CHAPTER 8

Scripting

Scripting provides a way to control the operation of HEC-DSSVue in a non-interactive way. The user can build and save scripts to be executed later - possibly on different data sets.

This chapter provides an introduction to scripting, describing the components of the user interface, scripting language and application program interface (API), and offering examples illustrating how to use the API.

8.1 Executing Scripts

HEC-DSSVue allows the execution of scripts in interactive and batch modes. Scripts are executed interactively by starting the HEC-DSSVue program and selecting the desired script from the Toolbar or the Script Selector from the **Utilities** menu. Scripts are executed in batch mode by starting the HEC-DSSVue program with a script file name as a parameter (e.g. *HEC-DSSVue.exe c:\test\myScript.py*). Scripts can be launched from the HEC-DSSVue program from the **Script Menu**, or from the **Toolbar** or from the **Script Selector**.

Interactive scripts are not passed any parameters upon script execution. In a script executed interactively the variable *sys.argv* is a list of length one, with the only element set to the empty string (e.g., *sys.argv = [" "]*).

Scripts executed in batch mode may take parameters from the command line (e.g., *HEC-DSSVue.exe c:\test\myScript.py a b c*). In a script executed in batch mode the variable *sys.argv* is a list whose length is one greater than the number of parameters passed on the command line, with the first element set to the file name of the executing script and the remaining elements set to the parameters (e.g., *sys.argv = ["c:\\test\ \myScript.py", "a", "b", "c"]*).

8.1.1 Script Menu

The **Script Menu** (see Figure 8.1, page 8-2) can display a list of available scripts and execute a script by selecting it from the menu. The **Script**

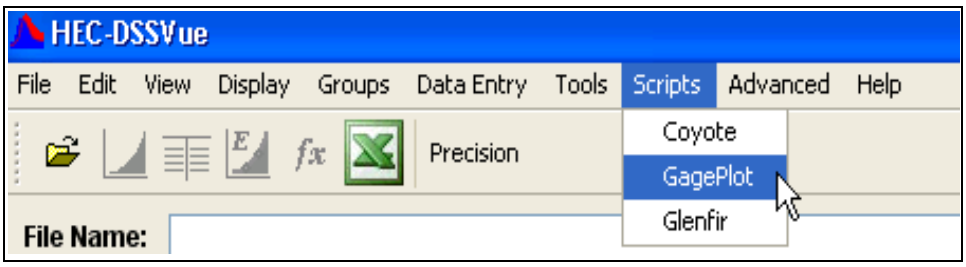

Figure 8.1 Running Scripts from the Script Menu

Menu, shown in Figure 8.1, is not made visible on the main HEC-DSSVue menu bar until scripts are set to be displayed on the menu bar from the **Script Editor**, available from the **Tools** menu. To have a script displayed in the **Script Menu**, simply edit the script in the **Script Editor**, select the **Show in Scripts Menu** checkbox at the top of the editor (Figure 8.2).

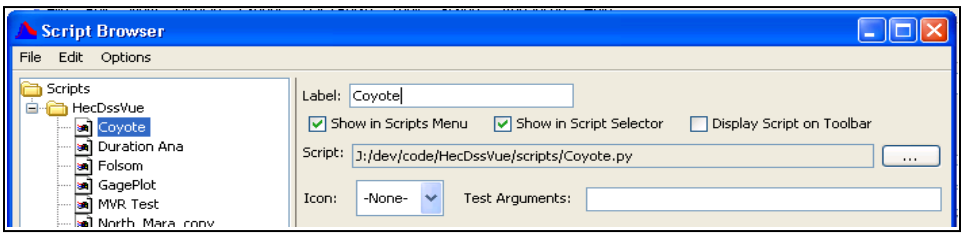

Figure 8.2 Show in Scripts Menu Checkbox in Script Editor

8.1.2 Main Toolbar

The main toolbar can display buttons for all the available scripts which have the **Display Script on Toolbar** box checked in the Script Editor (Figure 8.2). Buttons are displayed in alphabetical order. After selecting the check box, the menu bar will look like it is shown in Figure 8.3 below.

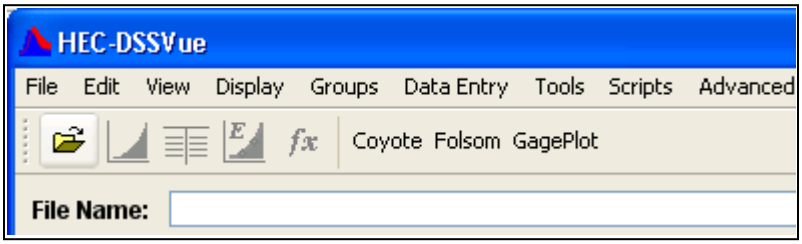

Figure 8.3 Script Menu Bar

8.1.3 Script Selector

The **Script Selector** (see Figure 8.4, page 8-3) displays buttons for all the available scripts which have the **Display Script on Toolbar** box checked in the **Script Editor** (Figure 8.2). Buttons are displayed in alphabetical order.

To access the **Script Selector**, select the **Script Selector** command from the **Utilities** menu of HEC-DSSVue. Once the **Script Selector** is open, it will remain open until you close it.

When you press a button, the Jython script engine will execute the associated script.

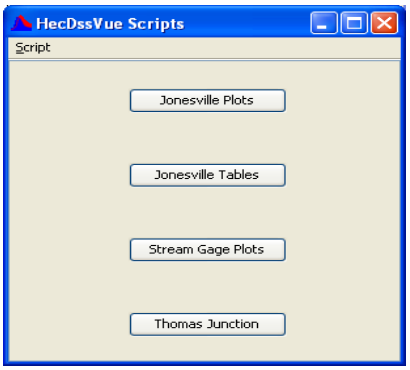

Figure 8.4 Script Selector

8.2 Script Editor

The **Script Editor**, shown in Figure 8.5, allows you to add, delete, and modify scripts.

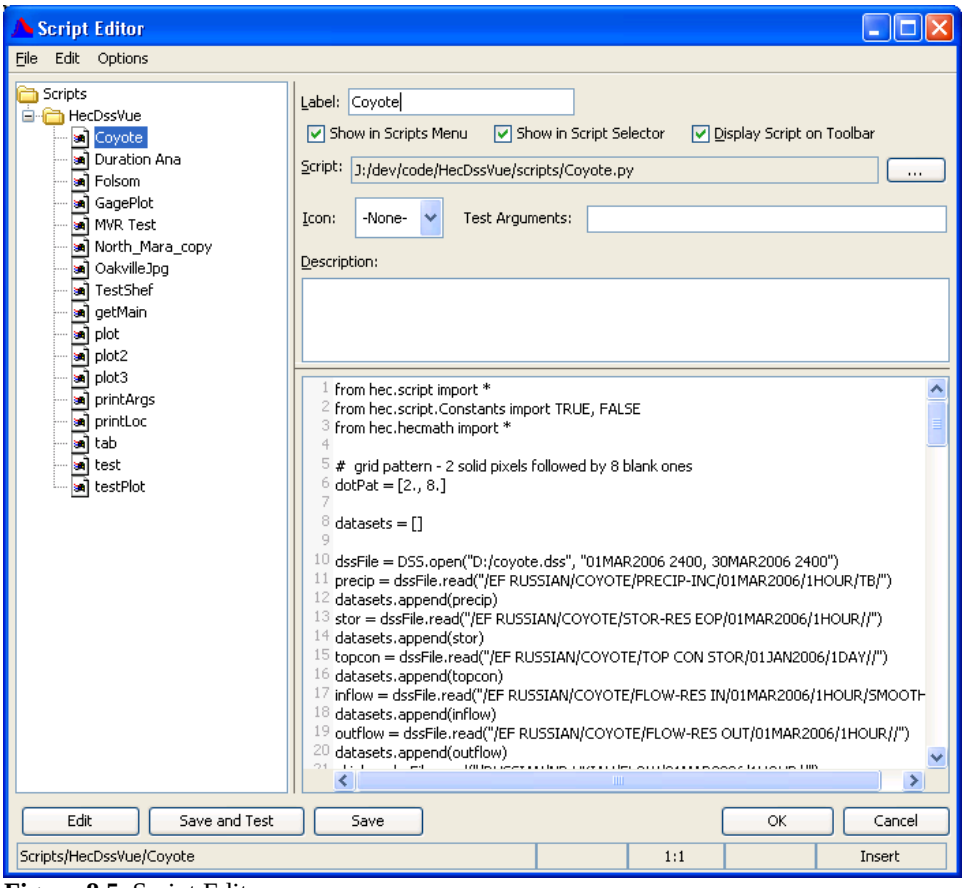

Figure 8.5 Script Editor

You can access the **Script Editor** from HEC-DSSVue's **Utilities** menu by clicking **Script Editor**. Alternatively, you can get to the **Script Editor** from the shortcut menu of the **Script Selector** (Figure 8.4). In the **Script Selector**, right click on a button to access the shortcut menu, and then select **Edit**. The **Script Browser** will open with that script selected and ready for editing.

Components of the **Script Editor** include the **Menu Bar**, the **Editor Panel**, and the **Tree Hierarchy**. The following sections describe these components.

8.2.1 Menu Bar

The **Menu Bar** (Figure 8.6) contains three primary menu items, **File**, **Edit**, and **Options**.

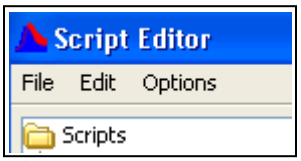

Figure 8.6 Menu Bar

File Menu Commands

New	Creates a new script stored at the currently selected position. Available only when a folder node is the selected node in the scripts tree.
Open	Edits the currently displayed script. Double clicking on the script also edits the currently displayed script. Available only when a script node is the selected node in the scripts tree.
Import	Imports a file into the script browser. If the import is successful the browser is placed in edit mode. Available only when a folder node is the selected node in the scripts tree.
Save	Saves the current script. Available only when a script is being edited.
Save As	Saves the current script, allowing the user to change the label. Note that the name of the script file does not change. Available only when a script is being edited.
Delete	Deletes the currently opened script. Prompts user for confirmation. Available only when a script node is the selected node in the scripts tree.
Test	Executes the currently selected script.
Close	Closes the **Script Editor**.

Edit Menu Commands

Cut Script	Cuts the script at the currently selected tree node to the system clipboard. Available only when a script node is selected in the tree view.
Copy Script	Copies the script at the currently selected tree node to the system clipboard. Available only when a script node is selected in the tree view.
Paste Script	Pastes the script in the system clipboard to the currently selected tree node. Available only when a folder node is selected in the tree view.

Option Menu Commands

Set Font	Opens a dialog box that allows setting of the font used in the script text area. Fixed-space fonts such as "Courier New" and "Lucida Console" are recommended over proportional fonts.
Set Tab Size	Opens a dialog box that allows setting the number of spaces that will be displayed in the script text area for each tab character.

8.2.2 Editor Panel

You can select and edit scripts in the **Editor Panel** (Figure 8.7) of the **Script Editor**.

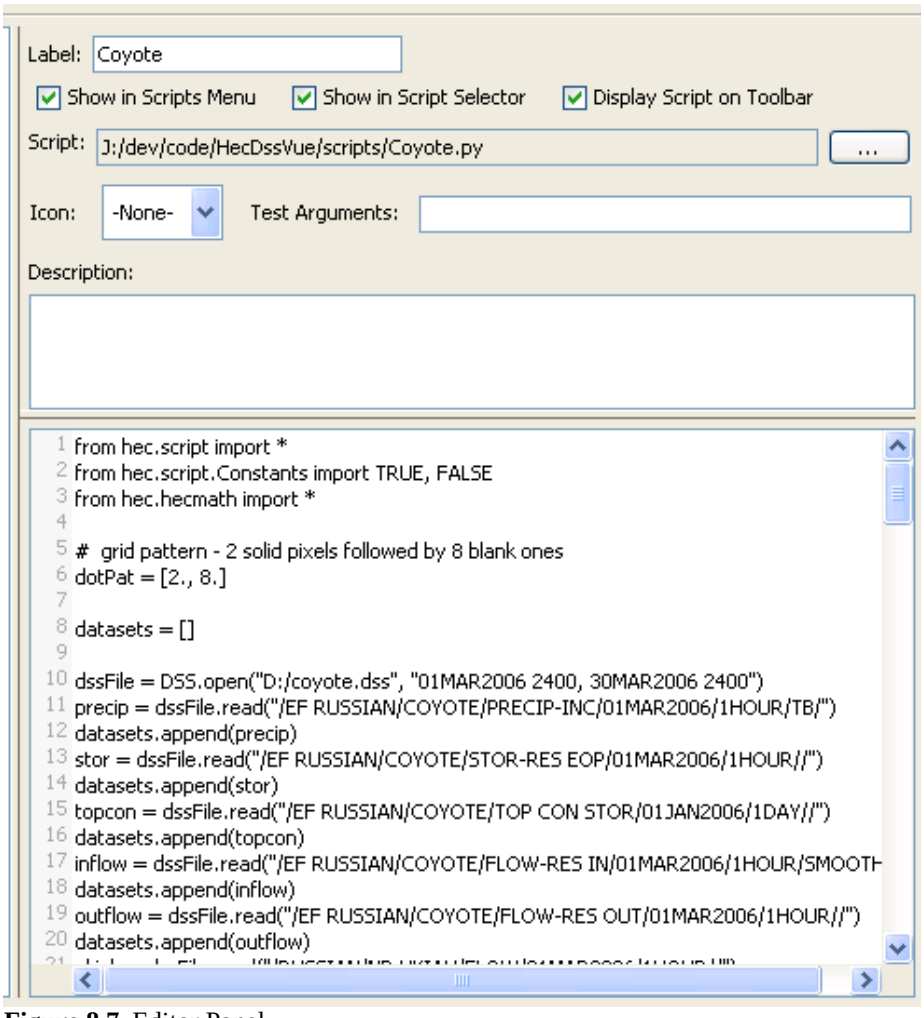

Figure 8.7 Editor Panel

The **Label** field allows you to specify the label displayed on a script's button in the **Script Selector**.

Script displays the name of the file in which the script is stored.

Selecting **Show in Scripts Menu** causes the script todisplay in the **Scripts** menu.

Selecting **Show in Script Selector** causes the script to be displayed on the **Script Selector**.

Selecting **Display Script on Toolbar** causes the script to be displayed on the **Toolbar**.

The **Icon** field allows you to choose the Icon to display for the script's button. If you do not select an icon, the script name displays in the script's button.

The **Description** field allows you to add a description of the script. The first line of your description serves as a tooltip for the corresponding button on the **Script Selector** and **Toolbar**.

The **Script Text** field contains the script text itself and serves as an editing window for creating new scripts.

The script text field has a context menu that can be accessed by right-clicking in the script text field.

Script Text Field Context Menu Commands

Cut	Copies the currently-selected script text to the system clipboard and removes it from the script.
Copy	Copies the currently-selected script text to the system clipboard and leaves it in the current script.
Paste	Copies text in the system clipboard into the script at the current cursor location.
Select All	Selects all the text in the script.
Find	Opens a dialog that allows the user to search for specific text in the script. If text is currently selected in the script, the dialog is initialized with this text.
Find Next	Locates the next text in the script that matches the conditions of the most recently executed find command.
Find Previous	Locates the previous text in the script that matches the conditions of the most recently executed find command.
Goto Line	Opens a dialog that allows the user to cause the cursor to jump to the beginning of a specified line in the script text.

8.2.3 Tree Hierarchy

The **Tree Hierarchy**, as shown in Figure 8.8, uses a Windows Explorer-style tree structure to allow you to navigate folders in your directory structure and access scripts. Scripts are stored in a "*scripts*" directory under the directory where HEC-DSSVue was installed.

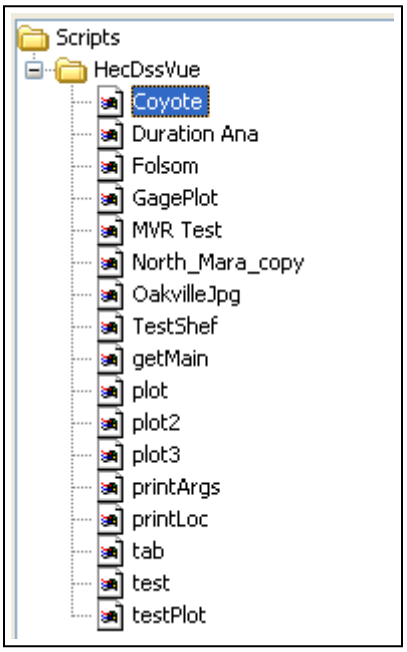

Figure 8.8 Tree Hierarchy

The **Tree Hierarchy** also has a context menu that displays **Cut Script**, **Copy Script**, and **Edit Script** commands for script nodes and **New Script**, **Import Script**, and **Paste Script** for folder nodes. These commands cause the same actions as the **File** and **Edit** menu commands discussed above.

To access the context menu, point to a node in the "tree" and right-click with your mouse.

8.3 Scripting Basics

Scripting in HEC-DSSVue is accomplished using Jython, an implementation of the Python programming language designed specifically for integration with the Java programming language. More information about Jython can be found at the official Jython website – www.jython.org.

Python (of which Jython is an implementation) is an interpreted language with simple syntax and high-level data types. This section is not a comprehensive Python tutorial, but rather a simple primer to allow the creation of simple scripts. This primer does not cover defining classes in Python.

The official Python website - www.python.org - has links to online Python tutorials as well as programming books.

8.3.1 Outputting Text

Text information can be displayed in the console window using the print statement which has the syntax:

print [*item*[, *item*…]]

The *item*s are comma-separated and do not need to be of the same type. The print statement will insert a space between every specified item in the output.

Example 1: Outputting Text

```
print “Testing myFunction, i =”, i, “, x =”, x
```

8.3.2 Data Types and Variables

Python has Boolean, integer, long integer, floating-point, imaginary number, and sequence and dictionary data types. Sequences are divided into mutable (or changeable) sequences called lists, immutable sequences called **tuples**. Tuples may be enclosed in parentheses or may be "naked". Strings are special tuples of characters that have their own syntax. Dictionaries are like sequences but are indexed by non-numeric values. In addition, Python also has a special type called **None**, which is used to indicate the absence of any value. The only valid values for the Boolean type are True and False. In certain circumstances these values also evaluate to the integers 1 and 0, respectively.

Python variable names consist of an upper- or lower-case letter or the “_” (underscore) character followed by an unlimited number of upper- or lower-case characters, digits or underscore characters.

Like many languages, Python uses the single equals sign, "=", to assign values to variables. However, Python allows assignment to multiple variables from a single sequence, as long as the number of variables on the left of the equal sign is the same as the length of the sequence on the right side. If the sequence on the right side is a dictionary, only the dictionary keys are assigned to the variables on the left.

There are situations regarding the HEC-DSSVue API where it is necessary or desirable to set a time-series value to "missing" or to test whether a time-series value is missing. The *hec.script* module supplies a constant to use in these situations.

The currently-defined constants in the *hec.script* module are:

Constant	Type	Represents
Constants.TRUE	integer	True
Constants.FALSE	integer	False
Constants.UNDEFINED	floating-point	missing data value

Example 2: Variable Types

```
# set some integer values
i = 0
j = 1
k = -10998
m = True

# set a long integer
n = 7922816251426433759354395033 6L

# set some floating-point values
x = 9.375
y = 6.023e23
z = -7.2e-3
t = Constants.UNDEFINED

# set some strings
string_1 = “abc”
string_2 = ‘xyz’
string_3 = “he said \“I won’t!\””
string_4 = ‘he said “I will not!”‘
string_5 = “””this is a
            multi-line string”””

# set a tuple – tuples are contained within ()
tuple_1 = (1, 2, “abc”, x, None)

# set a list – lists are contained within []
list_1 = [1, 2, “abc”, x, tuple_1]

# set a dictionary, using key : value syntax
# dictionaries are contained within {}
dict_1 = {“color” : “red”, “size” : 10, “list” : [1, 5, 8]}

# multiple assignment
a, b, c = string_1
a, b, c = “abc”
a, b, c, d, e = tuple_1
a, b, c, d, e = (1, 2, “abc”, x, None)
a, b, c, d, e = 1, 2, “abc”, x, None     # “Naked” tuple assignment
a, b, c, d, e = list_1
a, b, c, d, e = [1, 2, “abc”, x, tuple_1]
a, b, c = dict_1
a, b, c = {“color” : “red”, “size” : 10, “list” : [1, 5, 8]}
```

Indexing into sequence types is done using [*i*] where *i* starts at 0 for the first element. Subsets of sequence types (called slices) are accessed using [*i:j:k*] where *i* is the first element in the subset, *j* is the element **after** the last element in the subset, and *k* is the step size. If negative numbers are used to specify and index or slice, the index is applied to the **end** of the sequence, where [-1] specifies the last element, [-2] the next-to last and so on. If *k* is omitted in slice syntax it defaults to 1. If *i* is omitted in slice syntax it defaults to 0. If *j* is omitted in slice syntax it defaults to the length of the sequence, so list_1 [0:len(list_1)] is the same as list_1[:]. Indexing into dictionaries is done using [*x*] where *x* is the key.

The number of elements in a sequence type or dictionary is returned by the len() function.

Example 3: Sequence Indexing and Slicing

```
string_4[3]            # 4th element
string_4[3:5]          # 4th & 5th elements
list_1[0::2]           # even elements
list_1[1::2]           # odd elements
list_1[-1]             # last element
list_1[2:-1]           # 3rd through next-to-last element
list_1[2:len(list_1)]  # 3rd through last element (also list_1[2:])
dict_1["size"]         # value associated with "size" key
i = len(list_1)        # length of list_1
```

8.3.3 Operators

Each of the following operators can be used in the form a = b operator c. Each can also be used as an assignment operator in the form a operator= b (e.g., *a += 1*, *x **= 2*).

+	arithmetic addition
-	negation or arithmetic subtraction
*	arithmetic multiplication
/	arithmetic division
//	integer arithmetic division
**	arithmetic power
%	arithmetic modulo
&	bit-wise and
|	bit-wise or
~	bit-wise not
^	bit-wise xor (exclusive or)
<<	bit-wise left shift
>>	bit-wise right shift

Each of the following operators returns True or False and can be used in conditional expressions as discussed in Section 8.3.6.

>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
!=	not equal to
==	equal to

8.3.4 Comments

Python uses the "#" (hash) character to indicate comments. Everything from the "#" character to the end of the line is ignored by the interpreter. Comments may not be placed after a program line continuation ("\") character on the same input line.

8.3.5 Program Lines

Unless otherwise indicated, every input line corresponds to one logical program statement. Two or more statements can be combined on line input line by inserting the ";" (semicolon) character between adjacent statements. A single statement may be continued across multiple input lines by ending each line with the "\" (back slash) character. Comments may not be placed after a program line continuation ("\") character on the same input line.

Example 4: Input vs. Program Lines

```
# multiple statements per line
r = 1; pi = 3.1415927; a = pi * r ** 2

# multiple lines per statement
a = \
  pi * \
  r ** 2
```

Input lines are grouped according to their function. Input lines forming the body of a conditional, loop, exception handler, or function or class definition must be grouped together. Input lines not in any of the construct comprise their own group. In Python, grouping of input lines is indicated by indentation. All lines of a group must be indented the same number of spaces. A horizontal tab character counts as eight spaces on most systems. In some Python documentation, a group of input lines is called a *suite*.

Example 5: Input Line Grouping

```
# this is the main script group
dist = x2 – x1
if dist > 100.:
    # this is the “if” conditional group
    y = dist / 2.
    z = y ** 2.
else :
    # this is the “else” conditional group
    y = dist.
    z = y ** 2. / 1.5
# back to main script group
q = y * z
```

8.3.6 Conditional Expressions

Conditional expressions have the form:

```
if [not] condition :
  if-group
[elif [not] condition :
  elif-group]
[else :
  else-group]
```

The ":" (colon) character must be placed after each condition. The condition in each test is an expression built from one or more simple conditions using the form:

```
simple-condition ( and | or ) [not] simple-condition
```

Parentheses can be used to group conditions. The simple-condition in each expression is either an expression using one of the conditional operators mentioned before or is of the form:

```
item [not] in sequence
```

If the statement group to be processed upon a condition is a single statement, that statement may follow the condition on the same line (after the colon character).

Example 6: Conditional Expressions

```
if (x < y or y >= z) and string_1.index("debug") != -1 :
   # do something
   …
elif z not in value_list or (x < z * 2.5) :
   # do something different
   …
else :
   # do something else
```

Example 7: Simple Conditional Expressions

```
if x1 < x2 : xMax = x2
else       : xMax = x1
```

8.3.7 Looping

Python supports conditional looping and iterative looping. For each type, the body of the loop (the loop-group) can contain **break** and/or **continue** statements.

The **break** statement immediately halts execution of the loop-group and transfers control to the statement immediately following the loop.

The **continue** statement skips the remainder of the current iteration of the loop-group and continues with the next iteration of the loop-group

Conditional Looping

Python supports conditional looping with the **while** statement, which has the form:

```
while condition :
    loop-group
```

Conditional looping executes the body of the loop (the loop-group) as long as the condition evaluates to true.

Example 8: Conditional Looping

```
# print the first 10 characters
string_1 = “this is a test string”
i = 0
while i < 10 :
   print string_1[i]
   i += 1
```

Iterative Looping

Python supports iterative looping with the **for** statement, which has the form:

for *item* in *sequence* :
   loop-group
[else :
   else-group]

Iterative looping executes the body of the loop (the loop-group) once for each element in *sequence*, first setting *item* to be that element. If the iteration proceeds to completion without being interrupted by a break statement the else-group will be executed, if specified.

Example 9: Iterative Looping

```
# print the first 10 characters
string_1 = “this is a test string”
for i in range(10) :
   print string_1[i]

# print all the characters
string_1 = “this is a test string”
for i in range(len(string_1)) :
   print string_1[i]

# print all the characters (more Python-y)
string_1 = “this is a test string”
for c in string_1 :
   print c
```

The **range([*start*,] *stop*[, *increment*])** helper function generates a sequence of numbers from *start* (default = 0) to *stop*, incrementing by *increment* (default = 1). Thus range(4) generates the sequence (0, 1, 2, 3).

8.3.8 Defining and Using Functions

In Python, functions are defined with the syntax:

```
def functionName([arguments]) :
    function-body
```

Function names follow the same naming convention as variable names specified in Section 8.3.2. The arguments are specified as a comma-delimited list of variable names that will be used within the function-body. These variables will be positionally assigned the values used in the function call. More complex methods of specifying function arguments are specified in Python tutorials and references listed at the official Python website (www.python.org).

A function must be defined in a Python program before it can be called. Therefore, function definitions must occur earlier in the program than calls to those functions.

A function may optionally return a value of any type or a naked tuple of values.

Example 10: Defining and Using Functions

```
def printString(stringToPrint) :
  "Prints a tag plus the supplied string"
  tag = "function printString : "
  print tag + stringToPrint

def addString(string_1, string_2) :
  "Concatenates 2 strings and returns the result"
  concatenatedString = string_1 + string_2
  return concatenatedString

testString = "this is a test"
printString(testString)
wholeString = addString("part1:", "part2")
printString(wholeString)
printString(addString("this is ", "another test"))
```

8.3.9 Modules, Functions and Methods

A function is a procedure which takes zero or more parameters, performs some action, optionally modifies one or more of the parameters and optionally returns a value or sequence of values.

A class is the definition of a type of object. All objects of that type (class) have the same definition and thus have the same attributes and behavior. Classes may define functions that apply to individual objects of that type. These functions are called methods.

An object is an instance of a class, which behaves in the way defined by the class, and has any methods defined by the class.

Python provides many functions and classes by default. In our examples, we have used functions **len()** and **range()** which Python provides by default. We have also used the classes list and string, which Python also provides by default. We didn't use any methods of the class list, but we used the string method **index()** in Example 7 (*string_1.index("debug") != -1*). It is important to note that the object method **index()** doesn't apply to the string class in general, but to the specific string object string_1.

There are other functions and classes which Python does not provide by default. These functions and classes are grouped into modules according to their common purpose. Examples of modules are "os" for operating system functions and "socket" for socket-based network functions. Before any of the functions or classes in a module can be accessed, the module must be imported with the import statement, which has the syntax:

```
from module import *
```

Other methods of using the import statement are specified in Python tutorials and references listed at the official Python website (www.python.org). In the Jython implementation, Java packages can be imported as if they were Python modules, and the Java package *java.lang* is imported by default.

Example 11: Using a Function from an Imported Module

```
# use the getcwd() function in the os module to get
# the current working directory

from os import *
cwd = getcwd()
```

A module *does not* have to be imported in order to work with objects of a class defined in that module *if* that object was returned by a function or method already accessible. For example, the Python module "string" does not have to be imported to call methods of string objects, but *does* have to be imported to access string functions.

8.3.10 Handling Exceptions

Certain errors within a Python program can cause Python to raise an exception. An exception that is not handled by the program will cause the program to display error information in the console window and halt the program.

Python provides structured exception handling with the following constructs:

```
try :
  try-group
except :
  except-group
[else :
  else-group]

try :
  try-group
finally :
  finally-group
```

In the try-except-else construct, if an exception is raised during execution of the try-group, the control immediately transfers to the first line of the except-group. If no exception is raised during execution of the try-group, the control transfers to the first line of the else-group, if present. If there is no exception raised and no else-group is specified, the control transfers to the first line after the except-group.

The two constructs cannot be combined into a try-except-finally construct, but the same effect can be obtained by making a try-except-else construct the try-group of a try-finally construct.

Example 12: Exception Handling

```
try :
   try :
      string_1.find(substring) # may raise an exception
   except :
      print substring + " is not in " + string_1
      # do some stuff that might raise another exception
      …
   else :
      print substring + " is in " + string_1
      # do some stuff that might raise another exception
      …
finally :
   print "No matter what, we get here!"
```

More exception handling information, including filtering on specific types of exceptions, exception handler chains, and raising exceptions, is provided in Python tutorials and references listed at the official Python website (www.python.org). In the Jython implementation, instances or

subclasses of *java.lang.Throwable* can also be handled as exceptions. Detailed information can be found at the Java API website (http://java.sun.com/javase/6/docs/api/).

8.4 Displaying Messages

It is often useful to display messages to inform the user that something has occurred, to have the user answer a Yes/No question, or offer debugging information to help determine why a script isn't working as expected. Text information can be displayed in the console window as described in Section 8.3.1.

8.4.1 Displaying Message Dialogs

The **MessageBox** class in the *hec.script* module, and the **MessageDialog** class in the **MessageDialog** module have several functions used to display messages in message box dialogs. The dialogs can be one of four different types: Error, Warning, Informational or Plain. The difference between the **MessageBox** functions and their **MessageDialog** counterparts is the **MessageBox** versions are modal, while the **MessageDialog** versions are modeless (see Section 8.5.4).

MessageBox and **MessageDialog** functions with multiple buttons return a string containing the text of the button that was selected to dismiss the message box.

Note: Do not use the **MessageBox** or **MessageDialog** functions in a script that is to run unattended since these functions cause scripts to pause for user interaction. **MessageDialog** functions should only be used in batch mode scripts (see Section 8.5.3).

MessageBox and **MesssageDialog** functions are described in Table 8.1.

Example 13: Display Error Dialog with MessageBox class

```
from hec.script import *

MessageBox.showError("An Error Occurred", "Error")
```

Example 14: Display OK/Cancel Dialog with MessageDialog class

```
from hec.script import *
import MessageDialog
   ok=MessageDialog.showOkCancel("Continue with Operation", "Confirm")
```

Table 8.1 MessageBox and MessageDialog Functions

Function	Returns	Comments
showError(string message, string title)	None	Display an error dialog to you with the message and title
showWarning(string message, string title)	None	Display a warning dialog to you with the message and title
showInformation(string message, string title)	None	Display a Informational dialog to you with the message and title
showPlain(string message, string title)	None	Display a plain dialog to you with the message and title
showYesNo(string message, string title)	string	Display a Yes/No dialog to you with the message and title
showYesNoCancel(string message, string title)	string	Display a Yes/No/Cancel dialog to you with the message and title
showOkCancel(string message, string title)	string	Display a Ok/Cancel dialog to you with the message and title

8.5 Headless Operation

Often it is desirable to run scripts in the "background" so that you do not see the interactions or tables or graphics, but instead have the appropriate functions executed or *.png* files created. Most math scripts do not write to the screen, so this is not an issue. If you call higher level functions, there is the potential that some dialogs may appear, especially if there are errors. The **ListSelection** class tries to figure out how you are running a script and will only print messages if you are running it in the background, but show message dialogs if your are running in the foreground. This means that you can get different message dialog behavior if you run a script from HEC-DSSVue that accesses **ListSelection** than if you run it in a batch mode.

