QuARTIC™
v1.0

QuARTIC™
Qualified Automated Reporting Tool for Inputs and Calculation notebooks

Lance Larsen

Information Systems Laboratories, Inc.
Idaho Falls, ID

July 21, 2021

Prepared By:

Lance Larsen	

Table 0.2: Revision History

Issue No.	**Section**	**Description**
1	All sections	Initial Revision.

Table of Contents

1 Introduction

QuARTIC™ is a tool for including live calculations in an engineering calculation document. It was specifically designed for analytical model development to maintain consistency between design information, the model calculation notebook, which is used to calculate the various parameters used in a simulation model, and the model itself. Modules are available for integration with analysis tools like TRACE and RELAP5.

Maintaining consistency between the design information, the calculation notebook, and the model, in order to maintain high quality standards, is typically a time consuming and error prone process. QuARTIC™ was designed to make that process less error prone and labor intensive so that the burden of quality assurance is significantly reduced allowing projects to progress more quickly with higher confidence in the quality of the models that are developed.

A typical method of model development and documentation includes the following process, using a TRACE model as an example. First the information needed to build the model, such as plant geometry and materials, reactor kinetics parameters, safety system control logic information, etc, is gathered. Next, the information that was gathered in the first step is used to calculate required TRACE model inputs, such as volumes, flow areas, cell lengths, etc. The calculated data is then used to populate the TRACE model components. Documentation of the model often follows as a separate step. This typical approach to TRACE model development is depcited in Figure 1.1. This includes three primary deliverables:

1. The calculations performed to generate the model perhaps performed in a tool like excel.
2. Documentation of the calculations used to generate the model.
3. The model that is used to perform simulations.

The circular boundaries in Figure 1.1 represent interfaces between the deliverables in the the typical process where manual effort is required in order to keep the deliverables consistent. This approach introduces significant potential for the following types of errors:

- Transcription errors may occur, where values are not transferred correctly between the different deliverables.
- Calculation dependency errors can occur where a value is update in the calculations, but values in dependent calculations are missed and not updated appropriately.
- Changes in the equations used in the calculation tool may not get transferred to the documentation.

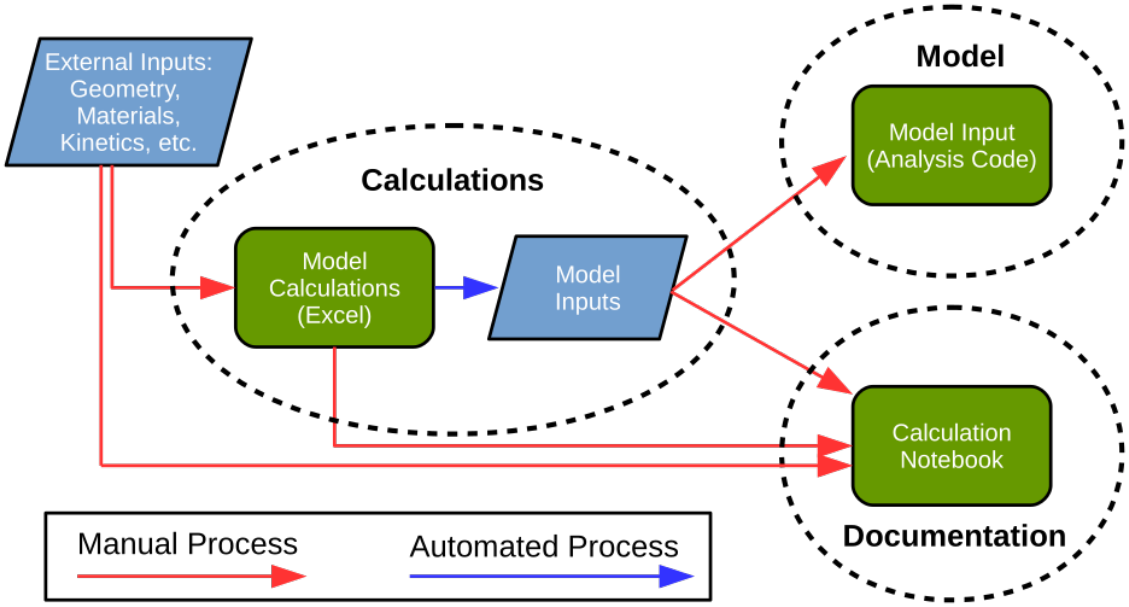

Figure 1.1: Typical Model Development Process

- Failure to transfer changes that occur in one of the deliverable items to the other deliverables.

Because of the potential for errors, significant time must be spent in the review process *just to verify consistency*. Keeping the deliverables consistent proves to be a difficult challenge, and review can be a time consuming (costly) process. Because of the burden of review, making small improvements to the model can require significant effort.

QuARTIC™ improves this process significantly by:

1. Including inputs in a parameters database (spreadsheet document), that becomes a single point to references for all inputs needed to develop the model. Each parameter is given a unique variable so that the parameter can be used in symbolic calculations.
2. Allowing symbolic calculations, which are expanded using values from the parameters database, to be embedded directly in the documentation rather than in an external tool like excel.
3. Ouput values that are model parameters are identified and written to an CSV file which provides a summary of the key calculated values.
4. QuARTIC™ modules are available for some tools that allow QuARTIC™ to automatically insert calculated parameters into the model to simplify the process of maintaining consistency between the documentation and the model.

The improved process is depicted in Figure 1.2.

The process of gathering the information necessary to build the model is still a manual process, as well as the process of setting up the symbolic equations in the calculation notebook. But there is no

