LabTalk Scripting Guide

Contents last updated August 2013. For the most up-to-date documentation, including detailed examples, visit: www.OriginLab.com/doc/LabTalk.

OriginLab Corporation
One Roundhouse Plaza
Northampton, MA 01060
USA
(413) 586-2013
(800) 969-7720
Fax (413) 585-0126
www.OriginLab.com

Table of Contents

1 Introduction

In this guide we introduce LabTalk, the scripting language in Origin. LabTalk is designed for users who wish to write and execute scripts to perform analysis and graphing of their data. The purpose of this guide is to help users who are generally familiar with programming in a scripting language to take advantage of the scripting capabilities in Origin. We provide sufficient detail for a user with basic knowledge of Origin to begin tailoring the software to meet their unique needs.

The guide starts with a quick introduction to LabTalk, followed by a chapter on language fundamentals, and a chapter outlining various ways to organize and execute scripts within the Origin environment. The remaining chapters are organized by various functional areas of Origin, such as importing, graphing, data analysis, user interaction, and automation.

A few reference tables are included at the end. However this guide is not a full language reference. The full LabTalk language reference documentation is accessible from the **Help** menu in Origin. New features are continually introduced to LabTalk with successive versions of Origin. These are typically marked with a version number stamp (i.e., 8.1, typically in a bold and/or red-colored font) in the language reference help file.

This guide should be used in conjunction with other resources for learning LabTalk, which are listed in the **Resources for Learning LabTalk** chapter.

This guide provides several script examples. To try these examples you can either type in the script, or you can simply copy and paste the script from the soft file version of this guide accessible from the **Help** menu.

2 Getting Started with LabTalk

2.1 Hello World

We begin with a classic example to show you how to execute LabTalk script.

1. Open Origin, and from the **Window** menu, select the **Script Window** option. The **Script Window** will open.
2. In this window, type the following text and then press Enter:

```
type "Hello World"
```

3. Here we used LabTalk command **type**. Commands can be abbreviated down to 2 characters, try:

```
ty "Hello World"
```

Origin will output the text **Hello World** directly beneath your command.

Note that when you press Enter, Origin adds a semicolon, **;**, at the end of the line and also executes that line of script.
To repeat the execution of a line of script, place the cursor anywhere within the line and press Enter. If you place the cursor at the end of the line, you need to remove the **;** before pressing Enter to execute that line of script.

Now let us see how to execute multiple lines of script from the script window:

1. With a workbook window active in Origin, open the **Script Window**
2. Type the following lines of script in the script window. At the end of each line, press Enter after typing the **;**. This will prevent execution of the line. We will later execute all lines together.

```
type "The current workbook is %h";
type "This book has $(page.nlayers) sheet(s)";
type "There are $(wks.ncols) columns in the active sheet";
```

3. Using the mouse, drag and select all lines of script. If using keyboard, place the cursor at the beginning of the script, then hold down the Shift key, and use the arrow keys to highlight all lines.
4. Press Enter to execute all selected lines of script. Depending on the workbook that was active, the output in the script window will be similar to the following text:

```
The current workbook is Book1
```

```
This book has 1 sheet(s)
There are 2 columns in the active sheet
```

In the above example, we used the **%H** String Register that holds the currently active window name (which could be a workbook, a matrix, or a graph). We then used the **page** and **wks** LabTalk Objects to get the number of sheets in the book and the number of columns in the sheet. The **$()** is a substitution notations which tells Origin to evaluate the expression within the **()** and return its value.

If you are typing in multiple lines of script in the Script Window, you can add a **;** at the end of a line and then press Enter to avoid execution of the line. This allows you to type in multiple lines without executing each line. You can then select all lines and press Enter to execute them all.

2.2 Using = to Get Quick Output

The script window can be used as a calculator to return results interactively. Type the following script in the script window and press Enter:

```
3 + 5 =
```

Origin computes the result and displays it in the next line:

```
3 + 5 = 8
```

The **=** character is typically used as an assignment operator, with a left- and right-hand side. When the right-hand side is missing, the interpreter will evaluate the expression on the left of the **=** character and print the result in the script window.

In the following example, we introduce the concept of variables in LabTalk. Entering the following assignment statement in the script window:

```
double A = 2
```

creates a variable A and initializes its value to 2. Then you can perform some arithmetic on variable A, such as multiplying by PI (a constant defined in Origin, π) and assign the result back to A:

```
A = A*PI
```

To display the current value of A, type:

```
A =
```

Press Enter and Origin responds with:

```
A = 6.2831853071796
```

In addition, there are List command to view a list of variables and their values. Type the following command and press Enter:

```
list
```

Origin will open the **LabTalk Variables and Functions** dialog that lists all variables.

You can also get a dump of a specific type of variables, for example

```
list v
```

to list the numeric variables.

2.3 Other Ways to Execute Script

In previous examples, you saw how to execute script from the **Script Window**. Origin provides several other ways to organize and execute LabTalk script. These are outlined in detail in the **Running and Debugging LabTalk Scripts** chapter. Here we take a quick look at a few of the methods to execute script: (1) from the **Custom Routine** toolbar button, (2) from a custom menu item, and (3) from a button in a graph page.

2.3.1 Custom Routine Button

Origin provides a convenient way to save script and run it with the push of a toolbar button.

1. While holding down **Ctrl+Shift** on your keyboard, press the **Custom Routine** button () located in the Standard Toolbar.
2. This opens **Code Builder**, Origin's native script editor. The file being edited is called **Custom.ogs**. The code has one section, **[Main]**, and contains one line of script:

```
[Main]
type -b $General.Userbutton;
```

3. Replace that line with the following:

```
[Main]
type -b "Hello World";
```

4. Then press the **Save** () button in the **Code Builder** window.
5. Now go back to the Origin application window and click the  button.

Origin will again output the text Hello World, but this time, because of the **-b** switch used with the **type** command, the text will be presented in a pop-up window.

2.3.2 Custom Menu Item

LabTalk script can be executed from a custom menu item.

1. Select the menu item **Tools: Custom Menu Organizer...** to open the **Custom Menu Organizer** dialog.
2. Make the **Add Custom Menu** tab active. Then right-click inside the left panel and select **New Main Popup** from the context menu.
3. In the right panel, enter a name for **Popup Text**, such as **My Menu**. then click outside of the edit box.

4. Select **My Menu** from the left panel, and then right click on it, and select **Add Item** from the context menu.
5. In the right panel, change the **Item Text** to **Hello World**, then add the following script to the **LabTalk Script** text box:

```
type -b "Hello World";
```

6. Click the **Close** button, and in the window that pops up, press **Yes** to save the menu changes as **Default** menu. In the file dialog that opens, press **Save** to save the file with the default name to the default folder (User Files Folder).
7. A new menu named **My Menu** should now appear in the menu bar, to the left of the **Window** menu. Click on this new menu item, and then click on the **Hello World** entry in the drop-down. A Hello World dialog will pop up.

2.3.3 Button in a Graph

Origin also provides the ability to add a button to a graph or worksheet, and then execute LabTalk script by pushing that button. This allows for script to be saved with a specific project or window.

1. Press the **New Graph** button () located in the Standard Toolbar to create a new graph.
2. Press the **Text Tool** button () in the Tools Toolbar, and then click on the newly created graph and type the text **My Button**. Then click outside the text to finish editing the text.
3. Right click on the text to bring up the context menu, and then select **Programming Control** to open the **Programming Control** dialog.
4. In the dialog, select **Button Up** from the **Script, Run After:** drop-down list, and then type the following script in the edit box:

```
type -b "Hello World";
```

5. Click OK to close the dialog. Now the text label becomes a button. Click the button. A Hello World dialog will pop up.

2.4 Script Example

We now present a script example that walks you through a particular scenario of importing and processing data, and then saving the project. This example uses several LabTalk language features such as **Commands**, **Objects**, and **X-Functions**. You will learn more details about these language features in subsequent chapters.

NOTE: We will use the **Script Window** to execute these statements. To execute a single line of code, make sure that you leave out the **;** at the end before pressing Enter. For multiple lines of code, at the end of each line, press Enter after the **;** to continue entering the next line. After you have typed in all lines, select them all and then press Enter to execute.

Let's start with a new project using the **doc** command and the **-n** switch. If the current project needs saving, this command will prompt user to save.

```
doc -n
```

Now let's import a data file from the Samples folder. We will first use the **dlgfile** X-function to locate the desired file:

```
dlgfile gr:=ASCII
```

Then select the file **S15-125-03.dat** from the **\Samples\Import and Export** sub folder located in your Origin installation folder, and click **Open**.

The above process will load the file path and name into a variable named **fname$**. You can examine the value of this variable by typing:

```
fname$=
```

Now let's import this files into the active workbook. We will use the **impasc** X-Function with options to control naming, so that the file name does not get assigned to the workbook:

```
impasc Options.Names.FNameToBk:=0
```

Now we want to perform some data processing of the **Position** column. We first define a range variable to point to this column.:

```
range rpos = "Position"
```

Since the column we select is also the 4th column in the current worksheet, we can also use index number to specify it.

```
range rpos = 4
```

You can check what range variables are currently defined, using this command:

```
list a
```

We now normalize the column so that the values go from 0 to 100. To check what X-Functions are available for normalization, we can use the command:

```
lx *norm*
```

The above command will dump X-Functions where the name contains **norm**. There are several X-Functions for normalizing data. For our current purpose we will use the **rnormalize** X-Function.

To get help on the syntax for this particular X-Function, you can type:

```
rnormalize -h
```

to dump the information, or type:

```
help rnormalize
```

to open the help file.

Let us now normalize the position column:

```
rnormalize irng:=rpos method:=range100 orng:=<input>
```

The normalized data will be placed in the same column, replacing the original data, as we set the output range variable **orng** to be **<input>**.

When using X-Functions, you can leave out the variable names, if they are specified in the correct order. The above line of code can therefore be written as:

```
rnormalize rpos range100 orng:=<input>
```

The reason we still specified the name **orng** is because there are other variables that precede this particular variable, which are not relevant to our current calculation and were therefore not included in the command.

Now let's do some changes to the folder in the project. There are several X-Functions for managing project folders, and we will use some of them:

```
// Get the name of the current worksheet
string name$ = wks.name$;
// go to root folder
pe_cd ..;
// rename Folder1 to be the same as worksheet
pe_rename Folder1 name$;
```

Now let's list all the sub folders and workbooks under the root folder:

```
pe_dir
```

Finally, let's save the Origin Project to the User Files Folder. The location of the user files folder is stored in the string register **%Y**. You can examine where your User Files Folder is by checking this variable:

```
%Y =
```

Now let's use the **save** command to save our project to User Files Folder, with the name MyProject.opj.

```
save %yMyProject
```

In the above command **%Y** will be replaced with the User Files Folder path, and thus our project will be saved in the correct location.

2.5 Where to Go from Here?

The answer to this question is the subject of the rest of the LabTalk Scripting Guide. The examples above only scratch the surface, but have hopefully provided enough information for you to get a quick start and excited to learn more. The next chapter lists various resources available for learning LabTalk.

3 Resources for Learning LabTalk

Additional resources are available for learning LabTalk.

3.1 Online Documentation

Most up-to-date documentation for LabTalk, including updates to this guide, can be found online at this location: http://www.originlab.com/doc/labtalk

3.2 Script Examples

Various Script Examples are shipped with Origin. These are accessible from the **Help** menu, and are contained in the sub folder: **<Origin Installation Folder>\Samples\LabTalk Script Examples**.

3.3 X-Function Script Examples

Press the **F11** key in Origin to open the **XF Script Dialog**. This dialog provides many script examples specific to calling X-Functions, organized in various categories such as Import, Fitting, Signal Processing, and Spectroscopy.

3.4 LabTalk Forum

Post your question on the LabTalk forum. Go to: http://www.originlab.com/forum and then select the **LabTalk Forum**. Our forums are monitored by our technical staff, plus you may get ideas and answers from other power users as well.

3.5 Training and Consulting

OriginLab and our distributors worldwide offer Training and Consulting services to help you with advanced customization using LabTalk. Please contact us for further details.

4 Language Fundamentals

In this chapter, we introduce various aspects of the LabTalk language structure. In the first section you will learn about general language features such as data types, variables, operators, conditional and loop structures, macros and functions. The second section covers features that are unique to LabTalk, such as range and substitution notation, objects, methods and properties, and accessing X-Functions.

4.1 General Language Features

These pages contain information on implementing general features of the LabTalk scripting language. You will find these types of features in almost every programming language.

4.1.1 Data Types and Variables

LabTalk Data Types

LabTalk supports 9 data types:

Type	Comment
Double	Double-precision floating-point number
Integer	Integers
Constant	Numeric data type that value cannot be changed once declared
Dataset	Array of numeric values
String	Sequences of characters
StringArray	Array of strings
Range	Refers to a specific region of Origin object (workbook, worksheet, etc.)
Tree	Emulates data with a set of branches and leaves

Graphic Object	Objects like labels, arrows, lines, and other user-created graphic elements

Numeric

LabTalk supports three numeric data types: **double**, **int**, and **const**.

1. Double: double-precision floating-point number; this is the default variable type in Origin.
2. Integer: integers (**int**) are stored as double in LabTalk; truncation is performed during assignment.
3. Constant: constants (**const**) are a third numeric data type in LabTalk. Once declared, the value of a **constant** cannot be changed.

```
// Declare a new variable of type double:
double dd = 4.5678;
// Declare a new integer variable:
int vv = 10;
// Declare a new constant:
const em = 0.5772157;
```

Note: LabTalk does not have a **complex** datatype. You can use a complex number in a LabTalk expression only in the case where you are adding or subtracting. LabTalk will simply ignore the imaginary part and return only the real part. (The real part would be wrong in the case of multiplication or division.) Use **Origin C** if you need the **complex** datatype. Columns in Workbooks can be defined as Numeric with DataType of complex, in which case +, -, *, / all work as expected.

```
// Only valid for addition or subtraction:
realresult = (3-13i) - (7+2i);
realresult=;
// realresult = -4
```

Dataset

The Dataset data type is designed to hold an array of numeric values.

Temporary Loose Dataset

When you declare a **dataset** variable it is stored internally as a local, temporary loose dataset. Temporary means it will not be saved with the Origin project; loose means it is not affiliated with a particular worksheet. Temporary loose datasets are used for computation only, and cannot be used for plotting.

The following brief example demonstrates the use of this data type (Dataset Method and $ Substitution Notation are used in this example):

```
// Declare a dataset 'aa' with values from 1-10,
// with an increment of 0.2:
dataset aa={1:0.2:10};
```

```
// Declare integer 'nSize',
// and assign to it the length of the new array:
int nSize = aa.GetSize();

// Output the number of values in 'aa' to the Script Window:
type "aa has $(nSize) values";
```

Project Level Loose Dataset

When you create a dataset by vector assignment (without declaration) or by using the Create (Command) it becomes a project level loose dataset, which can be used for computation or plotting.

Create a project-level loose dataset by assignment,

```
bb = {10:2:100}
```

Or by using the **Create command**:

```
create %(strWks$) -wdn 10 aa bb;
```

For more on project-level and local-level variables see the section below on Scope of Variables.

For more on working with Datasets, see Datasets.

For more on working with %( ), see Substitution Notation.

String

LabTalk supports string handling in two ways: string variables and string registers.

String Variables

String variables may be created by declaration and assignment or by assignment alone (depending on the desired variable scope), and are denoted in LabTalk by a name comprised of continuous characters (see Naming Rules below) followed by a **$**-sign (i.e., *stringName$*):

```
// Create a string with local/session scope by declaration and assignment

// Creates a string named "greeting",
// and assigns to it the value "Hello":
string greeting$ = "Hello";

// $ termination is optional in declaration, mandatory for assignment
string FirstName, LastName;
FirstName$ = Isaac;
LastName$ = Newton;

// Create a project string by assignment without declaration:
greeting2$ = "World";//global scope and saved with OPJ

// string variable can make use of string class methods
string str$ = Johann Sebastian Bach;
str.Find('Sebastian')=;
```

For more information on working with string variables, see the String Processing section.

String Registers

Strings may be stored in String registers, denoted by a leading %-sign followed by a letter of the alphabet (i.e., %A-%Z). String Registers are always global in scope.

```
/* Assign to the string register %A the string "Hello World": */
%A = "Hello World";
```

For current versions of Origin, we encourage the use of string variables for working with strings, as they are supported by several useful built-in methods; for more, see String(Object). If, however, you are already using string registers, see String Registers for complete documentation on their use.

StringArray

The StringArray data type handles arrays of strings in the same way that the Datasets data type handles arrays of numbers. Like the String data type, StringArray is supported by several built-in methods; for more, see StringArray (Object).

The following example demonstrates the use of StringArray:

```
// Declare string array named "aa",
// and use built-in methods Add, and GetSize:
StringArray aa;             // aa is an empty string array
aa.Add("Boston");           // aa now has one element: "Boston"
aa.Add("New York");         // aa has a second element: "New York"

/* Prints "aa has 2 strings in it:" then each string. */
type "aa has $(aa.GetSize()) strings in it:";
loop(ii,1,aa.GetSize())
{
   ty aa.GetAt(ii)$;
}
```

Range

The range data type allows functional access to many data-related Origin objects, referring to a specific region in a workbook, worksheet, graph, layer, or window.

The general syntax is:

range *rangeName* = [*WindowName*]*LayerNameOrIndex*!*DataRange[subRange]*

which can be made specific to data in a workbook, matrix, or graph:

range *rangeName* =
[*BookName*]*SheetNameOrIndex*!*ColumnNameOrIndex*[*RowBegin*:*RowEnd*]

range *rangeName* = [*MatrixBookName*]*MatrixSheetNameOrIndex*!*MatrixObjectNameOrIndex[CellBegin:CellEnd]*

range *rangeName* =[*GraphName*]*LayerNameOrIndex*!*DataPlotIndex[RowBegin:RowEnd]*

The special syntax **[*??*]** is used to create a range variable to access a loose dataset.

For example:

```
// Access Column 3 on Book1, Sheet2:
range cc = [Book1]Sheet2!Col(3);
// Access second curve on Graph1, layer1:
range ll = [Graph1]Layer1!2;
// Access second matrix object on MBook1, MSheet1:
range mm = [MBook1]MSheet1!2;
// Access loose dataset tmpdata_a:
range xx = [??]!tmpdata_a;
```

Notes:

- CellRange can be a single cell, (part of) a row or column, a group of cells, or a non-contiguous selection of cells.
- Worksheets, Matrix Sheets, and Graph Layers can each be referenced by name or index.
- You can define a range variable to represent an origin object, or use range directly as an X-Function argument.
- Many more details on the **range data type** and uses of **range variables** can be found in the Range Notation.

Tree

LabTalk supports the standard tree data type, which emulates a tree structure with a set of branches and leaves. The branches contain leaves, and the leaves contain data. Both branches and leaves are called nodes.

Leaf: A node that has no children, so it can contain a value

Branch: A node that has child nodes and does not contain a value

A leaf node may contain a variable that is of **numeric**, **string**, or **dataset** (vector) type.

Trees are commonly used in Origin to set and store parameters. For example, when a dataset is imported into the Origin workspace, a tree called **options** holds the parameters which determine how the import is performed.

Specifically, the following commands import ASCII data from a file called "SampleData.dat", and set values in the **options tree** to control the way the import is handled. Setting the **ImpMode** leaf to a value of 4 tells Origin to import the data to a new worksheet. Setting the NumCols leaf (on the Cols branch) to a value of 3 tells Origin to only import the first three columns of the *SampleData.dat* file.

```
string str$ = system.path.program$ + "Samples\Graphing\Group.dat";
impasc fname:=str$
/* Start with new sheet */
```

```
options.ImpMode:=4
/* Only import the first three columns */
options.Cols.NumCols:=3;
```

Declare a tree variable named **aa**:

```
// Declare an empty tree
tree aa;
// Tree nodes are added automatically during assignment:
aa.bb.cc=1;
aa.bb.dd$="some string";

// Declare a new tree 'trb' and assign to it data from tree 'aa':
tree trb = aa;
```

The tree data type is often used in X-Functions as both an input and output data structure. For example:

```
// Put import file info into 'trInfo'.
impinfo t:=trInfo;
```

Tree nodes can be strings. The following example shows how to copy a treenode with string data to worksheet columns:

```
//Import the data file into worksheet
newbook;
string fn$=system.path.program$ + "\samples\statistics\automobile.dat";
impasc fname:=fn$;
tree tr;
//Perform statistics on a column and save results to a tree variable
discfreqs irng:=2 rd:=tr;
// Assign strings to worksheet column.
newsheet name:=Result;
col(1) = tr.freqcount1.data1;
col(2) = tr.freqcount1.count1;
```

Tree nodes can also be vectors. Prior to **Origin 8.1 SR1** the only way to access a vector in a Tree variable was to make a direct assignment, as shown in the example code below:

```
tree tr;
// If you assign a dataset to a tree node,
// it will be a vector node automatically:
tr.a=data(1,10);
// A vector treenode can be assigned to a column:
col(1)=tr.a;
// A vector treenode can be assigned to a loose dataset, which is
// convenient since a tree node cannot be used for direct calculations
dataset temp=tr.a;
// Perform calculation on the loose dataset:
col(2)=temp*2;
```

You can access elements of a vector tree node directly, with statements such as:

```
// Following the example immediately above,
col(3)[1] = tr.a[3];
```

that assigns the third element of vector **tr.a** to the first row of column 3 in the current worksheet.

You can also output analysis results to a tree variable, like the example below.

```
newbook;
//Import the data file into worksheet
string fn$=system.path.program$ + "\samples\Signal Processing\fftfilter1.dat";
impasc fname:=fn$;
tree mytr;
//Perform FFT and save results to a tree variable
fft1 ix:=col(2) rd:=mytr;
page.active=1;
col(3) = mytr.fft.real;
col(4) = mytr.fft.imag;
```

More information on trees can be found in the chapter on Origin Projects,**Accessing Metadata** section.

Graphic Objects

The new LabTalk variable type GObject allows the control of graphic objects in any book/layer.

The general syntax is:

GObject *name* = [*GraphPageName*]*LayerIndex*!*ObjectName*;

GObject *name* = [*GraphPageName*]*LayerName*!*ObjectName*;

GObject *name* = *LayerName*!*ObjectName*; // active graph

GObject *name* = *LayerIndex*!*ObjectName*; // active graph

GObject *name* = *ObjectName*; // active layer

You can declare GObject variables for both existing objects as well as for not-yet created objects.

For example:

```
GObject myLine = line1;
draw -n myLine -l {1,2,3,4};
win -t plot;
myLine.X+=2;
/* Even though myLine is in a different graph
that is not active, you can still control it! */
```

For a full description of Graphic Objects and their properties and methods, please see Graphic Objects.

Variables

A variable is simply an instance of a particular data type. Every variable has a name, or identifier, which is used to assign data to it, or access data from it. The assignment operator is

the equal sign (=), and it is used to simultaneously create a variable (if it does not already exist) and assign a value to it.

Variable Naming Rules

Variable, dataset, command, and macro names are referred to generally as identifiers. When assigning identifiers in LabTalk:

- Use any combination of letters and numbers, but note that:
  - the identifier cannot be more than 25 characters in length.
  - the first character cannot be a number.
  - the underscore character "_" has a special meaning in dataset names and should be avoided.
- Use the **Exist** (Function) to check if an identifier is being used to name a window, macro, tool, dataset, or variable.
- Note that several common identifiers are reserved for system use by Origin, please see System Variables for a complete list.

Handling Variable Name Conflicts

The **@ppv** system variable controls how Origin handles naming conflicts between project, session, and local variables. Like all system variables, **@ppv** can be changed from script anytime and takes immediate effect.

Variable	Description
@ppv=0	This is the DEFAULT option and allows both session variables and local variables to use existing project variable names. In the event of a conflict, session or local variables are used.
@ppv=1	This option makes declaring a session variable with the same name as an existing project variable illegal. Upon loading a new project, session variables with a name conflict will be disabled until the project is closed or the project variable with the same name is deleted.
@ppv=2	This option makes declaring a local variable with the same name as an existing project variable illegal. Upon loading of new project, local variables with a name conflict will be disabled until the project is closed or the project variable with the same name is deleted.
@ppv=3	This is the combination of @ppv=1 and @ppv=2. In this case, all session and local variables will not be allowed to use project variable names. If a new project is loaded, existing session or local variables of the same name will be disabled.

Listing and Deleting Variables

Use the LabTalk commands **list** and **del** for listing variables and deleting variables, respectively.

```
/* Use the LabTalk command "list" with various options to list
variables; the list will print in the Script Window by default: */

list a;       // List all the session variables
list v;       // List all project and session variables
list vs;      // List all project and session string variables
list vt;      // List all project and session tree variables

// Use the LabTalk command "del" to delete variables:

del -al <variableName>;  // Delete specific local or session variable
del -al *;               // Delete all the local and session variables

// There is also a viewer for LabTalk variables:
// "ed" command can also open the viewer
list;                     // Open the LabTalk Variables Viewer
```

Please see the **List** (Command), and **Del** (Command) (in Language Reference: Command Reference) for all listing and deleting options.

If no options are specified, running either the List or Edit command will open the LabTalk Variables and Functions dialog and list all variables and functions.

Scope of Variables

The scope of a variable determines which portions of the Origin project can see and be seen by that variable. With the exception of the **string**, **double** (numeric), and **dataset** data types, LabTalk variables must be declared. The way a variable is declared determines its scope. Variables created without declaration (**double**, **string**, and **dataset** only!) are assigned the Project/Global scope. Declared variables are given Local or Session scope. Scope in LabTalk consists of three (nested) levels of visibility:

- Project variables
- Session variables
- Local variables

Project (Global) Variables

- Project variables, also called Global variables, are saved with the Origin Project (*.OPJ). Project variables or Global variables are said to have Project scope or Global scope.

- Project variables are automatically created without declarations for variables of type **double**, **string**, and **dataset** as in:

```
// Define a project (global scope) variable of type double:
myvar = 3.5;
// Define a loose dataset (global scope):
temp = {1,2,3,4,5};
// Define a project (global scope) variable of type string:
str$ = "Hello";
```

- All other variable types must be declared, which makes their default scope either Session or Local. For these you can force Global scope using the @global system variable (below).

Session Variables

- Session variables are not saved with the Origin Project, and are available in the current Origin session across projects. Thus, once a session variable has been defined, they exist until the Origin application is terminated or the variable is deleted.
- When both a session variable and project variable share the same name, the session variable takes precedence.
- Session variables are defined with variable declarations, such as:

```
// Defines a variable of type double:
double var1 = 4.5;
// Define loose dataset:
dataset mytemp = {1,2,3,4,5};
```

It is possible to have a project variable and session variable share the same name. In such a case, the session variable takes precedence. See the script example below:

```
aa = 10;
type "First, aa is a project variable equal to $(aa)";
double aa = 20;
type "Then aa is a session variable equal to $(aa)";
del -al aa;
type "Now aa is project variable equal to $(aa)";
```

And the output is:

```
First, aa is a project variable equal to 10
Then aa is a session variable equal to 20
Now aa is project variable equal to 10
```

Local Variables

Local variables exist only within the current scope of a particular script.

Script-level scope exists for scripts:

- enclosed in curly braces {},
- in separate *.OGS files or individual sections of *.OGS files,
- inside the Column/Matrix Values Dialog, or

- behind a custom button (Button Script).

Local variables are declared and assigned values in the same way as session variables:

```
loop(i,1,10){
     double a = 3.5;
     const e = 2.718;
     // some other lines of script...
}
// "a" and "e" exist only inside the code enclosed by {}
```

It is possible to have local variables with the same name as session variables or project variables. In this case, the local variable takes precedence over the session or project variable of the same name, within the scope of the script. For example, if you run the following script (Please refer to Run LabTalk Script From Files for details on how to run such script):

```
[Main]
double aa = 10;
type "In the main section, aa equals $(aa)";
run.section(, section1);
run.section(, section2);

[section1]
double aa = 20;
type "In section1, aa equals $(aa)";

[section2]
type "In Section 2, aa equals $(aa)";
```

Origin will output:

```
In the main section, aa equals 10
In section1, aa equals 20
In Section 2, aa equals 10
```

Forcing Global Scope

At times you may want to define variables or functions in a *.OGS file, but then be able to use them from the Script Window (they would, by default, exist only while the *.OGS file was being run). To do so, you need to use the **@global** system variable, which when given a value of **1**, forces all variables to have global or project level scope (its default value is **0**). For Example:

```
[Main]
@global = 1;
// the following declarations become global
range a = 1, b= 2;
if(a[2] > 0)
{
  // begin a local scope
  range c = 3; // this declaration is still global
}
```

Upon exiting the *.OGS, the **@global** variable is automatically restored to its default value, **0**.

Note that one can also control a block of code by placing @global at the beginning and end such as:

```
@global=1;
double alpha=1.2;
double beta=2.3;
Function double myPeak(double x, double x0)
{
   double y = 10*exp(-(x-x0)^2/4);
   return y;
}
@global=0;
double gamma=3.45;
```

In the above case variables alpha, beta, and the user-defined function myPeak will have global scope, where as the variable gamma will not.

4.1.2 Programming Syntax

Programming Syntax

A LabTalk script is a single block of code that is received by the LabTalk interpreter. A LabTalk script is composed of one or more complete programming statements, each of which performs an action.

Each statement in a script should end with a semicolon, which separates it from other statements. However, single statements typed into the Script window for execution should not end with a semicolon.

Each statement in a script is composed of words. Words are any group of text separated by white space. Text enclosed in parentheses is treated as a single word, regardless of white space. For example:

```
type This is a statement;                  // Single LabTalk statement

ty s1; ty s2; ty s3;                      // Three statements
```

Parentheses are used to create long words containing white space. For example, in the script:

```
menu 3 (Long Menu Name);
```

the open parenthesis signifies the beginning of a single word, and the close parenthesis signifies the end of the word.

Statement Types

LabTalk supports five types of statements:

- Assignment Statements
- Macro Statements

- Command Statements
- Arithmetic Statement
- Function Statements

Assignment Statements

The assignment statement takes the general form:

LHS* = *expression ;

expression (RHS, right-hand side) is evaluated and put into *LHS* (left-hand side). If *LHS* does not exist, it is created if possible, otherwise an error will be reported.

When a new data object is created with an assignment statement, the object created is:

- A string variable if *LHS* ends with a **$** as in *stringVar*$ = "Hello."
- A numeric variable if *expression* evaluates to a scalar.
- A dataset if *expression* evaluates to a range.

When new values are assigned to an existing data object, the following conventions apply:

- If *LHS* is a dataset and *expression* is a scalar, every value in *LHS* is set equal to *expression*.
- If *LHS* is a numeric variable, then *expression* must evaluate into a scalar. If *expression* evaludate into a dataset, *LHS* retrieves the first element of the dataset.
- If both *LHS* and *expression* represent datasets, each value in *LHS* is set equal to the corresponding value in *expression*.
- If *LHS* is a string, then *expression* is assumed to be a string expression.
- If the *LHS* is the **object.property** notation, with or without $ at the end, then this notation is used to **set** object properties, such as the number of columns in a worksheet, like wks.ncols=3;

Examples of Assignment Statements

Assign the variable **B** equal to 2.

```
B = 2;
```

Assign **Test** equal to B raised to the third power.

```
Test = B^3;
```

Assign **%A** equal to Austin TX.

```
%A = Austin TX;
```

Assign every value in **Book1_B** to 4.

```
Book1_B = 4;
```

Assign each value in **Book2_B** to the corresponding position in **Book1_B.**

```
Book1_B = Book2_B;
```

Sets the row heading width for the **Book1** worksheet to 100, using the worksheet object's **rhw** property. The **doc -uw** command refreshes the window.

```
Book1!wks.rhw = 100;  doc -uw;
```

The calculation is carried out for the values at the corresponding index numbers in **more** and **yetmore**. The result is put into **myData** at the same index number.

```
myData = 3 * more + yetmore;
```

Note: If a string register to the left of the assignment operator is enclosed in parentheses, the string register is substitution processed before assignment. For example:

```
%B = DataSet;
(%B) = 2 * %B;
```

The values in DataSet are multiplied by 2 and put back into DataSet. **%B** still holds the string "DataSet".

Similar to string registers, the assignment statement is also used for string variables, like:

```
fname$=fdlg.path$+"test.csv";
```

In this case, the *expression* is a string expression which can be string literals, string variables, or a concatenation of multiple strings with the **+** character.

Macro Statements

Macros provide a way to alias a script, that is, to associate a given script with a specific name. This name can then be used as a command that invokes the script.

For more information on macros, see Macros

Command Statements

The third statement type is the command statement. LabTalk offers commands to control or modify most program functions.

Each command statement begins with the command itself, which is a unique identifier that can be abbreviated to as little as two letters (as long as the abbreviation remains unique, which is true in most cases). Most commands can take options (also known as switches), which are single letters that modify the operation of the command. Options are always preceded by the dash "-" character. Commands can also take arguments. Arguments are either a script or a data object. In many cases, options can also take their own arguments.

Command statements take the general form:

```
command [option] [argument(s)];
```

The brackets [] indicate that the enclosed component is optional; not all commands take both options and arguments. The brackets are not typed with the command statement (they merely denote an optional component).

Methods (Object) are another form of command statement. They execute immediate actions relating to the named object. Object method statements use the following syntax:

```
ObjectName.Method([options]);
```

For example:

The following script adds a column named new to the active worksheet and refreshes the window:

```
wks.addcol(new);
doc -uw;
```

The following examples illustrate different forms of command statements:

Integrate the dataset myData from zero.

```
integ myData;
```

Adding the -r option and its argument, baseline, causes myData to be integrated from a reference curve named baseline.

```
integ -r baseline myData;
```

The repeat command takes two arguments to execute:

1. the number of times to execute, and
2. a script, which indicates the instruction to repeat.

This command statement prints "Hello World" in a dialog box three times.

```
repeat 3 {type -b "Hello World"}
```

Arithmetic Statement

The arithmetic statement takes the general form:

```
dataObject1 operator dataObject2;
```

where

- *dataObject1* is a dataset or a numeric variable.
- *dataObject2* is a dataset, variable, or a constant.
- *operator* can be +, -, *, /, or ^.

The result of the calculation is put into *dataObject1*. Note that *dataObject1* cannot be a function. For example, **col(3) + 25** is an illegal usage of this statement form.

The following examples illustrate different forms of arithmetic statements:

If **myData** is a dataset, this divides each value in **myData** by 10.

```
myData / 10;
```

Subtract **otherData** from **myData**, and put the result into **myData**. Both datasets must be Y or Z datasets (see **Note**).

```
myData - otherData;
```

If A is a variable, increment A by 1. If A is a dataset, increment each value in A by 1.

```
A + 1;
```

Note: There is a difference between using datasets in arithmetic statements versus using datasets in assignment statements. For example, **data1_b + data2_b** is computed quite differently from **data1_b = data1_b + data2_b**. The latter case yields the true point-by-point sum without regard to the two datasets' respective X-values. The former statement, **data1_b + data2_b**, adds the two data sets as if each were a curve in the XY-plane. If therefore, **data1_b** and **data2_b** have different associated X-values, one of the two series will require interpolation. In this event, Origin interpolates based on the first dataset's (**data1_b** in this case) X-values.

Function Statements

The function statement begins with the characteristics of a function -- an identifier -- followed by a quantity, enclosed by parentheses, upon which the function acts.

An example of a function statement is:

```
sum(dataset);
```

For more on functions in LabTalk, see Functions.

Using Semicolons in LabTalk

Separate Statements with a Semicolon

Like the C programming language, LabTalk uses semicolons to separate statements. In general, every statement should end with a semicolon. However, the following rules clarify semicolon usage:

- Do not use a semicolon when executing a single statement script in the Script window.
    - An example of the proper syntax is: **`type "hello"`** (ENTER).
    - The interpreter automatically places a semicolon after the statement to indicate that it has been executed.
- Statements ending with { } block can skip the semicolon.
- The last statement in a { } block can also skip the semicolon.

In the following example, please note the differences between the three type command:

```
if (m>2) {type "hello";} else {type "goodbye"}
type "the end";
```

The above can also be written as:

```
if (m>2) {type "hello"} else {type "goodbye"}
type "the end";
```

or

```
if (m>2) {type "hello"} else {type "goodbye"};
type "the end";
```

Leading Semicolon for Delayed Execution

You can place a ';' in front of a script to delay its execution. This is often needed when you need to run a script inside a button that will delete the button itself, like to issue window closing or new project commands. For example, placing the following script inside a button will possibly lead to a crash

```
// button to close this window
type "closing this window";
win -cn %H;
```

To fix this, the script should be written as

```
// button to close this window
type "closing this window";
;win -cn %H;
```

The leading ';' will place all scripts following it to be delayed when executed. Sometimes you may want a specific group of statements delayed, then you can put them inside {script} with a leading ';', for example:

```
// button to close this window
```

```
type "closing this window";
;{type "from delayed execution";win -cn %H;}
type "actual window closing code will be executed after this";
```

Extending a Statement over Multiple Lines

There are times when, for the sake of readability, you want to extend a single statement over more than one line. One way to do this is with braces {}. When an "open brace", **{**, is encountered in a script file, Origin searches for a "closed brace" , **}**, and executes the entire block of text as one statement. For example, the following macro statement:

```
def openDialog {layer -s 1;  axis x;};
```

can also be written:

```
def openDialog {
                  layer -s 1;
                  axis x;
                };
```

Both scripts are executed as a single statement, even though the second statement spans multiple lines.

Note: There is a limit to the length of script that can be included between a set of braces {}. The scripts between the {} are translated internally and the translated scripts must be less than 1140 bytes (after substitution). In place of long blocks of LabTalk code, programmers can use LabTalk macros or the run.section() and run.file() object methods. To learn more, see Passing Arguments.

Comments

LabTalk script accepts two comment formats:

Use the "//" character to ignore all text from **//** to the end of the line. For example:

```
type "Hello World";  //Place comment text here.
```

Use the combination of "/*" and "*/" character pairs to begin and end, respectively, any block of code or text that you do not want executed. For example:

```
type Hello /* Place comment text here,
              or a line of code:
              and even more ... */
World;
```

Note: Use the "**#**!" characters to begin debugging lines of script. The lines are only executed if `system.debug = 1`.

Order of Evaluation in Statements

When a script is executed, it is sent to the LabTalk interpreter and evaluated as follows:

The script is broken down into its component statements

Statements are identified by type using the following recognition order: assignment, macro, command, arithmetic, and function. The interpreter first looks for an exposed (not hidden in

parentheses or quotation marks) assignment operator. If none is found, it looks to see if the first word is a macro name. It then checks if the first word is a command name. The interpreter then looks for an arithmetic operation, and finally, the interpreter checks whether the statement is a function.

The recognition order can have significant effect on script functions. For example, the following assignment statement:

```
type = 1;
```

assigns the value 1 to the variable type. This occurs even though type is (also) a LabTalk command, since assignments come before commands in recognition order. However, since commands precede arithmetic expressions in recognition order, in the following statement:

```
type + 1;
```

the command is carried out first, and the string, + 1, prints out.

The statements are executed in the order received, using the following evaluation priority

- Assignment statements: String variables to the left of the assignment operator are not expressed unless enclosed by parentheses. Otherwise, all string variables are expressed, and all special notation (**%()** and **$()**) is substitution processed.
- Macro statements: Macro arguments are substitution processed and passed.
- Command statements: If a command is a raw string, it is not sent to the substitution processor. Otherwise, all special notation is substitution processed.
- Arithmetic statements: All expressions are substitution processed and expressed.

4.1.3 Operators

Introduction

LabTalk supports assignment, arithmetic, logical, relational, and conditional operators:

Arithmetic Operators	+ - * / ^ & |
String Concatenation	+
Assignment Operators	= += -= *= /= ^=
Logical and Relational Operators	> >= < <= == != && ||
Conditional Operator	? :

These operations can be performed on scalars and in many cases they can also be performed on vectors (datasets). Origin also provides a variety of built-in numeric, trigonometric, and statistical functions which can act on datasets.

When evaluating an expression, Origin observes the following precedence rules:

1. Exposed assignment operators (not within brackets) are evaluated.
2. Operations within brackets are evaluated before those outside brackets.
3. Multiplication and division are performed before addition and subtraction.
4. The (>, >=, <, <=) relational operators are evaluated, then the (== and !=) operators.
5. The logical operators || is prior to &&.
6. Conditional expressions (?:) are evaluated.

Arithmetic Operators

Origin recognizes the following arithmetic operators:

Operator	Use
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiate (X^Y raises X to the Yth power) (see note below)
&	Bitwise And operator. Acts on the binary bits of a number.
|	Bitwise Or operator. Acts on the binary bits of a number.

Note: For 0 raised to the power n (0^n), if n > 0, 0 is returned. If n < 0, a missing value is returned. If n = 0, then 1 is returned (if @ZZ = 1) or a missing value is returned (if @ZZ = 0).

These operations can be performed on scalars and on vectors (datasets). For more information on scalar and vector calculations, see Performing Calculations below.

The following example illustrates the use of the exponentiate operator: Enter the following script in the Command window:

```
1.3 ^ 4.7 =
```

After pressing ENTER, 3.43189 is printed in the Command window. The next example illustrates the use of the **bitwise and** operator. Enter the following script in the Command window:

```
if (27&41 == 9)
{type "Yes!"}
```

After pressing ENTER, **Yes!** is displayed in the Command window.

Note: 27&41 == 9 because

```
27 = 0000000000011011
```

```
41 = 0000000000101001
```

with bitwise & yields:

```
0000000000001001 (which is equal to 9)
```

Note: Multiplication must be explicitly included in an expression. For example, 2*X must be used instead of 2X to indicate the multiplication of the variable X by the constant 2.

Define a constant

We can also define global constants in the ORGSYS.CNF file:

const pi = 3.141592653589793

A Note about Logarithmic Conversion

- To convert a dataset to a logarithmic scale, use the following syntax:

```
col(c) = log(col(c));
```

- To convert a dataset back to a linear scale, use the following syntax:

```
col(c) = 10^(col(c));
```

String Concatenation

Very often you need to concatenate two or more strings of either the string variable or string register type. All of the code segments in this section return the string "Hello World."

The string concatenation operator is the plus-sign (+), and can be used to concatenate two strings:

```
aa$ ="Hello";
bb$="World";
cc$=aa$+" "+bb$;
cc$=;
```

To concatenate two string registers, you can simply place them together:

```
%J="Hello";
%k="World";
%L=%J %k;
%L=;
```

If you need to work with both a string variable and a string register, follow these examples utilizing %( ) substitution:

```
aa$ ="Hello";
%K="World";
dd$=%(aa$) %K;
dd$=;
dd$=%K;
dd$=aa$+" "+dd$;
dd$=;
%M=%(aa$) %K;
%M=;
```

Assignment Operators

Origin recognizes the following assignment operators:

Operator	Use
=	Simple assignment.
+=	Addition assignment.
-=	Subtraction assignment.
*=	Multiplication assignment.
/=	Division assignment.
^=	Exponential assignment.

These operations can be performed on scalars and on vectors (datasets). For more information on scalar and vector calculations, see Performing Calculations in this topic.

The following example illustrates the use of the **-=** operator.

In this example, 5 is subtracted from the value of A and the result is assigned to A:

```
A -= 5;
```

In the next example, each value in Data1_B is divided by the corresponding value in Book1_A, and the resulting values are assigned to Book1_B.

```
Book1_B /= Book1_A;
```

In addition to these assignment operators, LabTalk also supports the increment and decrement operators for scalar calculations (not vector).

Operator	Use
++	Add 1 to the variable contents and assign to the variable.
--	Subtract 1 from the variable contents and assign to the variable.

The following **for** loop expression illustrates a common use of the increment operator ++. The script prints the data stored in the second column of the current worksheet to the Command window:

```
for (ii = 1; ii <= wks.maxrows; ii++)
   {type ($(col(2)[ii])); }
```

Logical and Relational Operators

Origin recognizes the following logical and relational operators:

Operator	Use
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
&&	And
||	Or

An expression involving logical or relational operators evaluates to either true (non-zero) or false (zero). Logical operators are almost always found in the context of Conditional and Loop Structures.

Numeric Comparison

The most common comparison is between two numeric values. Generally, at least one is a variable. For instance:

```
if aa<3 type "aa<3";
```

Or, both items being compared can be variables:

```
if aa<=bb type "aa<=bb";
```

It is also possible, using parentheses, to make multiple comparisons in the same logical statement:

```
if (aa<3 && aa<bb) type "aa is lower";
```

String Comparison

You can use the == and != operators to compare two strings. String comparison (rather than numeric comparison) is indicated by open and close double quotations (" ") either before, or after, the operator. The following script determines if the %A string is empty:

```
if (%A == ""){type "empty"};
```

The following examples illustrates the use of the == operator:

```
x = 1;     // variable x is set to 1
%a = x;    // string a is set to "x"
if (%a == 1);
      type "yes";
else
```

```
      type "no";
```

The result will be yes, because Origin looks for the value of %a (the value of x), which is 1. In the following script:

```
x = 1;     // variable x is set to 1
%a = x;    // string a is set to "x"
if ("%a" == 1)
      type "yes";
else
      type "no";
```

The result will be no, because Origin finds the quotation marks around %a, and therefore treats it as a string, which has a character x, rather than the value 1.

Conditional Operator (?:)

The ternary operator or conditional operator (?:) can be used in the form:

Expression1* ? *Expression2* : *Expression3

This expression first evaluates Expression1. If Expression1 is true (non-zero), Expression2 is evaluated. The value of Expression2 becomes the value for the conditional expression. If Expression1 is false (zero), then Expression3 is evaluated and Expression3 becomes the value for the entire conditional expression. Note that Expressions1 and Expressions2 can themselves be conditional operators. The following example assigns the value which is greater (m or n), to variable:

```
m = 2;
n = 3;
variable = (m>n?m:n);
variable =
```

LabTalk returns: **variable = 3**

In this example, the script replaces all column A values between 5.5 and 5.9 with 5.6:

```
col(A) = col(A)>5.5&&col(A)<5.9?5.6:col(A);
```

Note: A Threshold Replace function treplace(dataset, value1, value2 [, condition]) is also available for reviewing values in a dataset and replacing them with other values based on a condition. In the treplace(dataset, value1, value2 [, condition]) function, each value in the dataset is compared to value1 according to the condition. When the comparison is true, the value may be replaced with Value2 or -Value2 depending on the value of condition. When the comparison is false, the value is retained or replaced with a missing value depending on the value of condition. The treplace() function is much faster than the ternary operator.

Performing Calculations

You can use LabTalk to perform both

- scalar calculations (mathematical operations on a single variable), and
- vector calculations (mathematical operations on entire datasets).

Scalar Calculations

You can use LabTalk to express a calculation and store the result in a numeric variable. For example, consider the following script:

```
inputVal = 21;
myResult = 4 * 32 * inputVal;
```

The second line of this example performs a calculation and creates the variable, myResult. The value of the calculation is stored in myResult.

When a variable is used as an operand, and will store a result, shorthand notation can be used. For example, the following script:

```
B = B * 3;
```

could also be written:

```
B *= 3;
```

In this example, multiplication is performed with the result assigned to the variable B. Similarly, you can use +=, -=, /=, and ^=. Using shorthand notation produces script that executes faster.

Vector Calculations

In addition to performing calculations and storing the result in a variable (scalar calculation), you can use LabTalk to perform calculations on entire datasets as well.

Vector calculations can be performed in one of two ways: (1) strictly row-by-row, or (2) using linear interpolation.

Row-by-Row Calculations

Vector calculations are always performed row-by-row when you use the two general notations:

datasetB = scalarOrConstant <operator> datasetA;

datasetC = datasetA <operator> datasetB;

This is the case even if the datasets have a different numbers of elements. Suppose there are three empty columns in your worksheet: A, B, and C. Run the following script:

```
col(a) = {1, 2, 3};
col(b) = {4, 5};
col(c) = col(a) + col(b);
```

The result in column C will be {5, 7, --}. That is, Origin outputs a missing value for rows in which one or both datasets do not contain a value.

Vector calculations can also involve a scalar. In the above example, type:

```
col(c) = 2 * col(a);
```

Column A is multiplied by 2 and the results are put into the corresponding rows of column C.

Instead, execute the following script (assuming *newData* does not previously exist):

```
newData = 3 * Book1_A;
```

A temporary dataset called *newData* is created and assigned the result of the vector operation.

Calculations Using Interpolation

Origin supports interpolation through range notation and X-Functions such as interp1 and interp1xy. Please refer to Interpolation for more details.

4.1.4 Conditional and Loop Structures

The structure of the LabTalk language is similar to C. LabTalk supports:

- Loops, which allow the program to repetitively perform a set of actions.
- Decision structures, which allow the program to perform different sets of actions depending on the circumstances.

Loop Structures

All LabTalk loops take a script as an argument. These scripts are executed repetitively under specified circumstances. LabTalk provides four loop commands:

Command	Syntax
repeat	repeat value {*script*};
loop	loop (variable, startVal, endVal) {*script*};
doc -e	doc -e object {*script*};
for	for (expression1; expression2; expression3) {*script*};

The LabTalk for-loop is similar to the for loop in other languages. The repeat, loop, and doc -e loops are less familiar, but are easy to use.