8.5.1 Headless on Windows

On the PC, you can set the location of plots and tables off-screen so that they are not visible to the user, create the *.png* or *.jpg* file, and then close the plot or table. For example:

```
plot = Plot.newPlot()
plot.setSize(600, 500)
plot.setLocation(-10000, -10000)
```

```
…
plot.showPlot()
…
plot.saveToPng("C:/temp/myPlot.png")
plot.close()
```

8.5.2 Headless on UNIX

On UNIX, you must have an X-Window server on which to generate a plot or table. This is done using a Virtual X-Window server, such as Xvfb. The programs treat this as a normal X-Window display. Like the PC, you run the scripts as normal, although you don't need to set at location:

```
plot = Plot.newPlot()
plot.setSize(600, 500)
…
plot.showPlot()
…
plot.saveToPng("/tmp/myPlot.png")
plot.close()
```

For more information, refer to:
http://www.x.org/archive/X11R6.8.2/doc/Xvfb.1.html

On UNIX it is generally advantageous to use the same script to run in an interactive mode as well as a batch mode. This can be accomplished by checking to see if the DISPLAY environment variable is set. If it isn't, then check that Xvfb is running and set the DISPLAY variable to ":1.0". For example:

```
#!/bin/ksh
# ****************************************************************
if [ "$DISPLAY" = "" ]
then
   XvfbRunning=`ps -ef | grep Xvfb | grep -v grep | wc -l | sed -e "s/ //g"`
   if [ $XvfbRunning -gt 0 ]
   then
      DISPLAY=":1.0"
      export DISPLAY
      printf "\nWARNING : Using virtual display – “
      printf "no graphics will be visible.\n\n"
   else
      printf "\nWARNING : Virtual display server is not running."
      printf "\n          Set DISPLAY variable or start Xvfb.\n\n"
      return -1
   fi
fi
# ****************************************************************
# Execute the program here….
```

8.5.3 Interactive and Batch Mode Scripts

A script is said to be *interactive* if it is executed from the HEC-DSSVue menu bar, tool bar, **Script Selector**, or **Script Editor**. With any of these methods, the user interacts with the HEC-DSSVue GUI to lauch a script. Conversely, a script is said to be in *batch mode* if it is executed from a command prompt or shortcut as a parameter to the HEC-DSSVue program.

To run a script in batch mode, you specify the file name of the script as a parameter to HEC-DSSVue, following the program name; for example:

```
HEC-DSSVue.exe C:\HecDSSVueDev\HecDssVue\scripts\Oakville.py
```

In Windows, this can be passed through a *.bat* file and then you can run that .bat file; e.g., *RunOakville.bat*:

```
HEC-DSSVue.exe C:\HecDSSVueDev\HecDssVue\scripts\Oakville.py
```

On Windows, a shortcut can be utilized instead of a batch file. To create a shortcut for a script, copy the HEC-DSSVue shortcut and pass the script file name in following the *Target* in the Shortcuts Properties dialog box (Figure 8.9). Right click on the original HEC-DSSVue shortcut on the desktop and click **Copy**. Right click on an empty area on the desktop and click **Paste**. Right-click on the copy and Rename to an appropriate name (.e.g., your script name), right click again and click **Properties**. Enter the script name following the program name in the *Target* area, as shown in Figure 8.9.

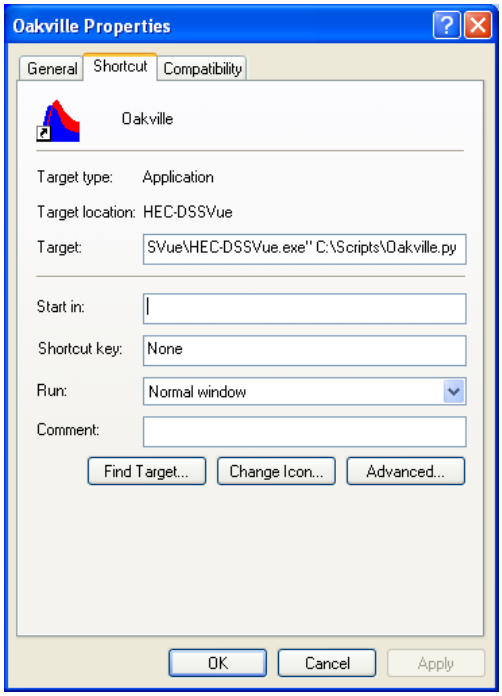

Figure 8.9 Shortcut Properties Dialog Box

In UNIX, a file is made executable by changing its mode to be executable, e.g., *RunOakville*:

```
HEC-DSSVue /usr1/HecDssVue/scripts/Oakville.py
chmod 555 RunOakville
```

If a batch mode script requires parameters, they must be appended to the command line (or target field in the shortcut) after the name of the script:

```
HEC-DSSVue.exe C:\HecDSSVueDev\HecDssVue\scripts\xyz.py a b c
```

A script may determine whether it is executing in interactive or batch mode by calling the *isInteractive()* **ListSelection** method in Table 8.3 (see page 8-23).

One characteristic of batch mode scripts is that they any GUI components (plot, tables, etc…) displayed during their execution are destroyed when the script completes. While this behavior is acceptable for generating files containing tables and plot images, it is often desirable for a batch mode script to display GUI components that persist until the user dismisses them. A batch mode script may always utilize one of the **MessageDialog** functions listed in Table 8.1. However, if a batch mode script uses one of the window types listed in Table 8.2, it can call the *stayOpen()* method of the window to gain persistence without the need for an extra dialog.

Table 8.2 Window Types with *stayOpen()* Method

Window
ListSelection
Plot (G2dDialog)
Table (HecDataTable)
CompareFiles
ExcelFrame
Math
SelectRecords
UndeleteRecords

The **stayOpen()** method causes the window to wait and stay open until the window is closed. Like the **MessageDialog** functions, the *stayOpen()* **cannot be used in an interactive script**. It will cause the program to freeze. Plots will stay open automatically when in an interactive mode. An example use for *Oakville.py* is:

```
#  Display the plot and wait for them to close
plot.stayOpen()    # Do not use in an interactive script
```

8.5.4 Modality in Scripts

Modality refers to whether an application window allows interaction with other windows of the same application. A **modal** window prevents interaction with other application windows as long as it is displayed. Once the modal window is dismissed, interaction with other application windows resumes. Conversely, **modeless** windows allow interaction with other application windows while they are displayed. Most HEC-DSSVue windows are modeless, although error dialogs and dialogs created with the **MessageBox** functions described inTable 8.1 are modal. **MessageDialog** functions described in Table 8.1 that are modeless, can not be executed in

interactive scripts unless they are called in a separate thread from the graphics display. Writing multi-threaded scripts is beyond the scope of this manual.

8.6 Accessing the Main Program Window

Situations arise that necessitate accessing the main HEC-DSSVue program window. Examples of such situations are:

- Having a script determine whether it is running in an interactive mode or without a graphical display.
- Having a script gather information about interactively-selected pathnames and time windows for automated processing.
- Having a script launch the interactive graphical editor.

The **ListSelection** class in the *hec.dssgui* module facilitates these activities. An import statement of the form "from *hec.dssgui* import ListSelection" is necessary to use the **ListSelection** class. (For CWMS-VUE, the import statement is "from *hec.cwmsVue* import CwmsListSelection").

If the script is executing in batch mode then a main program window will not exist. If the script needs to have the user utilize the graphical editor, the script will need to create a main program window for the duration of the graphical edit session.

8.6.1 ListSelection Class

The static functions of the **ListSelection** class are listed in Table 8.3.

Table 8.3 ListSelection Static Functions

Function	Returns	Description
createMainWindow()	ListSelection	Returns a new ListSelection object. This should not be called unless the script is not running in interactive mode.
getMainWindow()	ListSelection	Returns the main program window (ListSelection object) of the script. Returns None if the script is not running in interactive mode.
isInteractive()	Boolean	Returns if the program is being run interactively or in a batch mode.

At the end of a batch script that calls *createMainWindow()*, you should close DSS files and exit the **ListSelection** call using the *finish()* method. This function properly shuts down the program without altering any properties. For example:

```
mainWindow = ListSelection.createMainWindow()
…
mainWindow.finish()
```

Don't call *finish()* when running interactively (unless you want the program to terminate). **ListSelection** objects can be used to gather information about pathnames and time windows that the user selected interactively before launching the script. **ListSelection** objects can also be used to launch the interactive graphical editor. The **ListSelection** methods are listed in Table 8.4.

Table 8.4 ListSelection Methods

Method	Returns	Description
addToolBarButton(JButton b)	None	Adds a button to the main toolbar.
addToSelection(String path)	None	Adds a pathname to the selection list.
clear()	None	Clears selected pathnames.
close()	None	Close the current DSS file.
copyPathnamesToClipboard()	None	Copies selected pathnames to the clipboard
copyRecords(Boolean)	None	Copies selected records to the second DSS file opened if 0, or displays an open dialog
delete()	None	Deletes the selected records
duplicateRecords()	None	Duplicates the selected records
exitProgram()	None	Exits the program, saving settings
exportShef(String filename)	None	Exports the selected records to SHEF in the file given
exportShef(String filename, Vector timeSeriesContainers)	None	Exports the provided data to SHEF in the file given
exportToFile()	None	Exports images, etc. by opening a Open dialog
fileCheck()	None	Runs a file check on the opened DSS file
finish()	None	Exits without saving settings
getCustomMenu()	JMenu	Returns the custom menu
getDataManager()	CombinedData Manager	Returns the combined data manager for the current file
getDisplayMenu()	Jmenu	Returns the Display menu
getDSSFilename()	string	Returns the current DSS file name
getEndTime()	HecTime	Returns the current end time of the ListSelection as an HecTime object.[1]
getEditMenu()	Jmenu	Returns the Edit menu
getExport Menu()	Jmenu	Returns the Export menu
getFileMenu()	Jmenu	Returns the File menu
getImportMenu()	Jmenu	Returns the Import menu
getNumberSelectedPathnames()	integer	Returns the number of selected pathnames
getScriptMenu()	Jmenu	Returns the script menu
getSecondDataManager()	CombinedData Manager	Returns the combined data manager for the second file

Method	Returns	Description
getSelectedDataContainers()	List	Reads data for the selected pathnames and returns it in a list of data containers
getSelectionList()	list of DataReference objects	Returns a list of DataReference objects that represent the interactive pathname and time window selections.
getSelectedPaths()	Vector	Returns a Vector of Strings of pathnames in the open DSS file.
getStartTime()	HecTime	Returns the current start time of the ListSelection as an HecTime object.[1]
getTimeWindow(HecTime startTime, HecTime endTime)	Boolean	Returns whether the start time and end time provided are defined
getToolBar()	JtoolBar	Returns the tool bar
getViewMenu()	Jmenu	Returns the View menu
graphicalEdit()	None	Launches the graphical editor for the interactively-selected pathnames and time windows.
graphicalEdit(TimeSeriesContainer tsc)	None	Launches the graphical editor for the specified TimeSeriesContainer.
graphicalEdit(list tscList)	None	Launches the graphical editor for all TimeSeriesContainers in the list.
importGenericFiles()	None	Launches a file open dialog to import files
importImages()	None	Launches a file open dialog to import image files
importShef()	None	Launches a file open dialog to import SHEF files
importShef(string filename)	None	Imports the SHEF file
isDssFileAccessible()	Boolean	Returns if the DSS file set earlier is accessible currently
isRemote()	Boolean	Returns whether the DSS file opened is remote and accessed using DSSManager
isVisible()	Boolean	Returns whether the dialog is currently visible on the screen
math()	None	Reads the selected data and then opens the math dialog
mergeFiles(string toDssFilename)	None	Copies all the data from the opened DSS file into the one specified
openDSSFile (string DSSFileName)	Boolean	Opens the DSS file name of the ListSelection and returns whether it was opened or not
pairedDataEntry()	None	Opens the paried data entry dialog
plot()	None	Reads the selected data and then plots it
read(string pathname)	DataContainer	Reads the recorded specified by the pathname
read(string pathname, HecTime start, HecTime end)	DataContainer	Reads the recorded specified by the pathname and time window

Method	Returns	Description
read(string pathname, String timeWindow)	DataContainer	Reads the recorded specified by the pathname and time window
refreshCatalog()	None	Refreshes the catalog for the opened file
registerExportPlugin(JmenuItem menuItem)	None	Adds the menu item to the export menu
registerImportPlugin(ImportPlugin importPlugin, JmenuItem menuItem, string dropAndDragExtension)	None	Adds the menu item to the import menu and associates the plugin and drag and drop extension to that item
registerPlugin(integer menuNumber, JmenuItem menuItem)	None	Adds the menu item to main menus. Menu numbers are the following: FILE = 0; EDIT = 1; VIEW = 2; DISPLAY = 3; GROUPS = 4; DATA_ENTRY = 5; TOOLS = 6; CUSTOM = 7; SCRIPTS = 8; ADVANCED = 9; HELP = 10;
renameRecords()	None	Brings up the rename dialog for the selected pathnames
runFile()	None	Reads the selected data and runs the programs associated with them (e.g., *.pdf, .mp3*)
runFile(Vector pathnames)	None	Reads the selected pathnames and runs the programs associated with them (e.g., *.pdf, .mp3*)
save(DataContainer dataContainer)	Boolean	Saves the data container in the opened DSS file. Returns if success or not
saveAs(DataContainer dataContainer)	Boolean	Opens the Save As dialog and saves the data container in the opened DSS file. Returns if success or not
selectAll()	Boolean	Adds all pathnames to the selection list
setCustomMenu(string menuText)	None	Sets the text displayed on the menu bar
setDssLogFile(string logFile)	None	Writes DSS messages to the file provided
setScriptMenu(string menuText)	None	Sets the text displayed on the menu bar
setSelectedPathnames(Vector selectedPathnames)	None	Sets the selected list of pathnames
setTimeWindow(string timeWindow)	None	Sets the current time window of the ListSelection using a string value to define the start time and end time
setTimeWindow(string start, string end)	None	Sets the current time window of the ListSelection using a string value to define the start time and end time
setTimeWindow(HecTime start, HecTime end)	None	Sets the current time window of the ListSelection using an Hec Time object to define the start time and end time

Method	Returns	Description
setVisible(Boolean visible)	None	Sets whether the dialog is visible on the screen.
squeeze()	None	Squeezes the open DSS file
tabulate()	Jframe	Tabulates the selected data
tabularEdit()	Jframe	Displays the tabular editor for the selected data
textEntry()	None	Displays the text entry dialog
textFileEntry()	None	Displays an open dialog to read a text file into text format
timeSeriesDataEntry()	None	Displays the time series data entry dialog
undeleteAll()	None	Undeletes all deleted records in the DSS file
undeleteSelected()	None	Undeletes the selected records
updateCatalog()	None	Creates a new catalog, if needed (refreshCatalog forces a new catalog)
updateMessageField(string message)	None	Puts the message in the message field

[1] The current start and end times of a **ListSelectio**n object are not necessarily the same as the start and end times of any **DataReference** objects in the **ListSelection**. Rather they represent the start and end times that will be applied to subsequent selections.

8.6.2 DataReference Class

The *getSelectionList()* **ListSelection** method returns a Vector of DataReference objects that represent the interactive selections. Each individual selection may have a different DSS filename, pathname, and time window than other selections. The methods of the **DataReference** class are listed in Table 8.5.

Table 8.5 DataReference Methods

Method	Returns	Description
getFilename()	string	Returns the name of the DSS file from which this selection was made.
getNominalPathname()	string	Returns the nominal pathname for this selection. If a condensed set, this will not be a real pathname, but show the date span in the D part
getPathname()	string	Returns the pathname for this selection
getPathnameList()	Vector	Returns a list of the pathnames for this selection. This will be only more than one if a condensed reference
getTimeWindow(HecTime startTime, HecTime endTime)	Boolean	Sets the startTime and endTime parameters with the start time and end time of the selection, respectively. Returns True if a time window is defined for the selection. Returns False otherwise.
hasTimeWindow()	Boolean	Returns True if a time window is defined for the selection. Returns False otherwise.

8.7 Reading and Writing to HEC-DSS Files

Retrieving and storing data from a HEC-DSS file is done through the class **HecDss** in the *hec.heclib.dss* package. The older technique of using **DSSFile** objects returned from *DSS.open()* calls is still supported, but lacks the ability to operate directly with **DataContainer** objects. HecDss inherits from DSSFile in HecMath and retains all the functionality of that class for backwards compatibility. The first call that should be made is the static member *HecDss.open(string filename)*, which will return a **HecDss** object to retrieve and store data, as well as perform math and other functions.

You need to import the classes that you use as well as the hec.script package. To do this, include the follow lines at the beginning of your script:

```
from hec.script import *
from hec.heclib.dss import *
```

8.7.1 HecDss Class

```
from hec.heclib.dss import *

HecDss.open(string filename)
HecDss.open(string filename, string startTime, string
endTime)
HecDss.open(string filename, string timeWindow)
HecDss.open(string filename, Boolean fileMustExist)
```

The **HecDss** class is used to gain access to an HEC-DSS file, as illustrated in Example 15. You must import HecDss from *hec.heclib.dss*.

Example 15: Opening and Releasing a DSS File

```
from hec.heclib.dss import *

theFile = HecDss.open("MyFile.dss")
or
theFile = HecDss.open("C:/temp/sample.dss", 1)

#  Finished using the file – release it
theFile.done()
```

Using one of the methods that specifies a time window affects the operation of the *get()* and *read()* methods as described in Tables 8.6 and 8.8 of the **Pattern Strings** section.

Note: The preferred manner of releasing access to an HEC-DSS file in most cases is to use the *done()* method and not the *close()* method. This is

because HEC-DSS maintains information about file access and can grant access to a file released with *done()* much more quickly than it can to one released with *close()*. HEC-DSS will close files when a script exits or when they have not been accessed for a while. The only time that the *close()* method should be called explicity is to perform some operating system operation on it, such as renaming or deleting the file.

8.7.2 HecDss Retrieve and Store Functions

HecDss objects are used to retrieve and store data sets in a DSS file, as well as several utility functions for DSS files. Data is retrieved and stored using **DataContainers**. **DataContainers** are simple classes that are database independent and are how most **Hec** classes exchange data. **DataContainers** are a base class for several data types, such as time series data, paired data, gridded data, etc. **DataContainers** and their related classes are described following the **HecDss** class. If the data set you are retrieving does not exist, the **DataContainer** will have the *numberValues* set to 0 (zero).

HecDss, and most classes accessed through scripts, use Java Exceptions for error handling. Use a try block to catch and process errors. If data cannot be stored, **HecDss** will throw an exception. Refer to section 8.3.11 on **Handling Exceptions** for more information.

Tip: From the HEC-DSSVue main screen, select a pathname, right click and select **Copy Pathnames to Clipboard**. You can then paste the pathname into your script and avoid having to type it. You cannot use *condensed* pathnames for these functions.

The **HecDss** primary retrieve and storage methods are described in Table 8.6. Additional primary methods are shown in Table 8.7 and secondary methods are shown in Table 8.6. Time Series Data Storage Methods (Regular and Irregular) are shown in Table 8.9.

Pattern Strings

The pattern strings used with the getCatalogedPathnames(…) methods filter the list of returned pathnames in a user-specified manner. Pattern strings can be specified in two modes: pathname mode and pathname part mode. Both modes make use of pathname part filters:

Table 8.6 HecDss Retrieval and Storage Methods

Method	Returns	Description
get(String pathname)	DataContainer	Retrieves data for the single pathname and returns in a **DataContainer**. **DataContainer** will be a **TimeSeriesContainer**, **PairedDataContainter**, etc., depending on the data type. If a default time window has been set via one of the *open()* or *setTimeWindow()* methods, the "D" part of a time series pathname is ignored and the data for the time window is retrieved.
get(string pathname, Boolean getEntireDataSet)	DataContainer	Ignores the "D" part of time series data and retrieves data for the entire date span in the DSS file.
get(DSSIdentifier dssid)	DataContainer	Retrieves data for the pathname and time window given in the dssid
put(DataContainer dataContainer)	None	Store the data in **DataContainer** in DSS. The **DataContaine**r must be a **TimeSeriesContainer** or **PairedDataContainte**r, etc. Throws an exception if the data cannot be stored

Table 8.7 HecDss Additional Primary Methods

Method	Returns	Description
done()	None	Tells HEC-DSS that you are finished using it and releases the file. Use this instead of *close()*.
getPathnameList()	list	Returns a list of the pathnames in the file.
getTimeSeriesExtents(String pathname, HecTime start, HecTime end)	Boolean	Sets the contents of the start and end parameters to the date/time of the first and last piece of data in the entire data span. Returns whether the pathname exists.
isOpened ()	Boolean	Returns if the DSS file is accessible.
open(String dssFileName)	HecDss	Opens the DSS file and returns a **HecDss** object for further access. Creates the file if it does not exist. Throws an exception if the file cannot be created or opened.
open(String dssFileName, Boolean fileMustExist)	HecDss	Opens the DSS file and returns a **HecDss** object for further access. Throws an exception if you indicate the file must exist and it does not.
recordExists(String pathname)	Boolean	Tests to see if the single record exists.

Example 16: Reading from DSS

```
from hec.script import *
from hec.heclib.dss import *
import java

try :
  try :
    myDss = HecDss.open("C:/temp/sample.dss")
    flow = myDss.get("/RUSSIAN/NR UKIAH/FLOW/01MAR2006/1HOUR//")
    plot = Plot.newPlot("Russian River at Ukiah")
    plot.addData(flow)
    plot.showPlot()
  except Exception, e :
    MessageBox.showError(' '.join(e.args), "Python Error")
  except java.lang.Exception, e :
    MessageBox.showError(e.getMessage(), "Error")
finally :
  myDss.done()
```

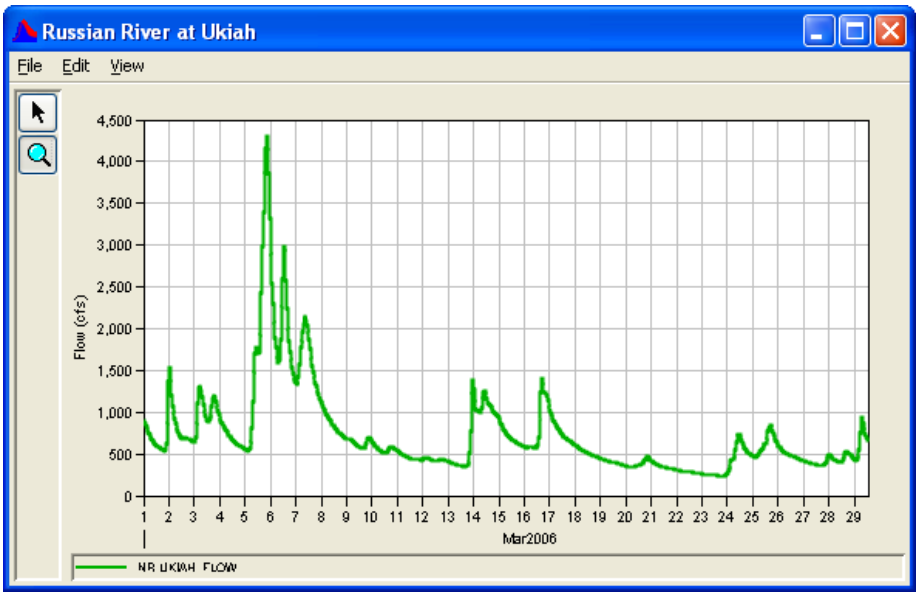

Figure 8.10 Example 16 Results

Table 8.8 HecDss Secondary Methods

Method	Returns	Description
close()	None	Close the DSS file. Only call this when you need the file closed (e.g. to rename it); otherwise call done()
copyRecordsFrom(String toDSSFilename, list of strings pathnameList)	integer Success = 0 Fail < 0	Copy records from the open DSS file to the DSS file name specified in the call.
copyRecordsFrom(String toDSSFilename, Vector pathnameList)	integer Success = 0 Fail < 0	Copy records from the open DSS file to the DSS file name specified in the call.
copyRecordsInto(String fromDSSFilename, list of strings pathnameList)	integer Success = 0 Fail < 0	Copy records from the file specified in the call into the currently open DSS file

Method	Returns	Description
copyRecordsInto(string fromDSSFilename, Vector pathnameList)	integer Success = 0 Fail < 0	Copy records from the file specified in the call into the currently open DSS file
delete(list of strings pathnameList)	integer Success = 0 Fail < 0	Deletes the records corresponding the list of pathnames
delete(Vector pathnameList)	integer Success = 0 Fail < 0	Deletes the records corresponding the list of pathnames
duplicateRecords(list of strings pathnameList, list of strings newPathnameList)	integer Success > 0 Fail < 0	Duplicates the records in the first list giving the names in the second list (same DSS file)
duplicateRecords(Vector pathnameList, Vector newPathnameList)	integer Success > 0 Fail < 0	Duplicates the records in the first list giving the names in the second list (same DSS file)
getCatalogedPathnames()	list of strings	Returns a list of all cataloged pathnames without generating a new catalog.
getCatalogedPathnames(Boolean forceNew)	list of strings	Returns a list of all cataloged pathnames, optionally generating a new catalog first.
getCatalogedPathnames(string pattern)	list of strings	Returns a list of all cataloged pathnames that match the specified pattern without generating a new catalog.
getCatalogedPathnames(string pattern, Boolean forceNew)	list of strings	Returns a list of all cataloged pathnames that match the specified pattern, optionally generating a new catalog first.
getDataManager()	CombinedDataManager	Gets the data manager for this DSS file
getEndTime()	string	Returns the end time set
getFileName()	string	Returns the name of the DSS file
getStartTime()	String	Returns the start time set
isRemote()	Boolean	Returns if the DSS file is being served on a remote machine (client/server mode)
read(string pathname)[1]	HecMath	Returns an **HecMath** object that holds the data set specified by pathname. If a default time window has been set via one of the *open()* or *setTimeWindow()* methods, the "D" part of a time series pathname is ignored and the data for the time window is retrieved.
read(string pathname, string timeWindow) [1]	HecMath	Returns an **HecMath** object that holds the data set specified by pathname with the specified time window.

Method	Returns	Description
read (string pathname, string startTime, string endTime) [1]	HecMath	Returns an HecMath object that holds the data set specified by pathname with the specified time window.
renameRecords (list of strings pathnameList, list of strings newPathnameList)	integer Success > 0 Fail < 0	Renames the records in the first list to those in the second list.
renameRecords (Vector pathnameList, Vector newPathnameList)	integer Success > 0 Fail < 0	Renames the records in the first list to those in the second list.
setIrregularStoreMethod (integer storeMethod)	None	See table following
setRegularStoreMethod (integer storeMethod)	None	See table following
setTimeWindow(string timeWindow)	None	The default time window for this DSSFile.
setTimeWindow(string startTime, string endTime)	None	The default time window for this DSSFile.
setTrimMissing(Boolean trim)	None	Sets whether TimeSeriesMath objects retrieved via calls to read(…) will have missing data trimmed from the beginning and end of the time window. [2]
write(HecMath dataset) [1]	integer	Write the data set to the DSS file. A return value of zero indicates success.
write (TimeSeriesMath timeSeriesData, string storeMethod)	integer	See table following

[1] Currently, the *read(…)* and *write(…)* methods can operate only on time-series data, paired data stream rating data, and text data represented by **TimeSeriesMath, PairedDataMath, StreamRatingMath**, and **TextMath** objects, respectively. Other record types, such as text data and gridded data are not yet supported by these methods.

[2] By default, **TimeSeriesMath** objects retrieved via calls to *read(…)* contain data only between the first non-missing value and the last non-missing value within the specified time window. Calling *setTrimMissing (False)* causes the data retrieved to include all data for the specified time window, including blocks of missing values at the beginning and end of the time window.

- Pathname mode = ***/***A*filter***/***B*filter***/***C*filter***/***D*filter***/***E*filter***/***F*filter***/***
- Pathname part mode = [**A=***Afilter*] [**B=***Bfilter*] [**C=***Cfilter*] [**D=***Dfilter*] [**E=***Afilter*] [**F=***Ffilter*]

In pathname mode, filters must be supplied for all pathname parts. In pathname part mode, only those parts that will not match everything must be specified.

Filters are comprised of the following components:

- **Normal text characters.** These characters are interpreted as they appear. For example, a pattern string of "B=XYZ" specifies matching every pathname that has a B-part of "XYZ".

Example 17: Reading a complete data set from DSS

```
from hec.script import *
from hec.heclib.dss import *
import java

try :
  try :
    myDss = HecDss.open("C:/temp/sample.dss")
    flow = myDss.get("/MISSISSIPPI/ST. LOUIS/FLOW//1DAY/OBS/", 1)

    if flow.numberValues == 0 :
      MessageBox.showError("No Data", "Error")
    else :
      plot = Plot.newPlot("Mississippi")
      plot.addData(flow)
      plot.showPlot()

  except Exception, e :
    MessageBox.showError(' '.join(e.args), "Python Error")
  except java.lang.Exception, e :
    MessageBox.showError(e.getMessage(), "Error")
finally :
  myDss.done()
```

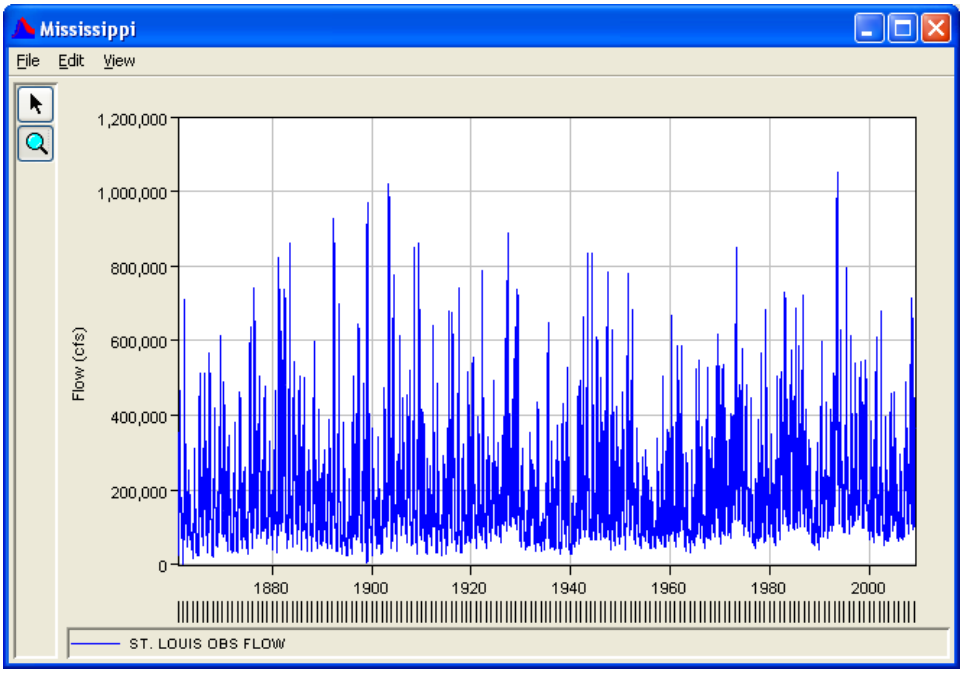

Figure 8.11 Example 17 Results

- **Special characters.** The special characters are comprised of the following list:
  - **'@'** or **'*'** (used interchangeably). This character can be specified as the first and/or last character of a filter and specifies matching a string of zero or more characters. For example, a pattern string of "B=XYZ@" specifies matching every pathname that as a B-part that begins with "XYZ". A pattern string of "B=*XYZ" specifies matching every

Example 18: Writing time series data to DSS

```
from hec.script import *
from hec.heclib.dss import *
from hec.heclib.util import *
from hec.io import *
import java

try :
  try :
    myDss = HecDss.open("C:/temp/test.dss")
    tsc = TimeSeriesContainer()
    tsc.fullName = "/BASIN/LOC/FLOW//1HOUR/OBS/"
    start = HecTime("04Sep1996", "1330")
    tsc.interval = 60
    flows = [0.0,2.0,1.0,4.0,3.0,6.0,5.0,8.0,7.0,9.0]
    times = []
    for value in flows :
      times.append(start.value())
      start.add(tsc.interval)
    tsc.times = times
    tsc.values = flows
    tsc.numberValues = len(flows)
    tsc.units = "CFS"
    tsc.type = "PER-AVER"
    myDss.put(tsc)

  except Exception, e :
    MessageBox.showError(' '.join(e.args), "Python Error")
  except java.lang.Exception, e :
    MessageBox.showError(e.getMessage(), "Error")
finally :
  myDss.done()


Output:
    -----DSS---ZOPEN:  New File Opened,  File: C:/temp/test.dss
                  Unit:   71;  DSS Version: 6-QD
 -----DSS---ZWRITE:  /BASIN/LOC/FLOW/01SEP1996/1HOUR/OBS/
    -----DSS---ZCLOSE Unit:   71,   File: C:/temp/test.dss
            Pointer Utilization:  0.25
            Number of Records:      1
            File Size:     70.2  Kbytes
            Percent Inactive:   0.0
```

pathname that has a B-part that ends with "XYZ". A pattern string of "B=*XYZ*" specifies matching every pathname that "XYZ" anywhere in the B-part. Note that the **@** or ***** character must be the first and/or last character of the filter (e.g. a pattern string of "B=ABC*XYZ" is invalid).