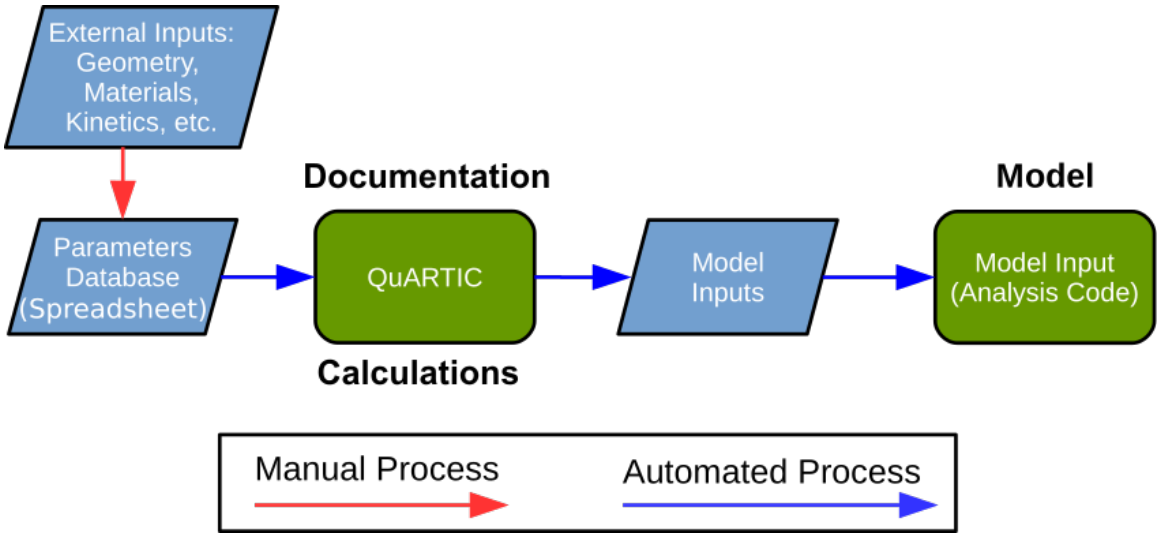

Figure 1.2: Improved TRACE Model Development Process

longer the need for a separate tool for performing calculations and documenting the calculations, so the calculation notebook is consistent with the actual calculations by definition. In addition, for analysis tools with a supporting QuARTIC™ module, the model parameters calculated in the notebook can be configured to automatically populate the model. In order to achieve this goal, QuARTIC™ is built on top of the following technologies:

1. Parameters are read from a set of CSV files which may be exported from a spreadsheet tool such as excel or libreoffice calc. Details about the format of the parameters database can be found in Section 2.3.

2. The calculation notebook is written in latex format with embedded scripting for calculations and other automation tasks. There are a few different programs available that process latex files and convert them to PDF. For QuARTIC™, the lualatex processor is used. Latex is a powerful text based technical documentation language that is commonly used for technical papers and books. Many resources for writing latex can be found on the internet. The latex file with embedded scripting is referred to as a template file in this documentation since the template file must be preprocessed by the QuARTIC™ scripts to generate an actual latex file that can then be processed by lualatex to generate the final PDF document. Section 2.4 provides instructions for building latex template files.

3. Lua is used as an embedded scripting language inside the template files for generating dynamic content (such as calculations). The Lua commands that are available for use within a template are described in Section 2.4.7.

2 Using QuARTIC™

In order to use QuARTIC™, one or more QuARTIC™ templates file needs to be created. In QuARTIC™, just as with latex, a document can be divided into multiple files which are combined in the final PDF document. The instructions for creating template files are provided in Section 2.4, with information about some of the QuARTIC™ features described in Section 2.4.7.

In order to generate the final PDF document, two steps are required. First the template files must be processed by QuARTIC™ in order to generate the final latex files. Then the latex files must then be processed by a latex processor to generate the final pdf.

It is recommended that both the template files and the final latex documents be maintained in a source control repository as part of the QA process. A source control repository is used to track changes to a document and see exactly what has been modified. Examining changes to the final latex documents between versions can be helpful for QA purposes to verify that all document changes are intended and are accurate.

2.1 Installation

In order to use QuARTIC™, the QuARTIC™ executable and a latex processor must be installed. Lualatex is a tool that processes latex documents, and is recommended for processing the latex documents generated by QuARTIC™. QuARTIC™ has been test using lualatex v 0.95 from the texlive 2016 distibution. Texlive is an opensource collection of tools for processing latex documents and can be downloaded from https://www.tug.org/texlive/. This site contains instructions for installation on windows, linux, and MacOS.

2.2 Running QuARTIC™

The QuARTIC™ executable is used to convert QuARTIC™ template files (latex with embedded calculations and scripting) into latex files. The basic format for executing QuARTIC™ is:

```
> QuARTIC.exe [options] <template_file>
```

The command line options listed below are available for the QuARTIC™ script. Note that the '=' following the command line option without spaces is required.

-a=<command line args file> A file containing command line arguments to use.

-c=<csv input dir> Specify the location of the csv parameters database files. All CSV files in the folder will be read in.

-init=<CSV init file> This is a CSV file that contains any values needed before processing other CSV files. This can be used to set preferred units so that parameters with with units will automatically be converted to preferred units as they are loaded (default init.csv).

-loglvl=<log level> Set the log level, which controls the type of information that is logged to the screen. Higher values result in more information being logged. The log levels are 1=ERROR, 2=WARNING, 3=INFO, 4=DEBUG.

-h Print the help message.

-H Show hidden equations.

-m Mark output parameters. This causes the parameters to be shown in **bold magenta** format when they are calculated.

-o=<latex output directory> The location where the final latex files will be stored after processing the template files (default ../Latex/)

-p Print items loaded from the CSV input files. Note that the -c option must be specified in order for this option to function.

-t=<template directory> Path to the latex template files (default ../tmp/)

-TI=<TRACE input file folder> Path to the TRACE input file.

-Ti=<TRACE input file name> The name of the TRACE input file to update using parameter calculated in the calculation notebook.

-TO=<TRACE output folder> Pathe where updated TRACE input files will be written (default is the TRACE input folder specified via -TI)

-To=<TRACE output file name> This is the name of the updated TRACE input file that is written to te TRACE output folder.

-Tp=<TRACE output param file> This is the TRACE input file with variable names inserted in place of values. This is use to verify that values are inserted in the correct location. It is also used to check that values calculated in the calculation notebook are inserted as expected.

-X Flag to turn off compilation of the template files. This is used when manually editing the template files for debug purposes.

When running from the command line, it is inconvenient to pass in a long list of command line parameters. By default, QuARTIC™ will look for a file named 'QuARTIC.ini' and load default parameters from this file if present in the same directory as QuARTIC.lua. This can be overriden by passing in a **'-a=<args file>'** option which defines a different args file to read in. Any options specified on the command line will override options in the args file.

2.2.1 QuARTIC™ Messages

As QuARTIC™ runs, it prints various messages. Some messages indicate the current task being performed by QuARTIC™, such as loading parameters from a CSV file. Other messages may indicate warnings or errors. Warnings and errors are preceded by a "WARNING" or "ERROR" label respectively. The '-loglvl' command line option can be used to control the amount of detail that is printed to the console by QuARTIC™.

2.3 The Parameters Database

The parameters database is used to initialize data that will be used in the calculation notebook. The parameters database is a set of CSV files that are processed by QuARTIC™ prior to compiling and running the template files. The CSV files are best maintained in a spreadsheet document and exported as CSV. Macros are available for excel and libreoffice that export spreadsheet tabs as CSV files.

QuARTIC™ expects the parameter database to be structured in a specific way that allows data to be conveniently represented as tables, but formal enough to know how to interpret the data. A single CSV file can contain multiple tables of data. Each table starts with a header that specifies the table type (or command) in the first column, and table fields in the columns that follow. The table fields may come in any order, and columns may be left blank if desired. The tables will be easier to work with however if tables are organized with the fields in the same order in each CSV file.

Each table type (command) supports a specific set of fields, some of which are necessary and some which are optional. Custom fields can also be added if useful. Table entries follow the header row. Each table entry will leave the first column blank, and will fill in values as appropriate for the fields identified in the header row. Comment lines can be added by putting a '#' character in the first column. Blank lines are ignored.

