSettings.getLogFileUseCustomSavePath()`
Default: `false`
Description: If set to `true`, the user can define a specific, absolute save path under which the PScript log will be saved. If set to `false`, PScript will save log files under the default path:
`C:/Documents and Settings/USERNAME/Application Data/Lenge/PHOTOSS X.YZ/PScript`

Set / Get Custom Log Save Path
Code set: `PScriptSettings.setLogFileCustomSavePath(string myPathString)`
Code get: `mystring = PScriptSettings.getLogFileCustomSavePath()`
Default: `C:/Documents and Settings/USERNAME/Application Data/Lenge/PHOTOSS X.YZ/PScript`
Description: Sets the save path for the PScript logs to the given path string.

Set / Get Overwrite Log File
Code set: `PScriptSettings.setLogFileOverwrite(bool setBool)`
Code get: `myBool = PScriptSettings.getLogFileOverwrite()`
Default: `false`
Description: Sets the bool value which defines whether an old PScript log-file will be overwritten (`true`) or if the new log information will be appended to the log (`false`).

12.4 History Settings

Use History

Code set: `PScriptSettings.setUseHistory(bool setBool)`
Code get: `myBool = PScriptSettings.getUseHistory()`
Default: `true`
Description: If set to `true`, the command history will be displayed in the PScript Command History Area. The command history does only apply to commands which have been entered manually at the PScript prompt. It does not apply to scripts which are started by using the "run" buttons.

Clear History

Code: `PScriptSettings.clearCommandHistory()`
Default: *NA*
Description: If clicked (or called), the history will be cleared.

12.5 MATLAB® Settings

Use identical Matlab workspace for PHOTOSS and PScript

Code set: `PScriptSettings.setUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript(bool setBool)`
Alternative code set: `matlab.setUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript(bool setBool)`
Code get: `myBool = PScriptSettings.getUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript()`
Alternative Code get: `matlab.getUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript()`
Default: `true`
Description: If set to `true`, PHOTOSS Matlab components and the PScript MATLAB® workspace will be identical. You can automatically access PHOTOSS MATLAB® component workspace contents when using PScript. Please bear in mind that PScript will access the Matlab component workspace before or after a simulation run but not during a simulation run. For more details please refer to the chapter PScript and MATLAB® on page 19. If set to `false`, PScript and PHOTOSS MATLAB® components will use separate workspaces; variables cannot be transferred between the two workspaces.

Check for Matlab Errors

Code set: `PScriptSettings.setCheckForMatlabErrors(bool setBool)`
Alternative code set: `matlab.setCheckForMatlabErrors(bool setBool)`
Code get: `myBool = PScriptSettings.getCheckForMatlabErrors()`
Alternative code get: `myBool = matlab.getCheckForMatlabErrors()`
Default: `true`
Description: If set to `true`, errors and warnings which occur during the execution of a MATLAB® script will also automatically be shown in the PScript console. If set to `false`, the errors will *not* be shown. This means, that a running PScript will *never* be aborted even if MATLAB® errors will occur. Disabling this option is *not* recommended.

Break on Matlab Errors

Code set: `PScriptSettings.setBreakOnMatlabErrors(bool setBool)`
Alternative code set: `matlab.setBreakOnMatlabErrors(bool setBool)`
Code get: `myBool = PScriptSettings.getBreakOnMatlabErrors()`
Alternative code get: `myBool = matlab.getBreakOnMatlabErrors()`
Default: `true`
Description: If set to `true`, errors which occur during the execution of a MATLAB® script will result in a PScript error which will abort the currently running PScript. If set to `false`, the currently running MATLAB® script will be aborted but the currently running PScript will be resumed. Disabling this option is *not* recommended.

Matlab Timeout (in Seconds)

Code set: `PScriptSettings.setMatlabTimeoutInSeconds(int setInt)`
Alternative code set: `matlab.setMatlabTimeoutInSeconds(int setInt)`
Code get: `myInt = PScriptSettings.getMatlabTimeoutInSeconds()`
Alternative code get: `myInt = matlab.getMatlabTimeoutInSeconds()`
Default: `10.000 seconds`

Description: Allows the user to specify a timeout value for the execution of MATLAB® scripts during the execution of a PScript / a PHOTOSS simulation. If MATLAB® has not finished the execution of a script during the timeout period, a popup window will appear which allows the user to choose between several options regarding how to deal with the MATLAB® timeout. Please make sure that the timeout has been set to a sufficiently high time interval for your purposes

12.6 Include Settings

You can use the include settings to specify paths where your own (Sub-)PScripts reside. When you are invoking an `include()` operation, PScript will automatically search for the included PScript in the directories you have specified in the PScript include settings. For more details please refer to chapter **PScript Code Samples**, Example No. 16.

Add Include Path

Code: `PScriptSettings.addIncludePath(string pathString)`
Default: *NA*
Description: Adds a new path to the include list. PScript will not check whether the supplied path is valid. If the string you have specified is a file path and not a directory path, PScript will issue a warning. Keep in mind that you can also manually change any path of the Auto Include list from an absolute path to a relative path.

Remove Include Path

Code: `PScriptSettings.removeIncludePath(string pathString)`
Default: *NA*
Description: Removes the specified item from the include list. If the item you have specified does not exist, PScript will issue a warning.

Get Include Paths

Code: `incVec = PScriptSettings.getIncludePaths()`
Default: *NA*
Description: Returns a string vector which includes all include paths in the specified order. If no include paths have been defined, an empty vector will be returned.

Include Paths Contain

Code: `myBool = PScriptSettings.includePathsContain(string pathString)`
Default: `false`
Description: Returns a bool which specifies whether the supplied include path is already present in the include list.

Move Up / Down Include Path

Code up: `PScriptSettings.moveUpIncludePath(string pathString)`
Code down: `PScriptSettings.moveDownIncludePath(string pathString)`
Default: *NA*
Description: Moves the specified include path item up (or down) in the include path queue. If the specified item does not exist, PScript will issue a warning.

12.7 Auto Include Settings

The PScript Auto Include Settings work similar to the Include Settings with one exception: The script(s) you have specified will be run automatically *every time* when PScript is opened. This can become very convenient when you have designed global startup scripts or libraries which would be tedious to include from hand in each script. Please keep in mind that the command `clearWorkspace` destroys *all* libraries or objects and variables which you might have included in your Auto Includes. To make these objects available without having to restart PScript, you can use the command:

```
PScriptSettings.runAutoIncludes()
```

Add Auto Include
Code: `PScriptSettings.addAutoInclude(string pathString)`
Default: *NA*
Description: Adds a full (absolute) path to the Auto Include list. PScript will not check whether the supplied path is valid. If the string you have specified is a directory path and not a file path, PScript will issue a warning

Run Auto Includes
Code: `PScriptSettings.runAutoIncludes()`
Default: *NA*
Description: All PScripts which are included in the PScript Settings Auto-Include list, are run in the given order. This function can be useful when you have cleared the PScript workspace using `.clearWorkspace()` and you want to quickly restore all your libraries, etc. without having to call each individual script manually.

Remove Auto Include
Code: `PScriptSettings.removeAutoInclude(string pathString)`
Default: *NA*
Description: Removes the specified item from the Auto Include list. If the item you have specified does not exist, PScript will issue a warning.

Get Auto Includes
Code: `autoVec = PScriptSettings.getAutoIncludes()`
Default: *NA*
Description: Returns a string vector which includes all Auto Includes in the specified order. If no Auto Includes have been defined, an empty vector will be returned.

Auto Includes Contain
Code: `myBool = PScriptSettings.autoIncludesContain(string pathString)`
Default: `false`
Description: Returns a bool which specifies whether the supplied Auto Include is already present in the Auto Include list.

Move Up / Down Auto Include
Code up: `PScriptSettings.moveUpAutoInclude(string pathString)`
Code down: `PScriptSettings.moveDownAutoInclude(string pathString)`
Default: *NA*
Description: Moves the specified Auto Include item up (or down) in the Auto Include queue. If the specified item does not exist, PScript will issue a warning.

13 Using PScript in Command Line Mode / Using PScript with Condor®

You can also start a PScript and PHOTOSS from the Command Line Interface (CLI). Therefore, you need to change to your PHOTOSS installation directory. In the following example we will assume that it has been set to: `C:\PHOTOSS\`. The Default path would normally be:

```
C:\Documents and Settings\USERNAME\Application Data\Lenge\PHOTOSS X.YZ\
```

You can now load and automatically execute any PScript by using the following syntax:

```
C:\PHOTOSS\PHOTOSS.exe -r "ScriptPath\Scriptname.pscript"
```

Where the phrase `ScriptPath` stands for the full path to where your PScript actually resides. E.g. `C:\MyScripts\myPScript.pscript`.

If you want to execute the PScript in a PHOTOSS instance that is already opened, type:

```
C:\PHOTOSS\PHOTOSS.exe -c -r "ScriptPath\Scriptname.pscript"
```

You should also keep in mind that for certain scenarios it may be useful to automatically close the PHOTOSS application after the PScript has finished - e.g. for the combined use of PHOTOSS, PScript and Condor®. In this case, you may use the line:

```
application.close()
```

as a last line in your PScript. For example, if you want to submit a PHOTOSS and PScript job which should be computed on a grid by using Condor® it is necessary that both PScript and PHOTOSS terminate after the job has been completed (so that the grid computing tool "knows" that the job has been finished).

For more information on how to create PHOTOSS / PScript Condor® Jobs, example job files and how to distribute these jobs to various machines in your simulation pool, please refer to the PHOTOSS Grid Computing Manual which is also included in your PHOTOSS installation.

14 PScript Syntax Highlighting and Code Assistant

The PScript editor offers syntax highlighting (SHL) and a comfortable code assistant to make it easier for you to create your own PScripts. SHL can be enabled and disabled using the PScripts settings; the default setting is `true`. The following SHL features are currently included:

- code comments (green)
- headlines for code comments (green)
- Qt script basic functionality such as `for` and `if` statements (dark blue)
- `int`, `double` and `float` values (dark blue)
- strings (red)
- PScript prefixes such as `matlab` and `application` (blue)
- PScript MATLAB®-specific suffixes and functions such as `matlab.setValue()` (blue and set in italics)
- application and simulation suffixes and functions such as `application.openSimulation()`(blue and set in italics)
- user defined `variables`(black)
- numbers (dark red)

The code assistant can be enabled and disabled using the PScript settings (see page 22), the default setting is `true`. When you are typing your PScript code, the code assistant will automatically come up with suggestions to complete the current expression.
For example, when trying to append a suffix to a `matlab`.-expression, the code assistant will automatically offer all valid suffixes for the PScript MATLAB® object, such as `matlab.getValue()`, `matlab.setValue()`, etc.