- **'#'** or **'!'** (used interchangeably). This character must be specified as the first character of a filter and specifies matching of every string *except* the remainder of the filter (e.g. it negates the remainder of the filter). A pattern string of "B=!XYZ*" specifies matching every pathname that does *not* have a B-part that begins with "XYZ".

Table 8.9 Time Series Data Storage Methods (Regular and Irregular)

	Integer	String	Description
Regular	0	REPLACE_ALL	Overwrite every value in the existing data
	1	REPLACE_MISSING_VALUES_ONLY	Don't overwrite non-missing values in the existing data
	2	REPLACE_ALL_CREATE	REPLACE_ALL + if new data has entire records of missing data, create empty records in existing data
	3	REPLACE_ALL_DELETE	REPLACE_ALL + if new data has entire records of missing data, delete records from existing data
	4	REPLACE_WITH_NON_MISSING	Overwrite every value in the existing data only if replacement is non-missing
Irregular	0	MERGE	Result for time window will be combination of existing and specified data
	1	DELETE_INSERT	Result for time window will be specified data only

- **No character**. The absence of any character specifies an empty filter, which matches a blank pathname part. Both of the following pattern strings match every pathname that has a blank A-part:
  - "//*/*/*/*/*/"
  - "A="

Using *getCatalogedPathnames(…)* with a pattern string utilizes the pathname matching in the underlying DSS library. To accomplish more sophisticated pathname filtering, use one of the *getCatalogedPathnames(…)* methods in conjunction with python text parsing and matching utilities.

Time Windows

Dates and times used to specify time windows should not contain blank characters and should include the 4 digit year. An example date and time

is "04MAR2003 1400". If a single string is used to specify the time window, the starting date and time must precede the ending date and time, for example *dssFile.setTimeWindow* ("04MAR2003 1400 06APR2004 0900"). A relative time may be used in the single string command, where the letter "T" represents the current time, and the days or hours can be subtracted or added to that. For example *dssFile.setTimeWindow("T-14D T-2H")* would specify the starting time as current time minus fourteen days and the ending time as the current time minus two hours.

Time Windows (specified by starting time and ending time) effect how the *DSSFile.read(…)* method operates for the various data types.

- **Time-series data.** The time window supplied to DSS.open(…) or *dssFile*.setTimeWindow(…) specifies the default time window for all subsequent *dssFile.read(…)* operations. If no time window is supplied to *DSS.open(…)* then the default time window is undefined until *dssFile.setTimeWindow(…)* is called. If the default time window is undefined, then all *dssFile*.read(…) operations involving time-series data *must* specify a time window either implicitly via the D-part of the specified pathname or explicitly via the *startTime* and *endTime* parameters. The order of precedence for time windows is as follows:
  - **Explicit time window.** Specified in *dssFile.read(pathname, startTime, endTime)* or *dssFile.read(pathname, timeWindow)*. The D-part of the pathname is ignored and may be empty.
  - **Default time window.** Specified in *DSS.open(filename, startTime, endTime)* or *DSS.open(filename, timeWindow)* or *dssFile.setTimeWindow(startTime, endTime)* or *dssFile.setTimeWindow(timeWindow)*. The D-part of the pathname is ignored and may be empty. The default time window can be set to undefined by calling *dssFile.setTimeWindow("", "")*.
  - **Implicit time window.** Specified as the D-part of the pathname supplied to *dssFile.read(pathname)* when the default time window is undefined. The D-part of the pathname must not be empty.
- **Paired data**. Time windows have no effect on reading paired data records.
- **Stream rating data**. Stream rating data are comprised of a time-series of individual rating records for a common location. The reading of stream rating data is not affected by default time windows. The implicit time window for reading stream rating data is the entire time span covered by the rating records. An explicit time window may be supplied by using the *dssFile.read(pathname, startTime, endTime)* method. If an explicit time window is specified, a (possible) subset of the rating records is retrieved that

cover the specified time window. The explicit time window is interpreted as the time window containing all time-series data that is to be rated via the stream rating data. The set of rating records retrieved for an explicit time window meets the following criteria.

- The earliest record retrieved is the latest rating that is on or before the start of the time window. If no such record exists, the earliest record in the rating time-series is retrieved.
- The latest record retrieved is the earliest rating that is on or after the end of the time window. If no such record exists, the latest record in the rating time-series is retrieved.
- All rating records between the earliest and latest retrieved records are also retrieved.

■ **Text data**. Time windows have no effect on reading text records.

8.8 DataContainer Class

The **HecDss** *get()* method returns a **DataContainer** object and the put() method takes a **DataContainer** object as a parameter. **DataContainers** are a base for different types of data. The **TimeSeriesContainer** and **PairedDataContainer** classes discussed below are both types of **DataContainer** classes. **DataContainer** objects have no methods that can be called by the user, but all data fields of **DataContainer** objects are directly accessible. **DataContainer** data fields are described in Table 8.10.

You can also exchange **DataContainers** with **HecMath** objects. Both **TimeSeriesContainer** and **PairedDataContainer** classes are extracted from **HecMath** objects using the *getData()* method as documented in Section 8.15.35. **HecMath** objects can be updated from **DataContainer** objects using the *setData()* method as documented in Section 8.15.109.

Table 8.10 DataContainer Data Fields

Field	Type	Description
fullName	string	The full name associated with the data in the data store (DSS pathname, if the DataContainer is associated with a DSS file)
location	string	The location associated with the data (DSS pathname B-part if the DataContainer is associated with a DSS file).
subVersion	string	The sub-version associated with the data.
version	string	The version associated with the data (DSS pathname F-part if the DataContainer is associated with a DSS file).
watershed	string	The watershed associated with the data (DSS pathname A-part if the DataContainer is associated with a DSS file).

Example 19: Extracting a DataContainer from HecMath

```
from hec.script import *
from hec.hecmath import *
theFile = DSS.open("myFile.dss")
flowDataSet = theFile.read("/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/")
theFile.done()
flowData = flowDataSet.getData()
thisWatershed = flowData.watershed
```

8.8.1 TimeSeriesContainer Class

TimeSeriesContainer is a type of DataContainer that contains information about time series data. TimeSeriesContainer objects are returned by the get () method and are required in the **put()** method of HecDss objects. TimeSeriesContainer objects have all the data fields described Section 8.8 (DataContainerClass) in addition to those described in Table 8.11. New TimeSeriesContainer objects can be created by a script if the TimeSeriesContainer class has been imported from the *hec.io* module (e.g. "from hec.io import *" or "from *hec.io* import TimeSeriesContainer"). The new object can be created by calling TimeSeriesContainer () (e.g. "myTSC = TimeSeriesContainer ()").

Table 8.11 TimeSeriesContainer Data Fields

Field	Type	Description
endTime	Integer[1]	The end time of the time window. If the data were retrieved, the end time may be later than the last time in the times list.
interval	Integer	The interval, in minutes, between each set of consecutive times in the times list. For irregular-interval times, the interval field is set to –1.
numberValues	Integer	The length of values and times lists.
parameter	string	The parameter associated with the data.
quality	list of integers[2]	The optional list of quality flags. If this list is present, there must be a quality for each value in the values array. If this list is not present, the quality field is set to None.
startTime	integer[2]	The start time of the time window. If the data were retrieved, the start time may be earlier than the first time in the times list.
subLocation	string	The sub-location associated with the data.
subParameter	string	The sub-parameter associated with the data.
times	list of integers[1]	The list of times. There must be a time for each value in the values list, and times must increase from one index to the next.
timeZoneID	string	The time-zone for times in the times list. If unknown, the timeZoneID field is set to None

Field	Type	Description
timeZoneRawOffset	integer	The offset, in milliseconds, from UTC to the time zone for the times in the times list.
type	string	The type of the data (e.g. "INST-VAL", "INST-CUM", ""PER-AVER", "PER-CUM").
units	string	The units of the data.
values	list of floating-point	The data values, each of which has a corresponding time in the times list and optionally a corresponding quality in the quality list. All lists must be the same length.

[1] Integer times from this field can be converted to string representations by using the *set()* and *dateAndTime()* methods of **HecTime** objects and can be generated by the **HecTime** *value()* method.
[2] Quality values are interpreted according to Table 8.12.

Table 8.12 Data Quality Bits

Bit(s)	Description
1	Set when the datum has been tested (screened).
2	Set when the datum passed all tests.
3	Set when the datum is missing (either originally missing or set to missing by a test).
4	Set when at least one test classified the datum as questionable.
5	Set when at least one test classified the datum as rejected.
6-7	Set by the RANGE test. Interpreted as a 2-bit unsigned integer with values having the following meanings: **0** value of datum < than 1st limit or no range test applied **1** 1st limit <= value of datum < 2nd limit **2** 2nd limit <= value of datum < 3rd limit **3** 3rd limit <= value of datum
8	Set when the datum has been changed from the original value.
9-11	Datum replacement indicator. Interpreted as a 3-bit unsigned integer with values having the following meanings: **0** datum was not replaced (original value) **1** datum was replaced by an automated tool **2** datum was replaced by an interactive tool (e.g., fill operaton) **3** datum was replaced by manual edit in an interactive tool **4** original value was accepted or restored in an interactive tool **5-7** reserved for future use
12-15	Datum replacement value computation method. Interpreted as a 4-bit unsigned integer with values having the following meanings: **0** datum was not replaced (original value) **1** datum value computed by linear interpolation **2** datum value was entered manually **3** datum value was replaced with a missing value **4** datuma value was entered graphically **5-15** reserved for future use
16	set when datum failed an absolute magnitude test
17	set when datum failed a constant value test

Bit(s)	Description
18	set when datum failed a rate of change test
19	set when datum failed a relative magnitude test
20	set when datum failed a duration magnitude test
21	set when datum failed a negative incremental value test
22	reserved for future use
23	set when datum is excluded from testing (e.g. DATCHK GAGEFILE entry)
24	reserved for future use
25	set when datum failed a user-defined test
26	set when datum failed a distribution test
27-31	reserved for future use
32	set when datum is protected from being replaced

Example 20: Using a TimeSeriesContainer Object

```
from hec.script import *
from hec.heclib.dss import *
from hec.io import TimeSeriesContainer
from hec.heclib.util import HecTime

watershed = “GREEN RIVER”
loc = “OAKVILLE”
param = “STAGE”
ver = “OBS”
startTime = “12Oct2003 0100”
values = [12.36, 12.37, 12.42, 12.55, 12.51, 12.47, 12.43, 12.39]
hecTime = HecTime()
tsc = TimeSeriesContainer()
tsc.watershed = watershed
tsc.location = loc
tsc.parameter = param
tsc.version = ver
tsc.fullName = “/%s/%s/%s//1HOUR/%s/” % \
        (watershed, loc, param, ver)
tsc.interval = 60
hecTime.set(startTime)
times = []
for value in values :
        times.append(hecTime.value())
        hecTime.add(tsc.interval)
tsc.values = values
tsc.times = times
tsc.startTime = times[0]
tsc.endTime = times[-1]
tsc.numberValues = len(values)
tsc.units = “FEET”
tsc.type = “INST-VAL”
dssFile = HecDss.open(“myFile.dss”)
dssFile.put(tsc)
dssFile.done()
```

8.8.2 PairedDataContainer Class

PairedDataContainer is a type of **DataContainer** that contains information about paired data. **PairedDataContainer** objects are returned by the *get ()* method. **PairedDataContainer** objects are described in Section 8.8 in addition to those described in Table 8.13. New **PairedDataContainer** objects can be created by a script if the **PairedDataContainer** class has been imported from the *hec.io* module (e.g. "from hec.io import *" or "from hec.io import PairedDataContainer"). The new object can be created by calling **PairedDataContainer ()** (e.g. "myPDC = PairedDataContainer ()").

Table 8.13 PairedContainer Data Fields

Field	Type	Description
date	string	The date associated with the paired-data.
datum	floating-point	The zero-stage elevation of a stream gauge.
labels	list of strings	The list of labels used to identify each of the y-ordinates lists. If there is only 1 y-ordinates list (e.g. numberCurves == 1), the labels field is set to None.
labelsUsed	Boolean	A flag specifying whether labels are used. This field is set to True if there is more than 1 y-ordinates list and False otherwise.
numberCurves	integer	The number of y-ordinates lists.
numberOrdinates	integer	The length of the x-ordinates list and each of the y-ordinates lists.
offset	floating-point	The offset value of a stream rating.
shift	floating-point	The shift value for a stream rating.
transformType	integer	Type of transformation to use (1 = "LINLIN", 2 = "LOGLOG")
xOrdinates	list of floating-point	The x-ordinate values. Each y-ordinate values list must be of the same length as this field.
xparameter	string	The parameter of the x-ordinates.
xtype	string	The type of the x-ordinates ("UNT" for unitary or "LOG" for logarithmic).
xunits	string	The units of the x-ordinates.
yOrdinates	list of lists of floating-point	The y-ordinate values. Each list of y-ordinate values must be of the same length as the list of x-ordinate values. The nth value of each y-ordinates list is associated with the nth value of the x-ordinates list.
yparameter	string	The parameter of the y-ordinates.
ytype	string	The type of the y-ordinates ("UNT" for unitary or "LOG" for logarithmic).
yunits	string	The units of the y-ordinates.

Example 21: Using a PairedDataContainer Object

```
from hec.script import *
from hec.heclib.dss import *
from hec.io import PairedDataContainer

watershed = “GREEN RIVER”
loc = “OAKVILLE”
xParam = “STAGE”
yParam = “FLOW”
date = “12Oct2003”
stages = [0.4, 0.5, 1.0, 2.0, 5.0, 10.0, 12.0]
flows = [0.1, 3, 11, 57, 235, 1150, 3700]
pdc = PairedDataContainer()
pdc.watershed = watershed
pdc.location = loc
pdc.parameter = param
pdc.version = ver
pdc.fullname = “/%s/%s/%s-%s///%s/” % \
      (watershed, loc, xParam, yParam, date)
pdc.xOrdinates = stages
pdc.yOrinates = [flows]
pdc.numberCurves = 1
pdc.numberOrdinates = len(stages)
pdc.labelsUsed = False
pdc.xunits = “FEET”
pdc.yunits = “CFS”
pdc.xtype = “LOG”
pdc.ytype = “LOG”
pdc.xparameter = xParam
pdc.yparameter = yParam
pdc.date = date
pdc.transformType = 2dssFile = HecDss.open(“myFile.dss”)
dssFile.put(pdc)
dssFile.done()
```

8.9 HecTime Class

HecTime objects are used to manipulate dates and times and to convert dates and times among different formats. To use **HecTime** objects, the **HecTime** class must be imported from the *hec.heclib.util* module (e.g. "from hec.heclib.util import *" or "from hec.heclib.util import HecTime"). After importing the class, a new HecTime object can be created by calling HecTime() (e.g. "myTime = HecTime()"). **HecTime** methods are described in Table 8.14.

Table 8.14 HecTime Methods

Method	Returns	Description
add(HecTime increment)	None	Adds the specified increment to the object’s date and time.
add(integer increment)	None	Adds the specified increment in minutes to the object’s date and time.

Method	Returns	Description
addDays(integer days)	None	Adds the specified number of days to the time
addHours (integer hours)	None	Adds the specified number of hours to the time
addMinutes (integer minutes)	None	Adds the specified number of minutes to the time
addSeconds (integer seconds)	None	Adds the specified number of seconds to the time
adjustToIntervalOffset(integer intervalMinutes, integer offset)	None	Changes the time to be at the standard time for that interval, according to HEC-DSS. For example, will adjust daily data to 2400 hours of that day.
compareTimes(HecTime other)	integer	Returns one of the following values: -1 The object's date and time is less than the other object's 0 The objects' dates and times are equal 1 The object's date and time is greater than the other object's
computeNumberIntervals(HecTime otherTime, integer intervalInMins)	integer	Determines the number of periods between the time set and the time passed in, where intervalInMins is the interval length.
convertTimeZone(HecTime hecTime, TimeZone fromTimeZone, TimeZone toTimeZone)	None	Changes the time passed in from the first time zone to the second time zone. Adjusts for daylight savings time, according to the time given.
date()	string	Returns a string representation of the object's date. Same as date(2).
date(integer format)	string	Returns a string representation of the object's date, formatted according to the integer parameter.[1]
dateAndTime()	string	Returns a string representation of the object's date and time. Same as dateAndTime(2).
dateAndTime(integer format)	string	Returns a string representation of the object's date and time, formatted according to the integer parameter. [1]
day()	integer	Returns the day portion of the object's date as an integer
dayOfWeek()	integer	Returns the day of the week with Sunday starting as 1, Monday 2
dayOfWeekName()	string	Returns the name of the day of the week, such as "Sunday"

Method	Returns	Description
dayOfYear()	integer	Returns the Julian day of the year, with Jan.1 returned as "1".
equalTo(HecTime other)	Boolean	Returns True if the object's date and time is equal to the other object's date and time. Returns False otherwise.
greaterThan(HecTime other)	Boolean	Returns True if the object's date and time is greater (later) than the other object's date and time. Returns False otherwise.
greaterThanEqualTo(HecTime other)	Boolean	Returns True if the object's date and time is greater (later) than or equal to the other object's date and time. Returns False otherwise.
getJavaDate(integer minutesTimezoneOffset)	Date	Returns the Java Date for this time
getMinutes()	long	Returns the number of milliseconds since Jan 1, 1900
getTimeInMillis()	long	Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT
getTimeWindow (String userLine, HecTime startTime, HecTime endTime)	integer Success=0 Fail = -1	Takes a user entered line with start date/time and end date/time and converts to HecTime.
getXMLDateTime (integer minutesTimezoneOffset)	string	Returns the date in XML format
hour()	integer	Returns the hours portion of the object’s time as an integer
isDefined()	Boolean	Returns True if the object is set to a valid date and time and False if not.
isDateDefined()	Boolean	Returns True if the date portion is valid (ignores time) and False if not.
isTimeDefined()	Boolean	Returns True if the time portion is valid (ignores date) and False if not.
increment(integer numberPeriods, integer minutesInPeriod)	None	Adds the number of periods to the date
isoDate()	integer	Returns the date in ISO format (YYMMDD)
isoTime()	integer	Returns the time in ISO format (HHMMSS)
lessThan(HecTime other)	Boolean	Returns True if the object's date and time is less (earlier) than the other object's date and time. Returns False otherwise.
lessThanEqualTo(HecTime other)	Boolean	Returns True if the object's date and time is less (earlier) than or equal to the other object's date and time. Returns False otherwise.

Method	Returns	Description
julian()	integer	Returns the number of days since Jan 1, 1900
minute()	integer	Returns the minutes portion of the object's time as an integer
month()	integer	Returns the month portion of the object's date as an integer
notEqualTo(HecTime other)	Boolean	Returns False if the object's date and time is equal to the other object's date and time. Returns True otherwise.
second()	integer	Returns the seconds portion of the object's time as an integer
setCurrent()	None	Sets the date/time to the current time.
set (integer time)	None	Sets the object to the date and time represented by the integer (minutes since 31Dec1899 00:00)
set(string dateAndTime)	integer	Sets the object to the date and time represented by the string, and returns zero if successful.
set (string date, string time)	integer	Sets the object to the date represented by the date string and the time represented by the time string, and returns zero if successful.
set (HecTime time)	None	Sets the object to the date and time represented by the HecTime parameter.
setDate(string date)	integer	Sets the object to the date represented by the string, and returns zero if successful. The time portion of the object is not modified.
setTime(string time)	integer	Sets the object to the time represented by the string, and returns zero if successful. The date portion of the object is not modified.
setJulian(integer julian)	None	Sets the date to the Julian days (Since Jan 1, 1900)
setXML (String dateTime)	integer Success=0	Sets the date and time according to the XML string
setUndefined()	None	Sets the object to represent an undefined time, as if the object had just been created.
showTimeAsBeginningOfDay(Boolean showBeginning)	None	Specifies whether the object is to show midnight times as 00:00 (vs 24:00)

Method	Returns	Description
subtract(HecTime increment)	None	Subtracts the specified increment from the object's date and time.
subtract(integer increment)	None	Subtracts the specified increment in minutes from the object's date and time.
time()	string	Returns a string representation of the object's time.
toString()	string	Returns the date and time in string format, for format style 2
toString(integer format)	string	Returns the date and time in string format, for format style given
value()	integer	Returns the object's date and time as in the number of minutes since 31Dec1899 0000.
year()	integer	Returns the year portion of the object's date as an integer

[1] The format of the string returned by the *date(integer format)* method and the date portion of the string returned by the *dateAndTime(integer format)* method are displayed in Table 8.15.

Table 8.15 HecTime Date Formats

0	June 2, 1985	10	June 2, 85	100	JUNE 2, 1985	110	JUNE 2, 85
1	Jun 2, 1985	11	Jun 2, 85	101	JUN 2, 1985	111	JUN 2, 85
2	2 June 1985	12	2 June 85	102	2 JUNE 1985	112	2 JUNE 85
3	June 1985	13	June 85	103	JUNE 1985	113	JUNE 85
4	02Jun1985	14	02Jun85	104	02JUN1985	114	02JUN85
5	2Jun1985	15	2Jun85	105	2JUN1985	115	2JUN85
6	Jun1985	16	Jun85	106	JUN1985	116	JUN85
7	02 Jun 1985	17	02 Jun 85	107	02 JUN 1985	117	02 JUN 85
8	2 Jun 1985	18	2 Jun 85	108	2 JUN 1985	118	2 JUN 85
9	Jun 1985	19	Jun 85	109	JUN 1985	119	JUN 85

8.10 Plotting Basics

The title, viewport, axis label, axis tics, and legend of a plot, each of which are accessible via scripts, are identified in Figure 8.12.

Example 22: Creating a Plot

```
myPlot = Plot.newPlot()
or
thePlot = Plot.newPlot("Elevation vs Flow")
```

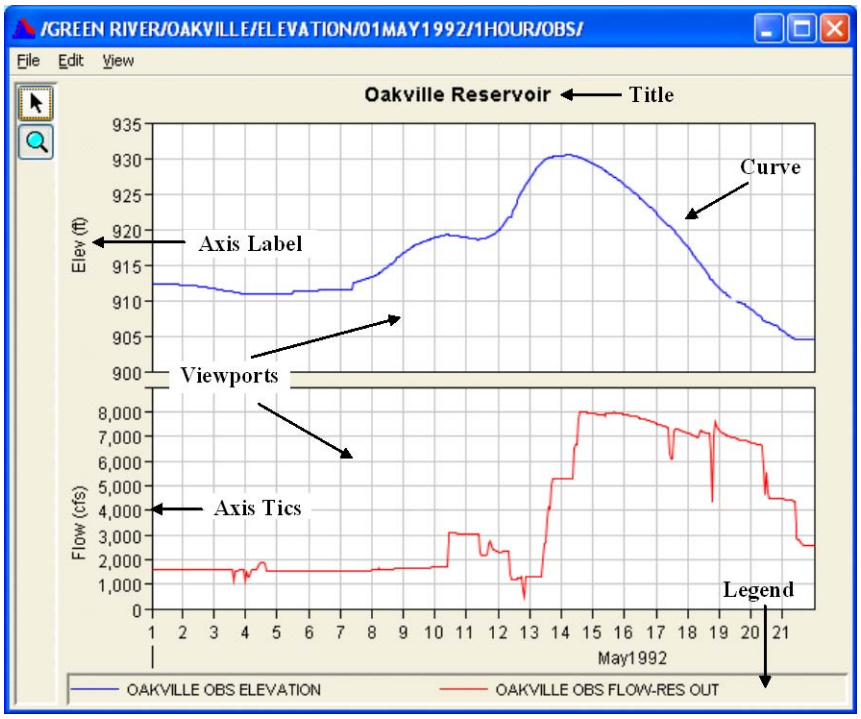

Figure 8.12 Plot Components

8.10.1 Plot Class

```
Plot.newPlot()
Plot.newPlot(string title)
```

The **Plot** class in the *hec.script* module is used to create a new Plot dialog. It contains two methods to create a Plot dialog, each of which returns a G2dDialog object.

8.10.2 Changing Plot Component Attributes

Not all Plot Component attributes are visible by default, and setting the attribute may not make that attribute visible. Often it is necessary to set the visibility of the attribute by calling *setAttributeVisible(True)*. Reading a flow data set from a DSS file, plotting the data set, setting the minor Y grid color to black and making it display are illustrated in Example 23.

8.10.3 G2dDialog Class

G2dDialog objects are the dialog that plots display in. **G2dDialog** methods are described in Table 8.16.

Example 23: Plotting DSS Data

```
from hec.script import *
from hec.script.Constants import TRUE, FALSE

theFile = HecDss.open("myFile.dss")          # open myFile.dss
thePath = "/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/"
flowDataSet = theFile.get(thePath)           # read a path name
thePlot = Plot.newPlot()                     # create the plot
thePlot.addData(flowDataSet)                 # add the flow data set to the plot
thePlot.showPlot()                           # show the plot
viewport0=thePlot.getViewport(0)             # get the first viewport
viewport0.setMinorGridYColor("black")        # set the viewport's minor Y grid to black
viewport0.setMinorGridYVisible(TRUE)         # tell the minor Y grid to display
```

Table 8.16 G2dDialog Methods

Method	Returns	Description
addData(DataContainer dc)	None	Add the DataContainer specified by dc to the plot. Must be called before showPlot(). *Do not use this if a PlotLayout object is used on this plot.*
applyTemplate(string templateFile)	None	Apply the given template to this plot
configurePlotLayout()	None	Display the "Configure Plot Layout" dialog for this plot
configurePlotLayout(PlotLayout layout)	None	Configures the plot layout for this plot according to the specified PlotLayout object. *If this method is used, do not use the addData() method with the same plot.*
close()	None	Closes the plot
configurePlotTypes()	None	Display the configure plot types dialog
copyToClipboard()	None	Copy the plot to the system clipboard
defaultPlotProperties()	None	Display the default plot properties dialog
exportProperties()	None	Allows you to save the properties of the plot to a disk.
exportProperties(string templateName)	None	Allows you to save the properties of the plot to the file specified by templateName.
getCurve(HecMath filenam)	G2dLine	Return the G2dLine for the DataSet specified by dataSet

Method	Returns	Description
getCurve(string dssPath)	G2dLine	Return the G2dLine for the path specified in dssPath
getHeight()	integer	Return the height of the dialog in screen coordinates.
getLegendLabel(DataContainer dc)	G2dLabel	Return the legend label object for the specified data container.
getLocation()	Point	Return the location of the dialog in screen coordinates.[1]
getPlotTitle()	G2dTitle	Return the title for the G2dDialog
getPlotTitleText()	string	Return the text of the title for the G2dDialog
getSize()	Dimension	Return the dimensions of the dialog in screen coordinates.
getViewport(HecMath filenam)	Viewport	Return the Viewport that contains the curve specified by dataSet
getViewport(integer viewportIndex)	Viewport	Return the viewport at index specified by viewportIndex
getViewport(string dataSetPath)	Viewport	Return the Viewport that contains the curve specified by dataSetPath
getWidth()	Integer	Return the width of the dialog in screen coordinates.
hide()	None	Hide the dialog
iconify()	None	Minimize (iconify) the dialog
isPlotTitleVisible()	Boolean	Return the visibility state of the title of this plot.
maximize()	None	Maximize the dialog
minimize()	None	Minimize (iconify) the dialog
newPlotLayout()	PlotLayout	Return a PlotLayout object that can be used to configure the layout of this plot.
plotProperties()	None	Display the plot properties dialog for this plot
print()	None	Display the print dialog for this plot

Method	Returns	Description
printMultiple()	None	Display the print multiple dialog for this plot
printPreview()	None	Display the print preview dialog for this plot
printToDefault()	None	Prints using the printer defaults such as page format and printer. This method does not display the printer dialog for user interaction.
repaint()	None	Forces the plot to be refreshed.
restore()	None	Restore the dialog from a minimized or maximized state
saveAs()	None	Display the saveAs dialog for this plot
saveToJpeg(string filename)	None	Save the plot to the Jpeg file specified by fileName
saveToJpeg(string filename, integer quality)	None	Save the plot to the Jpeg file specified by filename, with the specified quality[2].
saveToMetafile(string filename)	None	Save the plot to the Windows Meta file specified by filename
saveToPng(string filename)	None	Save the plot to the Portable Network Graphics file specified by filename
saveToPostscript(string filename)	None	Save the plot to the PostScript file specified by filename
setLegendBackgound(string color)	None	Sets the background color of the legend.
setLegendLabelText(DataContainer dc, string text)	None	Sets the legend label text
setLegendLocation(string location)	None	Sets the location of the legend[3].
setLocation(integer x,integer y)	None	Sets the location of the dialog in screen coordinates.[1]
setPlotTitleText(string text)	None	Sets the text of the title for this plot
setPlotTitleVisible(Boolean state)	None	Sets the visibility of the title for this plot

Method	Returns	Description
setSize(integer width, integer height)	None	Sets the size of the dialog in screen coordinates.
setVisible(Boolean visible)	None	Makes the plot visible
showPlot()	None	Show the dialog
stayOpen()	None	Keeps the plot on the screen for batch mode only
tabulate()	HecDataTableFrame	Display the table view of this plot

[1] The coordinate system used is a graphics coordinate system, where X increases to the right and Y increases downward from the origin (0,0) which is located in the top left corner of the display. Locations set or retrieved refer to the top left corner of the plot in reference to this coordinate system.
[2] The specified quality is limited to an effective range of 0 – 100, inclusive. Higher qualities produce larger files and take longer to generate. The *saveToJpeg(fileName)* call currently produces the same results as *saveToJpeg(fileName, 75)*.
[3] Valid legend locations are "Right" and "Bottom".

Example 24: Plot Dialog

```
from hec.script import *                                # for Plot class
from hec.heclib.dss import *                            # for DSS class
theFile = HecDss.open("myFile.dss")                     # open myFile.dss
thePath = "/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/"
flowDataSet = theFile.get(thePath)                      # read a path name
thePlot = Plot.newPlot()                                # create a new Plot
thePlot.addData(flowDataSet)                           # add flow data container
thePlot.showPlot()                                      # show the plot
thePlot.setLocation(50,50)                              # moves plot to 50,50
```

8.10.4 PlotLayout Class

PlotLayout objects hold information about the layout of the plot dialog. The use of **ViewportLayout** objects, in conjunction with **PlotLayout** objects, allows scripts to specify the same layout information accessible interactive via the "Configure Plot Layout" dialog. A **PlotLayout** object is obtained by calling *Plot.newPlotLayout()*. **PlotLayout** methods are described in Table 8.17.

8.10.5 ViewportLayout Class

ViewportLayout objects hold information about the layout of an individual viewport within the plot dialog. The use of **ViewportLayout** objects, in conjunction with **PlotLayout** objects, allows scripts to specify the same layout information accessible interactive via the "Configure Plot Layout" dialog. A **ViewportLayout** object is obtained by calling one of

Table 8.17 PlotLayout Methods

Method	Returns	Description
addViewport()	ViewportLayout	Adds a ViewportLayout to the PlotLayout with a default weight of 100. Returns a reference to the new ViewportLayout.
addViewport(floating-point weight)	ViewportLayout	Adds a ViewportLayout to the PlotLayout with the specified weight. Returns a reference to the new ViewportLayout.
hasLegend()	Boolean	Returns whether this PlotLayout is configured to display the legend.
hasToolbar()	Boolean	Returns whether this PlotLayout is configured to display the toolbar.
getViewportCount()	integer	Returns the number of ViewportLayout objects currently in the PlotLayout object.
getViewports()	java.util.List of ViewportLayouts	Returns the ViewportLayout objects currently in the PlotLayout object.
getViewportWeights()	list of floating-points	Returns the weights of the ViewportLayout objects currently in the PlotLayout object.
setHasLegend(Boolean state)	None	Configures the PlotLayout object to display the legend or not, depending upon the specified state.
setHasToolbar(Boolean state)	None	Configures the PlotLayout object to display the toolbar or not, depending upon the specified state.

the *addViewport* methods of a **PlotLayout** object. **ViewportLayout** objects are only used to configure the plot layout. Manipulation of axis labels, background colors, etc. is performed using Viewport objects as described in table below. **ViewportLayout** methods are described in Table 8.18.