Repeat

The **repeat** loop is used when a set of actions must be repeated without any alterations.
Syntax: **repeat *value* {*script*};**

Execute *script* the number of times specified by *value*, or until an error occurs, or until the *break* command is executed.

For example, the following script types the string three times:

```
repeat 3 { type "line of output"; };
```

Loop

The **loop** loop is used when a single variable is being incremented with each successive loop.
Syntax: **loop (*variable*, *startVal*, *endVal*) {*script*};**

A simple increment loop structure. Initializes *variable* with the value of *starVal*. Executes *script*. Increments *variable* and tests if it is greater than *endVal*. If it is not, executes *script* and continues to loop.

For example, the following script outputs numbers from 1 to 4:

```
loop (ii, 1, 4)  {type "$(ii)";};
```

Note: The **loop** command provides faster looping through a block of script than does the **for** command. The enhanced speed is a result of not having to parse out a LabTalk expression for the condition required to stop the loop.

Doc -e

The doc -e loop is used when a script is being executed to affect objects of a specific type, such as graph windows. The **doc -e** loop tells Origin to execute the script for each instance of the specified object type.

Syntax: **doc -e *object* {*script*};**

The different object types are listed in the document command.

For example, the following script prints the windows title of all graph windows in the project:

```
doc -e P {%H=}
```

For

The **for** loop is used for all other situations.

Syntax: **for (*expression1*; *expression2*; *expression3*) {*script*};**

In the **for** statement, *expression1* is evaluated. This specifies initialization for the loop. Second, *expression2* is evaluated and if true (non-zero), the *script* is executed. Third, **expression3**, often incrementing of a counter, is executed. The process repeats at the second step. The loop terminates when *expression2* is found to be false (zero). Any expression can consist of multiple statements, each separated by a comma.

For example, the following script output numbers from 1 to 4:

```
for(ii=1; ii<=4; ii++)
{
   type "$(ii)";
}
```

Note: The **loop** command provides faster looping through a block of script.

Decision Structures

Decision structures allow the program to perform different sets of actions depending on the circumstances. LabTalk provides three decision-making structures: if, if-else, and switch.

- The **if** command is used when a script should be executed in a particular situation.
- The if-else command is used when one script must be executed if a condition is true (non-zero), while another script is executed if the condition is false (zero).
- The **switch** command is used when more than two possibilities are included in a script.

If, If-Else

Syntax:

1. **if (*testCondition*) *sentence1*; [else *sentence2*;]**

2. **if (*testCondition*) {*script1*} [else {*script2*}]**

Evaluate *testCondition* and if true, execute *script1*. Expressions without conditional operators are considered true if the result of the expression is non-zero.

If the optional **else** is present and *testCondition* is false (zero), then execute *script2*. There should be a space after the else. Strings should be quoted and string comparisons are **not** case sensitive.

Single statement script arguments should end with a semicolon. Multiple statement script arguments must be surrounded by braces {}. Each statement within the braces should end with a semicolon. It is not necessary to follow the final brace of a script with a semicolon.

For example, the following script opens a message box displaying "Yes!":

```
%M = test;
if (%M == "TEST")  type -b "Yes!";
else type -b "No!";
```

The next script finds the first point in column A that is greater than -1.95:

```
newbook;
col(1)=data(-2,2,0.01);
val = -1.95;
get col(A) -e numpoints;
for(ii = 1 ; ii <= numpoints ; ii++)
{
   // This will terminate the loop early if true
   if (Col(A)[ii] > val)  break;
}
if(ii > numpoints - 1)
   ty -b No number exceeds $(val);
else
   type -b The index number of first value > $(val) is $(ii)
The value is $(col(a)[ii]);
```

It is possible to test more than one condition with a single **if** statement, for instance:

```
if(a>1 && a<3) b+=1;  // If true, increment b by 1
```

The && (logical And) operator is one of several logical operators supported in LabTalk.

Switch

The switch command is used when more than two possibilities are included in a script. For example, the following script returns b:

```
ii=2;
switch (ii)
{
     case 1:
         type "a";
         break;
     case 2:
         type "b";
```

```
        break;
    case 3:
        type "c";
        break;
    default:
        type "none";
        break;
}
```

Break and Progress Bars

LabTalk provides a break command. When executed, this causes an exit from the loop and, optionally, the script. This is often used with a decision structure inside a loop. It is used to protect against conditions which would invalidate the loop test conditions. The break command can be used to display a progress status dialog box (progress bar) to show the current progress through the loop.

Exit

The exit command prompts an exit from Origin unless a flag is previously set to prevent the exit.

Continue

The continue command can be used within loops. When executed, the remainder of the loop is ignored and the interpreter jumps to the next iteration of the loop. This is often used with a decision structure inside a loop and can exclude illegal values from being processed by the loop script.

For example, in the following for loop, continue skips the type statement when ii is less than zero.

```
for (ii = -10; ii <= 10; ii += 2)
{
   if (ii < 0)
     continue;

   type "$(sqrt(ii))";
}
```

Sections in a Script File

In addition to entering the script in the Label Control dialog, you can also save it as an Origin Script (OGS) file. An Origin script file is an ASCII text file which consists of a series of one or more LabTalk statements. Often, you can divide the statements into sections. A section is declared by a section name surrounded by square brackets on its own line of text:

[*SectionName*]

Scripts under a section declaration belong to that section until another section declaration is met. A framework for a script with sections will look like the following:

```
...
Scripts;
...
[Section 1]
...
Scripts;
...
[Section 2]
...
Scripts;
...
```

Scripts will be run in sequence until a new section flag is encountered, a return statement is executed or an error occurs. To run a script in sections, you should use the

run.section(*FileName*, *SectionName*)

command. When filename is not included, the current running script file is assumed, for example:

run.section(, Init)

The following script illustrates how to call sections in an OGS file:

```
type "Hello, we will run section 2";
run.section(, section2);

[section1]
type "This is section 1, End the script.";

[section2]
type "This is section 2, run section 1.";
run.section(, section1);
```

To run the script, you can save it to your Origin user folder as test.ogs, and type the following in the command window:

```
run.section(test);
```

If code in a section could cause an error condition which would prematurely terminate a section, you can use a variable to test for that case, as in:

```
[Test]
SectionPassed = 0;
// Here is where code that could fail can be run
...
SectionPassed = 1;
```

If the code failed, then SectionPassed will still have a value of 0. If the code succeeded, then SectionPassed will have a value of 1.

4.1.5 Macros

Definition of the Macros

The command syntax,

```
define macroName {script}
```

defines a macro called *macroName*, and associates it with the given *script*. MacroName can then be used like a command, and invokes the given script.

For example, the following script defines a macro that uses a loop to print a text string three times.

```
def hello
{
      loop (ii, 1, 3)
            { type "$(ii). Hello World"; }
};
```

Once the hello macro is defined, typing the word *hello* in the Script window results in the printout:

```
1. Hello World
2. Hello World
3. Hello World
```

Once a macro is defined, you can also see the script associated with it by typing

```
define macroName;
```

Passing Arguments to Macros

Macros can take up to five arguments. The %1-%5 syntax is used within the macro to access the value of each argument. A macro can accept a number, string, variable, dataset, function, or script as an argument. Passing arguments to a macro is similar to passing arguments to a script.

If arguments are passed to a macro, the macro can report the number of arguments using the macro.nArg object property.

For example, the following script defines a macro named *myDouble* that expects a single numeric argument. The given argument is then multiplied by 2, and the result is printed.

```
def myDouble { type "$(%1 * 2)"; };
```

If you define this macro and then type the following in the Script window:

```
myDouble 5
```

Origin outputs the result to the Script Window:

```
10
```

You could modify this macro to take two arguments:

```
def myDouble { type "$(%1 * %2)"; };
```

Now, if you type the following in the Script window:

```
myDouble 5 4
```

Origin outputs:

```
20
```

Macro Property

The macro object contains one property which stores the number of arguments passed to the macro.

Property	Access	Description
Macro.nArg	Read only, numeric	This property stores the number of arguments passed to the macro.

For example:

The following script defines a macro called *TypeArgs*. If three arguments are passed to the TypeArgs macro, the macro types the three arguments to the Script window.

```
Def TypeArgs
{
      if (macro.narg != 3)
      {
            type "Error! You must pass 3 arguments!";
      }
      else
      {
            type "The first argument passed was %1.";
            type "The second argument passed was %2.";
            type "The third argument passed was %3.";
      }
};
```

If you define the *TypeArgs* macro as in the example, and then type the following in the Script window:

```
TypeArgs  One;
```

Origin returns the following to the Script window:

```
Error! You must pass 3 arguments!
```

If you define the *TypeArgs* macro as in the example, and then type the following in the Script window:

```
TypeArgs  One Two Three;
```

Origin returns the following to the Script window:

```
The first argument passed was One.
The second argument passed was Two.
The third argument passed was Three.
```

4.1.6 Functions

Functions are the core of almost every programming language; the following introduces function syntax and use in LabTalk.

Built-In Functions

LabTalk supports many operations through built-in functions, a listing and description of each can be found in Function Reference. Functions are called with the following syntax:

***outputVariable* = *FunctionName*(Arg1, Arg2, ..., Arg N);**

Below are a few examples of built-in functions in use.

The **Count** (Function) returns an integer count of the number of elements in a vector.

```
// Return the number of elements in Column A of the active worksheet:
int cc = count(col(A));
```

The **Ave** (Function) performs a group average on a dataset, returning the result as a range variable.

```
range ra = [Book1]Sheet1!Col(A);
range rb = [Book1]Sheet1!Col(B);
// Return the group-averaged values:
rb = ave(ra, 5);    // 5 = group size
```

The **Sin** (Function) returns the sin of the input angle as type **double** (the units of the input angle are determined by the value of **system.math.angularunits**):

```
system.math.angularunits=1;  // 1 = input in degrees
double dd = sin(45);         // ANS: DD = 0.7071
```

User-Defined Functions

Support for multi-argument user-defined functions has been supported in LabTalk since Origin 8.1. The syntax for user-defined functions is:

function *dataType funcName*(*Arg1*, *Arg2*, ..., *ArgN*) {*script*;}

Minimum Origin Version Required: 8.6 SR0

Note:

1. The function name should be less than 42 characters.
2. Both arguments and return values support **string, double, int, dataset**, and **tree** data types. The default argument type is **double**. The default return type is **int**.
3. By default, arguments of user-defined functions are passed by value, meaning that argument values inside the function are *NOT* available outside of the function. However, passing arguments by reference, in which changes in argument values inside the function *WILL* be available outside of the function, is possible with the keyword **REF**.

Here are some simple cases of numeric functions:

```
// This function calculates the cube root of a number
```

```
function double dCubeRoot(double dVal)
{
    double xVal;
    if(dVal<0) xVal = -exp(ln(-dVal)/3);
    else xVal = exp(ln(dVal)/3);
    return xVal;
}
// As shown here
dcuberoot(-8)=;
```

The function below calculates the geometric mean of a dataset:

```
function double dGeoMean(dataset ds)
{
    double dG = ds[1];
    for(int ii = 2 ; ii <= ds.GetSize() ; ii++)
        dG *= ds[ii]; // All values in dataset multiplied together
    return exp(ln(dG)/ds.GetSize());
}
// Argument is anything returning a datset
dGeoMean(col("Raw Data"))=;
```

This example defines a function that accepts a range argument and returns the mean of the data in that range:

```
// Calculate the mean of a range
function double dsmean(range ra)
{
   stats ra;
   return stats.mean;
}
// Pass a range that specifies all columns ...
// in the first sheet of the active book:
range rAll = 1!(1:end);
dsMean(rAll)=;
```

This example defines a function that counts the occurrences of a particular weekday in a Date dataset:

```
function int iCountDays(dataset ds, int iDay)
{
    int iCount = 0;
    for(int ii = 1 ; ii <= ds.GetSize() ; ii++)
    {
        if(weekday(ds[ii], 1) == iDay) iCount++;
    }
    return iCount;
}
// Here we count Fridays
iVal = iCountDays(col(1),6); // 6 is Friday in weekday(data, 1) sense
```

```
iVal=;
```

Functions can also return datasets ..

```
// Get only negative values from a dataset
function dataset dsSub(dataset ds1)
{
    dataset ds2;
    int iRow = 1;
    for(int ii = 1 ; ii <= ds1.GetSize() ; ii++)
    {
        if(ds1[ii] < 0)
        {
            ds2[iRow] = ds1[ii];
            iRow++;
        }
    }
    return ds2;
}
// Assign all negative values in column 1 to column 2
col(2) = dsSub(col(1));
```

or strings ..

```
// Get all values in a dataset where a substring occurs
function string strFind(dataset ds, string strVal)
{
    string strTest, strResult;
    for( int ii = 1 ; ii <= ds.GetSize() ; ii++ )
    {
        strTest$ = ds[ii]$;
        if( strTest.Find(strVal$) > 0 )
        {
            strResult$ = %(strResult$)%(CRLF)%(strTest$);
        }
    }
    return strResult$;
}
// Gather all instances in column 3 where "hadron" occurs
string MyResults$ = strFind(col(3),"hadron")$; // Note ending '$'
MyResults$=;
```

Passing Arguments by Reference

This example demonstrates a function that returns a **tree** node value as an int (one element of a tree variable). In addition, passing by reference is illustrated using the **REF** keyword.

```
// Function definition:
Function int GetMinMax(range rr, ref double min, ref double max) {
  stats rr;
```

```
  //after running the stats XF, a LabTalk tree variable with the
  //same name is created/updated
  min = stats.min;
  max = stats.max;
  return stats.N;
}

// Call function GetMinMax to find min max for an entire worksheet:
double y1,y2;
int nn = getminmax(1:end,y1, y2);
type "Worksheet has $(nn) points, min=$(y1), max=$(y2)";
```

A more detailed example using tree variables in LabTalk functions and passing variables by reference, is available in our online Wiki.

Another example of passing string argument by reference is given below that shows that the $ termination should not be used in the function call:

```
//return range string of the 1st sheet
//actual new book shortname will be returned by Name$
Function string GetNewBook(int nSheets, ref string Name$)
{
   newbook sheet:= nSheets result:=Name$;
   string strRange$ = "[%(Name$)]1!";
   return strRange$;
}
```

When calling the above function, it is very important that the Name$ argument should not have the $, as shown below:

```
string strName$;
string strR$ = GetNewBook(1, strName)$;
strName$=;
strR$=;
```

Dataset Functions

Origin also supports defining mathematical functions that accept arguments of type double and return type double. The general syntax for such functions is:

funcName(X) = *expressionInvolvingX*.

We call these dataset functions because when they are defined, a dataset by that name is created. This dataset, associated with the function, is then saved as part of the Origin project. Once defined, a dataset function can be referred to by name and used as you would a built-in LabTalk function.

For example, enter the following script in the Script window to define a function named **Salary**:

```
Salary(x) = 52 * x
```

Once defined, the function may be called anytime as in,

```
Salary(100)=
```

which yields the result **Salary(100)=5200**. In this case, the resulting dataset has only one element. But if a vector (or dataset) were passed as an input argument, the output would be a dataset containing the same number of elements as the input.

As with other datasets, user-defined dataset functions are listed in dialogs such as **Plot Setup** (and can be plotted like any other dataset), and in the **Available Data** list in dialogs such as **Layer n**.

If a 2D graph layer is the active layer when a function is defined, then a dataset of 100 points is created using the X axis scale as the X range and the function dataset is automatically added to the plot layer.

The **Function Graph Template** (FUNCTION.OTP, accessible from the **Standard** Toolbar or the **File: New** menu) also creates and plots dataset functions.

Origin's **Function Plots** feature allows new dataset functions to be easily created from any combination of built-in and user-defined functions. In addition, the newly created function is immediately plotted for your reference.

Access this feature in either of two ways:

1. Click on the **New Function** button in the **Standard** toolbar, 
2. From the Origin drop-down menus, select **File: New** and select **Function** from the list of choices, and click **OK**.

From there, in the **Function** tab of the **Plot Details** dialog that opens, enter the function definition, such as, *F1(x) = 5*sin(x)+1* and press **OK**. The function will be plotted in the graph.

You may define another function by clicking on the **New Function** button in the graph and adding another function in **Plot Details**. Press **OK**, and the new function plot will be added to the graph. Repeat if more functions are desired.

Fitting Functions

In addition to supporting many common functions, Origin also allows you to create your own fitting functions to be used in non-linear curve fitting. User-defined fitting functions can also be used to generate new datasets, but calling them requires a special syntax:

nlf_FitFuncName(ds, *p1*, *p2*, ..., *pn*)

where the fitting function is named **FitFuncName**, **ds** is a dataset to be used as the independent variable, and *p1--pn* are the parameters of the fitting function.

As a simple example, if you defined a simple straight-line fitting function called **MyLine** that expected a y-intercept and slope as input parameters (in that order), and you wanted column C in the active worksheet to be the independent variable (X), and column D to be used for the function output, enter:

```
// Intercept = 0, Slope = 4
Col(D) = nlf_MyLine(Col(C), 0, 4)
```

Scope of Functions

User-defined functions have a scope (like variables) that can be controlled. For more on scope see Data Types and Variables. Please note that there are no project functions or global functions, that is different from scope of variable.

Similar to Session Variables, the scope of a function can be expanded for general use throughout the current Origin session across projects by preceding the function definition with the assignment **@global=1**.

You can associate functions with a project by defining them in the project's ProjectEvents.OGS file, using the @Global=1 to promote them to session level.

To create a user-defined function for use across sessions, add commands in **MACROS.CNF** to run the function definition .OGS files.

User-defined functions can be accessed from anywhere in the Origin project where LabTalk script is supported, provided the scope of definition is applicable to such usage. Thus for example, a function defined with preceding assignment **@global=1** that returns type double or dataset, can be used in the **Set Values** dialog **Column Formula** panel.

With preceding assignment **@global=1**, the function can be called anywhere.

```
[Main]
   @global=1;  // promote the following function to session level
   function double dGeoMean(dataset ds)
   {
       double dG = ds[1];
       for(int ii = 2 ; ii <= ds.GetSize() ; ii++)
           dG *= ds[ii]; // All values in dataset multiplied together
       return exp(ln(dG)/ds.GetSize());
   }
   // can call the function in [main] section
   dGeoMean(col(1))=;
[section1]
   // the function can be called in this section too
   dGeoMean(col(1))=;
```

If the function is defined in a section of a ***.ogs** file without **@global=1**, then it can only be called in its own section.

```
[Main]
   function double dGeoMean(dataset ds)
   {
       double dG = ds[1];
       for(int ii = 2 ; ii <= ds.GetSize() ; ii++)
           dG *= ds[ii]; // All values in dataset multiplied together
       return exp(ln(dG)/ds.GetSize());
   }
   // can call the function in [main] section
```

```
   dGeoMean(col(1))=;
[section1]
   // the function can NOT be called in this section
   dGeoMean(col(1))=;  // an error: Unknown function
```

If the function is defined in a block without **@global=1**, it can not be called outside this block.

```
[Main]
{  // define the function between braces
   function double dGeoMean(dataset ds)
   {
       double dG = ds[1];
       for(int ii = 2 ; ii <= ds.GetSize() ; ii++)
           dG *= ds[ii]; // All values in dataset multiplied together
       return exp(ln(dG)/ds.GetSize());
   }
}
   // can Not call the function outside the braces
   dGeoMean(col(1))=;  // an error: Unknown function
```

Tutorial: Using Multiple Function Features

The following mini tutorial shows how to add a user-defined function at the Origin project level and then use that function to create function plots.

1. Start a new Project and use **View: Code Builder** menu item to open Code Builder.
2. Expand the Project branch on the left panel tree and double-click to open the **ProjectEvents.OGS** file. This file exists by default in any new Project.
3. Under the **[AfterOpenDoc]** section, add the following lines of code:
   @global=1;
   Function double myPeak(double x, double x0)
   {
   double y = 10*exp(-(x-x0)^2/4);
   return y;
   }
4. Save the file and close Code Builder.
5. In Origin, save the Project to a desired folder location. The OGS file is saved with the Project, so the user-defined function is available for use in the Project.
6. Open the just saved project again. This will trigger the **[AfterOpenDoc]** section to be executed and thus our myPeak function to be defined.
7. Click on the **New Function** button in the **Standard** toolbar

8. In the **Function** tab of **Plot Details** dialog that opens, enter the function definition:
   F1(x) = myPeak(x, 3)
   and press OK. The function will be plotted in the graph.
9. Click on the **New Function** button in the graph and add another function in **Plot Details** using the expression:
   F2(x) = myPeak(x, 4)
   and press OK.
10. The second function plot will be added to the graph.
11. Now save the Project again and re-open it. The two function plots will still be available, as they refer to the user-defined function saved with the Project.
12. You can assure yourself that the above really works by first exiting Origin, reopening Origin, and running the project again, checking that the **myPeak** function is defined upon loading the project.

4.2 Special Language Features

These pages contain information on implementing advanced features of the LabTalk scripting language. Some of the concepts and features in this section are unique to Origin.

4.2.1 Range Notation

Introduction to Range

Inside your Origin Project, data exists in four primary places: in the columns of a worksheet, in a matrix, in a loose dataset, or in a graph. In any of these forms, the range data type allows you to access your data easily in a standard way.

Once a range variable is created, you can work with the range data directly; reading and writing to the range. Examples below demonstrate the creation and use of many types of range variables.

Before Origin Version 8.0, data were accessed via datasets as well as cell(), col(), and wcol() functions. The cell(), col(), and wcol() functions are still very effective for data access, provided that you are working with the active sheet in the active book. The Range notation essentially expanded upon these functions to provide general access to any book, sheet, or plot inside an Origin Project.

Note : *Not all X-Functions can handle complexities of ranges such as multiple columns or noncontiguous data. Where logic or documentation does not indicate support, a little experimentation is in order.*

Note : Data inside a graph are in the form of Data Plots and they are essentially references to columns, matrix or loose datasets. There is no actual data stored in graphs.

Declaration and Syntax

Similar to other data types, you can declare a **Range** variable using the following syntax:

range [-*option*] *RangeName* = *RangeString*

The left-hand side of the range assignment is uniform for all types of range assignments. Note that the square brackets indicate that the option switch is an optional parameter. At the present (8.1), option switches only apply when assigning a range from a graph. Range names follow Origin variable naming rules; please note that system variable names should be avoided.

The right-hand side of the range assignment, ***RangeString***, changes depending on what type of object the range points to. Individual Range Strings are defined in the sections below on Types of Range Data.

Range notation is used exclusively to define range variables. It cannot be used as a general notation for data access on either side of an expression.

Accessing Origin Objects

A range variable can be assigned to the following types of Origin Objects:

- column
- worksheet
- page
- graph layer
- loose dataset

Once assigned, the range will represent that object so that you can access the object properties and methods using the range variable.

A range may consist of some subset or some combination of standard Origin Objects. Examples include:

- column subrange
- block of cells
- XY range
- XYZ range
- composite range

Types of Range Data

Worksheet Data

For worksheet data, ***RangeString*** takes the form:

[*WorkBookName*]*SheetNameOrIndex*!*ColumnNameOrIndex*[*CellIndex*]

where *ColumnName* can be either the *Long Name* or the *Short Name* of the column.

In any ***RangeString***, a span of continuous sheets, columns, or rows can be specified by providing pairs of sheet, column, or row indices (respectively), separated by a colon, as in ***index1*:*index2***. The keyword **end** can replace ***index2*** to indicate that Origin should pick up all of the indicated objects. For example:

```
range rs = [Book1]4:end!           // Get sheets 4 through last
range rd = [Book2]Sheet3!5:10;     // Get columns 5 through 10
```

In the case of rows the indices must be surrounded by square brackets, so a full range assignment statement for several rows of a worksheet column looks like:

```
range rc1 = [Book1]Sheet2!Col(3)[10:end];   // Get rows 10 through last
range rc2 = [Book1]Sheet2!Col(3)[10:20];    // Get rows 10 through 20
```

The old way of accessing cell contents, via the Cell function is still supported.

If you wish to access column label rows using range, please see Accessing Metadata and the Column Label Row Reference Table.

Column

When declaring a range variable for a column on the active worksheet, the book and sheet part can be dropped, such as:

```
range rc = Col(3)
```

You can further simplify the notation, as long as the actual column can be identified, as shown below:

```
range aa=1;        // col(1) of the active worksheet
range bb=B;        // col(B) of the active worksheet
range cc="Test A"; // col with Long Name ("Test A"), active worksheet
```

Multiple range variables can be declared on the same line, separated by comma. The above example could also have been written as:

```
range aa = 1, bb = B, cc = "Test A";
```

Or if you need to refer to a different book sheet, and all in the same sheet, then the book sheet portion can be combined as follows:

```
range [Book2]Sheet3 aa=1, bb=B, cc="Test A";
```

Because Origin does not force a column's Long Name to be unique (i.e., multiple columns in a worksheet can have the same Long Name), the Short Name and Long Name may be specified together to be more precise:

```
range dd = D"Test 4";  // Assign Col(D), Long Name 'Test 4', to a range
```

Once you have a column range, use it to access and change the parameters of a column:

```
range rColumn = [Book1]1!2;       // Range is a Column
rColumn.digitMode = 1;            // Use Set Decimal Places for display
rColumn.digits = 2;               // Use 2 decimal places
```

Or perform computations:

```
// Point to column 1 of sheets 1, 2, and 3 of the active workbook:
range aa = 1!col(1);
range bb = 2!col(1);
```

```
range cc = 3!col(1);
cc = aa+bb;
```

When performing arithmetic on data in different sheets, you need to use range variables. Direct references to range strings are not yet supported. For example, the script **Sheet3!col(1) = Sheet1!col(1) + Sheet2!col(1);** will not work! If you really need to write in a single line without having to declare range variables, then use Dataset Substitution.

Page and Sheet

Besides a single column of data, a range can be used to access any portion of a page object:

Use a range variable to access an entire workbook:

```
// 'rPage' points to the workbook named 'Book1'
range rPage = [Book1];

// Set the Long Name of 'Book1' to "My Analysis Worksheets"
rPage.LongName$ = My Analysis Worksheets;
```

Use a range variable to access a worksheet:

```
range rSheet = [Book1]Sheet1!;      // Range is a Worksheet (WKS object)
rSheet.name$ = "Statistics";        // Rename Sheet1 to "Statistics".
rSheet.AddCol(StdDev);              // Add a column named StdDev
```

Column Subrange

Use a range variable to address a column subrange, such as

```
// A subrange of col(a) in book1 sheet2
range cc = [book1]sheet2!col(a)[3:10];
```

Or if the desired workbook and worksheet are active, the shortened notation can be used:

```
// A subrange of col(a) in book1 sheet2
range cc = col(a)[3:10];
```

Using range variables, you can perform computation or other operations on part of a column. For example:

```
range r1=1[5:10];
range r2=2[1:6];
r1 = r2; // copy values in row 1 to 6 of column 2 to rows 5 to 10 of column 1
r1[1]=;
// this should output value in row 5 of column 1, which equates to row 1 of column 2
```

Block of Cells

Use a range to access a single cell or block of cells (may span many rows and columns) as in:

```
range aa = 1[2];                    // cell(2,1), row2 of col(1)
range bb = 1[1]:3[10];              // cell(1,1) to cell(10,3)
```

Note: A range variable representing a block of cells can be used as an X-Function argument only, direct calculations are not supported.

Matrix Data

For matrix data, the ***RangeString*** is

[*MatrixBookName*]*MatrixSheetNameOrIndex*!*MatrixObject*

Variable assignment can be made using the follow syntax:

```
// Second matrix object on MBook1, MSheet1
range mm = [MBook1]MSheet1!2;
```

Access the cell contents of a matrix range using the notation ***RangeName*[*row*, *col*]**. For example:

```
range mm=[MBook1]1!1;
mm[2,3]=10;
```

If the matrix contains complex numbers, the string representing the complex number can be accessed as below:

```
string str$;
str$ = mm[3,4]$;
```

Graph Data

For graph data, the ***RangeString*** is

[*GraphWindowName*]*LayerNameOrIndex*!*DataPlot*

An example assignment looks like

```
range ll = [Graph1]Layer1!2;        // Second curve on Graph1, Layer1
```

Option Switches -w, -wx, -wy and -wz

For graph windows, you can use range -w and range -wx, range -wy, range -wz options to get the worksheet column range of a plotted dataset.

range -w always gets the worksheet range of the most dependent variable - which is the Y value for 2D plots and the Z value or matrix object for 3D plots. And since Origin 9.0 SR0, multiple ranges are supported for range -w.

range -wx, range -wy, and range -wz will get the worksheet range of the corresponding X, Y and Z values, respectively.

range -wx, range -wz Require Version: 9.0 SR0

```
// Make a graph window the active window ...
// Get the worksheet range of the Y values of first dataplot:
range -w rW = 1;

// Get the worksheet range of the corresponding X-values:
range -wx rWx = 1;

//Get the worksheet range of the corresponding Y-values:
range -wy rWy = 1;

//Get the worksheet range of the corresponding Z-values:
range -wz rWz = 1;
```

```
// Get the graph range of the first dataplot:
range rG = 1;

// Get the current selection (%C); will resolve data between markers.
range -w rC = %C;
```

Note that in the script above, **rW = [Book1]Sheet1!B** while **rG = [Graph1]1!1**.

Data Selector Ranges on a Graph

You can use the Data Selector tool to select one or more ranges on a graph and to refer to them from LabTalk. For a single selected range, you can use the MKS1, MKS2 system variables. Starting with version 8.0 SR6, a new X-Function, **get_plot_sel**, has been added to get the selected ranges into a string that you can then parse. The following example shows how to select each range on the current graph:

```
string strRange;
get_plot_sel str:=strRange;
StringArray sa;
sa.Append(strRange$,"|"); // Tokenize it
int nNumRanges = sa.GetSize();
if(nNumRanges == 0)
{
   type "there is nothing selected";
   return;
}
type "Total of $(nNumRanges) ranges selected for %C";
for(int ii = 1; ii <= nNumRanges; ii++)
{
   range -w xy = sa.GetAt(ii)$;
   string strWks$ = "Temp$(ii)";
   create %(strWks$) -wdn 10 aa bb;
   range fitxy = [??]!(%(strWks$)_aa, %(strWks$)_bb);
   fitlr iy:=xy oy:=fitxy;
   plotxy fitxy p:=200 o:=<active> c:=color(red) rescale:=0 legend:=0;
   type "%(xy) fit linear gives slope=$(fitlr.b)";
}
// clear all the data markers when done
mark -r;
```

Additional documentation is available for the the Create (Command) (for creating loose datasets), the [??] range notation (for creating a range from a loose dataset), the **fitlr** X-Function, and the StringArray (Object) (specifically, the **Append** method, which was introduced in Origin 8.0 SR6).

Loose Dataset

Loose Datasets are similar to columns in a worksheet but they don't have the overhead of the book-sheet-column organization. They are typically created with the create command, or automatically created from an assignment statement without Dataset declaration.

The ***RangeString*** for a loose dataset is:

[??]!*LooseDatasetName*

Assignment can be performed using the syntax:

```
range xx = [??]!tmpdata_a;       // Loose dataset 'tmpdata_a'
```

To show how this works, we use the **plotxy** X-Function to plot a graph of a loose dataset.

```
// Create 2 loose datasets
create tmpdata -wd 50 a b;
tmpdata_a=data(50,1,-1);
tmpdata_b=normal(50);
// Declare the range and explicitly point to the loose dataset
range aa=[??]!(tmpdata_a, tmpdata_b);
// Make a scatter graph with it:
plotxy aa;
```

Please read more about Using Ranges in X-Functions.

Loose datasets belong to a project, so they are different from a Dataset variable, which is declared, and has either session or local scope. Dataset variables are also internally loose datasets but they are limited to use in calculations only; they cannot be used in making plots, for example.

Methods of Range

Once a range variable is created, the following methods can be used by this range

Method	Description
range.getSize()	Return the size of the range. This method works for a dataset range, such as column, matrix object, graph plot, block of cells, loose dataset, etc. Note that, for a block of cells, it only returns the size of the first sub column specified in the range declaration.
range.setSize()	Set the size of the range. This method works for a dataset range, such as column, matrix object, graph plot, block of cells, loose dataset, etc. If the range is block of cells, it only set the size for the first sub column specified in the range declaration.
range.getLayer()	If the range has an attached layer (graph layer, worksheet, or matrix layer), this method will return the **uid** of the layer, to get the name of

	the layer, you need the **$** sign after the method, such as "rng.getLayer()$ = ".
range.getPage()	If the range has an attached page (graph page, workbook, or matrixbook), this method will return the **uid** of the page, to get the name of the page, you need the **$** sign after the method, such as "rng.getPage()$ = ".
range.reverse()	This method works for a dataset range, such as column, matrix object, graph plot, block of cells, loose dataset, etc. It will reverse the data order of the range. If the range is block of cells, it only reverses the data order of the first sub column specified in the range declaration. The X-Function, colReverse, will do the same thing.

Unique Uses of Range

Manipulating Range Data

A column range can be used to manipulate data directly. One major advantage of using a range rather than the direct column name, is that you do not need to be concerned with which page or layer is active.

For example:

```
// Declare two range variables, v1 and v2:
range [Book1]Sheet1 r1=Col(A), r2=Col(B);

// Same as col(A)=data(1,30) if [book1]sheet1 is active:
r1 = data(1,30);
r2 = uniform(30);

// Plot creates new window so [Book1]Sheet1 is NOT active:
plotxy 2;
sec -p 1.5;            // Delay
r2/=4;                 // But our range still works; col(A)/=4 does NOT!
sec -p 1.5;            // Delay
r2+=.4;
sec -p 1.5;            // Delay
r1=10+r1/3;
```

Direct calculations on a column range variable that addresses a range of cells is supported. For example:

```
range aa = Col(A)[10:19]; // Row 10 to 19 of column A
aa += 10;                 // All elements in aa increase by 10
```

Support for sub ranges in a column has expanded.

```
// Range consisting of column 1, rows 7 to 13 and column 2, rows 3 to 4
// Note use of parentheses and comma separator:
range rs = (1[7:13], 2[3:4]);
del rs; // Supported since 8.0 SR6

// Copying between sub ranges
range r1 = 1[85:100];
range r2 = 2;
// Copy r1 to top of column 2
r2 = r1; // Supported in 8.1
// 8.1 also complete or incomplete copying to sub range
range r2 = 2[17:22];
r2 = r1; // Only copies 6 values from r1
range r2 = 3[50:200];
r2 = r1; // Copies only up to row 65 since source has only 16 values
```

Dynamic Range Assignment

Sometimes it is beneficial to be able to create a new range in an automated way, at runtime, using a variable column number, or the name of another range variable.

Define a New Range Using an Expression for Column Index

The **wcol()** function is used to allow runtime resolution of actual column index, as in

```
int nn = 2;
range aa=wcol(2*nn +1);
```

Define a New Range Using an Existing Range

The following lines of script demonstrate how to create one range based on another using the %( ) substitution notation and wks (object) methods. When the **%( )** substitution is used on a range variable, it always resolves it to a **[Book]Sheet!** string, regardless of the type:

```
range rwks = sheet3!;
range r1= %(rwks)col(a);
```

in this case, the new range *r1* will resolve to **Sheet3!Col(A)**.

This method of constructing new range based on existing range is very useful because it allows code centralization to first declare a worksheet range and then to use it to declare column ranges. Lets now use the *rwks* variable to add a column to Sheet 3:

```
rwks.addcol();
```

And now define another range that resolves to the last (rightmost) column of range **rwks**; that is, it will point to the newly made column:

```
range r2 = %(rwks)wcol( %(rwks)wks.ncols );
```

With the range assignments in place it is easy to perform calculations and assignments, such as:

```
r2=r1/10;
```

which divides the data in range **r1** by 10 and places the result in the column associated with range **r2**.

X-Function Argument

Many X-functions use ranges as arguments. For example, the *stats* X-Function takes a vector as input and calculates descriptive statistics on the specified range. So you can type:

```
stats [Book1]Sheet2!(1:end); // stats on the second sheet of book1
stats Col(2);                // stats on column 2 of active worksheet

// stats on block of cells, col 1-2, row 5-10
stats 1[5]:2[10];
```

Or you can use a range variable to do the same type of operation:

```
/* Defines a range variable for col(2) of 1st and 2nd sheet,
rows 3-5, and runs the stats XF on that range: */
range aa = (1,2)!col(2)[3:5]; stats aa;
```

The input vector argument for this X-Function is then specified by a range variable.

Some X-Functions use a special type of range called XYRange, which is essentially a composite range containing X and Y as well as error bar ranges.
The general syntax for an XYRange is

```
(rangeX, rangeY)
```

but you can also skip the rangeX portion and use the standard range notation to specify an XYRange, in which case the default X data is assumed.
The following two notations are identical for XYRange,

```
(, rangeY)
rangeY
```

For example, the integ1 X-Function takes both input and output XYRange,

```
// integrate col(1) as X and col(2) as Y,
// and put integral curve into columns 3 as X and 4 as Y
integ1 iy:=(1,2) oy:=(3,4);

// same as above except result integral curve output to col(3) as Y,
// and sharing input's X of col(1):
integ1 iy:=2 oy:=3;
```

Listing, Deleting, and Converting Range Variables

Listing Range Variables

Use the **list** LabTalk command to print a list of names and their defined bodies of all session variables including the range variables. For example:

```
list a; // List all session variables
```

If you issue this command in the Command Window, it prints a list such as:

```
Session:
 1       MYRANGE   [book1]sheet1!col(b)
 2         MYSTR  "abc"
 3            PI   3.1415926535898
```

As of Origin 8.1, more switches have been added (given below) to list particular session variables:

Option	What Gets Listed	Option	What Gets Listed
a	All session variables	aa	String arrays (session)
ac	Constants (session)	af	Local Function (session)
afc	Local Function Full Content (session)	afp	Local Function Prototype (session)
ag	Graphic objects (session)	ar	Range variables (session)
as	String variables (session)	at	Tree variables (session)
av	Numeric variables (session)	--	--

Deleting Range Variables

To delete a range variable, use the **del** LabTalk command with the -ra switch. For example:

```
range aa=1;  // aa = Col(1) of the active worksheet
range ab=2;  // ab = Col(2) of the active worksheet
range ac=3;  // ac = Col(3) of the active worksheet
range bb=4;  // bb = Col(4) of the active worksheet
list a;      // list all session variables; will include aa, ab, ac, bb
del -ra a*;  // delete all range variables beginning with the letter "a"

// The last command will delete aa, ab, and ac.
```

The table below lists options for deleting variables.

Option	What Gets Deleted/Cleared	Option	What Gets Deleted/Cleared
ra	Any Local/Session variable	al	same as -ra
rar	Range variable	ras	String variable
rav	Numeric variable	rac	Constant
rat	Tree variable	raa	String array
rag	Graphic object	raf	Local/Session Function

Converting Range to UID

Each Origin Object has a short name, long name, and universal identifier (UID). You can convert between range variables and their UIDs as well as obtain the names of pages and layers using the functions **range2uid**, **uid2name**, and **uid2range**. See LabTalk Objects for examples of use.

Special Notations for Range

XY and XYZ Range

Designed as inputs to particular X-Functions, an XY Range is an ordered pair designating two worksheet columns as XY data. Similarly, an (XYZ Range) is an ordered triple containing three worksheet columns representing XYZ data.

For instance, the **fitpoly** X-Function takes an XY range for both input and output:

```
// Fit a 2nd order polynomial to the XY data in columns 1 and 2;
// Put the coefficients into column 3 and the XY fit data in cols 4 and 5:
fitpoly iy:=(1,2) polyorder:=2 coef:=3 oy:=(4,5);
```

XY Range using # and ? for X

There are two special characters '?' and '#' introduced in (8.0 SR3) for range as an X-Function argument. '?' indicates that the range is forced to use worksheet designation, and will fail if the range designation does not satisfy the requirement. '#' means that the range ignores designations and uses row number as the X designation. However, if the Y column has even sampling information, that sampling information will be used to provide X.

For example:

```
plotxy (?, 5);        // if col(5) happens to be X column call fails
plotxy (#, 3);        // plot col(3) as Y and use row number as X
```

These notations are particularly handy in the **plotxy** X-Function, as demonstrated here:

```
// Plot all columns in worksheet using their column designations:
plotxy (?,1:end);
```

Tag Notations in Range Output

Many X-Functions have an output range that can be modified with tags, including *template*, *name*, and *index*. Here is an example that can be used by the Discrete Frequency X-Function, **discfreqs**

```
discfreqs irng:=1 freq:=1 rd:="[Result]<new template:=table.otw index:=3>";
```

The output is directed to a Workbook named **Result** by loading a template named TABLE.OTW as the third sheet in the Result book.

Support of tag notation depends on the particular X-Function, so verify tag notation is supported before including in production code.

Composite Range

A Composite Range is a range consisting of multiple subranges. You can construct composite ranges using the following syntax:

```
// Basic combination of three ranges:
(range1, range2, range3)

// Common column ranges from multiple sheets:
(sheet1,sheet2,sheet3)!range1

// Common column ranges from a range of sheets
(sheet1:sheetn)!range1
```

To show how this works, we will use the **wcellcolor** X-Function to show range and **plotxy** to show XYRange. Assuming we are working on the active book/sheet, with at least four columns filled with numeric data:

```
// color several different blocks with blue color
wcellcolor (1[1]:2[3], 1[5]:2[5], 2[7]) color(blue);

// set font color as red on some of them
wcellcolor (1[3]:4[5], 2[6]:3[7]) color(red) font;
```

To try **plotxy**, we will put some numbers into the first sheet, add a new sheet, and put more numbers into the second sheet.

```
// plot A(X)B(Y) from both sheets into the same graph.
plotxy (1:2)!(1,2);

// Activate workbook again and add more sheets and fill them with data.
// Plot A(X)B(Y) from all sheets between row2 and row10:
plotxy (1:end)!(1,2)[2:10];
```

Note: There exists an inherent ambiguity between a composite range, composed of ranges **r1** and **r2** as in (**r1**,**r2**), and an XY range composed of columns named **r1** and **r2**, i.e., (**r1**,**r2**). Therefore, it is important that one keep in mind what type of object is assigned to a given range variable!

4.2.2 Substitution Notation

Introduction

When a script is executed, it is sent to the LabTalk interpreter. Among other tasks, the interpreter searches for special substitution notations, which are identified by their initial characters, % or $. When a substitution notation is found, the interpreter replaces the original string with another string, as described in the following section. The value of the substituted string is unknown until the statement is actually executed. Thus, this procedure is called a run-time string substitution.

There are three types of substitutions described below:

- String register substitution, %A - %Z
- %( ) Substitution, a powerful notation to resolve **%(str)**, **%(range)**, worksheet info and column dataset names, worksheet cells, legend and etc.
- $( ) Substitution, where $(*expression*) resolves the numeric expression and formats the result as a string

%A - %Z

Using a string register is the simplest form of substitution. String registers are substituted by their contents during script execution, for example

```
FDLOG.Open(A);   // put file name into %A from dialog
%B=FDLOG.path$;  // file path put into %B
doc -open %B%A;  // %B%A forms the full path file name
```

String registers are used more often in older scripts, before the introduction of string variables (Origin 8), which allows for more reliable codes. To resolve string variables, %( ) substitution is used, and is discussed in the next section.

%( ) Substitution

String Expression Substitution

While LabTalk commands often accept numeric expressions as arguments, none accept a string expression. So if a string is needed as an argument, you have to pass in a string variable or a string expression using the %( ) substitution to resolve run-time values. The simplest form of a string expression is a single string variable, like in the example below:

```
string str$ = "Book2";
win -o %(str$) {wks.ncols=;}
```

Keyword Substitution

The %( ) substitution notation is also used to insert non-printing characters (also called control characters), such as tabs or carriage returns into strings. Use LabTalk keywords to access these non-printing characters. For example,

```
// Insert a carriage-return, line-feed (CRLF) into a string:
string ss$ = "Hello%(CRLF)Goodbye";
ss$=;     // ANS: 'Hello', 'Goodbye' printed on separate lines

// Or use %() substitution to display the string variable:
type %(ss$);
```

Worksheet Column and Cell Substitution

The following notation allows you to access worksheet cells as a string as well as to get the column dataset name from any workbook sheet. Before Origin 8, each book had only one sheet so you could refer to its content with the book name only. Since Origin 8 supports

multiple worksheets, we recommend that you use **[workbookname]sheetname** to refer to a specific sheet, unless you are certain that the workbook contains only one sheet.

To return individual cell contents, use the following syntax:

- This notation references the active sheet in the named book

%(*workbookName*, *column*, *row*)

- New Origin 8 notation that specifies book and sheet

%([*workbookname*]*sheetname*, *column*, *row*[,*format*])

For example, if the third cell in the fourth column in the active worksheet of Book1 contains the value 25, then entering the following statement in the Script window will set A to 25 and put double that value in another sheet in Book1.

```
A = %(Book1, 4, 3);
%([Book1]Results, 1, 4) = 2 * A;
```

To return the contents of a text cell, use a string variable:

```
string strVar$ = %(Book1, 2, 5); // Note : No end '$' needed here
strVar$ = ;
```

Before 8.1, you must use column and row index and numeric cell will always return full precision. Origin 8.1 has added support for *column* to allow both index and name, and *row* will also support Label Row Characters such as L for longname. There is also an optional *format* argument that you can use to further specify numberic cell format when converting to string. Assuming Book2, sheet3 col(Signal)[3] has a numeric value of 12.3456789, then

```
//format string C to use current column format
type "Col(Signal)[3] displayed value is %([Book2]Sheet3,Signal,3,C)";
A=%([Book2]Sheet3,Signal,3);//full precision if format not specified
A=;// shows 12.3456789
type "Showing 2 decimal places:%([Book2]Sheet3,Signal,3,.2)";
```

To return a dataset name, use the following syntax:

- Older notation for active sheet of named book

%(*workbookName*, *column*)

- New Origin 8 book sheet notation

%([*workbookName*]*sheetName*, *column*)

- You can also use index

%([*workbookName*]*SheetIndex*, *column*)

where ***column*** must be an index prior to Origin 8.1 which added support for column name.

For example:

```
%A = %(%H, 2);           // Column 2 of active sheet of active book
type %A;
%B = %([Book1]Sheet3,2); // Column 2 of Book1, Sheet3
type %B;
```

In the above example, the name of the dataset in column 2 in the active worksheet is substituted for the expression on the right, and then assigned to %A and %B. In the second case, if the named book or sheet does not exist, no error occurs but the substitution will be invalid.

Note: You can use parentheses to force assignment to be performed on the dataset whose name is contained in a string register variable instead of performing the assignment on the string register variable itself.

```
%A = %(Book1,2);   // Get column 2 dataset name
type %A;           // Types the name of the dataset
(%A) = %(Book1,1); // Copy column 1 data to column 2
```

Calculation Involving Datasets from Another Sheet

The ability to get a dataset name from any book or sheet (Dataset Substitution) can be very useful in doing calculations involving columns in different sheets, like:

```
// Sum col(1) from sheet2 and 3 and place the result into col(1) of the active sheet
col(1)=%([%H]sheet2, 1) + %([%H]sheet3, 1);

// subtract by col "signal" in the 1st sheet of book2 and
// put result into the active book's sheet3, "calibrated" col
%([%H]sheet3, "calibrated")=col(signal) - %([Book2]1,signal);
```

The ***column*** name should be quoted if using long name. If not quoted, then Origin will first assume short name, if not found, then it will try using long name. So in the example above,

```
%([%H]sheet3, "calibrated")
```

will force a long name search, while

```
%([Book2]1,signal)
```

will use long name only if there is no column with such a short name.

Worksheet Information Substitution

Similar to worksheet column and cell access with substitution notation, the @ Substitution (worksheet info substitution) make uses of the **@** character to differentiate from a column index or name in the 2nd argument to specify various options to provide access to worksheet info and meta data.

Prior to Origin 8, the following syntax is used and is still supported for the active sheet:

%(*workbookName*, *@option*, *columnNumber*)

It is recommended that you use the newer notation introduced in Origin 8:

%([*workbookName*]*worksheetName*, *@option*, *columnNumber*)

Here, *option* can be one of the following:

Option	Return Value
@#	Returns the total number of worksheet columns. *ColumnNumber* can be omitted.

@C	Returns the column name.
@DZ	Remove trailing zeros when **Set Decimal Places** or **Significant Digits** is chosen in the **Numeric Display** drop down list of the **Worksheet Column Format** dialog box. 0 = display trailing zeros. 1 = remove trailing zeros for Set Decimal Places =. 2 = remove trailing zeros for Significant Digits =. 3 = remove for both.
@E#	If *columnNumber* = 1, returns the number of Y error columns in the worksheet. If *columnNumber* = 2, returns the number of Y error columns in the current selection range. If *columnNumber* is omitted, *columnNumber* is assumed to be 1.
@H#	If **columnNumber** = 1, returns the number of X error columns in the worksheet. If *columnNumber* = 2, returns the number of X error columns in the current selection range. If *columnNumber* is omitted, *columnNumber* is assumed to be 1.
@PC	Page Comments
@PC1	Page Comments, 1st line only
@PL	Page Long Name
@PN	Page short Name
@SN	Sheet Name
@SC	Sheet Comments
@OY	Returns the offset from the left-most selected Y column to the *columnNumber* column in the current selection.
@OYX	Returns the offset from the left-most selected Y column to the *columnNumber* Y column counting on Y columns in the current selection.
@OYY	Returns the offset from the left-most selected Y column to the *columnNumber* X column counting on X columns in the current selection.

