

PScript Basic Manual

By Dipl.-Phys. Matthias Westhäuser

Acknowledgments:

Nicolas Luck, Jan-Hendrik Menke, Christian Remmersmann

Release date:

October 14, 2010; PHOTOSS 5.10

Contents

1	Copyright Information	2
2	Acknowledgments	2
3	What is PScript?	3
4	Basic Functionality of PScript	5
5	What PScript cannot (yet) do	7
6	PScript Conventions	8
7	PScript Demo Walkthrough	10
8	Access to Simulations, Components, Parameters and Results (Quickstart Guide)	11
9	PScript and MATLAB®	15
10	PScript Console Main Window	16
11	PScript Console Menu Functionality	17
12	PScript Settings Dialog	18
12.1	Console Settings	18
12.2	Editor Settings	19
12.3	Log File Settings	20
12.4	History Settings	21
12.5	MATLAB® Settings	22
12.6	Include Settings	23
12.7	Auto Include Settings	23
13	PScript Syntax Highlighting and Code Assistant	25
14	PScript Code Samples	26
15	PScript Function Reference	28
15.1	PScript Options / Settings	28
15.2	General Functions	28
15.3	Generating Random Numbers in PScript / PHOTOSS Random Generator settings	29
15.4	Functions concerning the PHOTOSS application class:	30
15.5	Functions concerning the PHOTOSS/PScript simulation, component and result class:	31
15.6	PScript MATLAB® specific functions	39
16	Basic JavaScript Mathematical Function Reference	41
17	External References	42
	Index	43

1 Copyright Information

This manual was written by Dipl.-Phys. Matthias Westhäuser. It is a basic manual including a function reference for the PHOTOSS PScript language and functionality.

This manual may be copied and printed for scientific use and non-commercial use as long as no changes to the document have been made. Registered PHOTOSS clients may copy and use the document for internal purposes - this does *not* include the distribution of this document to third parties, etc. Commercial use of the document in the form of selling, reselling or republication is *prohibited without the consent of the author.*"

The author does by no means guarantee the accuracy or correctness of the information in this document. The author is also *not* responsible for any damage which might arise due to the use of the information in this manual.

Please keep in mind that MATLAB® is a product of "The Math Works™". For reasons of simplicity, MATLAB® will also be referred to as "Matlab" in this document when considering code examples. For more information please visit the external homepage of the vendor: <http://www.mathworks.com> .

Condor® is a product of the Condor Research Project at the University of Wisconsin-Madison (UW-Madison). For more information please visit the external homepage of the vendor:
<http://www.cs.wisc.edu/condor> .

2 Acknowledgments

The author would like to thank Dipl.-Inf. Nicolas Luck, Jan-Hendrik Menke and Dipl.-Phys. Christian Remmersmann for general comments and assistance with quality assurance concerning this document.

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 - Basic Functionality of PScript** on page 5 gives a short overview of what the PScript engine is capable of.
- Section **5 - What PScript cannot (yet) do** on page 7 will list what the current version of PScript cannot (yet) do. These features might be supported in upcoming versions of PHOTOSS and PScript.
- Section **6 - PScript Conventions** on page 8 will familiarize you with all important conventions and writing styles when making your own PScripts.
- Section **7 - PScript Demo Walkthrough** on page 10 will show you how easily you can make your own PScripts using a "Hello World"-example.
- Section **8 - Access to Simulations, Components, Parameters and Results (Quickstart Guide)** on page 11 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 15 will illustrate the powerful interaction between PScript, PHOTOSS and MATLAB®.
- Section **10 - PScript Console Main Window** on page 16 will familiarize you with the PScript main window and its functionality.