The following command types are supported:

Table 2.1: Parameter Database Commands

Command/Fields	**Description**
acronyms	A table of acronyms and definitions for acronyms used in the parameter descriptions
- acronym	The acronym or abbreviation.
- definition	Definition of the acronym or abbreviation
parameter	Define a parameter (or variable) to be used in the document.
- name	The parameter name. Some special conventions apply to the name. This will be discussed in Section 2.3.2.

Table 2.1: Parameter Database Commands (continued)

Command/Fields	**Description**
- value	The value associated with the parameter. This can be a number or a string.
- units	This column is used to specify the units for the parameter. Note that if a compatible unit is specified in the preferred units, the value is converted to preferred units for use in the calculation notebook. To override this, set the calcunits field. Conventions for specifying units is discussed further in Section 2.3.3.
- ref	Short name of the reference the parameter comes from.
- loc	Specific location of the parameter within the reference.
- desc	Description of the parameter.
- acronyms	An '|' separated list of acronyms used in the parameter description. This is used for detecting which acronyms should be included in the autogenerated acronyms table.
- fmt	Format of the numeric value. This can be used to control the number of significant digits for a variable. The format follows the Lua string convention for formatting numerical values.
- tex	Optional latex representation of the variable (in math mode syntax). Note that QuARTIC™ does create a default latex representation for variables. This is described in Section 2.3.2.
preferred	Specify preferred units for a specific unit type
- units	The preferred units. For example the preferred units for length might be 'm' (meters) or 'ft' (feet). The preferred unit can be overridden for a parameter by specifying the 'calcunits' field.
references	A list of references used by parameters in the parameter database
- ref	A short name for the reference. Most parameters should include a reference, and the short 'ref' name is used for the reference.
- title	The title for the referenced item.
- doc_id	A document id perhaps from an external reference system
- rev	The document revision information
- desc	A description of the document.

One and two dimensional arrays can also be included in the parameters database via the 'array' command. However, the format of arrays is not consistent with some of the conventions discussed

above. So array commands are discussed further in Section 2.3.1.

The following example shows how commands may be organized in a spreadsheet. This spreadsheet can then be exported as a CSV file.

Table 2.2: Parameter Database Commands

# Comments can be added as lines that start with a '#' symbol.						
acronyms		**acronym**				**definition**
		LC				QuARTIC
		PD				Pressure Dome
		XSEC				Crossover Section
parameter	**name**	**value**	**units**	**acronym**	**ref**	**desc**
	`Vol_PD`	220	`m^3`	LC|PD	DOC	LC volume of the PD
	`Len_PD`	220	m	PD	DOC	Length of the PD
	`Vol_XSEC`	10.3	`m^3`	XSEC	DOC	Volume of the XSEC
preferred	**unit**					
	`cm^3`					Preferred Volume (comment)
references	**ref**	**rev**				**title**
	DOC	1				Reference document title

2.3.1 Specifying Arrays

It is common to have a set of one dimensional arrays that are the same length that contain related data. The 'array' command can be used to construct a set of related one dimensional arrays. This can be organized in a spreadsheet as follows:

Table 2.3: One Dimensional Array Command

array	<array 1 name>	<array 2 name>	...
units	<array 1 units>	<array 2 units>	...

Table 2.3: One Dimensional Array Command

	<value 1>	<value 1>	...
	<value 2>	<value 2>	...
	<value 3>	<value 3>	...
	...	...	...

The array names must conform to the Param naming convention (Section 2.3.2). In some cases, it is useful to provide a description, reference information, or other fields for one or more of the array parameters. To do this, a 'parameter' command may be used after the array is specified, where the parameter has the same name as one of the arrays in the array table. The parameter item should not specify a value, but may specify a 'unit' for the parameter. The 'units' row is optional in the array since the array values may not have units, and since the units may be specified as part of the 'parameter' command if desired. Note that the use of a 'parameter' command after the array applies to two dimensional arrays as well. The following is a spreadsheet example array:

Table 2.4: One Dimensional Array Example

array	**Time**	**Level**	**Pressure**
units	**s**	**m**	**MPa**
	1	0.2	1.6
	1.5	0.3	1.7
	2	0.5	1.9
	2.5	0.7	2.0

If a description and reference is needed for the array parameters, then the following spreadsheet layout could be used (note that units can be included with the array or parameter command or excluded altogether if not needed):

Table 2.5: One Dimensional Array with Parameter Command

array	**Time**	**Level**			**Pressure**
	1	0.2			1.6
	1.5	0.3			1.7
	2	0.5			1.9
	2.5	0.7			2.0

Table 2.5: One Dimensional Array with Parameter Command

parameter	**name**	**units**	**ref**	**loc**	**desc**
	Time	s	Jrnl	pg 52	Simulation Time
	Level	m	Jrnl	pg 57	Water Level
	Pressure	MPa	Jrnl	pg 54	System Pressure

The format of a 2D array is ad follows:

Table 2.6: Two Dimensional Array Command

array	<array name>			...
	<y axis name>	<y axis val 1>	<y axis val 2>	...
	<x axis name>			...
	<x axis val 1>	<array val 1,1>	<array val 1,2>	...
	<x axis val 2>	<array val 2,1>	<array val 2,2>	...
	...	...	...	...

With a two dimensional array, the units for the array and for the axes must be specified using a 'parameter' command if units are required. The following is an example two dimensional array.

Table 2.7: Two Dimensional Array Example

array	Tmp			
	Yloc	0.2	0.4	0.6
	Xloc			
	0.1	95.2	99.3	103.7
	0.2	96.1	101	104.2
	0.3	97.4	102	105.6
parameter	name	units		
	Tmp	F		

Table 2.7: Two Dimensional Array Example

	Yloc	ft		
	Xloc	ft		

Note that arrays may be used in calculations, but when arrays are used in calculations, the equation is displayed symbolically, and will be shown with scalar values inserted, but array values will be shown symbolically. No final result is automatically printed. The results can be shown in a table however. See Section 2.4.7.2 for information on working with arrays.

2.3.2 Param Naming Conventions

The parameters defined in the parameters database are intended to be useable in calculations in the calculation notebook. In order to use parameters in the calculation notebook, the parameters must use a name that is a valid Lua variable name. Lua variables are case sensitive and may include letters, numbers, and underscore characters, but may not start with a number.