Table 8.18 ViewportLayout Methods

Method	Returns	Description
addCurve(string axis, DataContainer curve)	None	Adds the specified curve to the specified axis of the ViewportLayout object.
getMajorGridXStyleString()	string	
getMajorGridYStyleString()	string	
getMinorGridXStyleString()	string	
getMinorGridYStyleString()	string	

Method	Returns	Description
getY1Data()	List	Returns a *java.util.List* of all curves that have been added to the Y1 axis of this object
getY2Data()	List	Returns a *java.util.List* of all curves that have been added to the Y2 axis of this object
hasY1Data()	Boolean	Returns whether any curves have been added to the Y1 axis of this object
hasY2Data()	Boolean	Returns whether any curves have been added to the Y2 axis of this object
setMajorGridXStyleString(string majorGridXStyle)	None	
setMajorGridYStyleString(string majorGridYStyle)	None	
setMinorGridXStyleString(string minorGridXStyle)	None	
setMinorGridYStyleString(string minorGridYStyle)	None	
setLinear(string axisName)	None	
setLogarithmic(string axisName)	None	
scaleAxisFromOpposite(string axis)	None	

The script in Example 25 reads precipitation, stage and flow data set from a DSS file, and configures a plot to display the precipitation on top in a viewport that occupies thirty percent of the available space and to display the stage and flow on separate axes of a bottom viewport that occupies the remaining seventy percent of available space.

Example 25: PlotLayout and ViewportLayout Objects

```
from hec.script import *                                    # for Plot class
from hec.heclib.dss import *                                # for DSS class
theFile = HecDss.open("myFile.dss")                         # open myFile.dss
precipPath = "/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/"
stagePath = "/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/"
flowPath = "/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/"
precipData = theFile.get(precipPath)                        # read the precip
stageData = theFile.get(stagePath)                          # read the stage
flowData = theFile.get(flowPath)                            # read the flow
thePlot = Plot.newPlot()                                    # create a new Plot
layout = Plot.newPlotLayout()                               # create a new PlotLayout
topView = layout.addViewport(30)                            # get the top viewport
bottomView = layout.addViewport(70)                         # get the bottom viewport
topView.addCurve("Y1", precipData)                          # add the precip to top
bottomView.addCurve("Y1", stageData)                        # add the stage to bottom
bottomView.addCurve("Y2", flowData)                         # add the flow to bottom
thePlot.configurePlotLayout(layout)                         # configure the plot
thePlot.showPlot()                                          # show the plot
```

8.10.6 Viewport Class

Viewport objects hold the data set curves. **Viewport** methods are described in Table 8.19.

Table 8.19 Viewport Methods

Method	Returns	Description
addAxisMarker(AxisMarker marker)	None	Adds a marker line described by the AxisMarker parameter
addXAxisMarker()	None	Display the Axis Marker Properties Dialog for a marker line to add to the X axis
addXAxisMarker(floating-point value)	None	Add an X Axis marker at the location specified by value
addXAxisMarker(string value)	None	Add a X Axis marker at the location specified by value
addYAxisMarker()	None	Display the Axis Marker Properties Dialog for a marker line to add to the Y axis
addYAxisMarker(string value)	None	Add a Y Axis marker at the location specified by value
editProperties()	None	Display the Edit Properties dialog for this Viewport
getAxis(string axisName)	Axis	Return the **Axis** specified by **axisName** for this Viewport
getAxisLabel(string axisName)	AxisLabel	Return the **AxisLabel** for the axis specified by **axisName** for this Viewport
getAxisTics(string axisName)	AxisTics	Return the **AxisTics** for the axis specified by **axisName** for this Viewport
getBackground()	Color	Return the background color for the Viewport as a Color.
getBackgroundString()	string	Return the background color name for the Viewport as a string.
getBorderColor()	Color	Return the border color for the Viewport as a Color.
getBorderColorString()	string	Return the background color name for the Viewport as a string
getBorderWeight()	float	Return the border weight for this Viewport
getFillPatternString()	string	Return the fill pattern for this Viewport as a String
getMajorGridXColor()	Color	Return the color of the vertical lines of the major grid for this Viewport as a Color

Method	Returns	Description
getMajorGridXColorString ()	string	Return the color of the vertical lines of the major grid for this Viewport as a string
getMajorGridXWidth ()	floating point	Return the width of the vertical lines of the major grid for this Viewport
getMajorGridYColor ()	Color	Return the color of the horizontal lines of the major grid of this Viewport as a Color
getMajorGridYColorString ()	string	Return the color of the horizontal lines of the major grid for this Viewport as a string
getMajorGridYWidth ()	floating point	Return the width of the vertical lines of the major grid for this Viewport
getMinorGridXColor()	Color	Return the color of the vertical lines of the minor grid for this Viewport as a color
getMinorGridXColorString()	string	Return the color of the vertical lines of the minor grid for this Viewport as a string
getMinorGridXWidth()	floating point	Return the width of the vertical lines of the minor grid for this Viewport
getMinorGridYColor()	Color	Return the color of the horizontal lines of the minor grid for this Viewport as a Color.
getMinorGridYColorString()	string	Return the color of the horizontal lines of the minor grid for this Viewport as a string
getMinorGridYWidth()	floating point	Return the width of the vertical lines of the minor grid for this Viewport
isBackgroundVisible()	Boolean	Return whether the background is drawn for this Viewport
isBorderVisible()	Boolean	Return whether the border is drawn for this Viewport
isMajorGridXVisible()	Boolean	Return whether the vertical lines of the major grid are drawn for this Viewport
isMajorGridYVisible ()	Boolean	Return whether the horizontal lines of the major grid are drawn for this Viewport

Method	Returns	Description
isMinorGridXVisible()	Boolean	Return whether the vertical lines of the minor grid are drawn for this Viewport
isMinorGridYVisible ()	Boolean	Return whether the horizontal lines of the minor grid are drawn for this Viewport
setBackground(string colorString)	None	Set the background to the color specified by colorString
setBorderColor(string borderColor)	None	Set the border color for this Viewport
setBorderWeight(floating-point borderWeight)	None	Set the border weight for this Viewport
setBackgroundVisible(Boolean state)	None	Set whether to draw the background for this Viewport
setBorderVisible(Boolean state)	None	Set whether to draw the border for this Viewport
setFillPattern(string pattern)	None	Set the fill pattern for this Viewport
setGridColor(string colorString)	None	Set the color of the horizontal and vertical lines of the major and minor grids for this Viewport.
setGridXColor(string colorString)	None	Set the color of the vertical lines of the major and minor grids for this Viewport.
setGridYColor(string colorString)	None	Set the color of the horizontal lines of the major and minor grids for this Viewport.
setMajorGridXColor(string majorGridXColor)	None	Set the color of the vertical lines of the major grid for this Viewport.
setMajorGridXVisible(Boolean state)	None	Set whether to draw the vertical lines of the major grid for this Viewport
setMajorGridXWidth(floating-point gridLineWidth)	None	Set the width of the vertical lines of the major grid for this Viewport
setMajorGridYColor(string majorGridYColor)	None	Set the color of the horizontal lines of the major grid for this Viewport.
setMajorGridYVisible(Boolean state)	None	Set whether to draw the horizontal lines of the major grid for this Viewport
setMajorGridYWidth(floating-point gridLineWidth)	None	Set the width of the horizontal lines of the major grid for this Viewport
setMinorGridXColor(string minorGridXColor)	None	Set the color of the vertical lines of the minor grid for this Viewport.

Method	Returns	Description
setMinorGridXVisible(Boolean state)	None	Set whether to draw the vertical lines of the minor grid for this Viewport
setMinorGridXWidth(floating-point gridLineWidth)	None	Set the width of the vertical lines of the minor grid for this Viewport
setMinorGridYColor(string minorGridYColor)	None	Set the color of the horizontal lines of the minor grid for this Viewport.
setMinorGridYVisible(Boolean state)	None	Set whether to draw the horizontal lines of the minor grid for this Viewport
setMinorYGridWidth(floating-point gridLineWidth)	None	Set the width of the horizontal lines of the minor grid for this Viewport

8.10.7 AxisMarker Class

AxisMarker objects hold complete descriptions of marker lines to be added to viewports. **AxisMarker** objects have fields that are settable by the user to create marker lines of various styles. New **AxisMarker** objects are created by calls to *AxisMarker()* (e.g., *myMarker = AxisMarker()*).

AxisMarker fields are described in Table 8.20.

Table 8.20 AxisMarker Fields

Field	Type	Description	Default
axis	string	"X" or "Y"	"Y"
fillColor	string	Color of the filled area.	"black"
fillPattern	string	Pattern of the filled area.	"solid"
fillStyle	string	Specifies whether the filled area is to be above or below the marker line, or to not fill at all.	"none"
labelAlignment	string	Specifies whether the label text is to appear left justified, right justified or centered.	"left"
labelColor	string	Color of the label text	"black"
labelFont	string	The font to use for the label.[1]	None
labelPosition	string	Specifies whether the label text is to appear above, below, or in the center of the marker line	"above"
labelText	string	Text to appear with marker line	""
lineColor	string	Color of the marker line	"black"
lineStyle	string	Style of the marker line	"solid"

Field	Type	Description	Default
lineWidth	floating point	Width of the marker line	1.0
value	string	Location of marker on axis (e.g. "712.5" or "23Aug2003 1015")	"0"

[1] Fonts are specified as name[,style[,size]] where style is Plain, Bold, Italic, or Bold Italic (e.g. "Arial,BoldItalic,12", "Lucida Console,Plain,10").

The script in Example 26 reads a data set from a DSS file, plots that data set, sets the Viewport's background to light gray and adds a marker line on the Y axis.

Example 26: Viewport Class

```
from hec.script import *                                # for Plot class
from hec.script.Constants import TRUE, FALSE
from hec.heclib.dss import *                            # for DSS class
theFile = HecDss.open("myFile.dss")                     # open myFile.dss
thePath = "/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/"
flowDataSet = theFile.read(thePath)                     # read a path name
thePlot = Plot.newPlot()                                # create a new Plot
thePlot.addData(flowDataSet)                            # add the flow data
viewport0=thePlot.getViewport(0)                        # get the first Viewport
viewport0.setBackground("lightgray")                    # set the Viewport's bg
viewport0.setBackgroundVisible(TRUE)                    # tell Viewport to draw bg
marker = AxisMarker()                                   # create a new marker
marker.axis = "Y"                                       # set the axis
marker.value = "20000"                                  # set the value
marker.labelText = "Damaging Flow"                      # set the text
marker.labelColor = "red"                               # set the text color
marker.lineColor = "red"                                # set the line color
marker.fillColor = "red"                                # set the fill color
marker.fillType = "above"                               # set the fill type
maker.fillPattern = "diagonal cross"                    # set the fill pattern
viewport0.addAxisMarker(marker)                         # add the marker to the
                                                        # viewport
```

8.10.8 Axis Class

Axis methods are described in Table 8.21.

Table 8.21 Axis Methods

Method	Returns	Description
getLabel()	string	Return the Axis label
getMajorTic()	floating-point	Return the major tic interval for this Axis
getMinorTic()	floating-point	Return the minor tic interval for this Axis
getNumTicLabelLevels()	integer	Return the number of tic label levels for this Axis

Method	Returns	Description
getScaledLabel()	String	Return the label with scientific notation
getScaleMax()	floating-point	Return the maximum value for this Axis
getScaleMin()	floating-point	Return the minimum value for this Axis
getTicColor()	Color	Return the tic color
getTicColorString()	String	Return the Tic color as a String
getTicTextColor()	Color	Return the tic text color
getTicTextColorString()	String	Return the tic text color as a String
getViewMax()	floating-point	Return the maximum value for the (possibly) zoomed view for this Axis
getViewMin()	floating-point	Return the minimum value for the (possibly) zoomed view for this Axis
isComputingMajorTics()	Boolean	Return if major tics are to be computed
isComputingMinorTics()	Boolean	Return if minor tics are to be computed
isReversed()	Boolean	Returns whether the Axis is reversed.[1]
setComputeMajorTics(Boolean state)	None	Set whether to compute major tics
setComputeMinorTics(Boolean state)	None	Set whether to compute minor tics
setLabel(string label)	None	Set the label of this Axis
setLinear()	None	
setLogarithmic	None	
setMajorTicInterval(floating-point interval)	None	Set the major tic interval for this Axis to interval
setMinorTicInterval(floating-point interval)	None	Set the minor tic interval for this Axis to interval
setNumTicLabelLevels(integer layers)	None	Set the maximum number of tic label layers to specified number. -1 is unrestricted. Used mostly for time series axis.
setReversed(Boolean state)	None	Set the reversed state of the Axis.[1]
setScaleLimits(floating-point min, floating-point max)	None	Sets the minimum and maximum values for the axis (range of un-zoomed view)
setTicColor(String colorString)	None	Set the tic color to the color represented by colorString

Method	Returns	Description
setTicTextColor(String colorString)	None	Set the tic text color to the color represented by colorString
setViewLimits(floating-point min, floating-poin max)	None	Zooms based on world coordinates
unZoom()	None	Returns the view to the full axis range.
zoomByFactor(floating-point factor)	None	Change the zoom scaling by the given factor

[1] The coordinate system used is a graphics coordinate system with the origin (0,0) located at the top left corner of the display, with X increasing to the right and Y increasing downward. The reversed state is in respect to this coordinate system (i.e. Y is reversed if it increases upward).

The script in Example 27 reads a data set from a DSS file, adds that data set to a new Plot, and zooms in on the Y Axis.

Example 27: Using Axis Objects

```
from hec.script import *                          # for Plot class
from hec.heclib.dss import *                      # for DSS class
thePlot = Plot.newPlot()                          # create a Plot
dssFile = HecDss.open("C:/mydb.dss")     # open the DSS file
flow = dssFile.get("/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/")
                                                  # read a data set
thePlot.addData(flow)                             # add the data set
thePlot.showPlot()                                # show the plot
viewport0 = thePlot.getViewport(0)                # get the first Viewport
yaxis = viewport0.getAxis("Y1")                   # get the Y1 axis
yaxis.setScaleLimits(0., 25000.)                  # set the scale
yaxis.zoomByFactor(."5")                          # zoom in
```

8.10.9 AxisTics Class

AxisTics methods are described in Table 8.22.

The script in Example 28 creates a new Plot with a data set read from DSS and tells the data set's axis tics to draw its minor tic marks.

Table 8.22 AxisTics Methods

Method	Returns	Description
areMajorTicLabelsVisible()	Boolean	Return whether the major tic labels are visible.
areMajorTicsVisible()	Boolean	Return whether the major tics are visible
areMinorTicLabelsVisible()	Boolean	Return whether the minor tic labels are visible
areMinorTicsVisible()	Boolean	Return whether the minor tics are visible

Method	Returns	Description
computeRatingFromOppositeAxis()	None	When used on the right (Y2) AxisTics object, with related curves on the Y1 and Y2 axes (e.g. stage and flow, or elevation and storage), causes the AxisTics to behave in a non-linear fashion such that Y1 and Y2 curves are coincident.
editProperties()	None	Display the Edit Properties Dialog for the AxisTics
getAxis()	Axis	Returns a reference to the axis that this object draws
getAxisTicColor()	Color	Return the tic color
getAxisTicColorString()	String	Return the tic color as a String
getFontSizes()	tuple of 3 integers	Return the regular, tiny, min and max font sizes for this AxisTics
getMajorTicLength()	Integer	Return the major tic length
getMinorTicLength()	integer	Return the minor tic length
setAxisTicColor(string colorString)	None	Set the tic color to the color represented by colorString
setFontSizes(integer sz,integer tiny, integer min, intege max)	None	Set the regular, tiny, min and max font sizes for this AxisTics
setMajorTicLabelsVisible(Boolean state)	None	Set the visibility of the major tic labels
setMajorTicLength(int ticLength)	None	Set the major tic length
setMajorTicsVisible(Boolean state)	None	Set the visibility of the major tics
setMinorTicLabelsVisible(Boolean state)	None	Set the visibility of the minor tic labels.
setMinorTicLength(int ticLength)	None	Set the minor tic length
setMinorTicsVisible(Boolean state)	None	Set the visibility of the minor tics

Example 28: Using AxisTics Objects

```
from hec.script import *                    # for Plot class
from hec.script.Constants import TRUE, FALSE
from hec.heclib.dss import *                # for DSS class
thePlot = Plot.newPlot()                    # create a Plot
dssFile = HecDss.open("C:/mydb.dss")        # open the DSS file
flow = dssFile.get("/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/")
                                            # read a data set
thePlot.addData(flow)                       # add the data set
thePlot.showPlot()                          # show the plot
viewport0 = thePlot.getViewport(flow)       # get the viewport for the #flow data set
yAxisTics = viewport0.getAxisTics("Y1")     # get the axis tics for the #Viewport
yAxisTics.setMinorTicsVisible(TRUE)         # tell axis tics to show tics
```

8.10.10 G2dLine Class

G2dLine methods are described in Table 8.23.

Table 8.23 G2dLine Methods

Method	Returns	Description
areSymbolsAutoInterval()	Boolean	Return whether the symbols for this line are placed at program-decided intervals
areSymbolsVisible()	Boolean	Return whether this line draws its symbols
editLineProperties()	None	Method that allows the editing of line properties. This method displays a visible dialog for line editing.
getFillColor()	Color	Return the fill color for this line
getFillColorString()	string	Return the fill color for this line as a String
getFillPatternString()	string	Return the fill pattern for this line as a String
getFillTypeString()	string	Return the Fill type for this line as a String.
getFirstSymbolOffset()	integer	Return the offset for the first symbol for this line
getLineColor()	Color	Return the line color for this line
getLineColorString()	string	Return the line color for this line as a String
getLineStepStyleString()	string	Return the line step style for this line as a String
getLineStyleString()	string	Return the line style for this line as a string
getLineWidth()	floating-point	Return the Line Width of the line
getNumPoints()	integer	Returns the Number of Points that this line has
getSymbolFillColor()	Color	Return the symbol fill color for this line's symbols
getSymbolFillColorString()	string	Return the symbol fill color for this line's symbols as a String
getSymbolInterval()	integer	Return the interval of data points (>0) on which symbols are drawn.
getSymbolLineColor()	Color	Return the symbol line color for this line's symbols
getSymbolLineColorString()	string	Return the symbol line color for this line's symbols as a String
getSymbolSize()	floating-point	Return the symbol size for this line

Method	Returns	Description
getSymbolSkipCountl()	integer	Return the number of points skipped between symbols (same as getSymbolInterval() – 1)
getSymbolTypeString()	string	Return the symbol type for this line as a string
isLineVisible()	Boolean	Return this line is drawn
setFillColor(string fillColor)	None	Set the fill color for this line
setFillPattern(string fillPattern)	None	Set the fill pattern for this line
setFillType(string fillType)	None	Set the Fill type for this line
setFirstSymbolOffset(integer offset)	None	Set the offset for first symbol for this line
setLineColor(string lineColor)	None	Set the line color for this line
setLineStepStyle(string stepStyle)	None	Set the line step style for this line
setLineStyle(string style)	None	Set the line style for this line
setLineVisible(Boolean state)	None	Set whether to draw this line
setLineWidth(floating-point width)	None	Set the width for this line
setSymbolFillColor(string symbolFillColor)	None	Set the symbol fill color for this line's symbols
setSymbolInterval(integer interval)	None	Set the interval of data points (>0) on which symbols are drawn.
setSymbolLineColor(string symbolLineColor)	None	Set the symbol line color for this line's symbols
setSymbolsAutoInterval(Boolean state)	None	Set whether to have the program decide the interval at which to draw symbols
setSymbolSize(floating-point size)	None	Set the symbol size for this line
setSymbolSkipCount(integer count)	None	Set the number of points skipped between symbols.
setSymbolsVisible(Boolean state)	None	Set whether to draw the symbols for this line
setSymbolType(string symbolType)	None	Set the symbol type for this line

The script in Example 29 creates a plot with a data set read from DSS, the script then tells that data set's curve to draw its symbols auto skipped.

Example 29: Using G2dLine Objects

```
from hec.script import *                                    # for Plot class
from hec.script.Constants import TRUE, FALSE
from hec.heclib.dss import *                                # for DSS class
thePlot = Plot.newPlot()                                    # create a Plot
dssFile = HecDss.open("C:/mydb.dss")                        # open the DSS file
flow = dssFile.get("/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/")
                                                            # read a data set
thePlot.addData(flow)                                       # add the data set
thePlot.showPlot()                                          # show the plot
stageCurve = thePlot.getCurve(stage)                        # get the stage curve
stageCurve.setSymbolsAutoInterval(TRUE)                     # turn on symbols auto skip
```

8.10.11 G2dLabel, G2dTitle, and AxisLabel Classes

G2dLabel, G2dTitle and AxisLabel methods are described in Table 8.24.

Table 8.24 Label Methods

Method	Returns	Description
editProperties()	None	Display the Edit Properties Dialog for the label
getAlignmentString()	string	Return the text alignment for this label as a String
getBackground()	Color	Return the background color for the label[1]
getBackgroundString()	string	Return the background color for the label as a String[1]
getBorderStyleString()	string	Return the border style for this label as a string
getBorderWeight()	floating-point	Return the border weight for this label
getFillColor()	Color	Return the fill color for this label as a Color[1]
getFillColorString()	string	Return the fill color for this label as a string[1]
getFillPatternString()	string	Return the fill pattern for this label as a string[1]
getFontFamily()	string	Return the font family for the label
getFontSize()	integer	Return the font size for the label
getFontSizes()	tuple of 3 integers	Return the regular, tiny, min and max font sizes for this label
getFontString()	string	Return the font for the label as a string[2].
getFontStyleString()	string	Return the font style for the label as a String
getForeground()	Color	Return the foreground color for the label
getForegroundString()	string	Return the foreground color for the label as a String
getIcon()	Icon	Return the Icon to display for this label
getIconPath()	string	Return the Icon path to display for this label
getRotation()	integer	Return the text rotation for this label
getSpacing()	integer	Return the spacing around this label
getText()	string	Return the text for the label
isBackgroundVisible()	Boolean	Return whether the background is visible
isBorderVisible()	Boolean	Return whether the border is visible
setAlignment(string alignment)	None	Set the text alignment for this label

Method	Returns	Description
setBackground(string colorString)	None	Set the background color for the label[1]
setBackgroundVisible(Boolean state)	None	Set the background visibility for the label
setBorderColor(string colorString)	None	Set the border color for this label
setBorderStyle(string style)	None	Set the border style for this label
setBorderVisible(Boolean state)	None	Set the border visibility for this label
setBorderWeight(floating-point weight)	None	Set the border weight for this label
setFillColor(string color)	None	Set the fill color for this label[1]
setFillPattern(string pattern)	None	Set the fill pattern for this label[1]
setFont(string font)	None	Set the font for the label[2].
setFontFamily(string fam)	None	Set the font family for the label
setFontSize(integer sz)	None	Set the font size for the label
setFontSizes(integer sz, integer tiny, integer min, integer max)	None	Set the regular, tiny, min and max font sizes for this label
setFontStyle(string style)	None	Set the font style for the label
setForeground(string colorString)	None	Set the foreground color for the label
SetIcon(Icon icon)	None	Set the Icon to display for this label
SetIcon(string iconPath)	None	Set the Icon to display for this label
setRotation(integer rotation)	None	Set the text rotation for this label
setSpacing(integer space)	None	Set the spacing around this label
SetText(string text)	None	Set the text for the label

[1] In the current version, fill color and background color are synonymous (e.g. fills are performed with the background color). Future version may support separate fill and background colors.
[2] Fonts are specified as name[,style[,size]] where style is Plain, Bold, Italic, or Bold Italic (e.g. "Arial,BoldItalic,12", "Lucida Console,Plain,10").

The script in Example 30 creates a plot from a DSS data set and sets the Y1 axis label text to blue.

Example 30: Using AxisLabel Objects

```
from hec.script import *                          # for Plot class
from hec.heclib.dss import *                        # for DSS class
thePlot = Plot.newPlot()                            # create a Plot
dssFile = HecDss.open("C:/mydb.dss")                # open the DSS file
flow = dssFile.get("/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/")
                                                    # read a data set
thePlot.addData(flow)                               # add the data set
thePlot.showPlot()                                # show the plot
viewport0 = thePlot.getViewport(0)                # get the first viewport
yaxislabel = viewport0.getAxisLabel("Y1")         # get the Y1 axis label
yaxislabel.setForeground("blue")                  # set the Y1 axis label text to blue
```

8.10.12 Templates

Template files saved interactively from HEC-DSSVue may be applied to plots via scripting. When saving a template interactively from the plot window via the "Save Template…" entry on the "File" menu, HEC-DSSVue:

1. Chooses the "My Documents" subdirectory of the directory specified in the USERPROFILE environment variable as the default location for the template file.
2. Appends ".template" to the end of the specified file name.

The *applyTemplate(string filename)* G2dDialog method requires the actual file name for the template file. To apply a template saved in the default directory, the complete template file name must be re-created as demonstrated in Example 31.

Example 31: Applying Template Saved in Default Directory

```
import os                                                    # for getenv() & sep
from hec.script import *                                     # for Plot class
from hec.heclib.dss import *                                 # for DSS class
thePlot = Plot.newPlot()                                     # create a Plot
dssFile = HecDss.open("C:/mydb.dss")                         # open the DSS file
flow = dssFile.get("/BASIN/LOC/FLOW/01NOV2002/1HOUR/OBS/")
                                                             # read a data set
thePlot.addData(flow)                                        # add the data set
thePlot.showPlot()                                           # show the plot
templateName = "myTemplate"                                  # template base name
templateFileName =                          \                # re-create the file name
  os.getenv("userprofile")                  \
  + os.sep                                  \
  + "My Documents"                          \
  + os.sep                                  \
  + templateName                            \
  + ".template"
thePlot.applyTemplate(templateFileName)                      # apply the template
```

8.11 Plot Component Properties

The following tables are the valid values to be used when calling plot related functions that take a color (setBackground(string color), etc…), an alignment (setAlignment()), a rotation (setRotation()), a fill pattern (setFillPattern()), a fill type (setFillType()), a line style (setLineStyle()), a step style (setLineStepStyle()).or a symbol type (setSymbolType()).

8.11.1 Colors

Colors can be specified either by a *String* or by a *java.awt.Color* object. If setting a color through the use of a *String* object the String can either be a standard color name (i.e. "darkred") or an RGB string (i.e. "255,20,20"). Standard color names are listed in Table 8.25.

Table 8.25 Standard Colors

black	darkmagenta	green	lightorange	orange
blue	darkorange	lightblue	lightpink	pink
cyan	darkpink	lightcyan	lightpurple	purple
darkblue	darkpurple	lightgray	lightred	red
darkcyan	darkred	lightgreen	lightyellow	white
darkgray	darkyellow	lightmagenta	magenta	yellow
darkgreen	gray			

8.11.2 Alignment

Supported text alignments are: Left, Center, and Right.

8.11.3 Positions

Supported text positions are: Above, Center, and Below.

8.11.4 Rotation

Supported text rotation values are: 0, 90, 180, 270.

8.11.5 Fill Patterns

Supported fill patterns are listed inTable 8.26.

Table 8.26 Fill Patterns

Solid	Horizontal	Vertical
Cross	FDiagonal	BDiagonal
Diagonal Cross	None	

8.11.6 Fill Types

Supported fill types are: None, Above, and Below.

8.11.7 Line Styles

Supported line style values are listed in Table 8.27.

Table 8.27 Line Styles

Solid	Dash	Dot
Dash Dot	Dash Dot-Dot	

8.11.8 Step Style

Supported step style values are: Normal, Step, and Cubic.

8.11.9 Symbol Types

Supported symbol type values are listed in Table 8.28.

Table 8.28 Symbol Types

Asterisk	Backslash	Backslash Square
Circle	Diamond	Forwardshlash
Forwardslash Square	Hash	Hash Diamond
Hash Square	Hash Triangle	Hash Triangle2
Hourglass	Open Circle	Open Diamond
Open Hourglass	Open Square	Open Triangle
Open Triangle2	Pipe	Pipe Diamond
Pipe Square	Plus	Plus Circle
Plus Diamond	Plus Square	Square
Triangle	Triangle2	X
X Circle	X Square	X Triangle
X Triangle2		

8.12 Tables

Tables allow you to view data in a vertical scrolling window that shows the ordinates, the dates and times and the values for the selected data sets.

8.12.1 Tabulate Class

```
Tabulate.newTable()
Tabulate.newTable(string title)
```

The Tabulate class in the *hec.script* module is used to create a new Table dialog. It contains two functions to create a Table dialog, each of which returns as a HecDataTableFrame object.

Creation of a table is illustrated in Example 32.

Example 32: Creating a Table

```
from hec.script import *
myTable = Tabulate.newTable()
or
from hec.script import *
myTable = Tabulate.newTable("Elevation vs Flow")
```

8.12.2 HecDataTableFrame Class

HecDataTableFrame methods are described in Table 8.29.

Table 8.29 HecDataTableFrame Methods

Method	Returns	Description
addData(DataContainer dc)	integer	Adds Data Set to the table.
close()	None	Closes the table
export()	None	Brings up the Table Export Options dialog.
export(string fileName, TableExportOptions options)	None	Exports table to specified file with specified options
exportAsHTML(string fileName)	None	Exports table in HTML format to specified file with no title and elements indented with tabs
exportAsHTML(string fileName, string title, string indent)	None	Exports table in HTML format to specified file with specified title and indentation string
exportAsXML(string fileName)	None	Exports table in XML format to specified file with no title and elements indented with tabs
exportAsXML(string fileName, string title, string indent)	None	Exports table in XML format to specified file with specified title and indentation string
getCellBackground(integer row, integer column)	Color	Returns the background color of the specified cell as a Color
getCellBackgroundString(integer row, integer column)	string	Returns the background color of the specified cell as a string
getCellForeground(integer row, integer column)	Color	Returns the foreground color of the specified cell as a Color
getCellForegroundString(integer row, integer column)	string	Returns the foreground color of the specified cell as a string

Method	Returns	Description
getColumn(DataContainer dc)	integer	Returns the number of the column that contains the specified data, if the parameter is time series data, or the number of the column that contains the x-ordinates if the parameter is paired data
getColumn(string header)	integer	Returns the number of the column that has the specified header text. Line breaks in the header text are specified as "\n"
getColumnBackground(integer column)	Color	Returns the background color of the specified column as a Color
getColumnBackgroundString(integer column)	string	Returns the background color of the specified column as a string
getColumnForeground(integer column)	Color	Returns the foreground color of the specified column as a Color
getColumnForegroundString(integer column)	string	Returns the foreground color of the specified column as a string
getColumnHeaderBackgroung(integer column)	Color	Returns the background color of the header of the specified column as a Color
getColumnHeaderBackgroungString(integer column)	string	Returns the background color of the header of the specified column as a string
getColumnHeaderFontString(integer column)	string	Returns the font of the header of the specified column as a string.[1]
getColumnHeaderForegroung(integer column)	Color	Returns the foreground color of the header of the specified column as a Color
getColumnHeaderForegroungString(integer column)	string	Returns the foreground color of the header of the specified column as a string
getColumnLabel(integer colNum)	string	Returns the column header text for the specified column
getColumnLabels()	list of strings	Returns the column header text for all columns
getColumnWidth(integer colNum)	integer	Returns the width of the specified column in pixels
getColumnWidths()	list of integers	Returns a list of all the column widths in pixels
getCommasState()	Boolean	Get whether the commas are shown
getDateTimeAsTwoColumnsState()	Boolean	Get whether date/time columns are shown as 1 or 2 columns in the table

Method	Returns	Description
getExportString(TableExportOptions options)	string	Returns a string representation of the table exported according to the specified options
getHeight()	integer	Return the height of the table in screen coordinates.
getHTMLExportString()	string	Returns a string representation of the table exported in HTML format with no title and elements indented with tabs
getHTMLExportString(string title, string indent)	string	Returns a string representation of the table exported in HTML format with the specified title and indentation string
getLocation()	Point	Returns the location of the table in screen coordinates[2].
getRowBackground(integer row)	Color	Returns the background color of the specified row as a Color
getRowBackgroundString(integer row)	string	Returns the row background color of the specified column as a string
getRowForeground(integer row)	Color	Returns the foreground color of the specified row as a Color
getRowForegroundString(integer row)	string	Returns the row foreground color of the specified column as a string
getSize()	Dimension	Return the dimensions of the table in screen coordinates.
getTableTitle()	G2dTitle	Returns the title of the HecDataTableFrame object as a G2dTitle object.
getTableTitleText()	string	Returns the title of the HecDataTableFrame object as a string.
getWidth()	integer	Return the width of the table in screen coordinates.
GetXMLExportString()	string	Returns a string representation of the table exported in XML format with no title and elements indented with tabs
getXMLExportString(string title, string indent)	string	Returns a string representation of the table exported in XML format with the specified title and indentation string
hide()	None	Hide the table
iconify()	None	Minimize (iconify) the table
maximize()	None	Maximize the table
minimize()	None	Minimize (iconify) the table

Method	Returns	Description
newTable()	HecDataTableFrame	Static, same as Tabulate.newTable()
newTable(string title)	HecDataTableFrame	Static, same as Tabulate.newTable(title)
print()	None	Display the print table
restore()	None	Restore the table from a minimized or maximized state
setCellBackgound(integer row, integer column, string color)	None	Sets the background color of the specified cell to the specified color
setCellForeground(integer row, integer column, string color)	None	Sets the foreground color of the specified cell to the specified color
setColumnBackgound(integer column, string color)	None	Sets the background color of the specified column to the specified color
setColumnForeground(integer column, string color)	None	Sets the foreground color of the specified column to the specified color
setColumnHeaderBackgound(integer column, string color)	None	Sets the background color of the header of the specified column to the specified color
setColumnHeaderFont(integer column, string font)	None	Sets the font of the header of the specified column to the specified font.[1]
setColumnHeaderForeground(integer column, string color)	None	Sets the foreground color of the header of the specified column to the specified color.
setColumnLabel(integer column, string label)	None	Sets the column header text of the specified column to the specified label.
setColumnLabels(list labels)	None	Sets the column header text of all columns to the labels specified in the list of strings
setColumnPrecision(integer colNum, integer precision)	None	Sets the number of decimal places to display for the specified column
setColumnWidth(integer colNum, integer width)	None	Sets the width of the specified column in pixels
setColumnWidths(list widths)	None	Sets the width in pixels of all the columns to those specified in the parameter (list of integers)
setCommasState(Boolean showCommas)	None	Set state to show commas or not
setDateTimeAsTwoColumnsState(integer showDateTimeAs2Columns)	None	Set whether date/time columns should show as 1 or 2 columns in the table

Method	Returns	Description
setLocation(integer x, integer y)	None	Sets the location of the table in screen coordinates[2].
setSize(int width, int height)	None	Sets the size of the table in screen coordinates.
setRowBackgound(integer row, string color)	None	Sets the background color of the specified row to the specified color
setRowForeground(integer row, string color)	None	Sets the foreground color of the specified row to the specified color
setTableTitleText(string title)	None	Sets the title of the HecDataTableFrame object.
showTable()	None	Show the table

[1] Fonts are specified as *name*[,*style*[,*size*]] where *style* is Plain, Bold, Italic, or Bold Italic (e.g. "Arial,BoldItalic,12", "Lucida Console,Plain,10").