- Section **11 - PScript Console Menu Functionality** on page 17 gives a detailed overview of the PScript menu.
- Section **12 - PScript Settings Dialog** on page 18 describes in detail all available PScript Settings.
- Section **13 - PScript Syntax Highlighting and Code Assistant** on page 25 will introduce you to the comfortable Code Assistant and PScript Syntax Highlighting abilities.
- Section **14 - PScript Code Samples** on page 26 includes an overview of all available PScript Code Samples which are included in your PHOTOS installation.
- Section **15 - PScript Function Reference** on page 28 includes detailed descriptions for all functions of PScript.
- Section **16 - Basic JavaScript Mathematical Function Reference** on page 41 is a short introduction to the basic mathematical functions which are automatically included in PScript and JavaScript.
- Section **17 - External References** on page 42 lists a few hyperlinks to Internet sources for further information.

4 Basic Functionality of PScript

Please also refer to the section **14 - PScript Code Samples** on page **26** and **15 - PScript Function Reference** on page **28** for more details and explanations.

- PScript can be used to 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 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 global parameter / simulation parameter can be accessed and modified.
- Any formula used to calculate a global parameter / component parameter can be accessed and modified.
- 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).
- 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 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 15 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 `print` any valid `string` to both (or either) the PHOTOSS and the PScript console.
- 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 42 for details.
- You can use all typical algorithmic constructs and operators supported by Qt Scripts - this of course includes loops (`for` / `while`), `if`-statements, etc. Please see the reference [JavaScript] on page 42 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 **16** on page 41 and the reference [MATH] on page 42 for details.

5 What PScript cannot (yet) do

- **PScript cannot create, delete or link any component.** The creation or deletion of a component on the simulation grid is currently not supported. You cannot move or (re-)link any existing component on the grid by using PScript; you have to do it manually. Creation or (re-)linking of PHOTOSS components may be supported in future PScript / PHOTOSS versions.
- **PScript cannot be used to change components, component parameters or simulation parameters while the simulation is running.** PScript can be used to force PHOTOSS to "kill" a simulation (or a running PScript), though. Once the simulation has finished, you again have full access to all component parameters / simulation parameters and the simulation results.
- **Global parameters in the PHOTOSS workspace currently cannot be created (or deleted) by PScript.** You can create, change and delete any globally accessible parameter in the *PScript workspace* and assign it to any component parameter / simulation parameter / global parameter, though.
- **PScript cannot be used in combination with the command line mode or with Condor®.** . This feature may be supported in future PScript / PHOTOSS versions.

6 PScript Conventions

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 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 14, page 26.
- 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()`.

Conventions concerning PScript Objects

- 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, 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.
 - A simulation object always holds the global variables (if you have defined any).
 - A simulation object can be assigned to one or more variables.
 - A simulation object can be created by opening a 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 8 on page 11).
- 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 **PScript Quickstart Guide** on page 11, the **PScript Code Samples** described in chapter 14 on page 26 and the **PScript Function Reference** on page 28.

- 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 "F5").
- The text "Hello PScript User!" should be returned in the Console Window. If the text does not appear, check your **PScript Settings Dialog** (page 18) and choose "Restore Defaults".
- Congratulations, you have just created and executed your first PScript!
- For more complex scripts, please refer to the **PScript Quickstart Guide** on page 11.

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 26 and the **PScript Function Reference** on page 28 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:

```
1 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
```

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 PScript
  console
```

Now, accessing a component object through a simulation object can be done by using the "." operator. In many cases, it is 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" 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:

```
1 myFilterObject = simulationObject.getComponent("Analytical Filter 1") //access a
   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:

```
1 componentObject.listParameters() //print a list of all parameters of the current
   component to the PScript console
```

Accessing a single parameter in PScript works the following way:

```
1 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:

```
1 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:

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

Setting parameters works in a similar manner:

```
1 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:

```

1 parameterNames = componentObject.getParameterNames() //read all parameter names into a
  string vector
2 parameterValues = componentObject.getParameterValues() // read all parameter values into
  a string vector
3 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:

```

1 simObject.run()

```

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

```

1 componentObject.listResults()
2 // or alternatively:
3 simulationObject.componentObject.listResults()

```

Accessing a single result value is also easy:

```

1 resultValue = componentObject.getResultValue("Result Name")
2 //or alternatively:
3 resultValue = simulationObject.componentObject.getResultValue("Result Name")