Parameters also have to be represented in the Latex document, and thus need a valid representation in latex math mode syntax. The latex representation of a parameter can be specified explicitly in the parameters database by setting a 'tex' field for the parameter. However, a tex representation of the parameter name is autogenerated according to the following rules:

- The first underscore used in a parameter name represents a subscript. Each underscore thereafter is converted to a dot. For example, the name `VAR_121_x_23` is represented as $VAR_{121.x.23}$
- A range is specified in the subscript using the form NxN where the captital N values represent numbers. The 'x' value is converted to a dash. The range must come at the end of the subscript to be recognized as a range. For example, `VAR_121_1x10` is represented as $VAR_{121.1-10}$.
- Double underscores before the first underscore are represented as an underscore character. For example, `VAR__1__2_name` is represented as $VAR\_1\_2_{name}$

2.3.3 Conventions for Specifying Units

2.4 Template Files

Template files are written in latex format. A brief discussion of latex is presented in Section 2.4.1.

2.4.1 Latex

Latex, which is pronounced as 'Lah-tek', is a document preparation system for technical documents that has been around for many years. Latex documents are written in plain text, and as a philosophy, latex tries to make it easy for authors to add content without having to focus too much on appearance. A document writer uses tags to identify section headings, and blank lines to indicate the end of a paragraph. However, aside from blank lines, whitespace is generally ignored.

If you are unfamiliar with latex, there are a number of resources available on the internet that provide assistance. You can search for "A Beginners Guide to Latex" by David Xiao, which provides a brief introduction to latex. This shows how to format a latex document, how to include sections, tables, lists, labels and cross references, and how to add equations or mathematical expressions. A more comprehensive introduction is the "Latex for Beginners" workbook. This covers the topics addressed in "A Beginners Guide to Latex". In addition, it shows how to add a table of contents, how to use font effects, such as bold or italics, how to adjust font color or size, how to add figures, and how to add a bibliography and reference bibliography items.

Latex has a basic set of features built in that are useful for writing technical documents. The basic set of features can be expanded or modified using latex packages that add additional capability. The site http://tug.ctan.org has a repository of latex packages, and documentation for the packages tends to be included. Note that texlive has many packages preinstalled, so that they are available for use without requiring a separate installation process. Summaries of commonly used latex packages can be found on the web. One such summary is located at https://en.wikibooks.org/w/index.php?title=LaTeX/Package_Reference.

2.4.2 Template File Syntax

Latex is a useful tool for writing technical documentation. However, it does not have a convenient way to perform live calculations (i.e., symbolic calculations that are automatically expanded based on values defined for the variables). In order to add this capability, a syntax was defined for embedding Lua scripting inside a latex document. Embedded scripting is useful in some other ways as well. For example, the parameters that are used in the document are tracked, and an autogenerated table of inputs with references can be inserted into the document sith a simple script command.

Script blocks are included in the document using special markers that identify the block of text as Lua code. A few different block types are supported:

Code Block: @[<Lua code>]@ or @ at start of line
This is used to include a block or line of Lua code in the latex document. For example, to call the function 'InputTable' one of the following can be used:

```
@[ InputTable() ]@
@ InputTable()
```

Equation Block: @$ <eqn>$@ or @$$ <eqn>$$@
This is used to include an inline equation or a standalone equation. The inline equation

is typically included in within a paragraph, or sometimes a table cell, while the standard equation is on its own line in the document, and may be numbered.

Comment Block: @# #@ or # at start of line
A comment block or line is skipped and not included in the document.

Some of these blocks include modifiers or options that adjust the behavior. These are discussed in more detail, in the sections that follow, with examples of usage.

2.4.2.1 Code Blocks

Script blocks are used to directly include Lua code in a document. The code can be used for a wide variety of purposes, but on primary purpose is to write content to the latex document. Because this is so common, a special modifier has been included in the code block syntax. The code block syntax is:

```
@[-=? <Lua code> -]@       (Inline block)
@?  <line of Lua code>     (Single line block)
```

An inline block can be used to insert Lua code inside a paragraph. It can also be used to insert Lua code that spans multiple lines. The start and end markers allow flexibility in the placement of inline blocks. The block can start and end at any location within the document, except for within another block, and allows arbitrary Lua code to be included, within a few limitations. For example, the Lua code cannot contain ']' since this is used to close a block. There are also some predefined names that the block should not use.

The single line block only requires a marker at the start of the line (no spaces are allowed before the '@' symbol), but can only span a single line. This is useful for short commands, but is less flexible.

2.4.2.1.1 Code Block Modifiers

The modifiers/options must come immediately after the opening marker ('@[' for multiline blocks or '@' for single line blocks) or immediately before the closing marker (']@'), with no spaces between.

The whitespace (-) modifier

The '-' modifiers for the multline block, which may be included with the opening or closing markers, cause any whitespace and newline before or after the text (depending on whether the minus is used with the opening or closing marker) to be removed from the document. The minus sign must be placed immediately after the opening marker ('@[') or immediately before the closing marker (']@'). No other modifier can be placed between the (-) and the marker.

In general, latex is insensitive to spaces and newlines, in the sense that multiple spaces are treated as a single space when generating the pdf, and multiple blank lines are treated as a single blank line (marking the start of a new paragraph). Occasionally the removal of whitespace and newline

will impact the format of the final PDF, but in general, it mostly impacts how the final 'tex' file looks. Removal of extra whitespace can make the document look neater and easier to read.

The write (=) modifier As noted previously, one of the most common purposes of including a block of Lua code is to write dynamically generated content (i.e., content that can change) to the latex document. The '=' modifier causes the item contained in the block to be written to the document. As an example, assuming a parameter $p = 5$ MPa, and assuming a variable $a = 100$, and a function 'add2' is defined that takes a value and adds 2, as in the following code block (which does not include a write modifier), the table that follows shows how these values can be printed.

```
@[
a = 100

function add2(x)
  return x + 2
end
]@
```

Block Syntax	Result	Description
`@[= p ]@`	$p = 5$ MPa	When a parameter is printed, the name of the parameter is included as well as the value and units.
`@[= a ]@`	100	For a standard Lua variable, the value of the variable is printed.
`@[= add2(5) ]@`	7	For a function, the value returned from the function is printed.

The delay (?) modifier

In some cases, a value may need to be printed in the document before it has been defined. In this case, the template cannot write the value to the output because the value is not known. The delay (?) modifier causes the item to be written out after the document has been fully processed. Thus the item can be printed even if it was defined after the request was made to write the item.

Note that this works well if the value of an item (i.e., variable or parameter) is only specified once. However, if the value changes in the document, the value that is printed may not be the desired value. Thus, it is recommended that this only be used with values that are defined once and not modified. In the example below, the value of b is not defined until after it is printed.

```
the value of b is @[=? b ]@.

@ b = 10
```

This results in: the value of b is 10.