@T	Returns the column type. 1 = Y , 2 = disregarded, 3 = Y error, 4 = X , 5 = label, 6 = Z, and 7 = X error.
@W	Returns information stored at the Book or Sheet level as well as imported file information. Refer to the table below for the @W group of variables.
@X	Returns the index number of the worksheet's first X column. Columns are enumerated from left to right, starting from 1. Use the syntax: %(worksheetName, @X);
@Xn	Returns the column short name of the worksheet's first X column. Use the syntax: %(worksheetName, @Xn);
@Y	Returns the offset from the left-most selected column to the *columnNumber* column in the current selection.
@Y-	Returns the column number of the first Y column to the left. Returns columnNumber if the column is a Y column, or returns 0 when the Y column doesn't exist. Use the syntax: %(worksheetName, @Y-, ColumnNumber);
@Y#	If *columnNumber* = 1, returns the number of Y columns in the worksheet. If *columnNumber* = 2, returns the number of Y columns in the current selection range. If *columnNumber* is omitted, *columnNumber* is assumed to be 1.
@Y+	Returns the column number of the first Y column to the right. Returns *columnNumber* if the column is a Y column, or returns 0 when the Y column doesn't exist. Use the syntax: %(worksheetName, @Y+, ColumnNumber);
@YS	Returns the number of the first selected Y column to the right of (and including) the *columnNumber* column.
@Z#	If columnNumber = 1, returns the number of Z columns in the worksheet. If *columnNumber* = 2, returns the number of Z columns in the current selection range. If *columnNumber* is omitted, *columnNumber* is assumed to be 1.

The options in this table are sometimes identified as @ options or @ variables.

Information Storage and Imported File Information

The **@W** variables access metadata stored within Origin workbooks, worksheets, and columns, as well as information stored about imported files.

Use a similar syntax as above, replacing column number with variable or node information:

%([*workbookName*]*worksheetName*!*columnName*, *@option*, *varOrNodeName*)

Option	Return Value
@W	Returns the information in *varOrNodeName*; the variable is understood to be located at the workbook level, which can be seen in workbook Origanizer. When it is used, there is no need to specify *worksheetName!ColumnName*.
@WL	Returns the information in *varOrNodeName*; the variable is understood to be located at workbook level, which can be seen in workbook Origanizer. It refers to the workbook long name.
@WF*n*	Returns the information in *varOrNodeName* for the *n*th imported file. The variable can be seen in the workbook Organizer.
@WS	Returns the information in *varOrNodeName*; the variable is understood to be located at the worksheet level, which can be seen in workbook Organizer. When it is used, there is no need to specify *ColumnName*.
@WM	Returns the information in *varOrNodeName*; the variable is understood to be located at worksheet level, which can be seen in workbook Organizer. It refers to the worksheet comment.
@WC	Returns the information in *varOrNodeName*; the variable is understood to be located at the column level, which can be seen in the **Column Properties** dialog.

Examples of @ Substitution

This script returns the column name of the first column in the current selection range (for information on the **selc1** numeric system variable, see System Variables):

```
%N = %(%H, @col, selc1);  %N =;
```

The following line returns the active page's long name to a string variable:

```
string PageName$ = %(%H, @PL);
```

The script below returns the column type for the fourth column in Book 2, Sheet 3:

```
string colType$ = %([Book2]Sheet3, @T, 4);
colType$=;
```

An import filter can create a tree structure of information about the imported file that gets stored with the workbook. Here, for a multifile import, we return the number of points in the 3rd dataset imported into the current book:

```
%z=%(%H,@WF3,variables.header.noofpoints);
```

```
%z=
```

If the currently active worksheet window has six columns (XYYYYY) and columns 2, 4, and 5 are selected, then the following script shows the number of the first selected Y column to the right of (and including) the column whose index is equal to *columnNumber* (the third argument):

```
loop(ii,1,6)
{
    type -l %(%H, @YS, ii),;
}
type;
```

This outputs:

```
   2,2,4,4,5,0,
```

Legend Substitution

Graph legends also employ the %( ) substitution notation. The first argument must be an integer to differentiate it from other %( ) notations, where the first argument is a worksheet specifier. The legend substitution syntax is:

%(*n*[, *@option*])

where *n* is the index of the desired data plot in the current layer. The variable *n* might be followed by more options, typically plot designation character (X, Y or Z) associated with the data plot, which when not specified will be assumed to be Y. The *@option* parameter is an optional argument that controls the legend contents. For example:

```
// In the legend of the current graph layer ...
// display the Long Name for the first dependent dataset.
legend.text$ = %(1Y, @LL)

// Equivalent command (where, Y, the default, is understood):
legend.text$ = %(1, @LL)
```

Alternatively, to display the Short Name for the second independent (i.e., X) dataset in the legend use:

```
legend.text$ = %(2X, @LS)
```

The complete list of **@options** is found in the @ text-label options.

Note: This style of legend modification is limited in that it only changes a single legend entry, but the syntax is good to understand, as it can be used in the **Plot Details** dialog.

The legendupdate X-Function provides an easier and more comprehensive way to modify or redraw a legend from Script!

$( ) Substitution

The $() notation is used for numeric to string conversion. This notation evaluates the given expression at run-time, converts the result to a numeric string, and then substitutes the string for itself.

The notation has the following form:

$(*expression* [, *format*])

where *expression* can be any mathematical expression, but typically a single number or variable, and *format* can be either an Origin output format or a C-language format.

Default Format

The square brackets indicate that *format* is an optional argument for the **$()** substitution notation. If *format* is excluded Origin will carry *expression* to the number of decimal digits or significant figures specified by the **@SD** system variable (which default value is 14). For example:

```
double aa =  3.14159265358979323846;
type $(aa);    // ANS: 3.1415926535898
```

Origin Formats

Minimum Origin Version Required: 8.5.1 SR0

Origin has several native options to format your output.

Format	Description
**n*	Display *n* significant digits
n	Display *n* significant digits, truncating trailing zeros
S**n*	Display *n* significant digits, in scientific notation
E**n*	Display *n* significant digits, in engineering format
.n	Display *n* decimal places
S.*n*	Display *n* decimal places, in scientific notation
E.*n*	Display *n* decimal places, in engineering format
D*c*	Display date in the format customized by the *c* string.

D*n*	Display date in format *n* from the **Display** drop down list of the **Column Properties** dialog box
T*c*	Display time in the format customized by the *c* string.
T*n*	Display time in format *n* from the **Display** drop down list of the **Column Properties** dialog box
#*n*	Display an integer to *n* places, zero padding where necessary

This block of script demonstrates several examples of Origin formats:

```
xx = 1.23456;
type "xx = $(xx, *2)";  // ANS: 1.2
type "xx = $(xx, .2)";  // ANS: 1.23

yy = 1.10001;
type "yy = $(yy, *4)";   // ANS: 1.100
type "yy = $(yy, *4*)";  // ANS: 1.1

zz = 203465987;
type "zz = $(zz, E*3)";  // ANS: 203M
type "zz = $(zz, S*3)";  // ANS: 2.03E+08

type "$(date(7/20/2009), D1)";  // ANS: Monday, July 20, 2009

type "$(date(7/20/2009), Dyyyy'-'MM'-'dd)";  // ANS: 2009-07-20

type "$(time(14:31:04), T4)";   // ANS: 02 PM

type "$(time(14:31:04), Thh'.'mm'.'ss)";  // ANS: 02.31.04

type "$(45, #5)";    // ANS: 00045
```

Note: For dates and times *n* starts from zero.

C-Language Formats

The *format* portion of the $() notation also supports C-language formatting statements.

Option	Un/Signed	Output	Input Range
d, i	SIGNED	Integer values (of decimal or integer value)	-2^31 -- 2^31 -1

f, e, E, g, G	SIGNED	Decimal, scientific, decimal-or-scientific	+/-1e290 -- +/-1e-290
o, u, x, X	UNSIGNED	Octal, Integer, hexadecimal, HEXADECIMAL	-2^31 -- 2^32 - 1

Note: In the last category, negative values will be expressed as two's complement.

Here are a few examples of C codes in use in LabTalk:

```
double nn = -247.56;
type "Value: $(nn,%d)";   // ANS: -247

double nn = 1.23456e5;
type "Values: $(nn, %9.4f), $(nn, %9.4E), $(nn, %g)";
// ANS: 123456.0000, 1.2346E+005, 123456

double nn = 1.23456e6;
type "Values: $(nn, %9.4f), $(nn, %9.4E), $(nn, %g)";
// ANS: 123456.0000, 1.2346E+006, 1.23456e+006

double nn = 65551;
type "Values: $(nn, %o), $(nn, %u), $(nn, %X)";
// ANS: 200017, 65551, 1000F
```

Combining Origin and C-language Formats

Origin supports the use of formats *E* and *S* along with C-language format specifiers. For example:

```
xx = 1e6;
type "xx = $(xx, E%4.2f)";  // ANS: 1.00M
```

Displaying Negative Values

The command parsing for the **type** command (and others) looks for the **-** character as an option switch indicator. If you assign a negative value to the variable K and try to use the type command to express that value, you must protect the **-** by enclosing the substitution in quotes or parentheses. For example:

```
K = -5;
type "$(K)"; // This works
type ($(K)); // as does this
type $(K);   // but this fails since type command has no -5 option
```

Dynamic Variable Naming and Creation

Note that in assignment statements, the $() notation is substitution-processed and resolved to a value regardless of which side of the assignment operator it is located.

This script creates a variable A with the value 2.

```
A = 2;
```

Then we can create a variable A2 with the value 3 with this notation:

```
A$(A) = 3;
```

You can verify it by entering **A$(A) =** or **A2 =** in the Script window.

For more examples of $() substitution, see Numeric to String conversion.

%*n* Macro and Script Arguments

Substitutions of the form **%*n***, where *n* is an integer 1-5 (up to five arguments can be passed to a macro or a script), are used for arguments passed into macros or sections of script.

In the following example, the script defines a macro that takes two arguments (%1 and %2), adds them, and outputs the sum to a dialog box:

```
def add {type -b "(%1 + %2) = $(%1 + %2)"}
```

Once defined, the macro can be run by typing:

```
add -13 27;
```

The output string reads:

(-13 + 27) = 14

since the expression $(%1 + %2) resolves to 14.

4.2.3 LabTalk Objects

LabTalk script programming provides access to various objects and their properties. These objects include components of the Origin project that are visible in the graphical interface, such as worksheets columns and data plots in graphs. Such objects are referred to as **Origin Objects**, and are the subject of the next section, Origin Objects.

The collection of objects also includes other objects that are not visible in the interface, such as the INI object or the System object. The entire set of objects accessible from LabTalk script is found in Alphabetical Listing of Objects.

In general, every object has properties that describe it, and methods that operate on it. What those properties and methods are depend on the particular object. For instance, a data column will have different properties than a graph, and the operations you perform on each will be different as well. In either case, we need a general syntax for accessing an object's properties and calling it's methods. These are summarized below.

Also, because objects can be renamed, and objects of different scope may even share a name, object names can at times be ambiguous identifiers. For that reason, each object is assigned a unique universal identifier (UID) by Origin and functions are provided to go back and forth between an object's name and it's UID.

Properties

A property either sets or returns a number or a text string associated with an object with the following syntax:

objName.property (For numeric properties)
***objName.property*$** (For text properties)

Where *objName* is the name of the object; *property* is a valid property for the type of object. When accessing text objects, you should add the **$** symbol after *property*.

For example, you can set object properties in the following way:

```
// Set the number of columns on the active worksheet to 10
wks.ncols = 10;
// Rename the active worksheet 'MySheet'
wks.name$ = MySheet;
```

Or you can get property values:

```
pn$ = page.name$;  // Get that active page name
layer.x.from = ;   // Get and display the start value of the x-axis
```

Methods

Methods are a form of immediate command. When executed, they carry out a function related to the object and return a value. Object methods use the following syntax:

***objName.method*(*arguments*)**

Where *objName* is the name of the object; *method* is a valid method for the type of object; and *arguments* determine how the method functions. Some arguments are optional and some methods do not require any arguments. However, the parentheses "()" must be included in every object method statement, even if their contents are empty.

For example, the following code uses the **section** method of the **run** object to call the **Main** section within a script named **computeCircle**, and passes it three arguments:

```
double RR = 4.5;
string PA$ = "Perimeter and Area";
run.section(computeCircle, Main, PA$ 3.14 R);
```

Object Name and Universal Identifier (UID)

Each object has a short name, a long name, and most objects also have a universal identifier (UID). Both the short name and long name can be changed, but an object's UID will stay the same within a project (also known as an OPJ file). An object's UID can change if you append one project to another one, at which time all object UID's will go through a refresh process to ensure the uniquness of each object in the newly combined project.

Since many LabTalk functions require the name of an object as argument, and since an object can be renamed, the following functions are provided to convert between the two:

- ***nVal* = range2uid(*rangeName*$)**
- ***str*$ = uid2name(*nVal*)$**

- ***str*$ = uid2range(*nVal*)$**

A related function is **NameOf(range$)** with the general syntax:

- ***str*$ = nameof(*rangeName*$)**

Its use is demonstrated in the following example:

```
// Establish a range variable for column 1 (in Book1, Sheet1)
range ra=[Book1]1!1;
// Get the internal name associated with that range
string na$ = NameOf(ra)$;
// na$ will be 'Book1_A'
na$ =;
// Get the UID given the internal name
int nDataSetUID = range2uid(na$);
```

Besides a range name, the UID can be recovered from the names of columns, sheets, or books themselves:

```
// Return the UID of column 2
int nColUID = range2uid(col(2));
// Return the UID of a sheet or layer
int nLayerUID = range2uid([book2]Sheet3!);
// Return the UID of the active sheet or layer
nLayerUID =range2uid(!);
// Return the UID of sheet3 of the active workbook
nLayerUID =range2uid(sheet3!);
// Return the UID of the column with index 'jj' within a specific sheet
nColUID = range2uid([Book1]sheet2!wcol(jj));
```

Additionally, the **range2uid** function works with the system variable **%C**, which holds the name of the active data plot or data column:

```
// Return the UID of the active data plot or selected column
nDataSetUID = range2uid(%C);
```

Getting Page and Layer from a Range Variable

Given a range variable, you can get its corresponding Page and Layer UID. The following code shows how to make a hidden plot from XY data in the current sheet and to obtain the hidden plot's graph page name:

```
plotxy (1,2) ogl:=<new show:=0>; // plot A(x)B(y) to a new hidden plot
range aa=plotxy.ogl$;
int uid=aa.GetPage();
string str$=uid2Name(uid)$;
type "Result graph name is %(str$)";
```

Getting Book And Sheet from a Plot

You can also get a data plot's related workbook and worksheet as range variables. The following code (requires Origin 8 SR2) shows how to get the Active plot (%C) as a column

range and then retrieve from it the corresponding worksheet and book variables allowing complete access to the plot data:

```
// col range for active plot, -w switch default to get the Y column
range -w aa=%C;
// wks range for the sheet the column belongs to
range ss = uid2range(aa.GetLayer())$;
// show sheet name
ss.name$=;
// book range from that col
range bb = uid2range(aa.GetPage())$;
// show book name
bb.name$=;
```

There is also a simpler way to directly use the range string return from GetLayer and GetPage in string form:

```
// col range for active plot, -w switch default to get the Y column
range -w aa=%C;
// sheet range string for the sheet the column belongs to
range ss = aa.GetLayer()$;
// show sheet name
ss.name$=;
// book range string from that col
range bb = aa.GetPage()$;
// show book name
bb.name$=;
```

When you create a range mapped to a page, the range variable has the properties of a PAGE (Object).
When you create a range mapped to a graph layer, the range variable has the properties of a LAYER (Object).
When you create a range mapped to a workbook layer (a worksheet or matrix sheet), the range variable has the properties of a WKS (Object).

4.2.4 Origin Objects

Then there is a set of LabTalk Objects that is so integral to scripting in Origin that we give them a separate name: Origin Objects. These objects are visible in the graphical interface, and will be saved in an Origin project file (.OPJ). Origin Objects are the primary components of your Origin Project. They are the following:

1. Page (Workbook/Graph Window/Matrix Book) Object
2. Worksheet Object
3. Column Object
4. Layer Object
5. Matrix Object

6. Dataset Object
7. Graphic Object

Except loose datasets, Origin objects can be organized into three hierarchies:

Workbook -> Worksheet -> Column

Matrix Book -> Matrix Sheet -> Matrix Object

Graph Window -> Layer -> Dataplot

In the sections that follow, tables list object methods and examples demonstrate the use of these objects in script.

4.2.5 String registers

Introduction

String Registers are one means of handling string data in Origin. Before Version 8.0, they were the only way and, as such, current versions of Origin continue to support the use of string registers. However, users are now encouraged to migrate their string processing routines toward the use of proper string variables, see String Processing for comparative use.

String register names are comprised of a %-character followed by a single alphabetic character (a letter from A to Z). Therefore, there are 26 string registers, i.e., %A--%Z, and each can hold 266 characters (except %Z, which can hold up to 6290 characters).

String registers are of global (session) scope; this means that they can be changed by any script at any time. Sometimes this is useful, other times it is dangerous, as one script could change the value in a string register that is being used by another script with erroneous and confusing results.

Ten (10) of the 26 string registers are reserved for use as system variables, and their use could result in errors in your script. They are grouped in the ranges %C--%I, and %X--%Z. All of the reserved string registers are summarized in the table below.

String Registers as System Variables

String registers hold up to 260 characters. String register names are comprised of a **%**-character followed by a single alphabetic character (a letter from A to Z); for this reason, string registers are also known as % variables. Of the 26 possible string registers, the following are reserved as system variables that have a special meaning, and they should not be reassigned in your scripts. It is often helpful, however, to have access to (or operate on) the values they store.

String Variable	Description
%C	The name of the current active dataset.
%D	Current Working Directory, as set by the cd command. (New in Origin 8)
%E	The name of the window containing the latest worksheet selection.
%F	The name of the dataset currently in the fitting session.
%G	The current project name.
%H	The current active window title.
%I	The current baseline dataset.
%X	The path of the current project.
%Y	The full path name to the User Files folder, where the user .INI files as well as other user-customizable files are located. %Y can be different for each user depending on the location they selected when Origin was started for the first time. Prior to Origin 7.5, the path to the various user .INI files was the same as it was to the Origin .EXE. Beginning with Origin 7.5, we added multi-user-on-single-workstation support by creating a separate "User Files" folder. To get the Origin .EXE path(program path), use the following LabTalk statement: `%a = system.path.program$` In Origin C, pass the appropriate argument to the GetAppPath() function (to return the INI path or the EXE path).
%Z	A long string for temporary storage. (maximumn 6290 characters)

String registers containing system variables can be used anywhere a name can be used, as in the following example:

```
// Deletes the current active dataset:

del %C;
```

String Registers as String Variables

Except the system variable string registers, you can use string registers as string variables, demonstrated in the following examples:

Assigning Values to a String Variable

Entering the following assignment statement in the Script window:

```
%A = John
```

defines the contents of the string variable **%A** to be **John**.

String variables can also be used in substitution notation. Using substitution notation, enter the following assignment statement in the Script window:

```
%B = %A F Smith
```

This sets **%B** equal to **John F Smith**. Thus, the string variable to the *right* of the assignment operator is expressed, and the result is assigned to the *identifier on the left* of the assignment operator.

As with numeric variables, if you enter the following assignment statement in the Script window:

```
%B =
```

Origin returns the value of the variable:

John F Smith

Expressing the Variable Before Assignment

By using parentheses, the string variable on the left of the assignment operator can be expressed before the assignment is made. For example, enter the following assignment statement in the Script window:

```
%B = Book1_A
```

This statement assigns the string register %B the value **Book1_A**. If **Book1_A** is a dataset name, then entering the following assignment statement in the Script window:

```
(%B) = 2*%B
```

results in the dataset being multiplied by **2**. String register **%B**, however, still contains the string **Book1_A**.

String Comparison

When comparing string registers, use the "equal to" operator (==).

- If string registers are surrounded by quotation marks (as in, "%a"), Origin literally compares the string characters that make up each variable name. For example:

```
aaa = 4;
bbb = 4;
%A = aaa;
%B = bbb;
if ("%A" == "%B")
    type "YES";
```

```
else
    type "NO";
```

The result will be **NO**, because in this case **aaa != bbb**.

- If string registers are not surrounded by quotation marks (as in, %a), Origin compares the values of the variables stored in the string registers. For example:

```
aaa = 4;
bbb = 4;
%A = aaa;
%B = bbb;
if (%A == %B)
    type "YES";
else
    type "NO"
```

The result will be **YES**, because in this case the values of the strings (rather than the characters) are compared, and **aaa == bbb == 4**.

Substring Notation

Substring notation returns the specified portion of a string. The general form of this notation is:

%[*string*, *argument*];

where ***string*** contains the string itself, and ***argument*** specifies which portion of the string to return.

For the examples that follow, enter this assignment statement in the Script window:

```
%A = "Results from Data2_Test"
```

The following examples illustrate the use of argument in substring notation:

To do this:	Enter this script:	Return value:
Search for a character and return all text to the left of the character.	`%B = %[%A, '_']; %B =`	Results from Data2
Search for a character and return all text to the right of the character.	`%B = %[%A, >'_']; %B =`	Test
Return all text to the left of the specified character position.	`%B = %[%A, 8]; %B =`	Results
Return all text between two specified character positions (inclusive).	`%B = %[%A, 14:18]; %B =`	Data2
Return the `#n` token, counting from the left.	`%B = %[%A, #2]; %B =`	from<

Return the length of the string.	`ii = %[%A]; ii =`	ii = 23

Other examples of substring notation:

To do this:	Enter this script:	Return value:
Return the ith token separated by a specified separator (in this case, a tab)	`%A = 123342 456; for (ii = 1; ii <= 3; ii++) { Book1 A[ii] = %[%A, #ii,\t] };`	Places the value **123** in **Book1_a[1]**, **342** in **Book1_a[2]**, and **456** in **Book1_a[3]**.
Return the **@n** line	`%Z = "First line second line"; %A = %[%Z, @2];`	Places the second line of the **%Z** string into **%A**. To verify this, type **%A =** in the Script window.

Note:

When using quotation marks in substring or substitution notation:

- Space characters are *not* ignored.
- String length notation includes space characters.

For example, to set **%A** equal to **5** and find the length of **%A**, type the following in the Script window and press *Enter*:

```
%A = " 5 ";
ii = %[%A];
ii = ;
```

Origin returns: **ii = 3**.

A Note on Tokens

A token can be a word surrounded by white space (spaces or TABS), or a group of words enclosed in any kind of brackets. For example, if:

```
%A = These (are all) "different tokens"
```

then entering the following in the Script window:

Scripts	Returns
`%B = %[%A, #1]; %B=`	These
`%B = %[%A, #2]; %B=`	are all

`%B = %[%A, #3]; %B=`	different tokens

4.2.6 X-Functions Introduction

The X-Function is a new feature, introduced in Origin 8, that provides a framework for building Origin tools. Most X-Functions can be accessed from LabTalk script to perform tasks like object manipulation or data analysis.

The general syntax for issuing an X-Function command from script is as follows, where square-brackets [ ] indicate optional statements:

xfname [-options] arg1:=value arg2:=value ... argN:=value;

Note that when running X-Functions, Origin uses a combined colon-equal symbol, "**:=**", to assign argument values. For example, to perform a simple linear fit, the fitlr X-Function is used:

```
// Data to be fit, Col(A) and Col(B) of the active worksheet,
// is assigned, using :=, to the input variable 'iy'
fitlr iy:=(col(a), col(b));
```

Also note that, while most X-Functions have optional arguments, it is often possible to call an X-Function with no arguments, in which case Origin uses default values and settings. For example, to create a new workbook, call the newbook X-Function:

```
newbook;
```

Since X-Functions are easy and useful to run, we will use many of them in the following script examples. Details on the options (including getting help, opening the dialog and creating auto-update output) and arguments for running X-Functions are discussed in the Calling X-Functions and Origin C Functions section.

4.3 LabTalk Script Precedence

Now that we know that there are several objects, like Macros, Origin C functions, X-Functions, OGS files, etc. So, we should be careful to avoid naming conflicts between these objects, which could cause confusion and lead to incorrect results. If duplicate names are unavoidable, LabTalk will run objects according to set of precedence rules. The following list of LabTalk objects are arranged top to bottom in descending precedence.

1. Macros
2. OGS Files
3. X-Functions
4. LT object methods, like run.file(*FileName*)
5. LT callable Origin C functions
6. LT commands (can be abbreviated)

5 Calling X-Functions and Origin C Functions

5.1 X-Functions

X-Functions are a primary tool for executing tasks and tapping into Origin features from your LabTalk scripts. The following sections outline the details that will help you recognize, understand, and utilize X-Functions in LabTalk.

5.1.1 X-Functions Overview

X-Functions provide a uniform way to access nearly all of Origin's capabilities from your LabTalk scripts. The best way to get started using X-Functions is to follow the many examples that use them, and then browse the lists of X-Functions accessible from script provided in the LabTalk-Supported X-Functions section.

Syntax

You can recognize X-Functions in script examples from their unique syntax:

xFunctionName input:=<*range*> argument:=<*name/value*> output:=<*range*> -switch;
General Notes:

- X-Functions can have multiple inputs, outputs, and arguments.
- X-Functions can be called with any subset of their possible argument list supplied.
- If not supplied a value, each required argument has a default value that is used.
- Each X-Function has a different set of input and output arguments.

Notes on X-Function Argument Order:

- By default, X-Functions expect their input and output arguments to appear in a particular order.
- Expected argument order can be found in the help file for each individual X-Function or from the Script window by entering **XFunctionName -h**.
- If the arguments are supplied in the order specified by Origin, there is no need to type out the argument names.
- If the argument names are explicitly typed, arguments can be supplied in any order.
- It allows to omit parts of argument names at the first few positions in the order specified by Origin, then from the first one with argument name, the followings must also type out the argument name, but can be supplied in any order.

- The argument name can be shorten by trimming some characters from behind of the argument name, but the shorten name needs to be unique.

The following examples use the **fitpoly** X-Function to illustrate these points.

Examples

The **fitpoly** X-Function has the following specific syntax, giving the order in which Origin expects the arguments:

fitpoly iy:=(inputX,inputY) polyorder:=n coef:=columnNumber oy:=(outputX,outputY) N:=numberOfPoints;

If given in the specified order, the X-Function call,

```
// need to specify 0 for Fit Intercept and Fit Intercept At
// for the proper order
fitpoly (1,2) 4 0 0 3 (4,5) 100;
```

tells Origin to fit a 4th order polynomial with 100 points to the X-Y data in columns 1 and 2 of the active worksheet, putting the coefficients of the polynomial in column 3, and the X-Y pairs for the fit in columns 4 and 5 of the active worksheet.

In contrast, the command with all options typed out is a bit longer but performs the same operation:

```
fitpoly iy:=(1,2) polyorder:=4 coef:=3 oy:=(4,5) N:=100;
```

In return for typing out the input and output argument names, LabTalk will accept them in any order, and still yield the expected result:

```
fitpoly coef:=3 N:=100 polyorder:=4 oy:=(4,5) iy:=(1,2);
```

Another way is to omit just some argument names, then followed by other arguments with names in any order, like below script, which gets the same result as above.

```
fitpoly (1,2) 4 oy:=(4,5) N:=100 coef:=3;
```

And it allows to shorten the argument name if the shorten name is unique in the argument list, such as

```
fitpoly iy:=(1,2) poly:=4 co:=3 o:=(4,5) N:=100;
```

poly is short for **polyorder**, and **co** for **coef**, and **o** for **oy**. If using the following script, there will be an error.

```
fitpoly i:=(1,2) poly:=4 co:=3 o:=(4,5) N:=100;
```

Because there are two argument names (**iy** and **intercept**) begin with letter **i**. Here Origin cannot tell which argument **i** is standard for, that is to say **i** is not unique.

Also, the inputs and outputs can be placed on separate lines from each other and in any order, as long as they are explicitly typed out.

```
fitpoly
coef:=3
N:=100
polyorder:=4
oy:=(4,5)
iy:=(1,2);
```

Notice that the semicolon ending the X-Function call comes only after the last parameter associated with that X-Function.

Option Switches

Option switches such as **-h** or **-d** allow you to access alternate modes of executing X-functions from your scripts. They can be used with or without other arguments. The option switch (and its value, where applicable) can be placed anywhere in the argument list. This table summarizes the primary X-Function option switches:

Name	Function
-h	Prints the contents of the help file to the Script window.
-d	Brings up a graphical user interface dialog to input parameters.
-s	Runs in silent mode; results not sent to Results log.
-t <*themeName*>	Uses a pre-set theme.
-r <*value*>	Sets the output to automatically recalculate if input changes.

For more on option switches, see the section X-Function Execution Options.

Generate Script from Dialog Settings

The easiest way to call an X-Function is with the *-d* option and then configures its settings using the graphical user interface (GUI).

In the GUI, once the dialog settings are done, you can generate the corresponding LabTalk script for the configuration by selecting the **Generate Script** item in the dialog theme fly-out menu. Then a script which matches the current GUI settings will be output to script window and you can copy and paste it into a batch OGS file or some other project for use.

5.1.2 X-Function Input and Output

X-Function Variables

X-Functions accept LabTalk variable types (except StringArray) as arguments. In addition to LabTalk variables, X-Functions also use special variable types for more complicated data structures.

These special variable types work only as arguments to X-Functions, and are listed in the table below (Please see the **Special Keywords for Range** section below for more details about available key words.):

Variable Type	Description	Sample Constructions	Comment
XYRange	A combination of X, Y, and optional Y Error Bar data	1. (1,2) 2. <new> 3. (1,2:end) 4. (<input>,<new>) 5. [book2]sheet3!<new>	For graph, use index directly to indicate plot range (1,2) means 1st and 2nd plots on graph
XYZRange	A combination of X, Y, and Z data	1. (1,2,3) 2. <new> 3. [book2]sheet3!(1,<new>,<new>)	
ReportTree	A Tree based object for a Hierarchical Report Must be associated with a worksheet range or a LabTalk Tree variable	1. <new> 2. [<input>]<new> 3. [book2]sheet3	
ReportData	A Tree based object for a collection of vectors Must be associated with a worksheet range or a LabTalk Tree variable. Unlike ReportTree, ReportData outputs to a regular worksheet and thus can be used to append to the end of existing data in a worksheet. All the columns in a ReportData object must be grouped together.	1. <new> 2. [<input>]<new> 3. [book2]sheet3 4. [<input>]<input>!<new>	

To understand these variable types better, please refer to the real examples in the **ReportData Output** section below, which have shown some concrete usages.

Special Keywords for Range

<new>
Adding/Creating a new object
<active>
Use the active object
<input>
Same as the input range in the same X-Function
<same>
Same as the previous variable in the X-Function
<optional>
Indicate the object is optional in input or output
<none>
No object will be created

ReportData Output

Many X-Functions generate multiple output vectors in the form of a **ReportData** object. Typically, a ReportData object is associated with a worksheet, such as the **Fit Curves** output from the **NLFit** X-Function. Consider, for example, the output from the **fft1** X-Function:

```
// Send ReportData output to Book2, Sheet3.
fft1 rd:=[book2]sheet3!;
// Send ReportData output to a new sheet in Book2.
fft1 rd:=[book2]<new>!;
// Send ReportData output to Column 4 in the active workbook/sheet.
fft1 rd:=[<active>]<active>!Col(4);
// Send ReportData output to a new sheet in the active workbook.
fft1 rd:=[<active>]<new>!;
// Send ReportData output to a tree variable named tr1;
// If 'tr1' does not exist, it will be created.
fft1 rd:=tr1;
```

Sending ReportData to Tree Variable

Often, you may need the ReportData output only as an intermediate variable and thus may prefer not to involve the overhead of a worksheet to hold such data temporarily.

One alternative then is to store the datasets that make up the Report Data object using a Tree variable, which already supports bundling of multiple vectors, including support for additional attributes for such vectors.

The output range specification for a worksheet is usually in one of the following forms: **[Book]Sheet!**, **<new>**, or **<active>**. If the output string does not have one of these usual book-sheet specifications, then the output is automatically considered to be a LabTalk Tree name.

The following is an example featuring the **avecurves** X-Function. In this example, the resulting ReportData object is first output to a tree variable, and then one vector from that tree is placed at a specific column-location within the same sheet that houses the input data. ReportData output typically defaults to a new sheet.

```
int nn = 10;
col(1)=data(1,20);  //fill some data
loop(i,3,nn){wcol(i)=normal(20);};
range ay=col(2); //for 'avecurves' Y-output
Tree tr; // output Tree
avecurves (1,3:end) rd:=tr;
// Assign tree node (vector) 'aveY' to the range 'ay'.
// Use 'tr.=' to see the tree structure.
ay=tr.Result.aveY;
ay[L]$="Ave Y"; // set its LongName
// Plot the raw data as scatter-plot using the default-X.
plotxy (?,3:end) p:=201;
// Add the data in range 'ay' to the same as line-plot.
plotxy ay o:=<active> p:=200;
```

Sending ReportData Directly to a Specific Book/Sheet/Column Location

If you are happy with simply putting the result from the X-Function into the input sheet as new columns, then you can also do the following:

```
avecurves (1,2:5) rd:=[<input>]<input>!<new>;
```

Or if you would like to specify a particular column of the input sheet in which to put the ReportData output, you may specify that as well:

```
avecurves (1,2:5) rd:=[<input>]<input>!Col(3);
```

Subsequent access to the data is more complicated, as you will need to write additional code to find these new columns.

Realize that output of the ReportData type will contain different amounts (columns) of data depending on the specific X-Function being used. If you are sending the results to an existing sheet, be careful not to overwrite existing data with the ReportData columns that are generated.

5.1.3 X-Function Execution Options

X-Function Option Switches

The following option switches are useful when accessing X-Functions from script:

Switch	Full Name	Function
-cf	--	Copy column format of the input range, and apply it to the output range.
-d	-dialog	Brings up a dialog to select X-Function parameters.
-db	--	Variation of dialog; Brings up the X-Function dialog as a panel in the current workbook.
-dc *IsCancel*	--	Variation of dialog; Brings up a dialog to select X-Function parameters. Set *IsCancel* to 0 if click the **OK** button, set to 1 if click the **Cancel** button. When clicking the **Cancel** button, no error message like **#User Abort!** dumps to Script Window and the script after X-Function can be executed.
-h	-help	Prints the contents of the help file to the Script window.
-hn	--	Loads and compiles the X-Function without doing anything else. If the X-Function has already been compiled and loaded, it will do nothing.
-hs	--	Variation of -h; Prints only the Script Usage Examples.
-ht	--	Variation of -h; Prints only the Treenode information, if any exists.
-hv	--	Variation of -h; Prints only the Variable list.
-hx	--	Variation of -h; Prints only the related X-Function information.
-r 1	-recalculate 1	Sets the output to automatically recalculate if input changes.
-r 2	-recalculate 2	Sets the output to recalculate only when manually prompted to do so.
-s	-silent	Runs in silent mode; results are not sent to Results log.
-sb	--	Variation of -s; suppresses error messages and Results

		log output.
-se	--	Variation of -s; suppresses error messages, does not suppress Results log output.
-sl	-silent	Same as -s.
-ss	--	Variation of -s; suppresses info messages to the script window.
-t <*Name*>	-theme	Uses the designated preset theme.

Recalculate is not supported when <input> is used an an <output>.

For options with an existing **Full Name**, either the shortened switch name or the full name may be used in script. For instance, for the X-Function **smooth**,

```
smooth -h
```

is the same as

```
smooth -help
```

Examples

Using a Theme

Use the theme named **FivePtAdjAve** to perform a smoothing operation on the XY data in columns 1 and 2 of the active worksheet.

```
smooth (1,2) -t FivePtAdjAve
```

Note: A path does not need to be specified for the theme file since Origin automatically saves and retrieves your themes. Themes saved in one project (*.OPJ) will be available for use in other projects as well.

Setting Recalculate Mode

Set the output column of the **freqcounts** X-Function to automatically recalculate when data in the input column changes.

```
freqcounts irng:=col(1) min:=0 max:=50 stepby:=increment inc:=5
   end:=0 count:=1 center:=1 cumulcount:=0 rd:=col(4) -r 1;
// Set Recalculate to Auto with '-r 1'.
```

Open X-Function Dialog

While running an X-Function from script it may be desirable to open the dialog to interactively supply input. In this simple example, we perform a smoothing operation using a percentile filter (method:=2) and specifying a moving window width of 25 data points. Additionally, we open the dialog (-d) associated with the smooth X-Function allowing the selection of input and output data, among other options.

```
smooth method:=2 npts:=25 -d
```

Copy Format from Input to Output

Use an FFT filter with the **-cf** option switch to format the output data to match that of the input data:

```
// Import a *.wav file; imported *.wav data format is short(2).
fname$ = system.path.program$ + "Samples\Signal Processing\sample.wav";
newbook s:=0; newsheet col:=1; impWav options.SparkLines:=0;
string bkn$=%H;

// By default, all analysis results are output as datatype double.
// -cf is used here to make sure the output data to be short(2)
fft_filters -cf [bkn$]1!col(1) cutoff:=2000
oy:=(<input>,<new name:="Lowpass Sound Frequency">);
```

5.1.4 X-Function Exception Handling

The example below illustrates trapping an X-Function error with LabTalk, so that an X-Function call that is likely to generate an error does not break your entire script.

For X-Functions that ***do not*** return an error code, two functions exist to check for errors in the last executed X-Function: **xf_get_last_error_code()** and **xf_get_last_error_message()$**. These functions should be used in situations where the potential exists that a particular X-Function could fail.

In this example, the user is given the option of selecting a file for import, but if that import fails (e.g. user picked file type inappropriate for the import) we need to handle the remaining code.

```
dlgfile gr:=*.txt;  // Get the file name and path from user
impasc -se;  // Need to use -se switch for execution to continue, see note below
if( 0 != xf_get_last_error_code() )
{
    strError$ = "XFunction Failed: " + xf_get_last_error_message()$;
    type strError$;
    break 1;  // Stop execution
}
// Data import probably succeeded, so our script can continue
type continuing...;
```

Note the use of the general X-Function option **-se** to suppress error messages. You can also use **-sl** to suppress error logging and **-sb** to suppress both. It is necessary to use one of these options in order for script execution to continue to the next line when the X-Function call fails.

Looping Over to Find Peaks

In the following example, we loop over all columns in a worksheet to find peaks. If no peak is found in a particular column, the script continues with the rest of the columns. It is assumed here that a worksheet with suitable data is active.

```
for(int ii=2; ii<=wks.ncols; ii++)
{
  // Find peak in current column, suppress error message from XF
  Dataset mypeaks;
  pkfind $(ii) ocenter:=mypeaks -se; // Need to use -se for execution to continue

  // Check to see if XF failed
  if( 0 != xf_get_last_error_code() )
  {
    type "Failed on column $(ii): %(xf_get_last_error_message()$)";
  }
  else
  {
    type Found $(mypeaks.getsize()) peaks in column $(ii);
  }
}
```

5.2 Origin C Functions

The following subsections detail how to call Origin C functions from your LabTalk scripts.

5.2.1 Loading and Compiling Origin C Functions

Loading and Compiling Origin C Function or Workspace

Before you call your Origin C function from Origin, your function must be compiled and linked in the current Origin session. To programmatically compile and link a source file, or to programmatically build a workspace from a LabTalk script use the run.loadOC method of the LabTalk run object.

```
    err = run.LoadOC("myFile",[option]);
```

Example

Use option to scan the .h files in the OC file being loaded, and all other dependent OC files are also loaded automatically:

```
// Load and compile Origin C function in the file iw_filter.c
// with the option=16, so to also load all dependent Orign C
// files by scanning for .h files included in iw_filter.c
```

```
if(run.LoadOC(OriginLab\iw_filter.c, 16) != 0)
{
   type "Failed to load iw_filter.c!";
   return 0;
}
```

Now, open **Code Builder** by menu **View: Code Builder**, and in the **Workspace** panel (if not see this panel, open by **View: Workspace** menu item in **Code Builder**) of **Code Builder**, you can see the *iw_filter.c* file is under the **Temporary** folder.

Adding Origin C Source Files to System Folder

Once a file has been opened in Code Builder, one can simply drag and drop the file to the **System** branch of the Code Builder workspace. This will then ensure that the file will be loaded and compiled in each new Origin session. For more details, please refer to Code Builder documentation.

You can programmatically add a source file to the system folder so that it will be available anytime Origin is run.

```
run.addOC(C:\Program Files\Originlab\Source Code\MyFunctions.c);
```

This can be useful when distributing tools to users or making permanently available functions that have been designed to work with Set Column Values.

Adding Origin C Files to Project (OPJ)

Origin C files (or files with any extension/type) can also be appended to the Origin project (OPJ) file itself. The file will then be saved with the OPJ and extracted when the project is opened. In case of Origin C files, the file is then also compiled and linked, and functions within the file are available for access. To append a file to the project, simply drag and drop the file to the **Project** branch of Code Builder or right-click on Project branch and add the file. For more details, please refer to Code Builder documentation.

5.2.2 Passing Variables To and From Origin C Functions

When calling a function of any type it is often necessary to pass variables to that function and likewise receive variables output by the function. The following summarizes the syntax and characteristics of passing LabTalk variables to Origin C functions.

Sytnax for calling Origin C Function from LabTalk

Origin C functions are called from LabTalk with sytnax such as:

```
// separate parameters by commas (,) if more than one
int iret = myfunc(par1, par2....);

// no need for parentheses and comma if there is no assignment
myfunc par1;
```

```
// function returns no value, and no parameter, parentheses optional
myfunc;
```

Variable Types Supported for Passing To and From LabTalk

The following table lists Origin C variable types that can be passed to and from LabTalk when calling an Origin C Function:

Variable Type	Argument to OC Function	Return from OC Function
double	Yes	Yes
int	Yes	Yes
bool (true or false)	No, pass **int** instead, 0 for false, and other integer for true.	No, return **int** instead, 0 for false, 1 for true.
string	Yes	Yes
int, double array	Yes	Yes
string array	Yes, but cannot pass by reference	Yes

Note:

1. The maximum number of arguments that Origin C function can have to call from LabTalk is 80.
2. LabTalk variables can be passed to Origin C numeric function by reference.

5.2.3 Updating an Existing Origin C File

Introduction

There are cases where a group leader or a developer wants to release a new version of an Origin C file to other Origin users. In such cases, if the end users have already installed an older version of the Origin C file, they will have a corresponding .OCB file in their User Files Folder (UFF). It is possible that the time stamp of the new Origin C file is older than the time stamp of the .OCB file. When this happens Origin will think the .OCB file is already updated and will not recompile the new Origin C file. To avoid this possible scenario it is best to delete the .OCB file when the new Origin C file is installed. Once deleted, Origin will be forced to remake the .OCB file and will do so by compiling the new Origin C file.

Manually Deleting OCB Files

The OCB file corresponding to the Origin C file in question, can be manually deleted from the OCTEMP folder in the Users Files Folder on the end user's computer. Depending on the location of the Origin C file, it is possible for the OCB file to be in nested subfolders within the OCTemp folder. Once located, the end user can delete the OCB file and rebuild their workspace to create an updated OCB file.

Programmatically Deleting OCB Files

A group leader or developer can programmatically delete the corresponding OCB files using LabTalk's Delete command with the OCB option. This is very useful when distributing Origin C files in an Origin package and it is not acceptable to have the end user manually delete the .OCB files.

Below are some examples of how to call LabTalk's Delete command with the OCB option:

```
del -ocb filepathname1.c
del -ocb filepathname1.ocw
del -ocb filepathname1.c filepathname2.c // delete multiple files
del -ocb %YOCTEMP\filename.c // use %Y to get to the Users Files Folder
```

5.2.4 Using Origin C Functions

To extend the functions, you can also define an Origin C function (see Creating and Using Origin C Code for details) which returns a single value, and call the function from command window. For example,

1. Open **Code Builder** by menu **View: Code Builder**.
2. In **Code Builder**, create a new *.c file by menu **File: New**. In the **New File** dialog, give a file name, *MyFuncs* for example, and click OK.
3. Start a new line at the end of this new file, and add the following code.

```
double MyFunc (double x)
{
    return sin(x) + cos(x);
}
```

4. Click menu item **Build: Build** to compile and link the file.
5. If no error, the function defined above is now available in LabTalk. Run the following script in the Command Window.

```
newbook;  // create a new workbook
col(A) = data(1, 32);  // fill row number
col(B) = MyFunc(col(A));  // call the Origin C function, result is put to
column B
```

6 Running and Debugging LabTalk Scripts

Origin provides several options for executing and storing LabTalk scripts. The first part of this chapter profiles these options. The second part of the chapter outlines the script debugging features supported by Origin.

6.1 Running Scripts

The following section documents 11 ways to execute and/or store LabTalk scripts. But first, it is important to note the relationship between scripts and the objects they work on.

Active Window Default

When working on an Origin Object, like a workbook or graph page, a script always operates on the active window by default. If the window is inactive, you may use win -a to activate it.

```
win -a book2;              // Activate the window named book2
col(b) = {1:10};  // Fill 1 to 10 on column B of book2
```

However, working on active windows with *win -a* may not be stable. In the execution sequence of the script, switching active windows or layers may have delay and may lead to unpredictable outcome.

It is preferable to always use win -o *winName* {*script*} to enclose the script, then Origin will temporarily set the window you specified as the active window (internally) and execute the enclosed script exclusively on that window. For example, the following code will create a new project, fill the default book with some data, and make a plot and then go back to add a new sheet into that book and make a second plot with the data from the second sheet:

```
doc -s;doc -n;//new project with default worksheet
string bk$=%H;//save its book short name
//fill some data and make new plot
wks.ncols=2;col(1)=data(1,10);col(2)=normal(10);
plotxy (1,2) o:=<new>;
//now the newly created graph is the active window
//but we want to run some script on the original workbook
win -o bk$ {
  newsheet xy:="XYY";
  col(1)=data(0,1,0.1);col(2)=col(1)*2;col(3)=col(1)*3;
  plotxy (1,2:3) plot:=200 o:=<new>;
}
```

Please note that *win -o* is the only LabTalk command that allows a string variable to be used. As seen above, we did not have to write

```
win -o %(bk$)
```

as this particular command is used so often that it has been modified since Origin 8.0 to allow string variables. In all other places you must use the %( ) substitution notation if a string variable is used as an argument to a LabTalk command.

Where to Run LabTalk Scripts

While there are many places in Origin that scripts can be stored and run, they are not all equally likely. The following sub-sections have been arranged in an assumed order of prevalence based on typical use.

The first two, on (1) Running Scripts from the Script and Command Windows and (2) Running Scripts from Files, will be used much more often than the others for those who primarily script. If you only read two sub-sections in this chapter, it should be those. The others can be read on an as-needed basis.

6.1.1 From Script and Command Window

Two Windows exist for direct execution of LabTalk: the (older) Script Window and the (newer) Command Window. Each window can execute single or multiple lines of script. The Command Window has a prompt and will execute all code entered at the prompt.

The Script Window has only a cursor and will execute highlighted code or code at the current cursor position when you press Enter. Both windows accept Ctrl+Enter without executing. When using Ctrl+Enter to add additional lines, you must include a semicolon **;** at the end of a statement.

The Command Window includes Intellisense for auto-completion of X-Functions, a command history and recall of line history (Up and Down Arrows) while the Script Window does not. The Script Window allows for easier editing of multiline commands and longer scripts.

Below is an example script that expects a worksheet with data in the form of one X column and multiple Y columns. The code finds the highest and lowest Y values from all the Y data, then normalizes all the Y's to that range.

```
// Find the lowest minimum and the highest maximun
double absMin = 1E300;
double absMax = -1E300;
loop(ii,2,wks.ncols)
{
    stats $(ii);
    if(absMin > stats.min) absMin = stats.min;
    if(absMax < stats.max) absMax = stats.max;
}
// Now normalize each column to that range
loop(ii,2,wks.ncols)
{
    stats $(ii);
    wcol(ii)-=stats.min;                // Shift to minimum of zero
    wcol(ii)/=(stats.max - stats.min);  // Normalize to 1
```

```
    wcol(ii)*=(absMax - absMin);          // Normalize to range
    wcol(ii)+=absMin;                     // Shift to minimum
}
```

To execute in the Script Window, paste the code, then select all the code with the cursor (selected text will be highlighted), and press Enter.

To execute the script in the Command Window, paste the code then press Enter. Note that if there were a mistake in the code, you would have it all available for editing in the Script Window, whereas the Command Window history is not editable and the line history does not recall the entire script.