To accept a suggestion of the assistant, highlight it using the arrow keys “up” and “down” or by clicking in the suggestion list and press “Return” or “Tab”. If you do not want to accept a suggestion of the code assistant, press “Esc” to close the assistant or keep writing.

15 PScript Code Samples

To create a more powerful PScript, the following code samples might be extremely helpful. You can access them by selecting `Code Samples` ⇒ ... from the PScript main menu. You will need write permission in your user home directory to successfully run the examples. The default path is:
`C:/Documents and Settings/USERNAME/Application Data/Lenge/PHOTOSS X.YZ/`,
where `X.YZ` denotes the currently installed version of PHOTOSS.

- `01 Hello World.pscript` demonstrates how to write output to the PScript console.
- `02 Open And Run Simulation.pscript` teaches you how to use PScript to open, run, clear and close a PHOTOSS simulation
- `03 Directories.pscript` will make you familiar with the syntax necessary to switch the PScript directory and use paths.
- `04 Script Output.pscript` shows you how to direct Output to the PScript and / or PHOTOSS console
- `05 Investigate Simulation.pscript` will illustrate how all simulation parameters, components and component parameters inside a PHOTOSS simulation can be accessed by PScript.
- `06 Change Parameter.pscript` will enable you to change parameters of single components and the simulation parameters. You will learn how to change parameters of the type "double", "radio box", "pull down menu" and "formula" / "string".
- `07 Change Parameter for a Type of Component.pscript` enables you to change parameters for a group components which belong to the same component type (e.g. "Analytical Filter"). Also introduces the PScript "foreach" functionality.
- `08 Reflective Simulation.pscript` explains how to evaluate results of components / analyzers and continue a parameter variation until user-defined criteria are met.
- `09 Creating Components.pscript` shows how components can be created and linked by using PScript.
- `10 Open And Run Parameter Variation.pscript` explains how to create, run and analyze a parameter variation in PScript.
- `11 Matlab Interface Interaction.pscript` demonstrates you how to use both the MATLAB® interface and PScript at the same time. You will learn how to call MATLAB® scripts using PScript and how to exchange data between the MATLAB® and PScript workspace.
- `12 Simulation Parameters.pscript` shows how simulation parameters can be accessed and changed using PScript.
- `13 Type Conversions.pscript` demonstrates some basic type conversions in PScript.
- `14 Simulation Control Handling.pscript` illustrates the basic concept of how to handle PHOTOSS simulations with PScript.
- `15 Handling Networks and Iterators.pscript` teaches you how to set parameters or access results of components inside a network or iterator component.
- `16 Includes.pscript` will familiarize you with several easy ways to include Sub-PScripts containing sub-functions, etc. in your Main PScript.
- `17 Generic Scripting.pscript` demonstrates how generic scripts can be created which offer great flexibility and functionality at the same time.

16 PScript Function Reference

The following list includes all PScriptfunctions for the interaction between PScript, PHOTOSS and MATLAB®. Each function will be followed by a short example code and explanation to illustrate how it works.

16.1 PScript Options / Settings

Please refer to the chapter **12 - PScript Settings Dialog** on page **22** for a detailed explanation of the PScript settings and all available functions.

16.2 General Functions

General functions can be called directly in the PScript context. They do not have to be called on any object. Table 1 lists all the general functions which will be explained in more detail below.

Table 1. Overview of General Functions

Function Name	Short description	Page
`foreach`	iterates over all elements of a vector	32
`clearWorkspace()`	removes all objects and variables	32
`collectGarbage()`	invokes garbage collection	32
`execute()`	executes CMD command	32
`include()`	include `.pscript` file	32
`resetEngine()`	invoke `clearWorkspace()` and `runAutoIncludes()`	32
`runAutoIncludes()`	execute all automatically included `.pscript` files	32
`who()`	list all available objects in the workspace	33

Function: `foreach(double array, function(current element name){function code xyz})`
Code: `foreach(s.getComponentsByType("Laser"), function(c){c.setParameterValue("Gamma","1.3")})`
Description: Iterates overall elements of a one-dimensional input vector and carries out the function specified in the function code section. The current element can be addressed by using the "current element name" inside the function code.

Function: `clearWorkspace()`
Code: `clearWorkspace()`
Description: Removes all user-defined objects and variables (e.g. arrays, vectors, scalar values, etc.) from the PScript workspace. Note that PScript Auto Includes will not be executed.

Function: `collectGarbage()`
Code: `collectGarbage()`
Description: Invokes the garbage collection to remove inaccessible objects (etc.) in the script environment.

Function: `execute(string commandLineCode)`
Code: `execute("cmd.exe")`
Description: Executes the command given in the `commandLineCode string`. Please keep in mind that the current program directory of PScript and PHOTOSS might also have an effect on the programs you invoke with the `execute`-command.

Functions: `include(string scriptName)//if the path is present in the include list`
`include(string scriptNamePath)//if the path is not present in the include list`
Code: `include("myNewScript.pscript")`
Description: When called, the code specified in PScript file whose name is given by the `string scriptName` will be evaluated. If the path where the given PScript resides is present in the include list, you just have to specify the script name. If the path is not present in the include list, you have to specify an absolute or relative path. Refer to chapter 12 - PScript Settings Dialog on page 22 for more details on the include list.

Function: `resetEngine()`
Code: `resetEngine()`
Description: Same as `clearWorkspace` with the exception that PScriptAuto Includes *will* be executed.

Function: `runAutoIncludes()`
Code: `runAutoIncludes()`
Description: Same as `PScriptSettings.runAutoIncludes()` (see page 32).

Function: `who()`
Code: `who()`
Code: `application.who()`
Code: `matlab.who()`
Code: `PScriptSettings.who()`
Description: Calling the `who()` function will produce a list of all currently available objects in the PScript workspace which will be displayed in the console. The list contains all variables of primitive data types such as `int`, `double`, `bool`, `string` and all functions as well as all "'objects"' in the workspace which can be created by calling `myObject = new Object()`. The `who` function can be called on the `application`, `matlab` and `PScriptSettings` as well to show all functions (and objects) contained in these objects. Please keep in mind that `who` does not work recursively and only displays objects at the top level.

16.3 Generating Random Numbers in PScript

To generate random numbers, you have to call the functions on an `rng` object which is automatically available to you once a PScript workspace has been opened. All functions related to the generation of random numbers are listed in table 2 and are explained in more detail below. Please note that the PScript Random Generator is in *not* connected to the random generator which is used in a PHOTOSS *simulation*! If you want to change the random generator settings of a simulation, you have to change the corresponding *simulation parameter*, e.g. by using
`s.setSimulationParameterValue("randomGeneratorSeed", -100)`, etc.

Table 2. Overview of Random Generator-Related Functions

Function Name	Short description	Page
`rng.fastpoisson()`	generate a fast poisson-distributed number	34
`rng.gauss()`	generate a normally-distrubuted number	34
`rng.getDefaultSeed()`	returns the default seed for the `rng` object	34
`rng.negExp()`	generate a negatively exponentially-distributed number	34
`rng.pareto()`	generate a pareto-distributed number	34
`rng.poisson()`	generate a poisson-distributed number	34
`rng.setDefaultSeed()`	set the default seed for the `rng` object	34
`rng.setDeterministic()`	set the `rng` to deterministic mode	35
`rng.setRandom()`	set the `rng` to random mode	35
`rng.uniform()`	generate a uniformly-distributed number	35

Function: `double fastpoisson(double lambda)`
Code: `alpha = rng.fastpoisson(2.31)`
Description: Returns a fast Poisson-distributed double random number with the Poisson parameter lambda.

Function: `double gauss(double mean, double var)`
Code: `alpha = rng.gauss(0.43, .084)`
Description: Returns a normally distributed double random number with specified mean and variance.

Function: `int getDefaultSeed()`
Code: `default_seed = rng.getDefaultSeed()`
Description: Writes the default seed for the PScript random generator of the current simulation into a PScript variable.

Function: `double negExp(double lambda)`
Code: `alpha = rng.negExp(2.31)`
Description: Returns a negatively exponentially distributed random number with parameter lambda.

Function: `double pareto(double min, double mean)`
Code: `alpha = rng.pareto(2.0, 1.34)`
Description: Returns a Pareto-distributed double random number larger or equal to the minimal value and with specified mean.

Function: `double poisson(double lambda)`
Code: `alpha = rng.poisson(2.31)`
Description: Returns a Poisson-distributed double random number with the Poisson parameter lambda.

Function: `setDefaultSeed(int seed)`
Code: `rng.setDefaultSeed(-324525)`
Description: Sets the default seed for the PScript random generator. This default seed will be used when calling `setDeterministic()` without a parameter.

Function: `setDeterministic(int seed)`
Code: `rng.setDeterministic(-352215)//Supply a user seed`
Alternative Code: `rng.setDeterministic()//Use the default seed`
Description: Sets the PScript random generator of the current simulation to deterministic mode and supplies a seed for the generation of random numbers. The seed must be chosen to have a negative sign. If no integer seed is supplied, the default-seed for the random generator will be chosen.

Function: `setRandom()`
Code: `rng.setRandom()`
Description: Sets the PScript random generator of the current simulation to a pseudo random mode. Each time you restart the current simulation, the random generator will be initiated with a different, (semi-)randomly chosen seed.

Function: `double uniform(double min, double max)`
Code: `alpha = rng.uniform(0.00 , 1.00)`
Description: Returns a uniformly distributed double random number in the interval [min, max].

16.4 Functions concerning the application class:

The functions concerning the PHOTOSS application class can only be called on the `application` object. All functions are listed in table 3 and are explained in detail below.

Table 3. Overview of Application-Related Functions

Function Name	Short description	Page
`application.appendStringToFile()`	appends a given string to a file	36
`application.close()`	closes PHOTOSS and PScript	36
`application.closeSimulations()`	closes all opened simulations	36
`application.copyFromTo()`	copies all files from a given folder to another folder	37
`application.createDirectory()`	creates a new directory	37
`application.deleteDirectory()`	deletes an existing directory	37
`application.deleteFile()`	deletes specified file	37
`application.deleteFileType`	deletes all files of specified type	37
`application.doesFileExist`	returns, whether a file or directory exists at a given path	37
`application.getApplicationDataDirectory()`	returns the path where PHOTOSS saves files, etc.	37
`application.getCurrentDirectory()`	returns path of the working directory of PScript	38
`application.getPHOTOSSExecutableDirectory()`	returns the path to the PHOTOSS executable	37
`application.getPHOTOSSOptionDescription()`	returns the description of specified option	37
`application.getPHOTOSSOptionValue()`	returns the value of the specified option	37
`application.getScriptDirectory()`	returns the directory of current PScript	38
`application.getSimulations()`	returns a list of all opened simulations	38
`application.getUserHomeDirectory()`	returns path of the user home directory	38
`application.info()`	displays text in the console	38
`application.listPHOTOSSOptions()`	lists all PHOTOSS options	38
`application.loadVariables()`	loads all or some selected variables from a file	38
`application.newSimulation()`	creates and saves a new simulation	38
`application.openSimulation()`	opens a simulation	38
`application.readStringArrayFromFile()`	reads data from a file and stores it into a string array	38
`application.saveVariables()`	saves all or some selected variables to a file	38
`application.setCurrentDirectory()`	sets the working directory of PScript	38
`application.setPHOTOSSDefaultOptions()`	reset all PHOTOSS options to default	39
`application.setPHOTOSSOptionValue()`	set specified PHOTOSS option	39
`application.timestamp()`	returns a current timestamp	39
`application.who()`	lists all objects contained in the `application` object	33
`application.writeStringArrayToFile()`	reads data from a string array and stores it into a file	39

Function: `application.appendStringToFile(string destinationPath)`
Code: `application.appendStringToFile("C:\\Test\\")`
Description: This function appends a string to a given file in the specific folder given by `destinationPath`. The string will be add in a new row after the last given row of the file. Absolute and relative paths are possible.