[2] The coordinate system used is a graphics coordinate system, where X increases to the right and Y increases downward from the origin (0,0) which is located in the top left corner of the display. Locations set or retrieved refer to the top left corner of the plot in reference to this coordinate system.

8.13 TableExportOptions Class

TableExportOptions objects hold complete descriptions of options for exporting tables to fixed-column-width or column-delimited text files. **TableExportOptions** objects have fields that are settable by the user to create marker lines of various styles. New **TableExportOptions** objects are created by calls to TableExportOptions() (e.g., *myOptions = TableExportOptions()*). **TableExportOptions** fields are described in Table 8.30.

Table 8.30 TableExportOptions Fields

Field	Type	Description	Default
delimiter	string (one character)	Placed between fields if fixedWidthCol is False	'\t' (tabcharacter)
quotedStrings	Boolean	Specifies whether to enclose text in quotes	False
title	string	Title of the table	None
fixedWidthCols	Boolean	Specifies whether fields are exported as fixed-width columns or fields separated by delimiter	False
columnHeader	Boolean	Specifies whether the column headers are to be exported	True
rowHeader	Boolean	Specifies whether the row headers are to be exported	False
gridLines	Boolean	Specifies whether the table will be exported with text "lines" between the rows and columns	False

The script in Example 33 creates a table from two DSS data sets and exports the table as a comma-separated-value text file.

Example 33: Filling, Displaying and Exporting a Table

```
from hec.heclib.dss import *                                # for DSS
from hec.script import *                                    # for Tabulate
file = “C:/mydb.dss"                                        # specify the DSS file
dssfile = HecDss.open(file)                                 # open the file
# read 2 records
stage = dssfile.get("//AXEMA/STAGE/01OCT2001/1HOUR/OBS/")
flow = dssfile.get("//AXEMA/FLOW/01OCT2001/1HOUR/OBS/")
theTable = Tabulate.newTable()                              # create the table
theTable.setTitle("Test Table")                             # set the table title
theTable.setLocation(5000,5000)                             # set the location of the table
off
                                                            the screen
theTable.addData(flow)                                      # add the data
theTable.addData(stage)
theTable.showTable()                                        # show the table
flowCol = theTable.getColumn(flow)                          # adjust columns
stageCol = theTable.getColumn(stage)
flowWidth = theTable.getColumnWidth(flow)
stageWidth = theTable.getColumnWidth(stage)
theTable.setColumnPrecision(flowCol, 0)
theTable.setColumnPrecision(stageCol, 2)
theTable.setColumnWidth(flowCol, flowWidth - 10)
theTable.setColumnWidth(stageCol, stageWidth + 10)
# get new export options
# delimit with commas
# set the title
# set the output file name
# export to the file
# close
```

8.14 HecMath Class

The objects returned from the *HecDss*..read(…) methods and supplied to the *HecDss*.write(…) method are HecMath objects. There are currently three types of HecMath classes: TimeSeriesMath class, PairedDataMath class, and StreamRatingMath class which represent time-series data, paired data, and stream rating data, respectively.

An HecMath object can be created for writing to new DSS data by first creating a new DataContainer as discussed in Sections 8.8.1 (TimeSeriesContainer Class) and 8.8.2 (), and then calling the HecMath.createInstance() function with the DataContainer object as the only parameter (e.g. myTimeSeriesMath = HecMath.createInstance(myTimeSeriesContainer)). The methods for HecMath Objects, which are described in Section 8.14, are listed inTable 8.31.

Table 8.31 HecMath Methods

Method	Returns	Description Section
abs()	HecMath	8.15.1
accumulation()	TimeSeriesMath	8.15.2
acos()	HecMath	8.15.3
add(floating-point constant)	HecMath	8.15.4
add(TimeSeriesMath dataset)	TimeSeriesMath	8.15.5
applyMultipleLinearRegression(string startTimeString, string endTimeString, sequence datasets, floating-point minLimit, floating-point maxLimit)	TimeSeriesMath	8.15.6
asin()	HecMath	8.15.7
atan()	HecMath	8.15.8
ceil()	HecMath	8.15.9
centeredMovingAverage(integer number, Boolean onlyValid, Boolean useReduced)	TimeSeriesMath	8.15.10
conicInterpolation(TimeSeriesMath dataset, string inputType, string outputType, floating-point storageFactor)	TimeSeriesMath	8.15.11
convertToEnglishUnits()	HecMath	8.15.12
convertToMetricUnits()	HecMath	8.15.13
correlationCoefficients(TimeSeriesMath dataset)	LinearRegression Statistics	8.15.14
cos()	HecMath	8.15.15
cyclicAnalysis()	sequence of TimeSeriesMath	8.15.16
decayingBasinWetnessParameter(TimeSeries Math tsPrecip, floating-point decayRate)	TimeSeriesMath	8.15.17
divide(floating-point constant)	HecMath	8.15.18
divide(TimeSeriesMath dataset)	TimeSeriesMath	8.15.19
estimateForMissingPrecipValues(integer maxMissing)	TimeSeriesMath	8.15.20
estimateForMissingValues(integer maxMissing)	TimeSeriesMath	8.15.21
exp()	HecMath	8.15.22
exponentiation(floating-point constant)	HecMath	8.15.23
exponentiation(HecMath tsMath)	HecMath	8.15.24
extractTimeSeriesDataForTimeSpecification(string timeLevel, string range, Boolean isInclusive, integer intervalWindow, Boolean setAsIrregular)	TimeSeriesMath	8.15.25
firstValidDate()	integer	8.15.26
firstValidValue()	floating-point	8.15.27
floor()	HecMath	8.15.28

Method	Returns	Description Section
flowAccumulatorGageProcessor(TimeSeries Math dataset)	TimeSeriesMath	8.15.29
fmod(HecMath tsMath)	HecMath	8.15.30
forwardMovingAverage(integer number)	TimeSeriesMath	8.15.31
forwardMovingAverage(integer numberToAverageOver, Boolean OnlyValidValues, Boolean useReduced)	Hecmath	8.15.32
generateDataPairs(TimeSeriesMath dataset, Boolean sort)	PairedDataMath	8.15.33
generateRegularIntervalTimeSeries(string startTime, string endTime, string timeInterval, string timeOffset, floating-point initialValue)	TimeSeriesMath	8.15.34
getData()	DataContainer	8.15.35
getType()	string	8.15.36
getUnits()	string	8.15.37
gmean(list of HecMath tsMathArray)	HecMath	8.15.38
hmean(list of HecMath tsMathArray)	HecMath	8.15.39
integerDivide(floating-point constant)	HecMath	8.15.40
integerDivide(HecMath tsMath)	HecMath	8.15.41
interpolateDataAtRegularInterval(string timeInterval, string timeOffset)	TimeSeriesMath	8.15.42
inverse()	HecMath	8.15.43
isEnglish()	Boolean	8.15.44
isMetric()	Boolean	8.15.45
isMuskingumRoutingStable(integer subReachCount, floating-point muskingumK, floating-point muskingumX)	string	8.15.46
lastValidDate()	integer	8.15.47
lastValidValue()	floating-point	8.15.48
log()	HecMath	8.15.49
log10()	HecMath	8.15.50
max()	floating-point	8.15.51
max(list of HecMath tsMathArray)	HecMath	8.15.52
maxDate()	integer	8.15.53
mean()	floating-point	8.15.54
mean(list of HecMath tsMathArray)	HecMath	8.15.55
med(list of HecMath tsMathArray)	HecMath	8.15.56
mergePairedData(PairedDataMath dataset)	PairedDataMath	8.15.57
mergeTimeSeries(TimeSeriesMath dataset)	TimeSeriesMath	8.15.58
min()	floating-point	8.15.59
min(list of HecMath tsMathArray)	HecMath	8.15.60
minDate()	integer	8.15.61

Method	Returns	Description Section
modifiedPulsRouting(TimeSeriesMath dataset, integer subReachCount, floating-point muskingumX)	TimeSeriesMath	8.15.62
modulo(floating-point constant)	HecMath	8.15.63
modulo(HecMath tsMath)	HecMath	8.15.64
multipleLinearRegression(sequence datasets, floating-point minLimit, floating-point maxLimit)	PairedDataMath	8.15.65
multiply(floating-point constant)	HecMath	8.15.66
multiply(TimeSeriesMath dataset)	TimeSeriesMath	8.15.67
muskingumRouting(integer subReachCount, floating-point muskingumK, floating-point muskingumX)	TimeSeriesMath	8.15.68
neg()	HecMath	8.15.69
numberInvalidValues()	integer	8.15.70
numberMissingValues()	integer	8.15.71
numberQuestionedValues()	integer	8.15.72
numberRejectedValues()	integer	8.15.73
numberValidValues()	integer	8.15.74
olympicSmoothing(integer number, Boolean onlyValid, Boolean useReduced)	TimeSeriesMath	8.15.75
p1(list of HecMath tsMathArray)	HecMath	8.15.76
p2(list of HecMath tsMathArray)	HecMath	8.15.77
p5(list of HecMath tsMathArray)	HecMath	8.15.78
p10(list of HecMath tsMathArray)	HecMath	8.15.79
p20(list of HecMath tsMathArray)	HecMath	8.15.80
p25(list of HecMath tsMathArray)	HecMath	8.15.81
p75(list of HecMath tsMathArray)	HecMath	8.15.82
p80(list of HecMath tsMathArray)	HecMath	8.15.83
p89(list of HecMath tsMathArray)	HecMath	8.15.84
p90(list of HecMath tsMathArray)	HecMath	8.15.85
p95(list of HecMath tsMathArray)	HecMath	8.15.86
p99(list of HecMath tsMathArray)	HecMath	8.15.87
periodConstants(TimeSeriesMath dataset)	TimeSeriesMath	8.15.88
polynomialTransformation(TimeSeriesMath dataset)	TimeSeriesMath	8.15.89
polynomialTransformationWithIntegral(TimeSeriesMath dataset)	TimeSeriesMath	8.15.90
product(list of HecMath tsMathArray)	HecMath	8.15.91

Method	Returns	Description Section
ratingTableInterpolation(TimeSeriesMath dataset)	TimeSeriesMath	8.15.92
replaceSpecificValues(HecDouble from, HecDouble to)	HecMath	8.15.93
reverseRatingTableInterpolation(TimeSeriesMath dataset)	TimeSeriesMath	8.15.94
rms(list of HecMath tsMathArray)	HecMath	8.15.95
round()	HecMath	8.15.96
roundOff(integer digits, integer place)	HecMath	8.15.97
screenWithConstantValue(String durationStr, floating-point rejectTolerance, floating-point questionTolerance, floating-point minThreshold, integer maxMissing)	HecMath	8.15.98
screenWithDurationMagnitude(String durationStr, floating-point minRejectLimit, floating-point minQuestionLimit, floating-point maxRejectLimit, floating-point maxQuestionLimit)	HecMath	8.15.99
screenWithForwardMovingAverage(integer numberToAverageOver, floating-point changeLimit)	HecMath	8.15.100
screenWithForwardMovingAverage(integer number, floating-point changeLimit, Boolean setMissing, string invalidQuality)	TimeSeriesMath	8.15.101
screenWithMaxMin(floating-point min, floating-point max, floating-point rate, Boolean setMissing, string invalidQuality)	TimeSeriesMath	8.15.102
screenWithMaxMinfloating-point minValueLimit, floating-point maxValueLimit, floating-point changeLimit)	HecMath	8.15.103
screenWithMaxMin(floating-point minValueLimit, floating-point maxValueLimit, floating-point changeLimit, Boolean setInvalidToSpecified, floating-point invalidValueReplacement, string qualityFlagForInvalidValue)	HecMath	8.15.104
screenWithMaxMin(floating-point minRejectLimit, floating-point minQuestionLimit, floating-point maxQuestionLimit, floating-point maxRejectLimit)	HecMath	8.15.105
screenWithRateOfChange(floating-point minRejectLimit, floating-point minQuestionLimit, floating-point maxQuestionLimit, floating-point maxRejectLimit)	HecMath	8.15.106
setCurve(string name)	None	8.15.107

Method	Returns	Description Section
setCurve(integer number)	None	8.15.108
setData(DataContainer data)	None	8.15.109
setLocation(string location)	None	8.15.110
setParameter(string parameter)	None	8.15.111
setPathname(string pathname)	None	8.15.112
setTimeInterval(string interval)	None	8.15.113
setType(string type)	None	8.15.114
setUnits(string units)	None	8.15.115
setVersion(string version)	None	8.15.116
setWatershed(string watershed)	None	8.15.117
shiftAdjustment(TimeSeriesMath dataset)	TimeSeriesMath	8.15.118
shiftInTime(string timeShift)	TimeSeriesMath	8.15.119
sign()	HecMath	8.15.120
sin()	HecMath	8.15.121
skewCoefficient()	floating-point	8.15.122
snapToRegularInterval(string timeInterval, string timeOffset, string timeBackward, string timeForward)	TimeSeriesMath	8.15.123
sqrt()	HecMath	8.15.124
standardDeviation()	floating-point	8.15.125
standardDeviation (list of HecMath tsMathArray)	HecMath	8.15.126
straddleStaggerRouting(integer avgCount, integer lagCount, integer subReachCount)	TimeSeriesMath	8.15.127
subtract(floating-point constant)	HecMath	8.15.128
subtract(TimeSeriesMath dataset)	TimeSeriesMath	8.15.129
successiveDifferences()	TimeSeriesMath	8.15.130
sum()	floating-point	8.15.131
sum (list of HecMath tsMathArray)	HecMath	8.15.132
tan()	HecMath	8.15.133
timeDerivative()	TimeSeriesMath	8.15.134
transformTimeSeries(string timeInterval, string timeOffset, string functionType)	TimeSeriesMath	8.15.135
transformTimeSeries(TimeSeriesMath dataset, string functionType)	TimeSeriesMath	8.15.136
truncate()	HecMath	8.15.137
twoVariableRatingTableInterpolation(TimeSeriesMath dataset1, TimeSeriesMath dataset2)	TimeSeriesMath	8.15.138
variance(list of HecMath tsMathArray)	HecMath	8.15.139

8.15 Math Functions

Math functions are accessible through the general class called **HecMath**. **HecMath** objects hold data sets and allow you to perform mathematical operations on them. They can also be passed to plots and tables to display the data. A **HecMath** object is either a **TimeSeriesMath** object or a **PairedDataMath** object, which handle time series and paired data sets, respectively.

Before using **PairedDataMath** methods, be sure to read the description for the *setCurve* method. Paired data sets may contain multiple curves. The *setCurve* method provides user control over which paired data curve is operated upon by subsequent function calls.

8.15.1 Absolute Value

abs()

Derive a new time series or paired data set from the absolute value of values of the current data set. For time series data, missing values are kept as missing. For paired data sets, use the *setCurve* method to first select the paired data curve(s).

See also: `setCurve()`
Parameters: Takes no parameters
Example: `NewDataSet = dataSet.abs()`
Returns: A new **HecMath** object of the same type as the current object

8.15.2 Accumulation (Running)

accumulation()

Derive a new time series by computing a running accumulation of the current time series.

For time points in which the current time series value are missing, the value in the accumulation time series remains constant (same as the accumulated value at the last valid point location).

Parameters: Takes no parameters
Example: `NewTimeSeries = timeSeries.accumulation()`
Returns: A new **TimeSeriesMath** object

8.15.3 Arccosine Trigonometric Function

acos()

Derive a new time series or paired data set from the arccosine of values of the current data set. The resultant data set values are in radians. For time series data, missing values are kept as missing.

For paired data sets, use the *setCurve* (see Sections 8.15.07 and 8.15.08) function to first select the paired data curve (or all curves) to apply the function. By default the function is applied to all paired data curves.

See also: setCurve()
Example: `newDataSet = dataSet.acos()`
Parameters: Takes no parameters
Returns: A **HecMath** object of the same type as the current object

8.15.4 Add a Constant

add(floating-point constant)

Add the value constant to all valid values in the current time series or paired data set. For time series data, missing values are kept as missing.

For paired data, constant is added to y-values only. Use the *setCurve* method to first select the paired data curve(s).

See also: add(HecMath dataSet)
setCurve()
Example: `newDataSet = dataSet.add(2.5)`
Parameters: `constant` - A floating-point value
Returns: A new **HecMath** object of the same type as the current object

8.15.5 Add a Data Set

add(TimeSeriesMath tsData)

Add the values in the data set *tsData* to the values in the current data set. The function only applies to time series data sets.

When adding one time series data set to another, there is no restriction that times in the two data sets match exactly. However, only values with coincident times will be added. Times in the current time series data set that cannot be matched with times in the second data set are set to missing.

Values in the current data set that are missing are kept as missing. Either or both data sets may be regular or irregular interval time series. This function will not merge data sets. Use the *mergeTimeSeries* (for time series data sets) or the *mergePairedData* (for paired data sets) functions for this purpose.

See also: add(floating-point constant)
mergeTimeSeries(TimeSeriesMath)
mergePairedData(PairedDataMath)
Example: `newTsData = tsData.add(otherTsData)`
Parameters: `tsData` - A **TimeSeriesMath** object
Returns: A new **TimeSeriesMath** object

8.15.6 Apply Multiple Linear Regression Equation

```
applyMultipleLinearRegression(string startTimeString,
   string endTimeString,
   sequence tsDataSetSequence,
   floating-point minimumLimit,
   floating-point maximumLimit)
```

Apply the regression coefficients contained in the current paired data set to the array of time series data sets in **tsDataSetSequence** to develop a new time series data set. The *applyMultipleLinearRegression* function applies the multiple linear regression coefficients computed with the *multipleLinearRegression* function (see Section 8.15.65).

For the general linear regression equation, a dependent variable, Y, may be computed from a set independent variables, Xn:

$$Y = B0 + B1*X1 + B2*X2 + B3*X3$$

where Bn are linear regression coefficients.

For time series data sets, an estimate of the original time series data set values may be computed from a set of independent time series data sets using regression coefficients such that:

$$TsEstimate(t) = B0 + B1*TS1(t) + B2*TS2(t) + \ldots + Bn*TSn(t)$$

where Bn are the set of regression coefficients and TSn are the time series data sets contained in *tsDataSetSequence*.

The number of regression coefficients in the current **PairedDataMath** object must be one more than the number of independent time series data sets in *tsDataSetSequence*. The collection of selected time series data sets must be in the same order as when the regression coefficients were computed with the *multipleLinearRegression* method.

All the time series data sets must be regular interval and have the same time interval. The function filters the data to determine the time period common to all time series data sets and uses only those points in the regression analysis. For any given time, if a value is missing in any time series, the value in resultant time series is set to missing.

The parameters *minimumLimit* and *maximumLimit* can be used to specify the range of valid values for the resultant data set. Values which fall outside the specified range are set to missing. *minimumLimit* or *maximumLimi***t** may be entered as *Constants.UNDEFINED* to ignore the minimum or maximum value check.

If *startTimeString* or *endTimeString* are blank strings, the start and end time of the resultant time series will be defined by the time period common to all time series data sets in *tsDataSetSequence*. Otherwise the time series start and end may be defined using *startTimeString* and *endTimeStrin***g** which have the usual HEC time window format (e.g. "01JAN2001 1400").

Names, parameter type and unit labels for the new time series data set are copied over from the first time series data set in *tsDataSetSequence*. The F-part in the new data set is set to "COMPUTED".

Parameters:

`startTimeString` – A string containing an HEC time (e.g. "01JAN2001 1400") specifying the start time of the resultant time series data set. May be blank (" ")

`endTimeString` – A string containing an HEC time (e.g. "01JAN2001 1400") specifying the ending time of the resultant time series data set. May be blank (" ")

`tsDataSetSequence` – Sequence of **TimeSeriesMath** objects. Must all be regular interval and have the same time interval.

`minimumLimit` – A floating-point value specifying the minimum valid value in the resultant time series data set. Set to Constants.UNDEFINED to ignore this option.

`maximumLimit` – A floating-point value specifying the maximum valid value in the resultant time series data set. Set to Constants.UNDEFINED to ignore this option.

Example:

```
newTsData =
pairedData.applyMultipleLinearRegression(
“01Jan2000 0000”,
“31Dec2000 2300”,
(tsData1, tsData2, tsData3),
Constants.UNDEFINED,
Constants.UNDEFINED)
```

Returns: A new regular interval **TimeSeriesMath** object

Generated Exceptions: Throws a *HecMathException* if the number of data sets in *tsDataSetSequence* is not equal to the number of regression coefficients -1, or if the data sets in *tsDataSetSequence* are not regular interval time series data sets with the same interval time.

8.15.7 Arcsine Trigonometric Function

```
asin()
```

Derive a new time series or paired data set from the arcsine of values of the current data set. The resultant data set values are in radians. For time series data, missing values are kept as missing.

For paired data sets, use the *setCurve* (see Sections 8.15.107 and 8.15.108) function to first select the paired data curve (or all curves) to apply the function. By default the function is applied to all paired data curves.

See also: setCurve()
Example: `newDataSet = dataSet.asin()`
Parameters: Takes no parameters
Returns: A **HecMath** object of the same type as the current object

8.15.8 Arctangent Trigonometric Function

```
atan()
```

Derive a time series or paired data set computed from the arctangent of values of the current data set. For time series data, missing values are kept as missing. If the cosine of the current time series value is zero, the value is set missing.

For paired data sets, use the *setCurv*e method to first select the curve(s).

See also: setCurve()
Example: `newDataSet = dataSet.atan()`
Parameters: Takes no parameters
Returns: A new **HecMath** object of the same type as the current object

8.15.9 Ceiling Function

```
ceil()
```

Derive a time series or paired data set with values of the current time series rounded up to the nearest whole number that is greater to or equal to the value. For time series data, missing values are kept as missing.

For paired data sets, use the *setCurve* method to first select the curve(s).

See also: setCurve(), floor()
Example: `newDataSet = dataSet.ceil()`
Parameters: Takes no parameters
Returns: A new **HecMath** object of the same type as the current object

8.15.10 Centered Moving Average Smoothing

```
centeredMovingAverage(integer numberToAverageOver,
  boolean onlyValidValues,
  boolean useReduced)
```

Derive a new time series from the centered moving average of *numberToAverageOver* values in the current time series. numberToAverageOver must be an odd integer greater than two.

If *onlyValidValues* is set to true, then if any points in the averaging interval are missing, the point in the new time series is set to missing. If *onlyValidValues* is set to false and missing values are contained in the averaging interval, a smoothed point is still computed using the remaining valid values in the interval. If there are no valid values in the averaging interval, the point is set to missing.

If *useReduced* is set to true, then centered moving average points can still be computed at the beginning and end of the time series, even if there are less than *numberToAverageOver* values in the averaging interval. If *useReduced* is set to false, then the first and *last numberToAverageOver*/2 points of the resultant time series are set to missing.

Parameters:

`numberToAverageOver` – An integer containing the number of values to average over for computing the centered moving average. Must be odd and greater than two.

`onlyValidValues` – Either True or False, specifying whether all values in the averaging interval must be valid for the computed point in the new time series to be valid.

`useReduced` – Either True or False, specifying whether to allow points at the beginning and end of the resultant time series to be computed from a reduced (less than *numberToAverageOver*) set of points.

Example:

```
avgData = tsData.centeredMovingAverage (5, TRUE, TRUE)
```

Returns: A new **TimeSeriesMath** object
Generated Exceptions: Throws a *HecMathException* if the *numberToAverageOver* is less than three or not odd.

8.15.11 Conic Interpolation from Elevation/Area Table

```
conicInterpolation(TimeSeriesMath tsData,
  string inputType,
  string outputType,
  floating-point storageScaleFactor )
```

Use the conic interpolation table in the current paired data set to develop a new time series data set from the interpolation of **tsData**.

The current paired data should be an Elevation-Area table. However, the first data pair is the initial conic depth, and the storage value at the first elevation in the table. If the initial conic depth is undefined, the function will calculate a value. Elevation-Area values in the table must be in ascending order.

tsData is either a time series of reservoir elevation or storage. The type is specified by setting *inputType* as "S(TORAGE)" or "E(LEVATION)". The desired output time series type is similarly set using outputType. The valid settings for *outputTyp*e are "S(TORAGE)", "E(LEVATION)" or "A(REA)". *inputType* and *outputType* must not be the same.

storageScaleFactor is an optional parameter used to scale input (by multiplying) and output (by dividing) storage values. For example, if the area in the conic interpolation table is expressed in square feet, *storageScaleFactor* could be set to 43,560 to convert the storage output to acre-feet.

Parameter type in the new time series is set according to *outputType*. If the output time series values are elevation, the time series units are set to the paired data x-units label. If the output time series values are area, the time series units are set to the paired data y-units label. If the output is storage, the units are not set and should be set by the user with the *setUnits* function.

See also: setUnits()
Parameters:
`tsData` – A TimeSeriesMath object representing elevation or storage.

`inputType` – A string specifying the parameter type for the input time series, either "S(TORAGE)" or "E(LEVATION)". Only the first character of the string is interpreted by the function.
`outputType` – A string specifying the parameter type for the output time series, either "S(TORAGE)", "E(LEVATION)" or "A(REA)". Only the first character of the string is interpreted by the function.
`storageScaleFactor` – A floating-point number used to scale input (by multiplying) and output (by dividing) storage values.

Examples:

```
tsStorage =
conicElevAreaCurve.conicInterpolation(
tsElev,
“Elevation”,
“Storage”,
1.0)

tsArea =
conicElevAreaCurve.conicInterpolation(
tsElev,
“Elevation”,
“Area”,
1.0)
```

Returns: A **new TimeSeriesMath** object
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if *inputType* or *outputTyp*e cannot be interpreted as one of the allowed values; if *inputType* and *outputTyp*e are the same parameters; if values in the conic interpolation table are not in ascending order.

8.15.12 Convert Values to English Units

`convertToEnglishUnits``()`

Perform unit conversion of data values and unit labels in the current time series or paired data set from Metric (SI) units to English units. Determination of the unit system will be based upon the current units labels and parameter types. If the data units are already in English units or the unit system cannot be determined, no conversion occurs.
For paired data, both x and y values are converted. For time series data, missing values remain missing.

See also: convertToMetricUnits(), isEnglish(), isMetric()
Example: `englishDataSet= siDataSet.convertToEnglishUnits()`
Parameters: Takes no parameters
Returns: A **HecMath** object of the same type as the current object

8.15.13 Convert Values to Metric (SI) Units

`convertToMetricUnits``()`

Perform unit conversion of data values and unit labels in the current time series or paired data set from English units to Metric (SI) units. Determination of the unit system will be based upon the current units' labels and parameter types. If the units are already in Metric units or the unit system cannot be determined, no conversion occurs.

For paired data, both x and y values are converted. For time series data, missing values remain missing.

See also: convertToEnglishUnits(), isEnglish(), isMetric()
Parameters: Takes no parameters
Example: `siDataSet = englishDataSet.convertToMetricUnits()`
Returns: An **HecMath** object of the same type as the current object

8.15.14 Correlation Coefficients

`correlationCoefficients``(TimeSeriesMath tsData)`

Computes the linear regression and other correlation coefficients between data in the current time series and **tsData**. Values in the current time series and **tsData** are matched by time to form data pairs for the correlation analysis. The data sets may be either regular or irregular time interval data.

The correlations statistics computed by the function are:

Number of Valid Values
Regression Constant
Regression Coefficient
Determination Coefficient
Standard Error of Regression
Adjusted Determination Coefficient
Adjusted Standard Error of Regression

These values are contained in a **LinearRegressionStatistics** object. The current **TimeSeriesMath** object forms the values of the independent variable (x-values), while values of the second time series comprise the dependent variable (y-values). The linear regression coefficients thus express how values in the second data set can be derived from values in the primary data set:

$$TS2(t) = a + b * TS1(t)$$

where "a" is the regression constant and "b" the regression coefficient.

See also: LinearRegressionStatistics
Parameters: tsData - A **TimeSeriesMath** object that forms the dependent variable for the regression analysis
Example:

```
linearRegressionData =
tsData.correlationCoefficients(otherTsData)
```

Returns: A **LinearRegressionStatistics** object holding the correlation data.
Generated Exceptions: Throws an *hec.hecmath.HecMathException* if the times in the current time series do not exactly match times in **tsData**.

8.15.15 Cosine Trigonometric Function

cos()

Derive a new time series or paired data set from the cosine of values of the current data set. The resultant data set values are in radians. For time series data, missing values are kept as missing.

For paired data sets, use the *setCurve* (see Sections 8.15.107 and 8.15.108) function to first select the paired data curve (or all curves) to apply the function. By default the function is applied to all paired data curves.

See also: setCurve()
Example: `newDataSet = dataSet.cos()`
Parameters: Takes no parameters
Returns: A **HecMath** object of the same type as the current object

8.15.16 Cyclic Analysis (Time Series)

cyclicAnalysis()

Derive a set of cyclic statistics from the current regular interval time series data set. The time series data set must have a time interval of "1HOUR", "1DAY" or "1MONTH". The function sorts the time series values into statistical "bins" relevant to the time interval. Values for the 1HOUR interval data are sorted into twenty-four bins representing the hours of the day, 0100 to 2400. The 1DAY interval data is apportioned to 365 bins for the days of the year. The 1MONTH interval data is sorted into twelve bins for the months of the year.

The format of the resultant data sets is as a "pseudo" time series for the year 3000. For example, the cyclic analysis of one month of hourly interval data will produce pseudo time series data sets having twenty-four

hourly values for the day January 1, 3000. If the statistical parameter is the "maximum" value, then the twenty-four values represent the maximum value occurring at that hour of the day in the current time series. The cyclic analysis of daily interval data will produce pseudo time series data sets having 365 daily values for the year 3000. The cyclic analysis of monthly interval data will result in pseudo time series data sets having twelve monthly values for the year 3000.