Origin also has a native script editor, Code Builder, which is designed for editing and debugging both LabTalk and Origin C code. To access Code Builder, enter **ed.open()** into the script or command window, or select the  button from the Standard Toolbar.

6.1.2 From Files

LabTalk script usually requires an Origin Object and are thus restricted to an open project. Scripts can also be saved to a file on disk to be called from any project. Script files can be called with up to five arguments. This section outlines the use of LabTalk scripts saved to a file.

Creating and Saving Script Files

LabTalk scripts can be created and saved from any text editor, including Origin's Code Builder.

To access Code Builder, select the  icon from the Standard Toolbar. Create a new document of type LabTalk Script File and type or paste your code into the editor window and then save with a desired filename and path (use the default OGS file extension).

The OGS File Extension

LabTalk scripts can be saved to files and given any extension, but for maximum flexibility they are given the OGS file extension, and are therefore also known as **OGS files**. You may save script files to any accessible folder in your file system, but specific locations may provide additional advantages. If an OGS file is located in your User Files Folder, you will not have to provide a path when running your script.

An OGS file can also be attached to the Origin Project (OPJ) rather than saving it to disk. The file can be added to the **Project** node in Code Builder and will then be saved with the project. Drag the filename from the User folder and drop into the Project folder. Script sections in such attached OGS files can be called using the **run.section()** object method similar to calling sections in a file saved on disk. Only the file name needs to be specified, as Origin will first look for the

file in the project itself and execute the code if filename and section are found as attachments to the project.

Sections in an OGS File

Script execution is easier to follow and debug when the code is written in a modular way. To support modular scripting, LabTalk script files can be divided into sections, which are declared by placing the desired section name in square brackets [ ] on its own line:

[*SectionName*]

Lines of script under the section declaration belong to that section. Execution of LabTalk in a section ends when another section declaration is met, when a return statement is executed or when a Command Error occurs. The framework of a typical multi-section OGS file might look like the following:

```
[Main]
// Script Lines
ty In section Main;

[Section 1]
// Script Lines
ty In section 1;

[Section 2]
// Script Lines
ty In section 2;
```

Note here that **ty** issues the **type** command, which is possible since no other commands in Origin begin with the letters 'ty'.

Running an OGS File

You can use the **run** object to execute script files or in certain circumstances LabTalk will interpret your file name as a **command** object. To use a file as a command object, the file extension must be OGS. See the **Note** below for additional information.

Compare the following call formats:

run.section(OGSFileName, SectionName[,arg1 arg2 ... arg5])

run.file(OGSFileName[arg1 arg2 ... arg5])

OGSFileName.SectionName [arg1 arg2 ... arg5]

OGSFileName [arg1 arg2 ... arg5]

Specifically, if you save a file called **test.ogs** to your Origin User Files folder:

```
// Runs [Main] section of test.ogs using command syntax, else runs
// unsectioned code at the beginning of the file, else does nothing.
test;
```

```
// Runs only section1 of test.ogs using command syntax:
test.section1;

// Runs only section1 of test.ogs with run.section() syntax:
run.section(test, section1)
```

Note: After saving the OGS file, you need to run the **cd** X-Function (`cd 1;`) to change to the folder where the file was saved or use **dir** to list files in the *current working folder* - if that is where the file is. Otherwise, Origin does not detect the file and will not see it as a runnable command.

After Origin recognizes an OGS filename as an object, run the OGS file by entering its name or name.sectionname into the Script Window or Command Window. If either the file name or section name contains a space quotes must surround both. For example:

```
// Run a LabTalk Script named 'My Script.ogs' located in the folder
//'D:\OgsFiles'.

// Change the current directory to 'D:\OgsFiles'
cd D:\OgsFiles; // This causes Origin to scan that folder for OGS files
// This runs the code in section 'Beta Test' of 'My Scripts.ogs'
// passing three arguments separated by spaces (protected by quotes where needed)
"My Scripts.Beta Test" "Redundant Test" 5 "Output Averages";
```

There are many examples in Origin's **Samples\LabTalk Script Examples** folder which can be accessed by executing:

```
cd 2;
```

Passing Arguments in Scripts

When you use the **run.section()** object method to call a script file (or one of its sections) or a macro, you can pass arguments with the call. Arguments can be literal text, numbers, numeric variables, or string variables.

When passing arguments to script file sections or to macros:

- The section call or the macro call must include a space between each argument being passed. When using run.section, a comma must separate the section name from the first argument only.
- When you pass literal text or string variables as arguments, each argument must be surrounded by quotation marks (in case the argument contains more than one word, or is a negative value). Passing numbers or numeric variables doesn't require quotation mark protection, except when passing negative values.
- You can pass up to five arguments, separated by **Space**, to script file sections or macros. In the script file section or macro definition, argument placeholders receive the passed arguments. These placeholders are %1, %2, %3, %4, and %5. The placeholder for the first passed argument is %1, the second is %2, etc. These placeholders work like string variables in that they are substituted prior to execution

of the command in which they are embedded. The number of arguments passed is contained in *macro.narg*.

As an example of passing literal text as an argument that is received by %1, %2, etc., Suppose a TEST.OGS file includes the following section:

```
[output]
type "%1 %2 %3";
```

and you execute the following script:

```
run.section(test.ogs, output, "Hello World" from LabTalk);
```

Here, %1 holds "Hello World", %2 holds "from", and %3 holds "LabTalk". After string substitution, Origin outputs

```
Hello World from LabTalk
```

to the Script window. If you had omitted the quotation marks from the script file section call, then %1 would hold "Hello", %2 would hold "World", and %3 would hold "from". Origin would then output

```
Hello World from
```

Passing Numeric Variables by Reference

Passing numeric variable arguments by reference allows the code in the script file section or macro to change the value of the variable.

For example, suppose your application used the variable **LastRow** to hold the row number of the last row in column B that contains a value. Furthermore, suppose that the current value of **LastRow** is 10. If you pass the variable **LastRow** to a script file section whose code appends five values to column B (starting at the current last row), after appending the values, the script file section could increment the value of the **LastRow** variable so that the updated value of **LastRow** is 15.

See example:

If a TEST.OGS file includes the following section:

```
[adddata]
   loop (n, 1, 5)
   {
       %1[%2 + n] = 100;
   };
   %2 = %2 + (n - 1);
   return 0;
```

And you execute the following script:

```
col(b) = data(1, 10);   // fill data1_b with values
get col(b) -e lastrow;  // store last row of values in lastrow
run.section(test.ogs, adddata, col(b) lastrow);
lastrow = ;
```

Then **LastRow** is passed by reference and then updated to hold the value 15.

Passing Numeric Variables by Value

Passing numeric variable arguments by value is accomplished by using the $() substitution notation. This notation forces the interpreter to evaluate the argument before sending it to the script file section or macro. This technique is useful for sending the value of a calculation for future use. If the calculation were sent by reference, the entire expression would require calculation each time it was interpreted.

In the following script file example, numeric variable *var* is passed by reference and by value. %1 will hold the argument that is passed by reference and %2 will hold the argument that is passed by value. Additionally, a string variable (%A) consisting of two words is sent by value as a single argument to %3.

```
[typing]
   type -b "The value of %1 = %2 %3";
   return 0;
```

Save the section to *Test.OGS* and run the following script on command window:

```
var = 22;
%A = "degrees Celsius";
run.section(test.ogs, typing, var $(var) "%A");
```

Then a dialog box pop-up and says: "The value of var = 22 degrees Celsius".

Guidelines for Naming OGS Files and Sections

Naming rules for OGS script files differ based on how they will be called. The section above discusses the two primary methods: calling using the **run.section()** method or calling directly from the Script or Command window (the **command** method).

When Using the Run.section() Method

- There is no restriction on the length or type of characters used to name the OGS file.
- Specifying the filename extension is optional for files with the OGS extension.
- When using run.section( ) inside an OGS file to call another section of that same OGS file, the filename may be omitted, for instance:

```
[main]
run.section( , calculate);

[calculate]
cc = aa + bb;
```

When Using the Command Method

- The name of the OGS file must conform to the restrictions on command names: 25 characters or fewer, must not begin with a number or special character, must not contain spaces or underscore characters.
- The filename extension *must* be OGS and must *not* be specified.

Section Name Rules (When Using Either Method)

- When SectionName is omitted,
    1. Origin looks for a section named **main** and executes it if found
    2. If no **main** section is found, but code exists at the beginning of the file without a section name, then that code is executed
    3. Otherwise Origin does nothing and does not report an error

Do not give an OGS file the same name as an existing Origin function or X-Function!

Setting the Path

In Origin 7.5, script files (*.OGS) could be run from both the **Origin System** and **User Files** folders, and these are the current working directory by default. If your script file resides there, there is no need to change the path. If the script file was not located in either of these two folders, the full path needed to be specified in the run.section() object method. Since Origin 8, the idea of the Current Working Folder (CWF) was introduced, allowing you to run your own script files and X-Functions located in the CWF you have specified.

Per MS-DOS convention, Origin uses the cd X-Function to display the CWF:

```
// Entering this command displays the current working folder
// in the Script Window.
cd
```

and unless it has been changed, the output is similar to:

```
current working directory:
C:\Documents and Settings\User\My Documents\OriginLab\Origin8.1\User Files\
```

However, if you write many scripts, you will want to organize them into folders, and call these scripts from where they reside. Also, Origin provides sample scripts that you may want to run from their respective directories.

In the case or run.section( ) scripts can reside in subfolders of the User Files Folder and you can use relative referencing such as:

```
run.section(subfolder1\scriptA,main); // ScriptA.ogs is in subfolder1
run.section(subfolder2\scriptA,main); // ScriptA.ogs is in subfolder2
```

You can set the Current Working Folder from script. For example, to run an OGS file named **ave_curves.ogs**, located in the Origin system sub-folder **Samples\LabTalk Script Examples**, enter the following:

```
// Create a string variable to hold the complete path to the desired
//script file
// by appending folder path to Origin system path:
path$ = system.path.program$ + "Samples\LabTalk Script Examples\";
// Make the desired path the current directory.
cd path$;
```

```
// Call the function
ave_curves;
```

You can create a set of pre-defined paths. The cdset X-Function is used to list all the pre-defined paths and add/change the CWF. By typing

```
// The 'cdset' command displays pre-defined paths
//in the Script Window.
cdset
```

you should see three paths like below if you have not changed them yet.

```
1 = C:\Documents and Settings\User\My Documents\OriginLab\Origin8.1\User Files\
2 = C:\Program Files\OriginLab\Origin81\Samples\LabTalk Script Examples\
3 = C:\Program Files\OriginLab\Origin81\
```

If you want to set the second path above to be the CWF, just type:

```
// Changes the CWF to the folder path specified
// by pre-defined path #2.
cd 2
```

To add a new path to pre-defined folder set, first change to the new path, making it the CWF, then add it to the set by using **cdset** X-Function with the specified index. For example:

```
cd D:\Files\Filetype\Script;  // Set this new path as CWF
// Add this path to pre-defined folder list, to the 4th postion (index 4)
// If there already is a path with index 4, it will be over-written
cdset 4;
// If the CWF is changed manually, it can now be reset to
// 'D:\Files\Filetype\Script\' by entering 'cd 4'.
```

A few tips for working with the **cdset** command:

- Folder paths added to the pre-defined set in one project are saved for use with other projects.
- To see the current paths displayed to the Script Window, enter 'cdset' by itself on a line in the Script Window.
- Up to 9 pre-defined paths are supported.
- Indices can be assigned out of order.
- A new path, assigned to an index for which a current path exists, will overwrite the current path.

As the three default pre-defined paths show above, the second one contains several sample script files (with the **OGS** extension). Similar to DOS, you can go to this folder by *cd 2*, then see the valid OGS using the dir X-Function, and then run any available script file in this folder, such as:

```
// Set 2nd folder as the CWF
cd 2;
// List all ogs and X-Function in the CWF
dir;
// Run a script file
```

```
// Note that the file extension is not needed when calling it
autofit;
```

You can also load the script file in the CWF into Code Builder by using ed.open() method. Such as:

```
// In this case, the OGS extension on the filename is required!
ed.open(pick_bad_data.ogs)
```

Running LabTalk from Origin C

Besides running .OGS files directly, LabTalk commands and scripts can also be run from Origin C. For more information, please refer to LabTalk Interface global function of Origin C help document.

6.1.3 From Set Values Dialog

The Set Values Dialog is useful when calculations on a column of data are based on functions that may include references to other datasets.

The column designated by **Set Values** is filled with the result of an expression that you enter (the expression returns a dataset). The expression can be made to update automatically (Auto), when requested by the user (Manual), or not at all (None).

For more complex calculations, where a single expression is not adequate, a Before Formula Scripts panel in the dialog can include any LabTalk script.

Auto and Manual updates create lock icons, and respectively, at the top of the column. A green lock indicates updated data; A yellow lock indicates pending update; A red lock indicates broken functionality.

In cases where the code is self-referencial (i.e. the column to be set is included in the calculation) the Auto and Manual options are reset to None.

Below are two examples of script specifically for the Set Values Dialog. Typically short scripts are entered in this dialog.

Expression using another column

While limited to expressions (the right side of an equation) as in:

```
// In column 3
// Scale a column - useful for fitting where very large
//or very small numbers are problematic
col(2)*1e6;
```

the conditional expression can be useful in some situations:

```
// Set negative values to zero
col(2)<0?0:col(2);
```

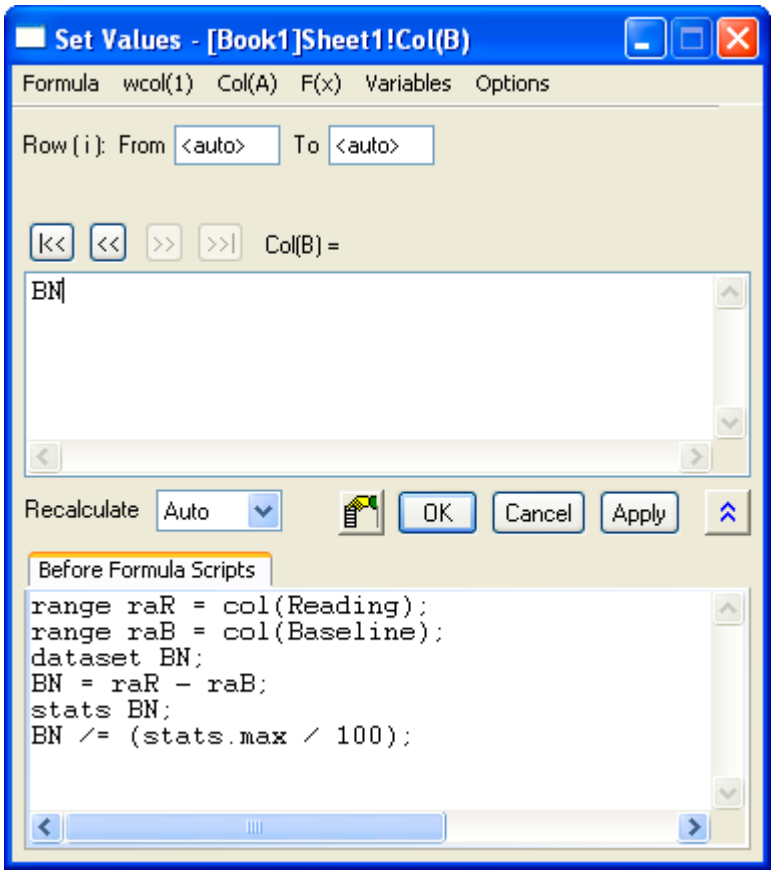
Using Before Formula Scripts Section

In the **Before Formula Scripts** section of the **Set Column Values** dialog, a script can be entered that will be executed by Origin just before the formula itself is executed. This feature is

useful for carrying out operations that properly setup the formula itself. The following example demonstrates the use of such a script:

```
// In column BaseNormal
// In the expression section ..
BN
// In the Before Formula Scripts section ..
range raR = col(Reading);   // The signal
range raB = col(Baseline);  // The Baseline
dataset BN;
BN = raR - raB;             // Subtract the baseline from the signal
stats BN;                   // Get statistics of the result
BN /= (stats.max / 100);    // Normalize to maximum value of 100
```

The following image is a screenshot of the code above entered into the **Set Column Values** dialog:



6.1.4 From Worksheet Script

The Worksheet Script dialog is mostly provided for backward compatibility with older versions of Origin that did not have the Auto Update feature in the **Set Values** dialog and/or did not have import filters where scripts could be set to run after import.

Scripts can be saved in a Worksheet (so workbooks have separate Worksheet Scripts for each sheet) and set to run after either importing into this Worksheet and/or changes in a particular dataset (even one not in this Worksheet).

Here is a script that is attached to Sheet3 which is set to run on Import (into Sheet3) or when the A column of Sheet2 changes.

```
range ra1 = Sheet1!1;
range ra2 = Sheet1!2;
range ra3 = Sheet2!A; // Our 'Change in Range' column
range ra4 = 3!2;      // Import could change the sheet name ..
range ra5 = 3!3;      // .. so we use numeric sheet references
ra5 = ra3 * ra2 / ra1 * ra4;
```

6.1.5 From Script Panel

The Script Panel (accessed via the context menu of a Workbook's title bar) is a hybrid of the Script Window and Command Window.

- Like the Script Window, it can hold multiple lines and you can highlight select lines and press Enter to execute.
- Like the Command Window, there is a history of what has been executed.
- Unlike the Script window, whose content is not saved when Origin closes, these scripts are saved in the project.

```
// Scale column 2 by 10
col(2)*=10;

// Shift minimum value of 'mV' column to zero
stats col(mV);
col(mV)-=stats.min;

// Set column 3 to column 2 normalized to 1
stats 2;
col(3) = col(2)/stats.max;
```

6.1.6 From Graphical Objects

Graphic Objects (text, lines and shapes) can be tied to events and contain script that runs on those events. Since graphical objects are attached to a page, they are saved in Templates, Window files and Project files.

Buttons

Some of your scripts may be used so often that you would like to automate their execution by assigning one or more of them to a button on the Origin project's graphical-user interface (GUI). To do so, follow the steps below:

From a new open Origin project:

1. Select the text tool from the tool menu on the left side of the project window --> 
2. Now click in the open space to the right of the two empty columns of the blank worksheet in the **Book1** window. This will open a text box. Type "Hello" in the text box and press enter--you have now created a label for the button.
3. Now hold down the ALT key while double-clicking on the text that you just created. A window called **Programming Control** will appear.
4. In the lower text box of the **Programming Control** window, again type our script text exactly:

   ```
   type -b "Hello World";
   ```

5. Also in the **Programming Control** window, in the **Script, Run After** box, select **Button Up**, and click **OK**.
6. You have now created a button that, when pressed, executes your script and prints "Hello World" in a pop-up window.

Unlike a text script which exists only in the **Classic Script Window**, this button and the script it runs will be saved when you save your Origin project.

Lines

Here is a script that creates a vertical line on a graph that can be moved to report the interpolated value of your data at the X position represented by the line:

```
// Create a vertical line on our graph
draw -n MyCursor -l -v $(x1+(x2-x1)/2);
MyCursor.HMOVE = 1;               // Allow horizontal movement
MyCursor.color = color(orange);   // Change color to orange
MyCursor.linewidth = 3;           // Make line thicker
// Add a label to the graph
label -sl -a $(MyCursor.x) $(Y2+0.05*(Y2-Y1)) -n MyLabel $(%C(MyCursor.x));
// Assign a script to the line ..
MyCursor.script$="MyLabel.x = MyCursor.x;
MyLabel.y = Y2 + MyLabel.dy;
doc -uw;";
// .. and make the script run after the line is moved
MyCursor.Script = 2;
```

Other Objects

Any Graphical Object (text, lines and shapes) can have an attached script that runs when an event occurs.

In this example, a rectangle (named RECT) on a graph is set to have a script run when the rectangle is either Moved or Sized.

1. Use the Rectangle tool on the **Tools** toolbar to draw a rectangle on a graph.
2. Use the Back(data) tool on the **Object Edit** toolbar to push the rectangle behind the data.
3. Hold down the Alt key and double-click on the rectangle to open **Programming Control**.
4. Enter the following script:

```
%B = %C;
%A = xof(%B);
dataset dsRect;
dsRect = ((%A >= rect.x1) && (%A <= rect.x2) &&
          (%B >= rect.y3) && (%B <= rect.y1))?%B:0/0;
stats dsRect;
delete dsRect;
type -a Mean of $(stats.mean);
```

5. Choose the **Moved or Sized** event in the Script, Run After drop down list.
6. Click OK.

When you Move or Resize this rectangle, the script calculates the mean of all the points within the rectangle and types the result to the Script Window.

6.1.7 ProjectEvents Script

You may want to define functions, perform routine tasks, or execute a set of commands, upon opening, closing, or saving your Origin project. In Origin 8.1 a file named ProjectEvents.ogs is attached to the Origin Project (OPJ) by default.

A template version of this file is shipped with Origin and is located in the **EXE** folder. This template file is attached to each new project. The file can be viewed and edited by opening **Code Builder** and expanding the **Project** node in the left panel.

Sections of ProjectEvents.ogs

The **ProjectEvents.ogs** file, by default, contains three sections that correspond to three distinct events associated with the project:

1. AfterOpenDoc: This section will be executed immediately after the project is opened
2. BeforeCloseDoc: This section will be executed before the project is closed
3. BeforeSaveDoc: This section will be executed before the project is saved

Utilizing ProjectEvents.ogs

In order for this file and its contents to have an effect, a user needs to edit the file and save it in Code Builder, and then save the project. The next time the project is opened, the script code contained in this attached OGS file will be executed upon the specified event (given by the pre-defined section name).

For example, if a user defines a new function in the [AfterOpenDoc] section of **ProjectEvents.ogs**, saves it (in Code Builder), and then saves the project in Origin, that function will be available (if defined to be global) for use any time the project is re-opened. To make the function available in the current session place the cursor (in Code Builder) on the section name to execute, select the **Debug** drop-down menu, and select the **Execute Current Section** option. Then in the Origin Script Window, issuing the **list a** command will confirm that the new function appears and is available for use in the project.

A brief tutorial in the Functions demonstrates the value of **ProjectEvents.ogs** when used in conjunction with LabTalk's dataset-based functions.

You can add your own sections to this OGS file to save custom routines or project-specific script code. Such sections will not be event-driven, but can be accessed by name from any place that LabTalk script can be executed. For example, if you add a section to this file named [MyScript], the code in that section can be executed after opening the project by issuing this command from the script window:

```
run.section(projectevents,myscript);
```

A ProjectEvents.ogs script can also be made to run by opening the associated Origin Project (OPJ) from a command console external to Origin.

6.1.8 From Import Wizard

The Import Wizard can be used to import ASCII, Binary or custom file formats (when using a custom program written in Origin C). The Wizard can save a filter in select locations and can include script that runs after the import occurs. Once created, the filter can be used to import data and automatically execute script. This same functionality applies when you drag a file from Explorer and drop onto Origin if **Filter Manager** has support for the file type.

For example,

- Start the Import Wizard
- Browse to the Origin Samples\Spectroscopy folder and choose **Peaks with Base.DAT**
- Click Add, then click OK
- Click Next six times to get to the *Save Filters* page
- Check **Save Filter** checkbox
- Enter an appropriate Filter file name, such as **Subtract Base and Find Peaks**

- Check **Specify advanced filter options** checkbox
- Click Next
- Paste the following into the text box:

```
range raTime = 1;              // Get the Time column as a range
range raAmp = 2;               // Get the Amp column as a range
range raBase = 3;              // Get the Base column as a range
wks.addcol(Subtracted);        // Create a column called Subtracted
range raSubtracted = 4;        // Get the Subtracted column as a range
raSubtracted = raAmp - raBase; // Subtract Base from Amp
pkFind iy:=(1,4);              // Find peaks in the Subtracted data
range raPeaks = 5;             // Get the peak index column as a range
for( idx = 1; idx <= raPeaks.GetSize() ; idx++ )
{
    pkidx = raPeaks[idx];
    ty Peak found at $(raTime[pkidx]) with height of $(raSubtracted[pkidx]);
}
```

- Click Finish

This is what happens:

- The filter is saved
- The import runs using this filter
- After the import, the script runs which creates the subtracted data and the **pkFind** function locates peak indices. Results are typed to the Script Window.

6.1.9 From Nonlinear Fitter

The Nonlinear Fitter has a **Script After Fitting** section on the **Code** page of the NLFit dialog. This can be useful if you want to always do something immediately after a fit. As an example, you could access the fit parameter values to do further calculations or accumulate results for further analysis.

In this example, the **Script After Fitting** section adds the name of the fit dataset and the calculated peak center to a Workbook named **GaussResults**:

```
// This creates a new book only the first time
if(exist(GaussResults)!=2)
{
    newbook name:=GaussResults sheet:=1 option:=1 chkname:=1;
    GaussResults!wks.col1.name$= Dataset;
    GaussResults!wks.col2.name$= PeakCenter;
}

// Get the tree from the last Report Sheet (this fit)
getresults iw:=__REPORT$;
```

```
// Assign ranges to the two columns in 'GaussResults'
range ra1 = [GaussResults]1!1;
range ra2 = [GaussResults]1!2;

// Get the current row size and increment by 1
size = ra1.GetSize();
size++;

// Write the Input data range in first column
ra1[size]$ = ResultsTree.Input.R2.C2$;
// and the Peak Center value in the second
ra2[size] = ResultsTree.Parameters.xc.Value;
```

6.1.10 From an External Application

External applications can communicate with Origin as a COM Server. Origin's COM Object exposes various classes with properties and methods to other applications. For complete control, Origin has the **Execute** method which allows any LabTalk - including LabTalk callable X-Functions and OriginC function - to be executed. In this example (using Visual Basic Syntax), we start Origin, import some data, do a Gauss fit and report the peak center :

```
' Start Origin
Dim oa
Set oa = GetObject("", "Origin.Application")
'oa.Execute ("doc -m 1") ' Uncomment if you want to see Origin
Dim strCmd, strVar As String
Dim dVar As Double
' Wait for Origin to finish startup compile
' (30 seconds is specified here,
'  but function may return in less than 1 second)
oa.Execute ("sec -poc 30")

'Project is empty so create a workbook and import some data
oa.Execute ("newbook")
strVar = oa.LTStr("SYSTEM.PATH.PROGRAM$") + _
         "Samples\Curve Fitting\Gaussian.DAT"
oa.Execute ("string fname") ' Declare string in Origin
oa.LTStr("fname$") = strVar ' Set its value
oa.Execute ("impasc")       ' Import

' Do a nonlinear fit (Gauss)
strCmd = "nlbegin 2 Gauss;nlfit;nlend;"
oa.Execute (strCmd)
' Get peak center
dVar = oa.LTVar("nlt.xc")
```

```
    strVar = "Peak Center at " + CStr(dVar)
    bRet = MsgBox(strVar, vbOKOnly, "Gauss Fit")

    oa.Exit
    Set oa = Nothing
    End
```

There are more detailed examples of COM Client Applications in the **Samples\Automation Server** folder.

6.1.11 From Console

When Origin is started from the command-line of an external console (such as Windows **cmd** window), it reads any command *beyond* the **Origin.exe** call to check if any optional arguments have been specified.

Syntax of Command Line Arguments

All command line arguments are optional. The syntax for passing arguments to Origin is:
Origin.exe [-switch *arg*] [*origin_file_name*] [*labtalk_scripts*]

- *-switch arg*
  Multiple switches can be passed. Most switches follow the above notation except -r, -r0 and -rs, which use LabTalk scripts as arguments and execute the scripts after Origin C startup compile. See the Switches table and examples below for available switches and their function.
- *origin_file_name*
  This file name must refer to an Origin project file or an Origin window file. A path may be included and the file extension must be specified.
- *labtalk_scripts*
  Optional script to run after the OPJ is open. This is useful when the script is very long.

Switches

Switch	Argument	Function
-A	*cnf file*	Specifies a configuration file to add to the list specified in the INI file. Configuration files can include any LabTalk command, but typically contain menu commands and macro definitions. You can not specify a path nor should you include a file extension. The file must be in the Origin Folder and must have a CNF

		extension. For example: *C:\Program Files\OriginLab\Origin8\Origin8.exe -a myconfig* **Note:** When passing the .cnf file on the command line using -a switch, Origin C may not finish startup compiling, and the licensing has probably not been processed by the time the .cnf file is processed. So, when you want to include X-Functions in your .cnf file, it's better to use -r or -rs switch instead of -a.
-B	<none>	Run script following OPJ path-name after the OPJ is open similar to -R but before the OPJ's attached ProjectEvents.ogs, such that you can use this option to pass in variables to ProjectEvents.ogs. This option also have the advantage of using all the command line string at the end so it does not need to be put into parenthesis as is needed by -R. (8.1)
-C	*cnf file*	Specifies a new configuration file to override the specification in the INI file. Configuration files can include any LabTalk command, but typically contain menu commands and macro definitions.
-H	<none>	Hide the Origin application. Script Window will still show if it is open by internal control.
-HS	<none>	Same as -h, but in addition to also prevent **Script Window** to open. This is important for scheduled tasks to run reliably.(9.0 SR1)
-I	*ini file*	Specifies an initialization file to use in place of ORIGIN.INI. In general this switch should precede other switches if more than one switch is used.
-L	*level*	Specifies at which menu level to start Origin at.
-M	<none>	Run the Origin application as minimized. (8.1)
-OCW	*ocw file*	Load the Origin C workspace file.
-P	*full path*	Directs the network version of Origin to look for client-specific files in the specified path.

-R	(*script*)	Run the LabTalk *script* after any specified OPJ has been loaded. **Note:** This script will execute after Origin C startup compile.
-R0	(*script*)	Run the LabTalk *script* before any specified OPJ has been loaded. **Note:** This script will execute after Origin C startup compile.
-RS	*scripts*	Similar to -R but without having OPJ specified. All the remaining string from the command line will be used as LabTalk script and run after Origin C startup compile has finished. (8.1)
-SLOG	*file name*	Change script window output to a file. If no path is provided, then the file will be in the user file folder. If no *file name* is specified and another switch follows, like **-slog -hs**, then **Script_Log.txt** will be created in the user file folder.(9.0 SR1)
-TL	*file name*	Specifies the default page template.
-TM	*otm file*	Specifies the default matrix template.
-TG	*otp file*	Specifies the default graph template. Same as -TP.
-TP	*otp file*	Specifies the default graph template. Same as -TG.
-TW	*otw file*	Specifies the default worksheet template.
-W	<none>	Directs the network version of Origin to look for client-specific files in the Start In folder or Working directory.

Examples

Loading an Origin Project File

The following is an example of a DOS **.bat* file. First it changes the current directory to the Origin exe directory. It then calls Origin and passes the following command line arguments:

- *-r0 (type -b "opj will open next")*
  Uses -r0 to run script before the Origin project is loaded.
- *-r (type -b "OPJ is now loaded")*
  Uses -r to run script after the Origin project is loaded.

- *c:\mypath\test.opj*
  Gives the name of the Origin project to open.

Please note that these -r, -r0, -rs, -b switches will wait for Origin C startup compiling to finish and thus you can use X-Functions in such script, as in:

cd "C:\Program Files\OriginLab\Origin8" origin8.exe -r0 (type -b "opj will open next") -r (type -b "OPJ is now loaded") c:\mypath\test.opj

For more complicated scripts, you will be better off putting them into an OGS file, and then running that OGS file from the command line.

The following example will run the script code in the main section of the **startup.ogs** file located in the User Files Folder. When the file name given to the **run.section** method does not contain a path, LabTalk assumes the file is in the User Files Folder. The following command line argument illustrates use of the **run.section** object method:

C:\Program Files\OriginLab\Origin8\Origin8.exe -rs run.section(startup.ogs, main)

A simple **startup.ogs** file to demonstrate the above example can be:

```
[main]
    type -b "hello, from startup.ogs";
```

Run OPJ-Based Custom Program with Command-Line Control

The **ProjectEvents.ogs** script attached to an OPJ file can be used to create an OPJ-centered task-processing tool. In the following example, an OPJ can be used to run a program either by opening the OPJ directly, or by calling it from a command line console external to Origin. In addition, we can set a project variable in the command line to indicate whether the OPJ was opened by a user from the Origin GUI or as part of a command-line argument.

We will create an OPJ with the following ProjectEvents.ogs code:

```
[AfterOpenDoc]
Function doTask()
{
   type -a "Doing some task...";
   // code to do things
   type "Done!";
}
//%2 = 2 for command line, but also for dble-click OPJ
// so we better control it exactly with this variable
CheckVar FromCmdLine 0;
if(FromCmdLine)
{
   type -b "Coming from command line";
   doTask();
   sec -p 2;//wait a little before closing
   exit;
}
else
{
```

```
    type -N "Do you want to do the task now?";
    doTask();
}
```

To run this OPJ (call it *test*) from a command line, use the -B switch to ensure the **FromCmdLine** variable is defined before **[AfterOpenDoc]** is executed:

<exepath>Origin81.exe -b <opjpath>test.opj FromCmdLine=1

Batch Processing with Summary Report in Origin

The following example demonstrates starting Origin from a command line shell (i.e., Windows **cmd**) by entering a long script string containing the **-rs** switch.

The script performs several actions:

1. Sets up a string variable (*fname$*) with multiple file names,
2. Calls an X-Function (batchprocess) to perform batch processing using an existing analysis template,
3. Calls an X-Function (*expasc*) to Export the result to a CSV file (*c:\test\my batch\output.csv*),
4. Suppresses a prompt to save the Origin Project (OPJ) file (*doc -s*), and
5. Exits the Origin application.

To begin, issue this command at an external, system-level command prompt (such as Windows **cmd**), replacing the Origin installation path given with the one on your computer or network:

C:\Program Files\OriginLab\OriginPro81\Origin81.exe -m -rs template$="C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\autofit.ogw"; fname$="C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\step01.dat%(CRLF)C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\step02.dat%(CRLF)C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\step03.dat%(CRLF)C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\step04.dat%(CRLF)C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\step05.dat%(CRLF)C:\Program Files\OriginLab\OriginPro81\Samples\Curve Fitting\step06.dat"; batchprocess batch:=template name:=template$ fill:="Data" append:="Summary" ow:=[Summary Book]"Summary Sheet"!; expasc iw:=[Summary Book]"Summary Sheet"! type:=csv path:="c:\test\my batch output.csv"; doc -s; exit;

Batch Processing with Summary Report in External Excel File

This example demonstrates using an external Excel file to generate a Summary Report using Batch Processing.

In one single continuous command line, the following is performed:

1. Origin is launched and an existing Origin Project file (OPJ) is loaded which contains
    - an Origin workbook to be used as Analysis Template, and
    - an externally linked Excel file to be used as the report book.

2. All files matching a particular wild card specification (in this case, file names beginning with *T* having the **.csv* extension) are found.
3. The batchProcess X-Function is called to perform batch processing of the files.
4. The Excel window will contain the summary report at the end of the batch processing operation. This window, linked to the external Excel file, is saved and the Origin project is closed without saving.
5. You can directly open the Excel file from the **\Samples\Batch Processing** subfolder to view the results.

To begin, issue this command at an external, system-level command prompt (such as Windows **cmd**), replacing the Origin installation path given with the one on your computer or network:

C:\Program Files\OriginLab\OriginPro81\Origin81.exe -rs string path$=system.path.program$+"Samples\Batch Processing\";string opj$=path$+"Batch Processing with Summary Report in External Excel File.opj";doc -o %(opj$);findfiles ext:="T.csv";win -a Book1;batchProcess batch:=0 fill:="Raw Data" append:="My Results" ow:="[Book2]Sheet1!" number:=7 label:=1;win -o Book2 {save -i}; doc -s; exit;*

Additional information on batch processing from script (using both Loops and X-Functions) is available in a separate Batch Processing chapter.

6.1.12 On A Timer

The **Timer** (Command) executes the **TimerProc** macro, and the combination can be used to run a script every **n** seconds.

The following example shows a timer procedure that runs every 2 seconds to check if a data file on disk has been modified, and it is then re-imported if new.

In order to run this scipt example, perform the following steps first:

1. Create a simple two-column ascii file **c:\temp\mydata.dat** or any other desired name and location
2. Start a new project and import the file into a new book with default ascii settings. The book short name changes to **mydata**
3. Create a line+symbol plot of the data, and set the graph x and y axis rescale property to auto so that graph updates when new data is added
4. Keep the graph as the active window
5. Save the script below to the [AfterOpenDoc] section of the ProjectEvents.OGS file attached to the project.
6. Add the following command to the [BeforeCloseDoc] section of ProjectEvents.OGS:

```
timer -k;
```

7. Save the Origin Project, close, and then re-open the project. Now any time the project is opened, the timer will start, and when the project is closed the timer will stop executing.
8. Go to the data file on disk and edit and add a few more data points
9. The timer procedure will trigger a re-import and the graph will update with the additional new data

```
// Set up the TimerProc macro
def TimerProc {
   // Check if file exists, and quit if it does not
   string str$="c:\temp\mydata.dat";
   if(0 == exist(str$) ) return;

   // Get date/time of file on disk
   double dtDisk = exist(str$,5);

   // Run script on data book
   // Assuming here that book short name is mydata
   win -o mydata {
        // Get date/time of last import
        double dtLast = page.info.system.import.filedate;

        // If file on disk is newer, then re-import the file
        if( dtDisk > dtLast ) reimport;
   }
}

// Set TimerProc to be executed every 2 seconds
timer 2;
```

The **Samples\LabTalk Script Examples** subfolder has a sample Origin Project named **Reimport File Using Timer.OPJ** which has script similar to above set up. You can open this OPJ to view the script and try this feature.

6.1.13 On Starting Origin

When the Origin application is launched, there are multiple events that are triggered. Your LabTalk script can be set to execute with each event using the **OEvents.OGS** file.

For example, after all Origin C functions have been compiled on startup, you may want your custom script to execute. The following example demonstrates adding user-defined LabTalk functions on starting Origin. These functions will then be available in every Origin session.

1. Create a new .OGS file, say *MyLTFuncs.OGS*, in your Origin User File Folder, with the following script:

```
[DefFuncs]
@global = 1;
function int myswap(ref double a, ref double b)
{
   double temp = a;
   a = b;
   b = temp;
   return 0;
}
```

2. Copy the *OEvents.OGS* file from the Origin EXE folder to your User Files Folder. Alternatively, when opening the file from the EXE folder just make sure to then save it to your User Files Folder.

   Note: Please copy and then edit all system .OGS files, .CNF files, etc. in your User File Folder.

3. This OEvents.OGS file includes several sections that indicate when to run the script, such as, *[AfterCompileSystem]*, *[BeforeOpenDoc]*, and *[OnExitOrigin]*.
4. In the section named *[AfterCompileSystem]*, add the following script:

```
// Run my LabTalk function definition script file
run.section(MyLTFuncs, DefFuncs);
```

5. To run sections in OEvents.OGS, you also need to edit the *Origin.ini* file in your User File Folder. Close Origin if running, and then edit Origin.ini and uncomment (remove ;) in the line under "OEvents" section, so that it is as below:

```
Ogs1 = OEvents
; Ogs2 = OEvents
; Origin can trigger multiple system events
; Uncomment this line and implement event handlers in OEvents.ogs
```

   Note: More than one event handler file may exist in *Origin.ini* and the name is not restricted to OEvents.OGS.

6. Start a new Origin session, and run the following testing script to check that your user-defined function is now working:

```
double a = 1.1;
double b = 2.2;
```

```
ty "At the beginning, a = $(a), and b = $(b)";
myswap(a, b);
ty "After swap, a = $(a), and b = $(b)";
```

Note1: If you need to call Origin C functions from your custom script associated with events, you need to ensure that the Origin C file is compiled and the functions are available for script access. See Loading and Compiling Origin C Function for details.

Note2: Since the events are indirectly determined by the ORIGIN.INI file you can create custom environments by creating multiple INI files. You can launch Origin using a custom INI file by specifying on the command line as in the CMD console or in a Shortcut. See Script From Console

6.1.14 From a Custom Menu Item

LabTalk script can be assigned to custom menu items. The **Custom Menu Organizer** dialog accessible from the **Tools** main menu in Origin provides easy access to add and edit main menu items. The **Add Custom Menu** tab of this dialog can be used to add a new main menu entry and then populate it with sub menu items including pop-up menus and separators. Once a menu item has been added, LabTalk script can be assigned for that item. The menu items can be made available for all window types or a specific window type.

The custom menu configuration can then be saved and multiple configuration files can be created and then loaded separately using the **Format: Menu** main menu. For further information please view the help file page for the Custom Menu Organizer dialog.

6.1.15 From a Toolbar Button

LabTalk script files can also be run from buttons on the Origin toolbar. In Getting Started with LabTalk chapter, we have introduced how to run Custom Routine from a toolbar button, here we will introduce more details. Three files enable this to happen:

1. A bitmap file that defines the appearance of the button. Use one of the set of buttons provided in Origin or create your own.
2. A LabTalk script file that will be executed when the user clicks the button.
3. An INI file that stores information about the button or button group. Origin creates the INI file for you, when you follow the procedure below.

We will assume for now that you have a bitmap image file (BMP) that will define the button itself (if you are interested in creating one, example steps are given below).

First, use **CodeBuilder** (select  on the Origin Standard Toolbar to open) or other text editor, to develop your LabTalk script (OGS) file. Save the file with the OGS extension. You may

divide a single script file into several sections, and associate each section with a different toolbar button.

Putting a Button on an Origin Toolbar

To put the button on an Origin toolbar, use this procedure:

1. In Origin, select **View:Toolbars** to open the Customize Toolbar dialog.
2. Make the **Button Groups** Tab active.
3. Click the **New** button in the **Button Group** to open the Create Button Group dialog.
4. Enter a new Group Name.
5. Enter the Number of Buttons for this new Group.
6. Click the Browse button to locate your bitmap file. This file should be in your User directory.
7. Click **OK**.
8. The **Save As** dialog will open. Enter the same name as that of your bitmap file. Click **OK** to save the INI file. You will now see that your group has been added to the Groups list and your button(s) is now visible.

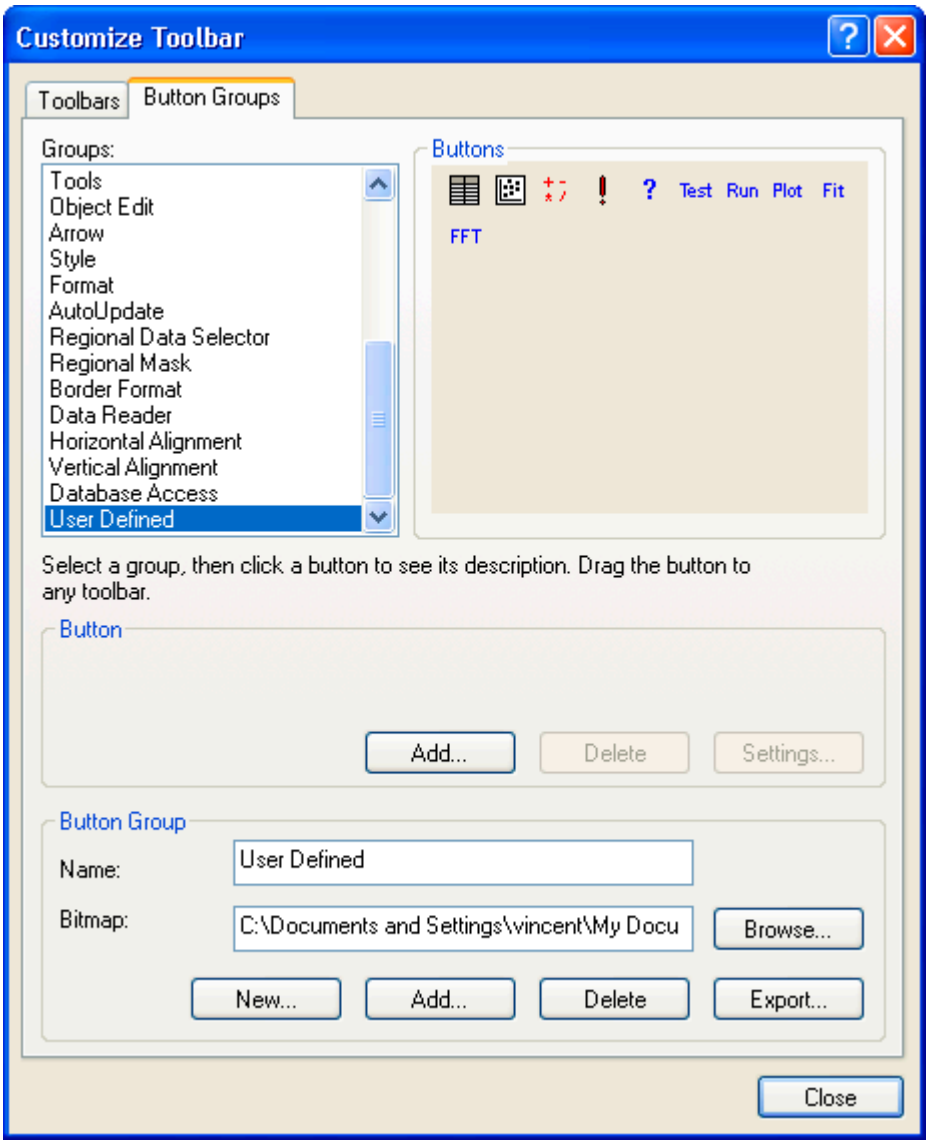
When creating a custom button group for export to an OPX file, consider saving your button group's initialization file, bitmap file(s), script file(s), and any other support files to a user-created subfolder in your **User Files** folder. When another Origin user installs your OPX file, your custom subfolder will automatically be created in the user's **User Files** folder, and this subfolder will contain the files for the custom button group. This allows you to keep your custom files separate from other Origin files.

Match the Button with a LabTalk Script (OGS) File

1. Click on the button to select it.
2. Click the Settings button, to open the Button Settings dialog.
3. Click the Browse button to locate your OGS file.
4. Enter the Section Name of the OGS file and any arguments in the Argument List.
5. Enter something descriptive in the Tool Tip Text text box.
6. Enter a status bar message in the Status Bar text box.
7. Click **OK**.
8. Repeat these steps for each of the buttons in your Button Group.
9. Drag the first button out onto your Origin workspace. A toolbar is created. You can now drag all other buttons onto this toolbar.

Custom Buttons Available in Origin

The following dialog can be accessed from the **View: Toolbars** menu option in Origin. On the **Button Groups** tab, scroll down to select the **User Defined** group:



Drag any of these buttons onto the Origin toolbar to begin using them. Use the procedure outlined above to associate a script with a given button.

Creating a Bitmap File for a New Button

To create a bitmap file, using any program that allows you to edit and save a bitmap image (BMP) such as Window's Paint. The following steps will help you get started:

1. Using the bitmap for the built-in user defined toolbar is a good place to begin. In Windows Paint, select **File:Open**, browse to your User Files folder and select **Userdef.bmp**.
2. Set the image size. Select **Image:Attributes**. The height needs to remain at 16 and should not be changed. Each button is 16 pixels high by 16 pixels wide. If your toolbar will be only 2 buttons then change the width to 32. The width is always 16 times the number of buttons, with a maximum of 10 buttons or a width of 160.
3. Select **View:Zoom:Custom:800%**. The image is now large enough to work with.
4. Select **View:Zoom:Show Grid**. You can now color each pixel. The fun begins - create a look for each button.
5. Select **File:Save As**, enter a new File name but leave the **Save as type** to **16 Color Bitmap**.

6.2 Debugging Scripts

This section covers means of debugging your LabTalk scripts. The first part introduces interactive execution of script. The second presents several debugging tools, including Origin's native script editor, Code Builder. And the third covers the error handling.

6.2.1 Interactive Execution

You can execute LabTalk commands or X-functions line-by-line (or a selection of multiple lines) to execute step-by-step. The advantage of this procedure is that you can verify the result of the issued command, and according to the result or error, you can act appropriately.

To execute LabTalk commands interactively, you can enter them in the following places:

- Classic Script Window
- Command Window in Origin's main window
- Command & Results Windows in Code Builder

The characteristics and the advantages of each window are as follows:

Classic Script Window

This window can be open from the **Window** main menu. This is the most flexible place for advanced users to execute LabTalk scripts. Enter key will execute

1. the current line if cursor has no selection
2. the selected block if there is a selection

You can use Ctrl+Enter to add a line without executing. There is also a **Script Execution** option on the Edit menu to toggle between editing and interactive execution.

Command Window in Origin's Main Window

You can enter a LabTalk command at the command prompt in the Command Window. The result would be printed immediately after the entered command line. Command Window has various convenient features such as command history panel, auto-completion, roll back support to utilize previously executed commands, to execute a block of previously executed commands, to save previously executed commands in an OGS file, etc. You cannot edit multiline scripts within the Command Window.

To learn how to use the Command window, see **The Origin Command Window** chapter in the Origin help file.

Command & Results Windows in Code Builder

Code Builder is Origin's integrated development environment useful in debugging LabTalk scripts as well as Origin C code, X-Function code, etc. In Code Builder, use various convenient debugging tools like setting up break points, step-by-step execution, inspection of the values of variables, etc.