```

Accessing a result unit (if available) works similarly:

```

1 resultUnit = componentObject.getResultUnit("Result Name")
2 // or alternatively:
3 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:

```

1 resultValues = simulationObject.componentObject.getResultValues() // returns a string
  vector holding all the result values of a component
2 resultNames = simulationObject.componentObject.getResultNames() // returns a string
  vector holding all the result names of a component
3 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:

```
1 myParamName = simObject.getSimulationParameterValue("Parameter Name")
```

Setting a parameter works in a similar manner:

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

Accessing Global Variables:

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

```
1 simulationObject.listGlobalVariables()
```

Reading the value of a global variable works this way:

```
1 varValue = simulationObject.getGlobalVariableValue("Variable Name")
```

Reading a global variable unit (if it exists) is also easy:

```
1 varUnit = simulationObject.getGlobalVariableUnit("Variable Name")
```

Setting a global variable value can be done using:

```
1 simulationObject.setGlobalVariableValue("Variable Name", variableValue)
```

Congratulations! You have already mastered all basic skills necessary to create PScripts. For more details please refer to the **PScript Code Samples** in chapter 14, page 26, or the **PScript Function Reference** in chapter 15, on page 28.

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 26 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:

```
1 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:

```
1 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:

```
1 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 18 for more information.
- PScript can create a common, single MATLAB® workspace for both the PScript MATLAB® environment and the PHOTOSS MATLAB® Component environment (this is the default). 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 18 for more information.

10 PScript Console Main Window

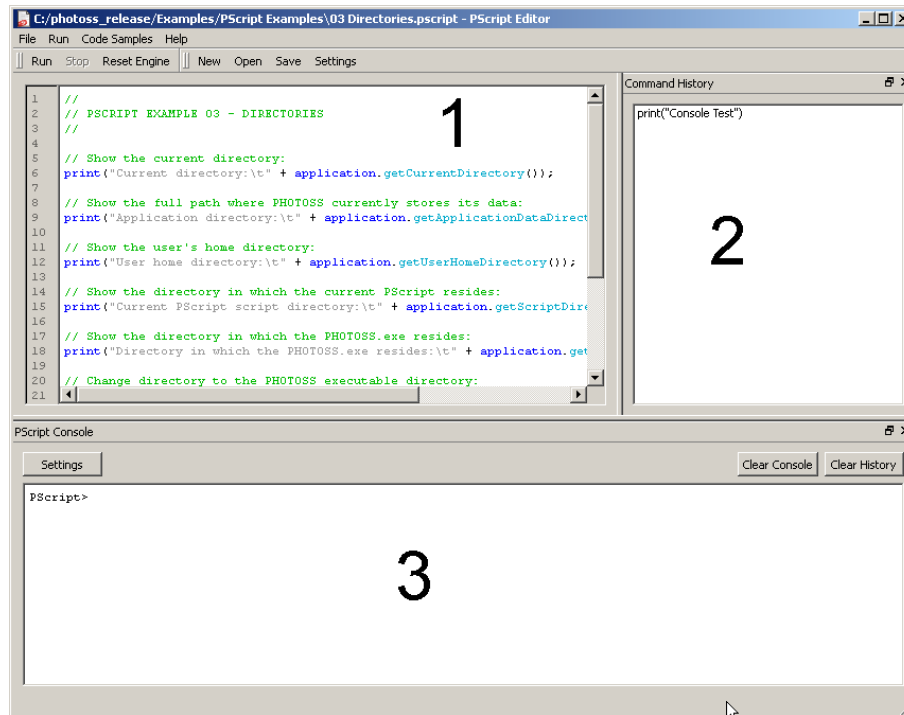


Figure 1. The PScript Console Main Window is divided into the Code Area (1), the Command History (2) and the Console Area (3).

The PScript Console Main Window consists of three 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 18)
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).

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 18 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 18 for more details on the specific options.

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: `mySettingsObjekt =PScriptSettings.getAll()`

Default: *NA*

Description: The get-methods reads all PScript Settings into a PScript Settings object which can be copied. The set-methods 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 PScript console. 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: `false`

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 15. 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: `600 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 `application.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 PScript Syntax Highlighting and Code Assistant

The PScript editor offers basic 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 `false`. The following SHL features are currently included:

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

The code assistant can be enabled and disabled using the PScript settings (see page 18), 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. Pressing the "Tab" key will automatically draw the first choice from the list. If you do not want to accept a suggestion of the code assistant, press "Esc" to close the assistant.

14 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 Investigate Parameter Variation.pscript` shows how parameters of a parameter variation can be displayed in 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 PScript and the MATLAB® interface 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 Global Variables.pscript` illustrates how global variables can be accessed and changed using PScript.