Function: `application.close()`
Code: `application.close()`
Description: Closes the PScript main window and (!) the PHOTOSS application. This function should be used when operating in the batch mode in combination with a grid computing tool because the grid computing tool might require PHOTOSS to shut down in order to acknowledge that the current job has been completed. Support for this functionality will be brought to you in future releases.

Function: `application.closeSimulations()`
Code: `application.closeSimulations()`
Description: Closes all opened Simulations.

Function: `application.copyFromTo(string sourcePath, string destinationPath)`
Code: `application.copyFromTo("C:\\Test\\", "D:\\Example\\")`
Description: Copies all files in the folder given by `sourcePath` to the folder specified by the argument `destinationPath` If the destination folder does not exist, PScript will try to create it. If the source folder does not exist, an error message will be displayed.

Function: `application.createDirectory(string directoryPath)`
Code: `application.createDirectory("C:\\Test\\")`
Description: Creates a new, empty directory for the given directory path. If the directory already exists, nothing will happen. An error message will be displayed if the directory can not be created.

Function: `application.deleteDirectory(string directoryPath)`
Code: `application.deleteDirectory("C:\\Test\\")`
Description: Deletes an existing directory for the given directory path and all files and subfolders within.

Function: `application.deleteFile(string fileNameString)`
Code: `application.deleteFile("useless.pho")`
Description: Deletes the file with given name in the directory which is specified by `.getCurrentDirectory()` as a default. You can also specify a `string` which holds the absolute path to a file you want to delete. Please keep in mind that PScript deletes the given file *permanently* - it will not be moved to the recycle bin. A message will be displayed in the PScript console, if the file is deleted successfully. If the file you have specified does not exist, a warning will be issued by PScript.

Function: `application.deleteFileType(string DirectoryString, string fileTypeString)`
Code: `application.deleteFileType("*.pho")`
Alternative Code: `application.deleteFileType("pho")`
Description: Deletes all types of the given type in the directory which is specified by `.getCurrentDirectory()` as a default. You can also specify a `string` which holds the absolute path to a directory in which you want to delete all files of the given type. Please keep in mind that PScript deletes the given files *permanently* - they will not be moved to the recycle bin. If no file(s) of the type you have specified do exist, a warning will be issued by PScript.

Function: `bool application.doesFileExist(string filePath)`
Code: `application.doesFileExist("C:\\Test.pscript")`
Description: Returns `true`, if a file or directory will exist for a given file path.

Function: `string application.getApplicationDataDirectory()`
Code: `mystring = application.getApplicationDataDirectory()`
Description: Returns a string containing the directory which is used by PHOTOSS to save program options, etc. Note that you do need permission to write data to this directory or PHOTOSS / PScript may not run properly. The default directory is:
`C:/Documents and Settings/USERNAME/Application Data/Lenge/PHOTOSS X.YZ`

Function: `string application.getPHOTOSSExecutableDirectory()`
Code: `mystring = application.getPHOTOSSExecutableDirectory()`
Description: Returns a string containing the name of the directory in which the PHOTOSS.exe file of the currently opened instance of PHOTOSS resides.

Function: `string application.getPHOTOSSOptionDescription(string optionNameString)`
Code: `myString = application.getPHOTOSSOptionDescription("forceClearPromptly")`
Description: Displays the description of the given PHOTOSS option in the PScript Console. Trying to access non-existing options will produce an error. Use `application.listPHOTOSSOptions()` to get an overview of which options are available. Keep in mind that these options are the same as in the PHOTOSS GUI.

Function: `bool /string application.getPHOTOSSOptionValue(string optionNameString)`
Code: `myOptionBool = application.getPHOTOSSOptionValue("forceClearPromptly")`
Description: Read the given PHOTOSS option value into a string or boolean variable (depending on the type of option you have specified). Trying to access non-existing options will produce an error. Use `application.listPHOTOSSOptions()` to get an overview of which options are available. Keep in mind that these options are the same as in the PHOTOSS GUI.

Function: `string application.getScriptDirectory()`
Code: `mystring = application.getScriptDirectory()`
Description: Returns a string containing the name of the directory in which the currently running PScript resides. This function can only be evaluated *during* a script run. It is useful when you want access to directories and files relative to the actively running script.

Function: `simulationObject[] application.getSimulations()`
Code: `mySimulationArray = application.getSimulations()`
Description: Returns a vector containing the names of all active PHOTOSS simulations.

Function: `string application.getUserHomeDirectory()`
Code: `mystring = application.getUserHomeDirectory()`
Description: Returns a string containing the path to the current user's home directory

Function: `string application.info(string message)`
Code: `application.info("This is a test message.")`
Description: Prints a string to both the PHOTOSS application and the PScript output consoles.

Function: `application.loadVariables(string filePathString)`
Code: `application.loadVariables("C:\\Test\\PScript.xml")`
Description: This function loads variables from a XML-File at a given filepath. If no filepath is specified, the file will be loaded from the UserHomeDirectory.

Function: `application.listPHOTOSSOptions()`
Code: `application.listPHOTOSSOptions()`
Description: Lists all available PHOTOSS options and their current setting in the PScriptconsole.

Function: `application.newSimulation(string filePathString)`
Code: `application.newSimulation("C:\myNewSimulation.pho")`
Description: Creates a new, empty simulation and saves it to the specified filePathString. If the path (and filename) specified in the filePathString is not valid, PScript will issue an error.

Function: `simulationObject application.openSimulation(string filename)`
Code: `s = application.openSimulation("MyExampleSimulation.pho")`
Description: Opens the specified simulation file (*.pho) with the PHOTOSS application. If no absolute path is given, PScript will try to open a simulation relative to the path specified in `application.getCurrentDirectory()`

Function: `application.readStringArrayFromFile(string sourcePath)`
Code: `application.readStringArrayFromFile("C:\\Test\\")`
Description: This function reads data from a given file in the specific folder given by `sourcePath` and stores it into a string array. Absolute and relative paths are possible. The whole data from one row will be stored in one array position as a string. The first row is stored in the first position of the array and so on.

Function: `application.saveVariables(string filePathString, string variables)`
Code: `application.saveVariables("C:\\Test\\", ["testVariable1","testVariable2","testVariable3"])`
Description: This function saves specific variables into a XML-File at a given filepath. If no variables are specified, all variables will be saved at the given destinationPath. If there are no arguments, all variables will be stored in the ScriptDirectory. If the ScriptDirectory is empty, all variables will be stored in the UserHomeDirectory.

Functions: `application.setCurrentDirectory(string pathString)`
`string application.getCurrentDirectory()`
Code set: `application.setCurrentDirectory("C:/MyTestDirectory")`
Code get: `myDir = application.getCurrentDirectory()`
Description: Set-method changes the current directory to the specified directory. This is the directory, from which PScript and PHOTOSS will open or save to simulation files. The get-method will read the directory into a string.

Function: `bool application.setPHOTOSSDefaultOptions()`
Code: `application.setPHOTOSSDefaultOptions()`
Description: Resets all PHOTOSS options to their default value.

Function: `application.setPHOTOSSOptionValue(string optionNameString bool / string optionValue)`
Code: `application.setPHOTOSSOptionValue("forceClearPromptly", true)`
Description: Set the given PHOTOSS option to the specified value (either a `bool` or a `string`). Trying to access non-existing options will produce an error. Use `application.listPHOTOSSOptions()` to get an overview of which options are available. Keep in mind that these options are the same as in the PHOTOSS GUI. However, the GUI names have been abbreviated for easy access.

Function: `string application.timestamp()`
Code: `application.timestamp()`
Description: This function will return a current timestamp based on the system clock with the following structure: timestamp = year + month + day + "T" + hours + minutes + seconds; e.g. 20130503T112235.

Function: `application.writeStringArrayToFile(string destinationPath)`
Code: `application.writeStringArrayToFile("C:\\Test\\")`
Description: This function writes data from a string array and writes it into a given file in the specific folder given by `destinationPath`. Absolute and relative paths are possible. The function iterates over the whole array from the beginning to the end. The string in the first position of the array will be stored in the first row of the given file and so on. If the file is not empty, the previous data will be overwritten.

16.5 Functions concerning the simulation class:

Please note that these functions only work if you have already opened at least one simulation by using `sim = application.openSimulation()`.

All functions need to be called on a simulation object which will referred to as `sim` for the remainder of this paragraph. The functions are listed in table 4 and are explained in more detail below.

Table 4. Overview of Simulation-Related Functions

Function Name	Short description	Page
`sim.clear()`	clears the simulation	40
`sim.close()`	closes the simulation	40
`sim.containsComponent()`	returns whether the component type exists in given simulation, network or iterator	40
`sim.deleteComponent()`	deletes specified component	41
`sim.deleteAllComponents()`	deletes all components	41
`sim.getComponent()`	returns component object with specified name	41
`sim.getComponents()`	returns all component objects in a list	41
`sim.getComponentsByType()`	returns all component objects of specified type	41
`sim.investigate()`	show all components and parameters in specified simulation	41
`sim.isOpen()`	returns whether the simulation is opened	41
`sim.listComponents()`	lists all components in the simulation	41
`sim.run()`	runs the simulation	41
`sim.save()`	save the simulation	42
`sim.saveAs()`	save the simulation under specified path and file name	42

Function: `simulationObject.clear()`
Code: `mySimulation.clear()`
Description: Clear the currently selected simulation. Clearing a simulation does *not* delete the results or parameters of the simulation in the PHOTOSS workspace. You can still access the results using `.getResultValue()`, etc.

Function: `simulationObject.close()`
Code: `mySimulation.close()`
Description: Closes the currently selected simulation. If you have made any changes to the simulation setup (parameters, etc.), you will be asked if you want to save the simulation if you have not already done so.