Fourteen pseudo time series data sets are derived by the cyclic analysis function for the following statistical parameters:

Number of values processed for each time interval
Maximum value
Time of maximum value
Minimum value
Time of minimum value
Average value
Probability exceedence percentiles for 5%, 10%, 25%, 50% (median value), 75%, 90%, and 95%
Standard deviation

The fourteen pseudo time series of cyclic statistics are returned by the function as an array of time series data sets. The parameter part of the record path for each time series is modified to indicate the type of the statistical parameter. For a flow record, the parameter "FLOW" would become "FLOW-MAX" for the maximum values statistics, "FLOW-P5" for the five percent percentile statistics, etc.

Parameters: Takes no parameters
Example: `cyclicData = tsData.cyclicAnalysis()`
Returns: A sequence of fourteen **TimeSeriesMath** objects, each of which is a pseudo time series data sets representing a statistical parameter.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the time series is not regular interval or does not have a time interval of "1HOUR", "1DAY", or "1MONTH".

8.15.17 Decaying Basin Wetness Parameter

```
decayingBasinWetnessParameter(TimeSeriesMath tsPrecip,
floating-point decayRate)
```

Compute a time series of decaying basin wetness parameters from the regular interval time series data set of incremental precipitation, **tsPrecip,** by:

$$TSResult(t) = Rate * TSResult(t-1) + TSPrecip(t)$$

where Rate is **decayRate**, and 0 < Rate < 1.

The first value of the resultant time series data set, *TSResult(1)*, is set to the first value in the current time series data set. The current time series data set can be the same time series data set as **tsPrecip**. Missing values in the precipitation time series are taken as zero when applying the above equation.

Parameters:

`tsPrecip` – A regular interval **TimeSeriesMath** object representing precipitation

`decayRate` – a floating-point number in the range 0 < decayRate < 1.

Example:

```
tsWetness =
tsPrecip.decayingBasinWetnessParameter(
tsPrecip,
0.87)
```

Returns: A new **TimeSeriesMath** object

8.15.18 Divide by a Constant

`divide``(floating-point constant)`

Divide all valid values in the current time series or paired data set by the value constant. For time series data, missing values are kept as missing. For paired data, constant divides the y-values only. Use the *setCurve* method to select the paired data curve(s).

See also: divide(TimeSeriesMath tsData); setCurve()

Parameters:

`constant` - A floating-point value to divide the values in the current data set (cannot be zero)

Example: `newDataSet = dataSet.divide(1.1)`

Returns: A new **HecMath** object of the same type as the current object

8.15.19 Divide by a Data Set

`divide``(TimeSeriesMath tsData)`

Divide valid values in the current data set by the corresponding values in the data set **tsDat**a. Both data sets must be time series data sets.

When dividing one time series data set by another, there is no restriction that times in the two data sets match exactly. However, only values with coincident times will be divided. Times in the current time series data set that cannot be matched with times in the second data set are set to missing. Values in the current data set that are missing are kept as missing. If a value in the second data set is zero or missing, the value in the resultant data set is set to missing (divide by zero not allowed). Either or both data sets may be regular or irregular interval time series.

See also: divide(floating-point constant)
Parameters: `tsData` - A time series data set
Example: `newTsData = tsData.divide(otherTsData)`
Returns: A new **TimeSeriesMath** object

8.15.20 Estimate Values for Missing Precipitation Data

`estimateForMissingPrecipValues``(integer maxMissingAllowed)`

Linearly interpolate estimates for missing values in the current regular or irregular interval time series data set. The current data set is expected to be cumulative precipitation and the data must be of type "INST-CUM". Use the *estimateForMissingValues* method for filling missing values in other types of time series data.

The rules used for interpolation of missing cumulative precipitation data are:

- If the values bracketing the missing period are increasing with time, only interpolate if the number of successive missing values does not exceed the value of *maxMissingAllowed*.
- If the values bracketing the missing period are decreasing with time, do not estimate for any missing values.
- If the values bracketing the missing period are equal, then estimate any number of missing values.

See also: estimateForMissingValues()
Parameters: `maxMissingAllowed` - an integer value for the maximum number of consecutive missing values between valid values
Example:

```
newPrecip =tsPrecip.estimateForMissingPrecipValues(5)
```

Returns: A new **TimeSeriesMath** object

8.15.21 Estimate Values for Missing Data

`estimateForMissingValues`**`(integer maxMissingAllowed)`**

Linearly interpolate estimates for missing values in the current regular or irregular interval time series data set. Do not interpolate if the number of successive missing values exceeds *maxMissingAllowed*.

See also: estimateForMissingPrecipValues()
Parameters: `maxMissingAllowed` - an integer value for the maximum number of consecutive missing values allowed for interpolation
Example: `newTsData = tsData.estimateForMissingValues(5)`
Returns: A new TimeSeriesMath object

8.15.22 Exponent

`exp``()`

Derive a new time series or paired data set which is the e raised to the values of the current time series. For time series data, values that are missing in the current time series remain missing in the new time series. Also, values less than 0.0 will be set to missing the new time series.

For paired data sets, use the setCurve method to first select the paired data curve(s).

See also: setCurve()
Parameters: Takes no parameters
Example: `squaredDataSet = dataSet.exp()`
Returns: A new **HecMath** object of the same type as the current object

8.15.23 Exponentiation Function

`exponentiation``(floating-point constant)`

Derive a new time series or paired data set from the exponentiation of values in the current data set by constant, by:

$$T2\ (i) = T1(i)^{constant}$$

For time series data, values that are missing in the current time series remain missing in the new time series.

For paired data sets, use the *setCurve* method to first select the paired data curve(s).

See also: setCurve()

Parameters: `constant` – a floating-point value representing the exponent.
Example: `squaredDataSet = dataSet.exponentiation(2.)`
Returns: A new **HecMath** object of the same type as the current object

8.15.24 Exponentiation Timeseries Function

`exponentiation``(HecMath tsMath)`

Raise values in the current time series to the power of the parameter time series, **tsMath**. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current time series and **tsMat**h, the value in the current timeseries will be raised to the power of the value in **tsMat**h provided both values are valid (not missing). Points in the current time series which cannot be matched to valid points in **tsMath** are set to missing. Values in the current time series which are missing remain missing in the new time series.The new time series will always have quality defined. If a specific quality value in the parameter time series is questionable or rejected, that quality will be copied to the new time series.

Parameters: tsMath the time series of exponents for the current time series.
Example: `squaredDataSet = dataset.exponentiation`**`(HecMath tsMath)`**
Returns: a new time series resulting from the subtraction operation..
Derive a new time series or paired data set from the exponentiation of values in the current data set by constant, by:

$$T2(i) = T1(i)^{constant}$$

For time series data, values that are missing in the current time series remain missing in the new time series. For paired data sets, use the setCurve method to first select the paired data curve(s).

See also: setCurve()
Parameters: `constant` – a floating-point value representing the exponent.
Example: `squaredDataSet = dataSet.exponentiation(2.)`
Returns: A new **HecMath** object of the same type as the current object.

8.15.25 Extract Time Series Data at Unique Time Specification

`extractTimeSeriesDataForTimeSpecification``(`

```
string timeLevelString,
string rangeString,
boolean isInclusive,
integer intervalWindow,
boolean setAsIrregular )
```

Select/extract data points from the current regular or irregular interval time series data set based upon user defined time specifications. For example, the function may be used to extract from hourly interval data, the values observed every day at noon.

timeLevelString defines the time level/interval for extraction (year, month, day of the month, day of the week, or twenty-four hour time).

rangeString defines the interval range for data extraction applicable to the time level. For example, if *timeLevelString* is "MONTH", a valid range would be "JAN-MAR". The **rangeString** variable can define a single interval value (e.g. "JAN" - select data from January only) or a beginning and ending range (e.g. "JAN-MAR" - select data for January through March). The valid *timeLevelString* and *rangeString* values are shown in Table 8.32.

Table 8.32 Valid timeLevelString and rangeString Values

timeLevelString	rangeString	Example rangeString
"YEAR"	Four-digit year value	"1938" or "1938-1945"
"MONTH"	Standard three-character abbreviation for month	"JAN" or "JAN-MAR" or "OCT-FEB"
"DAYMON(TH)"	Day of the month or "LASTDAY" string	"15" or "1-15 or "27-5" or "16-LASTDAY"
"DAYWEE(K)"	Standard three-character abbreviation for day of the week	"MON" or "SUN-TUE" or "FRI-WED"
"TIME"	Four digit 24-hour military-style clock time	"2400" or "0300-0600" or "2200-0130"

If desired, you may use one of the enumerated string constants to specify *timeLevelString*:

Year	TimeSeriesMath.LEVEL_YEAR_STRING
Month	TimeSeriesMath. LEVEL_MONTH_STRING
Day of Month	TimeSeriesMath. LEVEL_DAYMONTH_STRING
Day of Week	TimeSeriesMath.LEVEL_DAYWEEK_STRING
24-hour time	TimeSeriesMath.LEVEL_TIME_STRING

The parameter *isInclusive* determines whether the data extraction operation is either inclusive or exclusive of the specified range. For example, if *isInclusive* is "True" and the range is set to "JAN-MAR" for the "MONTH" time level, the extracted data will include all data in the months January through March for all the years of time series data. If *isInclusive* is "False" for this example, the extracted data covers the time April through December (is exclusive of the period January through March).

intervalWindow is only used when the *timeLevelString* is "TIME." *intervalWindow* is the minutes before and after the time of day within which the data will be extracted. *intervalWindow* effectively increases the time range at the beginning and end *intervalWindow* minutes. For example, with a *rangeString* of "0300" and an *intervalWindow* of 10, data will be extracted from the selected time series if times falls within in the period 0250 to 0310.

setAsIrregular defines whether the extracted data is saved as regular interval or irregular interval data. Most often the time series data formed by the extraction process will no longer be regular interval, and *setAsIrregular* should be set to "True". Setting *setAsIrregular* to "False" will force an attempt to save the data as regular interval time data.

Parameters:

`timeLevelString` – A string specifying the time level selection.
`rangeString` – A string specifying time or time range for selection. Must be consistent with *timeLevelStrin*g.
`isInclusive` – Either True or False, value. If true, data is extracted inclusive of the range specified by rangeString. If false, data is extracted exclusive of the range specified by *rangeString*.
`intervalWindow` – An integer value representing the minutes before and after the time of day within which the data will be extracted. Only applied when the *timeLevelString* is "TIME".
`setAsIrregular` – Either True orFalse, value. If true, data is automatically set as irregular time interval data. If false, the function will attempt to classify the data as regular time interval data.

Example:

```
SelectedData =
  tsData.extractTimeSeriesDataForTimeSpecification(
  “DAYMONTH”,
  “16-LASTDAY”,
  TRUE,
  0,
  TRUE)
```

Returns: A new **TimeSeriesMath** object
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the function could not successfully interpret *timeLevelString* or *rangeString*.

8.15.26 First Valid Date

```
firstValidData()
```

Find the date and time of the first valid time series value

See also: firstValidValue()
Parameters: Takes no parameters
Example: `squaredDataSet = dataSet.firstValidData()`
Returns: An integer representing the date and time of the first valid value as an integer value translatable by **HecTime.**

8.15.27 First Valid Value

```
firstValidValue()
```

Find the first valid value in the time series

See also: firstValidDate()
Parameters: Takes no parameters
Example: `squaredDataSet = dataSet.firstValidValue ()`
Returns: The floating-point value of the first valid time series value

8.15.28 Floor Function

```
floor()
```

Derive a time series or paired data set with values of the current time series rounded down to the nearest whole number that is less than or equal to the value. For time series data, missing values are kept as missing.

For paired data sets, use the *setCurv*e method to first select the curve(s).

See also: setCurve(), ceil()
Example: `newDataSet = dataSet.floor()`
Parameters: Takes no parameters
Returns: A new **HecMath** object of the same type as the current object

8.15.29 Flow Accumulator Gage (Compute Period Average Flows)

```
flowAccumulatorGageProcessor(TimeSeriesMath tsCounts)
```

Derive a new time series of period-average flows from a flow accumulator type gage. The current time series is assumed to containe the accumulated flow data, while the parameter time series, **tsCounts**, is assumed to have the corresponding time series of counts. The two time series data sets must match times exactly. The two time series are combined to compute a new time series of period average flow:

TsNew(t) = (TsAccFlow(t) - TsAccFlow(t-1)) / (TsCount(t) - TsCount(t-1))

where *TsAccFlow* is the gage accumulated flow time series and *TsCount* is the gage time series of counts.

In the above equation, if TsAccFlow(t), TsAccFlow(t-1), TsCount(t) or TsCount(t-1) are missing, TsNew(t) is set to missing. The new time series is assigned the data type "PER-AVER".

Parameters: `tsCounts` – A **TimeSeriesMath** object containing the counts.
Example:

```
tsPerAvgFlow =
tsAccumFlow.flowAccumulatorGageProcessor(tsCounts)
```

Returns: A new TimeSeriesMath object.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if times in the current object do not exactly match the times in **tsCounts**.

8.15.30 Modulo Function with both Arguments are Greater than Zero

```
fmod(HecMath tsMath)
```

Return the remainder of integer division of current time series by the parameter time series, **tsMath**. A new time series will be created which duplicates the time points of the current time series, where time points match for the current time series and **tsMath**, the value in the current time series will be devided by the value in **tsMath** provided both values are valid (no missing). Devide by zero is not allowed.When the **tsMath** time series value is zero, the value for the new time series will be set to missing. Also, points in the current time series which can not be match to valid points in **tsMath** are set to missing. For time series data, missing values are kept as missing. The new time series will always have quality defined. If a specific quality value in the parameter time series is questionable or rejected, that quality will be copied to the new time series.

For paired data sets, use the *setCurve* method to first select the curve(s).

See also: setCurve(),modulo()

Parameters:
tsMath the time series to be divided into the current time series.
`constant` – a floating-point value representing the exponent.
Example: `squaredDataSet = dataSet.fmod(2.)`
Returns: A new **HecMath** object of the same type as the current object.

8.15.31 Forward Moving Average Smoothing

`forwardMovingAverage``(integer numberToAverageOver)`

Derive a new time series from the forward moving average of *numberToAverageOver* values in the current time series. *numberToAverageOve*r must be an integer greater than two. If the averaging interval contains a missing value, the smoothed value is computed from the remaining valid values in the interval. However, if there are less than two valid values in the interval, the value in the resultant data set is set to missing.

Parameters: `numberToAverageOver` – An integer containing the number of values to average over for computing the forward moving average.
Example: `tsAveraged = tsData.forwardMovingAverage(4)`
Returns: A new **TimeSeriesMath** object.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the *numberToAverageOver* is less than two.

8.15.32 Forward Moving Average Smoothing of Time Series

`forwardMovingAverage``(integer numberToAverageOver, Boolean onlyValidValues, Boolean useReduced)`

Derive a new time series from the forward moving average of the last *numberToAverageOver* values of the current time series. If *onlyValidValues* is set to true, then if points in the averaging interval are missing values, the point in the new time series is set to missing. If *onlyValidValues* is set to false and missing values are contained in the averaging interval, a smoothed point is still computed using the valid values in the interval. If there are no valid values in the averaging interval, the point in the new time series is set to missing.

If *useReduced* is set to true, then forward moving average points can be still be computed at the beginning of the time series even if there are less than *numberToAverageOver* values in the interval. If *useReduced* is set to false, then the first *numberToAverageOver* points of the new time series are set to missing.

Parameters:

`numberToAverageOver` – An integer containing the number of values to average over for computing the forward moving average.

`onlyValidValues` – Either True or False, specifying whether all values in the averaging interval must be valid for the computed point in the new time series to be valid.

`useReduced` – Either True or False, specifying whether to allow points at the beginning and end of the resultant time series to be computed from a reduced (less than *numberToAverageOver*) set of points.

Example: `tsAveraged = tsData.forwardMovingAverage (4,TRUE,TRUE)`

Returns: A new time series computed from the forward moving average of current time series.

Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the *numberToAverageOver* is less than two.

8.15.33 Generate Data Pairs from Two Time Series

`generateDataPairs``(TimeSeriesMath tsData, Boolean sort)`

Generate a paired data set by pairing values (by time) from the current time series data set and the time series data set **tsDat**a. The values of the current time series form the x-ordinates, while values from **tsData** form the y-ordinates of the resulting paired data set. The times in the two time series data sets must match exactly. If a value for a time is missing in either time series, no data value pair is formed or added to the paired data set. If sort is "True", data pairs in the paired data set are sorted by ascending x-value.

The units and parameter type from the current time series data set are assigned to the paired data set x-units and x-parameter type. The units and parameter type from tsData are assigned to the paired data set y-units and y-parameter type.

An example application of the function would be to mate a time series record of stage to one of flow to generate a stage-flow paired data set.

Parameters:

`tsData` – A **TimeSeriesMath** object that forms the y-ordinates of the resulting paired data set.

`sort` – Either True or False, value. If true, sort data pairs in ascending x-value. If false, leave unsorted.

Example: `ratingCurve = tsStage.generateDataPairs(tsFlow)`

Returns: A **PairedDataMath** object with x-ordinates from the current time series, and y-ordinates from **tsData**.

Generated Exceptions: Throws a *hec.hecmath.HecMathException* if times from the current time series and **tsData** do not match exactly.

8.15.34 Generate a Regular Interval Time Series

```
generateRegularIntervalTimeSeries(string startTimeString,
  string endTimeString,
  string timeIntervalString,
  string timeOffsetString,
  floating-point initialValue)
```

Generate a new regular interval time series data set from scratch with times and values specified by the parameters. This is a function provided by the **TimeSeriesMath** module, and not an object method.

The parameters *startTimeString* and *endTimeString* are strings used to specify the beginning and ending time of the generated data set. These two parameters have the form of the standard HEC time string (e.g. "01JAN2001 0100").

The regular time interval is specified by *timeIntervalString*, and is a valid HEC time increment string (e.g. "1MIN", "15MIN", "1HOUR", "6HOUR", "1DAY", "1MONTH").

timeOffsetString is used to shift times in the resultant time series from the standard interval time. As an example, the offset could be used to shift times in regular hourly interval data from the top of the hour to six minutes past the hour. The parameter has the form "nT", where "n" is an integer number and "T" is one of the time increments: "M(INUTES)", "D(AYS)", "H(OUR)", "W(EEKS)", "MON(THS)" or "Y(EARS)" (characters in the parenthesis are optional). For example, a time offset of nine minutes would be expressed as "9M" or "9MIN".

Values in the time series data set are initialized to *initialValue*.

Parameters:

`startTimeString` - a string specifying a standard HEC time defining the time series data start date/time.

`endTimeString` - a string specifying a standard HEC time defining the time series data end date/time.

`timeIntervalString` - a string specifying a valid DSS regular time interval which defines the time interval of the new time series.

`timeOffsetString` – a string specifying the offset of the new time points from the regular interval time. This string may be an empty string or None.

`initialValue` - a floating-point number set to the initial value for all time series points. Set to *HecMath.UNDEFINED* to set all values to missing.

Example:

```
newTsData=TimeSeriesMath.generateRegularIntervalTimeSeries(
  “01FEB2002 0100”,
  “28FEB2002 2400”,
  “1HOUR”,
  “0M”,
  100.)
```

Returns: A new regular interval **TimeSeriesMat**h object initialized to **initialValue**. Data units and type are unset.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if time parameters cannot be successfully interpreted.

8.15.35 Get Data Container

`getData``()`

Returns a copy of the *hec.io.DataContainer* for the current data set. For time series data sets, returns a *hec.io.TimeSeriesContainer*. For paired data sets, returns a *hec.io.PairedDataContainer*.

The *hec.io.TimeSeriesContainer* contains the time series values for a time series data set. The *hec.io.PairedDataContainer* contains the paired data values for a paired data set.

Parameters: Takes no parameters
Example: `container = dataset.getData()`
Returns: A *hec.io.DataContainer*

8.15.36 Get Data Type for Time Series Data Set

`getType``()`

Get the data type for a time series data set.

Parameters: Takes no parameters
Example: dataSet.getType()
Returns: A string - "INST-CUM", "INST-VAL", "PER-AVER" or "PER-CUM".

8.15.37 Get Units Label for Data Set

`getUnits``()`

Get the units label of the current data set. For a paired data set, returns the y-units label.

Parameters: Takes no parameters
Example: `dataSet.getUnits()`
Returns: A string

8.15.38 Gmean

```
gmean(list of HecMath tsMathArray)
```

Determine the geometric mean of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the geometric mean of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quality values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: tsMathArray the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet. gmean(list of HecMath tsMathArray)`
Returns: a new time series representing the geometric mean of all time series.

8.15.39 Hmean

```
hmean(list of HecMath tsMathArray)
```

Determine the harmonic mean of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the harmonic mean of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to

missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quality values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: t*sMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet. hmean(list of HecMath tsMathArray)`
Returns: a new time series representing the harmonic mean of all time series.

8.15.40 Integer Division by a Constant

`integerDivide``(floating-point constant)`

Divide values in the current time series by a constant and truncate the result to an integer value. Times in which values are missing in the current time series remain missing in the new time series.

Parameters: constant the value to divide values in current time series.
Example: `dataSet.integerDivide(2.0)`
Returns: a new time series resulting from the integer division operation.

8.15.41 Integer Divison by an Object

`integerDivide``(HecMath tsMath)`

Divide the current time series by the parameter time series, **tsMath** and truncate the result to an integer value. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current time series and **tsMath**, the value in the current time series will be divided by the value in **tsMath** provided both values are valid (not missing). Divide by zeroes are not allowed. When the **tsMath** time series value is zero, the value for the new time series will be set to missing. Points in the current time series which cannot be matched to valid points in **tsMath** are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. If a specific quality value in the parameter time series is questionable or rejected, that quality will be copied to the new time series.

Parameters: tsMath the time series to be divided into the current time series.
Example: `newDataSet = dataset.integerDivide(HecMath tsMath)`
Returns: a new time series resulting from the integer division operation.

8.15.42 Interpolate Time Series Data at Regular Intervals

```
interpolateDataAtRegularInterval(
  string timeIntervalString,string timeOffsetString)
```

Derive a regular interval time series data set by interpolation of the current regular or irregular interval time series data set.

The new time interval is set by *timeIntervalString* which must be a valid HEC time interval string (e.g. "1MIN", "15MIN", "1HOUR", "6HOUR", "1DAY", "1MONTH").

Times in the resultant time series may be shifted (offset) from the regular interval time by the increment specified by *timeOffsetString*. As an example, the offset could be used to shift times from the top of the hour to six minutes past the hour. If no offset is used *timeOffsetString* should be a blank or empty string.

Whether the time series data type is "INST-VAL", "INST-CUM", "PER-AVE", or "PER-CUM" controls how the interpolation is performed. Interpolated values are derived from "INST-VAL" or "INST-CUM" data using linear interpolation. Values are derived from "PER-AVE" data by computing the period average value over the time interval. Values are derived from "PER-CUM" data by computing the period cumulative value over the new time interval.

For example, if the original data set is hourly data and the new regular interval data set is to have a six hour time interval:

The value for "INST-VAL" or "INST-CUM" type data is computed from the linear interpolation of the hourly points bracketing the new six hour time point.

The value for "PER-AVE" type data is computed from the period average value over the six hour interval.

The value for "PER-CUM" type data is computed from the accumulated value over the six hour interval.

The treatment of missing value data is also dependent upon data type. Interpolated "INST-VAL" or "INST-CUM" points must be bracketed or coincident with valid (not missing) values in the original time series; otherwise the interpolated values are set as missing. Interpolated "PER-AVE" or "PER-CUM" data must contain all valid values over the interpolation interval; otherwise the interpolated value is set as missing.

Parameters:

`timeIntervalString` – A string specifying the regular time interval for the resultant time series.

`timeOffsetString` – A string specifying the offset of the new time points from the regular interval time. This variable may be an empty string (" ").

Example:

```
newTsData =
tsData.interpolateDataAtRegularInterval(
  “15MIN”,
  “  ”)
```

Returns: A new regular interval **TimeSeriesMath** object.

8.15.43 Inverse (1/X) Function

`inverse``()`

Derive a new time series or paired data set from the inverse (1/x) of values of the current data set. The inverse value is computed by 1.0 divided by the value of the current data set. If a data value is equal to 0.0, the value in the resultant data set is set to missing. For time series data, if the original value is missing, the value remains missing in the resultant data set.

For paired data sets, use the *setCurve* method to first select the paired data curve(s).

See also: setCurve()
Parameters: Takes no parameters
Example: `newDataSet = dataSet.inverse()`
Returns: A **HecMath** object of the same type as the current object.

8.15.44 Determine if Data is in English Units

`isEnglish``()`

Determine if the current time series or paired data set is in English units. The function examines the data set parameter type and units label to establish the unit system.

See also: isMetric(); convertToEnglishUnits()
Parameters: No parameters
Example: `if dataSet.isEnglish() : print "English Units"`
Returns: True if the data set units are English, otherwise False.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the unit system cannot be determined (parameter type and units label undefined).

8.15.45 Determine if Data is in Metric Units

```
isMetric()
```

Determine if the current time series or paired data set is in Metric (SI) units. The function examines the data set parameter type and units label to establish the unit system.

See also: isEnglish(); convertToMetricUnits()
Parameters: Takes no parameters
Example: `if dataSet.isMetric() : print "SI Units"`
Returns: True if the data set units are Metric, otherwise False.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the unit system cannot be determined (parameter type and units label undefined).

8.15.46 Determine if Computation Stable for Given Muskingum Routing Parameters

```
isMuskingumRoutingStable(integer numberSubreaches,
  floating-point muskingumK,
  floating-point muskingumX)
```

Check for possible instability for the given Muskingum Routing parameters.
Test if the input parameters satisfy the stability criteria:

$$1/(2(1-x)) <= K/deltaT <= 1/2x$$

where deltaT = (time series time interval)/numberSubreaches

Parameters:

`numberSubreaches` – integer specifying the number of routing subreaches.

`muskingumK` –floating-point number specifying the Muskingum "K" parameter, in hours.
`muskingumX` - floating-point number specifying the Muskingum "x" parameter, between 0.0 and 0.5 (inclusive).

Example:

```
warning = tsDataSet.isMuskingumRoutingStable(
reachCount,
kVal,
xVal)
  if warning :
  print warning
  return
```

Returns: A string if the stability criteria is not met. The string contains a warning message detailing the specific instability problem. Otherwise returns None.

Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the current time series is not a regular interval time series, or if values for *numberSubreaches* or *muskingumX* are invalid.

8.15.47 Last Valid Value's Date and Time

`lastValidDate``()`

Find and return the date and time of the last valid (non-missing) value in a time series data set.

Parameters: Takes no parameters
Example: `tsData.lastValidDate()`
Returns: An integer value translatable by **HecTime** representing the date and time of the last valid time series value.

8.15.48 Last Valid Value in a Time Series

`lastValidValue``()`

Find and return the last valid (non-missing) value in a time series data set.

Parameters: Takes no parameters
Example: `tsData.lastValidValue()`
Returns: A floating-point value representing the last valid time series value.

8.15.49 Natural Log, Base "e" Function

`log``()`

Derive a new time series or paired data set from the natural log (log base "e") of values of the current data set. Missing values in the original data set remain missing. Values less than or equal to 0.0 will be set to missing.

For paired data sets, use the *setCurve* method to first select the paired data curve(s).

See also: log10(), setCurve()
Parameters: Takes no parameters
Example: `newDataSet = dataSet.log()`
Returns: A new **HecMath** object of the same type as the current object.

8.15.50 Log Base 10 Function

`log10``()`

Derive a new time series or paired data set from the log base 10 of values of the current data set. Missing values in the original data set remain missing. Values less than or equal to 0.0 will be set to missing.

For paired data sets, use the *setCurve* method to first select the paired data curve(s).

See also: `log(), setCurve()`
Parameters: Takes no parameters
Example: `newDataSet = dataSet.log10()`
Returns: A new **HecMath** object of the same type as the current object.

8.15.51 Maximum Value in a Time Series

`max``()`

Find and return the maximum value of the current time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `maxVal = tsData.max()`
Returns: A floating-point value representing the maximum value of the current time series.

8.15.52 Maximum Value in a Time Series (*tsMathArrary*)

```
max(list of HecMath tsMathArray)
```

Determine the maximum of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the maximum of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quality values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.max(list of HecMath tsMathArray)`
Returns: a new time series representing the maximum values of all time series.

8.15.53 Maximum Value's Date and Time

```
maxDate()
```

Find and return the date and time of the maximum value for the current time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `maxDateTime = tsData.maxDate()`
Returns: An integer value translatable by **HecTime** representing the date and time of the maximum time series value.

8.15.54 Mean Time Series Value

```
mean()
```

Compute the mean value of the current time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `meanVal = tsData.mean()`
Returns: A floating-point value representing the mean value of the current time series.

8.15.55 Mean Time Series Value (*tsMathArray*)

```
mean(list of HecMath tsMathArray)
```

Determine the arithmetic mean of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the arithmetic mean of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.mean(list of HecMath tsMathArray)`
Returns: a new time series representing the arithmetic mean of all time series.

8.15.56 Median Time Series Value

```
med(list of HecMath tsMathArray)
```

Determine the median (50th percentile) of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the median of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to

missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.med(list of HecMath tsMathArray)`
Returns: a new time series representing the median of all time series.

8.15.57 Merge Paired Data Sets

```
mergePairedData(PairedDataMath pdData)
```

Merge the current paired data set with the paired data set *pdDat*a. The resultant paired data set includes all the paired data curves from the current data set. Depending upon a previous use of the *setCurveMethod* on *pdData*, a single selected paired data curve or all curves from *pdData* are appended to the merged data set. The x-values for the two paired data sets must match exactly.

See also: setCurve()
Parameters: `pdData` - a paired data set with x-ordinates matching those of the current data set.
Example: `mergedCurve = curve.mergePairedData(anotherCurve)`
Returns: A *new PairedDataMath* object.

8.15.58 Merge Two Time Series Data Sets

```
mergeTimeSeries(TimeSeriesMath tsData)
```

Merge data from the current time series data set with the time series data set **tsDat**a. The resultant time series data set includes all the data points in the two time series, except where the data points occur at the same time. When data points from the two data sets are coincident in time, valid values in the current time series take precedence over valid values from **tsData**. However, if a coincident point is set to missing in the current time series data set, a valid value from **tsData** will be used for time in the resultant data set. If the values are missing for both data sets, the value is missing in the resultant data set.

The data sets for merging may have either regular or irregular time interval time series data. The data sets are tested to determine if they both have the same regular time interval. If not, the resultant data set is typed as an irregular interval data set.

Parameters: `tsData` - a time series data set for merging with the current time series data set.
Example: `tsMerged = tsData.mergeTimeSeries(otherTsData)`
Returns: A new **TimeSeriesMath** object.

8.15.59 Minimum Value in a Time Series

```
min()
```

Find and return the minimum value of the current a time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `minVal = tsData.min()`
Returns: A floating-point value representing the minimum value of the current time series.

8.15.60 Minimum Value in a Time Series (*tsMathArray*)

```
min(list of HecMath tsMathArray)
```

Determine the minimum of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the minimum of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArra*y the array of time series to be compared with the current time series.

Example: `newDataSet = dataSet.min(list of HecMath tsMathArray)`
Returns: a new time series representing the minimum values of all time series.

8.15.61 Minimum Value's Date and Time

```
minDate()
```

Find and return the date and time of the minimum value for the current time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `minDateTime = tsData.minDate()`
Returns: An integer value translatable by HecTime representing the date and time of the minimum time series value.

8.15.62 Modified Puls or Working R&D Routing Function

```
modifiedPulsRouting(TimeSeriesMath tsFlow,
  integer numberSubreaches,
  floating-point muskingumX)
```

The current data set is a paired data set containing the storage-discharge table for Puls routing, where the x-values are storage and the y-values are discharge. The function derives a new time series data set from the Modified Puls or Working R&D routing of the time series data set **tsFlow**. *numberSubreaches* is the number of routing subreaches.

The Working R&D method provides a means of including the effects of inflow on reach storage by use of the Muskingum "x" wedge coefficient. The Working R&D method is activated in the computation if muskingumX is greater than 0.0. However, muskingumX cannot be greater that 0.5.

Parameters:
`tsFlow` – A regular interval time series data set for routing.
`numberSubreaches` – Number of routing subreaches.
`muskingumX` - Muskingum "X" parameter, between 0.0 and 0.5 (inclusive). Enter 0.0 to route by the Modified Puls method, or a value greater than 0.0 to apply the Working R&D.
Example:

```
routedFlow =
storDichareCurve.modifiedPulsRouting(
  tsFlow,
  reachCount,
  coefficient)
```

Returns: A new **TimeSeriesMath** object.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the **tsMath** is not a regular interval time series; if *muskingumX* is less than 0.0 or greater than 0.5; if the current paired data set does not have both ascending x and y values.