2.4.2.2 Equation Blocks

Equation blocks are used to perform calculations in a calculation notebook and automatically document the equations. Two types of equation blocks are supported: inline equations (@$ <eqn>$@), which can be placed inside a paragraph or a table cell, and display equations (@$$ <eqn> $$@), which are represented in their own paragraph, and may include an equation number. The syntax for equation blocks is:

```
@$>#^ <equation(s)> $@
@$$>1-^ <equation(s)> $$@
```

Equations within the equation blocks are written in standard Lua mathematical syntax. All variable names must be valid Lua variable names, although these will be rendered as indicated in Section 2.3.2. More than one equation can be included in an equation block, and when a set of display equations is grouped, it is recommended that these be included in a single block since spacing between equations is large when equations are included in separate equation blocks. For example, consider the equation blocks shown below and rendered thereafter:

```
@$$ L_1 = 5*m $$@
@$$ L_2 = 2*m $$@
@$$ L_3 = 1*m $$@
```

$$L_1 = 5 \text{ m} \tag{2.4.1}$$

$$L_2 = 2 \text{ m} \tag{2.4.2}$$

$$L_3 = 1 \text{ m} \tag{2.4.3}$$

When this is included in a single equation block, the spacing is better:

```
@$$
L_1 = 5*m
L_2 = 2*m
L_3 = 1*m
$$@
```

$$L_1 = 5 \text{ m} \tag{2.4.4}$$

$$L_2 = 2 \text{ m} \tag{2.4.5}$$

$$L_3 = 1 \text{ m} \tag{2.4.6}$$

Sometimes it is useful to include numeric calculations with raw numbers rather than parameters. When raw numbers are used, the value is computed before generating an equation. Thus, the

calculation is not documented. For example, the equation block below produces only a final value of x as shown:

```
@$$
x = 12*4/3
$$@
```

$$x = 16 \tag{2.4.7}$$

To represent the actual calculation, the numbers need to be converted to parameters. Note that a number that is multipled by a parameter is automatically converted to a parameter, so it is adequate to convert 4 in the equation above to a parameter. The function `P_` is used to convert a number to a parameter as shown below:

```
@$$1
x = 12*P_(4)/3
$$@
```

$$x = \frac{12 \cdot 4}{3} = 16 \tag{2.4.8}$$

2.4.2.2.1 Equation Blocks Create 'Param' Objects Equation blocks are special in that the variables calculated in an equation block is a Param object (see Section 2.4.7.1). Param objects are the basic component of symbolic calculations in QuARTIC™.

An essential characteristic of a Param object defined in an equation block is that it records its own variable name. For example, when the following code block is used:

```
@$$ A_100 = 120*m $$@
```

$$A_{100} = 120 \text{ m} \tag{2.4.9}$$

then the parameter name is stored as `A_100.name` which has the value `A_100`. To display a Param name in the document, the form `$@[=A_100.tex]@$` may be used, which in this displays: A_{100}.

When used in subsequent equations, a named Param object displays its symbolic name (i.e., 'tex' value) in the equation. Normal Lua variables, as defined in Lua code blocks, do not have access to a variable name, and will not show a symbolic name if use in equations.

2.4.2.2.2 Equation Block Modifiers The modifiers (i.e., >, #, ^, <, -, and 1) are all optional, and can be used alone or in combination with other modifiers. The ordering does not matter. The purpose of each modifier is described below.

The Output Parameter Modifier (>)

The output modifier applies to both the inline and display equation blocks. This modifier marks a parameter as an output parameter. Output parameters are written to a csv file when the calculation notebook is processed. In the following, the parameters a, b, and c are marked as output parameters:

```
@$$>
a = 10*s
b = 20*s
c = 30*s
$$@
```

Show Numerical Value Only Modifier (#)

The show numerical value only modifier applies to the inline equation block. Typically, when an equation block is included, the name of the variable is shown, followed by the symbolic calculation, and then the numerical result as shown in the example below:

```
@$ V = A*L $@
```

$V = A \cdot L = 2\ \text{m}^2 \cdot 5\ \text{m} = 10\ \text{m}^3$

However, in some cases, it may be preferrable to show only the result. One common use for this is showing calculated values in a table, where the name pattern for the calculated values may be shown in the heading of the table. For example, a table might be used to calculate the volume from length and area. Suppose the following parameters are defined: $A_1 = 2\ \text{m}^2$, $A_2 = 2\ \text{m}^2$, $A_3 = 2\ \text{m}^2$, $L_1 = 3\ \text{m}$, $L_2 = 6\ \text{m}$ and $L_3 = 5\ \text{m}$. A table is included that calculates the volume in the last column using the equation below (where i is replaced by the specified index):

```
@$# V_i = A_i*L_i $@
```

i	A_i	L_i	V_i
1	$2\ \text{m}^2$	$3\ \text{m}$	$6\ \text{m}^3$
2	$2\ \text{m}^2$	$6\ \text{m}$	$12\ \text{m}^3$
3	$2\ \text{m}^2$	$5\ \text{m}$	$10\ \text{m}^3$

Hide Equation Modifier (^)

This modifier causes the calculation to be performed, but the result will not be displayed.

No Numerical Substitution Equation Modifier (<)

This causes the numerical substitution display of the equation to be suppressed. This is particularly

useful when QuARTIC fails to recognize that an equation has no symbolic variables. In this case, the numerical equation can be printed twice, so this option can be used to suppress the exteraneous printed equation.

```
@$$<
x = sin(30*deg)/2
$$@
```

$$x = \frac{\sin\left(30\ \text{deg}\right)}{2} = 0.25 \tag{2.4.10}$$

The Single Line Equation Modifier (1)

When a display equation block is included in a document, then the equation is typically represented on 3 lines. The first line shows the symbolic representation of the equation. The second line shows the equation with values inserted, and the third line shows the final value. Sometimes all three of these will fit on a single line. The 1 modifier causes the equations within the block to be represented on a single line. For example, suppose that $x = 2$ m and $y = 2$ m. The following equation is relatively short, but gets spread across 3 lines:

```
@$$ r = (x^2 + y^2)^0.5 $$@
```

$$\begin{aligned} r &= \left(x^2 + y^2\right)^{0.5} \\ &= \left((2\ \text{m})^2 + (2\ \text{m})^2\right)^{0.5} \\ &= 2.8284\ \text{m} \end{aligned} \tag{2.4.11}$$

The equation is represented more compactly with the '1' modifier.

```
@$$1 r = (x^2 + y^2)^0.5 $$@
```

$$r = \left(x^2 + y^2\right)^{0.5} = \left((2\ \text{m})^2 + (2\ \text{m})^2\right)^{0.5} = 2.8284\ \text{m} \tag{2.4.12}$$

The No Equation Label Modifier (-)

By default, display equations include an equation number. This is useful if one wishes to reference the equation in another part of the document. However, some equations may be local and temporary in nature, and it may be desirable to exclude an equation number. The '-' modifier eliminates the equation number from equations contained in the block. For example the following equations include no equation number:

```
@$$-
x = 5
y = 3
$$@
```

$$x = 5$$
$$y = 3$$

The Hide Equation Modifier (^)

The hidden equation modifier causes the equations in the block to be calculated, but no result is shown in the document. The -H command line option causes hidden equations in the document to be displayed.