Function: `componentObject[] simulationObject.containsComponent(string componentName)`
Code:

```
1 mySimulation.containsComponent("componentName") //Return all components with the name "
      componentName" in the whole simulation.
2 mySimulation.containsComponent("beginningNamePart*")  // Return all components that begin
       with the phrase "namePart" in their name.
3 mySimulation.containsComponent("*endNamePart")  // Return all components that end with
      the phrase "namePart" in their name.
4 mySimulation.myNetwork.containsComponent("componentName")  // Return all components named
       "componentName" in the specified network and subnetworks.
5 mySimulation.myNetwork.containsComponent("beginningNamePart*")  // Return all components
      that begin with the name phrase "beginningNamePart" in the specified network and
      subnetworks.
6 mySimulation.myNetwork.containsComponent("*endNamePart") //Return all components that end
       with the name phrase "endNamePart" in the specifies network and subnetworks.
```

Description: This function can be used to supply a name or name phrase of the desired component(s). It will return a vector holding all occurrences of the component in the simulation or network. If no component with the specified name or name phrase exists in the simulation or network, the function returns an empty vector. Keep in mind, that the name of a component is unique when considering a single network level. Components in different subnetworks can have the same name(s), though! However, they can be accessed without ambiguity since the returned component vector automatically contains the whole address information in which subnetwork the component resides. Use `componentVector[i].getName()` to receive the full address of the component.

Function: `simulationObject.deleteComponent(string componentNameString)`
Code: `mySimObject.deleteComponent()`
Description: Deletes the specified component in the given simulation. The component (and its parameters and results) will be completely removed from the simulation. The function can also be called on a component object object and it will directly delete the component. Trying to delete non-existing components will issue an error.

Function: `simulationObject.deleteAllComponents()`
Code: `mySimObject.deleteAllComponents()`
Description: Deletes all components in the given simulation. This function is the same as calling `deleteComponent()` on every component in the simulation manually. If a simulation does not contain any components, the function will return `false`.

Function: `componentObject simulationObject.getComponent(string componentName)`
Code: `myFilter = mySimulation.getComponent("Filter1")`
Description: Returns a component object with the desired name from the currently active simulation. PHOTOSS will automatically ensure that each component name on the same network level is unique. If the component with the desired name in the simulation does not exist, PScript will throw an exception.

Function: `componentObject simulationObject.getComponents()`
Code: `myFilter = mySimulation.getComponents()`
Description: Returns all component objects in a list from the currently active simulation.

Function: `componentObject[] simulationObject.getComponentsByType(string typeName)`
Code: `allCurrentFiltersArray = mySimulation.getComponentsByType("Analytical Filter")`
Description: Returns a vector which includes all component objects of the specified type in the given simulation. Components of the specified type which reside in (sub-)network or an iterator will always be included. If no component of the specified type does exist in the given PHOTOSS simulation, the function returns an empty vector

Function: `simulationObject.investigate()`
Code: `mySimObject.investigate()`
Description: Lists all available components, component parameters and results of the given simulation. Please keep in mind that the results will only be available after the simulation has finished.

Function: `bool simulationObject.isOpen()`
Code: `isSimOpen = mySimulationObject.isOpen()`
Description: Returns a bool which denotes whether the given simulation (object) is opened (`true`) or not (`false`).

Function: `simulationObject.listComponents()`
Code: `mySimulationObject.listComponents()`
Description: Displays an overview of all components inside the given simulation in the PScript console

Function: `simulationObject.run()`
Code: `mySimulationObject.run()`
Description: Starts the currently selected simulation. If the PHOTOSS option “Save automatically when starting a simulation” is activated, any changes made to the active simulation setup with PScript or PHOTOSS will be saved, overwriting the old *pho-file. Only one simulation can be run at a time. If the simulation consists of a parameter variation, only the current values of the simulation parameters (and its placeholders) will be used and the simulation is carried out only once. If you want to execute the full parameter variation, use the function `getParameterVariationScript()` described on page 16.7 instead.

Function: `simulationObject.save()`
Code: `mySimulation.save()`
Description: Saves the currently selected simulation (object). If the simulation file already exists, it will be overwritten without prompting a request.

Function: `simulationObject.saveAs(string fileName)`
Code: `mySimulation.saveAs("MyNewSimulation.pho")`
Description: Saves the currently selected simulation under the given filename. If the simulation file already exists, it will be overwritten without prompting a request.

16.6 Functions concerning the component class:

All functions of the PScript component class have to be called on a component object which can be obtained by using `comp = sim.getComponent()` as described on page 41. Also keep in mind that you can always call "combined" functions like `myVal = simulationObject.ComponentObject.getName()`, you do not have to save the component to a component object in order to call the following functions. All relevant functions are listed in table 5 and explained in more detail below.

Table 5. Overview of Component-Related Functions

Function Name	Short description	Page
`sim.createComponent()`	creates a new component on the grid	43
`sim.createAndConnectComponent()`	creates a new component on the grid and connects it	44
`sim.getComponent()`	returns the component object	41
`comp.deleteComponent()`	deletes the component	44
`comp.getBypass()`	returns whether the component is bypassed	44
`comp.getComponentType()`	returns the component type	44
`comp.getGridCoordinates()`	returns grid coordinates of current component	44
`comp.getIterationResultValues()`	return Results of an iterator	44
`comp.getName()`	returns the name of the component	45
`comp.getParameter()`	returns the parameter object for specified parameter	45
`comp.getParameterDescription()`	returns the description for specified parameter	45
`comp.getParameterNames()`	returns names of all component parameters	45
`comp.getParameters()`	returns parameter object of all component parameters	45
`comp.getParameterUnit()`	returns unit of specified parameter	45
`comp.getParameterValue()`	returns the value of specified parameter	45
`comp.getParameterValues()`	returns the values of all parameters	46
`comp.getPMDPath()`	is the current component is a PMD path member?	46
`comp.getResult()`	returns a result object of the specified result	46
`comp.getResultNames()`	returns the names of all component results	46
`comp.getResults()`	returns all result component result objects	46
`comp.getResultUnit()`	returns unit of specified result	46
`comp.getResultUnits()`	returns units of all component results	46
`comp.getResultValue()`	returns the value of the specified result	46
`comp.getResultValues()`	returns the values of all component results	46
`comp.isComponentOfType()`	returns whether the component is of specified type	46
`comp.isIterator()`	returns whether the component is of type iterator	47
`comp.isNetwork()`	returns whether the component is of type network	47
`comp.listParameters()`	lists all parameters of the component	47
`comp.listResults()`	lists all results of the component	47
`comp.setBypass()`	set the bypass of the component	47
`comp.setName()`	changes the name of the component	47
`comp.setParameterValue()`	sets the value of specified component parameter	47
`comp.setPMDPath()`	sets PMD path membership of the component	47

Functions:

```
simObject.createComponent(string compType, int xCoordinate, int yCoordinate)
simObject.createComponent(string compType, int xCoordinate, int yCoordinate, string compName)
```

Code: `mySimulationObject.createComponent("Single Mode Fiber", 5,6)`
Code: `mySimulationObject.createComponent("Single Mode Fiber", 5,6, "myNewFiber")`
Description: Creates a new component of the type specified in the `string componentType` at the given x- and y-coordinates. Optionally, you may also specify the name of the new component. If you do not specify a name, PHOTOSS will automatically determine the next available component name by using its default notation and increasing the numeral suffix by one if necessary. Trying to specify a component

type which is not defined or specifying a name which is already in use will issue an error.

Functions:

```
simObject.createAndConnectComponent(string oldCompName, string newCompType)
simObject.createAndConnectComponent(string oldCompName, string newCompType, string newCompName)
simObject.createAndConnectComponent(string oldCompName, int baseOutPort, string newCompType, int newInPort)
simObject.createAndConnectComponent(string oldCompName, int baseOutPort, string newCompType, int newInPort,
string newCompName)
```

Code: `mySimulationObject.createAndConnectComponent(s.pulseGen,"Single Mode Fiber")`
Code: `mySimulationObject.createAndConnectComponent(s.pulseGen,0, "Single Mode Fiber", 0 "newSMF")`
Description: This function creates a new component of the specified type on the grid and automatically connects it to the specified "old" component with the name `oldCompName`. You can also specify which Out-Port of the old component should be connected to which In-Port of the new component. If you do not specify the ports, the first free Out-Port of the old component will be connected with the first In-Port of the newly created component. You can also optionally specify a name for the new component. Thew new component will be placed to grid coordinates to the right of the existing component, if the space is unoccupied. If it is already occupied, PHOTOSS will try to find a free space near to the existing component.

Function: `componentObject.deleteComponent()`
Code: `myComponentObject.deleteComponent()`
Description: Deletes the component object on which the function is called. The component (and its parameters and results) will be completely removed from the simulation. The function can also be called on a simulation object and it will delete the component given by the supplied `componentNameString`. Trying to delete non-existing components will issue an error.

Function: `bool componentObject.getBypass()`
Code: `smfBypass = smf.getBypass()`
Description: Returns a bool value which specifies if the currently selected component is bypassed (`true`) or not (`false`). The default is `false`.

Function: `string componentObject.getComponentType()`
Code: `myCompType = s.iterator.getComponentType()`
Alternative Code: `myCompType = myCompObject.getComponentType()`
Description: Returns a string which contains the type of the given component object. Please note that the type of a component might not be related to its name (since you can assign your own names to any PHOTOSS component). The component type can never be changed; e.g. a "Pulse Generator" will always be a "Pulse Generator", regardless of its name.

Functions:

```
string componentObject.getGridCoordinates()
string componentObject.getGridCoordinates(string dimensionString)
```

Code: `coordVector = sim.smf.getGridCoordinates()`
Code: `xCoord = sim.smf.getGridCoordinates("x")`
Description: Returns a vector containing the x- and y-coordinates of the specified component. If used on a component which resides inside a network or an iterator, the coordinates inside the network are returned. If you specify a dimension (may be either `"x"` or `"y"`) the function will return the coordinate for the specified dimension only.

Function: `double[] iteratorComponent.componentObject.getIterationResultValues("resName")`
Code:

```
1 valVec = mySimulation.iterator1.SMF.getIterationResultValues("DGD") //Returns a vector
      containing all results with the name "DGD" in the component named "SMF" in the
      iterator "iterator1".
2 allValsVec = mySimulation.iterator1.SMF.getIterationResultValues() //Returns a vector
      containing all results of the component "SMF" inside the specified iterator component
       named "iterator1".
```

Description: The function returns a vector of all results or all specified results which are included in a given component which resides in an iterator component. If, for example, the iterator contains four iteration runs, each specified result name will result in a return vector of the length four (one for each iteration). If no result name is specified, all results of the specified component will returned. In this case, all results of the first iteration are sorted in descending order. The results of further iterations are sorted in the same manner. Please keep in mind that you need to check the appropriate radio boxes of all results of your component(s) you wish to receive (of course, you can alternatively use PScript to set the component parameters on the "results" tab, e.g. `s.smf.setParameterValue("DGD_derivation", true)`)

Function: `string componentObject.getName()`
Code: `myComponentName = mySimObject.analyticalFilter1.getName()`
Alternative Code: `myComponentName = myComponentObject.getName()`
Description: Returns a string containing the name of the specified component. You will find that this function can be useful when iterating over a vector of multiple component objects:
`compName= componentVec[i].getName()`

Function: `parameterObject componentObject.getParameter(string parameterName)`
Code: `smfLength = mySimObject.smf.getParameter("length")`
Alternative Code: `parameterObj = componentObject.getParameter("length")`
Description: Returns the specified parameter object of the specified component. If the specified parameter does not exist in the currently active component, an exception will be thrown. Please note that the parameter object is not the value of the parameter but an object containing the value, the name and the unit of the parameter. If you want to access the value of the parameter, use the `getValue()`-function on the parameter object or use the function `getParameterValue()` to directly access its value.