To learn how to use the Code Builder, see the Code Builder User's Guide in the Programming help file.

6.2.2 Debugging Tools

Origin provides various tools to help you to develop and debug your LabTalk scripts.

Code Builder (Origin feature)

Code Builder is Origin's integrated development environment to debug LabTalk scripts, Origin C code, X-Function code and fitting functions coded in Origin C. In Code Builder, use various convenient debugging tools like setting up break points, step-by-step execution and inspection of variable values. Code Builder can be opened by the ed.open() method.

To learn how to use the Code Builder, see the **Code Builder User's Guide** in the Programming help file.

Here is an example showing how to debug LabTalk script in Code Builder.

1. Open an OGS file by running the following script.

```
// Open an ogs file in Code Builder
file$ = system.path.program$ + "Samples\LabTalk Script
Examples\ave_traces.ogs";
ed.open(%(file$));
```

2. Set a break point on line 22 in the open file, by clicking on the margin to the left of this line:

```
fname$ = system.path.program$ + "Samples\Data
Manipulation\not_monotonic_multicurve.dat";
```

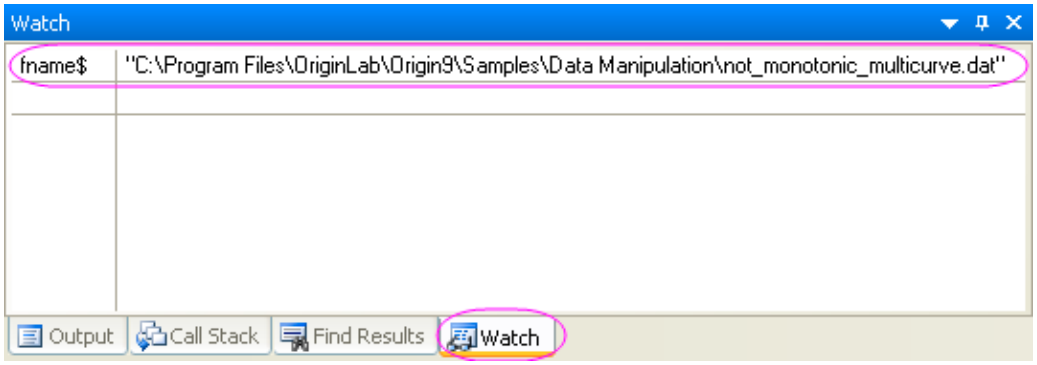
   The break point will look like this:

```
//test to make sure OriginPro is installed
if (system.product&1 != 1)
{
    type "This feature is only available in OriginPro 8.";
    break;
}

// Put the path of sample data into fname string variable which is the default used by impASC
fname$ = system.path.program$ + "Samples\Data Manipulation\not_monotonic_multicurve.dat";

newbook;// Create a new book
impASC;// import the file using all defaults
string bkn$ = %H; // save the book name as plotting will create new window to change %H
plotxy [bkn$]!((1,2), (3,4), (5,6), (7,8)) plot:=200;
```

3. Place the cursor on line 12 - the **[Main]** section - then select menu **Debug: Execute Current Section**. The **[Main]** section code will run and stop at the line with the break point.

```
//test to make sure OriginPro is installed
if (system.product&1 != 1)
{
    type "This feature is only available in OriginPro 8.";
    break;
}

// Put the path of sample data into fname string variable which is the default used by impASC
fname$ = system.path.program$ + "Samples\Data Manipulation\not_monotonic_multicurve.dat";

newbook;// Create a new book
impASC;// import the file using all defaults
string bkn$ = %H; // save the book name as plotting will create new window to change %H
plotxy [bkn$]!((1,2), (3,4), (5,6), (7,8)) plot:=200;
```

4. Now press F10 to execute the remaining script line by line. Code Builder provides the **Watch** window to view the value of a variable during debugging. For example, after pressing F10 once, open the **Watch** window by menu item **View: Watch** if it is not opened yet. Then type *fname$* in the left cell of the table in this window, the value will show in the right cell of the same row.



5. To execute the remaining script, press F5. It will complete unless encountering another break point.

Ed (object)

The **Ed** (object) provides script access to Code Builder, a dedicated editor for LabTalk script and Origin C code.

The **ed** object methods are:

Method	Brief Description
ed.open()	Open the Code Builder window.
ed.open(fileName)	Open the specified file in the Code Builder window.
ed.open(fileName, sectionName)	Open the specified OGS file at the specified section in the Code Builder window. (Defaults to file beginning if section not found.)

Open the Code Builder

```
ed.open()
```

Open a Specific File in Code Builder

The following command opens the file **myscript.ogs**

```
ed.open(E:\myfolder\myscript.ogs)
```

Open a File on a Pre-Saved Path

Use the **cd** X-Function to first switch to the particular folder:

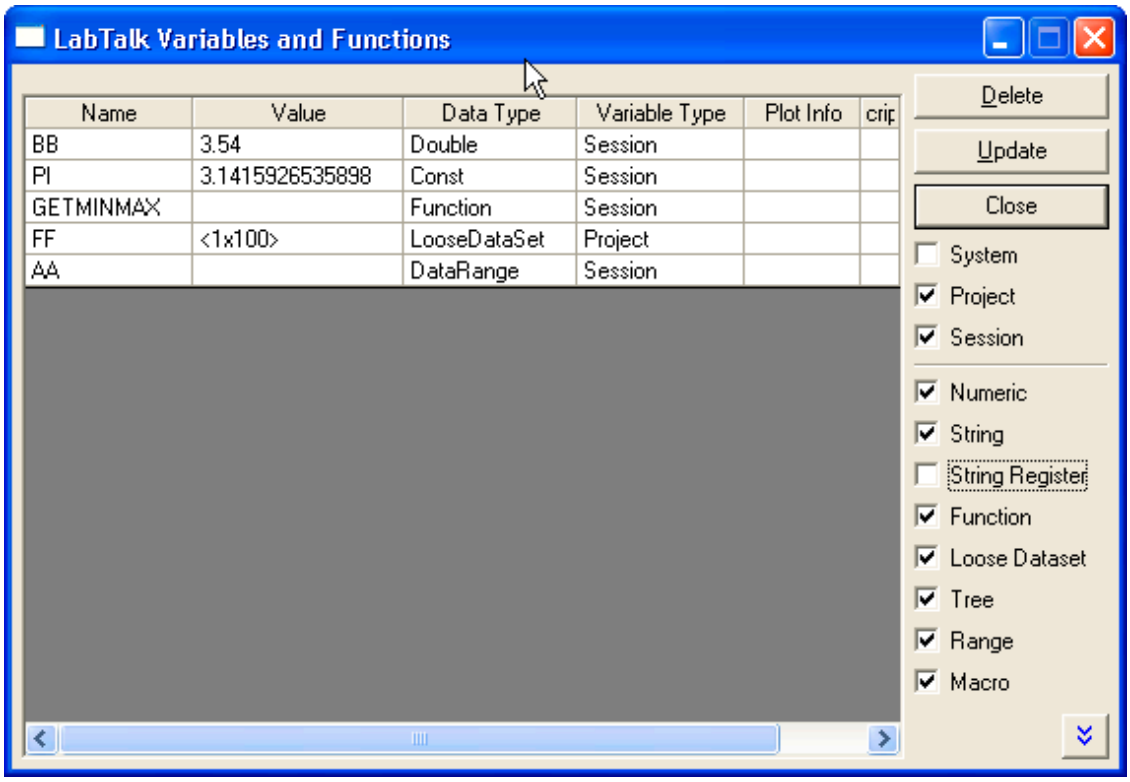
```
cd 2;
ed.open(autofit.ogs);
```

LabTalk Variables and Functions Dialog

The **list** command with no options as well as the **ed** command (different than the **ed** object) opens the LabTalk Variables dialog, which is a table of attributes for all variables in the current project. The attributes are variable name, value, type, subtype, property, plot information, and description.

This is a useful tool for script programmers as the current values and properties of variables can be viewed in real time. Additionally, variables can be sorted by any of their attributes, alphabetically in the case of text descriptors, numerically in the case of numeric values.

Check boxes exist on the right-hand side of the dialog that allow you to see any subset of the entire variable list.



Echo (system variable)

To debug and trace, this system variable, **Echo** prints scripts or error messages to the Command window (or Script window when entered there). To enable echo, type:

echo = *Number*

in the Script window (where *Number* is one of the following):

Number	Description
1	Display commands that generate an error;
2	Display scripts that have been sent to the queue for delayed execution;
4	Display scripts that involve commands;
8	Display scripts that involve assignments;
16	Display macros.

These values are bits that can be combined to produce cumulative effects. For example, echo = 12 displays both command and assignment scripts. Echo = 7 (includes echo = 1, echo = 2, and echo = 4) is useful for following script execution during menu command selection. To disable echo, type echo = 0 in the Script window

#!*script* (special syntax)

Embed debugging statements in your script using this notation. The # character tells the LabTalk interpreter to ignore the text until the end of the line. However, when followed by the ! character, the script is executed if the @B system variable (or System.Debug object property) is set to 1. The following example illustrates this option:

```
@B = 1;
range rr = [Book1]Sheet1!col(A);  // Range to column A
for (ii=1; ii<=10; ii+=1) {
  #!ii=; rr[ii]=;  // Embedded debugging script
  rr[ii]+=ii*10;
}
@B = 0;
#!type -a This line will not execute
```

The script sets @B equal to 1 to allow #! lines to execute. By setting @B to 0 , the last line will not execute.

{*script*} (special syntax)

An error in your LabTalk code will cause the code to stop at the point of the error and not execute any statements after the error. In cases where you would like the script to continue executing in such cases, you can use curly braces to define where error handling should begin and resume. For instance, in the following script,

```
type Start;
impasc fname:=BadFileName;
type End;
```

the word **Start** will print to the Script Window, but if **BadFileName** cannot be found, the script will stop executing at that point, and the word **End** will not print.

If, however, you surround the line in question with curly braces (i.e., {}), as in,

```
type Start;
{
   impasc fname:=BadFileName;
}
type End;
```

then **End** will print whether or not **BadFileName** is properly imported.

You can catch this condition with a variable:

```
flag = 1;
{
   impasc fname:=MyFile;
```

```
   flag = 0;
}
if(flag)
   type Error ocurred;
else
   type OK;
```

A similar situation occurs when a section in an OGS file fails. Code will silently return to the calling context. Use the above variable method to identify that code failed. In this case, the brackets are not needed:

```
[Called Section]
flag = 1;
BadCommand; // This line errors and silently returns
flag = 0;  // flag (which must be global variable) is 1, then above code failed.
```

@B(system variable), System.Debug (object property)

@B system variable controls the Debug mode to execute the LabTalk statements that begin with **#!** :

```
1 = enable
0 = disable
```

It is equivalent to **System.Debug** object property. which

@OC (system variable)

@OC system variable controls whether or not you can call Origin C functions from LabTalk.

Value	Description
@OC = 1 (default)	Origin C functions **CAN** be called
@OC = 0	Origin C functions **CANNOT** be called

@V(system variable), System.Version(object property)

@V indicates the Origin version number. @V and System.Version object property are equivalent.

@VDF (system variable)

If you set @VDF = 1, when you open a project file (.OPJ), Origin will report the Origin version in which the file was saved.

VarName= (command)

This command examines the value of any variable. Embed this in your script to display intermediate variable values in the Script window during script execution.

Example 1

The following command prints out the value of myHight variable:

```
myHight=
```

LabTalk:List (command)

The list command is used to examine your system environment. For example, the list s command displays all datasets (including temporary datasets) in the project.

ErrorProc (macro)

Macro type: Special event The ErrorProc macro is triggered ▪when the LabTalk interpreter detects a #Command Error. ▪when you click the Cancel button in any dialog box. ▪when you click the No button in dialog boxes that do not have a Cancel button. The ErrorProc macro is deleted immediately after it is triggered and executed. The ErrorProc macro is useful for error trapping.

NotReady (macro)

This macro displays the message "This operation is still under development..." in a dialog box with an OK button.

Type <ogsFileName> (command)

This variant of the *type* command prints out the contents of a specified script file (.OGS) in the current directory to the Command (or Script) window. Note that the file extension .OGS in ogsFileName may be omitted. The file name cannot include a path and must be in the working directory.

Examples:

The following script prints the contents of D: \temp\mytemp1.ogs and C:\myogs\hello.ogs.

```
cd D:\Temp;
type mytemp1.ogs; // Extension included
cd C:\temp;
type hello; // Extension omitted
```

Log to a File

To output the log information to a file, the type command is available. type -gb will specify the log file to output to, and begin the output routine. Then type -ge will end the routine and stop logging to the file. For example:

```
type -gb %Ylog.txt;  // Start typing text to a file, log.txt, if not exist, create
it
type aa;  // Write aa
type bb;  // Write bb
type cc;  // Write cc
type -ge;  // End writing
```

This can be particularly useful when your script is creating a large volume of output to the Script Window since it has only a 30000 byte buffer.

6.2.3 Error Handling

LabTalk scripts may be interrupted if an error has been thrown. But there are times when you want to continue the execution of the script even if an error is encountered. In this situation, Origin allows you to use a pair of curly braces ("{" and “}”) to enclose a part of the script that might generate an error. When Origin encounters an error within the section the remaining script up to the "}" is skipped and execution resumes outside the curly braces. In this sense, braces and run.section() commands have the same behavior.

The following is a simple example to show how to handle possible errors. Please note that before executing the scripts in the Script Window, you should create a new worksheet and make sure that column C does not exist.

```
// Script without error handling
type "Start the section";
stats col(c);
stats.max=;
type "Finished the section";
```

The line of code, `stats col(c);`, will throw an error, because Column C does not exist. Then, the script will terminate and only output:

```
Start the section
Failed to resolve range string, VarName = ix, VarValue = col(c)
```

Now we will introduce braces to use error handling. We can add a variable to indicate if an error occurred and make use of a System Variable to temporarily shut off Origin error messages:

```
// Script with error handling
type "Start the section";
int iNOE = @NOE;  // Save current Origin error message output flag
// The section that will generate an error
{
   @NOE = 0; // Shut off Origin error messages
   vErr = 1; // Set our error variable to true (1)
   stats col(c); // This is the code which could produce an error
   stats.max=; // Execution will continue only if no error occurs
   vErr = 0; // If NO error then our variable gets set to false (0)
}
@NOE = iNOE; // Restore Origin error messages
if(vErr) ty An error occurred. Continuing ...;
type "Finished the section";
```

The output will become

```
Start the section
An error occurred. Continuing ...
Finished the section
```

After the error on the `stats col(c)` line, code execution continues outside the closing brace (**}**) and we can trap our error and process as needed. You can comment out the lines related to @NOE if you want the Message Log to retain a record of all errors that occurred.

7 String Processing

This chapter introduces you to working with strings, including string variables, registers and arrays, converting numbers to strings, strings to numbers, and various methods available for string processing.

7.1 String Variables and String Registers

In Origin, string processing is supported in two ways: with string variables, and with string registers. In general, we encourage the use of string variables as they are more intuitive (i.e., more like strings in other programming languages) and are supported by many pre-defined string methods; both of which are advantages over string registers.

7.1.1 String Variables

A string variable is created by declaration and/or assignment, and its name is always followed by a $-sign. For example:

```
// Creates by declaration a variable named 'aa' of the string type;
//'aa' is empty (i.e., "")
string aa$;

// Assigns to 'aa' a character sequence
aa$ = "Happy";

// Creates and assigns a value to string variable 'bb',
//all on the same line
string bb$ = "Yes";

// Creates and assigns a value to string variable 'cc' with no declaration
//(see note below)
cc$ = "Global";
```

Note: Because string variable **cc** was not declared, it is given Global (or Project) scope, which means all routines, functions, or otherwise can see it. Declared variables **aa** and **bb** are given Local (or Session) scope. For more on scope, see Variables and Scope.

7.1.2 String Registers

Prior to Version 8.0, Origin supported string processing with string registers. As such, they continue to be supported by more recent versions, and you may see them used in script programming examples. There are 26 string registers, corresponding to the 26 letters of the English alphabet, each preceeded by a %-sign, i.e., %A--%Z. They can be assigned a character sequence just like string variables, the differences are in the way they are handled and interpreted, as the examples below illustrate. As a warning, several of the 26 string registers are reserved for system use, most notably, the ranges %C--%I, and %X--%Z. For complete documentation on their use, see String Registers.

7.2 String Processing

Using String Methods

These examples show multiple ways to get a substring (in this case a file name) from a longer string (a full file path). In the last of these, we demonstrate how to concatenate two strings.

Find substring, using getFileName()

In this example, a string method designed for a very specific but commonly needed task is invoked.

```
// Use the built-in string method, GetFileName():
string fname$="C:\Program Files\Origin 8\Samples\Import\S15-125-03.dat";
string str1$ = fname.GetFileName()$;
str1$=;
```

Find substring, using reverseFind(), mid() methods

This time, a combination of string methods is used:

```
// Use the functions ReverseFind and Mid to extract the file name:
string fname$="C:\Program Files\Origin 8\Samples\Import\S15-125-03.dat";
// Find the position of the last '\' by searching from the right.
int nn=fname.ReverseFind('\');
// Get the substring starting after that position and going to the end.
string str2$=fname.Mid(nn+1)$;
// Type the file name to the Script Window.
str2$=;
```

Find substring, token-based

Here, another variation of generic finding methods is chosen to complete the task.

```
// Use a token-based method to extract the file name:
```

```
string fname$="C:\Program Files\Origin 8\Samples\Import\S15-125-03.dat";
// Get the number of tokens, demarcated by '\' characters.
int nn=fname.GetNumTokens('\');
// Get the last token.
string str3$ = fname.GetToken(nn, '\')$;
// Output the value of that token to the Script Window.
str3$=;
```

String Concatenation

You can concatenate strings by using the '+' operator. As shown below:

```
string aa$="reading";
string bb$="he likes " + aa$ + " books";
type "He said " + bb$;
```

You may also use the **insert** string method to concatenate two strings:

```
string aa$ = "Happy";
string bb$ = " Go Lucky";
// insert the string 'aa' into string 'bb' at position 1
bb.insert(1,aa$);
bb$=;
```

For a complete listing and description of supported string methods, please see String (Object).

Using String Registers

String Registers are simpler to use and quite powerful, but more difficult to read when compared with string variables and their methods. Also, they are global (session scope) and you will have less control on their contents being modified by another program.

```
// Concatenate two strings using string registers

%A="Left";
%B="Handed";
%N="%A %B";
%N=          // "Left Handed"
// Extract the file name substring from the longer file path string:
%N="C:\Program Files\Origin 8\Samples\Import\S15-125-03.dat";
for(done=0;done==0; )
{
    %M=%[%N,>'\'];
    if(%[%M]>0) %N = %M;
    else done = 1;
}
%N=;
```

Extracting Numbers from a String

This example shows multiple ways to extract numbers from a string:

```
// String variables support many methods
string fname$="S15-125-03.dat";

int nn=fname.Find('S');
string str1$ = fname.Mid(nn+1, 2)$;
type "1st number = %(str1$)";

string str2$ = fname.Between("-", "-")$;
type "2nd number = %(str2$)";

int nn = fname.ReverseFind('-');
int oo = fname.ReverseFind('.') ;
string str3$ = fname.Mid(nn + 1, oo - nn - 1)$;
type "3rd number = %(str3$)";

type $(%(str2$) - %(str1$) * %(str3$));

// Using string Registers, we can use substring notation
%M = "S15-125-03.dat";
%N = %[%M,2:3];  // Specify start and end
type "1st number = %N";
%N = %[%M,>'S']; // Find string after 'S'
%N = %[%N,'-'];  // Find remaining before '-'
type "1st number = %N";
%O = %[%M,#2,\x2D];  // Find second token delimited by '-' (hexadecimal 2D)
type "2nd number = %O";
%P = %[%M,'.'];  // trim extension
%P = %[%P,>'-']; // after first '-'
%P = %[%P,>'-']; // after second '-'
type "3rd number = %P";
type $(%O - %N * %P);
```

7.3 Converting Strings to Numbers

The next few examples demonstrate converting a string of numeric characters to an actual number.

7.3.1 Converting String to Numeric

Using Substitution Notation

To convert a variable of type string to a variable of type numeric (double, int, const), consider the following simple example:

```
// myString contains the characters 456 as a string
string myString$ = "456";

// myStringNum now contains the integer value 456
int myStringNum = %(myString$);
```

The syntax **%(string$)** is one of two substitution notations supported in LabTalk. The other, $(*num*), is used to convert in the opposite direction; from numeric to string.

Using String Registers

This example demonstrates how to convert a string held in a string register to a numeric value.

```
// Similar to above, but performed using string registers:
string myString$ = "456";
// Assignment without quotes will evaluate the right-hand-side
%A = myString$;
// %A will be substituted, then right-hand-side evaluated
int aa = %A;
// 'aa' can be operated on by other integers
int bb = aa + 100;
bb=;                 // ANS: 556
```

7.4 Converting Numbers to Strings

The following examples demonstrate conversion of numeric variables to string, including notation to format the number of digits and decimal places.

7.4.1 Converting Numeric to String

Using Substitution Notation

To convert a variable of a numeric type (double, int, or const) to a variable of string type, consider the following simple example:

```
// myNum contains the integer value 456
int myNum = 456;
// myNumString now contains the characters 456 as a string
string myNumString$ = $(myNum);
```

The syntax **$(num)** is one of two substitution notations supported in LabTalk. The other, **%(string$)**, is used to convert in the opposite direction, from string to numeric, substituting a string variable with its content.

Formatting can also be specified during the type conversion:

```
$(number [,format])    // braces indicate that the format is optional
```

Format follows the C-programming format-specifier conventions, which can be found in any C-language reference, for example:

```
string myNumString2$ = $("3.14159",%3d);
myNumString2$=                          // "3"

string myNumString2$ = $("3.14159",%3.2f);
myNumString2$=                          // "3.14"

string myNumString2$ = $("3141.59",%6.4e);
myNumString2$=                          // "3.1416e+003"
```

For further information on this type of formatting, please see $() Substitution.

Using the Format Function

Another way to convert a numeric variable to a string variable uses the **format** function:

```
// call format, specifying 3 significant figures
string yy$=Format(2.01232, "*3")$;
// "2.01"
yy$=;
```

For full documentation of the **format** function see Format (Function)

7.4.2 Significant Digits, Decimal Places, and Numeric Format

LabTalk has native format specifiers that, used as part of LabTalk's Substitution Notation provide a simple means to format a number.

Use the * notation to set significant digits

```
x = 1.23456;
type "x = $(x, *2)";
```

In this example, x is followed by *2, which sets x to display two significant digits. So the output result is:

```
x = 1.2
```

Additionally, putting a * before ")" will cause the zeros just before the power of ten to be truncated. For instance,

```
y = 1.10001;
type "y = $(y, *4*)";
```

In this example, the output result is:

```
y = 1.1
```

The result has only 2 siginificant digits, because y is followed by *4* instead of *4.

Use the . notation to set decimal places

```
x = 1.23456;
type "x = $(x, .2)";
```

In this example, x is followed by .2, which sets x to display two decimal places. So the output result is:

```
x = 1.23
```

Use E notation to change the variable to engineering format

The E notation follows the variable it modifies, like the * notation. For example,

```
x = 1e6;
type "x = $(x, E%4.2f)";
```

where **%** indicates the start of the substitution notation, 4 specifies the total number of digits, .2 specifies 2 decimal places, and f is an indicator for floating notation. So the output is:

```
x = 1.00M
```

Use the $(x, S*n) notation to convert from engineering to scientific notation

In this syntax, *n* specifies the total number of digits.

```
x = 1.23456;
type "x = $(x,S*3)";
```

And Origin returns:

```
x = 1.23E0
```

7.5 String Arrays

This example shows how to create a string array, add elements to it, sort, and list the contents of the array.

```
// Import an existing sample file
newbook;
fpath$ = "Samples\Data Manipulation\US Metropolitan Area Population.dat"
string fname$ = system.path.program$ + fpath$;
impasc;

// Loop over last column and find all states
range rMetro=4;
stringarray saStates;
```

```
for( int ir=1; ir<=rMetro.GetSize(); ir++ )
{
   string strCell$ = rMetro[ir]$;
   string strState$ = strCell.GetToken(2,',')$;
   // Find instances of '-' in name
   int nn = strState.GetNumTokens("-");
   // Add to States string array
   for( int ii=1; ii<=nn; ii++ )
   {
        string str$ = strState.GetToken(ii, '-')$;
        // Add if not already present
        int nFind = saStates.Find(str$);
        if( nFind < 1 )
                 saStates.Add(str$);
   }
}

// Sort States string array and print out
saStates.Sort();
for(int ii=1; ii<=saStates.GetSize(); ii++)
   saStates.GetAt(ii)$=;
```

8 Workbooks Worksheets and Worksheet Columns

In this chapter we cover the Workbook -> Worksheet -> Column hierarchy, and how to access these objects from script. The concept of treating data in a worksheet as a *virtual matrix* is also covered.

8.1 Workbooks

8.1.1 Basic Workbook Operation

You can manipulate workbooks with the Page object and Window command. You can also use Data Manipulation X-Functions. With these tools, you can create new worksbooks, duplicate workbooks, save workbook as template, etc. Some practical examples are provided below.

Create New Workbook

The newbook X-Function can be used to create new workbook. With the arguments of this X-Function, you can specify the newly created workbook with Long Name, number of sheets, template to use, whether hidden, etc.

```
//Create a new workbook with the Long Name "MyResultBook"
newbook MyResultBook;

// Create a new workbook with 3 worksheets
// and use "MyData" as Long Name and short name
newbook name:="MyData" sheet:=3 option:=lsname;

// Create a new hidden workbook
// and the workbook name is stored in myBkName$ variable
newbook hidden:=1 result:=myBkName$;
// Output workbook name
myBkName$ = ;

// By default, the built-in template "Origin" is used to
// create the new workbook, you can also use a specified template
// Create a new workbook with the XYZ template
newbook template:=XYZ;
```

Also, the command win -ti is capable of creating a minimized new workbook from a template file.

```
// Create a new wookbook from the FFT template
// and Long Name and short name to be MyFFT, then minimize it
win -ti wks FFT MyFFT;
```

Open Workbook

If the workbook with data is saved (as extension of ogw), it can be opened by the doc -o command.

```
// The path of the workbook to open
string strName$ = system.path.program$;
strName$ += "Samples\Graphing\Automobile Data.ogw";
// Open the workbook
doc -o %(strName$);
```

Save Workbook

Origin allows you to save a workbook with data to a file (*.ogw), or as a template without data (*.otw), and for the workbook with analysis, it is able to be saved as an analysis template (*.ogw).

1. The command save -i is able to save the active workbook with data to an ogw file.

```
// Create a new workbook
newbook;
// Fill some data to col(1)
col(1) = uniform(32);
// Save this workbook with data to MyData.ogw under User Files Folder
save -i %YMyData.ogw;
```

2. The X-Function template_saveas is used to save workbook as a template.

```
// Create a new workbook with 3 sheets
newbook sheet:=3;
// Save this workbook as a template named My3SheetsBook
// in User Files Folder (default)
template_saveas template:=My3SheetsBook;
```

3. To save a workbook with analysis, the command save -ik can be used.

```
// Create a project
string strOpj$ = system.path.program$ + "Samples\Analysis.opj";
doc -o %(strOpj$);

// Activate the workbook to be saved as analysis template
win -a Book1J;
```

```
// Save this workbook as an analysis template
// name MyAnalysis.ogw under User Files Folder
save -ik %YMyAnalysis.ogw;
```

Close Workbook

To close workbook, just click the Close button in the top right corner of the workbook. And this behovior is done by command win -ca, and a dialog pops up to prompt user to delete or hide the workbook.

```
// Create a workbook, and name is stored in MyBook$ variable
newbook result:=MyBook$;

// Simulate the Close button clicking
win -ca %(MyBook$);
```

To close the workbook directly without prompting, and delete all the data, you can use command win -cd. And this is the same with Delete Workbook below.

```
// Create a new workbook for closing
newbook;

// close this workbook without prompting, and delete all the data
win -cd %H;
```

Show or Hide Workbook

There are three switches, *-ch*, *-h*, and *-hc*, in win command for showing or hiding workbook.

```
// Create 3 workbooks for hiding
newbook name:=MyBook1 option:=lsname;  // first workbook, MyBook1
newbook name:=MyBook2 option:=lsname;  // second workbook, MyBook2
newbook name:=MyBook3 option:=lsname;  // third workbook, MyBook3;

// Use -ch to hide the active workbook, MyBook3
// And the View Mode in Project Explorer is Hidden
win -ch 1;

// Use -hc to hide the first workbook (not the active one), MyBook1
// And the View Mode in Project Explorer is Hidden
win -hc 1 MyBook1;

// Use -h to hide the second workbook (active workbook), MyBook2
// The View Mode in Project Explorer is still Normal
win -h 1;

// Actually, MyBook2 is still the active workbook
// It is able to show it by:
```

```
win -h 0;

// To show MyBook1 and MyBook3, need to use the -hc switch to specify
// the workbook name
win -hc 0 MyBook1;
win -hc 0 MyBook3;
```

Name and Label Workbook

For a workbook, there will be short name, Long Name, and Comments. You can rename (short name) a workbook with win -r command, and use the page object to control Long Name and Comments, including how to show the workbook title (short name, Long Name, or both).

```
// Create a new workbook with name of "Data",
// and show both in workbook title
// both short name and Long Name are the same
// workbook title only shows short name
newbook name:=Data option:=lsname;

// Rename the workbook to "RawData"
win -r Data RawData;

// Change Long Name to be "FFT Data"
page.longname$ = "FFT Data";
// Add Comments, "1st group data for fft"
page.comments$ = "1st group data for fft";

// Let the workbook title shows Long Name only
page.title = 1;  // 1 = Long Name, 2 = short name, 3 = both
```

Activate Workbook

To activate a workbook, the command win -a can be used.

```
// The path of project to be opened
string strOpj$ = system.path.program$;
strOpj$ += "Samples\Curve Fitting\Intro_to_Nonlinear_Curve_Fit_Tool.opj";
// Open the project
doc -o %(strOpj$);

// Activate workbook, Book1, in the second subfolder of the project
win -a Book1;

// It also can put the workbook name into a variable
// Variable for the name of workbook, Gaussian, in the project
string strGau$ = Gaussian;
// Activate the Gaussian workbook in the first subfolder
```

```
win -a %(strGau$);
```

Most Origin commands operate on the active window, so you may be tempted to use *win -a* to activate a workbook and then run script after it to assume the active workbook. This will work in simple codes but for longer script, there might be timing issues and we recommend that you use window -o *winName* {*script*} instead. See A Note about Object that a Script Operates upon for more detail explanation.

Delete Workbook

To delete a workbook, you can use the win -c command, and this command will delete the workbook directly without prompts.

```
// The path of project to be opened
string strOpj$ = system.path.program$ + "Samples\Curve Fitting\2D Bin and Fit.opj";
// Open the project
doc -o %(strOpj$);
// Delete workbook, Book1, from the project
win -c Book1;

// To delete an active workbook, the workbook name can be omitted
// Or using %H to refer to the workbook name
win -a MatrixFit1;  // Activate the workbook MatrixFit1
win -c;  // Delete the workbook
// Or using this one
// win -c %H;

// It also allows to delete a workbook whose name is stored in a variable
// Create a new workbook using newbook X-Function
// And the name of this workbook is stored in string variable ToDel$
newbook result:=ToDel$;
// delete the workbook created just now
win -c %(ToDel$);
```

8.1.2 Workbook Manipulation

Origin provides the capabilities for workbook manipulation by using LabTalk script, such as duplicating, merging, splitting, etc.

Duplicate Workbook

To duplicate active workbook, the win -d command is used. It allows to specify a name for the duplicated workbook, and the new workbook is activated after duplicated. The command win -da is doing the similar thing, however, it keeps the active workbook active after duplicated.

```
// Open a project
string strOpj$ = system.path.program$;
```

```
strOpj$ += "Samples\LabTalk Script Examples\Loop_wks.opj";

doc -o %(strOpj$);

// Activate the workbook S2Freq1
win -a S2Freq1;

// Duplicate this workbook, and name it "MyCopy"
// And this new workbook will be activated
win -d MyCopy;

// Duplicate the MyCopy workbook, and name it "MyCopy2"
// But keep MyCopy still activated
win -da MyCopy2;
```

Merge Workbooks

To merge multiple workbooks into one new workbook, the X-Function, merge_book, is available.

```
// Open a project
string strOpj$ = system.path.program$;
strOpj$ += "Samples\LabTalk Script Examples\Loop_wks.opj";

doc -o %(strOpj$);

// Activate Sample1 folder
pe_cd /Sample1;

// Merge two workbooks (S1Freq1 and S1Freq2) in two subfolders
// User the source workbook name for the worksheet name in the merged workbook
merge_book fld:=recursive rename:=sname;

// Activate Sample2 folder
pe_cd /Sample2;

// Merge two workbooks (S2Freq1 and S2Freq2) in two subfolders
// User the source workbook name for the worksheet name in the merged workbook
merge_book fld:=recursive rename:=sname;

// Activeate the root folder
pe_cd /;

// Two new workbooks are created from the above script
// The names of these two workbooks begin with "mergebook"
// Now, merge these two workbooks into a new workbooks
```

```
// The worksheets in the final result workbook will name
// by using the original worksheet name
merge_book fld:=project single:=0 match:=wkbshort key:="mergebook*" rename:=wksname;
```

Split Workbook

The example above is merging multiple workbooks into one workbook. It is also able to split a workbook into multiple workbooks, which contain single worksheet. The wsplit_book X-Function is designed for this purpose.

```
// Open a project
string strOpj$ = system.path.program$;
strOpj$ += "Samples\Automation Server\Basic Stats on Data.opj";

doc -o %(strOpj$);

// There are three worksheets in the active workbook, RawData
// Now split this workbook into three workbooks
// And each workbook will contain one worksheet from the original workbook
wsplit_book fld:=active;
```

8.2 Worksheets

8.2.1 Basic Worksheet Operation

The basic worksheet operations include adding worksheet to workbook, activating a worksheet, getting and setting worksheet properties, deleting worksheet, etc. And these operations can be done by using Page and Wks objects, together with some Data Manipulation X-Functions. Some practical examples are provided below.

Add New Worksheet

The **newsheet** X-Function can be used to add new worksheets to a workbook.

```
// Create a new workbook with 3 worksheets,
// and use "mydata" as long name and short name
newbook name:="mydata" sheet:=3 option:=lsname;
// Add a worksheet named "source" with 4 columns to current workbook
newsheet name:=source cols:=4;
```

Activate a Worksheet

Workbook is an Origin object that contains worksheets which then contain columns. Worksheets in a workbook are internally layers in a page. In other words, a worksheet is derived from a layer object and a workbook derived from a page object. The active layer in a

page is represented by the *page.active* or *page.active$* property, and thus it is used to active a worksheet.

```
// Create a new workbook with 4 worksheets
newbook sheet:=4;

page.active = 2;  // Active worksheet by index
page.active$ = sheet3;     // Active worksheet by name
```

Modify Worksheet Properties

Using Worksheet Object

When a worksheet is active, you can type **wks.=** and press Enter to list all worksheet properties. Most of these properties are writable so you can modify it directly. For example:

```
// Rename the active worksheet
wks.name$ = Raw Data
// Set the number of columns to 4
wks.ncols = 4
// Modify the column width to 8 character
wks.colwidth = 8
// Show the first user-defined parameter on worksheet header
wks.userparam1 = 1
```

Two properties, wks.maxRows and wks.nRows are similar. The former one find the largest row index that has value in the worksheet, while the later set or read the number of rows in the worksheet. You can see the different in the following script.

```
newbook;  // Create a new workbook
col(b) = {1:10};  // Set column B with 1-10 for the first ten rows
wks.maxRows = ;
wks.nRows = ;
```

Origin outputs 10 for wks.maxRows; while outputs 32 for wks.nRows.

If the worksheet is not the active one, you can specify the full worksheet name (including workbook name) before **wks** object, the syntax is

[*WorkbookName*]*WorksheetNameOrIndex*!wks

Or you can use the range of the worksheet. For example

```
// Open a project
string strOpj$ = system.path.program$;
strOpj$ += "Samples\Automation Server\Basic Stats on Data.opj";
doc -o %(strOpj$);

wks.nCols = ;  // Output number of columns in the active worksheet
// Output number of columns in the worksheet [RawData]Data!
[RawData]Data!wks.nCols = ;
// Output the name of the first worksheet in RawData workbook
[RawData]1!wks.name$ = ;
```

```
// Use range
range rWks = [RawData]Data!;  // Range for the Data worksheet in RawData workbook
rWks.userparam1 = 1;  // Show the first user-defined parameter in worksheet
```

Using X-Functions

Besides wks object, you can also use X-Functions to modify worksheet properties. These X-Function names are usually with the starting letter "w". Such as wcolwidth, wcellformat and wclear, etc. So we can also resize the column with as below without using wks.colwidth:

```
wcolwidth 2 10; // Set the 2nd column width to 10
```

Delete Worksheet

The **layer -d** command can be used to delete a worksheet or graph layer.

```
// Create a new workbook with 6 worksheets
// Workbook name is stored into MyBook$
// And the first worksheet will be the active one
newbook sheet:=6 result:=MyBook$;

// Add a new worksheet with name of "My Sheet"
newsheet name:="My Sheet";

page.active = 1;  // Activate the first worksheet
layer -d;  // Delete the active worksheet

// Delete a worksheet by index
// Delete the third worksheet (or layer) in the active workbook (or graph)
layer -d 3;

// Delete a worksheet by name
layer -d "Sheet5";

// Delete a specified worksheet by range
range rs = [%(MyBook$)]"My Sheet"!;  // Define a range to a specified worksheet
layer -d rs;

// Delete a worksheet whose name is stored in a string variable
string strSheet$ = "Sheet3";
layer -d %(strSheet$);
```

To delete a worksheet whose name is stored in a string variable, there are some special string variables for some special worksheets, for example:

```
//__report$ holds the name of the last report sheet Origin created
layer -d %(__report$);
```

The variable **__report$** is an example of a system-created string variable that records the last-used instance of a particular object. A list of such variables can be found in Reference Tables.

8.2.2 Worksheet Data Manipulation

In this section we present examples of X-Functions for basic data processing. For direct access to worksheet data, see Range Notation.

Copy Worksheet Data

Copy a Worksheet

The wcopy X-Function is used to create a copy worksheet of the specified worksheet.

The following example duplicates the current worksheet, creating a new workbook with the copied worksheet:

```
wcopy 1! [<new>]1!;
```

Copy a Range of Cells

The wrcopy X-Function is used to copy a range of cells from one worksheet to another. It also allows you to specify a source row to be used as the Long Names in the destination worksheet.

The following script copies rows from 5 to 9 of [book1]sheet1! to a worksheet named CopiedValues in Book1 (if the worksheet does not exist it will be created), and assigns the values in row 4 from [book1]sheet1! to the long name of the destination worksheet, [book1]CopiedValues!

```
wrcopy iw:=[book1]sheet1! r1:=5 r2:=10 name:=4 ow:=CopiedValues!;
```

To copy column and matrix object, please refer to Copy Column and Copy Matrix Data respectively.

Reduce Worksheet Data

Origin has several data reducing X-Functions like reduce_ex, reducedup, reducerows and reducexy. These X-Functions provide different ways of creating a smaller dataset from a larger one. Which one you choose will depend on what type of input data you have, and what type of output data you want.

Examples

The following script will create a new X and Y column where the Y will be the mean value for each of the duplicate X values.

```
reducedup col(B);
```

The following script will reduce the active selection (which can be multiple columns or an entire worksheet, independent of X or Y plotting designation) by a factor of 3. It will remove rows 2 and 3 and then rows 5 and 6, leaving rows 1 and 4, etc. By default, the reduced values will go to a new worksheet.

```
reducerows npts:=3;
```

The following script will average every *n* numbers (5 in the example below) in column A and output the average of each group to column B. It is the same as the ave LabTalk function, which would be written as col(b)=ave(col(a),5):

```
reducerows irng:=col(A) npts:=5 method:=ave rd:=col(b);
```

Extract Worksheet Data

Partial data from a worksheet can be extracted using conditions involving the data columns, using the **wxt** X-function.

```
// Import a sample data file
newbook;
string fname$ = system.path.program$ + "samples\statistics\automobile.dat";
impasc;
// Define range using some of the columns
range rYear=1, rMake=2, rHP=3;
type "Number of rows in raw data sheet= $(rYear.GetSize())";
// Define a condition string and extract data
// to a new named sheet in the same book
string strCond$="rYear >= 1996 and rHP<70 and rHP>60 and rMake[i]$=Honda";
wxt test:=strCond$ ow:="Extracted Rows"! num:=nExtRows;
type "Number of rows extracted = $(nExtRows)";
```

Output To New Workbook

You can also direct the output to a new workbook, instead of a new worksheet in the existing workbook, by changing the following line:

```
wxt test:=strCond$ ow:=[<new name:="Result">]"Extracted"! num:=nExtRows;
```

As you can see, the only difference from the earlier code is that we have added the workbook part of the range notation for the **ow** variable, with the **<new>** keyword. (show links and indexing to <new> modifiers, options, like template, name, etc)

Use Wildcard Search

LabTalk uses * and ? characters for wildcard in string comparison. You can try changing the **strCond** as follows:

```
string strCond$ = "rYear >= 1996 and rHP<70 and rHP>60 and rMake[i]$=*o*";
```

to see all the other makes of cars with the letter **o**.

Delete Worksheet Data

Deleting the *N*th row can be accomplished with the **reducerows** X-Function, described above.

This example demonstrates deleting every *N*th column in a worksheet using a for-loop:

```
int ndel = 3; // change this number as needed;
```

```
int ncols = wks.ncols;
int nlast = ncols - mod(ncols, ndel);
// Need to delete from the right to the left
for(int ii = nlast; ii > 0; ii -= ndel)
{
   delete wcol($(ii));
}
```

Sort Worksheet

The following example shows how to perform nested sorting of data in a worksheet using the **wsort** X-Function:

```
// Start a new book and import a sample file
newbook;
string fname$ = system.path.program$ + "Samples\Statistics\automobile.dat";
impasc;
// Set up vectors to specify nesting of columns and order of sorting;
// Sort nested: primary col 2, then col 1, then col 3:
dataset dsCols = {2, 1, 3};

// Sort col2 ascending, col 1 ascending, col 3 descending:
dataset dsOrder = {1, 1, 0};

wsort nestcols:=dsCols order:=dsOrder;
```

Split Worksheet

Origin provides the X-Function, wsplit, for the purpose of splitting one worksheet's columns into multiple worksheets.

The example below is going to import multiple CSV files, and then get the Amplitude data from all the data file into a worksheet, and then convert this worksheet to matrix to make a contour plot.

```
// Create a new workbook
newbook;

// Find all csv files in the specified folder
string strPath$ = system.path.program$ + "Samples\Batch Processing\";
findfiles path:=strPath$ fname:=csvFiles$ ext:=csv;

// Import all found csv files into one worksheet
impCSV  fname:=csvFiles$  // All found csv files
   options.Mode:=1  // From second file, start new column
   options.names.FNameToSht:=0  // Not rename worksheet
   options.names.FNameToBk:=0  // Not rename workbook
```

```
   options.HeaderLines.SubHeaderLines:=2  // Two subheader lines
   options.HeaderLines.LongNames:=1  // First subheader line is LongName
   options.HeaderLines.Units:=2;  // Second subheader line is Units

// Split the worksheet according to the Long Name
// The columns with the same Long Name will be in the same result worksheet
// All the result worksheets will be in the same new workbook
wsplit mode:=label label:=L;

// Activate the Amplitude worksheet
page.active$ = Amplitude;

// Convert the Amplitude worksheet to matrix directly
w2m;

// Make a contour for the amplitude
worksheet -p 226 contour;
```

Unstack/Stack Categorical Data

Unstack Worksheet Columns

At times unstacking categorical data is desirable for analysis and/or plotting purposes. The wunstackcol X-Function is the most convenient way to perform this task from script.

In this example, categorical data is imported, and we want to unstack the data by a particular category, which we specify with the input range **irng2**. The data to be displayed (by category) is referenced by input range **irng1**. In this example, the column ranges are input directly, but range variables can also be used.

```
// Import automobile data
newbook;
string fpath$ = "\Samples\Statistics\Automobile.dat";
string fname$ = system.path.program$ + fpath$;
impasc;

// Unstack all other columns using automobile Make, stored in col 2
// Place "Make" in Comments row of output sheet
wunstackcol irng1:=(1, 3:7) irng2:=2 label:="Comments";
```

The result is a new worksheet with the unstacked data.

Stack Worksheet Columns

Staking categorical data is something like reverse operation of unstacking categorical data. In the original dataset, samples belong to different groups is stored in different columns. After stacking, the samples will be in different rows in the same column, with an additional column in

the worksheet providing the group information. You can use wstackcol to stack worksheet columns.

In the following example, we open a workbook with categorical data first. And then with the first worksheet activated, and stack column B, C, and D by rows, including another column to be A.

```
// Open a workbook
string strBook$ = system.path.program$;
strBook$ += "Samples\Statistics\Body.ogw";
doc -o %(strBook$);

// Stack column B, C, D in Male worksheet
// Include column A as another column
// Method is By Rows
wstackcol irng:=(2:4) tr.identifiers:={L} include:=1 method:=1;
```

The result is a new worksheet in the same workbook with the stacked data.

Pivot Table

The wpivot X-Function is available for the purpose of quickly summarizing the data, so to analyze, compare, and detect the relationships in the data. That is an easy way to present data information.

```
// Create a new workbook
// And import a data file
newbook;
fname$ = system.path.program$ + "Samples\Statistics\HouseholdCareSamples.xls";
impExcel lname:=1;

// Make sure "HQ Family Mart" worksheet is activate
// And make a copy of this worksheet
page.active$ = "HQ Family Mart";
wcopy ow:=[<new>]"HQ Family Mart"!;

// Pivot table, row source is Make
// Column source is Brand, and data is Number in shelf
// The result will show the number of products in shelf
// for different brands and different makes
wpivot  row:=col(D) col:=col(F) data:=col(K)
   method:=sum total:=1 sort_total:=no sum:=1;

// Activate the source data worksheet
page.active$ = "HQ Family Mart";

// Pivot table, row source, column source and data column are the same
// For the smaller values, it will combine them across columns
// by 10% of the total
```

```
// In the result worksheet, the column info. is put to user-defined parameter rows
// The row name is the name of column's Long Name in source worksheet
wpivot  row:=col(D)
   col:=col(F)
   data:=col(K)
   method:=sum total:=1 sort_total:=no sum:=1
   dir:=col threshold:=10  // Combine smaller values across column, by 10%
   // Put column info (from column's Long Name) to user-defined parameters row
   pos:=udl udlabel:=L;
```

Worksheet Filter

Worksheet Filter (Data Filter) in Origin is column-based filter to reduce rows of worksheet data by using the specified condition, so to hide the undesired rows for relevant data analysis and graphing. Three data formats are supported: numeric, text and date/time. In LabTalk, you can use the wks.col (wks.col.filter, wks.col.filter$, wks.col.filterenabled, wks.col.filterprescript$, and wks.col.filterx$) object to handle the data filter. And to run/re-apply the filter, use the wks.runfilter() method.

```
// Create a new workbook, and import the data
newbook;
string fname$ = system.path.program$ + "Samples\Statistics\Automobile.dat";
impasc;

// Set data filter for column 1, numeric type
wks.col1.filter = 1;  // Add filter
wks.col1.filterx$ = year;  // Set the variable "year" to represent column 1
// Set filter condition, between 1995 and 2000
wks.col1.filter$ = "year.between(1995,2000)";

// Set data filter for column 2, text type
wks.col2.filter = 1;  // Add filter
wks.col2.filterx$ = make;  // Set the variable "make" to represent column 2
// Set before query script
wks.col2.filterprescript$ = "string strFavorite$ = GMC";
wks.col2.filter$ = "make = strFavorite$";  // Set filter query string

// Run the worksheet filter
wks.runfilter();

// Disable the filter in column 1
wks.col1.filterenabled = 0;

// Re-apply the worksheet filter
wks.runfilter();
```

To detect whether there is filter in a worksheet, you can use the wks.hasfilter() method.

```
// If the active worksheet has filter, return 1, otherwise, return 0
wks.hasfilter() = ;
```

Insert Links into Worksheet Cells

Origin provides several cell linking syntax for adding links into worksheet cell, so to easily access or display the linked resource. Available cell linking syntax are included in the following table.