2.4.2.2.3 Equation Style In some cases, it is convenient to override equation modifiers. For example, a long equation might be included in a block of short equations. It makes sense to use the single line equation modifier (1). But it might be necessary to override this for one (or more) equations. Another example, is a block of equations where all the equations are output parameters (>) with exception of one to two equations. It is convenient to be able to modify the behavior on these equations.

Given a variable `param`, the following methods are available to override the modifiers:

- '`param.save = false`' indicates that are parameter is not an output parameter and overrides the output parameter modifier (>). This must come within the equation block AFTER the parameter equation. This is not required to come immediately after, but must be within the same equation block.
- '`param.save = true`' marks the parameter as an output parameter when the output parameter modifier is NOT used. This is useful if an output parameter is included in a block of multiple equations that are not output parameters. This must come in the equation block after the parameter equation.
- '`param.save = false`' will mark the equation as a multiline equation. This will override the single line equation modifier (1). This must be included after the parameter equation.
- '`param.style.one = true`' will mark the equation as a single line equation when the single line equation modifier is not used. This must be specified after the parameter equation.
- '`param.style.after = [[latex]]`' is used to change the latex that comes after each equation. By default the command `EqnSkip` is inserted, which is used to insert negative vertical space so that equations are grouped closer together. Note that the `EqnSkip` command should be defined at the top of the latex document.

The following can be set to modify the value for all equations within the equation block that follow the given command. This will not affect parameters where the value is set explicitly for the parameter.

- 'style.one = value'
- 'style.save = value'
- 'style.after = [[latex]]'

The following is an example of the one property with the single line property disabled for the last equation:

```
@$$1-
R  = 10*m  -- Cylindircal Annulus Outer Radius
Th = 0.5*m -- Anulus Thickess
r  = R-Th  -- Annulus Inner Radius
h  = 20*m  -- Cylindrical Annulus height
Vol = h*Pi*(R^2 - r^2)  -- Volume of Cylindrical Annulus
Vol.style.one = false
SA  = h*2*Pi*(R+r) + 2*pi*(R^2-r^2) -- Surface Area
SA.style.one = false
$$@
```

OR

```
@$$1-
R  = 10*m  -- Cylindrical Annulus Outer Radius
Th = 0.5*m -- Anulus Thickess
r  = R-Th  -- Annulus Inner Radius
h  = 20*m  -- Cylindrical Annulus height
style.one = false
Vol = h*Pi*(R^2 - r^2)  -- Volume of Cylindrical Annulus
SA  = h*2*Pi*(R+r) + 2*pi*(R^2-r^2) -- Surface Area
$$@
```

$$
\begin{aligned}
&\pi = 3.1415 \\
&R = 10 \text{ m} \\
&Th = 0.5 \text{ m} \\
&r = R - Th = 10 \text{ m} - 0.5 \text{ m} = 9.5 \text{ m} \\
&h = 20 \text{ m} \\
&Vol = h \cdot \pi \cdot \left(R^2 - r^2\right) \\
&\quad = 20 \text{ m} \cdot 3.1415 \cdot \left((10 \text{ m})^2 - (9.5 \text{ m})^2\right) \\
&\quad = 612.59 \text{ m}^3 \\
&SA = h \cdot 2 \cdot \pi \cdot (R + r) + 2 \cdot \pi \cdot \left(R^2 - r^2\right) \\
&\quad = 20 \text{ m} \cdot 2 \cdot 3.1415 \cdot (10 \text{ m} + 9.5 \text{ m}) + 2 \cdot 3.1415 \cdot \left((10 \text{ m})^2 - (9.5 \text{ m})^2\right) \\
&\quad = 2511.6 \text{ m}^2
\end{aligned}
$$

2.4.2.2.4 Inserting latex between equations To insert arbitrary latex between equations, the `latex([[<latex to insert>]])` command may be used within the equation block. Note that if this is done, the default latex placed after the equation may need to be changed for formatting to work correctly.

```
@$$1-
a = 14
a.style.after = [[]]
latex([[Comment between equations.
Note that this can be multiline.]])
b = 12
$$@
```

$$a = 14$$

Comment between equations. Note that this can be multiline.

$$b = 12$$

2.4.2.2.5 Handling Long Equations Sometimes equations are quite long, and need to be split across multiple lines when they are shown in the document. A special `brk(mode)` command is available for use in equations to specify where breaks are to occur. The `mode` parameter may have the values:

- 'symbolic' (in quotes) to indicate that only the symbolic equation should break to the next line at the location.
- 'value' (in quotes) to indicate that only the equation with numerical values inserted should break to the next line at this location.
- No value to indicate that the equation should break for both the symbolic and numerical equation.

Note that there are some places within an equation where breaks are not allowed (such as within a parenthesized block, or within terms contained in a fraction). Placing a break in such locations will cause errors when trying to convert the latex file to a PDF document. Below is an example of an equation with a long symbolic representation. Thus, the symbolic representation needs to have a break. The equation with valies substituted fits on a single line, so only the symbolic equation includes a break.

```
@$$-
Length_x_101 = 100*m
Length_x_102 = 50*m
Length_x_103 = 60*m
Length_x_104 = 80*m
Length_x_105 = 120*m
Length_x_106 = 50*m
Length_total = (Length_x_101 + Length_x_102 + Length_x_103
               + Length_x_104 + Length_x_105):brk('symbolic') + Length_x_106
$$@
```

$$
\begin{aligned}
Length_{x.101} &= 100 \text{ m} \\
Length_{x.102} &= 50 \text{ m} \\
Length_{x.103} &= 60 \text{ m} \\
Length_{x.104} &= 80 \text{ m} \\
Length_{x.105} &= 120 \text{ m} \\
Length_{x.106} &= 50 \text{ m} \\
Length_{total} &= Length_{x.101} + Length_{x.102} + Length_{x.103} + Length_{x.104} + Length_{x.105} \\
&\quad + Length_{x.106} \\
&= 100 \text{ m} + 50 \text{ m} + 60 \text{ m} + 80 \text{ m} + 120 \text{ m} + 50 \text{ m} \\
&= 460 \text{ m}
\end{aligned}
$$

2.4.2.3 Comment Blocks

Template comment blocks are text that is excluded from the final Latex document.

```
#  A '#' symbol at the start of the line indicates a comment line
@# This is a comment block, which can span multiple lines. #@
```

2.4.3 Including External Template Files

In some cases, it is useful to break documentation into multiple files. QuARTIC™ has the ability to include external template files in a template file. This can be useful for keeping the size of files manageable for large documents. As a simple example, consider a file called 'test.tmp' that contains the following text:

```
This text comes from 'test.tmp'.

@$$ x=1 $$@
```

The template file can be included via the 'include' function as follows, with the result shown afterward:

```
@[ include("test.tmp") ]@
```

This text comes from 'test.tmp'.

$$x = 1 \tag{2.4.13}$$

2.4.4 Dynamic Template Functions

The template file may contain sections that have a standard format, but document different data. QuARTIC™ has the ability to include functions which can be used to make portions of the document into macros that are expanded with data that is passed to the function. This can be used for simple parts of a document that are repeated frequently, or for large and complex sections. Note that macro functions can be included in the base document, or as external template files that act like libraries which may be useful in multiple documents.