Function: `string componentObject.getParameterDescription(string parameterNameString)`
Code: `myDesc = mySimObject.smf.getParameterDescription("DeltaBeta_1")`
Description: Returns the description string of the given parameter. This is the same description which is also shown in the GUI when you open the component parameter dialog and click on the appropriate parameter.

Function: `string[] componentObject.getParameterNames()`
Code: `nameVec = mySimObject.smf.getParameterNames()`
Alternative Code: `nameVec = myComponentObject.getParameterNames()`
Description: Returns a vector containing the names of all parameters of the specified component or component object.

Function: `parameterObject[] componentObject.getParameters()`
Code: `parObjVec = simulationObject.smf.getParameters()`
Alternative Code: `parObjVec = componentObject.getParameters()`
Description: Returns a vector containing all parameter objects of the specified component. Please note that the parameter objects are not the values of the parameters but objects containing the values, the names and the units of the parameters. If you want to directly access the value of the parameters, use the function `getParameterValues()`.

Function: `string componentObject.getParameterUnit(string parameterName)`
Code: `parUni = mySimObject.smf.getParameterUnit("length")`
Description: Returns a string containing the unit of the specified parameter. If the parameter does not have a unit, an empty string is returned.

Function: `string[] componentObject.getParameterUnits()`
Code: `parUniVec = simulationObject.smf.getParameterUnits()`
Alternative Code: `parUniVec = componentObject.getParameterUnits()`
Description: Returns a string vector containing the units of all parameters of the specified component (object). If one of the parameters does not contain a unit, the associated vector entry will be empty.

Function: `double /string componentObject.getParameterValue(string parameterName)`
Code: `length = mySimObject.smf.getParameterValue("length")`
Alternative Code: `length = componentObject.getParameterValue("length")`
Description: Returns a double or string value of the specified parameter for a given component (object

Function: `double[] /string[] componentObject.getParameterValues()`
Code: `parVals = mySimObject.smf.getParameterValues()`
Alternative Code: `parVales = componentObject.getParameterValues()`
Description: Returns the double or string vector with the values of all parameters for a given component (object).

Function: `bool getPMDPath()`
Code: `smfIsPMDMember = mySimObject.smf.getPMDPath()`
Description: Returns a bool value which specifies if the currently selected component is a PMD Path Member (`true`) or not (`false`). The Default is `false`.

Function: `resultObject componentObject.getResult(string resultName)`
Code: `resObj = mySimObject.smf.getResult("DGD")`
Alternative Code: `resObj = componentObject.getResult("DGD")`
Description: Returns a result object of the specified result of the given component (object). Please note, that the result object contains the name, value and unit of the specified result. If you want to access the result value, please use `resObj.getValue()` or the function `.getResultValue()` if you want to directly access the value.

Function: `string[] componentObject.getResultNames()`
Code: `resNamesVec = mySimObject.smf.getResultNames()`
Alternative Code: `resNamesVec = componentObject.getResultNames()`
Description: Returns a string vector containing the names of all results of the specified component (object).

Function: `resultObject[] componentObject.getResults()`
Code: `resObjVec = mySimObject.smf.getResults()`
Alternative Code: `resObjVec = componentObject.getResults()`
Description: Returns a vector containing the result objects all results of the specified component (object). Please note, that the elements of the result object vector contains the result objects (including their name, value and unit). If you want to access the result values, please use `resObjVec[i].getValue()` or the function `.getResultValues()` if you want to directly access the values

Function: `string componentObject.getResultUnit(string resultName)`
Code: `resUnit = mySimObject.smf.getResultUnit("DGD")`
Alternative Code: `resUnit = componentObject.getResultUnit("DGD")`
Description: Returns a string containing the unit of the specified result of the given component (object). If the result does not have a unit, an empty string is returned.

Function: `string[] componentObject.getResultUnits()`
Code: `resUnitVec = mySimObject.smf.getResultUnits()`
Alternative Code: `resUnitVec = componentObject.getResultUnits()`
Description: Returns a string vector containing the units of all results of the given component (object). If one of the results does not contain a unit, the associated vector entry will be empty.

Function: `double componentObject.getResultValue(string resultName)`
Code: `resVal = mySimObject.smf.getResultValue("DGD")`
Alternative Code: `resVal = componentObject.getResultValue("DGD")`
Description: Returns the value of the specified result of the given component (object).

Function: `double[] componentObject.getResultValues()`
Code: `resValVec = mySimObject.smf.getResultValues()`
Alternative Code: `resValVec = componentObject.getResultValues()`
Description: Returns the a vector containing the values of all results of the given component (object).

Function: `bool componentObject.isComponentOfType(string componentTypeName)`
Code: `myBool = mySimulationObject.myComponent.isComponentOfType("Pulse Generator")`
Alternative Code: `myBool = myComponentObject.isComponentOfType("Pulse Generator")`
Description: Returns, whether the given component is of the type which has been specified in the `componentTypeName string` (`true`) or not (`false`).

Function: `bool componentObject.isIterator()`
Code: `compIsIterator = myCompObject.isIterator()`
Alternative Code: `compIsIterator = mySimObject.myParamObject.isIterator()`
Description: Returns a bool value that specifies if the given component (object) is of the type iterator.

Function: `bool componentObject.isNetwork()`
Code: `compIsNetwork = myCompObject.isNetwork()`
Alternative Code: `compIsNetwork = mySimObject.myParamObject.isNetwork()`
Description: Returns a bool value that specifies if the given component (object) is of the type network.

Function: `componentObject.listParameters()`
Code: `myComponentObject.listParameters()`
Description: Displays an overview of all parameters of the given component (object) in the PScript console. The information includes the parameter names, values and units (if they exist).

Function: `componentObject.listResults()`
Code: `myComponentObject.listResults()`
Description: Displays an overview of all results of the given component (object) in the PScript console. The information includes the result names, values and units (if they exist).

Function: `componentObject.setBypass(bool bypassBool)`
Code: `myComponentObject.setBypass(true)`
Description: Sets the bypass property of the given component. If set to `true`, the component will be bypassed. Keep in mind that some PHOTOSS components cannot be bypassed (e.g. the pulse generator). If you attempt to bypass such a component, PScript will throw an exception.

Function: `componentObject.setName(string componentNameString)`
Code: `myComponentObject.setName("PulseGenerator75")`
Description: Sets the name of the component object on which the function is invoked to the given name string. If the name you have specified is already in use by other components on the PHOTOSS grid, the component you have selected will receive a warning and a suffix will be added to the component name to ensure it is unique. The suffix will contain the first available number. E.g. if you already have a component named "PulseGenerator" and invoke `myPG.setName("PulseGenrator")`, the name "PulseGenerator0" will be set. If the name "PulseGenerator0" is also already in use, the number at the end of the name string will be increased until a unique name has been found.

Function: `componentObject.setParameterValue(string parName, double parValue)`
Code: `smf.setParameterValue("length", 100)`
Description: Assign the value `parValue` to the parameter with the name "parName" of the given component (object). The given value may be a bool, a string, a double or an integer value. If you try to assign a value to a parameter which doesn't exist, PScript will throw an exception. Setting a parameter which is not accepted by the component (for example, setting a negative length for the length of a SSMF) will trigger the same response a component will give you when you try to enter the parameter value in the GUI. In most cases, the user-specified value will be rejected and the parameter will be reset to the last valid value.

Function: `componentObject.setPMDPath(bool pathmember)`
Code: `smf.setPMDPath(true)`
Description: Sets the PMD Path Membership of the given component object. If set to `true`, the component is a PMD Path Member. Keep in mind that some PHOTOSS components cannot be PMD Path members (e.g. the Analytical BERT). If you attempt to assign a PMD Path Membership to such a component, PScript will throw a warning.

16.7 Functions concerning the simulation parameter class:

In this subsection, all functions concerning the simulation parameter class are listed (see table 5). Please note that since release version 5.90 of PHOTOSS the simulation parameters cover the full functionality of the legacy construct "global variables" which is discontinued (refer to section 16.9 on page 55 for more information). Some of the functions described in table 5 can only be called on a simulation parameter object. You can obtain it by using `mySimParam = sim.getSimulationParameter()`.

PHOTOSS distinguishes between the built-in simulation parameters (which can also be accessed via their GUI dialog) and custom simulation parameters which may be defined by the user. Both types of simulation parameters are combined in the (general) "simulation parameters". However, dedicated access of either the built-in or custom parameters is possible (see below).

The built-in simulation parameters may not be deleted but they may be changed (according to their allowed parameter range) and may be used in a parameter variation. Custom simulation parameters may be created, deleted and modified to your liking. For a more detailed explanation on the inner workings of the PHOTOSS simulation parameters, parameter variations and the usage of automatically generated PScripts please refer to the PHOTOSS manual!

Please keep in mind, that a simulation parameter always has a *current* value. If it is part of a parameter variation, the *current* value successively assumes all values inside the variation vector.