Syntax	Description	Example
cell://*CellName*	Insert a link to another cell, given by *CellName*, so to display its contents in the current worksheet cell.	cell://[Book2]Sheet2!Col(B)[3]
range://*RangeName* [*DisplayedText*]	Insert a link to a range, given by *RangeName*, which can be book, sheet, column, etc. The contents in the current worksheet cell will show as a link, if clicked, the corresponding range will be activated. If the option *DisplayedText* is included, the displayed text in the cell is this text, but not the range.	range://[Book2]Sheet2!Col(B)[3]
graph://*GraphName*	Insert a link to a graph, given by *GraphName*, so the graph will be displayed in the current worksheet cell. If double-click on this cell, the corresponding graph window will be activated.	graph://Graph1
matrix://*MatrixObjectName*	Insert a link to a matrix object, given by *MatrixObjectName*, so the matrix object will be displayed in the current worksheet cell, as an image. If double-click on this cell, the corresponding matrix window will be activated.	matrix://[MBook1]MSheet1!2
notes://*NotesWindowName* [*DisplayedText*]	Insert a link to a Notes window, given by *NotesWindowName*. The contents in this cell will show as a link, if clicked, the corresponding Notes window will be activated.	notes://Notes

var://*LabTalkVariableName*	Insert a link to a LabTalk variable, and the value of this variable will show in this cell.	var://MyVar
str://*LabTalkStringVariable*	Insert a link to a LabTalk string variable, and this string will show in this cell.	str://MyBook$
http://*URL* [*DisplayedText*]	Insert a live URL into worksheet cells. The link will become active when finishing editing, and the corresponding page will be opened in a web browser if clicked. If the option *DisplayedText* is included, the displayed text in the cell is this text, but not the URL.	http://www.originlab.com
help://*HelpPage* [*DisplayedText*]	If the option *DisplayedText* is included, the displayed text in the cell is this text, but not the help link.	help://TUTORIAL.CHM/Tutorial/Import_Wizard.html
file://*FilePath*	Insert a link to an image file, given by *FilePath*. And the linked image will display in the current cell.	file://D:\Flower.jpg

8.2.3 Converting Worksheet to Matrix

You may need to re-organize your data by converting from worksheet to matrix, or vice versa, for certain analysis or graphing needs. This page provides information and examples of converting worksheet to matrix, and please refer to Converting Matrix to Worksheet for the "vice versa" case.

Worksheet to Matrix

Data contained in a worksheet can be converted to a matrix using a set of Gridding X-Functions.

The w2m X-Function converts matrix-like worksheet data directly into a matrix. Data in source worksheet can contain the X or Y coordinate values in the first column, first row, or a header row. However, because the coordinates in a matrix should be uniform spaced, you should have uniformly spaced X/Y values in the source worksheet.

If your X/Y coordinate values are not uniform spaced, you should use the Virtual Matrix feature instead of converting to a matrix.

The following example show how to perform direct worksheet to matrix conversion:

```
// Create a new workbook
newbook;
// Import sample data
string fname$ = system.path.program$ +
   "\samples\Matrix Conversion and Gridding\DirectXY.dat";
impasc;
// Covert worksheet to matrix, first row will be X and first column will be Y
w2m xy:=xcol xlabel:=row1 ycol:=1;
// Show X/Y values in the matrix window
page.cntrl = 2;
```

When your worksheet data is organized in XYZ column form, you should use Gridding to convert such data into a matrix. Many gridding methods are available, which will interpolate your source data and generate a uniformly spaced array of values with the X and Y dimensions specified by you.

The following example converts XYZ worksheet data by Renka-Cline gridding method, and then creates a 3D graph from the new matrix.

```
// Create a new workbook without sheets
newbook;
// Import sample data
string fname$ = system.path.program$ +
   "\samples\Matrix Conversion and Gridding\XYZ Random Gaussian.dat";
impasc;
// Convert worksheet data into a 20 x 20 matrix by Renka-Cline gridding method
xyz_renka 3 20 20;
// Plot a 3D color map graph
worksheet -p 242 cmap;
```

8.2.4 Virtual Matrix

Data arranged in a group of worksheet cells can be treated as a matrix and various plots such as 3D Surface, 3D Bars, and Contour can be created from such data. This feature is referred to as Virtual Matrix. The X and Y coordinate values can be optionally contained in the block of data in the first column and row, or also in a header row of the worksheet.

Whereas Matrix objects in Origin only support linear mapping of X and Y coordinates, a virtual matrix supports nonlinear or unevenly spaced coordinates for X and Y.

The virtual matrix is defined when data in the worksheet is used to create a plot. The plotvm X-Function should be used to create plots.

The following example shows how to use the plot_vm X-Function:

```
// Create a new workbook and import sample data
newbook;
string fname$=system.path.program$ + "Samples\Graphing\VSurface 1.dat";
impasc;
// Treat entire sheet as a Virtual Matrix and create a colormap surface plot
```

```
plotvm irng:=1! format:=xacross rowpos:=selrow1 colpos:=selcol1
 ztitle:="VSurface 1" type:=242 ogl:=<new template:=cmap>;
// Change X axis scale to log
layer.x.type=2;
```

8.3 Worksheet Columns

8.3.1 Basic Worksheet Column Operation

To perform operation on worksheet column, in most situation, you can use wks.col object, or the Range Notation to the column object.

Add or Insert Column

To add a column to the end of the worksheet, you can use the wks.addCol() method, which will add a column with the specified name, if the specified name is used or ignored, a generic name is chosen for the newly added column.

```
// Create a new workbook
newbook;

// Add a new column to the end, with name of Result
wks.addCol(Result);
```

The method above is only able to add one column to the end at a time. If you are going to add a multiple columns, you can add columns by setting the number of columns in the worksheet with the wks.nCols property. For example, the script below will add 3 columns to the end of the active worksheet with the generic names (Note: it is not able to specify the names in this way, please refer to **Rename and Label Column** section below).

```
// Create a new workbook
newbook;

// Add 3 columns to the end of worksheet
wks.nCols = wks.nCols + 3;
```

Besides adding columns to the end of the worksheet, it is also capable of inserting numbers of columns before the current column. First of all, it needs to specify which column (by 1-based index) is the current column using wks.col property, and then using wks.insert() method to insert column(s) before the current column. In the method, you need to specify a list of column names separated by space.

```
// Create a new workbook
newbook;

// Set column 2 to be the current column
wks.col = 2;
```

```
// Insert 3 column before column 2, with the specify column names
wks.insert(DataX DataY Result);
```

Move Column

The colmove X-Function allows you to move column(s) of data within a worksheet. It accepts an explicitly stated range (as opposed to a range variable), and the type of move operation as inputs.

```
// Make the first column the last (left to right) in the worksheet:
colmove rng:=col(1) operation:=last;

// Move columns 2-4 to the leftmost position in the worksheet:
colmove rng:=Col(2):Col(4) operation:=first;
```

Rename and Label Column

To rename (short name) a column, Origin provides the wks.col object with the *name$* property. Also, the Column Label Row Characters, ***G***, is able to rename column short name.

```
// Create a new workbook
newbook;

// Rename column 1 to DataX
wks.col1.name$ = DataX;
// Rename column 2 to DataY by using range
range rY = 2;  // range to column 2
rY.name$ = DataY;

// Add a new column
wks.addCol();
// Rename it with "G"
col(3)[G]$ = "Result";
```

The Column Label Row Characters are the convenient way to access the column labels, including Long Name, Units, Comments, Column Parameters, User-Defined Parameters, etc.

```
// Create a new workbook
newbook result:=BkName$;

// Show the following label rows:
// Long Name, Units, Comments, 1st Column Parameter
// and 1st User-Defined Parameter
wks.labels(LUCP1D1);

// Ranges to column 1 and 2
range r1 = [%(BkName$)]1!1;
range r2 = [%(BkName$)]1!2;
```

```
// Set Long Name by using col
col(1)[L]$ = Time;
col(2)[L]$ = Voltage;
// Set Units by using range
r1[U]$ = Sec;
r2[U]$ = V;
// Set Comments by using range
r1[C]$ = Sample1;
r2[C]$ = Sample1;
// Set Column Parameters by using range
r1[P1]$ = "Machine1";
r2[P1]$ = "Machine1";

// Rename the 1st User-Defined Parameter
wks.UserParam1$ = Current;

// Set Current label row
r1[Current]$ = 1mA;
r2[Current]$ = 1mA;
```

Hide/Unhide Column

To hide/unhide column(s), you can use the colHide X-Function.

```
// Create a new workbook
newbook;

// Set worksheet column number to 6
wks.nCols = 6;

// Hide the second column
colHide 2 hide;

// Hide the 3rd and 5th columns
colHide (3, 5) hide;
```

To show (unhide) column(s), it just changes the second argument from *hide* to *unhide*.

Swap Column

The colSwap X-Function is used to swap two specified columns.

```
// Create a new workbook
newbook;

// Swap the position of the 1st and 2nd columns
colSwap (1, 2);
```

The specified two columns is not needed to be adjacent.

```
// Create a new workbook
newbook;

// Set number of columns to be 6
wks.ncols = 6;

// Swap the 2nd and 4th columns
colswap (2, 4);
```

Modify Column Formats

Plot Designation

Plot designation for a column determines how the selected data will be handled by default for plotting and data analysis. Plot designation includes X, Y, Z, Z Error, Y Error, Label, etc. And you can change it by using wks.col.type.

```
// Import data
newbook;
string fname$ = system.path.program$;
fname$ += "Samples\Matrix Conversion and Gridding\XYZ Random Gaussian.dat";
impasc;

// Set column designation (column type)
wks.col = 3;  // Set column 3 to be current column
wks.col.type = 6;  // Z

// Select the 3rd column (Z column)
worksheet -s 3 1 3 -1;
// Make a color map surface with the template based on OpenGL
worksheet -p 103 glcmap;
```

Column Width

To set column width, the wcolwidth X-Function is available, or use wks.col.width.

```
// Open a workbook
string strPath$ = system.path.program$;
strPath$ += "Samples\Graphing\Automobile Data.ogw";
doc -o %(strPath$);

// To make column 2 show all the numbers but not ###
// Set width of column 2 to 6 characters
wcolwidth irng:=col(2) width:=6;
```

Data Format and Display

Setting a correct data format for a column helps to display the data in the column correctly, also helps to perform operations, such plotting, data analysis, etc. properly. There are many data format available for a column, such as Numeric, Text, Date, Time, Month, Day of Week, etc. To set format, please use wks.col object's *format* property.

```
// Import data
newbook;
string fname$ = system.path.program$;
fname$ += "Samples\Signal Processing\Average Sunspot.dat";
impasc;

// Set column 2 to Numeric (current is Text & Numeric)
wks.col2.format = 1;  // Numeric = 1

// Enable digit mode to be "Set Decimal Places"
// and set number of decimal places to 2
wks.col2.digitMode = 1;  // Set Decimal Places
wks.col2.digits = 2;  // Two decimal places
```

The following examples are showing the corresponding settings for different format.

1. Numeric

```
// Import data
newbook;
string fname$ = system.path.program$;
fname$ += "Samples\Curve Fitting\Enzyme.dat";
impasc;

// Set column 2 to Numeric (current is Text & Numeric)
wks.col2.format = 1;  // Numeric = 1
// Set display format with comma
wks.col2.subformat = 4;  // Display as Decimal: 1,000
// Data type to be short int
wks.col2.numerictype = 3;

// Do the same for column 3
wks.col3.format = 1;  // Numeric = 1
// Set display format with comma
wks.col3.subformat = 4;  // Display as Decimal: 1,000
// Data type to be short int
wks.col3.numerictype = 3;
```

2. Date

   For Date and Time format, if the data stored in a column is not Julian day numbers (looks like Date and Time format, actually is text), we cannot set the format as Date

or Time directly, or the look-like-Date-and-Time-format text will become missing value or something incorrect. To avoid this issue, Origin provides the **wks.col.setformat()** method.

```
// Import data
newbook;
string fname$ = system.path.program$;
fname$ += "Samples\Import and Export\Custom Date and Time.dat";
impasc;

// Set format of column 1 to be Date
// with a custom display format, which is like
// the current text display in the column
wks.col1.setformat(4, 22, dd'.'MM'.'yyyy HH':'mm':'ss'.'##);
// Set a familiar display format yyyy/MM/dd HH:mm:ss
wks.col1.subformat = 11;
```

3. Time

   Please refer to the description about Date above.

```
// Import data
newbook;
string fname$ = system.path.program$;
fname$ += "Samples\Import and Export\IRIG Time.dat";
impasc;

// Set format of column 1 to be Time
wks.col1.format = 3;  // Time = 3
// Display IRIG Time format DDD:HH:mm:ss.##
wks.col1.subformat = 16;
```

4. Month

```
// Set column 1 format as Month
// And show the whole name of month
wks.col1.format = 5;  // Month = 5
wks.col1.subformat = 2;  // Show the whole month's name
```

5. Day of Week

```
// Set column 1 format as Day of Week
// And show only the first letter of each day of week
wks.col1.format = 6;  // Day of Week = 6
wks.col1.subformat = 3;  // Show the first letter of each day of week
```

Add Sparkline to Column

The sparklines X-Function is used to add sparklines to the specified columns in the worksheet.

```
// Open a workbook
string strPath$ = system.path.program$;
strPath$ += "Samples\Graphing\Automobile Data.ogw";
doc -o %(strPath$);

// Turn on sparklines for all columns except the ones with "Year" Long Name
for(ii = 2; ii <= wks.nCols; ii+=5)
{
   sparklines sel:=0 c1:=ii c2:=ii+3;
}
```

Delete Column

The delete command is capable of removing a column from worksheet.

```
// Create a workbook
newbook;

// Delete column B
delete col(B);

// Add a new worksheet with 4 columns
newsheet cols:=4;

// Delete column 3 by using range
range rr = 3;  // column 3 in the newly added worksheet
delete rr;
```

If the column(s) you want to delete is (are) at the end of the worksheet, you can just set the number of worksheet columns to delete it (them), by using wks.nCols.

```
// Open a workbook
string strPath$ = system.path.program$;
strPath$ += "Samples\Graphing\Automobile Data.ogw";
doc -o %(strPath$);

// Delete last 20 columns from the opened worksheet
wks.nCols = wks.nCols-20;
```

8.3.2 Worksheet Column Data Manipulation

Basic Operation

Once you have loaded or created some numeric data, here are some script examples of things you may want to do.

Basic Arithmetic

Most often data is stored in columns and you want to perform various operations on that data in a row-wise fashion. You can do this in two ways in your LabTalk scripts: (1) through direct statements with operators or (2) using ranges. For example, you want to add the value in each row of column A to its corresponding value in column B, and put the resulting values in column C:

```
Col(C) = Col(A) + Col(B);  // Add
Col(D) = Col(A) * Col(B);  // Multiply
Col(E) = Col(A) / Col(B);  // Divide
```

The **-** and **^** operators work the just as above for subtraction and exponentiation respectively.

You can also perform the same operations on columns from different sheets with range variables:

```
// Point to column 1 of sheets 1, 2 and 3
range aa = 1!col(1);
range bb = 2!col(1);
range cc = 3!col(1);
cc = aa+bb;
cc = aa^bb;
cc = aa/bb;
```

When performing arithmetic on data in different sheets, you need to use range variables. Direct references to range strings are not supported. For example, the script **Sheet3!col(1) = Sheet1!col(1) + Sheet2!col(1);** will not work!

Functions

In addition to standard operators, LabTalk supports many common functions for working with your data, from trigonometric functions like **sin** and **cos** to Bessel functions to functions that generate statistical distributions like **uniform** and **Poisson**. All LabTalk functions work with single-number arguments of course, but many are also "vectorized" in that they work on worksheet columns, loose datasets, and matrices as well. Take the trigonometric function **sin** for example:

```
// Find the sine of a number:
double xx = sin(0.3572)
// Find the sine of a column of data (row-wise):
Col(B) = sin(Col(A))
// Find the sine of a matrix of data (element-wise):
```

```
[MBook2] = sin([MBook1])
```

As an example of a function whose primary job is to generate data consider the **uniform** function, which in one form takes as input *N*, the number of values to create, and then generates *N* uniformly distributed random numbers between 0 and 1:

```
/* Fill the first 20 rows of Column B
   with uniformly distributed random numbers: */
Col(B) = uniform(20);
```

For a complete list of functions supported by LabTalk see Alphabetic Listing of Functions.

Set Formula for Column

In the Origin GUI, the Set Column Values dialog can be used to generate or transform data in worksheet columns using a specified formula. Such transformation can also be performed in LabTalk by using the csetvalue X-Function. Here are some examples on how to set column value using LabTalk.

```
newbook;
wks.ncols = 3;
// Fill column 1 with random numbers
csetvalue formula:="rnd()" col:=1;
// Transform data in column 1 to integer number between 0 ~ 100
csetvalue formula:="int(col(1)*100)" col:=2;
// Specify Before Formula Script when setting column value
// and set recalculate mode to Manual
csetvalue formula:="mm - col(2)" col:=3 script:="int mm = max(col(2))"
recalculate:=2;
string str$ = [%h]%(page.active$)!;
newsheet cols:=1;
// Use range variables to refer to a column in another sheet
csetvalue f:="r1/r2" c:=1 s:="range r1=%(str$)2; range r2=%(str$)3;" r:=1;
```

Copy Column

The colcopy X-Function copies column(s) of data including column label rows and column format such as date or text and numeric.

The following example copies columns two through four of the active worksheet to columns one through three of sheet1 in book2:

```
// Both the data and format as well as each column long name,
// units and comments gets copied:
colcopy irng:=(2:4) orng:=[book2]sheet1!(1:3) data:=1
        format:=1 lname:=1 units:=1 comments:=1;
```

Sort Column

To sort a specified column, you can use wsort X-Function. And when using this X-Function to sort just one column, the arguments **c1** and **c2** should be the same column in worksheet, and the **bycol** also needs to be the same as **c1**.

```
// Create a new workbook
newbook;

// Fill first column with row number, and second column with uniform random number
col(1) = {1:32};
col(2) = uniform(32);

// Sort column 2 descending
wsort c1:=2 c2:=2 bycol:=2 descending:=1;
```

Reverse Column

The X-Function colreverse is available for reversing column.

```
// Create a new workbook
newbook;

// Fill first column with row number, and second column with uniform random number
col(1) = {1:32};
col(2) = uniform(32);

// Reverse column 1 by using index
colreverse rng:=1;  // colreverse rng:=col(A);  // this also works

// Reverse column 2 by using range variable
range rr = 2;
colreverse rng:=rr;
```

8.3.3 Date and Time Data

While the various string formats used for displaying date and time information are useful in conveying information to users, a mathematical basis for these values is needed to provide Origin with plotting and analysis of these values. Origin uses a modification of the Astronomical Julian Date system to store dates and time. In this system, time zero is 12 noon on January 1, 4713 BCE. The integer part of the number represents the number of days since time zero and the fractional part is the fraction of a 24 hour day. Origin offsets this value by subtracting 12 hours (0.50 days) to put day transitions at midnight, rather than noon.

The next few examples are dedicated to dealing with date and time data in your LabTalk scripts.

Note : Text that appears to be **Date** or **Time** may in fact be **Text** or **Text & Numeric** which would not be treated as a numeric value by Origin. Use the Column Properties dialog (double-click a column name or select a column and choose Format : Column) to convert a **Text** or **Text & Numeric** column to **Date** or **Time** Format. The Display format should match the text format in your column when converting.

Dates and Times

As an example, say you have Date data in Column 1 of your active sheet and Time data in Column 2. You would like to store the combined date-time as a single column.

```
/* Since both date and time have a mathematical basis,
   they can be added: */
Col(3) = Col(1) + Col(2);

// By default, the new column will display as a number of days ...
/* Use format and subformat methods to set
   the date/time display of your choice: */

// Format #4 is the date format
wks.col3.format = 4;
// Subformat #11 is MM/dd/yyyy hh:mm:ss
wks.col3.subformat = 11;
```

The column number above was hard-coded into the format statement; if instead you had the column number as a variable named **cn**, you could replace the number **3** with **$(cn)** as in **wks.col$(cn).format = 4**. For other **format** and **subformat** options, see LabTalk Language Reference: Object Reference: Wks.col (object).

If our date and time column are just text with a **MM/dd/yyyy** format in Column 1 and **hh:mm:ss** format in Column 2, the same operation is possible with a few more lines of code:

```
// Get the number of rows to loop over.
int nn = wks.col1.nrows;

loop(ii,1,nn){
  string dd$ = Col(1)[ii]$;
  string tt$ = Col(2)[ii]$;
  // Store the combined date-time string just as text
  Col(3)[ii]$ = dd$ + " " + tt$;
  // Date function converts the date-time string to a numeric date value
  Col(4)[ii] = date(%(dd$) %(tt$));
};
// Now we can convert column 4 to a true Date column
wks.col4.format = 4; // Convert to a Date column
wks.col4.subformat = 11; // Display as M/d/yyyy hh:mm:ss
```

Here, an intermediate column has been formed to hold the combined date-time as a string, with the resulting date-time (numeric) value stored in a fourth column. While they appear to be the same text, column C is literally just text and column D is a true Date.

Given this mathematical system, you can calculate the difference between two Date values which will result in a Time value (the number of days, hours and minutes between the two dates) and you can add a Time value to a Date value to calculate a new Date value. You can also add Time data to Time data and get valid Time data, but you cannot add Date data to Date data.

Formatting for Output

Available Formats

Use the **D** notation to convert a numeric date value into a date-time string using one of Origin's built-in Date subformats:

```
type "$(@D, D10)";
```

returns the current date and time (stored in the system variable **@D**) as a readable string:

```
7/20/2009 10:30:48
```

The **D10** option corresponds to the **MM/dd/yyyy hh:mm:ss** format. Many other output formats are available by changing the number after the D character, which is the index entry (from 0) in the **Date Format** drop down list of the Worksheet Column Format dialog box, in the line of script above. The first entry (index = 0) is the Windows Short Date format, while the second is the Windows Long Date format.

Note : *The **D** must be uppercase. When setting a worksheet subformat as in wks.col3.subformat = #, these values are indexed from 1.*

For instance

```
type "$(date(7/20/2009), D1)";
```

produces, using U.S. Regional settings,

```
Monday, July 20, 2009
```

Similarly, for time values alone, there is an analagous T notation, to format output:

```
type "$(time(12:04:14), T5)";   // ANS: 12:04 PM
```

Formatting dates and times in this way uses one specific form of the more general $() Substitution notation.

Custom Formats

There are three custom date and time formats - two of which are script editable properties and one which is editable in the Column Properties dialog or using a worksheet column object method.

1. **system.date.customformat*n*$**
2. **wks.col.SetFormat** object method.

Both methods use date-time specifiers, such as **yyyy'.'MM'.'dd**, to designate the custom format. Please observe that:

- The text portions (non-space delimiters) of the date-time specifier can be changed as required, but must be surrounded by single quotes.
- The specifier tokens themselves (i.e., yyyy, HH, etc.) are case sensitive and need to be used exactly as shown— all possible specifier tokens can be found in the Reference Tables: Date and Time Format Specifiers.
- The first two formats store their descriptions in local file storage and as such may appear different in other login accounts. The third format stores its description in the column itself.

D*nn* notation

Origin has reserved **D19** to **D21** (subformats 20 to 22, since the integer after D starts its count from 0) for these custom date displays. The options D19 and D20 are controlled by system variables **system.date.customformat1$** and **system.date.customformat2$**, respectively. To use this option for output, follow the example below:

```
 system.date.customformat1$ = MMM dd hh'.'mm tt;
type "$(Date(7/25/09 14:47:21),D19)";      // Output: Jul 25 02.47 PM

system.date.customformat2$ = yy','MM','dd H'.'mm'.'ss'.'####;
type "$(Date(7/27/09 8:22:37.75234),D20)"; // Output: 09,07,27 8.22.37.7523
```

Wks.Col.SetFormat object method

To specify a custom date display for a date column which is stored in the worksheet column, use the Wks.Col.SetFormat object method. When entering the custom date format specifier, be sure to surround any non-date characters with single quotes. Also note that this object method works on columns of the active worksheet only.

In the following example, column 4 of the active worksheet is set to display a custom date/time format:

```
// wks.format=4 (date), wks.subformat=22 (custom)
wks.col4.SetFormat(4, 22, yyyy'-'MM'-'dd HH':'mm':'ss'.'###);
doc -uw; // Refresh the worksheet to show the change
```

9 Matrix Books Matrix Sheets and Matrix Objects

Similar to workbooks and worksheets, matrices in Origin also employ a data organizing hierarchy: Matrix Book -> Matrix Sheet -> Matrix Object. Therefore, objects like Page and Wks encompass matrix books and matrix sheets as well as workbooks and worksheets. In addition, Origin provides many X-Functions for handling matrix data.

9.1 Basic Matrix Book Operation

Matrix book has the same data structure level with workbook in Origin, both are windows. So, you can manipulate matrix books with the Page object and Window command, which is similar to workbook.

9.1.1 Workbook-like Operations

Both matrix book and workbook are windows, and they share lots of similar operations, even using the same LabTalk script. So, the differences will be pointed out below, and if the same script is used, please refer to Basic Workbook Operation.

1. Create New Matrix Book
   When using X-Function newbook to create new matrix book, the argument **mat** must be 1. Here is the similar example to the one for workbook.

```
//Create a new matrix book with the Long Name "MyMatrixBook"
newbook mat:=1 name:=MyMatrixBook;

// Create a new matrix book with 3 matrix sheets
// and use "Images" as Long Name and short name
newbook mat:=1 name:=Images sheet:=3 option:=lsname;

// Create a new hidden matrix book
// and the matrix book name is stored in myBkName$ variable
newbook mat:=1 hidden:=1 result:=myBkName$;
// Output matrix book name
myBkName$ = ;
```

2. Open Matrix Book
   Use the same command, doc -o, as opening workbook, to open matrix book. The difference is that the extension of a matrix book is *ogm*.
3. Save Matrix Book
   Origin's matrix book with data is with the extension of **ogm**, and template without data is **otm**. To save matrix book to ogm file and otm file, the save -i command and template_saveas X-Function will be used respectively, that is also the same with workbook. However, matrix book is not able to be saved as an analysis template.
4. Close Matrix Book
   This is the same as workbook, see commands win -ca and win -cd.
5. Show or Hide Matrix Book
   This is the same as workbook, see switches *-ch*, *-h*, and *-hc* in win command.
6. Name and Label Matrix Book
   This is the same as workbook, see win -r command, and page object.
7. Activate Matrix Book
   This is the same as workbook, see win -a command. The command window -o *winName* {*script*} can be used to run the specified script for the named matrix book. See the opening pages of the Running Scripts chapter for a more detailed explanation.
8. Delete Matrix Book
   This is the same as workbook, see win -c command.

9.1.2 Show Image Thumbnails

To show or hide image thumbnails, the command matrix -it is available.

```
// Create a new matrix book
newbook mat:=1;
// Import an image
string strImg$ = system.path.program$;
strImg$ += "Samples\Image Processing and Analysis\bamboo.jpg";
impImage fname:=strImg$;

// Hide image thumbnails
matrix -it 0;
```

9.2 Matrix Sheets

Matrix sheet has the same data structure level as Worksheet in Origin. So they have a lot of common properties.

9.2.1 Basic Matrix Sheet Operation

Examples in this section are similar to those found in the Basic Worksheet Operation section, because many object properties and X-Functions apply to both Worksheets and Matrix Sheets. Note, however, that not all properties of the **wks** object apply to a matrix sheet, and one should verify before using a property in production code.

Add New Matrix Sheet

The **newsheet** X-Function with the *mat:=1* option can be used to add new matrix sheets to matrix book.

```
// Create a new matrix book with 3 matrix sheets,
// and use "myMatrix" as long name and short name
newbook name:="myMatrix" sheet:=3 option:=lsname mat:=1;
// Add a 100*100 matrix sheet named "newMatrix" to current matrix book
newsheet name:=newMatrix cols:=100 rows:=100 mat:=1;
```

Activate a Matrix Sheet

Similar to worksheets, matrix sheets are also layers in a page, and *page.active* and *page.active$* properties can access matrix sheets. For example:

```
// Create a new matrix book with 3 matrix sheets
newbook sheet:=3 mat:=1;

page.active = 2;  // Activate a matrix sheet by layer number
page.active$ = MSheet3;    // Activate a matrix sheet by name
```

Modify Matrix Sheet Properties

To modify matrix properties, use the wks object, which works on matrix sheets as well as worksheets. For example:

```
// Rename the matrix sheet
wks.name$ = "New Matrix";
// Modify the column width
wks.colwidth = 8;
```

Set Dimensions

Both the wks object and the mdim X-Function can be used to set matrix dimensions:

```
// Use the wks object to set dimension
wks.ncols = 100;
wks.nrows = 200;
// Use the mdim X-Function to set dimension
mdim cols:=100 rows:=100;
```

For the case of multiple matrix objects contained in the same matrix sheet, note that all of the matrix objects must have the same dimensions.

Set XY Mapping

Matrices have numbered columns and rows which are mapped to linearly spaced X and Y values. In LabTalk, you can use the mdim X-Function to set the mapping.

```
// XY mapping of matrix sheet
mdim cols:=100 rows:=100 x1:=2 x2:=4 y1:=4 y2:=9;
```

Delete Matrix Sheet

Use the layer -d commands to delete matrix sheet. For example:

```
layer -d;  // delete the active layer, can be worksheet, matrix sheet or graph layer
layer -d 3;  // by index, delete third matrix sheet in active matrix book
layer -d msheet1;  // delete matrix sheet by name
range rs = [mbook1]msheet3!;
layer -d rs;  // delete matrix sheet by range
// the matrix book name stored in a string variable
string str$ = msheet2;
layer -d %(str$);
```

9.2.2 Matrix Sheet Data Manipulation

Conversion Between Matrix Sheets and Matrix Objects

In Origin, a matrix sheet can hold multiple matrix objects. Use the mo2s X-Function to split multiple matrix objects into separate matrix sheets.

Use the ms2o X-Function to combine multiple matrix sheets into one (provided all matrices share the same dimensions).

```
// Merge matrix sheet 2, 3, and 4
ms2o imp:=MBook1 sheets:="2,3,4" oms:=Merge;
// Split matrix objects in MSheet1 into new sheets
mo2s ims:=MSheet1 omp:=<new>;
```

9.3 Matrix Objects

Matrix object is the basic unit for storing matrix data, and its container is matrix sheet, that relationship is like column and worksheet. The following pages will show the practical examples on the operation of matrix object.

9.3.1 Basic Matrix Object Operation

A matrix sheet can have multiple matrix objects, which share the same dimensions. A matrix object is analogous to a worksheet column and can be added or deleted, etc. The following sections provide some practical examples on the basic operations of matrix object.

Add or Insert Matrix Object

It allows to set the number of matrix objects in the matrix sheet by using wks.nmats, so to add matrix objects. Also, the method wks.addcol() can be used to add a matrix object.

```
// Set the number of matrix objects in the matrix sheet to 5
wks.nmats = 5;
// Add a new matrix object to a matrix sheet
wks.addCol();
// Add a named matrix object to a matrix sheet
wks.addCol(Channel2);
```

By default, the 1st matrix object in matrix sheet is the current matrix object, you can use the wks.col property. And the method wks.insert() will insert matrix object before the current matrix object.

```
// Create a new matrix book, and show image thumbnails
newbook mat:=1;
matrix -it 1;

// Insert a matrix object before the 1st one in the active matrix sheet
wks.insert();

// Set the 2nd matrix object to be the current one
wks.col = 2;

// Insert a matrix object before the 2nd one
wks.insert();
```

Activate Matrix Object

To activate a matrix object in the active matrix sheet, the **wks.active** is available.

```
// Create a new matrix book
newbook mat:=1;

// Add two more matrix objects to the active matrix sheet
wks.addCol();
wks.addCol();

// Show image thumbnails
matrix -it 1;
```

```
// Activate the second matrix object
wks.active = 2;
```

Switch Between Image Mode and Data Mode

The matrix command has provided the option for switching between image mode and data mode of the matrix object. Only the active matrix object appears in the matrix sheet.

```
matrix -ii 1;  // Show image mode
matrix -ii 0;  // Show data mode
```

Set Labels

For each matrix object, you can set Long Name, Comments, and Units, by using Range Notation, which is a matrix object.

```
// Create a new matrix book
newbook mat:=1;

// Set number of matrix object of 1st matrix sheet to be 3
wks.nMats = 3;

// Show image thumbnails
matrix -it 1;

// Activate 1st matrix object
wks.active = 1;

// Set Long Name, Units, and Comments
range rx = 1;  // 1st matrix object of the active matrix sheet
rx.lname$ = X;  // Long Name = X
rx.unit$ = cm;  // Unit = cm
rx.comment$ = "X Direction";  // Comment = "X Direction"

// Do the same thing for matrix object 2 and 3
wks.active = 2;
range ry = 2;
ry.label$ = Y;  // Long Name can also be set in this way
ry.unit$ = cm;
ry.comment$ = "Y Direction";
wks.active = 3;
range rz = 3;
rz.label$ = Z;
rz.unit$ = Pa;
rz.comment$ = Pressure;
```

Delete Matrix Object

To delete a matrix object, you can use the delete command.

```
// Delete a matrix object by range
range rs=[mbook1]msheet1!1;  // The first matrix object
del rs;
// or delete a matrix object by name
range rs=[mbook1]msheet1!Channel2;  // The object named Channel2
del rs;
```

9.3.2 Matrix Object Data Manipulation

In addition to the matrix command, Origin provides X-Functions for performing specific operations on matrix object data. In this section we present examples of X-Functions that available used to work with matrix object data.

Set Values in Matrix Object

Matrix cell values can be set either using the matrix -v command or the msetvalue X-Function. The matrix -v command works only on an active matrix object, whereas the X-Function can set values in any matrix sheet.

This example shows how to set matrix values and then turn on display of image thumbnails in the matrix window.

```
// Create a matrix book
newbook mat:=1;
int nmats = 10;
range msheet=1!;
// Set the number of matrix objects
msheet.Nmats = nmats;
// Set value to the first matrix object
matrix -v x+y;
range mm=1; mm.label$="x+y";
double ff=0;
// Loop over other objects
loop(i, 2, nmats-1) {
   msheet.active = i;
   ff = (i-1)/(nmats-2);
   // Set values
   matrix -v (5/ff)*sin(x) + ff*20*cos(y);
   // Set LongName
   range aa=$(i);
   aa.label$="$(5/ff,*3)*sin(x) + $(ff*20)*cos(y)";
}
// Fill last one with random values
```

```
msheet.active = nmats;
matrix -v rnd();
range mm=$(nmats); mm.label$="random";
// Display thumbnail images in window
matrix -it;
```

Copy Matrix Data

The mcopy X-Function is used to copy matrix data.

```
// Copy data from mbook1 into another matrix, mbook2.
mcopy im:=mbook1 om:=mbook2; // This command auto-redimensions the target
```

Conversion between Matrix Object and Vector

Two X-Functions, m2v and v2m, are available for converting matrix data into a vector, and vector data into a matrix, respectively. Origin uses row-major ordering for storing a matrix, but both functions allow for column-major ordering to be specified as well.

```
// Copy the whole matrix, column by column, into a worksheet column
m2v method:=m2v direction:=col;
// Copy data from col(1) into specified matrix object
v2m ix:=col(1) method:=v2row om:=[Mbook1]1!1;
```

Conversion between Numeric Data and Image Data

In Origin, matrices can contain image data (i.e., RGB) or numeric data (i.e., integer). The following functions are available to convert between the two formats.

```
// Convert a grayscale image to a numeric data matrix
img2m img:=mat(1) om:=mat(2) type:=byte;
// Convert a numeric matrix to a grayscale image
m2img bits:=16;
```

Manipulate Matrix Object with Complex Values

X-Functions for manipulating a matrix with complex values include map2c, mc2ap, mri2c, and mc2ri. These X-Functions can merge two matrices (amplitude and phase, or real and imaginary) into one complex matrix, or split a complex matrix into amplitude/phase or real/imaginary components.

```
// Combine Amplitude and Phase into Complex
map2c am:=mat(1) pm:=mat(2) cm:=mat(3);
// Combine Real and imaginary in different matrices to complex in new matrix
mri2c rm:=[MBook1]MSheet1!mat(1) im:=[MBook2]MSheet1!mat(1) cm:=<new>;
// Convert complex numbers to two new matrix with amplitude and phase respectively
mc2ap cm:=mat(1) am:=<new> pm:=<new>;
// Convert complex numbers to two matrix objects with real part and imaginary part
mc2ri cm:=[MBook1]MSheet1!Complex rm:=[Split]Real im:=[Split]Imaginary;
```

Transform Matrix Object Data

Use the following X-Functions to physically alter the dimensions or contents of a matrix. In the transformations below, except the flipping matrix object, others may change the dimensions of its matrix sheet, which will make the change on other matrix objects in this matrix sheet.

Crop or extract from Data or Image Matrix

When a matrix contains an image in a matrix, the X-Function mcrop can be used to extract or crop to a rectangular region of the matrix.

```
// Crop an image matrix to 50 by 25 beginning from 10 pixels
// from the left and 20 pixels from the top.
mcrop x:=10 y:=20 w:=50 h:=25 im:=<active> om:=<input>; // <input> will crop
// Extract the central part of an image matrix to a new image matrix
// Matrix window must be active
matrix -pg DIM px py;
dx = nint(px/3);
dy = nint(py/3);
mcrop x:=dx y:=dy h:=dy w:=dx om:=<new>; // <new> will extract
```

Expand Data Matrix

The X-Function mexpand can expand a data matrix using specified column and row factors. Biquadratic interpolation is used to calculate the values for the new cells.

```
// Expand the active matrix with both factor of 2
mexpand cols:=2 rows:=2;
```

Flip Data or Image Matrix

The X-Function mflip can flip a matrix horizontally or vertically to produce its mirror matrix.

```
// Flip a matrix vertically
mflip flip:=vertical;

// Can also use the "matrix" command
matrix -c h;  // horizontally
matrix -c v;  // vertically
```

Rotate Data or Image Matrix

With the X-Function mrotate90, you can rotate a matrix 90/180 degrees clockwise or counterclockwise.

```
// Rotate the matrix 90 degrees clockwize
mrotate90 degree:=cw90;

// Can also use the "matrix" command to rotate matrix 90 degrees
matrix -c r;
```

Shrink Data Matrix

The X-Function mshrink can shrink a data matrix by specified row and column factors.

```
// Shrink the active matrix by column factor of 2, and row factor of 1
mshrink cols:=2 rows:=1;
```

Transpose Data Matrix

The X-Function mtranspose can be used to transpose a matrix.

```
// Transpose the second matrix object of [MBook1]MSheet1!
mtranspose im:=[MBook1]MSheet1!2;

// Can also use the "matrix" command to transpose a matrix
matrix -t;
```

Split RGB Image into Separate Channels

The imgRGBsplit X-Functions splits color images into separate R, G, B channels. For example:

```
// Split channels creating separate matrices for red, green and blue
imgRGBsplit img:=mat(1) r:=mat(2) g:=mat(3) b:=mat(4) colorize:=0;
// Split channels and apply red, green, blue palettes to the result matrices
imgRGBsplit img:=mat(1) r:=mat(2) g:=mat(3) b:=mat(4) colorize:=1;
```

Please see Image Processing X-Functions for further information on image handling.

9.3.3 Converting Matrix to Worksheet

You may need to re-organize your data by converting from matrix to worksheet, or vice versa, for certain analysis or graphing needs. This page provides information and examples of converting matrix to worksheet, and please refer to Converting Worksheet to Matrix for the "vice versa" case.

Matrix to Worksheet

Data in a matrix can also be converted to a worksheet by using the m2w X-Function. This X-Function can directly convert data into worksheet, with or without X/Y mapping, or convert data by rearranging the values into XYZ columns in the worksheet.

The following example shows how to convert matrix into worksheet, and plot graphs using different methods according the form of the worksheet data.

```
// Create a new matrix book
win -t matrix;
// Set matrix dimension and X/Y values
mdim cols:=21 rows:=21 x1:=0 x2:=10 y1:=0 y2:=100;
// Show matrix X/Y values
page.cntrl = 2;
// Set matrix Z values
```

```
msetvalue formula:="nlf_Gauss2D(x, y, 0, 1, 5, 2, 50, 20)";
// Hold the matrix window name
%P = %H;
// Covert matrix to worksheet by Dierct method
m2w ycol:=1 xlabel:=row1;
// Plot graph from worksheet using Virtual Matrix
plot_vm irng:=1! xy:=xacross ztitle:=MyGraph type:=242 ogl:=<new template:=cmap>;
// Convert matrix to XYZ worksheet data
sec -p 2;
win -a %P;
m2w im:=!1 method:=xyz;
// Plot a 3D Scatter
worksheet -s 3;
worksheet -p 240 3D;
```

If the matrix data is converted directly to worksheet cells, you can then plot such worksheet data using the Virtual Matrix feature.

10 Graphing

Origin's breadth and depth in graphing support capabilities are well known. The power and flexibility of Origin's graphing features are accessed as easily from script as from our graphical user interface. The following sections provide examples of creating and editing graphs from LabTalk scripts.

10.1 Creating Graphs

Creating graphs is probably the most commonly performed operation in Origin. This section gives examples of two X-Functions that allow you to create graphs directly from LabTalk scripts: **plotxy** and **plotgroup**. Once a plot is created, you can use object properties, like page, layer, axis objects, and set command to format the graph.

10.1.1 Creating a Graph with the PLOTXY X-Function

plotxy is an X-Function used for general purpose plotting. It is used to create a new graph window, plot into a graph template, or plot into a new graph layer. It has a syntax common to all X-Functions:

plotxy option1:=*optionValue* option2:=*optionValue* ... option*N*:=*optionValue*

All possible options and values are summarized in the X-Function help for **plotxy**. Since it is somewhat non-intuitive, the **plot** option and its most common values are summarized here:

plot:=	Plot Type
200	Line
201	Scatter
202	Line+symbol
203	column

All of the possible values for the **plot** option can be found in the Plot Type IDs.

Plotting X Y data

Input XYRange referencing the X and Y

The following example plots the first two columns of data in the active worksheet, where the first column will be plotted as X and the second column as Y, as a line plot.

```
plotxy iy:=(1,2) plot:=200;
```

Input XYRange referencing just the Y

The following example plots the second column of data in the active worksheet, as Y against its associated X, as a line plot. When you do not explicitly specify the X, Origin will use the the X-column that is associated with that Y-column in the worksheet, or if there is no associated X-column, then an <auto> X will be used. By default, <auto> X is row number.

```
plotxy iy:=2 plot:=200;
```

Plotting X YY data

The following example plots the first three columns of data from Book1, Sheet1, where the first column will be plotted as X and the second and third columns as Y, as a grouped scatter plot.

```
plotxy iy:=[Book1]Sheet1!(1,2:3) plot:=201;
```

Plotting XY XY data

The following example plots the first four columns of data in the active worksheet, where the first column will be plotted as X against the second column as Y and the third column as X against the fourth column as Y, as a grouped line+symbol plot.

```
plotxy iy:=((1,2),(3,4)) plot:=202;
```

Plotting using worksheet column designations

The following example plots all columns in the active worksheet, using the worksheet column plotting designations, as a column plot. '?' indicates to use the worksheet designations; '1:end' indicates to plot all the columns.

```
plotxy iy:=(?,1:end) plot:=203;
```

Plotting a subset of a column

The following example plots rows 1-12 of all columns in the active worksheet, as a grouped line plot.

```
plotxy iy:=(1,2:end)[1:12] plot:=200;
```

Plotting into a graph template

The following example plots the first column as theta(X) and the second column as r(Y) in the active worksheet, into the *polar* plot graph template, and the graph window is named *MyPolarGraph*.

```
plotxy (1,2) plot:=192 ogl:=[<new template:=polar name:=MyPolarGraph>];
```

Plotting into an existing graph layer

The following example plots columns 10-20 in the active worksheet, using column plotting designations, into the second layer of Graph1. These columns can all be Y columns and they will still plot against the associated X column in the worksheet.

```
plotxy iy:=(?,10:20) ogl:=[Graph1]2!;
```

Creating a new graph layer

The following example adds a new Bottom-X Left-Y layer to the active graph window, plotting the first column as X and the third column as Y from Book1, Sheet2, as a line plot. When a graph window is active and the output graph layer is not specified, a new layer is created.

```
plotxy iy:=[Book1]Sheet2!(1,3) plot:=200;
```

Creating a Double-Y Graph

```
// Import data file
string fpath$ = "Samples\Import and Export\S15-125-03.dat";
string fname$ = system.path.program$ + fpath$;
impASC;

// Remember Book and Sheet names
string bkname$ = page.name$;
string shname$ = layer.name$;

// Plot the first and second columns as X and Y
// The worksheet is active, so can just specify column range
plotxy iy:=(1,2) plot:=202 ogl:=[<new template:=doubleY>];

// Plot the first and third columns as X and Y into the second layer
// Now that the graph window is the active window, need to specify Book
//and Sheet
plotxy iy:=[bkname$]shname$!(1,3) plot:=202 ogl:=2;
```

10.1.2 Create Graph Groups with the PLOTGROUP X-Function

According to the grouping variables (datasets), **plotgroup** X-Function creates grouped plots for page, layer or dataplot. To work properly, the worksheet should be sorted by the graph group data first, then the layer group data and finally the dataplot group data.

This example shows how to plot by group.

```
// Establish a path to the sample data
fn$ = system.path.program$ + "Samples\Statistics\body.dat";
```

```
newbook;
impASC fn$;      // Import into new workbook

// Sort worksheet--Sorting is very important!
wsort bycol:=3;

// Plot by group
plotgroup iy:=(4,5) pgrp:=Col(3);
```

This next example creates graph windows based on one group and graph layers based on a second group:

```
// Bring in Sample data
fn$ = system.path.program$ + "Samples\Graphing\Categorical Data.dat";
newbook;
impASC fn$;
// Sort
dataset sortcol = {4,3}; // sort by drug, then gender
dataset sortord = {1,1}; // both ascending sort
wsort nest:=sortcol ord:=sortord;
// Plot each drug in a separate graph with gender separated by layer
plotgroup iy:=(2,1) pgrp:=col(drug) lgrp:=col(gender);
```

Note : Each group variable is optional. For example, you could use one group variable to organize data into layers by omitting Page Group and Data Group. The same sort order is important for whichever options you do use.

10.1.3 Create 3D Graphs with Worksheet -p Command

To create 3D Graphs, use the Worksheet (command) (-p switch).

First, create a simple 3D scatter plot:

```
// Create a new book
newbook r:=bkn$;
// Run script on bkn$
win -o bkn$ {
   // Import sample data
   string fname$ = system.path.program$ +
        "\samples\Matrix Conversion and Gridding" +
        "\XYZ Random Gaussian.dat";
   impasc;
   // Save new book name
   bkn$ = %H;
   // Change column type to Z
   wks.col3.type = 6;
   // Select column 3
   worksheet -s 3;
```

```
    // Plot a 3D scatter graph by template named "3d"
    worksheet -p 240 3d;
};
```

You can also create 3D color map or 3D mesh graph. 3D graphs can be plotted either from worksheet or matrix. And you may need to do **gridding** before plotting.

We can run the following script after above example and create a 3D wire frame plot from matrix:

```
win -o bkn$ {
    // Gridding by Shepard method
    xyz_shep 3;
    // Plot 3D wire frame graph;
    worksheet -p 242 wirefrm;
};
```

10.1.4 Create 3D Graph and Contour Graphs from Virtual Matrix

Origin can also create 3D graphs, such as 3D color map, contour, or 3D mesh, etc., from worksheet by the plotvm X-Function. This function creates a virtual matrix, and then plot from such matrix. For example:

```
// Create a new workbook and import sample data
newbook;
string fname$=system.path.program$ + "Samples\Graphing\VSurface 1.dat";
impasc;
// Treat entire sheet as a Virtual Matrix and create a colormap surface plot
plotvm irng:=1! format:=xacross rowpos:=selrow1 colpos:=selcol1
 ztitle:="VSurface 1" type:=242 ogl:=<new template:=cmap>;
// Change X axis scale to log
// Nonlinear axis type supported for 3D graphs created from virtual matrix
LAYER.X.type=2;
```

10.2 Formatting Graphs

10.2.1 Graph Window

A graph window is comprised of a visual page, with an associated **Page** (Object). Each graph page contains at least one visual layer, with an associated **layer** object. The graph layer contains a set of X Y axes with associated **layer.x** and **layer.y** objects, which are sub-objects of the **layer** object.

When you have a range variable mapped to a graph page or graph layer, you can use that variable name in place of the word **page** or **layer**.

10.2.2 Page Properties

The **page** object is used to access and modify properties of the active graph window. To output a list of all properties of this object:

```
page.=
```

The list will contain both numeric and text properties. When setting a text (string) property value, the **$** follows the property name.

To change the Short name of the active window:

```
page.name$="Graph3";
```

To change the Long name of the active window:

```
page.longname$="This name can contain spaces";
```

You can also change Graph properties or attributes using a range variable instead of the **page** object. The advantage is that using a range variable works whether or not the desired graph is active.

The example below sets the active graph layer to layer 2, using a range variable to point to the desired graph by name. Once declared, the range variable can be used in place of **page**:

```
//Create a Range variable that points to your graph
range rGraph = [Graph3];
//The range now has properties of the page object
rGraph.active=2;
```

10.2.3 Layer Properties

The **layer** object is used to access and modify properties of the graph layer.