As a simple example, consider the following function which receives a variable which is a list and generates an enumerated list from the items:

```
@ function makeList(items)
Here are the list items:

\begin{enumerate}
@ for i, item in ipairs(items) do
  \item @[= item]@
@ end
\end{enumerate}
@ end
```

This can be called with a list of items as follows with the results shown afterward:

```
@ makeList({'First item', 'Second item', 'Third item','And so on'})
```

Here are the list items:

1. First item
2. Second item
3. Third item

4. And so on

2.4.5 Including Callbacks

For more complex sections that are rendered as macros, it is common to need to insert customized content. Consider an example case. The input for a pipe component for an analysis code is being documented. A macro called 'PipeDoc' is created that is passed a data structure with pipe inputs. An input called 'ID' is included that gives a pipe ID number, and an input called 'length' is available that is an array of pipe segment lengths in meters which will be shown in a table. The function might look something like the following (note that indentation is included for readability, but is not required in actual document):

```
@ function PipeDoc(cmp)
  @[ try("PipeLen" .. cmp.ID) ]@

  For pipe number @[= cmp.ID]@, the cell lengths are:

  \begin{tabular} {c c}
    \hline
    Cell \# & Length (m) \\
    \hline
    @# LOOP THROUGH THE CELL LENGTHS #@
    @[ for i, length in ipairs(cmp.length) do ]@
      @[=i]@  & @[=length]@ \\
    @[ end ]@
  \end{tabular}

  [Additional pipe info documented here]
@ end
```

If the function is called, and the callback macro is not defined, then the callback will be skipped and the rest of the content wil be rendered as shown below:

```
@ PipeDoc({ID=2, length={1,3,2,4}})
```

For pipe number 2, the cell lengths are:

Cell #	Length (m)
1	1
2	3
3	2
4	4

[Additional pipe info documented here]

If the function 'PipeLen2' is defined before 'PipeDoc' for component 2 is called, then the callback function will be called and included in the documentation as shown below:

```
@ function PipeLen2()
  For pipe 2, the pipe lengths are calculated as:

  @$$
  L2_1 = 1
  L2_2 = 3
  L2_3 = 2
  L2_4 = 4
  $$@
@ end

@ PipeDoc({ID=2, length={1,3,2,4}})
```

For pipe 2, the pipe lengths are calculated as:

$$L2_1 = 1 \tag{2.4.14}$$
$$L2_2 = 3 \tag{2.4.15}$$
$$L2_3 = 2 \tag{2.4.16}$$
$$L2_4 = 4 \tag{2.4.17}$$

For pipe number 2, the cell lengths are:

Cell #	Length (m)
1	1
2	3
3	2
4	4

[Additional pipe info documented here]

2.4.6 Lua Function Available in Template File

As with standard Lua, a template can access function in an external lua file by using the `require` command to load the module. However, a few functions, shown in the table below, are available by default.

Table 2.8: Functions Available by Default

Function	Description
isParam(var)	Check whether a variable is a `Param` object. Param objects are specified in an external input file or are generated via equations or may be created with the `Param` method below.
isUnit(var)	Check whether a variable is a Unit object. Several unit variables are predefined and can be used in equations to get unit consistency.
latexEsc(text)	Write a string as latex, converting special symbols so that they print in latex. For example, the symbols $, {, }, [, and] are converted to a form where they are printable.
markOutputParams	
Param	
ParamDefaultFields	
ParamNameMap	
util	This is actually a module with several functions available.

2.4.7 Lua Objects

This section discusses Lua objects that are specifically created for use in QuARTIC™.

2.4.7.1 Param Object

Param objects are the basic building blocks of symbolic calculation in QuARTIC™. When a parameters database is loaded from a set of parameters database files (Section 2.3), the parameter items specified in the parameters database are converted to Param objects which can then be used in equation blocks (Section 2.4.2.2) in the template file(s) (Section 2.4). As new parameters are calculated in equation blocks, these can also be used in calculations in the document.

Param may have any of the properties specified for the `parameter` command in Section 2.3. Not all of these are required and not all are meant for direct use in a document. Some derived properties are defined that are intended use instead. Consider the following example parameter with value '10.12345' (4 signifiant digits are kept):

$$P_i = 10.12 \text{ MPa} \tag{2.4.18}$$

Below is a list of the Param properties and methods with a description and usage example:

Table 2.9: Param Object Properties and Methods

Name	Usage	Result	Description
cite	`@[=P_i.cite]@`	Equation (2.4.18)	Provide a reference for the specified item. This may be a bibliography reference or an equation reference.
citeloc	`@[=P_i.citeloc]@`	Equation (2.4.18) on page 26	Provide a reference for the specified item with the location.
fmt	`@[ P_i.fmt = "%0.2f" ]@`		This sets the format of a parameter based on the Lua 'string.format()' command. If the format is not set explicitly, a default format will be used. The 'fmt' value is typically not printed in the document. This may also be used in an equation block after the variable has been assigned a value via an equation.
num	`@[=P_i.num]@`	10.12345	Get the numerical value of the param object without units. Since this is a numerical value, the value will not be formatted if 'P_i.num' is printed.
numstr	`@[=P_i.numstr]@`	10.123	Show the formatted numerical value without units.
save	`@[ P_i.save = true ]@`		Marker that indicates that a parameter will be used in an external file (i.e., a TRACE input file).
show	`@[=P_i.show]@`	$P_i = 10.123$ MPa	Show the variable and the associated value with units. Note that the value is automatically encapsulated in $ characters.

Table 2.9: Param Object Properties and Methods (continued)

Name	Usage	Result	Description
tblciteloc	`@[=P_i.tblciteloc]@`	Equation (2.4.18) on page 26	Display the citation and location in a table cell. Break the citation and location into separate lines (for compactness).
tex	`$@[=P_i.tex]@$`	P_i	Displays the latex representation of the variable. Note that $ is required around the value.
texstr	`@[=P_i.texstr]@`	P_i	Same as the 'tex' value, but automatically surrounds value with $.
units	`@[=P_i.units]@`	MPa	Show the param units. Parameters are not required to have units however.
val	`$@[=P_i.val]@$`	10.123 MPa	Get the units object (value and units). The $ equation marker must be included around parameters with units.
valstr	`@[=P_i.valstr]@`	10.123 MPa	Print the formatted value and units as a formatted string. $ is automatically added to the printed value.