Table 6. Overview of Simulation Parameter-Related Functions

Function Name	**Short description**	**Page**
`simParam.getValue()`	returns the *current* value of the sim param object	49
`simParam.setValue()`	returns the *current* value of the sim param object	49
`simParam.getUnit()`	returns the unit of the sim param object	49
`simParam.setUnit()`	sets the unit of the sim param object	50
`simParam.getName()`	returns the name of the sim param	50
`simParam.setName()`	sets the name of the custom sim param	50
`simParam.isVariation()`	returns, whether the sim param object is part of a parameter variation	50
`simParam.isBuiltIn()`	returns, whether the sim param object is a built-in sim param	50
`simParam.getVariationVector()`	returns the variation vector of the sim param object	50
`simParam.setVariationVector()`	sets the variation vector of the sim param object	50

Table 7. Overview of Simulation Parameter-Related Functions (continued)

Function Name	Short description	Page
sim.listSimulationParameters()	lists all sim params (custom, built-in and derived)	50
sim.listBuiltInSimulationParameters()	lists all built-in sim params	50
sim.listCustomSimulationParameters()	lists all custom sim params	50
sim.listDerivedSimulationParameters()	lists all derived sim params	51
sim.createSimulationParameter()	create a custom sim param	51
sim.deleteSimulationParameter()	deletes a custom sim param	51
sim.getSimulationParameter()	returns a sim param object for a (general) sim param	51
sim.getSimulationParameters()	returns a sim param object vector of all (general) sim params	51
sim.setSimulationParameterName()	resets the name of the specified sim param	51
sim.getSimulationParameterValue()	returns the *current* value of the given sim param	51
sim.setSimulationParameterValue()	sets the *current* value of the given sim param	51
sim.getSimulationParameterValues()	returns all *current* values of all sim param	51
sim.getSimulationParameterUnit()	returns the unit of the given sim param	51
sim.setSimulationParameterUnit()	sets the unit of the given custom sim param	52
sim.getSimulationParameterIsVariation()	returns whether the sim param is part of a parameter variation	52
sim.getSimulationParameterVariationVector()	returns the variation vector of the given sim param	52
sim.setSimulationParameterVariationVector()	sets the variation vector of the given sim param	52
sim.getParameterVariationScript()	returns parameter variation script of the simulation	52
sim.getBuiltInSimulationParameter()	returns a sim param object for a (built-in) sim param	52
sim.getBuiltInSimulationParameters()	returns a sim paramr object vector of all (built in) sim params	52
sim.getBuiltInSimulationParameterValue()	returns the value for a (built-in) sim param	52
sim.getBuiltInSimulationParameterUnit()	returns the unit for a (built-in) sim param	52
sim.setBuiltInSimulationParameterValue()	sets the value for a (built-in) sim param	53
sim.getCustomSimulationParameter()	returns a sim param object for a (custom) sim param	53
sim.getCustomSimulationParameters()	returns a sim param object vector of all (custom) sim params	53
sim.setCustomSimulationParameterName()	resets the name of the specified custom sim param	53
sim.getCustomSimulationParameterValue()	returns the value for a (custom) sim param	53
sim.getCustomSimulationParameterUnit()	returns the unit for a (custom) sim param	53
sim.setCustomSimulationParameterValue()	sets the value for a (custom) sim param	53
sim.setCustomSimulationParameterUnit()	sets the unit for a (custom) sim param	53
sim.getDerivedSimulationParameterValue()	gets the value for a (derived) sim param	53
sim.getDerivedSimulationParameterUnit()	gets the unit for a (derived) sim param	53

Function: `simulationParameterObject.getValue()`
Code: `mySimParamObject.getValue()`
Description: Identical to `getSimulationParameterValue()` but must be called directly on the simulation parameter object (may be both a custom or built-in simulation parameter).

Function: `simulationParameterObject.setValue(double paramValue)`
Code: `mySimParamObject.setValue(5)`
Description: Identical to `setSimulationParameterValue()` but must be called directly on the simulation parameter object (may be both a custom or built-in simulation parameter).

Function: `simulationParameterObject.getUnit`
Code: `mySimParamObject.getUnit()`
Description: Identical to `getSimulationParameterUnit()` but must be called directly on the simulation parameter object (may be both a custom or built-in simulation parameter).

Function: `simulationParameterObject.setUnit(string unitNameString)`
Code: `mySimParamObject.setUnit("THz")`
Description: Identical to `setSimulationParameterUnit()` but must be called directly on the simulation parameter object (may only be a custom simulation parameter).

Function: `simulationParameterObject.getName()`
Code: `mySimParamObject.getName()`
Description: Returns the name of the simulation parameter object (may be both a custom or built-in simulation parameter).

Function: `simulationParameterObject.setName(string nameString)`
Code: `mySimParamObject.setName("myParamName")`
Description: Sets the name of the simulation parameter object (may only be a custom simulation parameter). The name must not be occupied by an already existing simulation parameter, otherwise, an error will be issued. The name must not contain white spaces. Keep in mind that changing the name of a custom simulation parameter also changes the names of all placeholders you might have put into any component as parameter. If, e.g. you have a custom simulation parameter called "SMFlength", you can also assign it to any parameter of any component, e.g. the parameter "length" of the component "SMF1". Now, changing the name from "SMFlength" to (e.g.) "myLength" will cause the placeholder to assume the name "myLength" as well.

Function: `bool simulationParameterObject.isVariation()`
Code: `myBool = mySimParamObject.isVariation()`
Description: Returns if the simulation parameter object (may be both a custom or built-in simulation parameter) is part of a parameter variation. Works similar to `simulationParameterIsVariation()` but must be called directly on a simulation parameter object.

Function: `bool simulationParameterObject.isBuiltIn()`
Code: `myBool = mySimParamObject.isBuiltIn()`
Description: Returns if the simulation parameter object (may be both a custom or built-in simulation parameter) is a built in simulation parameter. Returns `false` otherwise.

Function: `double[] simulationParameterObject.getVariationVector()`
Code: `myVariationVector = mySimParamObject.getVariationVector()`
Description: Identical to `getSimulationParameterVariationVector()` but must be called directly on a simulation parameter object.

Function: `simulationParameterObject.setVariationVector(double[] variationVector)`
Code: `mySimParamObject.setVariationVector([1, 5, -4, 5.344])`
Description: Identical to `setSimulationParameterVariationVector()` but must be called directly on a simulation parameter object.

Function: `simulationObject.listSimulationParameters()`
Code: `sim.listSimulationParameters()`
Description: Lists all simulation parameters (custom, built-in and derived) in the PScript console. Their names, units and current values are depicted.

Function: `simulationObject.listBuiltInSimulationParameters()`
Code: `sim.listBuiltInSimulationParameters()`
Description: Lists all built-in simulation parameters in the PScript console. Their names, units and current values are depicted.

Function: `simulationObject.listCustomSimulationParameters()`
Code: `sim.listCustomSimulationParameters()`
Description: Lists all custom simulation parameters in the PScript console. Their names, units and current values are depicted. If no custom simulation parameters are defined, no output will be written to the console window.

Function: `simulationObject.listDerivedSimulationParameters()`
Code: `sim.listDerivedSimulationParameters()`
Description: Lists all derived simulation parameters in the PScript console. Their names, units and current values are depicted.

Function: `simulationParameterObject simulationObject.createSimulationParameter(string parameterName, double currentValue)`
Code: `mySimParam = sim.createSimulationParameter("myParam", 5)`
Description: Creates and returns a new simulation parameter with designated name and value. You can not create simulation parameters which already exist. Trying to do so will issue an error. Built-in simulation parameters may not be (re-)created, either. The specified name string must *not* contain white spaces.

Function: `simulationObject.deleteSimulationParameter(string parameterName)`
Code: `sim.deleteSimulationParameter("mySimParam")`
Description: Deletes a custom simulation parameter (if it exists). Trying to delete built-in or non-existent simulation parameters will issue an error.

Function: `simulationParameterObject simulationObject.getSimulationParameter(string parameterName)`
Code: `myParamObject = sim.getSimulationParameter("testParameter")`
Description: Returns the simulation parameter object for the specified (custom and/or built-in) simulation parameter. Specific functions such as (e.g.) `getValue()` and `setValue()` can now be called on the simulation parameter object. Trying to call for a non-existing (general) simulation parameter will cause an error. Please not that this function can be used to call for both built-in *and* custom simulation parameters!
Function: `simulationParameterObjectVector simulationObject.getSimulationParameters()`
Code: `myParamObjectVector = sim.getSimulationParameters()`
Description: Returns a vector of all simulation parameter objects for the (general) simulation parameters. Specific functions such as (e.g.) `getValue()` and `setValue()` can now be called each object of the vector.

Function: `simulationObject.setSimulationParameterName(string oldParamName, string newParamName)`
Code: `sim.setSimulationParameterName("myOldParamName", "myNiceNewName")`
Description: Resets the value for the specified simulation parameter. Trying to reset a non-custom simulation parameter or trying to assign a name which is already in use will issue an error. If the simulation parameter you want to rename is currently used in a formula and/or as placeholder in a component, *all* occurrences of the parameter will automatically be renamed as well.

Function: `double simulationObject.getSimulationParameterValue(string simParameterName)`
Code: `parVal = mySimObject.getSimulationParameterValue("f0")`
Description: Returns the value of the specified simulation parameter (may be custom, derived or built-in) of the given simulation.

Function: `simulationObject.setSimulationParameterValue(string parName, parValue)`
Code: `mySimulation.setSimulationParameterValue("ref_bitrate", 40.0)`
Description: Sets the specified simulation parameter (may be both custom or built-in) with the name "parName" to the given *current* value "parValue" of the given simulation. If the simulation parameter with the specified name does not exist, an exception will be thrown. If the specified value is not valid, a warning will be shown and PHOTOSS will use the last valid value for the simulation parameter instead.

Function: `double[] simulationObject.getSimulationParameterValues()`
Code: `simParValVec = mySimObject.getSimulationParameterValues()`
Description: Returns a vector containing the values of all simulation parameters (both custom and built-in) in the given simulation.

Function: `string simulationObject.getSimulationParameterUnit(string simParameterName)`
Code: `parUnit = mySimObject.getSimulationParameterUnit("f0")`
Description: Returns a string containing the unit of the specified simulation parameter (may be custom, derived or built-in) of the given simulation. If the parameter does not have a unit, an empty string is returned.

Function: `simulationObject.setSimulationParameterUnit(string parameterNameString, string unitNameString)`
Code: `mySimObject.setSimulationParameterUnit("myFrequencyParameter", "Thz")`
Description: Sets a string defining the unit of the specified simulation parameter of the given simulation. Works only for custom simulation parameters. Trying to change the unit of a derived or built-in simulation parameter will issue an error.

Function: `bool simulationObject.getSimulationParameterIsVariation(string parameterNameString)`
Code: `myBool = sim.getSimulationParameterIsVariation("f0")`
Description: Returns, whether the given simulation parameter (may be both custom or built-in parameter) is part of a parameter variation. Returns `true` if the variation vector of the specified parameter contains more than one element.

Function: `variationVector[] simulationObject.getSimulationParameterVariationVector(string paramName)`
Code: `myVec = sim.getSimulationParameterVariationVector("testParam")`
Description: Returns the variation vector of the specified simulation parameter (may be both a custom or built-in simulation parameter). If the parameter is not part of a parameter variation, the vector will only contain one element. If the parameter *is* part of a parameter variation, the vector will contain *all* values, the parameter will assume during a variation.

Function: `simulationObject.setSimulationParameterVariationVector(string paramName, double[] variationVec)`
Code: `sim.getSimulationParameterVariationVector("testParam", [1,2,5.5, 3.8461, -1 , 5])`
Description: Sets the variation vector of the specified simulation parameter (may be both a custom or built-in simulation parameter). If a vector with more than one element is supplied, the parameter will automatically count as "is part of a parameter variation" and it will assume all values of the vector you have supplied.