To set the graph layer dimensions:

```
//Set the layer area units to cm
layer.unit=3;
//Set the Width
layer.width=5;
//Set the Height
layer.height=5;
```

Fill the Layer Background Color

The **laycolor** X-Function is used to fill the layer background color. The value you pass to the function for color, corresponds to Origin's color list as seen in the Plot Details dialog (1=black, 2=red, 3=green, etc).

To fill the background color of layer 1 as green:

```
laycolor layer:=1 color:=3;
```

Set Speed Mode Properties

The **speedmode** X-Function is used to set layer speed mode properties.

Update the Legend

The **legendupdate** X-Function is used to update or reconstruct the graph legend on the page/layer.

10.2.4 Axis Properties

The **layer.x** and **layer.y** sub-object of the **layer** object is used to modify properties of the axes.

To modify the X scale of the active layer:

```
//Set the scale to Log10
layer.x.type = 2;
//Set the start value
layer.x.from = .001;
//Set the end value
layer.x.to = 1000;
//Set the increment value
layer.x.inc = 2;
```

If you wish to work with the Y scale, then simply change the x in the above script to a y. If you wish to work with a layer that is not active, you can specify the layer index, layer*N*.x.from. Example: layer3.y.from = 0;

The **Axis** command can also be used to access the settings in the Axis dialog.

To change the X Axis Tick Labels to use the values from column C, given a plot of col(B) vs. col(A) with text in col(C), from Sheet1 of Book1:

```
range aa = [Book1]Sheet1!col(C);
axis -ps X T aa;
```

10.2.5 Data Plot Properties

The **Set** (Command) is used to change the attributes of a data plot. The following example shows how the **Set** command works by changing the properties of the same dataplot several times. In the script, we use sec command to pause one second before changing plot styles.

```
// Make up some data
newbook;
col(a) = {1:5};
col(b) = col(a);
// Create a scatter plot
plotxy col(b);

// Set symbol size
// %C is the active dataset
sec -p 1;
set %C -z 20;
// Set symbol shape
sec -p 1;
set %C -k 3;
// Set symbol color
sec -p 1;
set %C -c color(blue);
// Connect the symbols
sec -p 1;
set %C -l 1;
// Change plot line color
sec -p 1;
set %C -cl color(red);
// Set line width to 4 points
sec -p 1;
set %C -w 2000;
// Change solid line to dash
sec -p 1;
set %C -d 1;
```

Here is another example which plots into a template, *DoubleY*, with two layers, and then sets dataplot style for the dataplot in the second layer:

```
// Importing data
newbook;
string fn$=system.path.program$ + "Samples\Curve Fitting\Enzyme.dat";
impasc fname:=fn$;
//declare active worksheet range
range rr = !;
//plot into a template
plotxy iy:=(1,2) plot:=200 ogl:=[<new template:=DoubleY>];
```

```
//plot into second layer of active graph, which is graph created from line above
plotxy iy:=%(rr)(1,3) plot:=200 ogl:=2!;
//declare range for first dataplot in layer 2
range r2 = 2!1;
//set line to dash
set r2 -d 1;
```

10.2.6 Legend and Label

Formatting the Legend and Label are discussed on Creating and Accessing Graphical Objects.

10.3 Managing Layers

10.3.1 Creating a panel plot

The **newpanel** X-Function creates a new graph with an n x m layer arrangement.

Creating a 6 panel graph

The following example will create a new graph window with 6 layers, arranged as 2 columns and 3 rows. This function can be run independent of what window is active.

```
newpanel col:=2 row:=3;
```

Remember that when using X-Functions you do not always need to use the variable name when assigning values; however, being explicit with col:= and row:= may make your code more readable. To save yourself some typing, in place of the code above, you can use the following:

```
newpanel 2 3;
```

Creating and plotting into a 6 panel graph

The following example will import some data into a new workbook, create a new graph window with 6 layers, arranged as 2 columns and 3 rows, and loop through each layer (panel), plotting the imported data.

```
// Create a new workbook
newbook;

// Import a file
path$ = system.path.program$ + "Samples\Graphing\";
fname$ = path$ + "waterfall2.dat";
impasc;
```

```
// Save the workbook name as newpanel will change %H
string bkname$=%H;

// Create a 2*3 panel
newpanel 2 3;

// Plot the data
for (ii=2; ii<8; ii++)
{
   plotxy iy:=[bkname$]1!wcol(ii) plot:=200 ogl:=$(ii-1);
}
```

10.3.2 Adding Layers to a Graph Window

The **layadd** X-Function creates/adds a new layer to a graph window. This function is the equivalent of the **Graph:New Layer(Axes) menu**.

Programmatically adding a layer to a graph is not common. It is recommended to create a graph template ahead of time and then use the **plotxy** X-Function to plot into your graph template.

The following example will add an independent right Y axis scale. A new layer is added, displaying only the right Y axis. It is linked in dimension and the X axis is linked to the current active layer at the time the layer is added. The new added layer becomes the active layer.

```
layadd type:=rightY;
```

10.3.3 Arranging the layers

The **layarrange** X-Function is used to arrange the layers on the graph page.

Programmatically arranging layers on a graph is not common. It is recommended to create a graph template ahead of time and then use the **plotxy** X-Function to plot into your graph template.

The following example will arrange the existing layers on the active graph into two rows by three columns. If the active graph does not already have 6 layers, it will not add any new layers. It arranges only the layers that exist.

```
layarrange row:=2 col:=3;
```

10.3.4 Moving a layer

The **laysetpos** X-Function is used to set the position of one or more layers in the graph, relative to the page.

The following example will left align all layers in the active graph window, setting their position to be 15% from the left-hand side of the page.

```
laysetpos layer:="1:0" left:=15;
```

10.3.5 Swap two layers

The **layswap** X-Function is used to swap the location/position of two graph layers. You can reference the layers by name or number.

The following example will swap the position on the page of layers indexed 1 and 2.

```
layswap igl1:=1 igl2:=2;
```

The following example will swap the position on the page of layers named Layer1 and Layer2.

```
layswap igl1:=Layer1 igl2:=Layer2;
```

Layers can be renamed from both the Layer Management tool as well as the Plot Details dialog. In the Layer Management tool, you can double-click on the Name in the Layer Selection list, to rename. In the left-hand navigation panel of the Plot Details dialog, you can slow double-click a layer name to rename.

To rename from LabTalk, use layer*n*.name$ where *n* is the layer index. For example, to rename layer index 1 to Power, use the following:
layer1.name$="Power";

10.3.6 Aligning layers

The **layalign** X-Function is used to align one or more layers relative to a source/reference layer.

The following example will bottom align layer 2 with layer 1 in the active graph window.

```
layalign igl:=1 destlayer:=2 direction:=bottom;
```

The following example will left align layers 2, 3 and 4 with layer 1 in the active graph window.

```
layalign igl:=1 destlayer:=2:4 direction:=left;
```

The following example will left align all layers in Graph3 with respect to layer 1. The 2:0 notation means for all layers, starting with layer 2 and ending with the last layer in the graph.

```
layalign igp:=graph3 igl:=1 destlayer:=2:0 direction:=left;
```

10.3.7 Linking Layers

The **laylink** X-Function is used for linking layers to one another. It is used to link axes scales as well as layer area/position.

The following example will link all X axes in all layers in the active graph to the X axis of layer 1. The Units will be set to % of Linked Layer.

```
laylink igl:=1 destlayers:=2:0 XAxis:=1;
```

10.3.8 Setting Layer Unit

The **laysetunit** X-Function is used to set the unit for the layer area of one or more layers.

10.4 Creating and Accessing Graphical Objects

Graphical Objects could be many types, Line, Polyline, Rectangle, Cycle, Polygon, Arrow, Text, Image, etc. Once an object is created and attached to a layer, you can see it by invoking the list -o command option. The following section shows you how to create, change, and delete an object by LabTalk.

10.4.1 Creating Objects

Creating Labels

A label is one type of graphic object and can be created using the **Label** command. If no name is specified when creating labels by the **label -n** command, Origin will name the labels automatically with "Text*n*", where *n* is the creation index.

When creating labels, you can use escape sequences in a string to customize the text display. These sequences begin with the backslash character (\). Enter the following script to see how these escape sequences work. When there are spaces or multiple lines in your label text, quote the text with a double quote mark.

```
label "You can use \b(Bold Text)
Subscripts and Superscripts like X\=(\i(i), 2)
\i(Italic Text)
\ab(Text with Overbar)
or \c4(Color Text) in your Labels";
```

The following script creates a new text label on your active graph window with the value from column 1, row 5 of sheet1 in book3. It works for both string and numeric.

```
label -s %([book3]Sheet1,1,5);
```

The following script creates a new text label on your active graph window from the value in row 1 of column 2 of sheet2 in book1. Note the difference from the above example - the cell(i,j) function takes row number as first argument. It works for a numeric cell only.

```
label -s $([book1]Sheet2!cell(1,2));
```

Besides, you can address worksheet cell values as your label contents. The following script creates a new text label on your active graph window from the value in row 1 of column 2 of sheet2 in book1. The value is displayed with 4 significant digits.

```
label -s $([book1]Sheet2!cell(1,2), *4);
```

The %( ) notation does not allow formatting and displays the value with full precision. You need to use $( ) notation if you wish to format the numeric value.

Creating Legends

A graph legend is just a text label with the object name **Legend**. It has properties common to all graphical objects. To output a list of all properties of the legend, simply enter the following:

```
legend.=
```

To view the object name of any graphical object right-click on it and select **Programming Control** from the context menu.

To update or reconstruct the graph legend, use the **legendupdate** X-function, which has the following syntax:

legendupdate [mode:=*optionName*]

The square brackets indicate that **mode** is optional, such that **legendupdate** may be used on its own, as in:

```
legendupdate;
```

which will use the default legend setting (short name) or use mode to specify what you would like displayed:

```
legendupdate mode:=0;
```

which will display the **Comment** field in the regenerated legend for the column of data plotted. All possible modes can be found in Help: X-Functions: legendupdate:

Note that either the index or the name of the mode may be used in the X-function call, such that the script lines,

```
legendupdate mode:=comment;
legendupdate mode:=0;
```

are equivalent and produce the same result.

The **custom** legend option requires an additional argument, demonstrated here:

```
legendupdate mode:=custom custom:=@WS;
```

All available custom legend options are given in the Text Label Options.

The following example shows how to use these functions and commands to update legends.

```
// Import sample data;
newbook;
string fn$ = system.path.program$ +
   "Samples\Curve Fitting\Enzyme.dat";
impasc fname:=fn$;
string bn$ = %H;
// Create a two panels graph
newpanel 1 2;
// Add dataplot to layers
for (ii=1; ii<=2; ii++)
{
   plotxy iy:=[bn$]1!wcol(ii+1) plot:=201 ogl:=$(ii);
}
// Upate whole page legends by worksheet comment + unit
legendupdate dest:=0 update:=0 mode:=custom custom:=@ln;
// Modify the legend settings for each layers
doc -e LW {
   // Set legend font size
   legend.fsize = 28;
   // Set legend font color
   legend.color = color(blue);
   // Move legend to upper-left of the layer
   legend.x = layer.x.from + legend.dx / 2;
   legend.y = layer.y.to - legend.dy / 2;
};
```

Note: To modify the text of the legend, you can also use the **label** command. One reason to use this would be if you wanted to display more than one text entry for each dataplot. The script below will update the legend text to display both the worksheet name and the X column's Comment:

```
label -sl -n legend "\l(1) %(1, @WS) %(1X, @LC)";
```

Creating Lines

Objects like lines, rectangles, are graphic objects, and you can use **draw** command to create them.

In the example below, you can see how to use the **-l** and **-v** switches to draw a **V**ertical **L**ine. The line will be drawn at the midpoint of the X axis, where X1 and X2 are system variables that store the X From and X To scale values respectively.

```
draw -l -v (X1+(X2-X1)/2);
```

To make the line movable, use the **-lm** switch.

```
draw -lm -v (X1+(X2-X1)/2);
```

10.4.2 Working on Objects

Position of Objects

Object position can either be controlled when creating it, or changed by object properties. The following table lists how these properties and commmands works:

Property / Command	Unit	Reference Point
label -p	Percentage	Top-left
label -px	Pixel of Screen	Top-left
object.top / object.left	Pixel of Page	Top-left
object.x / object.y	Layer coordinates	Center of Object
object.x1 / object.y1	Layer coordinates	Top-left

Notes: The pixel of a page can be found from the *Print/Dimensions* tab of Plot Details dialog.

For example:

```
win -T Plot; // Create an empty graph
// Create a text object at the layer center,
//  named as "MyText", and the context is "Hello World"
label -p 50 50 -n MyText Hello World;
sec -p 1;
// Place the label at (1, 5)
MyText.x1 = 1;
MyText.y1 = 5;
```

Change Object Properties

All graphical objects can use ***objectName*.=** to get or set object properties. Take label as example, the **object.x** and **object.y** properties specify the x and y position of the center of an object, and **object.dx** and **object.dy** specify the object width and height. These four properties are all using axis units, so we can combine these four properties with **layer.axis.from** and **layer.axis.to** to place the label in the proper position on a layer.

The following script example shows how to use label properties to place labels.

```
// Import sample data
newbook;
string fname$ = system.path.program$ +
```

```
   "Samples\Curve Fitting\Enzyme.dat";
impasc;
string bn$ = %H;
plotxy ((,2), (,3));
// Create a label and name it "title"
// Be note the sequence of option list, -n should be the last option
// -j is used to center the text
// -s enables the substitution notation
// -sa enables conversion of \n (new line)
// Subsitution is used to get text from column comments
label -j 1 -s -sa -n title
  Enzyme Reaction Velocity\n%([bn$]1!col(2)[c]$) vs. %([bn$]1!col(3)[c]$);
// Set font
title.font=font(Times New Roman);
// Set label font size
title.fsize = 28;
// Set label font color
title.color = color(blue);
// Placing label
title.x = layer.x.from + (layer.x.to - layer.x.from) / 2;
title.y = layer.y.to + title.dy / 2;
// Placing legend
legend.y = layer.y.from + (layer.y.to - layer.y.from) / 2;
legend.x = layer.x.to - legend.dx / 2;
```

10.4.3 Deleting an Object

To delete objects, use the label command with -r, -ra, and -rc switches:

Switch	Description
label -r *objectName*	Delete the specified object
label -ra *objectNamePrefix*	Delete all objects whose names start with *objectNamePrefix*
label -rc *objectName*	Remove specified object, with the connected objects

11 Importing

Origin provides a collection of X-Functions for importing data from various file formats such as ASCII, CSV, Excel, National Instruments DIAdem, pCLAMP, and many others. The X-Function for each file format provides options relevant to that format in addition to common settings such as assigning the name of the import file to the book or sheet name.

All X-Functions pertaining to importing have names that start with the letters **imp**. The table below provides a listing of these X-Functions. As with all X-Functions, help-file information is available at Script or Command line by entering the name of the X-Function with the **-h** option. For instance: entering **impasc -h** in the Script window will display the help file immediately below the command.

Name	Brief Description
impASC	Import ASCII file/files
impBin2d	Import binary 2d array file
impCSV	Import csv file
impDT	Import Data Translation Version 1.0 files
impEP	Import EarthProbe (EPA) file. Now only EPA file is supported for EarthProbe data.
impExcel	Import Microsoft Excel 97-2007 files
impFamos	Import Famos Version 2 files
impFile	Import file with pre-defined filter.
impHEKA	Import HEKA (dat) files
impIgorPro	Import WaveMetrics IgorPro (pxp, ibw) files
impImage	Import a graphics file
impinfo	Read information related to import files.
impJCAMP	Import JCAMP-DX Version 6 files

impJNB	Import SigmaPlot (JNB) file. It supports version lower than SigmaPlot 8.0.
impKG	Import KaleidaGraph file
impMatlab	Import Matlab files
impMDF	Import ETAS INCA MDF (DAT, MDF) files. It supports INCA 5.4 (file version 3.0).
impMNTB	Import Minitab file (MTW) or project (MPJ). It supports the version prior to Minitab 13.
impNetCDF	Import netCDF file. It supports the file version lower than 3.1.
impNIDIAdem	Import National Instruments DIAdem 10.0 dat files
impNITDM	Import National Instruments TDM and TDMS files(TDMS does not support data/time format)
impODQ	Import *.ODQ files.
imppClamp	Import pCLAMP file. It supports pClamp 9 (ABF 1.8 file format) and pClamp 10 (ABF 2.0 file format).
impSIE	Import nCode Somat SIE 0.92 file
impSPC	Import Thermo File
impSPE	Import Princeton Instruments (SPE) file. It supports the version prior to 2.5.
impWav	Import waveform audio file
reimport	Re-import current file

You can write your own import routines in the form of X-Functions as well. If the name of a user-created X-Function begins with **imp** and it is placed in the **\X-Functions\Import and Export** subfolder of the EXE, UFF or Group paths, then such functions will appear in the **File|Import** menu.

The following sections give examples of script usage of these functions for importing data, graphs, and images.

11.1 Importing Data

The following examples demonstrate the use of X-Functions for importing data from external files. The examples import ASCII files, but the appropriate X-Function can be substituted based on your desired filetype (i.e., CSV, Matlab); syntax and supporting commands will be the same. Since these examples import Origin sample files, they can be typed or pasted directly into the Script or Command window and run.

11.1.1 Import an ASCII Data File Into a Worksheet or Matrix

This example imports an ASCII file (in this case having a ***.txt** extension) into the active worksheet or matrix. Another X-Function, **findfiles**, is used to find a specific file in a directory (assigned to the string **path$**) that contains many other files. The output of the findfiles X-Function is a string variable containing the desired filename(s), and is assigned, by default, to a variable named **fname$**. Not coincidentally, the default input argument for the **impASC** X-Function is a string variable called **fname$**.

```
string path$ = system.path.program$ + "Samples\Import and Export\";
findfiles ext:=matrix_data_with_xy.txt;
impASC;
```

11.1.2 Import ASCII Data with Options Specified

This example makes use of many advanced options of the **impASC** X-Function. It imports a file to a new book, which will be renamed by the options of the **impASC** X-Function. Notice that there is only one semi-colon (following all **options** assignments) indicating that all are part of the call to **impASC**.

```
string fn$=system.path.program$ + "Samples\Spectroscopy\HiddenPeaks.dat";
impasc fname:=fn$
options.ImpMode:=3                        /* start with a new book */
options.Sparklines:=0                     /* turn off sparklines */
options.Names.AutoNames:=0                /* turn off auto rename */
options.Names.FNameToSht:=1               /* rename sheet to file name */
options.Miscellaneous.LeadingZeros:=1;    /* remove leading zeros */
```

11.1.3 Import Multiple Data Files

This example demonstrates importing multiple data files to a new workbook; starting a new worksheet for each file.

```
string fns, path$=system.path.program$ + "Samples\Curve Fitting\";
findfiles f:=fns$ e:="step1*.dat";   // find matching files in 'path$'
int n = fns.GetNumTokens(CRLF);      // Number of files found
string bkName$;
```

```
newbook s:=0 result:=bkName$;
impasc fname:=fns$                  // impasc has many options
options.ImpMode:=4                      // start with new sheet
options.Sparklines:=2                   // add sparklines if < 50 cols
options.Cols.NumCols:=3         // only import first three columns
options.Names.AutoNames:=0      // turn off auto rename
options.Names.FNameToBk:=0          // do not rename the workbook
options.Names.FNameToSht:=1         // rename sheet to file name
options.Names.FNameToShtFrom:=4     // trim file name after 4th letter
options.Names.FNameToBkComm:=1      // add file name to workbook comment
options.Names.FNameToColComm:=1     // add file name to columns comments
options.Names.FPathToComm:=1        // include file path to comments
orng:=[bkName$]A1!A[1]:C[0] ;
```

11.1.4 Import an ASCII File to Worksheet and Convert to Matrix

This example shows two more helpful X-Functions working in conjunction with **impASC**; they are **dlgFile**, which generates a dialog for choosing a specific file to import, and **w2m** which specifies the conversion of a worksheet to a matrix. It should be noted that the **w2m** X-Function expects linearly increasing Y values in the first column and linearly increasing X values in the first row: test this with **matrix_data_with_xy.txt** in the **Samples\Import and Export** folder.

```
dlgfile g:=ascii; // Open file dialog
impAsc; // Import selected file
// Use the worksheet-to-matrix X-Function, 'w2m', to do the conversion
w2m xy:=0 ycol:=1 xlabel:="First Row" xcol:=1
```

11.1.5 Related: the Open Command

Another way to bring data into Origin is with the **Open** (Command).

Open has several options, one of which allows a file to be open for viewing in a notes window:

open -n fileName [winName]

This line of script opens the ASCII file *fileName* to a **notes** window. If the optional *winName* is not specified, a new **notes** window will be created.

To demonstrate with an existing file, try the following:

```
%b = system.path.program$ + "Samples\Import and Export\ASCII simple.dat";
open -n "%b";
```

11.1.6 Import with Themes and Filters

Import with a Theme

When importing from the Origin GUI, you can save your import settings to a **theme file**. Such theme files have a *.OIS extension and are saved in the **\Themes\AnalysisAndReportTable** subfolder of the Origin **User Files Folder** (UFF). They can then be accessed using an X-Function with the **-t** option switch. The import is performed according to the settings saved in the theme file specified.

```
string fn$=system.path.program$ + "Samples\Spectroscopy\HiddenPeaks.dat";
// Assume that a theme file named "My Theme.OIS" exists
impasc fname:=fn$ -t "My Theme";
```

Import with an Import Wizard Filter File

Custom importing of ASCII files and simple binary files can be performed using the **Import Wizard** GUI tool. This tool allows extraction of variables from file name and header, and further customization of the import including running a script segment at the end of the import, which can be used to perform post-processing of imported data. All settings in the GUI can be saved as an **Import Filter File** to disk. Such files have extension of **.OIF** and can be saved in multiple locations.

Once an **import wizard filter file** has been created, the **impfile** X-Function can be used to access the filter and perform custom importing using the settings saved in the filter file.

```
string fname$, path$, filtername$;
// point to file path
path$ = system.path.program$ + "Samples\Import and Export\";
// find files that match specification
findfiles ext:="S*.dat";
// point to Import Wizard filter file
string str$ = "Samples\Import and Export\VarsFromFileNameAndHeader.oif";
filtername$ = system.path.program$ + str$;
// import all files using filter in data folder
impfile location:=data;
```

11.1.7 Import from a Database

Origin provides four functions for Database Queries. The basic functionality of Database importing is encapsulated in two functions as shown in this example using the standard Northwind database provided by Microsoft Office:

```
// The dbedit function allows you to create the query and connection
// strings and attach these details to a worksheet
dbedit exec:=0
sql:="Select Customers.CompanyName, Orders.OrderDate,
[Order Details].Quantity, Products.ProductName From
```

```
((Customers Inner Join Orders On Customers.CustomerID = Orders.CustomerID)
Inner Join [Order Details] On Orders.OrderID = [Order Details].OrderID)
Inner Join Products On Products.ProductID = [Order Details].ProductID"
connect:="Provider=Microsoft.Jet.OLEDB.4.0;User ID=;
Data Source=C:\Program Files\Microsoft Office\OFFICE11\SAMPLES\Northwind.mdb;
Mode=Share Deny None;Extended Properties="";
Jet OLEDB:System database="";
Jet OLEDB:Registry Path="";
Jet OLEDB:Database Password=***;
Jet OLEDB:Engine Type=5;
Jet OLEDB:Database Locking Mode=1;
Jet OLEDB:Global Partial Bulk Ops=2;
Jet OLEDB:Global Bulk Transactions=1;
Jet OLEDB:New Database Password="";
Jet OLEDB:Create System Database=False;
Jet OLEDB:Encrypt Database=False;
Jet OLEDB:Don't Copy Locale on Compact=False;
Jet OLEDB:Compact Without Replica Repair=False;
Jet OLEDB:SFP=False;Password="

// The dbimport function is all that's needed to complete the import
dbimport;
```

Two additional functions allow you to retrieve the details of your connection and query strings and execute a Preview/Partial import.

Name	Brief Description
dbEdit	Create, Edit, Load or Remove a query in a worksheet.
dbImport	Execute the database queried stored in a specific worksheet.
dbInfo	Read the sql string and the connection string contained in a database query in a worksheet.
dbPreview	Execute a limited import (defaults to 50 rows) of a query. Useful in testing to verify that your query is returning the information you want.

11.2 Importing Images

The ImpImage X-Function supports importing image files into Origin from script. By default, the image is stored in Origin as an image (i.e., RGB values). You have the option to convert the image to grayscale.

Multiple-file importing is supported. By default, multiple images will be appended to the target page by creating new layers. If importing to a matrix, each matrix-layer will be renamed to the corresponding imported file's name.

11.2.1 Import Image to Matrix and Convert to Data

This example imports a single image file to a matrix and then converts the (RGB color) image to grayscale values, storing them in a new matrix.

```
newbook mat:=1;              // Create a new matrix book
fpath$ = "Samples\Image Processing and Analysis\car.bmp";
string fname$ = system.path.program$ + fpath$;

// Imports the image on path 'fname$' to the active window
//(the new matrix book)
impimage;

// Converts the image to grayscale values,and puts them in a new matrix
// 'type' specifies bit-depth:  0=short (2-byte/16-bit, default);
// 1=byte (1-byte/8-bit)
img2m type:=byte;
```

11.2.2 Import Single Image to Matrix

This example imports a series of *.TIF images into a new Matrix Book. As an alternative to the **img2m** X-Function (shown above), the keyboard shortcuts **Ctrl+Shift+d** and **Ctrl+Shift+i** toggle between the matrix data and image representations of the file.

```
newbook mat:=1;
fpath$ = "Samples\Image Processing and Analysis\";
string fns, path$ = system.path.program$ + fpath$;
// Find the files whose names begin with 'myocyte'
findfiles f:=fns$ e:="myocyte*.tif";
// Import each file into a new sheet (options.Mode = 4)
impimage options.Mode:=4 fname:=fns$;
```

11.2.3 Import Multiple Images to Matrix Book

This example imports a folder of JPG images to different Matrix books.

```
string pth1$ = "C:\Documents and Settings\All Users\"
string pth2$ = "Documents\My Pictures\Sample Pictures\";
string fns, path$ = pth1$ + pth2$;
// Find all *.JPG files (in 'path$', by default)
findfiles f:=fns$ e:="*.jpg";
// Assign the number of files found to integer variable 'n'
// 'CRLF' ==> files separated by a 'carriage-return line-feed'
int n = fns.GetNumTokens(CRLF);
string bkName$;
string fname$;
// Loop through all files, importing each to a new matrix book
for(int ii = 1; ii<=n; ii++)
{
    fname$ = fns.GetToken(ii, CRLF)$;

    //create a new matrix page
    newbook s:=0 mat:=1 result:=bkName$;
    //import image to the first layer of the matrix page,
    //defaut file name is fname$
    impimage orng:=[bkName$]msheet1;
}
```

11.2.4 Import Image to Graph Layer

You also can import an Image to an existing GraphLayer. Here the image is only for display (the data will not be visible, unless it is converted to a matrix, see next example).

```
string fpath$ = "Samples\Image Processing and Analysis\cell.jpg";
string fn$ = system.path.program$ + fpath$;
impimage fname:=fn$ ipg:=graph1;
```

12 Exporting

Origin provides a collection of X-Functions for exporting data, graphs, and images. All X-Functions pertaining to exporting have names that start with the letters **exp**. The table below provides a listing of these X-Functions. As with all X-Functions, help-file information is available at Script or Command line by entering the name of the X-Function with the **-h** option. For instance: entering **expgraph -h** in the Script window will display the help file immediately below the command.

Name	Brief Description
expASC	Export worksheet data as ASCII file
expGraph	Export graph(s) to graphics file(s)
expImage	Export the active Image into a graphics file
expMatASC	Export matrix data as ASCII file
expNITDM	Export workbook data as National Instruments TDM and TDMS files
expWAV	Export data as Microsoft PCM wave file
expWks	Export the active sheet as raster or vector image file
img2GIF	Export the active Image into a gif file

12.1 Exporting Worksheets

12.1.1 Export a Worksheet

Your worksheet data may be exported either as an image (i.e., PDF) or as a data file.

Export a Worksheet as an Image File

The **expWks** X-Function can be used to export the entire worksheet, the visible area of the worksheet, or worksheet selection, to an image file such as JPEG, EPS, or PDF:

```
// Export the active worksheet to an EPS file named TEST.EPS,
```

```
// saved to the D:\ drive.
expWks type:=EPS export:=active filename:="TEST" path:="D:";
```

The **expWks** X-Function also provides options for exporting many worksheets at the same time using the **export** option, which if unspecified simply exports the active worksheet.

In the following example, *export:=book* exports all worksheets in the current workbook to the desired folder *path*:

```
expWks type:=PDF export:=book path:="D:\TestImages" filename:=Sheet#;
```

Worksheets are saved in the order they appear in the workbook from left to right. Here, the naming has been set to number the sheets in that order, as in 'Sheet1', 'Sheet2', etc. If more than 9 sheets exist, *filename:=Sheet##* will yield names such as 'Sheet01'.

Other options for *export* are *project*, *recursive*, *folder*, and *specified*.

The **expWks** X-Function is particularly useful in exporting custom report worksheets that user may create by placing graphs and other relevant analysis results in a single sheet for presentation, using formatting features such as merging and coloring cells.

Export a Worksheet as a Multipage PDF File

The **expPDFw** X-Function allows exporting worksheets to multi-page PDF files. This X-Function is then useful to export large worksheets, including custom report sheets, where the worksheet has more content than can fit in one page for the current printer settings. This X-Function offers options such as printing all sheets in a book or all sheets in the project, and options for including a cover page and adding page numbering.

Export a Worksheet as a Data File

In this example, worksheet data is output to an ASCII file with tabs separating the columns using the **expAsc** X-Function:

```
// Export the data in Book 2, Worksheet 3 using tab-separators to
// an ASCII file named TEST.DAT, saved to the D:\ drive.

expASC iw:=[Book2]Sheet3 type:=0 path:="D:\TEST.DAT" separator:=TAB;
```

Note, in this example, that *type* simply indicates the type of file extension, and may be set to any of the following values (type:=dat is equivalent to type:=0):

- 0=dat:*.dat,
- 1=text:Text File(*.txt),
- 2=csv:*.csv,
- 3=all:All Files(*.*)

12.2 Exporting Graphs

Here are three examples of exporting graphs using the X-Function **expGraph** called from LabTalk:

12.2.1 Export a Graph with Specific Width and Resolution (DPI)

Export a graph as an image using the **expGraph** X-Function. The image size options are stored in the nodes of tree variable named **tr1**, while resolution options (for all raster type images) are stored in a tree named **tr2**.

One common application is to export a graph to a desired image format specifying *both* the width of the image and the resolution. For example, consider a journal that requires, for a two-column article, that graphs be sent as high-resolution (1200 DPI), *.tif files that are 3.2 inches wide:

```
// Export the active graph window to D:\TestImages\TEST.TIF.
// Width = 3.2 in, Resolution = 1200 DPI

expGraph type:=tif path:="D:\TestImages" filename:="TEST"
    tr1.unit:=0
    tr1.width:=3.2
    tr2.tif.dotsperinch:=1200;
```

Possible values for tr1.unit are:

- 0 = inch
- 1 = cm
- 2 = pixel
- 3 = page ratio

Note: this is a good example of accessing data stored in a tree structure to specify a particular type of output. The full documentation for **tr1** can be found in the online and product (CHM) help.

12.2.2 Exporting All Graphs in the Project

Exporting all of the graphs from an Origin Project can be achieved by combining the **doc -e** command, which loops over all specified objects in a project with the **expGraph** X-Function.

For example, to export all graphs in the current project as a bitmap (BMP) image, as above:

```
doc -e P
{
  // %H is a string register that holds the name of the active window.
  expGraph type:=bmp path:="d:\TestImages\" filename:=%H
```

```
            tr1.unit:=2
            tr1.width:=640;
}
```

Several examples of **doc -e** can be found in Looping Over Objects.

12.2.3 Exporting Graph with Path and File Name

The string registers, %G and %X, hold the current project file name and path. Combine with the label command, you can place these information on page while exporting a graph. For example:

```
// Path of the project
string proPath$ = system.path.program$ + "Samples\Graphing\Multi-Curve Graphs.opj";
// Open the project
doc -o %(proPath$);
// Add file path and name to graph
win -a Graph1;
label -s -px 0 0 -n ForPrintOnly \v(%X%G.opj);
// Export graph to disk D
expGraph type:=png filename:=%H path:=D:\;
// Delete the file path and name
label -r ForPrintOnly;
```

12.3 Exporting Matrices

Matrices can store image data as well as non-image data in Origin. In fact, *all* images in Origin are stored as matrices, whether or not they are rendered as a picture or displayed as pixel values. A matrix can be exported no matter which type of content it holds.

Exporting matrices with script is achieved with two X-Functions: **expMatAsc** for a non-image matrix and **expImage** for an image matrix.

12.3.1 Exporting a Non-Image Matrix

To export a matrix that holds non-image data to an ASCII file use the **expMatAsc** X-Function. Allowed export extenstions are *.dat (type:=0), *.txt (type:=1), *.csv (type:=2), and all file types (type:=3).

```
// Export a matrix (in Matrix Book 1, Matrix Sheet 1) to a file of
// the *.csv type named TEST.CSV with xy-gridding turned on.

expMatASC im:=[MBook1]MSheet1 type:=2 path:="D:\TEST.CSV" xygrid:=1;
```

12.3.2 Exporting an Image Matrix

Matrix windows in Origin can contain multiple sheets, and each sheet can contain multiple matrix objects. A matrix object can contain an image as RGB values (default, reported as three numbers in a single matrix cell, each matrix cell corresponds to a pixel), or as gray-scale data (a single gray-scale number in each matrix cell).

For example, a user could import an image into a matrix object (as RGB values) and later convert it to gray-scale data (i.e., the gray-scale pixel values) using the **Image** menu. Whether the matrix object contains RGB or gray-scale data, the contents of the matrix can be exported as an image file to disk, using the **expImage** X-Function. For example, the following script command exports the first matrix object in Sheet 1 of matrix book MBook 1:

```
// Export the image matrix as a *.tif image:
expImage im:=[MBook1]1!1 type:=tif fname:="c:\flower"
```

When exporting to a raster-type image format (includes JPEG, GIF, PNG, TIF), one may want to specify the bit-depth as well as the resolution (in dots-per-inch, DPI). This is achieved with the **expImage** options tree, **tr**. The X-Function call specifying these options might look like this:

```
expImage im:=[MBook1]MSheet1! type:=png fname:="D:\TEST.PNG"
tr.PNG.bitsperpixel:="24-bit Color"
tr.PNG.dotsperinch:=300;
```

All nodes of the tree **tr**, are described in the online or product (CHM) help.

12.4 Exporting Videos

To export group of graphs as a video, you need to use the Video Writer (vw) object. In order to export a video with actual frames, you always need to create a video writer object, write some window into it as frames and then release the video writer. You can only work with one video writer at one time, i.e. each time after you create a video writer object with the **vw.Create( )** method, you must use **vw.Release( )** to release the video writer before you can use the **vw.Create( )** method again.

Here are several example scripts showing how to create, write graphs in or release a video writer object. You can also view a full example in the LabTalk Examples category.

12.4.1 Create a Video Writer Object

To export a video, the first thing is to create a video writer object with the **vw.Create( )** method. You need to at least specify the file name(including complete file path) of the video, and you can also specify the codec value for compression method, frames per second and video dimensions at this stage.

For example, this script create a video file named "test.avi" in an existing file path *D:\Exported Videos*\ with other settings as default (i.e. no compression, 1 frame per second, 640 px as width and 480 px as height):

```
vw.Create("D:\Exported Videos\test.avi");
//The above script is the same as the script below
//vw.Create("D:\Exported Videos\test.avi", 0, 1, 640, 480);
```

You can also define the compression method with FourCC code, for example the script below use WMV1 compressed format to create the video:

```
//Define codec with four character code
int codec = vw.FourCC('W', 'M', 'V', '1');
//Create a 800*600 video file as test.avi under user files folder
vw.Create(%Y\My Video.avi, codec, 1, 800, 600)
```

Sometimes you need to check whether the video creation is successful, the **vw.Create( )** method returns 0 if the video is successfully created, and returns a non-zero value if the creation fails. For example, the following script helps you to decide whether the video is indeed created.

```
//If file path D:\AAA exist,the following should return 0
//If the file path does NOT exist, it will return error code
int err = vw.Create(D:\AAA\test.avi);
if(err==0)
   type "video creation is successful";
else
   type "video creation failure, the error code is $(err)";
```

12.4.2 Write Graph(s) in a Video Writer Object

Once a video writer object is created, you can start to write graphs into it with the **vw.WriteGraph( )** method. In fact, not only graph window is supported to be written by this, but also other windows like function plot, workbook, matrix, layout.

For example, this script writes the current active window to the video.

```
vw.WriteGraph( );
```

You can specify the window name and also the number of frames to write, for example, the following script will add Graph1 as 5 frames:

```
vw.WriteGraph(Graph1,5);
```

And you can make use of a loop structure to, for example, add all graphs in a video, so that you do not need to write multiple lines of script. The example below writes all graph windows in the project into the video, each graph will be inserted as 2 frames.

```
doc -e P
{
   vw.WriteGraph(,2);
}
```

You can also get the error code from this method similarly as the last example of the create video writer session. And if the return value is 0, it means that write graph(or other windows) is successful.

12.4.3 Release a Video Writer Object

For each video writer object, it is essential to release the video writer so as to actually generate the video, the method used in this case is **vw.Release( )**.

The following scripts shows a complete example of generating a video file "example.avi" in user files folder with a newly created empty graph window.

```
int err = vw.Create(%Y\example.avi);
//Write existing graphs into the video if the video can be created.
if(0 == err)
{
    //Create an empty graph window with default template
    win -t plot;
    vw.WriteGraph( );
}
//Release the video writer
vw.Release( );
```

The **vw.Release( )** method similarly has a return value, if it is 0 then the video generation is successful, but if it is 1, it indicates the video generation fails.

13 The Origin Project

13.1 Managing the Project

13.1.1 The DOCUMENT Command

Document is a native LabTalk command that lets you perform various operations related to the Origin Project. The syntax for the **document** command is

```
document -option value;
```

Notes:

- *value* is not applicable for some options and is left out of the command
- For further details please see Document (Object).

Internally, Origin updates a property that indicates when a project has been modified. Attempting to Open a project when the current project has been modified normally triggers a prompt to Save the current project. The document command has options to control this property.

Start a New Project

```
// WARNING! This will turn off the Save project prompt
document -s;
// ''doc'' is short for ''document'' and ''n'' is short for ''new''
doc -n;
```

Open/Save a project

Use the doc -o command to open a project and the save command to save it.

```
// Open an Origin Project file
string fname$ = SYSTEM.PATH.PROGRAM$ + "Origin.opj";
doc -o %(fname$);   // Abbreviation of ''document -open''
// Make some changes
%(Data1,1) = data(0,100);
%(Data1,2) = 100 * uniform(101);
// Save the project with a new name in new location
fname$ = SYSTEM.PATH.APPDATA$ + "My Project.opj";
save %(fname$);
```

Append projects

Continuing with the previous script, we can Append other project file(s). Origin supports only one project file at a time.

```
// Append an Origin Project file to the current file
fname$ = SYSTEM.PATH.PROGRAM$ + "Origin.opj";
doc -a %(fname$); // Abbreviation of ''document -append''
// Save the current project - which is still ''My Project.opj''
save;
// Save the current project with a new name to a new location
save "C:\Data Files\working.opj";
```

Save/Load Child Windows

In Origin, a child window - such as a graph, workbook, matrix or Excel book - can be saved as a single file. Append can be used to add the file to another project. The appropriate extension is added automatically for Workbook, Matrix and Graph whereas you must specify .XLS for Excel windows.

```
// The save command acts on the active window
save -i C:\Data\MyBook;
```

Append can be used to load Child Window Types :

```
// Workbook(*.OGW), Matrix(*.OGM), Graph(*.OGG), Excel(*.XLS)
dlgfile group:=*.ogg;
// fname is the string variable set by the dlgfile X-Function
doc -a %(fname$);
```

For Excel, you can specify that an Excel file should be imported rather than opened as Excel

```
doc -ai "C:\Data\Excel\Current Data.xls";
```

Notes windows are a special case with special option switch:

```
// Save notes window named Notes1
save -n Notes1 C:\Data\Notes\Today.TXT;
// Read text file into notes window named MyNotes
open -n C:\Data\Notes\Today.txt MyNotes;
```

Saving External Excel Book

This is introduced in Origin 8.1, to allow an externally linked Excel book to be saved using its current file name:

```
save -i;
```

Refresh Windows

You can refresh windows with the following command:

```
doc -u;
```

13.1.2 Project Explorer X-Functions

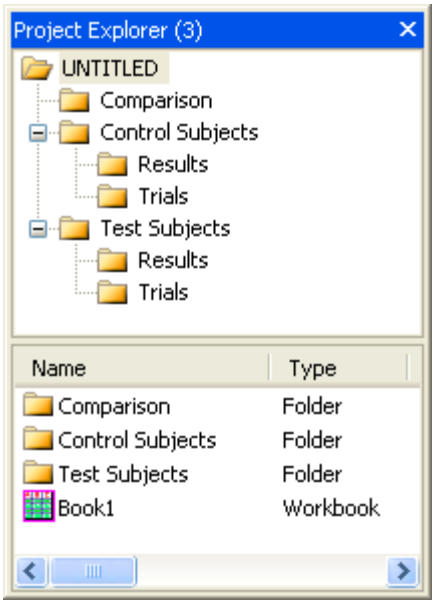
The following X-Functions provide DOS-like commands to create, delete and navigate through the subfolders of the project:

Name	Brief Description
pe_dir	Show the contents of the active folder
pe_cd	Change to another folder
pe_move	Move a Folder or Window
pe_path	Report the current path
pe_rename	Rename a Folder or Window
pe_mkdir	Create a Folder
pe_rmdir	Delete a Folder

In this example :

```
doc -s;                      // Clear Origin's 'dirty' flag
doc -n;                      // Start a new project
pe_cd /;                     // Go to the top level
pe_mkdir "Test Subjects";    // Create a folder
pe_cd "Test Subjects";       // Navigate to that folder
pe_mkdir "Trials";           // Create a sub-folder
pe_mkdir "Results";          //  and another
pe_cd /;                     // Return to the top level
pe_mkdir "Control Subjects"; // Create another folder
pe_cd "Control Subjects";    // Navigate to that folder
pe_mkdir "Trials";           // Create a sub-folder
pe_mkdir "Results";          //  and another
pe_cd /;                     // Return to the top level
pe_mkdir "Comparison";       // Create a folder
```

we create a folder structure that looks like this in Project explorer :



Note that if you have **Open in Subfolder** enabled in Tools : Options : [Open/Close] then you will have an additional folder named **Folder1**.

13.2 Accessing Metadata

Metadata is information which refers to other data. Examples include the time at which data was originally collected, the operator of the instrument collecting the data and the temperature of a sample being investigated. Metadata can be stored in Projects, Pages, Layers and Columns.

13.2.1 Column Label Rows

Metadata is most visible in a worksheet where column headers may contain information such as Long Name (L), Units (U), Comments(C), Sampling Interval and various Parameter rows, including User-Defined parameters.

The row indices for column label rows are assigned to characters, which are given in the Column Label Row reference table. Examples of use follow.

Read/Write Column Label Rows

At times you may want to capture or set the Column Label Rows or Column Header string from script. Access the label row by using the corresponding label row characters as a row index.

Note: Numeric cell access does not supported to use label row characters.

Here are a few examples of reading and writing column header strings:

```
Book1_A[L]$ = Time;   // Set the Long Name of column A to '''Time'''
Book1_A[U]$ = sec;    // Set the Units of column A to '''sec'''
string strC$ = col(2)[C]$;   // Read the Comments of column2 into strC$
// Get value from first system parameter row
double syspar1 = %(col(2)[p1]$);
Col(1)[L]$="Temperature";   // set Col(1) long name to "Temperature"
range bb = 2;               // declare a range variable for Col(2)
// Set the long name of Col(2) to that of Col(1) with the string " Data"
// appended
bb[L]$=Col(1)[L]$+" Data";
```

Note: For Origin 8.0, LabTalk variables took precedence over Column Label Row characters, for example:

```
int L = 4;    // For Origin 8.0 and earlier ...
Col(B)[L]$=   // Returns the value in row 4 of Col(B), as a string
```

But for Origin 8.1, this has been changed so that the column label rows (L, U, C, etc.) will take precedence:

```
int L = 4;    // For Origin 8.1 ...
Col(B)[L]$=   // Returns the Long Name of Col(B), as a string
```

The following example shows how to create and access user parameter rows

```
// Show the first user parameter row
wks.userParam1 = 1;
// Assign the 1st user parameter row a custom name
wks.userParam1$ = "Temperature";
// Write to a specific user parameter row of a column
col(2)[Temperature]$ = "96.8";
// Get a user-defined parameter row value
double temp = %(col(2)[Temperature]$);
```

Show/Hide Column Labels

You can set which column header rows are displayed and in what order by wks.labels object method. For the active worksheet, this script specifies the following column header rows (in the order given): Long Name, Unit, the first System-Parameter, the First User-Parameter, and Comments:

```
range ww = !;
ww.labels(LUP1D1C);
```

13.2.2 Even Sampling Interval

Origin users can set the sampling interval (X) for a data series (Y) to something other than the corresponding row numbers of the data points (default).

Accessing the Sampling Interval Column Label Row

When this is done, a special header row is created to remind the user of the custom interval (and initial value) applied. To access the text in this header row, simply use the **E** row-index character. This header is effectively read-only and cannot be set to an arbitrary string, but the properties from which this string is composed may be changed with either column properties (see the wks.col object) or the colint X-Function.

```
// Read the Sampling Interval header text of Column 1 to a string variable
string sampInt$ = Col(1)[E]$;
// If an initial value of 2 and increment of 0.5 was set for Col(1),
// the output will be:
sampInt$=;                          // "x0 = 2"
                                    // "dx = 0.5"
```

To see a Sampling Interval header, you can try the following steps:

1. Create a new worksheet and delete the X-column
2. Right-click at the top of the remaining column (i.e., B(Y)), such that then entire column is selected, and select **Set Sampling Interval** from the drop-down menu.
3. Set the initial and step values to something other than 1.
4. Click OK, and you will see a new header row created which lists the values you specified.

The next example demonstrates how to do this from script, using X-functions.

Also, when you import certain types of data, e.g. **.wav*, the sampling interval will show as a header row.

Sampling Interval by X-Function

Sampling Interval is special in that its display is formatted for the user's information. Programmatically, it is accessed as follows

```
// Use full formal notation of an X-Function
colint rng:=col(1) x0:=68 inc:=.25 units:=Degrees lname:="Temperature";
// which in shorthand notation is
colint 1 68 .25 Degrees "Temperature";
```

The initial value and increment can be read back using worksheet column properties:

```
double XInitial = wks.col1.xinit;
double XIncrement = wks.col1.xinc;
string XUnits$ = wks.col1.xunits$;
string XName$ = wks.col1.xname$;
```

Note: While these properties will show up in a listing of column properties (Enter **wks.col1.=** in the Script window to display the property names for column 1), unless a sampling interval is established:

- The strings **wks.col1.xunits$** and **wks.col1.xname$** will have no value.
- The numeric values **wks.col1.xinit** and **wks.col1.xinc** will each have a value of 1, corresponding to the initial value and increment of the row numbers.

13.2.3 Trees

Trees are a data type supported by LabTalk, and we also consider trees a form of metadata since they give structure to existing data. They were briefly introduced in the section dealing with Data Types and Variables, but appear again because of their importance to X-functions.

Many X-functions input and output data in tree form. And since X-functions are one of the primary tools accessible from LabTalk script, it is important to recognize and use tree variables effectively.

Access Import File Tree Nodes

After importing data into a worksheet, Origin stores metadata in a special tree-like structure at the page level. Basic information about the file can be retrieved directly from this structure:

```
string strName, strPath;
double dDate;
// Get the file name, path and date from the structure
strName$ = page.info.system.import.filename$;
strPath$ = page.info.system.import.filepath$;
dDate = page.info.system.import.filedate;
// Both % and $ substitution methods are used
ty File %(strPath$)%(strName$), dated $(dDate,D10);
```

This tree structure includes a tree with additional information about the import. This tree can be extracted as a tree variable using an X-Function:

```
Tree MyFiles;
impinfo ipg:=[Book2] tr:=MyFiles;
MyFiles.=; // Dump the contents of the tree to the script Window
```

Note: The contents of the *impinfo* tree will depend on the function used to import.