- `13 Type Conversions.pscript` shows 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.psscript` will familiarize you with several easy ways to include Sub-PScripts containing sub-functions, etc. in your Main PScript.
- 17 `Generic Scripting.psscript` demonstrates how generic scripts can be created which offer great flexibility and functionality at the same time.

15 PScript Function Reference

The following list includes all PScript functions 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.

15.1 PScript Options / Settings

Please refer to the chapter **12 - PScript Settings Dialog** on page **18** for a detailed explanation of the PScript settings and all available functions.

15.2 General Functions

Function: `foreach(double array, function(current element name){function code xyz})`

Code: `foreach(s.getComponentedByType("Analytical Filter"), function(c){c.setParameterValue("Type", "Gauss")})`

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.

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 18 for more details on the include list.

Function: `resetEngine()`

Code: `resetEngine()`

Description: Same as `clearWorkspace` with the exception that PScript Auto Includes *will* be executed.

Function: `runAutoIncludes()`

Code: `runAutoIncludes()`

Description: Same as `PScriptSettings.runAutoIncludes()` (see page 24).

15.3 Generating Random Numbers in PScript / PHOTOSS Random Generator settings

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. You have to open a PHOTOSS simulation first, though.

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. You have to open a PHOTOSS simulation first, though.

Function: `int getDefaultSeed()`

Code: `default_seed = rng.getDefaultSeed()`

Description: Writes the default seed for the PHOTOSS random generator of the current simulation into a PScript variable. You have to open a PHOTOSS simulation first, though.

Function: `double negExp(double lambda)`

Code: `alpha = rng.negExp(2.31)`

Description: Returns a negatively exponentially distributed random number with parameter lambda. You have to open a PHOTOSS simulation first, though.

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. You have to open a PHOTOSS simulation first, though.

Function: `double poisson(double lambda)`

Code: `alpha = rng.poisson(2.31)`

Description: Returns a Poisson-distributed double random number with the Poisson parameter lambda. You have to open a PHOTOSS simulation first, though.

Function: `setDefaultSeed(int seed)`

Code: `rng.setDefaultSeed(-324525)`

Description: Sets the default seed for the PHOTOSS random generator in the current simulation. This default seed will be used when calling `setDeterministic()` without a parameter. You have to open a PHOTOSS simulation first, though.

Function: `setDeterministic(int seed)`

Code: `rng.setDeterministic(-352215)//Supply a user seed`

Alternative Code: `rng.setDeterministic()//Use the default seed`

Description: Sets the PHOTOSS 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 PHOTOSS random generator will be chosen. You have to open a PHOTOSS simulation first, though.

Function: `setRandom()`

Code: `rng.setRandom()`

Description: Sets the PHOTOSS 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. You have to open a PHOTOSS simulation first, though.

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]. You have to open a PHOTOSS simulation first, though.

15.4 Functions concerning the PHOTOSS application class:

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.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. 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: `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.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: `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.getPScriptDirectory()`

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.

15.5 Functions concerning the PHOTOSS/PScript simulation, component and result class:

Please note that these functions only work if you have already opened at least one simulation with the `application.openSimulation`-function.

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: `bool simulationObject.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: `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.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: `globalVariableObject simulationObject.getGlobalVariable(string variableName)`

Code: `bitRate = mySimulation.getGlobalVariable("BitRate")`

Description: Returns an object of the specified global variable (aka "global parameter") in the currently active simulation. If the specified global variable does not exist in the currently opened simulation, an exception will be thrown. Please note that the global variable object holds the value, the name and the unit (if it exists) of the global variable. Use the `getValue()`-function on the global variable object to access its value. If you want to directly access the value of the global variable, use `getGlobalVariableValue()` instead.

Function: `globalVariableObject[] simulationObject.getGlobalVariables()`

Code: `globalVariablesVec = mySimulation.getGlobalVariables()`

Description: Returns a vector which contains the objects of all global variables (aka "global parameters") in the currently active simulation. The global variable objects each hold the name, the value and the unit (if it exists) of the individual global variable. Use the `globalVariablesVec[i].getValue()`-functionality on the global variables object to access their values or use `getGlobalVariableValue()` if you want to directly access their values instead.

Function: `string simulationObject.getGlobalVariableUnit(string globalVariableName)`

Code: `globalVarUnit = mySimulation.getGlobalVariableUnit("globalAlpha")`

Description: Returns a string which holds the unit of the specified global variable. If the variable does not have a unit, the string contains the string phrase "Unit". If the variable does have a unit, the string contains the phrase "[UnitName]".

Function: `double simulationObject.getGlobalVariableValue(string globalVariableName)`

Code: `globalVarVal = mySimulation.getGlobalVariableValue("globalAlpha")`

Description: Returns the value of the specified global variable. The return value may be a double or a string value.

Function: `double[] simulationObject.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: `parVals = 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: `simulationParameterObject[] simulationObject.getSimulationParameters()`

Code: `simParObjVec = mySimulation.getSimulationParameters()`

Description: Returns a vector containing the parameter objects of all simulation parameters in the given simulation. Please note that the simulation parameter objects are containing the names, values and units (if they exist) of the simulation parameters. Use `simParObjVec[i].getValue()` to access the individual values of the parameters or use the `.getSimulationParameterValues()`-function to directly access all parameter values.

Function: `double[] simulationObject.getSimulationParameterValues()`

Code: `simParValVec = mySimObject.getSimulationParameterValues()`

Description: Returns a vector containing the values of all simulation parameters 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 of the given simulation. If the parameter does not have a unit, an empty string is returned.

Function: `double simulationObject.getSimulationParameterValue(string simParameterName)`

Code: `parVal = mySimObject.getSimulationParameterValue("f0")`

Description: Returns the value of the specified simulation parameter of the given simulation.

Function: `string resultObject /parameterObject.getUnit()`

Code: `parUnit = myParameterObject.getUnit()`

Alternative Code: `resUnit = myResultObject.getUnit()`

Description: Returns the unit of a given component or result object. If the parameter does not have a unit, an empty string is returned.

Function: `double / string parameterObject /resultObject .getValue()`

Code: `paramValue = myParameterObject.getValue()`

Alternative Code: `resValue = myResultObject.getValue()`