Function: `string simulationObject.getParameterVariationScript()`
Code: `myScriptString = sim.getParameterVariationScript()`
Code: `eval(sim.getParameterVariationScript()`
Description: Returns the parameter variation script of the current simulation. You can execute the whole parameter variation by using the `eval(yourScript)` function but keep in mind that the proper way to carry out parameter variations is to either use the GUI mode or to copy (and use) the variation script directly from the GUI.

Function: `simulationParameterObject simulationObject.getBuiltInSimulationParameter(string parameterName)`
Code: `myParamObject = sim.getBuiltInSimulationParameter("f0")`
Description: Returns the simulation parameter object for the specified (built-in) simulation parameter. Specific functions such as (e.g.) `getValue()` and `setValue()` can now be called on the simulation parameter object. Trying to call for a non-existing (built-in) simulation parameter will cause an error. Please not that this function can not be used to call for custom simulation parameters! Use `getSimulationParameter()` or `getCustomSimulationParameter()` instead!

Function: `simulationParameterObject[] simulationObject.getBuiltInSimulationParameters()`
Code: `myParamObjectVector = sim.getBuiltInSimulationParameters()`
Description: Returns a vector of all simulation parameter objects for the (built-in) simulation parameters. Specific functions such as (e.g.) `getValue()` and `setValue()` can now be called each object of the vector.

Function: `double simulationObject.getBuiltInSimulationParameterValue(string paramName)`
Code: `simParVal = mySimObject.getBuiltInSimulationParameterValue("reference_bitrate")`
Description: Returns the value of the specified built-in simulation parameter (may be a double, boolean or string value).

Function: `double simulationObject.getBuiltInSimulationParameterUnit(string paramName)`
Code: `simParVal = mySimObject.getBuiltInSimulationParameterUnit("reference_bitrate")`
Description: Returns the unit of the specified built-in simulation parameter. If the speciefied built-in simulation parameter does not have a unit, an empty string is returned.

Function: `simulationObject.setBuiltInSimulationParameterValue(string paramName, double paramValue)`
Code: `mySimObject.setBuiltInSimulationParameterValue("reference_bitrate", 112)`
Description: Sets the value of the specified built-in simulation parameter (may be a double, boolean or string value). If you are trying to set a value which is outside the allowed parameter range of the specified built-in simulation parameter, PHOTOSS will automatically trigger an error (default) or a warning, depending on your choices in the PHOTOSS options dialog. It is strongly recommended to use the default setting and receive an error.

Function: `simulationParameterObject simulationObject.getCustomSimulationParameter(string parameterName)`
Code: `myParamObject = sim.getCustomSimulationParameter("myParameterName")`
Description: Returns the simulation parameter object for the specified (custom) simulation parameter. Specific functions such as (e.g.) `getValue()` and `setValue()` can now be called on the simulation parameter object. Trying to call for a non-existing (custom) simulation parameter will cause an error. Please not that this function can not be used to call for built-in simulation parameters! Use `getSimulationParameter()` or `getBuiltInSimulationParameter()` instead!

Function: `simulationParameterObjectVector simulationObject.getCustomSimulationParameters()`
Code: `myParamObjectVector = sim.getSimulationCustomParameters()`
Description: Returns a vector of all simulation parameter objects for the (custom) simulation parameters. Specific functions such as (e.g.) `getValue()` and `setValue()` can now be called each object of the vector. If no custom simulation parameters exist, an empty vector will be returned.

Function: `simulationObject.setCustomSimulationParameterName(string oldParamName, string newParamName)`
Code: `sim.setCustomSimulationParameterName("myOldParamName", "myNiceNewName")`
Description: Resets the value for the specified custom simulation parameter. Trying to reset a non-custom simulation parameter or trying to assign a name which is already in use will issue an error. If the simulation parameter you want to rename is currently used in a formula and/or as placeholder in a component, *all* occurrences of the parameter will automatically be renamed as well.

Function: `double simulationObject.getCustomSimulationParameterValue(string paramName)`
Code: `simParVal = mySimObject.getCustomSimulationParameterValue("myParam")`
Description: Returns the value of the specified custom simulation parameter (may be a double, boolean or string value).

Function: `string simulationObject.getCustomSimulationParameterUnit(string paramName)`
Code: `simParUnit = mySimObject.getCustomSimulationParameterUnit("myParam")`
Description: Returns the unit of the specified custom simulation parameter. If the speciefied custom simulation parameter does not have a unit, an empty string is returned.

Function: `simulationObject.setCustomSimulationParameterValue(string paramName, double paramValue )`
Code: `mySimObject.setCustomSimulationParameterValue("myParam", 5)`
Description: Sets the value of the specified custom simulation parameter.

Function: `simulationObject.setCustomSimulationParameterUnit(string paramName, string paramUnit)`
Code: `mySimObject.setCustomSimulationParameterUnit("myParam", "GHz")`
Description: Sets the unit of the specified custom simulation parameter.

Function: `double simulationObject.getDerivedSimulationParameterValue(string paramName)`
Code: `simParVal = mySimObject.getDerivedSimulationParameterValue("frequencyRange")`
Description: Returns the value of the specified derived simulation parameter (may be a double, boolean or string value).

Function: `string simulationObject.getDerivedSimulationParameterUnit(string paramName)`
Code: `simParUnit = mySimObject.getDerivedSimulationParameterUnit("myParam")`
Description: Returns the unit of the specified derived simulation parameter. If the speciefied derived simulation parameter does not have a unit, an empty string is returned.

16.8 Functions concerning the result class:

The functions which may be called on a result object are listed in table 8. To obtain a result object, you may use `getResult()` on a component object. All functions are explained in more detail below. Obviously, results are read-only; their units can not be changed either.

Table 8. Overview of Result-Related Functions

Function Name	Short description	Page
`component.getResult()`	returns the result object of the given result name	46
`result.getValue()`	sets the unit of the simulation parameter object	54
`result.getUnit()`	returns the name of the simulation parameter	54

Function: `double resultObject.getValue()`
Code: `myDouble = resObj.getValue()`
Description: Identical to `getResultValue()` but must be called directly on a result object.

Function: `string resultObjekt.getUnit()`
Code: `myUnitName = resObj.getUnit()`
Description: Identical to `getResultUnit()` but must be called directly on a result object.

16.9 Functions concerning the global variables:

Since the PHOTOSS release version 5.90 “Global Variables” are no longer required. All the functionality which was previously offered by using global variables is now handled by (custom) simulation parameters. Please refer to section 16.7 on page 48 for all functions concerning simulation parameters. When opening old `.pho` files, the old global variables will automatically be converted to (custom) simulation parameters.

The functions listed in table 9 are no longer supported and have been replaced by their simulation parameter counterparts. If you still have old scripts which call these functions, their new counterparts will automatically be called and a warning will be issued to inform you that the usage of the obsolete functions is discouraged.

Table 9. Overview of Obsolete Functions

Obsolete Function Name	New Function Pendant	Page
`sim.getGlobalVariable()`	`sim.getSimulationParameter()`	51
`sim.getGlobalVariables()`	`sim.getSimulationParameters()`	51
`sim.setGlobalVariableValue()`	`sim.setSimulationParameterValue()`	51
`sim.getGlobalVariableValue()`	`sim.getSimulationParameterValue()`	51
`sim.getGlobalVariableUnit()`	`sim.getSimulationParameterUnit()`	51
`sim.listGlobalVariables()`	`sim.listSimulationParameters()`	50
`sim.isGlobalVariableOfTypeVariation()`	`sim.getSimulationParameterIsVariation()`	52

16.10 Functions concerning linking and unlinking of components

Table 10. Overview of Functions concerning Linking / Unlinking

Function Name	Short description	Page
`sim.linkComponents()`	links the specified components	56
`comp.getInPortLinks()`	returns all components, which are connected to the specified In-Port	56
`comp.getOutPortLinks()`	returns all components, which are connected to the specified Out-Port	56
`sim.isInPortLinked()`	returns, whether the specified In-Port is linked	56
`sim.isOutPortLinked()`	returns, whether the specified Out-Port is linked	57
`comp.isLinked()`	returns, whether the component is linked	57
`comp.isLinkedTo()`	returns, whether the component is linked to another component	57
`comp.getLinkPorts()`	returns the In- and Out-Port through which two components are linked	57
`sim.unlinkComponents()`	unlink the specified components	57

Functions: `simulationObject.linkComponents(leftCompObj,int outPortNo ,rightCompObj, int inPortNo)`
`simulationObject.linkComponents(leftCompObj,rightCompObj)`
Code: `s.linkComponents(s.pulseGenerator, 0, s.coupler, 1))`
Description: You link together two components in PScript using this function. `leftCompObj` is the component object on the left hand side. You can optionally specify the number of its Out-Port and the number of the In-Port of the component object on the right hand side (`rightCompObj`). Keep in mind that the In-Ports and the Out-Ports are counted **0-based** (beginning with 0 for the first port). If you omit the In-Port and Out-Port numbers, PScript will connect the *first* Out-Port of the left hand side component object to the first *free* In-Port of the right hand side component object. If the component on the right side has no free In-Ports or no In-Ports at all, PScriptwill issue an error. PScript will generally issue an error if you try to access non-existent ports or if you try to connect port types which are not compatible (e.g. an electrical to an optical port). Keep in mind, that PHOTOSS components generally may have only *one* component connected to their In-Port but it is *always* possible to connect multiple components to a single Out-Port. Please also keep in mind that you can only link components which reside on the same network-level (e.g. "on top" or in the same sub-network or iterator).

Function: `componentObject.getInPortLinks(int portNumber)`
Code: `compVector = s.smf.getInPortLinks(0))`
Description: Returns a vector which holds all components which are connected to the specified In-Port of the component on which this function is called. Normally, the vector should not return more than one component, since In-Ports may only have one connection. If you specify a non-existing In-Port number (remember that the In-Ports are counted zero-based), you will receive an error. If the specified In-Port is currently not connected, you will receive an empty vector of length 0.

Function: `componentObject.getOutPortLinks(int portNumber)`
Code: `compVector = s.smf.getOutPortLinks(0))`
Description: Returns a vector which holds all components which are connected to the specified Out-Port of the component on which this function is called. If you specify a non-existing Out-Port number (remember that the Out-Ports are counted zero-based), you will receive an error. If the specified Out-Port is currently not connected, you will receive an empty vector of length 0.

Functions: `simulationObject.isInPortLinked(compObj,int inPortNo)`
`componentObject.isInPortLinked(int inPortNo)`
Code: `s.isInPortLinked(s.smf, 0))`
Code: `s.smf.isInPortLinked(0))`
Description: This function returns, whether the specified In-Port of the current (or specified) component is currently linked. The function may either be called on a component object or a simulation object.

If you try to address a port which does not exist, you will receive an error. Remember that ports are counted zero-based.