8.15.63 Modulo

```
modulo(floating-point constant)
```

Return the remainder of integer division of current time series by a constant.Times in which values are missing in the current time series remain missing in the new time series.

Parameters: constant the value to divide values in current time series.
Example: `newDataSet = dataSet.modulo(floating-point constant)`
Returns: a new time series resulting from the operation.

8.15.64 Modulo (tsMath)

```
modulo(HecMath tsMath)
```

Return the remainder of integer division of current time series by the parameter time series, **tsMat**h. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current time series and **tsMath**, the value in the current time series will be divided by the value in **tsMath** provided both values are valid (not missing). Divide by zeroes are not allowed. When the **tsMath** time series value is zero, the value for the new time series will be set to missing. Points in the current time series which cannot be matched to valid points in **tsMath** are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. If a specific quality value in the parameter time series is questionable or rejected, that quality will be copied to the new time series.

Parameters: **tsMath** the time series to be divided into the current time series.
Example: `newDataSet = dataSet.modulo(HecMath tsMath)`
Returns: a new time series resulting from the operation.

8.15.65 Multiple Linear Regression Coefficients

```
multipleLinearRegression( sequence tsDataSequence,
  floating-point minimumLimit,
  floating-point maximumLimit)
```

Compute the multiple linear regression coefficients between the current time series data set and the array of independent time series data sets in *tsDataSequence*. The function stores the regression coefficients in a new paired data set. This paired data set may be used with the *multipleLinearRegressio*n function to derive a new estimated time series data set.

For the general linear regression equation, a dependent variable, Y, may be computed from a set independent variables, Xn:

$$Y = B0 + B1*X1 + B2*X2 + B3*X3$$

where Bn are linear regression coefficients.

For time series data sets, an estimate of the original time series data set values may be computed from a set of independent time series data sets using regression coefficients such that:

$$TsEstimate(t) = B0 + B1*TS1(t) + B2*TS2(t) + \ldots + Bn*TSn(t)$$

where Bn are the set of regression coefficients and TSn are the time series data sets contained in *tsDataSequence*.

The parameters *minimumLimit* and *maximumLimit* may be used to exclude out of range values in the current time series data set from the regression determination. *minimumLimit* or *maximumLimit* may be entered as "Constants.UNDEFINED" to ignore the minimum or maximum value check.

See also: applyMultipleLinearRegression()

Parameters:

`tsDataSequence` – sequence of TimeSeriesMath objects, which form the independent variables in the regression equation. Must all be regular interval and have the same time interval.

`minimumLimit` – A floating-point value. Values in the current time series exceeding minimumLimit are excluded from the regression analysis. Set to Constants.UNDEFINED to ignore this option.

`maximumLimit` – A floating-point value. Values in the current time series exceeding maximumLimit are excluded from the regression analysis. Set to Constants.UNDEFINED to ignore this option.

Example:

```
regression = tsFlow.multipleLinearRegression (
  [tsUpstrFlow1, tsUpstrFlow2, tsUpstrFlow3],
  0.,
  100000.)
```

Returns: A **new PairedDataMath** object containing the computed regression coefficients.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the current data set and the data sets in *tsDataSequence* are not regular interval time series data sets with the same interval time.

8.15.66 Multiply by a Constant

multiply(floating-point constant)

Multiply the value constant to all valid values in the current time series or paired data set. For time series data, missing values are kept as missing. For paired data, constant multiplies the y-values only. Use the *setCurveMethod* to first select the paired data curve(s).

See also: multiply(TimeSeriesMath tsData); setCurve()
Parameters: `constant` - A floating-point precision value.
Example: `newDataSet = dataSet.multiply(1.5)`
Returns: A new **HecMath** object of the same type as the current object.

8.15.67 Multiply by a Data Set

multiply(TimeSeriesMath tsData)

Multiply valid values in the current data set by the corresponding values in the data set **tsData**. Both data sets must be time series data set.

When multiplying one time series data set to another, there is no restriction that times in the two data sets match exactly. However, only values with coincident times will be multiplied. Times in the current time series data set that cannot be matched with times in the second data set are set to missing. Values in the current data set that are missing are kept as missing. Either or both data sets may be regular or irregular interval time series.

See also: multiply(floating-point constant)
Parameters: `tsData` - A time series data set.
Example: `newTsData = tsData.multiply(otherTsData)`
Returns: A new **TimeSeriesMath** object.

8.15.68 Muskingum Hydrologic Routing Function

```
muskingumRouting(integer numberSubreaches,
  floating-point muskingumK,
  floating-point muskingumX)
```

Route the current regular interval time series data set by the Muskingum Routing method. The current data set must be a regular interval time series data set. *muskingumK* is the Muskingum "K" parameter, in hours, and *muskingumX* is the Muskingum "x" parameter. *muskingumX* cannot be less than 0.0 or greater than 0.5.

The set of Muskingum routing parameters may potentially produce numerical instabilities in the routed time series. Use the function *isMuskingumRoutingStable()* to test if the Muskingum routing parameters may potentially have instabilities.

See also: isMuskingumRoutingStable()
Parameters:
`numberSubreaches` – An integer specifying the number of routing subreaches.
`muskingumK` – A floating-point number specifying the Muskingum "K" parameter in hours.
`muskingumX` – A floating-point number specifying the Muskingum "x" parameter, between 0.0 and 0.5
Example:

```
routedFlows = tsFlows.muskingumRouting(reachCount, K, x)
```

Returns: A new TimeSeriesMath object.
Generated Exceptions: Throws a *hec.hecmath.HecMathException* if the current time series is not a regular interval time series; if *muskingumX* is less than 0.0 or greater than 0.5.

8.15.69 Negation Function

```
negative()
```

Derive a new time series composed of the negatives of the values of the current time series.Values which are missing in the original time series will be missing in the new time series.

Parameters: Takes no parameters
Example: `newDataSet = dataSet.neg()`
Returns: A new time series composed of the negatives of the values of the current time series.

8.15.70 Number of Invalid Values in a Time Series

numberInvalidValues()

Count and return the number of invalid values in the current time series data set.

Parameters: Takes no parameters
Example: `invalidCount = tsData.numberInvalidValues()`
Returns: An integer of the count of invalid (non-missing) time series values.

8.15.71 Number of Missing Values in a Time Series

numberMissingValues()

Count and return the number of missing values in the current time series data set.

Parameters: Takes no parameters
Example: `missingCount = tsData.numberMissingValues()`
Returns: An integer of the count of missing time series values.

8.15.72 Number of Questioned Values in a Time Series

numberQuestionedValues()

Count and return the number of questioned values in the current time series data set.

Parameters: Takes no parameters
Example: `quiestionedCount = tsData.numberQuestionedValues()`
Returns: An integer of the count of questioned (non-missing) time series values.

8.15.73 Number of Rejected Values in a Time Series

`numberRejectedValues()`

Count and return the number of rejected values in the current time series data set.

Parameters: Takes no parameters
Example: `rejectedCount = tsData.numberRejectedValues()`

Returns: An integer of the count of rejected (non-missing) time series values.

8.15.74 Number of Valid Values in a Time Series

```
numberValidValues()
```

Count and return the number of valid values in the current time series data set.

Parameters: Takes no parameters
Example: `validCount = tsData.numberValidValues()`
Returns: An integer of the count of valid (non-missing) time series values.

8.15.75 Olympic Smoothing

```
olympicSmoothing(integer numberToAverageOver,
  boolean onlyValidValues,
  boolean useReduced)
```

Derive a new time series from the Olympic smoothing of *numberToAverageOver* values in the current time series. *numberToAverageOver* must be and odd integer and greater than. Similar to centered moving average smoothing, except that the minimum and maximum values over the averaging interval are excluded from the computation.

If *onlyValidValues* is set to true, then if any values in the averaging interval are missing, the point in the resultant time series is set to missing. If *onlyValidValues* is set to false and there are missing values in the averaging interval, a smoothed point is still computed using the remaining valid values in the interval. If there are no valid values in the averaging interval, the point in the resultant time series is set to missing.

If *useReduced* is set to true, then moving average values can be still be computed at the beginning and end of the time series even if there are less than *numberToAverageOver* values in the interval. If *useReduced* is set to false, then the first and last *numberToAverageOver*/2 points of the resultant time series are set to missing.

Parameters:

`numberToAverageOver` – An integer specifying the number of values to average over for computing the smoothed time series. Must be an odd integer greater than two.

`onlyValidValues` – Either True or False, specifying whether all values in the averaging interval must be valid for the computed point in the resultant time series to be valid.
`useReduced` - Either True or False, specifying whether to allow points at the beginning and end of the smoothed time series to be computed from a reduced (less than *numberToAverageOver*) number of values. Otherwise, set the first and last *numberToAverageOver*/2 points of the new time series to missing.

Example: `avgData = tsData.olympicSmoothing(5, )`
Returns: A new TimeSeriesMath object.
Generated Exceptions: Throws a **HecMathException** if the *numberToAverageOver* is less than three or not odd.

8.15.76 P1 Function

p1(list of HecMath tsMathArray)

Determine the first percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 1st percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p1(list of HecMath tsMathArray)`
Returns: a new time series representing the first percentile of all time series.

8.15.77 P2 Function

p2(list of HecMath tsMathArray)

Determine the 2nd percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 2nd percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Examples: `newDataSet = dataSet.p2(list of HecMath tsMathArray)`
Returns: a new time series representing the 2nd percentile of all time series.

8.15.78 P5 Function

```
p5(list of HecMath tsMathArray)
```

Determine the fifth percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the fifth percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.

Example: `newDataSet = dataSet.p5(list of HecMath tsMathArray)`
Returns: a new time series representing the fifth percentile of all time series.

8.15.79 P10 Function

p10(list of HecMath tsMathArray)

Determine the tenth percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the tenth percentile of all time series for that time provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quality values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p10(list of HecMath tsMathArray)`
Returns: a new time series representing the tenth percentile of all time series.

8.15.80 P20 Function

p20(list of HecMath tsMathArray)

Determine the 20th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 20th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p20(list of HecMath tsMathArray)`
Returns: a new time series representing the 20th percentile of all time series.

8.15.81 P25 Function

p25(list of HecMath tsMathArray)

Determine the 25th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 25th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p25(list of HecMath tsMathArray)`
Returns: a new time series representing the 25th percentile of all time series.

8.15.82 P75 Function

p75(list of HecMath tsMathArray)

Determine the 75th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created

which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 75th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArra*y are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p75(list of HecMath tsMathArray)`
Returns: a new time series representing the 75th percentile of all time series.

8.15.83 P80 Function

```
p80(list of HecMath tsMathArray)
```

Determine the 80th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 80th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series. The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: tsMathArray the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p80(list of HecMath tsMathArray)`
Returns: a new time series representing the 80th percentile of all time series.

8.15.84 P89 Function

```
p89(list of HecMath tsMathArray)
```

Determine the 89th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 89th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series. The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p89(list of HecMath tsMathArray)`
Returns: a new time series representing the 89th percentile of all time series.

8.15.85 P90 Function

```
p90(list of HecMath tsMathArray)
```

Determine the 90th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 90th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series. The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.

Example: `newDataSet = dataSet.p90(list of HecMath tsMathArray)`
Returns: a new time series representing the 90th percentile of all time series.

8.15.86 P95 Function

`p95(list of HecMath tsMathArray)`

Determine the 95th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 95th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series. The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p95(list of HecMath tsMathArray)`
Returns: a new time series representing the 95th percentile of all time series.

8.15.87 P99 Function

`p99(list of HecMath tsMathArray)`

Determine the 99th percentile of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the 99th percentile of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series. The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe

quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataSet.p99(list of HecMath tsMathArray)`
Returns: a new time series representing the 99th percentile of all time series.

8.15.88 Period Constants Generation

`periodConstants(TimeSeriesMath tsData)`

Derive a new time series data set by applying values in the current time series data set to the times defined by the time series data set **tsData**. Both time series data sets may be regular or irregular interval. Values in a new time series are set according to:

$$ts1(j) \leq tsnew(i) < ts1(j+1), \quad TSNEW(i) = TS1(j)$$

where ts1 is the time in the current time series, TS1 is the value in the current time series, tsnew is the time in the new time series, TSNEW is the value in the new time series.

If times in the new time series precede the first data point in the current time series, the value for these times is set to missing. If times in the new time series occur after the last data point in the current time series, the value for these times is set to the value of the last point in the current time series. The interpolation of values with the *periodConstants* function is shown in Figure 8.13.

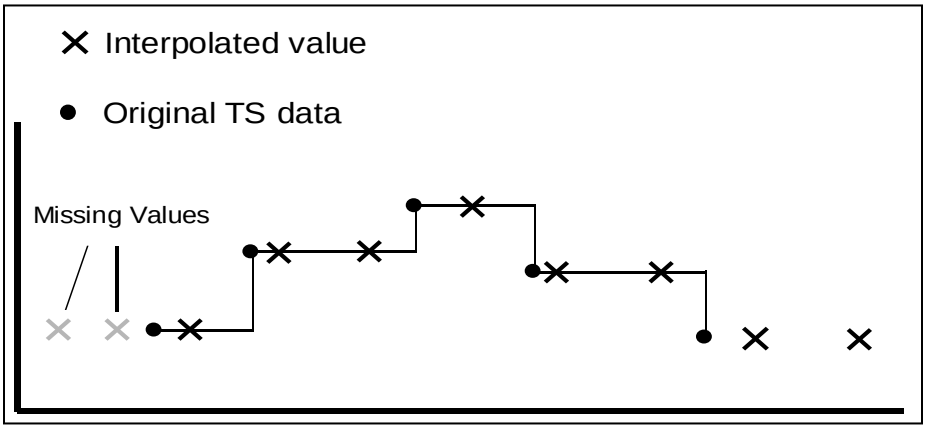

Figure 8.13 Interpolation of Time Series Values Using Period Constants Function

Parameters: tsData - a regular or irregular interval time series data set.

Example: `tsConstants = tsValues.periodConstants(tsData)`
Returns: A new **TimeSeriesMath** object.

8.15.89 Polynomial Transformation

`polynomialTransformation``(TimeSeriesMath tsData)`

Compute a polynomial transformation of a regular or irregular interval time series data set, **tsData**, using the polynomial coefficients stored in the current paired data set. Missing values in **tsData** remain missing in the resultant data set.

A new time series can be computed from an existing time series with the polynomial expression:

$$TS2(t) = B1 * TS1(t) + B2 * TS1(t)^{2} + ... + Bn * TS1(t)^{n}$$

where Bn are the polynomial coefficients for term "n."

Values for the polynomial coefficients are stored in the x-values of the current paired data set. Before the above equation is applied, values in the input time series are adjusted by subtracting off the paired data "datum" value if defined. The units label and parameter type for the resultant time series are copied from the current paired data set x-units and parameter type.

See also: polynomialTransformationWithIntegral()
Parameters: **tsData** - a regular or irregular interval time series data set.
Example: `tsXform = pdCoef.polynomialTransformation(tsData)`
Returns: A new **TimeSeriesMath** object.

8.15.90 Polynomial Transformation with Integral

`polynomialTransformationWithIntegral`**`(TimeSeriesMath tsData)`**

Compute a polynomial transformation with integral of a regular or irregular interval time series data set, **tsData**, using the polynomial coefficients stored in the current paired data set. Missing values in **tsData** remain missing in the resultant data set.

This function is similar to the polynomialTranformation method, and the same set of polynomial coefficients is used. The equation for the polynomial transform is modified so that the transform of **tsData** is computed from the integral of the polynomial coefficients:

$$TS2(t) = B1 * TS1(t)^{2}/2 + B2 * TS1(t)^{3}/3 + \ldots + Bn * TS1(t)^{n+1}/(n+1)$$

where Bn are the polynomial coefficients for term "n".

Values for the polynomial coefficients are stored in the x-values of the current paired data set. Before the above equation is applied, values in the input time series are adjusted by subtracting off the paired data "datum" value if defined. The units label and parameter type for the resultant time series are copied from the current paired data set x-units and parameter type.

See also: polynomialTransformation()
Parameters: **tsData** - a regular or irregular interval time series data set.
Example: `tsXform = pdCoef.polynomialTransformationWithIntegral(tsData)`
Returns: A new **TimeSeriesMath** object.

8.15.91 Product Function

`product``(list of HecMath tsMathArray)`

Multiply the current time series with the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be multiplied provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series. The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be multiplied with the current time series.
Example: `newDataSet = dataset.product(list of HecMath tsMathArray)`
Returns: a new time series representing the multiplication of the two time series

8.15.92 Rating Table Interpolation

`ratingTableInterpolation``(TimeSeriesMath tsData)`

Transform/interpolate values in the time series data set **tsData** using the rating table x-y values stored in the current paired data set. For example, you can use the function to transform a time series of stage to a time series of flow using a stage-flow rating table. **tsData** may be a regular or irregular time interval data set. Missing values in **tsData** are kept missing in the resultant data set.

Create the paired data set with the rating table option to set values for "datum", "shift", and "offset". By default these values are 0.0. The shift is added to and the datum subtracted from all input time series values. If the rating table is Log-Log, the table x-values are adjusted by subtracting the offset.

Units and parameter type in resultant time series data set are defined by the y-units label and parameter type of the current paired data set. All other names and labels are copied over from **tsData**.

See also: reverseRatingTableInterpolation()
Parameters: **tsData** - a regular or irregular interval **TimeSeriesMath** object.
Example:
```
tsFlow =
      stageFlowCurve.ratingTableInterpolation(tsStage)
```
Returns: A new **TimeSeriesMath** object.

8.15.93 Replace Specific Values

```
replaceSpecificValues(HecDouble from, HecDouble to)
```

Replace specific values for another in the time series.

The function searches through the time series array and replaces values that match the "from" value with the "to" value. **HecDouble** controls the precision to which the value is compared (e.g., 12.345 is to three places after the decimal).

Parameters: from value which will be replaced to value with which to replace with
Examples:
```
newDataSet = dataset.replaceSpecificValues
(HecDouble from, HecDouble to)
```
Returns: a copy of the current time series with the specific values replaced.

8.15.94 Reverse Rating Table Interpolation

```
reverseRatingTableInterpolation(TimeSeriesMath tsData)
```

Transform/interpolate values in the time series data set **tsData** using the reverse of the rating table stored in the current paired data set. For example, the function may be used to transform a time series of flow to a time series of stage using a stage-flow rating table. **tsData** may be a regular or irregular time interval data set. Missing values in **tsData** are kept missing in the resultant data set.

The paired data set should be created with the rating table option to set values for "datum", "shift", and "offset". By default, these values are 0.0. The shift is subtracted from, and the datum added to all input time series values. If the rating table is Log-Log, the table x-values are adjusted by subtracting the offset. Refer to the *ratingTableInterpolation()* description for comparison to this function.

Units and parameter type in resultant time series data set are defined by the x-units label and parameter type of the current paired data set. All other names and labels are copied over from the **tsData**.

See also: ratingTableInterpolation
Parameters: tsData - a regular or irregular interval **TimeSeriesMath** object.
Example:
```
tsStage = stageFlowCurve.reverseRatingTable
Interpolation(tsFlow)
```
Returns: A new **TimeSeriesMath** object.

8.15.95 RMS Function

```
rms(list of HecMath tsMathArray)
```

Determine the root mean square (rms, or quadraticharmonic mean) of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the rms of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* the array of time series to be compared with the current time series.
Example: `newDataSet = dataset.rms(list of HecMath tsMathArray)`
Returns: a new time series representing the rms of all time series.

8.15.96 Round to Nearest Whole Number

`round``()`

Rounds values in a time series or paired data set to the nearest whole number. The function rounds up the decimal portion of a number if equal to or greater than .5 and rounds down decimal values less than .5. For example:

10.5 is rounded to 11.
10.499 is rounded to 10.

The x-values in paired data sets are unaffected by the function, only the y-value data are rounded. For time series data sets, missing values are kept missing.

For paired data sets, use the *setCurve()* method to first select the paired data curve(s).

See also: roundOff(); truncate(); setCurve().
Parameters: Takes no parameters
Example: `roundedData = dataSet.round()`
Returns: A new **HecMath** object of the same type as the current object.

8.15.97 Round Off to Specified Precision

`roundOff``(integer significantDigits, integer powerOfTensPlace)`

Round values in a time series or paired data set to a specified number of significant digits and/or power of tens place. For the power of tens place, -1 specifies rounding to one-tenth (0.1), while +2 rounds to the hundreds (100). For example:

1234.123456 will round to:
1230.0 for number of significant digits = 3, power of tens place = -1
1234.1 for number of significant digits = 6, power of tens place = -1
1234 for number of significant digits = 6, power of tens place = 0
1230 for number of significant digits = 6, power of tens place = 1

The x-values in paired data sets are unaffected by the function, only the y-value data are rounded. For time series data sets, missing values are kept missing.

For paired data sets, use the *setCurve()* method to first select the paired data curve(s).

See also: round(); truncate(); setCurve().
Parameters:
`significantDigits` – An integer specifying the number of significant digits to use in the rounding.
`powerOfTensPlace` – An integer specifying the power of tens place to use in the rounding.
Example: `roundedData = dataSet.roundOff(5, -2)`
Returns: A new **HecMath** object of the same type as the current object.

8.15.98 Screen for Erroneous Values Based on Constant Value

```
screenWithConstantValue(string duration,
  floating-point rejectTolerance,
  floating-point questionTolerance,
  floating-point minThreshold,
  integer maxMissing)
```

Flag values in time series whose values do not change more than a specified amount over a specified duation as questionable or rejected.

Values in the time series are screened for quality. All (non-missing) values whose difference over the specified duration is below *rejectTolerance* are marked with a rejected quality flag. Other values whose difference are over the specified duration is below *questionTolerance* are marked with a questioned quality flag. Other values are marked with an okay quality flag unless they are already marked as rejected or questioned. Only values above the specified *minThreshold* value are screened as described. Only time periods that contain no more that *maxMissing* missing values are screened as described.

Parameters:
tsc - The time series to derive from.
durationStr - The time period over which to perform the test, of the form nX, where in is an integer and X is one of Minutes, Hours, or Days, which can be abbreviated to one or more characters.
rejectTolerance - Values will be marked as rejected if they do not differ by more than this amount over the specified duration. To

disable marking values as rejected, set this tolerance to less than zero.

questionTolerance - Values will be marked as questioned if they do not differ by more than this amount over the specified duration. To disable marking values as questioned, set this tolerance to less than zero.

minThreshold - Values will be screened only if they are greater than this value.

maxMissing - The maximum number of missing values to tolerate in the specified duration for screening purposes.

Example:

```
newDataSet = dataset.screenWithConstantValue(string
duration,
  floating-point rejectTolerance,
  floating-point questionTolerance,
  floating-point minThreshold,
  integer maxMissing)
```

Returns: A copy of the time series with values with the quality set as described.

8.15.99 Screen for Erroneous Values Based on Duration Magnitude

```
screenWithDurationMagnitude(string durationStr,
  floating-point minRejectLimit,
  floating-point minQuestionLimit,
  floating-point maxRejectLimit,
  floating-point maxQuestionLimit)
```

Flag values in time series whose accumulation over a specified period lies minimum/maximum limits values as questionable or rejected.

Values in the time series are screened for quality. All (non-missing) values whose accumulated value for the specified duration is below *minRejectLimit* or above *maxRejectLimit* are marked with a rejected quality flag. Other values whose accumulated value for the specified duration is below *minQuestionLimit* or above *maxQuestionLimit* are marked with a questioned quality flag. Other values are marked with an okay quality flag unless they are already marked as rejected or questioned.

Parameters:

DurationStr - The time period over which to perform the test, of the form nX, where in is an integer and X is one of Minutes, Hours, or Days, which can be abbreviated to one or more characters.

minRejectLimit - Values whose accumulated value for the specified duration is below this will be marked as rejected.

minQuestionLimit - Values whose accumulated value for the specified duration is below this, but not below

minRejectLimit - will be marked as questioned.
maxQuestionLimit - Values whose accumulated value for the specified duration is above this, but not above
maxRejectLimit - will be marked as questioned.
maxRejectLimit - Values whose accumulated value for the specified duration is above this will be marked as rejected.

Example:

```
screenedData = tsData.
screenWithDurationMagnitude(string durationStr,
  floating-point minRejectLimit,
  floating-point minQuestionLimit,
  floating-point maxRejectLimit,
  floating-point maxQuestionLimit)
```

Returns: A copy of the time series with values with the quality set as described.

8.15.100 Screen for Erroneous Values Based on Forward Moving Average

```
screenWithForwardMovingAverage(integer numberToAverageOver,
  floating-point changeLimit)
```

Flag values in time series exceeding maximum change from a forward moving average. A running forward moving average of valid values is progressively computed over *numberToAverageOver* points. Values exceeding the moving average computed for the preceeding point location by more than *changeLimit* are flagged or set to the "Missing" value depending upon the settings for *setInvalidToMissingValue* or *qualityFlagForInvalidValue* parameters. Values failing the quality test are excluded from the forward moving average computation for the subsequent points. The maximum change comparison is done only when consecutive values are not flagged.

Parameters:
numberToAverageOver - the number of values to average over for computing the forward moving average. Must be greater than two.
changeLimit - allowed deviation in the tested value from the forward moving average.

Example:

```
screenedData = tsData.screenWithForwardMovingAverage(
integer numberToAverageOver, floating-point
changeLimit)
```

Returns: a copy of the time series with values failing the quality test set to undefined and with the quality flag set to missing.

8.15.101 Screen for Erroneous Values Based on Forward Moving Average (Missing Values)

```
screenWithForwardMovingAverage(integer
numberToAverageOver,
  floating-point changeLimit,
  boolean setInvalidToMissingValue,
  string qualityFlagForInvalidValue)
```

Screen the current time series data set for possible erroneous values based on the deviation from the forward moving average over *numberToAverageOver* values computed at the previous point. If the deviation from the moving average is greater than changeLimit, the value fails the screening test. Data values failing the screening test are assigned a quality flag and/or are set to missing.

Missing values and values failing the screening test are not counted in the moving average and the divisor of the average is less one for each such value. At least two values must be defined in the moving average else the moving average is undefined and value being examined is screened acceptable.

If *setInvalidToMissingValue* is true, values failing the screening test are set to missing.

If *qualityFlagForInvalidValue* is set to a character or string recognized as a valid quality flag, the quality flag will be set for tested values. If there is no previously existing quality available for the time series, the quality flag array will be created for the time series. Values failing the quality test are set to the user specified quality flag for invalid values. If there is existing quality data and the time series value passes the quality test, the existing quality flag for the points is unchanged. If there was no previously existing quality and the time series value passes the quality test, the quality flag for the point is set to "Okay".

The acceptable values for *qualityFlagForInvalidValue* strings are: "M" or "Missing", "R" or "Rejected", "Q" or "Questionable". A blank string (" ") is entered to disable the setting of the quality flag.

For the example:

```
resultantDataSet = dataSet.screenWithForwardMovingAverage
(16, 100., TRUE,  "R")
```

the forward moving average will be computed over 16 values, values deviating from the moving average by more than 100.0 will be set to missing and flagged as rejected.

Parameters:

`numberToAverageOver` – An integer specifying the number of averaging values. Must be at least two.

`changeLimit` – A floating-point number specifying the maximum change allowed in the tested value from the forward moving average value.

`setInvalidToMissingValue` – Either True or False, specifying whether time series values failing the screening test are set to the "Missing" value.

`qualityFlagForInvalidValue` - A string representing the quality flag setting for values failing the screening test. The accepted character strings are: "M" or "Missing", "R" or "Rejected", "Q" or "Questionable". An empty string (" ") is entered to disable the setting of the quality flag.

Example:

```
screenedData = tsData.screenWithForwardMovingAverage
  (16, 100., TRUE, "R")
```

Returns: A new TimeSeriesMath object.

Generated Exceptions: Throws a *HecMathException* if *numberToAverageOver* is less than two; if an unrecognized quality flag is entered for *qualityFlagForInvalidValue* or if *setInvalidToMissing Value* is false and *qualityFlagForInvalidValue* is blank (no action would occur).

8.15.102 Screen for Erroneous Values Based on Maximum/Minimum Range (Missing Values)

```
screenWithMaxMin(floating-point minValueLimit,
  floating-point maxValueLimit,
  floating-point changeLimit,
  boolean setInvalidToMissingValue,
  string qualityFlagForInvalidValue)
```

Flag values in a time series data set exceeding minimum and maximum limit values or maximum change limit.

Values in the time series are screened for quality. Values below *minValueLimit* or above *maxValueLimit* or with a change from the previous time series value greater than *changeLimit* fail the screening test. The maximum change comparison is done only when consecutive values are not flagged.

If *setInvalidToMissingValue* is set to true, values failing the screening test are set to the "Missing" value.

If *qualityFlagForInvalidValue* is set to a character or string recognized as a valid quality flag, the quality flag will be set for tested values. If there is no previously existing quality available for the time series, the quality flag array will be created for the time series. Values failing the quality test are set to the user specified quality flag for invalid values. If there is existing quality data and the time series value passes the quality test, the existing quality flag for the points is unchanged. If there was no previously existing quality and the time series value passes the quality test, the quality flag is set to "Okay".

For example:

```
resultantDataSet = dataSet.screenWithMaxMin (0.0, 1000., 
100., FALSE, “R”)
```

time series values less than 0.0, or greater than 1000., or with a change from a previous point greater than 100 will be flagged as "Rejected". Flagged points however will not be set to the "Missing" value.

Parameters:

`minValueLimit` – A floating-point number specifying the minimum valid value limit.

`maxValueLimit` - A floating-point number specifying the maximum valid value limit.

`changeLimit` - A floating-point number specifying the maximum change allowed in the tested value from the previous time series value.

`setInvalidToMissingValue` - Either True, or False, specifying whether time series values failing the screening test are set to the "Missing" value.

`qualityFlagForInvalidValue` - A string representing the quality flag setting for values failing the screening test. The accepted character strings are: "M" or "Missing", "R" or "Rejected", "Q" or "Questionable". An empty string (" ") is entered to disable the setting of the quality flag.

Example:

```
screenedData = tsData.screenWithMaxMin(0., 1000., 100., 
FALSE, “R”)
```

Returns: A new **TimeSeriesMath** object.

Generated Exceptions: Throws a *HecMathException* if an unrecognized quality flag is entered for **`qualityFlagForInvalidValue`** or if **`setInvalidToMissingValue`** is false and **`qualityFlagForInvalidValue`** is blank (no action would occur).

8.15.103 Screen for Erroneous Values Based on Maximum/Minimum Range

```
screenWithMaxMin(floating-point minValueLimit,
  floating-point maxValueLimit,
  floating-point changeLimit)
```

Flag values in time series exceeding minimum and maximum limit values or maximum change limit.

Values in the time series are screened for quality. Values below *minValueLimit* or above *maxValueLimit* or with a change from the previous time series value greater than *changeLimit* fail the quality test. The maximum change comparison is done only when consecutive values are not flagged.

Values failing the screening test are set to the "Missing" value.

Parameters:
minValueLimit - minimum valid value limit.
maxValueLimit - maximum valid value limit.
changeLimit - maximum change allowed in the tested value from the previous time series value.

Example: `screenedData = screenWithMaxMin(floating-point minValueLimit, floating-point maxValueLimit, floating-point changeLimit)`

Returns: a copy of the time series with values failing the quality test set to missing.

8.15.104 Screen for Erroneous Values Based on Maximum/Minimum Range (Quality)

```
screenWithMaxMin(floating-point minValueLimit,
  floating-point maxValueLimit,
  floating-point changeLimit,
  boolean setInvalidToMissingValue,
  floating-point invalidValueReplacement,
  string qualityFlagForInvalidValue)
```

Flag values in time series exceeding minimum and maximum limit values or maximum change limit.

Values in the time series are screened for quality. Values below *minValueLimit* or above *maxValueLimit* or with a change from the previous time series value greater than *changeLimit* fail the quality test. The maximum change comparison is done only when consecutive values are not flagged.

Parameters:

minValueLimit - minimum valid value limit.
maxValueLimit - maximum valid value limit.
changeLimit - maximum change allowed in the tested value from the previous time series value.
setInvalidValueToSpecified - if true, time series values failing the quality test are set to the specified value.
invalidValueReplacement - The value to use for replacing invalid values is setInvalidToSpecifiedValue is set to "true".
qualityFlagForInvalidValue - character string representing the quality flag setting for values failing the quality test. The accepted character strings are: "M" or "Missing", "R" or "Rejected", "Q" or "Questionable".