Description: Returns the value of the given parameter or result object.

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.isGlobalVariableOfTypeVariation()`

Code: `myBool = mySimObject.isGlobalVariableOfTypeVariation()`

Description: Returns a bool value that specifies if the given global Variable is of the type variation.

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: `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.listGlobalVariables()`

Code: `mySimulationObject.listGlobalVariables()`

Description: Displays an overview of all global variables inside the given simulation (object) in the PScript console. The information includes the variable names, values and units (if they exist).

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: `simulationObject.listSimulationParameters()`

Code: `mySimObject.listResults()`

Description: Displays an overview of all simulation parameters of the given simulation in the PScript console. The information includes the simulation parameter names, values and units (if they exist).

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.

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.

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.

Functions: `simulationObject.setGlobalVariableValue(string varName, double varValue)`

`simulationObject.setGlobalVariableValue(string varName, double[] varValues)`

Code non-variation: `mySimObject.setGlobalVariableValue("Var1", 10.3)`

Code variation: `mySimObject.setGlobalVariableValue("Var2", [10.3,11.4,17.9])`

Assign the value `varValue` to the global variable with the name `varName`. If the global variable you want to modify is of the type variation, you can specify a vector of double values.

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.

Function: `simulationObject.setSimulationParameterValue(string parName, parValue)`

Code: `mySimulation.setSimulationParameterValue("ref_bitrate", 40.0)`

Description: Sets the simulation parameter with the name "parName" to the given 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 thrown and PHOTOSS will use the last valid value for the simulation parameter instead.

15.6 PScript MATLAB® specific functions

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 MATLAB® code 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)`

`bool matlab.getUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript()`

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`.

16 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], 17.

- 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 $\exp(\text{argument})$.
- 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.

17 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

MATLAB® specific functions, **39**

abs(), 41

Acknowledgments, **2**

acos(), 41

addAutoInclude(), **24**

addIncludePath(), 23

application class, **30**

application.close(), 30

asin(), 41

atan(), 41

autoIncludesContain(), 24

ceil(), 41

Chapter

Basic Functionality of PScript, **5, 6**

Basic JavaScript Mathematical Function Reference, **41**

External References, **42**

PScript and MATLAB®, **15**

PScript Code Samples, **26, 27**

PScript Console Main Window, **16**

PScript Console Menu Functionality, **17**

PScript Conventions, **8, 9**

PScript Demo Walkthrough, **10**

PScript Function Reference, **28–31, 33–40**

PScript Settings Dialog, **18–24**

PScript Syntax Highlighting and Code Assistant, **25**

Quickstart Guide, **11–14**

What is PScript?, **3, 4**

What PScript cannot (yet) do, **7**

clear(), 31

clearCommandHistory(), 21

clearWorkspace(), 28

close, *see* application.close(), *see* simulationObject.close()

collectGarbage(), 28

component class, **31**

component object, **8**

containsComponent(), 32

Copyright Information, **2**

cos(), 41

deleteFile(), 30

deleteFileType(), 30

eval(), 39

exp(), 41

fastpoisson(), 29

floor(), 41

for-loop, 10

foreach(), 28

gauss(), 29

general functions, **28**

getAll(), 18

getApplicationDataDirectory(), 30

getAutoIncludes(), 24

getBreakOnMatlabErrors(), **22, 39**

getBypass(), 32