Functions: `simulationObject.isOutPortLinked(compObj,int outPortNo)`
`componentObject.isOutPortLinked(int outPortNo)`
Code: `s.isOutPortLinked(s.smf, 0))`
Code: `s.smf.isOutPortLinked(0))`
Description: This function returns, whether the specified Out-Port of the current (or specified) component is currently linked. The function may either be called on a component object or a simulation object. If you try to address a port which does not exist, you will receive an error. Remember that ports are counted zero-based.

Function: `componentObject.isLinked()`
Code: `s.smf.isLinked())`
Description: This function returns, whether the specified component is linked. The link may be connected to an In-Port, an Out-Port or even both.

Function: `componentObject.isLinkedTo(compObj otherComponent)`
Code: `s.smf.isLinkedTo(s.pulseGenerator))`
Description: This function returns, whether the current component is linked to the specified component. If the function returns `true`, the components may be connected either from one of the source component's Out-Port to one of the destination component's In-Port or from one of the destination component's Out-Ports to one of the the source component's In-Ports. Trying to address non-existing components will issue an error.

Functions: `simulationObject.getLinkPorts(compObj firstComp,compObj secondComp)`
`componentObject.getLinkPorts(compObj secondComp`
Code: `s.getLinkPorts(s.smf, s.analyticalFilter))`
Code: `s.smf.getLinkPorts(s.analyticalFilter))`
Description: If the two specified components are linked together in such a way, that (at least) one of the first component's Out-Ports is connected to (at least) one of the second component's In-Ports (*not vice versa!*) you will receive an array containing the designated Out-Port of the first component (entry 0) and the designated In-Port of the second component (entry 1). If the specified components are not linked, you will receive an empty array (length 0). If the specified components are linked through multiple ports, the array will contain the ports for the *first* link.

Function: `simulationObject.unlinkComponents(componentObject left, int outPortLeft, componentObject right, int inPortRight)`
Code: `s.unlinkComponents(s.SMF, 1, s.Coupler, 2)`
`s.unlinkComponents(s.SMF, s.Coupler)`
Description: Removes a link between two PHOTOSS components. Trying to remove a link of components which are not linked together will produce a warning. If you do not specify the Outport and the Inport of the components, the *first* link between the two components which exists will be removed. Please note that some components may be connected through more than one link - only the first one will be removed!

16.11 PScript MATLAB® specific functions

A list of a all MATLAB® specific functions of PScript is shown in table 11. All functions are exlpained in more detail below. Please ensure that you have properly installed a compatible version of MATLAB® (you can verify this by using `matlab.isAvailable()` before calling these functions.

Table 11. Overview of MATLAB®-related Functions

Function Name	Short description	Page
`object.toMatlab()`	transfers the object to the MATLAB® workspace	58
`matlab.getBreakOnMatlabErrors()`	returns whether PScript should halt on MATLAB® errors	58
`matlab.setBreakOnMatlabErrors()`	sets whether PScript should halt on MATLAB® errors	58
`matlab.getCheckForMatlabErrors()`	returns whether PScript should look for MATLAB® errors	58
`matlab.setCheckForMatlabErrors()`	sets whether PScript should look for MATLAB® errors	58
`matlab.eval()`	evaluate the given expression using MATLAB®	59
`matlab.getLastError()`	returns the last error in the MATLAB® workspace	59
`matlab.getLastWarning()`	returns the last warning in the MATLAB® workspace	59
`matlab.getMatlabTimeoutInSeconds()`	returns the MATLAB® timeout in seconds	59
`matlab.setMatlabTimeoutInSeconds()`	sets the MATLAB® timeout in seconds	59
`matlab.getValue()`	retrieves a value from the MATLAB® workspace to the PScript workspace	59
`matlab.setValue()`	copies a value from the PScript workspace to the MATLAB® workspace	59
`matlab.isAvailable()`	checks whether MATLAB® is available	59
`matlab.getUseIdenticalMatlab WorkspaceForPHOTOSSAndPScript()`	returns whether PHOTOSS MATLAB® and PScript workspaces are treated as separate	59
`matlab.setUseIdenticalMatlab WorkspaceForPHOTOSSAndPScript()`	sets whether PHOTOSS MATLAB® and PScript workspaces are treated as separate	59
`matlab.who()`	lists all objects contained in the `matlab` object	33

Function: `.toMatlab(string variableNameString)`
Code: `s.SMF.getResult("All lines, Q").toMatlab("myQResult")`
Description: Directly transfer a PHOTOSS result to the MATLAB® workspace. The variable will be named after the given `variableNameString` in the function call. If you do not specify a name string, the variable will have the same name as the PHOTOSS result. Be advised that not specifying a unique result name string may cause older MATLAB® variables to be overwritten!

Functions: `matlab.setBreakOnMatlabErrors(bool breakBool)`
`bool matlab.getBreakOnMatlabErrors()`
Code set: `matlab.setBreakOnMatlabErrors(true)`
Code get: `myBool = matlab.getBreakOnMatlabErrors()`
Description: If set to `true`, errors which occur during the execution of a MATLAB® script will result in a PScript error which will abort the currently running PScript. If set to `false`, the currently running MATLAB® script will be aborted but the currently running PScript will be resumed. Disabling this option is *not* recommended.

Functions: `matlab.setCheckForMatlabErrors(bool checkBool)`
`bool matlab.getCheckForMatlabErrors()`
Code set: `matlab.setCheckForMatlabErrors(true)`
Code get: `myBool = matlab.getCheckForMatlabErrors()`
Description: If set to `true`, errors and warnings which occur during the execution of a MATLAB® script will also automatically be shown in the PScript console. If set to `false`, the errors will *not* be shown. This means, that a running PScript will *never* be aborted even if MATLAB® errors will occur. Disabling this option is *not* recommended.

Function: `matlab.eval(string command)`
Code: `matlab.eval("alpha = 10.334")`
Description: Sends a string command to the MATLAB® workspace and evaluates it. Any valid code in MATLAB® can be sent. This includes the execution of functions, scripts, etc. If no PScript MATLAB® workspace is currently opened, PScript will open a new workspace (this may take a few seconds, depending on your machine, so please be patient).

Function: `string matlab.getLastError()`
Code: `mystring = matlab.getLastError()`
Description: Copies the last error which occurred in the MATLAB® workspace to a string variable. If no error has occurred, it will return an empty string.

Function: `string matlab.getLastWarning()`
Code: `mystring = matlab.getLastWarning()`
Description: Copies the last warning which occurred in the MATLAB® workspace to a string. If no warning has occurred, it will return an empty string.

Functions: `matlab.setMatlabTimeoutInSeconds(int setInt)`
`int matlab.getMatlabTimeoutInSeconds()`
Code set: `matlab.setMatlabTimeoutInSeconds(600)`
Code get: `timeOut = matlab.getMatlabTimeoutInSeconds()`
Description: Sets the Matlab Timeout in seconds to the specified integer value. Please refer to the PScript Settings chapter for more details.

Function: `double matlab.getValue(string valName, matlab. dataType)`
Code: `mydouble = matlab.getValue("alpha", matlab.DOUBLE)`
Description: Transfers an already existing variable of the MATLAB® workspace to a specified PScript variable. Currently supported data types are:
`matlab.DOUBLE`, `matlab.DBLARRAY`, `matlab.DBLMATRIX`, `matlab.COMPLEX`, `matlab.CPLXMATRIX`, `matlab.STRING`, `matlab.BOOL`

Function: `bool matlab.isAvailable()`
Code: `isMatlabAvailable = matlab.isAvailable()`
Description: Returns a bool value which specifies if MATLAB® is available (`true`) or not (`false`). Please keep in mind that PScript supports the same MATLAB® versions as PHOTOSS. Make sure that your currently selected MATLAB® version is supported by both PHOTOSS and PScript.

Function: `bool setValue(string parName, parValue)`
Code: `matlab.setValue("alpha", 10.334)`
Description: Assigns the specified value with the name "parValue" (`double`, `float`, `int`, `bool`, `string`) to a variable named "parName" in the MATLAB® workspace. If the variable already exists, it will be overwritten; if not, it will be created. If no PScript MATLAB® workspace is currently opened, PScript will open a new workspace which may take a few seconds. The function will return `true` if the string has been successfully sent to MATLAB®; this return value will *not* denote whether the assignment operation in MATLAB® has been successful, however.

Functions: `bool matlab.setUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript(bool useBool)`
Code set: `matlab.setUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript(true)`
Code get: `myBool = matlab.getUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript()`
Description: Determines whether the PScript MATLAB® workspace and the PHOTOSS MATLAB® component workspace are identical. Please refer to the PScript Settings chapter for more details. The default value is `true`.

17 Basic JavaScript Mathematical Function Reference

In this chapter we will list a few of the most common mathematical functions of JavaScript. Be advised that the following list is not complete. For a complete reference please refer to [JAVASCRIPT], 18.

- Code Sample: `Math.abs(my_argument)`
  Description: Returns the absolute value of the argument.
- Code Sample: `Math.acos(my_argument)`
  Description: Returns the arcos of the input argument.
- Code Sample: `Math.asin(my_argument)`
  Description: Returns the arcsin of the input argument.
- Code Sample: `Math.atan(my_argument)`
  Description: Returns the arctan of the input argument.
- Code Sample: `Math.ceil(my_argument)`
  Description: Round argument to the next upper integer value.
- Code Sample: `Math.cos(my_argument)`
  Description: Returns the cosine of the input argument.
- Code Sample: `Math.exp(my_argument)`
  Description: Returns exp(argument).
- Code Sample: `Math.floor(my_argument)`
  Description: Round argument to the lower integer value.
- Code Sample: `Math.log(my_argument)`
  Description: Returns the natural logarithm of the argument.
- Code Sample: `Math.max(x,y)`
  Description: Returns the maximum value of [x,y]
- Code Sample: `Math.min(x,y)`
  Description: Returns the minimum value of [x,y]
- Code Sample: `Math.pow(x,y)`
  Description Returns the yth power of x.
- Code Sample: `Math.round(my_argument)`
  Description: Rounds to the next integer value
- Code Sample: `Math.sin(my_argument)`
  Description: Returns the sine of the input argument.
- Code Sample: `Math.sqrt(my_argument)`
  Description: Returns the square root of the input argument.
- Code Sample: `Math.tan(my_argument)`
  Description: Returns the tangens of the input argument.

18 External References

The following references might be useful when concerning editing, PScript, JavaScript, Qt Script, etc. These are *external* Internet references - the author of this manual is not responsible for the contents of the following websites.

- JAVASCRIPT: Core JavaScript Reference including JavaScript objects, methods, properties, statements, and many more helpful listings: http://www.webreference.com/programming/javascript/
- MATH: Core Java Script Reference concerning the Math object and its methods: http://www.webreference.com/javascript/reference/core_ref/math.html

Index