Example:
```
screenedData = tsData.screenWithMaxMin(floating-point minValueLimit, floating-point maxValueLimit,
  floating-point changeLimit,
  boolean setInvalidToMissingValue,
  floating-point invalidValueReplacement,
  string qualityFlagForInvalidValue)
```

Returns: a copy of the time series with values failing the quality test set to missing.

8.15.105 Screen for Erroneous Values Based on Maximum/Minimum Range (Limits)

```
screenWithMaxMin(floating-point minRejectLimit,
  floating-point minQuestionLimit,
  floating-point maxQuestionLimit,
  floating-point maxRejectLimit)
```

Flag values in time series exceeding minimum and maximum limit values as questionable or rejected. Values in the time series are screened for quality. Values below *minRejectLimit* or above *maxRejectLimit* are marked with a rejected quality flag. Other values below *minQuestionLimit* or above *maxQuestionLimit* are marked with a questioned quality flag. Other values are marked with an okay quality flag unless they are already marked as rejected or questioned.

Parameters:

minRejectLimit - Values below this will be marked as rejected.
minQuestionLimit - Values below this, but not below minRejectLimit will be marked as questioned.
maxQuestionLimit - Values above this, but not above maxRejectLimit will be marked as questioned.
maxRejectLimit - Values above this will be marked as rejected.

Example:
```
screenedData = tsData.screenWithMaxMin(floating-point minRejectLimit,
  floating-point minQuestionLimit,
```

```
  floating-point maxQuestionLimit,
  floating-point maxRejectLimit)
```

Returns: A copy of the time series with values with the quality set as described.

8.15.106 Screen for Erroneous Values Based on Rate of Change

```
screenWithRateOfChange(floating-point minRejectLimit,
  floating-point minQuestionLimit,
  floating-point maxQuestionLimit,
  floating-point maxRejectLimit)
```

Flag values in time series whose rate of change from the last valid value exceed minimum/maximum limits values as questionable or rejected.

Values in the time series are screened for quality. Values whose rate of change from the last valid value is below *minRejectLimit* or above *maxRejectLimit* are marked with a rejected quality flag. Other whose rates of change from the last valid value are values below *minQuestionLimit* or above *maxQuestionLimit* is marked with a questioned quality flag. Other values are marked with an okay quality flag unless they are already marked as rejected or questioned.

Rates are computed in units per hour.

Parameters:

minRejectLimit - Values whose rate of change from the last valid value are below this will be marked as rejected.

minQuestionLimit - Values whose rate of change from the last valid value are below this, but not below minRejectLimit - will be marked as questioned.

maxQuestionLimit - Values whose rate of change from the last valid value are above this, but not above maxRejectLimit will be marked as questioned.

maxRejectLimit - Values whose rate of change from the last valid value are above this will be marked as rejected.

Example:

```
newDataSet = dataset.screenWithRateOfChange(
floating-point minRejectLimit,
  floating-point minQuestionLimit,
  floating-point maxQuestionLimit,
  floating-point maxRejectLimit)
```

Returns: A copy of the time series with values with the quality set as described.

8.15.107 Select a Paired Data Curve by Curve Label

```
setCurve(string curveName)
```

Select, by curve label, the paired data curve for performing subsequent arithmetic operations or math functions. By default, a paired data set loaded from file has all curves selected.

A paired data set may contain more than one set of y-values. However, a user may wish to modify only one curve of the data set. For example, using the function ".add(2.0)" would by default add 2.0 to all y-values for all curves. The *setCurve()* call may be used to limit the operation to just one selected set of y-values. The function searches the paired data set list of curve labels for a match to *curveName*. If a match is found, that curve is set as the selected curve.

See also: setCurve(integer curveNumber)
Example: `damageCurve.setCurve("RESIDENTIAL")`
Parameters: curveName – The curve label (a string) to set as the selected curve.
Returns: Nothing
Generated Exceptions: Throws a **HecMathException** – if *curveName* is not found in the paired data set curve labels.

8.15.108 Select a Paired Data Curve by Curve Number

```
setCurve(integer curveNumber)
```

Select, by curve number, the paired data curve for performing subsequent arithmetic operations or math functions. By default, a paired data set loaded from file has all curves selected.

A paired data set may contain more than one set of y-values. However, a user may wish to modify only one curve of the data set. For example, using the function ".add(2.0)" would by default add 2.0 to all y-values for all curves. The *setCurve()* call can be used to limit the operation to just one selected set of y-values. The function sets a curve index internal to the paired data set. The option is to select one curve or all curves.

Curve numbering begins with "0". If a paired data set has two curves, the first curve is selected by, "setCurve(0)". To select the second curve, use "setCurve(1)".

All curves in a paired data set are selected by setting *curveNumber* to -1.

See also: setCurve(String curveName)

Parameters: curveNumber – An integer specifying the curve to set as the selected curve. Curve numbering begins with 0. Set to –1 to select all curves.
Example: `ruleCurve.setCurve(-1)`
Returns: Nothing

8.15.109 Set Data Container

```
setData(hec.io.DataContainer container)
```

Sets the data container for the current data set. For time series data sets, this is a *hec.io.TimeSeriesContaine*r. For paired data sets, container should be a *hec.io.PairedDataContainer*. Containers are generated by some of the other functions.

The *hec.io.DataContainer* class and the *hec.io.TimeSeriesContainer* and the *hec.io.PairedDataContainer* subclasses contain the time series and paired data values.

Parameters: container – A hec.io.TimeSeriesContainer for time series data sets, or a hec.io.PairedDataContainer for paired data sets.
Example: `dataSet.setData(TSContainer)`
Returns: Nothing
Generated Exceptions: Throws a **HecMathException** if container is not of type *hec.io.TimeSeriesContainer* for time series data sets or not of type *hec.io.PairedDataContainer* for paired data sets.

8.15.110 Set Location Name for Data Set

```
setLocation(String locationName)
```

Set the location name for a data set, which changes the B-Part of the HEC-DSS pathname. The new pathname will be used in plots, tables, and in the *write()* method of DSSFile objects.

Parameters: locationName – A string specifying the new location name for the data set.
Example: `dataSet.setLocation("OAKVILLE")`
Returns: Nothing

8.15.111 Set Parameter for Data Set

```
setParameterPart(String parameterName)
```

Set the parameter name for a data set, which changes the C-Part of the HEC-DSS pathname. The new pathname will be used in plots, tables, and in the *write()* method of DSSFile objects.

Parameters: parameterName – A string specifying the new parameter name for the data set.
Example: `dataSet.setParameterPart("ELEV")`
Returns: Nothing

8.15.112 Set Pathname for Data Set

`setPathname``(String pathname)`

Set the pathname for a data set to the specified pathname. Subsequent operations using the data set such as *getData()* or *DSSFile.write()* will use or reflect the new pathname.

Parameters: pathname – A string specifying the new pathname for the data set.
Example: `dataSet.setPathname("//OAKVILLE/STAGE//1HOUR/OBS/")`
Returns: Nothing

8.15.113 Set Time Interval for Data Set

`setTimeInterval``(String interval)`

Set the time interval for a data set, which changes the E-Part of the pathname. The new pathname will be used in plots, tables, and in the *write()* method of DSSFile objects.

Parameters: interval – A string specifying the new interval for the data set.
Example: `dataSet.setTimeInterval("1HOUR")`
Returns: Nothing

8.15.114 Set Data Type for Time Series Data Set

`setType``(string typeString)`

Set the data for a time series data set.

Parameters: typeString – A string specifying the data type for the data set. This should be "INST-CUM", "INST-VAL", "PER-AVER" or "PER-CUM"

Example: `dataSet.setType("PER-AVER")`
Returns: Nothing

8.15.115 Set Units Label for Data Set

`setUnits``(String unitsString)`

Set the units label for a data set. For a paired data set, the call sets the y-units label.

Parameters: unitsString – A string specifying the units label for the data set.
Example: `dataSet.setUnits("CFS")`
Returns: Nothing

8.15.116 Set Version Name for Data Set

`setVersion``(String versionName)`

Set the version name for a data set, which changes the F-Part of the pathname. The new pathname will be used in plots, tables, and in the *write()* method of DSSFile objects.

Parameters: version – A string specifying the new location for the data set.
Example: `dataSet.setVersion("OBSERVED")`
Returns: Nothing

8.15.117 Set Watershed Name for Data Set

`setWatershed``(String watershedName)`

Set the watershed (or river) name for a data set, which changes the A-Part of the pathname. The new pathname will be used in plots, tables, and in the *write()* method of DSSFile objects.

Parameters: watershedName – A string specifying the new watershed name for the data set.
Example: `dataSet.setWatershed("OAK RIVER")`
Returns: Nothing.

8.15.118 Shift Adjustment of Time Series Data

```
shiftAdjustment(TimeSeriesMath tsData)
```

Derive a new time series data set by linear interpolation of values in the current time series data set at the times defined by the time series data set **tsData**. If times in the new time series precede the first data point in the current time series, the value for these times is set to 0.0. If times in the new time series occur after the last data point in the current time series, the value for these times is set to the value of the last point in the current time series. Interpolation of values with the *shiftAdjustment* function is shown in Figure 8.14.

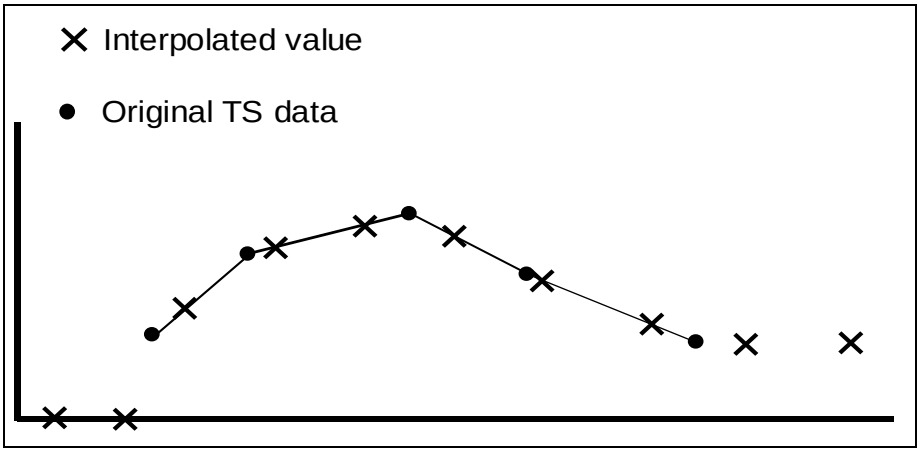

Figure 8.14 Interpolation of Time Series Values using Shift Adjustment Function

Both time series data sets may be regular or irregular interval. Interpolated points must be bracketed or coincident with valid (not missing) values in the original time series; otherwise the values are set as missing.

Parameters: tsData – A regular or irregular interval time series data set.
Example: `tsInterp = tsValues.shiftAdjustment(tsData)`
Returns: A new **TimeSeriesMath** object.

8.15.119 Shift Time Series in Time

```
shiftInTime(string timeShiftString)
```

Shift the times in the current time series data set by the amount specified with *timeShiftString*. The data set may be regular or irregular interval time series data. Data set values are unchanged.

timeShiftString has the form "nT", where "n" is an integer number and "T" is "M"(inute), "H"(our), "D"(ay), "W"(eek), "Mo"(nth),or "Y"(ear).

Only the first character is significant for "T", except for "Month", which requires at least two characters.

Parameters: *timeShiftString* – A string specifying the time increment to shift times in the current time series data set.
Example: `TsShifted = tsData.shiftInTime("3H")`
Returns: A new **TimeSeriesMath** object.

8.15.120 Sign Function

`sign``()`

Derive a new time series composed of the signs of the values of the current time series. Values which are missing in the original time series will be missing in the new time series. Values in the current time series will be represented as -1.0, 0.0 or 1.0 in the derived time series, depending on whether the original value is < 0.0, 0.0 or > 0.0.

Parameters: Takes no Parameters
Example: `newDataSet = dataSet.sign()`
Returns: A new time series composed of the negatives of the values of the current time series

8.15.121 Sine Trigonometric Function

`sin``()`

Derive a new time series or paired data set from the sine of values of the current data set. The resultant data set values are in radians. For time series data, missing values are kept as missing.

For paired data sets, use the *setCurveMethod* to first select the paired data curve(s).

See also: setCurve()
Parameters: Takes no parameters
Example: `newDataSet = dataSet.sin()`
Returns: A new **HecMath** object of the same type as the current object

8.15.122 Skew Coefficient

`skewCoefficient``()`

Compute the skew coefficient of the current time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `skewCoefficient = dataSet.skewCoefficient()`
Returns: A floating-point value representing the skew coefficient of the current time series.

8.15.123 Snap Irregular Times to Nearest Regular Period

```
snapToRegularInterval(string timeIntervalString,
  string timeOffsetString,
  string timeBackwardString,
  string timeForwardString)
```

"Snap" data from the current irregular or regular interval time series to form a new regular interval time series of the specified interval and offset. For example, a time series record from a gauge recorder collects readings 6 minutes past the hour. The function may be used to “snap” or shift the time points to the top of the hour.

The regular interval time of the resultant time series is specified by *timeIntervalString*. *timeIntervalString* is a valid HEC time increment string (e.g. "1MIN", "15MIN", "1HOUR", "6HOUR", "1DAY", "1MONTH").

Times in the resultant time series may be shifted (offset) from the regular interval time by the increment specified by *timeOffsetString*. As an example, the offset could be used to shift times from the top of the hour to instead six minutes past the hour. Data from the original time series is "snapped" to the regular interval if the time of the data falls within the time window set by the *timeBackwardString* and the *timeForwardString*. That is, if the new regular interval is at the top of the hour and the time window extends to nine minutes before the hour and fifteen minutes after the hour, an original data point at 0852 would be snapped to the time 0900 while a point at 0916 would be ignored.

timeOffsetString, timeBackwardString and *timeForwardString* are time increment strings expressed as "nT", where "n" is an integer number and "T" is one of the time increments: "M(INUTES)", "D(AYS)" or "H(OUR)" (characters in the parenthesis are optional). For the example of the previous paragraph, *timeIntervalString* would be "1HOUR", *timeOffsetString* would be "0M", *timeBackwardString* would be "9M" (or "9min") and *timeForwardString* would be "15M.” A blank string (" ") is equivalent to "0M".

By default values in the resultant regular interval time series data set are set to missing unless matched to times in the current time series data set within the time window tolerance set by *timeBackwardString* and *timeForwardString*.

Parameters:

`timeIntervalString` – A string specifying the regular time interval for the resultant time series.

`timeOffsetString` – A string specifying the offset of the new time points from the regular interval time. This variable may be an empty string (" ") or None.

`timeBackwardString` – A string specifying the time to look backwards from the regular time interval for valid time points.

`timeForwardString` – A string specifying the time to look forward from the regular time interval for valid time points.

Example: `rtsData = itsData.snapToRegularInterval("1HOUR", None, "5Min", "5Min")`

Returns: A new regular interval **TimeSeriesMath** object.

8.15.124 Square Root

`sqrt``()`

Derive a new time series or paired data set computed from the square root of values of the current data set. For time series data, missing values are kept as missing. Values less than zero are set to missing.

For paired data sets, use *setCurve* to first select the paired data curve(s).

See also: setCurve()

Parameters: Takes no parameters

Example: `newDataSet = dataSet.sqrt()`

Returns: A new **HecMath** object of the same type as the current object.

8.15.125 Standard Deviation of Time Series

`standardDeviation``()`

Compute the standard deviation value of the current time series data set. Missing values are ignored.

Parameters: Takes no parameters

Example: `stdDev = tsData.standardDeviation()`

Returns: A floating-point value representing the standard deviation of the current time series

8.15.126 Standard Deviation of Time Series (*tsMathArray*)

```
standardDeviation(list of HecMath tsMathArray)
```

Determine the standard deviation of the current time series and the each time series in the parameter, *tsMathArray*. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the standard deviation of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* - the array of time series to be compared with the current time series.
Example: `stdDev = tsData.standardDeviation(list of HecMath tsMathArray)`
Returns: a new time series representing the standard deviation of all time series.

8.15.127 Straddle Stagger Hydrologic Routing

```
straddleStaggerRouting(integer numberToAverage,
  integer numberToLag,
  integer numberSubreaches)
```

Route the current regular interval time series data set using the Straddle-Stagger hydrologic routing method. *numberToAverage* specifies the number of ordinates to average over (Straddle). *numberToLag* specifies the number ordinates to lag (Stagger). The number of routing subreaches is set by *numberSubreaches*.

Parameters:
numberToAverage – An integer specifying the number of ordinates to average over (Straddle).
numberToLag – An integer specifying the number of ordinates to lag (Stagger).

numberSubreaches – An integer specifying the number of routing subreaches.

Example: `tsRouted = tsFlow.straddleStaggerRouting (numberAver, lag, reachCount)`

Returns: A new **TimeSeriesMath** object.

8.15.128 Subtract a Constant

`subtract``(floating-point constant)`

Subtract the value constant from all valid values in the current time series or paired data set. For time series data, missing values are kept as missing.

For paired data, constant is subtracted from y-values only. Use the *setCurve* method to first select the paired data curve(s).

See also: subtract(HecMath hecMath); setCurve()

Parameters: constant - A floating-point value.

Example: `newDataSet = dataSet.subtract(5.3)`

Returns: A new **HecMath** object of the same type as the current object.

8.15.129 Subtract a Data Set

`subtract``(TimeSeriesMath tsData)`

Subtract the values in the data set tsData from the values in the current data set. Both data sets must be time series data set.

When subtracting one time series data set from another, there is no restriction that times in the two data sets match exactly. However, only values with coincident times will be subtracted. Times in the current time series data set that cannot be matched with times in the second data set are set missing. Values in the current data set that are missing are kept as missing. Either or both data sets may be regular or irregular interval time series.

See also: subtract(floating-point constant)

Parameters: tsData - a **TimeSeriesMath** object.

Example: `newDataSet = dataSet.subtract(otherDataSet)`

Returns: A new **TimeSeriesMath** object.

8.15.130 Successive Differences for Time Series

successiveDifferences()

Derive a new time series from the difference between successive values in the current regular or irregular interval time series data set. The current data must be of type "INST-VAL" or "INST-CUM". A value in the resultant time series is set to missing if either the current or previous value in the current time series is missing (need to have two consecutive valid values). If the data type of the current data set is "INST-CUM" the resultant time series data set is assigned the type "PER-CUM", otherwise the data type does not change.

See also: timeDirivative()
Parameters: Takes no parameters
Example: `newTsData = tsData.successiveDifferences()`
Returns: A new **TimeSeriesMath** object.
Generated Exceptions: Throws a **HecMathException** if the current data set is not of type "INST-VAL" or "INST-CUM".

8.15.131 Sum Values in Time Series

sum()

Sum the values of the current time series data set. Missing values are ignored.

Parameters: Takes no parameters
Example: `total = tsData.sum()`
Returns: A floating-point value representing the sum of all valid values of the current time series.

8.15.132 Sum Values in Time Series (*tsMathArray*)

sum(list of HecMath tsMathArray)

Add each of the time series in the parameter, *tsMathArray*, to the current time series. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current time series and each time series in *tsMathArray*, the values will be summed provided values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quality values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* - the array of time series to be added to the current time series.
Example: `tsData.sum(list of HecMath tsMathArray)`
Returns: a new time series representing the sum of the two time series

8.15.133 Tangent Trigonometric Function

`tan``()`

Derive a time series or paired data set computed from the tangent of values of the current data set. For time series data, missing values are kept as missing. If the cosine of the current time series value is zero, the value is set missing.

For paired data sets, use the *setCurve* method to first select the curve(s).

See also: setCurve()
Example: `newDataSet = dataSet.tan()`
Parameters: Takes no parameters
Returns: A new **HecMath** object of the same type as the current object.

8.15.134 Time Derivative (Difference per Unit Time)

`timeDerivative``()`

Derive a new time series data set from the successive differences per unit time of the current regular or irregular interval time series data set. For the time “t”,

$$TS2(t) = (TS1(t) – TS1(t-1)) / DT$$

where DT is the time difference between t and t-1. For the current form of the function, the units of DT are minutes.

A value in the resultant time series is set to missing if either the current or previous value in the original time series is missing (need to have two consecutive valid values). By default, the data type of the resultant time series data set is assigned as "PER-AVER".

See also: successiveDifferences()
Parameters: Takes no parameters
Example: `newTsData = tsData.timeDerivative()`
Returns: A new **TimeSeriesMath** object

8.15.135 Transform Time Series to Regular Interval

```
transformTimeSeries(string timeIntervalString,
  string timeOffsetString,
  string functionTypeString)
```

Generate a new regular interval time series data set from the current regular or irregular time series. The new time series is computed having the regular time interval specified by *timeIntervalString* and time offset set by *timeOffsetString*.

Values for the new time series are computed from the original time series data set using one of seven available functions. The function is selected by setting *functionTypeString* to one of the following types:

"INT"	-	Interpolate at end of interval
"MAX"	-	Maximum over interval
"MIN"	-	Minimum over interval
"AVE"	-	Average over interval
"ACC"	-	Accumulation over interval
"ITG"	-	Integration over interval
"NUM"	-	Number of valid data over interval

where "interval" is the interval between time points in the new time series.

The regular interval time of the new time series is specified by *timeIntervalString*. *timeIntervalString* is a valid HEC time increment string (e.g. “"1MIN", "15MIN", "1HOUR", "6HOUR", "1DAY", "1MONTH").

Times in the resultant time series may be shifted (offset) from the regular interval time by the increment specified by *timeOffsetString*. As an example, the offset could be used to shift times from the top of the hour to six minutes past the hour. Typically no offset is used.

The data type of the original time series data governs how values are interpolated. Data type "INST-VAL" (or "INST-CUM") considers the value to change linearly over the interval from the previous data value to the current data value. Data type "PER-AVER" considers the value to be constant at the current data value over the interval. Data type "PER-CUM" considers the value to increase from 0.0 (at the start of the interval)

up to the current value over the interval. Interpolation of the three data types is illustrated in Figure 8.15.

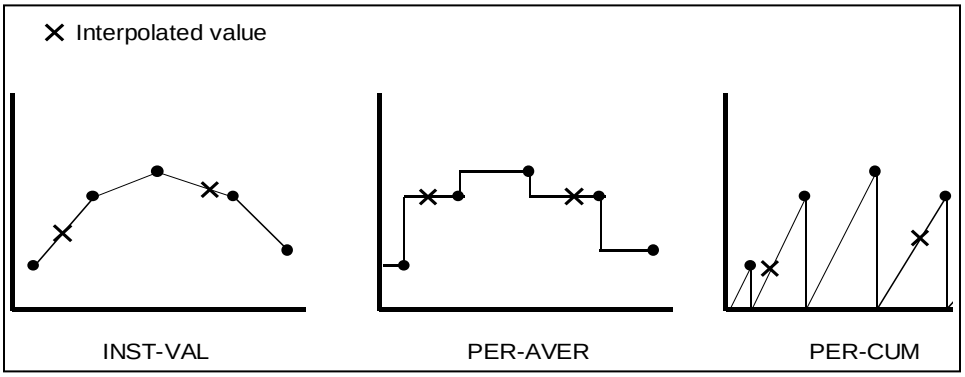

Figure 8.15 Interpolation of "INST-VAL", "PER-AVER" and "PER-CUM" Data

How interpolation is performed for a specific data type influences the computation of new time series values for the selected function. For example, if the data type is "INST-VAL", the function "Maximum over interval" is evaluated by: Finding the maximum value of the data points from the original time series that are inclusive in the new time interval. Linearly interpolate values at beginning and ending of the new time interval, and determine if these values represent the maximum over the interval.

Referring to the plots in Figure 8.15, the "Average over interval" function is applied to a time series by integrating the area under the curve between interpolated points and dividing the result by the interval time.

See also: `transformTimeSeries( TimeSeriesMath tsData, string functionTypeString)`

Parameters:

`timeIntervalString` – A string specifying the regular time interval for the resultant time series.

`timeOffsetString` – A string specifying the offset of the new time points from the regular interval time. This variable may be a blank string (" ").

`functionTypeString` – A string specifying the method for computing values for the new time series data set.

Example: `newTsData = tsData.transformTimeSeries("1Day", "0M", "AVE")`

Returns: A new regular interval **TimeSeriesMath** object.

8.15.136 Transform Time Series to Irregular Interval

`transformTimeSeries``(TimeSeriesMath tsData,string functionTypeString)`

Generate a new time series data set from the current regular or irregular time series. The times for the new data set are defined by the times in **tsData**, which may be a regular or irregular time series data set.

Values for the new time series are computed from the original time series data set using one of seven available functions. The function is selected by setting *functionTypeString* to one of the following types:

"INT" - Interpolate at end of interval
"MAX" - Maximum over interval
"MIN" - Minimum over interval
"AVE" - Average over interval
"ACC" - Accumulation over interval
"ITG" - Integration over interval
"NUM" - Number of valid data over interval

where "interval" is the interval between time points in the new time series.

The data type of the original time series data governs how values are interpolated. Data type "INST-VAL" (or "INST-CUM") considers the value to change linearly over the interval from the previous data value to the current value. Data type "PER-AVER" considers the value to be constant at the current value over the interval. Data type "PER-CUM" considers the value to increase from 0.0 (at the start of the interval) up to the current value over the interval. Interpolation of the three data types is illustrated in Figure 8.15 (page 8-156).

How interpolation is performed for a specific data type influences the computation of new time series values for the selected function. For example, if the data type is "INST-VAL", the function "Maximum over interval" is evaluated by: Finding the maximum value of the data points from the original time series that are inclusive in the new time interval. Linearly interpolate values at beginning and ending of the new time interval, and determine if these values represent the maximum over the interval.

Referring to the plots in Figure 8.15 (page 8-156), the "Average over interval" function is applied to a time series by integrating the area under the curve between interpolated points and dividing the result by the interval time.

See also: transformTimeSeries(string timeIntervalString, string timeOffsetString, string functionTypeString)

Parameters:

`tsMath` – A TimeSeriesMath object used to define the times for the new data set.

`functionTypeString` – A String specifying the method for computing values for the new time series data set.

Example: `newTsData = tsValues.transformTimeSeries (tsTimeTemplate, "MAX")`
Returns: A new **TimeSeriesMath** object

8.15.137 Truncate to Whole Numbers

`truncate``()`

Truncates values in a time series or paired data set to the nearest whole number. For example:

10.99 is truncated to 10.

The x-values in paired data sets are unaffected by the function, only the y-value data are truncated. Missing values remain missing.

For paired data, use the setCurve method to first select the curve(s).

See also: setCurve()
Parameters: Takes no parameters
Example: `newDataSet = dataSet.truncate()`
Returns: A new **HecMath** object of the same type as the current object

8.15.138 Two Variable Rating Table Interpolation

`twoVariableRatingTableInterpolation``(TimeSeriesMath tsDataX,TimeSeriesMath tsDataZ)`

Derive a new time series data set by using the x-y curves in the current paired data set to perform two-variable rating table interpolation of the time series *tsDataX* and *tsDataZ*. For two-variable rating table interpolation, the current paired data set should have more than one curve (multiple sets of y-values).

As an example, reservoir release is a function of both the gate opening height and reservoir elevation Figure 8.16 (page 8-159). For each gate opening height, there is a reservoir elevation-reservoir release curve, where reservoir elevation is the independent variable (x-values) and reservoir release the dependent variable (y-values) of a paired data set. Each paired data curve has a curve label. In this case, the curve label is assigned the gate opening height. Using the paired data set shown in Figure 8.16 (page 8-159), the function may be employed to interpolate time series values of reservoir elevation (*tsDataX*) and gate opening height (*tsDataZ*) to develop a time series of reservoir release.

No extrapolation is performed. If time series values from *tsDataX* or *tsDataZ* are outside the range bounded by the paired data, the new time series value is set to missing. Units and parameter type in the new time series are set to the y-units label and parameter of the current paired data set. All other names and labels are copied over from *tsDataX*.

Times for *tsDataX* and *tsDataZ* must match. Curve labels must be set for curves in the rating table paired data set and must be interpretable as numeric values.

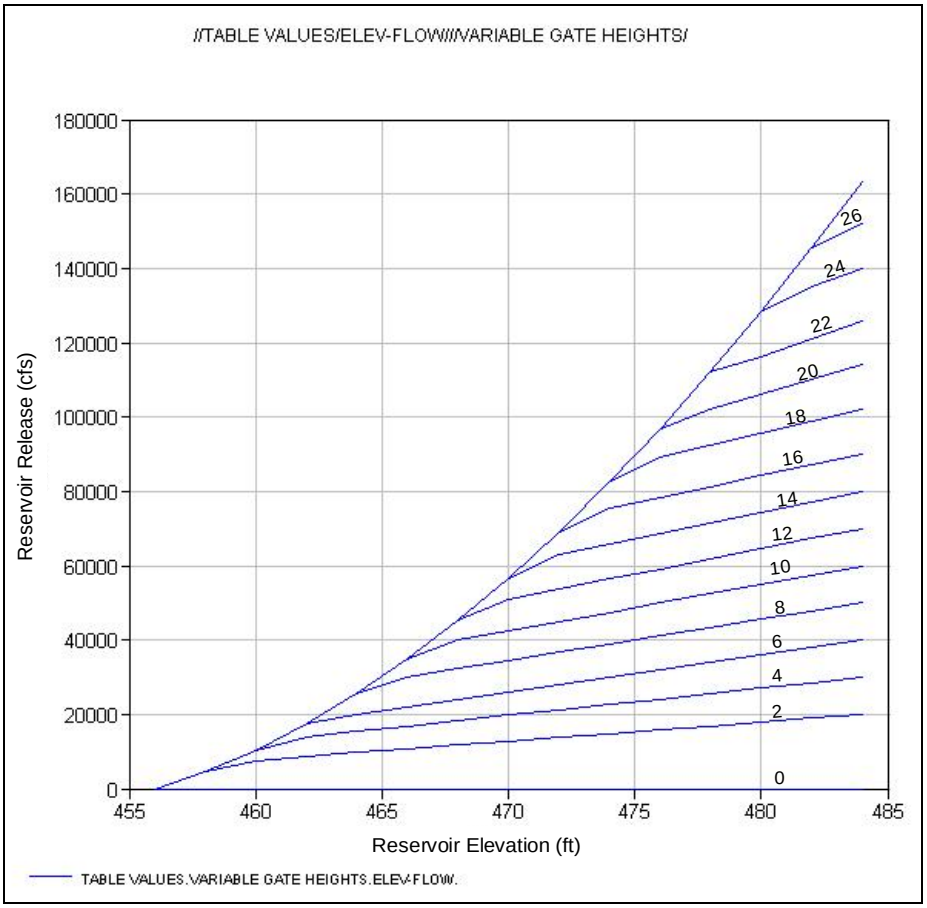

Figure 8.16 Example of two variable rating table paired data, reservoir release as a function of reservoir elevation and gate opening height (curve labels)

Parameters:

`tsDataX` – A regular or irregular interval **TimeSeriesMath** object, interpreted as x-ordinate values in the two variable interpolation.

`tsDataZ` – A regular or irregular interval **TimeSeriesMath** object, interpreted as z-ordinate values, (value defined by the paired data curve labels).

Example:
```
tsOutflow = gateCurve.twoVariableRatingTable
Interpolation( tsElevation, tsGateOpening)
```

Returns: A new **TimeSeriesMath** object.

Generated Exceptions: Throws a *HecMathException* if times do not match for *tsData***X** and *tsDataZ*; if the paired data curve labels are blank or cannot be interpreted as number values.

8.15.139 Variance Function

```
variance(list of HecMath tsMathArray)
```

Determine the variance of the current time series and the each time series in the parameter, *tsMathArra*y. A new time series will be created which duplicates the time points of the current time series. Where time points match for the current and *tsMathArray*, the values will be the variance of all time series for that time, provided the values for all time series are valid values (not missing). Points in the current time series which cannot be matched to valid points in *tsMathArray* are set to missing. Values in the current time series which are missing remain missing in the new time series.

The new time series will always have quality defined. For a specific time, if any of the quailty values in the current or parameter time series is questionable or rejected, the quality value for that time in the new time series will be set to the most severe quality for that time (e.g. if questionable and rejected are both encountered, the new quality will be set to rejected.)

Parameters: *tsMathArray* - the array of time series to be compared with the current time series.
Example:
```
newDataSet =
  dataSet.variance([ts1,ts2,ts3,ts4,ts5])
```
Returns: a new time series representing the variance of all time series.