getCheckForMatlabErrors(), **22, 39**

getComponent(), 12, **32**

getComponentsByType(), 32

getCurrentDirectory(), 31

getDefaultSeed(), 29

getGlobalVariable(), 33

getGlobalVariables(), 33

getGlobalVariableUnit(), 14, **33**

getGlobalVariableValue(), 14, **33**

getIncludeDetailedInformationInLogFile(), 20

getIncludeErrorsInLogFile(), 20

getIncludePaths(), 23

getIncludeTimeAndDateInLogFile(), 20

getIncludeWarningsInLogFile(), 21

getIterationResultValues(), 33

getLastError(), 39

getLastWarning(), 39

getLogFileCustomSavePath(), 21

getLogFileOverwrite(), 21

getLogFileUseCustomSavePath(), 21

getMatlabTimeoutInSeconds(), **22, 39**

getName(), 33

getParameter(), 34

getParameterDescription(), 34

getParameterNames(), 12, **34**

getParameters(), 34

getParameterUnit(), 12, **34**

getParameterUnits(), 34

getParameterValue(), 12, **34**

getParameterValues(), 35

getPHOTOSSExecutableDirectory(), 30

getPMDPath(), 35

getResult(), 35

getResultNames(), 13, **35**

getResults(), 35

getResultUnit(), 13, **35**

getResultUnits(), 13, **35**

getResultValue(), 13, **36**

getResultValues(), 13, **36**

getScriptDirectory(), 31

getShowCodeLineNumbers(), 19

getShowPHOTOSSErrorsInPScriptConsole(), 18

getShowPHOTOSSOutputInPScriptConsole(), 18

getShowPHOTOSSWarningsInPScriptConsole(), 18

getShowPrintedOutputInPHOTOSSConsole(), 19

getShowPScriptErrorsInPHOTOSSConsole(), 19

getShowPScriptOutputInPScriptConsole(), 19

getShowPScriptWarningsInPHOTOSSConsole(), 19

getShowTimeAndDateAtPrompt(), 18

getSimulationParameters(), 36

getSimulationParameterUnit(), 36

getSimulationParameterValue(), 13, **36**

getSimulationParameterValues(), 36

getSimulations(), 31

getUnit(), 36

getUseCodeAssistant(), 20

getUseHistory(), 21

getUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript(), setDefaultSeed(), 29
 22, 40
 getUseLogFile(), 20
 getUserHomeDirectory(), 11, **31**
 getUseSyntaxHighlighting(), 20
 getUseSyntaxHighlightingInConsole(), 20
 getValue(), 25, 36, **40**

include(), 28
 includePathsContain(), 23
 info(), 31
 investigate(), 36
 isAvailable(), 40
 isGlobalVariableOfTypeVariation(), 37
 isIterator(), 37
 isNetwork(), 37
 isOpen(), 37

listComponents(), 11, **37**
 listGlobalVariables(), 14, **37**
 listParameters(), 12, **37**
 listResults(), 13, **37**
 listSimulationParameters(), 37
 log(), 41

max(), 41
 min(), 41
 moveDownAutoInclude(), 24
 moveUpAutoInclude(), 24
 moveUpIncludePath(), 23

negExp(), 29

openSimulation(), 11, **31, 31**

pareto(), 29
 poisson(), 29
 pow(), 41
 print, 10

Quickstart Guide, **11–14**

random generator, **29**
 removeAutoInclude(), 24
 removeIncludePath(), 23
 resetEngine(), 28
 restoreDefaults(), 18
 result class, **31**
 round(), 41
 run(), 13, **37**
 runAutoIncludes(), 24, 28

save(), 38
 saveAs(), 11, **38**
 setAll(), 18
 setBreakOnMatlabErrors(), 22, **39**
 setBypass(), 38
 setCheckForMatlabErrors(), 22, **39**
 setCurrentDirectory(), 11, 31

setDeterministic(), 29
 setGlobalVariableValue(), 14, **38**
 setIncludeDetailedInformationInLogFile(), 20
 setIncludeErrorsInLogFile(), 20
 setIncludeTimeAndDateInLogFile(), 20
 setIncludeWarningsInLogFile(), 21
 setLogFileCustomSavePath(), 21
 setLogFileOverwrite(), 21
 setLogFileUseCustomSavePath(), 21
 setMatlabTimeoutInSeconds(), **22, 39**
 setParameterValue(), 12, **38**
 setPMDPath(), 38
 setRandom(), 29
 setShowCodeLineNumbers(), 19
 setShowPHOTOSSErrorsInPScriptConsole(), 18
 setShowPHOTOSSOutputInPScriptConsole(), 18
 setShowPHOTOSSWarningsInPScriptConsole(), 18
 setShowPrintedOutputInPHOTOSSConsole(), 19
 setShowPScriptErrorsInPHOTOSSConsole(), 19
 setShowPScriptOutputInPScriptConsole(), 19
 setShowPScriptWarningsInPHOTOSSConsole(), 19
 setShowTimeAndDateAtPrompt(), 18
 setSimulationParameterValue(), 14, **38**
 setUseCodeAssistant(), 20
 setUseHistory(), 21
 setUseIdenticalMatlabWorkspaceForPHOTOSSAndPScript(),
 22, 40
 setUseLogFile(), 20
 setUseSyntaxHighlighting(), 20
 setUseSyntaxHighlightingInConsole(), 20
 setValue(), 25, 40
 simulation class, **31**
 simulation object, **8**
 simulationObject.close(), 31
 sin(), 41
 sqrt(), 41

tan(), 41

uniform(), 30