2.4.7.2 Array Param Objects

In addition to representing single values, a Param object can represent a one or two dimensional array.

3 Debugging QuARTIC Template Errors

In order to effectively debug QuARTIC errors, it is useful to understand how QuARTIC generates a document. When QuARTIC is run to process a template file, a few steps occur:

1. QuARTIC processes the template file and converts it to a Lua script file. The name of the template file, and the location of the template file folder are specified via the command line. The Lua script file that is generated is then saved the the folder containing the QuARTIC scripts in a folder named 'tmp'.

2. QuARTIC then runs the Lua script to generate a lateX file that is ready for processing. This is placed in the output folder specified on the command line, and has the same name as the original template file.

3. At this point QuARTIC exits, and latex is called to process the file that was generated.

Errors are reported during stages 2 and 3. Template syntax errors, which can include things like using invalid Lua code, or fogetting to close a script block, are not generally noticed by the QuARTIC compiler, but will typically result in as error that is reported when the QuARTIC tries to run the Lua script file, or alternatively when latex is executed to generate the PDF document.

Errors that are reported by QuARTIC occur before latex is called, and latex will not be run if QuARTIC reports an error. QuARTIC will indicate a line number where the error occurred. The line number is a line number relative to the Lua script file that is generated by QuARTIC.

Latex errors are reported by the latex program that is used to process the latex file. Latex also reports line numbers for errors. In this case the line numbers are associated with the final latex document that is produced, and NOT the template file.

When fixing errors reported by QuARTIC, it is useful to have the template file and the Lua script file open. When fixing errors reported by latex, it is useful to have both the template file and the final generated latex file open for examination. Since the line numbers reported in the error are relative to the Lua script file or the final generated latex file, these files are useful to determining where in the template file to look to fix the problem. NOTE that errors MUST be fixed in the QuARTIC template file. Any changes to the Lua script file, or the generated latex file will be overwritten when QuARTIC is executed!

In general, when looking for errors reported in the Lua script file or in the latex file, it is useful to look at the line reported as well as at the few lines preceeding the error line. Often the error actually occurs on the line before the line reported. Note that both the Lua script file and the final latex contain text from the template file. The text surrounding the error is useful for determining the location of the error in the QuARTIC template file, so that the error can be located and fixed

in the template file.

Some knowledge of Lua syntax and latex syntax is useful when identifying errors. However, it is often sufficient to look for differences from other similar constructs in the script or generated latex document. Note that it is sometimes easier to spot errors in the QuARTIC template file. Other times, it is easier to spot errors in the Lua script file or the generated latex file.

3.1 Example of Fixing Error Reported by Lua Script

In the example below, a typo occurred where the function 'References' got replaced with the a call to a nonexistent function 's'. The example template code is shown below:

```
%===============================================================================
\chapter{Forces Description}
%===============================================================================

%===============================================================================
@[=? s() ]@
%===============================================================================
```

When this is executed QuARTIC reports the following error (note that latex does not run because QuARTIC failed to create the generated latex document):

```
----------------------
Generating Latex Files
----------------------
Writing '../../CalcNotebook/latex/HydroLoadsCalcNotebook.tex'
./tmp/HydroLoadsCalcNotebook.lua:1637: attempt to call global 's' (a table value)
```

The error is reporting that 's' is being 'called' as if it is a function, but it is not a function. Since an invalid function call is being made and QuARTIC fails. The contents of 'HydroLoadsCalcNotebook.lua' is shown with line numbers.

```
1628  _ENV:_write_([[
1629
    %=============================================================================
1630  \chapter{Forces Description}
1631
    %=============================================================================
1632
1633
1634
    %=============================================================================
1635  ]])
1636  _ENV:_write_(function() return
1637  s()  end)
```

On line 1637 we see 's()', which is the Lua syntax for calling a function named 's'. The lua code file has similarity to the template file, with some extra text added. This file generally gives enough context for you to do a text search in the QuARTIC template file to find where the error is occurring. In this case, there is no syntax error to fix - the function that is being called simply doesn't exist. At this point, you may recognize that something changed that you did not intend. Or maybe the function name is spelled wrong. If you don't recognize the change, assuming a source control tool such as SVN is being used to track changes to the file (which is highly recommended), it is useful to look and see if an unexpected change occurred in this location in the file. Fixing the error may be a simple matter of reverting the change to the file.

3.2 Example of Fixing Error Reported by Latex

After running QuARTIC, latex is used to process the file. The version of latex that QuARTIC has been tested with is lualatex. Latex prints a lot of information when processing files. Many things might look like errors, but typically most of the information is unimportant. It is usually just the final few lines that are important. Below is an example of an error reported by lualatex:

```
Underfull \vbox (badness 3525) has occurred while \output is active [9]
Underfull \vbox (badness 1496) has occurred while \output is active
    [10<./Figures/FigA6CrosbyLoopSeal.jpg>]
./HydroLoadsCalcNotebook.tex:500: Missing $ inserted.
<inserted text>
$
l.500

?
```

The '?' at the end is a latex prompt asking what you would like to do. Often you don't need to do anything other than close the window after you are done locating the error information. A couple lines above the '?' is 'l.500'. This indicates that the error was detected at line 500. Often the error

actually occurs on the line before this. A couple of lines above this is a message that indicates the name of the file (HydroLocaCalcNotebook.tex) and the line number (500) where the error occurred, followed by a description of the error. The error indicates that a '$' is missing. In latex, $ are used around inline math expressions. And this error may indicate that there is an unexpected $ added to the latex code.

```
496  \begin{align}
497  DX_{5.c4} = \frac{R_{SW663.and.665}}{2} = \frac{0.2286~\mathrm{m}}{2} =
    0.1143~\mathrm{m} \label{eq:DX_5_c4}
498  \end{align}
499  $$@
500
```

Here we see an extra '$$@' symbol. These are the symbols used to close a symbolic equation is QuARTIC (see Section 2.4.2.2), so it is probable that there was an error in the syntax used to add an equation. To figure out where to look in the template document, you can look for some surrounding text. Alternatively, you can search for the variable associated with `DX_{5.c4}`. Note that QuARTIC converts variable names used in the template file to names that are valid in latex. After the first underscore, it replaces underscores with dots. All QuARTIC variable names must have valid Lua variable names. Letters, numbers, and underscores are allowed, but no other symbols. Curly braces are used a lot in latex, so the first thing to do to recover the variable name is eliminate curly braces (i.e., `DX_5.c4`). As noted above, the dot is replaced with an underscore resulting in the QuARTIC variable name `DX_5_c4`. This is the variable name to search for in the template file. Search for this finds the following:

```
402  @$$>1
403  DX_5_c4 =  R_SW663_and_665/2 $$@
404  $$@
```

Note that there is an end of equation tag at the end of line 403 and there is also an end of equation tag on line 404. This is a scripting error. After removing the end of equation tag from line 403, the equation is formatted correctly.