If you import multiple files into one workbook (using either New Sheets, New Columns or New Rows) then you need to load a particular tree for each file as the Organizer only displays the system metadata from the last import:

```
Tree trFile;
int iNumFiles;
// Use the function first to find the number of files
impinfo trInfo:=trFile fcount:=iNumFiles;
// Now loop through all files - these are indexed from 0
for( idx = 0 ; idx < iNumFiles ; idx++ )
{
   // Get the tree for the next file
```

```
   impinfo findex:=idx trInfo:=trFile;
   string strFileName, strLocation;
   //
   strFileName$ = trFile.Info.FileName$;
   strLocation$ = trFile.Info.DataRange$;
   ty File %(strFileName$) was imported into %(strLocation$);
}
```

Access Report Page Tree

Analysis Report pages are specially formatted Worksheets based on a tree structure. You can get this structure into a tree variable using the getresults X-Function and extract results.

```
// Import an Origin Sample file
string fpath$ = "Samples\Curve Fitting\Gaussian.dat";
string fname$ = SYSTEM.PATH.PROGRAM$ + fpath$;
impasc;
// Run a Gauss fit of the data and create a Report sheet
nlbegin (1,2) gauss;
nlfit;
nlend 1 1;
// An automatically-created string variable, __REPORT$,
// holds the name of the last Report sheet created:
string strLastReport$ = __REPORT$;
// This is the X-Function which gets the Report into a tree
getresults tr:=MyResults iw:=%(strLastReport$);
// So now we can access those results
ty Variable\tValue\tError;
separator 3;
ty y0\t$(MyResults.Parameters.y0.Value)\t$(MyResults.Parameters.y0.Error);
ty xc\t$(MyResults.Parameters.xc.Value)\t$(MyResults.Parameters.xc.Error);
ty w\t$(MyResults.Parameters.w.Value)\t$(MyResults.Parameters.w.Error);
ty A\t$(MyResults.Parameters.A.Value)\t$(MyResults.Parameters.A.Error);
```

User Tree in Page Storage

Information can be stored in a workbook, matrix book or graph page using a tree structure. The following example shows how to create a section and add subsections and values to the active page storage area.

```
// Add a new section named Experiment
page.tree.add(Experiment);
// Add a sub section called Sample;
page.tree.experiment.addsection(Sample);
// Add values to subsection;
page.tree.experiment.sample.RunNumber = 45;
page.tree.experiment.sample.Temperature = 273.8;
```

```
// Add another subsection called Detector;
page.tree.experiment.addsection(Detector);
// Add values;
page.tree.experiment.detector.Type$ = "InGaAs";
page.tree.experiment.detector.Cooling$ = "Liquid Nitrogen";
```

Once the information has been stored, it can be retrieved by simply dumping the storage contents:

```
// Dump entire contents of page storage
page.tree.=;
// or programmaticaly accessed
temperature = page.tree.experiment.sample.temperature;
string type$ = page.tree.experiment.detector.Type$;
ty Using %(type$) at $(temperature)K;
```

You can view such trees in the page Organizer for Workbooks and Matrixbooks.

User Tree in a Worksheet

Trees stored at the Page level in a Workbook can be accessed no matter what Sheet is active. You can also store trees at the sheet level:

```
// Here we add two trees to the active sheet
wks.tree.add(Input);
// Dynamically create a branch and value
wks.tree.input.Min = 0;
// Add another value
wks.tree.input.max = 1;
// Add second tree
wks.tree.add(Output);
// and two more values
wks.tree.output.min = -100;
wks.tree.output.max = 100;

// Now dump the trees
wks.tree.=;
// or access it
ty Input $(wks.tree.input.min) to $(wks.tree.input.max);
ty Output $(wks.tree.output.min) to $(wks.tree.output.max);

// Access a sheet-level tree using a range
range rs = [Book7]Sheet2!;
rs!wks.tree.=;
```

You can view such trees in the page Organizer for Workbooks and Matrixbooks.

User Tree in a Worksheet Column

Individual worksheet columns can also contain metadata stored in tree format. Assigning and retrieving tree nodes is very similar to the page-level tree.

```
// Create a COLUMN tree
wks.col2.tree.add(Batch);
// Add a branch
wks.col2.tree.batch.addsection(Mix);
// and two values in the branch
wks.col2.tree.batch.mix.ratio$ = "20:15:2";
wks.col2.tree.batch.mix.BatchNo= 113210;
// Add branch dynamically and add values
wks.col2.tree.batch.Line.No = 7;
wks.col2.tree.batch.Line.Date$ = 3/15/2010;

// Dump the tree to the Script Window
wks.col2.tree.=;
// Or access the tree
batch = wks.col2.tree.batch.mix.batchno;
string strDate$ = wks.col2.tree.batch.Line.Date$;
ty Batch $(batch) made on %(strDate$) [$(date(%(strDate$)))];
```

You can view these trees in the Column Properties dialog on the User Tree tab.

13.3 Looping Over Objects

There may be instances where it is desirable to perform a certain task or set of tasks on every object of a particular type that exists in the Origin project. For example, you might want to rescale all of your project graph layers or add a new column to every worksheet in the project. The LabTalk document command (or **doc**) facilitates this type of operation. Several examples are shown here to illustrate the **doc** command.

13.3.1 Looping over Objects in a Project

The document command with the -e or -ef switch (or doc -e command), is the primary means for looping over various collections of objects in an Origin Project. This command allows user to execute multiple lines of LabTalk script on each instance of the Origin Object found in the collection.

Looping over Workbooks and Worksheets

You can loop through all worksheets in a project with the doc -e LB command. The script below loops through all worksheets, skipping the matrix layers:

```
//loop over all worksheets in project to print their names
//and the number of columns on each sheet
doc -e LB {
   if(exist(%H,2)==0) //not a workbook, must be a matrix
        continue;
   int nn = wks.nCols;
   string str=wks.Name$;
   type "[%H]%(str$) has $(nn) columns";
}
```

The following example shows how to loop and operate on data columns that reside in different workbooks of a project.

Open the sample project file available since Origin 8.1 SR2:
\\Samples\LabTalk Script Examples\Loop_wks.opj

In the project there are two folders for two different samples and a folder named **Bgsignal** for the background signals alone. Each sample folder contains two folders named **Freq1** and **Freq2**, which correspond to data at a set frequency for the specific sample.

The workbook in each **Freq** folder contains three columns including **DataX**, **DataY** and the frequency, which is a constant. The workbook's name in the **Bgsignal** folder is **Bgsig**. In the **Bgsig** workbook, there are three columns including **DataX** and two Y columns whose long names correspond to set frequencies in the workbook in each **Freq** folder.

The aim is to add a column in each workbook and subtract the background signal for a particular frequency from the sample data for the same frequency. The following Labtalk script performs this operation.

```
doc -e LB
{ //Loop over each worksheet.
   if(%H != "Bgsig") //Skip the background signal workbook.
   {
      Freq=col(3)[1]; //Get the frequency.
      wks.ncols=wks.ncols+1; //Add a column in the sample sheet.
      //bg signal column for Freq using long name.
      range aa=[Bgsig]1!col("$(Freq)");
      wcol(wks.ncols)=col(2)-aa; //Subtract the bg signal.
      wcol(wks.ncols)[L]$="Remove bg signal"; //Set the long name.
   }
}
```

For increased control, you may also loop through the books and then loop through the sheets in your code, albeit a bit more slowly than the code above.

The following example shows how to loop over all workbooks in the current/active Project Explorer Folder, and then loop over each sheet inside each book that is found:

```
int nbooks = 0;
// Get the name of this folder
string strPath;
pe_path path:=strPath;
// Loop over all Workbooks ...
```

```
// Restricted to the current Project Explorer Folder View
doc -ef W {
   int nsheets = 0;
   // Loop over all worksheets in each workbook
   doc -e LW {
      type Sheet name: %(layer.name$);
      nsheets++;
   }
   type Found $(nsheets) sheet(s) in %H;
   type %(CRLF);
   nbooks++;
}
type Found $(nbooks) book(s) in folder %(strPath$) of project %G;
```

Additionally, we can replace the internal loop using Workbook properties:

```
int nbooks = 0;
// Get the name of this folder
string strPath;
pe_path path:=strPath;
// Loop over all Workbooks ...
// Restricted to the current Project Explorer Folder View
doc -ef W {
   // Loop over all worksheets in each workbook
   loop(ii,1,page.nlayers) {
      range rW = [Book1]$(ii)!;
      type Sheet name: %(rw.name$);
   }
   type Found $(page.nlayers) sheet(s) in %H;
   type %(CRLF);
   nbooks++;
}
// Final report - %G contains the project name
type Found $(nbooks) book(s) in folder %(strPath$) of project %G;
```

Looping Over Graph Windows

Here we loop over all plot windows (which include all Graph, Function Plots, Layout pages and embedded Graphs).

```
doc -e LP
{
    // Skip over any embedded graphs or Layout windows
    if(page.IsEmbedded==0&&exist(%H)!=11)
    {
        string name$ = %(page.label$);
        if(name.Getlength()==0 ) name$ = %H;
```

```
        type [%(name$)]%(layer.name$);
    }
}
```

The following script prints the contents of all graph windows in the project to the default printer driver.

```
doc -e P print; // Abbreviation of ''document -each Plot Print''
```

Looping Over Workbook Windows

The document -e command can be nested as in this example that loops over all Y datasets within all Worksheets:

```
doc -e W
{
    int iCount = 0;
    doc -e DY
    {
        iCount++;
    }
    if( iCount < 2 )
      { type Worksheet %H has $(wks.ncols) columns,;
        type $(iCount) of which are Y columns; }
    else
      { type Worksheet %H has $(wks.ncols) columns,;
        type $(iCount) of which are Y columns; }
}
```

Looping over Columns and Rows

This example shows how to loop over all columns and delete every nth column

```
int ndel = 3; // change this number as needed;
int ncols = wks.ncols;
int nlast = ncols - mod(ncols, ndel);
// Need to delete from the right to the left
for(int ii = nlast; ii > 0; ii -= ndel)
{
   delete wcol($(ii));
}
```

This example shows how to delete every nth rows in a worksheet.

```
int ndel = 3;           // change this number as needed
range rr = col(1);      // Get a range for column 1
nrows = rr.GetSize();   // Get the number of rows
int nlast = nrows - mod(nrows, ndel);
// Need to delete from the bottom to the top
for(int ii = nlast; ii > 0; ii -= ndel)
```

```
{
   range rr = wcol(1)[$(ii):$(ii)];
   mark -d rr;
}
```

This script calculates the logarithm of four columns on Sheet1, placing the result in the corresponding column of Sheet2:

```
for(ii=1; ii<=4; ii++)
{
   range ss = [book1]sheet1!col($(ii));
   range dd = [book1]sheet2!col($(ii));
   dd = log(ss);
}
```

Looping Over Graphic Objects

You can loop over all Graphic Objects in the active layer. By wrapping this with two other options we can cover an entire project.

```
// For each Plot
doc -e P
{
    // For each Layer in each Plot
    doc -e LW
    {
        // For each Graphic Object in each Layer in each Plot
        doc -e G
        {   // Set Legend background to Shadow
            if("%B"=="Legend") %B.background = 2;
            // Set timestamp color to Blue
            if("%B"=="timestamp") %B.color = color(blue);
        }
    }
}
```

13.3.2 Perform Peak Analysis on All Layers in Graph

This example shows how to loop over all layers in a graph and perform peak analysis on datasets in each layer using a pre-saved Peak Analyzer theme file. It assumes the active window is a multi-layer graph, and each layer has one data curve. It further assumes a pre-saved Peak Analyzer theme exists.

```
// Block reminder messages before entering loop.
// This is to avoid either reminder message from popping up
// about Origin switching to the report sheet
type -mb 0;
// Loop over all layers in graph window
```

```
doc -e LW
{
   // Perform peak analysis with preset theme
   sec;
   pa theme:="My Peak Fit";
   watch;
   /* sec and watch are optional,
      they print out time taken for fitting data in each layer */
}
// Un-block reminder message
type -me;
```

14 Analysis and Applications

Origin supports functions that are valuable to certain types of data analysis and specific mathematic and scientific applications. The following sections provide examples on how to use some of these functions, broken down by categories of use.

14.1 Mathematics

In this section we feature examples of four common mathematical tasks in data processing:

14.1.1 Average Multiple Curves

Multiple curves (XY data pairs) can be averaged to create a single curve, using the **avecurves** X-Function. This X-Function provides several options such as using the input X values for the output curve, or generating uniformly spaced X values for the output and then interpolating the input Y data before averaging.

The following example demonstrates averaging with linear interpolation:

```
// Load sample data using existing 'loadDSC.ogs' script
string fpath$ = "Samples\LabTalk Script Examples\LoadDSC.ogs";
string LoadPath$=system.path.program$ + fpath$;
// If the data does not load properly, then stop script execution.
if(!run.section(%(LoadPath$), Main, 0)) break 1;
// Data should now be loaded now into the active workbook ...
// Get the name of the active workbook, %H points to the active workbook
string dscBook$=%H;

// Perform average on all data using linear interpolation
avecurves iy:=[dscBook$](1:end)!(1,2)
          rd:=[<input>]<new name:="Averaged Data">!
          method:=ave
          interp:=linear;
```

Once averaged, the data and the result can be plotted:

```
// plot all the data and the averaged curve, using the plotxy X-Function:
plotxy [dscBook$](1:end)!(1,2) plot:=200;
```

14.1.2 Differentiation

Finding the Derivative

The following example shows how to calculate the derivative of a dataset. Note that the **differentiate** X-Function is used, and that it allows higer-order derivatives as well:

```
// Import the data
newbook;
fname$ = system.path.program$ + "\Samples\Spectroscopy\HiddenPeaks.dat";
impasc;

// Calculate the 1st and 2nd derivatives of the data in Column 2:

// Output defaults to the next available column, Column 3
differentiate iy:=Col(2);
// Output goes into Column 4 by default
differentiate iy:=Col(2) order:=2;

// Plot the source data and the results

// Each plot uses Column 1 as its x-values
plotstack iy:=((1,2), (1,3), (1,4)) order:=top;
```

Finding the Derivative with Smoothing

The **differentiate** X-Function also allows you to obtain the derivatives using Savitsky-Golay smoothing. If you want to use this capability, set the **smooth** variable to 1. Then you can customize the smoothing by specifying the polynomial order and the points of window used in the Savitzky-Golay smoothing method. The example below illustrates this.

```
// Import a sample data with noise
newbook;
fpath$ = "\Samples\Signal Processing\fftfilter1.DAT";
fname$ = system.path.program$ + fpath$;
impasc;
bkname$=%h;

// Differentiate using Savitsky-Golay smoothing
differentiate iy:=col(2) smooth:=1 poly:=1 npts:=30;

// Plot the source data and the result
newpanel row:=2;
plotxy iy:=[bkname$]1!2 plot:=200 ogl:=1;
plotxy iy:=[bkname$]1!3 plot:=200 ogl:=2;
```

14.1.3 Integration

The **integ1** X-Function is capable of finding the area under a curve using integration. Both mathematical and absolute areas can be computed. In the following example, the absolute area is calculated:

```
//Import a sample data
newbook;
fname$ = system.path.program$ + "Samples\Mathematics\Sine Curve.dat";
impasc;

//Calculate the absolute area of the curve and plot the integral curve
integ1 iy:=col(2) type:=abs plot:=1;
```

Once the integration is performed, the results can be obtained from the integ1 tree variable:

```
// Dump the integ1 tree
integ1.=;
// Get a specific value
double area = integ1.area;
```

The X-Function also allows specifying variable names for quantities of interest, such as:

```
double myarea, ymax, xmax;
integ1 iy:=col(2) type:=abs plot:=1 area:=myarea y0:=ymax x0:=xmax;
type "area=$(myarea) %(CRLF)ymax=$(ymax) %(CRLF)xmax=$(xmax)";
```

Integration of two-dimensional data in a matrix can also be performed using the **integ2** X-Function. This X-Function computes the volume beneath the surface defined by the matrix, with respect to the z=0 plane.

```
// Perform volume integration of 1st matrix object in first matrix sheet
range rmat=[MBook1]1!1;
integ2 im:=rmat integral:=myresult;
type "Volume integration result: $(myresult)";
```

14.1.4 Interpolation

Interpolation is one of the more common mathematical functions performed on data, and Origin supports interpolation in two ways: (1) interpolation of single values and datasets through range notation and (2) interpolation of entire curves by X-Functions.

Using XY Range

An XY Range once declared can be used as a function. The argument to this function can be a scalar - which returns a scalar - or a vector - which returns a vector. In either case, the X dataset should be increasing or decreasing. For example:

```
newbook;
wks.ncols = 4;
col(1) = data(1,0,-.05);
```

```
col(2) = gauss(col(1),0,.5,.2,100);
range rxy = (1,2);
rxy(.67)=;
range newx = 3; // Use column as X column data
newx = {0, 0.3333, 0.6667, 1.0}; // Create our new X data
range newy = 4; // This is the empty column we will interpolate into
newy = rxy(newx);
```

You can then use such range variables as a function with the following form:
XYRangeVariable(RangeVariableOrScalar[,*connect*[,*param*]])

where *connect* is one of the following options:

line
straight line connection

spline
spline connection

bspline
b-spline connection

and *param* is smoothing parameter, which applies only to bspline connection method. If *param*=-1, then a simple bspline is used, which will give same result as bspline line connection type in line plots. If 'param *>=0, the NAG* ***nag_1d_spline_function*** *is used.*

> **Notes:** When using XY range interpolation, you should guarantee there are no duplicated ***x*** values if you specify ***spline*** or ***bspline*** as the connection method. Instead, you can use interpolation X-Functions.

From Worksheet Data

The following examples show how to perform interpolation using range as function, with data from a worksheet as the argument.

Example1: The following code illustrates the usage of the various smoothing parameters for **bspline**:

```
col(1)=data(1,9);              // Fill column 1 with row data
col(2)=normal(9);              // Fill column 2 with random values
col(3)=data(1,9,0.01);         // Fill Col(3) with desired X values
wks.col3.type = 4;
range bb=(1,2);                // Declare range using cols 1,2;
// Compute interpolated values using different parameter settings
loop(i, 4, 10) {
   wcol(i)=bb(col(3), bspline, $(i*0.1));
}
```

Example2: With an XY range, new Y values can be obtained at any X value using code such as:

```
// Generate some data
newbook;
```

```
wcol(1)={1, 2, 3, 4};
wcol(2)={2, 3, 5, 6};
// Define XYrange
range rr =(1,2);
// Find Y value by linear interpolation at a specified X value.
rr(1.23) = ;    // ANS: rr(1.23)=2.23
// Find Y value by linear interpolation for an array of X values.
wcol(3)={1.5, 2.5, 3.5};
range rNewX = col(3);
// Add new column to hold the calculated Y values
wks.addcol();
wcol(4) = rr(rNewX);
```

Example3: To find X values given Y values, simply reverse the arguments in the examples above. In the case of finding X given Y, the Y dataset should be increasing or decreasing.

```
// Generate some data
newbook;
wcol(1)={1, 2, 3, 4};
wcol(2)={2, 3, 5, 6};
// Define XYrange
range rr =(2,1); //swapping the X and Y
// Find X value by linear interpolation at a specified Y value.
rr(2.23) = ;    // ANS: rr(2.23)=1.23;
// Add new column to hold the calculated X values
wks.addcol();
range rNewX = wcol(3);
// Find X value by linear interpolation for an array of Y values:
wcol(4)={2.5, 3.5, 5.5};
range rNewY = wcol(4);
rNewX = rr(rNewY);
```

From Graph

You can also use range interpolation when a graph page is active.

Example 1: Interpolate an array of values.

```
// Define range on active plot:
range rg = %C;
// Interpolate for a scalar value using the line connection style:
rg(3.54)=;
// Interpolate for an array of values:
// Give the location of the new X values:
range newX = [Book2]1!1;
// Give the location where the new Y values (output) should go:
range newY = [Book2]1!2;
// Compute the new Y values:
newY = rg(newX);
```

Example 2: Specify the interpolation method.

```
// Define range on specific plot:
range -wx rWx = 2;     // Use X of 2nd plot in active layer
range -w rWy = 2;      // Use Y of 2nd plot in active layer
range rr = (rWx,rWy);  // Construct an XY range from two ranges
// Give the location where the new X values (output) should go:
range newX = [Book2]1!1;
newX = {5,15,25};
range newY1 = [Book2]1!2; // Range for new Y
range newY2 = [Book2]1!3; // Range for new Y
// Find new Y values by linear interpolation for an array of X values:
newY1 = rr(newX);
// Find new Y values by bspline interpolation for an array of X values:
newY2 = rr(newX,bspline);
```

Using Arbitrary Dataset

For two arbitrary datasets with the same length, where both are increasing or decreasing, Origin allows you to interpolate from one dataset to the other at a given value. The datasets can be a range variable, dataset variable, or column. The form to perform such interpolation is:

dataset1(value, dataset2)

which will perform interpolation on the group of XY data constructed by **dataset2** and **dataset1**, and it will return the so-called Y (**dataset1**) value at the given so-called X (**dataset2**) **value**. For example:

```
// Using datasets
dataset ds1 = {1, 2, 3, 4};
dataset ds2 = {2, 3, 5, 6};
// Return interpolated value in ds2 where X in ds1 is 1.23
ds2(1.23, ds1) = ;  // Return 2.23
// Return interpolated value in ds1 where X in ds2 is 5.28
ds1(5.28, ds2) = ;  // Return 3.28

// Using ranges
newbook;
wks.ncols = 3;
range r1 = 2;  // Column 2 in active worksheet
r1 = {1, 2, 3, 4};
range r2 = 3;  // Column 3 in active worksheet;
r2 = {2, 3, 5, 6};
r2(1.23, r1) = ;
r1(5.28, r2) = ;

// Using columns
col(3)(1.23, col(2)) = ;
```

```
col(2)(5.28, col(3)) = ;
```

Creating Interpolated Curves

X-Functions for Interpolation of Curves

Origin provides three X-Functions for interpolating XY data and creating a new output XY data pair:

Name	Brief Description
interp1xy	Perform interpolation of XY data and generate output at uniformly spaced X
interp1	Perform interpolation of XY data and generate output at a given set of X values
interp1trace	Perform interpolation of XY data that is not monotonic in X

Using Existing X Dataset

The following example shows how to use an existing X dataset to find interpolated Y values:

```
// Create a new workbook with specific column designations
newbook sheet:=0;
newsheet cols:=4 xy:="XYXY";
// Import a sample data file
fname$ = system.path.program$ + "Samples\Mathematics\Interpolation.dat";
impasc;

// Interpolate the data in col(1) and col(2) with the X values in col(3)
range rResult=col(4);
interp1 ix:=col(3) iy:=(col(1), col(2)) method:=linear ox:=rResult;

//Plot the original data and the result
plotxy iy:=col(2) plot:=202 color:=1;
plotxy iy:=rResult plot:=202 color:=2 size:=5 ogl:=1;
```

Uniformly Spaced X Output

The following example performs interpolation by generating uniformly spaced X output:

```
//Create a new workbook and import a data file
fname$ = system.path.program$ + "Samples\Mathematics\Sine Curve.dat";
newbook;
impasc;
```

```
//Interpolate the data in column 2
interp1xy iy:=col(2) method:=bspline npts:=50;
range rResult = col(3);

//Plot the original data and the result
plotxy iy:=col(2) plot:=202 color:=1;
plotxy iy:=rResult plot:=202 color:=2 size:=5 ogl:=1;
```

Interpolating Non-Monotonic Data

The following example performs trace interpolation on data where X is not monotonic:

```
//Create a new workbook and import the data file
fname$ = system.path.program$ + "Samples\Mathematics\circle.dat";
newbook;
impasc;

//Interpolate the circular data in column 2 with trace interpolation
interp1trace iy:=Col(2) method:=bspline;
range rResult= col(4);

//Plot the original data and the result
plotxy iy:=col(2) plot:=202 color:=1;
plotxy iy:=rResult plot:=202 color:=2 size:=1 ogl:=1;
```

Note that the interpolation X-Functions can also be used for extrapolating Y values outside of the X range of the input data.

Matrix Interpolation

The **minterp2** X-Function can be used to perform interpolation/extrapolation of matrices.

```
// Create a new matrix book and import sample data;
newbook mat:=1;
filepath$ = "Samples\Matrix Conversion and Gridding\Direct.dat";
string fname$=system.path.program$ + filepath$;
impasc;
// Interpolate to a matrix with 10 times the x and y size of the original
range rin = 1; // point to matrix with input data;
int nx, ny;
nx = rin.ncols * 10;
ny = rin.nrows * 10;
minterp2 method:=bicubic cols:=nx rows:=ny ;
```

OriginPro also offers the **interp3** X-Function which can be used to perform interpolation on 4-dimensional scatter data.

14.2 Statistics

This is an example-based section demonstrating support for several types of statistical tests implemented in script through X-Function calls.

14.2.1 Descriptive statistics

Origin provides several X-Functions to compute descriptive statistics, some of the most common are:

Name	Brief Description
colstats	Columnwise statistics
corrcoef	Correlation Coefficient
freqcounts	Frequency counts of a data set.
mstats	Compute descriptive statistics on a matrix
rowstats	Statistics of a row of data
stats	Treat selected columns as a complete dataset; compute statistics of the dataset.

For a full description of each of these X-Functions and its inputs and outputs, please see the Descriptive Statistics.

Descriptive Statistics on Columns and Rows

The **colstats** X-Function can perform statistics on columns. By default, it outputs the mean, the standard deviation, the number of data points and the median of each input column. But you can customize the output by assigning different values to the variables. In the following example, **colstats** is used to calculate the means, the standard deviations, the standard errors of the means, and the medians of four columns.

```
//Import a sample data with four columns
newbook;
fname$ = system.path.program$ + "Samples\Statistics\nitrogen_raw.txt";
impasc;

//Perform statistics on column 1 to 4
colstats irng:=1:4 sem:=<new> n:=<none>;
```

The **rowstats** X-Function can be used in a similar way. The following example calculates the means of the active worksheet; the results are placed in a new added column at the first of the worksheet.

> **Note: mean** and **sd** are defaulted to be <new> in output, if not needed, set to <none>.

```
newbook;
fname$ = system.path.program$ + "Samples\Statistics\engine.txt";
impasc; //Import a sample data

wunstackcol irng1:=1 irng2:=2; //Unstack columns
wtranspose type:=all ow:=<new>; //Transpose worksheet
range rr1 = 1:2;
delete rr1;
range rr2 = 2;
delete rr2; //delete empty columns
int nn = wks.ncols;
wks.addcol();
wks.col$(nn+1).lname$ = Mean;
wks.col$(nn+1).index = 2; //Add mean column
wks.addcol();
wks.col$(nn+2).lname$ = Sum;
wks.col$(nn+2).index = 3; //Add sum column

//Row statistics to get sum and average, saved to corresponding column.
rowstats irng:=4[1]:end[end] sum:=3 mean:=2 sd:=<none>;
```

Frequency Count

If you want to calculate the frequency counts of a range of data, use the **freqcounts** X-Function.

```
//Open a sample workbook
%a = system.path.program$ + "Samples\Statistics\Body.ogw";
doc -a %a;

//Count the frequency of the data in column 4
freqcounts irng:=4 min:=35 max:=75 stepby:=increment intervals:=5;
```

Correlation Coefficient

corrcoef X-Function can be used to compute the correlation coefficient between two datasets.

```
//import a sample data
newbook;
fname$ = system.path.program$ + "Samples\Statistics\automobile.dat";
impasc;
```

```
//Correlation Coefficient
corrcoef irng:= (col(c):col(g)) rt:= <new name:=corr>
```

14.2.2 Hypothesis Testing

Origin/OriginPro supports the following set of X-Functions for hypothesis testing:

Name	Brief Description
rowttest2 (Pro Only)	Perform a two-sample t-test on rows.
ttest1	Compare the sample mean to the hypothesized population mean.
ttest2	Compare the sample means of two samples.
ttestpair	Determine whether two sample means are equal in the case that they are matched.
vartest1	Determine whether the sample variance is equal to a specified value.
vartest2	Determine whether two sample variances are equal.

For a full description of these X-functions, including input and output arguments, please see the Hypothesis Testing.

One-Sample T-Test

If you need to know whether the mean value of a sample is consistent with a hypothetical value for a given confidence level, consider using the **one-sample T-test**. Note that this test assumes that the sample is a normally distributed population. Before we apply the one-sample T-test, we should verify this assumption.

```
//Import a sample data
newbook;
fname$ = system.path.program$ + "Samples\Statistics\diameter.dat";
impasc;

//Normality test
swtest irng:=col(a) prob:=p1;
if (p1 < 0.05)
{
   type "The sample is not likely to follow a normal distribution."
}
```

```
else
{
   // Test whether the mean is 21
   ttest1 irng:=col(1) mean:=21 tail:=two prob:=p2;
   if (p2 < 0.05) {
        type "At the 0.05 level, the population mean is";
               type "significantly different from 21."; }
   else {
        type "At the 0.05 level, the population mean is NOT";
               type "significantly different from 21."; }
}
```

Two-Sample T-Test

The **ttest2** X-Function is provided for performing **two-sample t-test**. The example below shows how to use it and print the results.

```
// Import sample data
newbook;
string fpath$ = "Samples\Statistics\time_raw.dat";
string fname$ = system.path.program$ + fpath$;
impAsc;

// Perform two-sample t-test on two columns
// Sample variance is not assumed to be equal
ttest2 irng:=(col(1), col(2)) equal:=0;

// Type some results
type "Value of t-test statistic is $(ttest2.stat)";
type "Degree of freedom is $(ttest2.df)";
type "P-value is $(ttest2.prob)";
type "Conf. levels in 95% is ($(ttest2.lcl), $(ttest2.ucl))";
```

The **rowttest2** X-Function can be used to perform a **two-sample T-test** on rows. The following example demonstrates how to compute the corresponding probability value for each row:

```
// Import sample data
newbook;
string fpath$ = "Samples\Statistics\ANOVA\Two-Way_ANOVA_raw.dat";
fname$ = system.path.program$ + fpath$;
impasc;

// Two-sample T-test on a row
rowttest2 irng1:=(col(a):col(c)) irng2:=(col(d):col(f))
          tail:=two prob:=<new>;
```

Pair-Sample T-Test

Origin provides the ttestpair X-Function for **pair-sample t-test** analysis, so to determine whether the means of two same-sized and dependent samples from a normal distribution are equal or not, and calculates the confidence interval for the difference between the means. The example below first imports a data file, and then perform **pair-sample t-test**, and then output the related results.

```
// Import sample data
newbook;
string fpath$ = "Samples\Statistics\abrasion_raw.dat";
string fname$ = system.path.program$ + fpath$;
impasc;

// Perform pair-sample t-test one two columns
// Hypothetical means difference is 0.5
// And Tail is upper tailed
ttestpair irng:=(col(1), col(2)) mdiff:=0.5 tail:=upper;

// Type the results
type "Value of paired-sample t-test statistic is $(ttestpair.stat)";
type "Degree of freedom for the paired-sample t-test is $(ttestpair.df)";
type "P-value is $(ttestpair.prob)";
type "Conf. levels in 95% is ($(ttestpair.lcl), $(ttestpair.ucl))";
```

One-Sample Test for Variance

X-Function vartest1 is used to perform a chi-squared variance test, so to determine whether the sample from a normal distribution could have a given hypothetical vaiance value. The following example will perform **one-sample test for variance**, and output the P-value.

```
// Import sample data
newbook;
string fpath$ = "Samples\Statistics\vartest1.dat";
string fname$ = system.path.program$ + fpath$;
impasc;

// Perform F-test
// Tail is two tailed
// Test variance is 2.0
// P-value stored in variable p
vartest1 irng:=col(1) var:=2.0 tail:=two prob:=p;

// Ouput P-value
p = ;
```

Two-Sample Test for Variance (F-Test)

F-test, also called **two-sample test for variance**, is performed by using vartest2 X-Function.

```
// Import sample data
newbook;
string fpath$ = "Samples\Statistics\time_raw.dat";
string fname$ = system.path.program$ + fpath$;
impasc;

// Perform F-test
// And Tail is upper tailed
vartest2 irng:=(col(1), col(2)) tail:=upper;

// Output the result tree
vartest2.=;
```

14.2.3 Nonparametric Tests

Hypothesis tests are parametric tests when they assume the population follows some specific distribution (such as normal) with a set of parameters. If you don't know whether your data follows normal distribution or you have confirmed that your data do not follow normal distribution, you should use nonparametric tests.

Origin provides support for the following X-Functions for non-parametric analysis:

Name	Brief Description
signrank1	Test whether the location (median) of a population distribution is the same with a specified value
signrank2/sign2	Test whether or not the medians of the paired populations are equal. Input data should be in raw format.
mwtest/kstest2	Test whether the two samples have identical distribution. Input data should be Indexed.
kwanova/mediantest	Test whether different samples' medians are equal, Input data should be arranged in index mode.
friedman	Compares three or more paired groups. Input data should be arranged in index.

As an example, we want to compare the height of boys and girls in high school.

```
//import a sample data
newbook;
```

```
fname$ = system.path.program$ + "Samples\Statistics\body.dat";
impasc;

//Mann-Whitney Test for Two Sample
//output result to a new sheet named mynw
mwtest irng:=(col(c), col(d)) tail:=two rt:=<new name:=mynw>;

//get result from output result sheet
page.active$="mynw";

getresults tr:=mynw;

//Use the result to draw conclusion
if (mynw.Stats.Stats.C3 <= 0.05); //if probability is less than 0.05
{
   type "At 0.05 level, height of boys and girls are differnt.";
   //if median of girls height is larger than median of boy's height
   if (mynw.DescStats.R1.Median >= mynw.DescStats.R2.Median)
        type "girls are taller than boys.";
   else
        type "boys are taller than girls."
}
else
{
   type "The girls are as tall as the boys."
}
```

14.2.4 Survival Analysis

Survival Analysis is widely used in the biosciences to quantify survivorship in a population under study. Origin supports three widely used tests:

Name	Brief Description
kaplanmeier	Kaplan-Meier (product-limit) Estimator
phm_cox	Cox Proportional Hazards Model
weibullfit	Weibull Fit

For a full description of these X-functions, including input and output arguments, please see the Survival Analysis.

Kaplan-Meier Estimator

If you want to estimate the survival ratio, create survival plots and compare the quality of survival functions, use the **kaplanmeier** X-Function. It uses product-limit method to estimate the survival function, and supports three methods for testing the equality of the survival function: Log Rank, Breslow and Tarone-Ware.

As an example, scientists are looking for a better medicine for cancer resistance. After exposing some rats to carcinogen DMBA, they apply different medicine to two different groups of rats and record their survival status for the first 60 hours. They wish to quantify the difference in survival rates between the two medicines.

```
// Import sample data
newbook;
fname$ = system.path.program$ + "Samples\Statistics\SurvivedRats.dat";
impasc;

//Perform Kaplan-Meier Analysis
kaplanmeier irng:=(1,2,3) censor:=0 logrank:=1
            rd:=<new name:="sf">
            rt:=<new name:="km">;

//Get result from survival report tree
getresults tr:=mykm iw:="km";

if (mykm.comp.logrank.prob <= 0.05)
{
    type "The two medicines have significantly different"
    type "effects on survival at the 0.05 level ...";
    type "Please see the survival plot.";

    //Plot survival Function
    page.active$="sf";
    plotxy iy:=(?, 1:end) plot:=200 o:=[<new template:=survivalsf>];

}
else
{
    type "The two medicines are not significantly different.";
}
```

Cox Proportional Hazard Regression

The **phm_cox** X-Function can be used to obtain the parameter estimates and other statistics associated with the Cox Proportional hazards model for fixed covariates. It can then forecast the change in the hazard rate along with several fixed covariates.

For example, we want to study on 66 patients with colorectal carcinoma to determine the effective prognostic parameter and the best prognostic index (a prognostic parameter is a parameter that determines whether a person has a certain illness). This script implements the **phm_cox** X-Function to get the relevant statistics.

```
//import a sample data
newbook;
string fpath$ = "Samples\Statistics\ColorectalCarcinoma.dat";
fname$ = system.path.program$ + fpath$;
impasc option.hdr.LNames:=1
       option.hdr.units:=0
       option.hdr.CommsFrom:=2
       option.hdr.CommsTo:=2;

//Perform Cox Regression
phm_Cox irng:=(col(1),col(2),col(3):end) censor:=0 rt:=<new name:="cox">;

//Get result from report tree
page.active$="cox";
getresults tr:=cox;

type "Prognostic parameters determining colorectal carcinoma are:";

page.active$="ColorectalCarcinoma";
loop(ii, 1, 7)
{
  // If probability is less than 0.05,
  // we can say it is effective for survival time.
  if (cox.paramestim.param$(ii).prob<=0.05)
    type wks.col$(ii+2).comment$;
}
```

Weibull Fit

If it is known *apriori* that data are Weibull distributed, use the **weibullfit** X-Function to estimate the weibull parameters.

```
//import a sample data
newbook;
fname$ = system.path.program$ + "Samples\Statistics\Weibull Fit.dat ";
impasc;

//Perform Weibull Fit
weibullfit irng:=(col(a), col(b)) censor:=1;
```

14.3 Curve Fitting

The curve fitting features in Origin are some of the most popular and widely used. Many users do not realize that the X-Functions performing the fitting calculations can be used just as easily from script as they can from Origin's graphical user interfaces. The following sections address curve fitting using LabTalk Script.

14.3.1 Linear, Polynomial and Multiple Regression

In LabTalk scripts, three simple quick use X-Functions, **fitLR**, **fitPoly**, and **fitMR**, are available for performing linear regression, polynomial regression, and multiple linear regression, respectively. And the **-h** switch can be used to see the argument list.

Linear Regression

fitLR finds a best fit straight line to a given dataset.

```
newbook;  // create a new book

// file name
string strFile$ = system.path.program$ + "Samples\Curve Fitting\Linear Fit.dat";
impasc fname:=strFile$;  // import the data

wks.addcol(FitData);  // add a column for fitted data, named FitData

// perform linear fit on the first ten points of column 1 (X) and column 2 (Y)
// and the fitted data is output to FitData column
fitLR iy:=(1,2) N:=10 oy:=col(FitData);
// a tree object named fitLR is created, and contains the output values
fitLR.a = ;  // output the fitted intercept
fitLR.b = ;  // output the fitted slope
fitLR.= ;  // output all the results, which include fitted intercept and slope
```

More examples about linear regression can be found in Curve Fitting sample page, or under **Fitting** category in **XF Script Dialog** (press F11 to open).

Polynomial Regression

Polynomial fitting is a special case wherein the fitting function is mathematically non-linear, but an analytical (non-iterative) solution is obtained. In LabTalk, **fitPoly** is used to control polynomial fitting.

```
newbook;  // create a new book;

// file name
string strFile$ = system.path.program$ + "Samples\Curve Fitting\Polynomial Fit.dat";
```

```
impasc fname:=strFile$;  // import data
wks.addcol(PolyCoef);  // add a new column for polynomial coefficients
wks.addcol(FittedX);  // add a new column for fitted X values
wks.addcol(FittedY);  // add a new column for fitted Y values

// perform polynomial fitting on column 1 (X) and column 3 (Y)
// polynomial order is 3
fitPoly iy:=(1,3) polyorder:=3 coef:=col(PolyCoef) oy:=(col(FittedX),col(FittedY));

// the results are stored in the tree named fitPoly, output it
fitPoly.= ;
```

Additionally, **fitPoly** provides the outputs for adjusted residual sum of squares, coefficient of determination, and errors in polynomial coefficients. For more detailed examples, please refer to Curve Fitting sample page, or **Fitting** category in **XF Script Dialog** (press F11 to open).

Multiple Linear Regression

Multiple linear regression studies the relationship between several predictor variables and a response variable, which is an extension of simple linear regression.

```
// create a new book and import some data
newbook;
fn$ = system.path.program$ + "Samples\Curve Fitting\Multiple Linear Regression.dat";
impasc fn$;
wks.addcol(FitValue);  // add a column for fitted values of dependent

// perform multiple linear regression
// column D is dependent, and column A, B, and C are independents
// the output results are stored in a tree, tr
fitMR dep:=col(D) indep:=col(A):col(C) mrtree:=tr odep:=col(FitValue);
tr.= ;  // output the result tree
```

For more examples, please refer to Curve Fitting sample page, or **Fitting** category in **XF Script Dialog** (press F11 to open).

Run Operation Classes to Perform Regression

The X-Functions depicted above are for simple quick use only to perform linear, polynomial and multiple regression. That is to say, some quantities are not available when using these three X-Functions. For full access to all quantities, the X-Function **xop** is provided to invoke the internal menu commands (operation commands), so to run the corresponding operation classes to perform regression. The following example shows how to use the X-Function **xop** to perform linear fit, and generate a report.

```
// create a new book and import data
newbook;
fname$ = system.path.program$ + "Samples\Curve Fitting\Linear Fit.dat";
impasc fname$;
```

```
tree lrGUI;  // GUI tree for linear fit
// initialize the GUI tree, with the FitLinear class
xop execute:=init classname:=FitLinear iotrgui:=lrGUI;

// specify the input data in the GUI tree
lrGUI.GUI.InputData.Range1.X$ = col(A);
lrGUI.GUI.InputData.Range1.Y$ = col(C);

// perform linear fit and generate a report with the prepared GUI tree
xop execute:=report iotrgui:=lrGUI;

xop execute:=cleanup;  // clean up linear fit operation objects after fitting
```

14.3.2 Non-linear Fitting

Non-linear fitting in LabTalk is X-function based and proceeds in three steps, each calling (at least) one X-function:

1. nlbegin: Begin the fitting process. Define input data, type of fitting function, and input parameters.
2. nlfit: Perform the fit calculations
3. nlend Choose which parameters to output and in what format

Besides *nlbegin*, you can also start a fitting process according to your fitting model or data by the following X-Functions:

- nlbeginr: Fitting multiple dependnet/independent variables' model
- nlbeginm: Fitting a matrix
- nlbeginz: Fitting XYZ worksheet data

Script Example

Here is a script example of the steps outlined above:

```
// Begin non-linear fitting, taking input data from Column 1 (X) and
// Column 2 (Y) of the active worksheet,
// specifying the fitting function as Gaussian,
// and creating the input parameter tree named ParamTree:
nlbegin iy:=(1,2) func:=gauss nltree:=ParamTree;
// Optional: let the peak center be fixed at X = 5
ParamTree.xc = 5;     // Assign the peak center an X-value of 5.
ParamTree.f_xc = 1;   // Fix the peak center (f_xc = 0 is unfixed).
// Perform the fit calculations:
nlfit;
// Optional: report results to the Script Window.
```

```
type Baseline y0 is $(ParamTree.y0),;
type Peak Center is $(ParamTree.xc), and;
type Peak width (FWHM) is $(ParamTree.w);
// end the fitting session without a Report Sheet
nlend;
```

Notes on the Parameter Tree

The data tree that stores the fit parameters has many options besides the few mentioned in the example above. The following script command allows you to see all of the tree nodes (names and values) at one time, displaying them in the **Script Window**.

```
// To see the entire tree structure with values:
ParamTree.=;
```

Note: since the non-linear fitting procedure is iterative, parameter values for the fit that are not fixed (by setting the fix option to zero in the parameter tree) can and will change from their initial values. Initial parameters can be set manually, as illustrated in the example above by accessing individual nodes of the parameter tree, or can be set automatically by Origin (see the **nlfn** X-function in the table below).

Table of X-functions Supporting Non-Linear Fitting

In addition to the three given above, there are a few other X-functions that facilitate non-linear fitting. The following table summarizes the X-functions used to control non-linear fitting:

Name	Brief Description
nlbegin	Start a LabTalk nlfit session on XY data from worksheet or graph. Note: This X-Function fits one independent/dependent model only. For multiple dependent/independent functions, use *nlbeginr* instead.
nlbeginr	Start a LabTalk nlfit session on worksheet data. It is used for fitting multiple dependent/independent variables functions.
nlbeginm	Start a LabTalk nlfit session on matrix data from matrix object or graph
nlbeginz	Start a LabTalk nlfit session on XYZ data from worksheet or graph
nlfn	Set Automatic Parameter Initialization option
nlpara	Open the Parameter dialog for GUI editing of parameter values and bounds

nlfit	Perform iterations to fit the data
nlend	End the fitting session and optionally create a report

For a full description of each of these X-functions and its inputs and outputs, please see the X-function Reference.

Qualitative Differences from Linear Fitting

Unlike linear fitting, a non-linear fit involves solving equations to which there is no analytical solution, thus requiring an iterative approach. But the idea---calling X-functions to perform the analysis---is the same. Whereas a linear fit can be performed in just one line of script with just one X-function call (see the Linear Fitting section), a non-linear fit requires calling at least three X-functions.

14.4 Signal Processing

Origin provides a collection of X-functions for signal processing, ranging from smoothing noisy data to Fourier Transform (FFT), Short-time FFT, Convolution and Correlation, FFT Filtering, and Wavelet analysis.

These X-Functions are available under the **Signal Processing** category and can be listed by typing the following command:

```
lx cat:="signal processing*";
```

Some functionality such as Short-time FFT and Wavelets are only available in OriginPro.

The following sections provide some short examples of calling the signal processing X-Functions from script.

14.4.1 Smoothing

Smoothing noisy data can be performed by using the **smooth** X-Function.

```
// Smooth the XY data in columns 1,2 of the worksheet
// using SavitzkyGolay method with a third order polynomial
range r=(1,2); // assume worksheet active with XY data
smooth iy:=r meth:=sg poly:=3;
```

To smooth all plots in a layer, you can loop over the plots as below:

```
// Count the number of data plots in the layer and save result in
//variable "count"
layer -c;
// Get the name of this Graph page
string gname$ = %H;
```

```
// Create a new book named smooth - actual name is stored in bkname$
newbook na:=Smoothed;
// Start with no columns
wks.ncols=0;
loop(ii,1,count) {
    // Input Range refers to 'ii'th plot
    range riy = [gname$]!$(ii);
    // Output Range refers to two, new columns
    range roy = [bkname$]!($(ii*2-1),$(ii*2));
    // Savitsky-Golay smoothing using third order polynomial
    smooth iy:=riy meth:=sg poly:=3 oy:=roy;
}
```

14.4.2 FFT and Filtering

The following example shows how to perform 1D FFT of data using the **fft1** X-Function.

```
// Import a sample file
newbook;
fname$ = system.path.program$ + "Samples\Signal Processing\fftfilter1.dat";
impasc;
// Perform FFT and get output into a named tree
Tree myfft;
fft1 ix:=2 rd:=myfft rt:=<none>;
// You can list all trees using the command: list vt
```

Once you have results in a tree, you can do further analysis on the output such as:

```
// Copy desired tree vector nodes to datasets
// Locate the peak and mean frequency components
dataset tmp_x=myfft.fft.freq;
dataset tmp_y=myfft.fft.amp;
// Perform stats and output results
percentile = {0:10:100};
diststats iy:=(tmp_x, tmp_y) percent:=percentile;
type "The mean frequency is $(diststats.mean)";
```

The following example shows how to perform signal filtering using the **fft_filters** X-Function:

```
// Import some data with noise and create graph
newbook;
string filepath$ = "Samples\Signal Processing\";
string filename$ = "Signal with High Frequency Noise.dat";
fname$ = system.path.program$ + filepath$ + filename$;
impasc;
plotxy iy:=(1,2) plot:=line;

// Perform low pass filtering
```

```
fft_filters filter:=lowpass cutoff:=1.5;
```

14.5 Peaks and Baseline

This section deals with Origin's X-Functions that perform peak and baseline calculations, especially valuable for analyses pertaining to spectroscopy.

14.5.1 X-Functions For Peak Analysis

The following table lists the X-Functions available for peak analysis. You can obtain more information on these functions from the X-Function Reference help file.

Name	Brief Description
pa	Perform peak analysis with a pre-saved **Peak Analyzer** theme file.
paMultiY	Perform batch processing of peak analysis on multiple Y datasets
pkFind	Pick peaks.
fitpeaks	Fit multiple peaks.
blauto	Create baseline anchor points.
interp1xy	Interpolate the baseline anchor points to create baseline.
subtract_ref	Subtract existing baseline dataset from source data.
smooth	Smooth the input prior to performing peak analysis.
integ1	Perform integration on the selected range or peak.

For peaks that do not require baseline treatment or other advanced options, you can also use peak functions to perform nonlinear fitting. For more information on non-linear fitting from script, please see the Curve Fitting section.

The following sections provide examples on peak analysis.

14.5.2 Creating a Baseline

This example imports a sample data file and creates baseline anchor points using the **blauto** X-Function.

```
newbook;
filepath$ = "Samples\Spectroscopy\Peaks on Exponential Baseline.dat";
fname$ = system.path.program$ + filepath$;
impASC;

//Create 20 baseline anchor points
range rData = (1,2), rBase =(3, 4);
blauto iy:=rData number:=20 oy:=rBase;
```

Plot the data and anchor points in same graph:

```
// plot a line graph of the data
plotxy rData 200 o:=[<new>];
// plot baseline pts to same layer as scatter
plotxy rBase 201 color:=2 o:=1!;
```

14.5.3 Finding Peaks

This example uses the **pkfind** X-Function to find peaks in XY data:

```
// Import sample pulse data
newbook;
fname$ = system.path.program$ + "Samples\Spectroscopy\Sample Pulses.dat";
impASC;
// Find all positive peaks above a peak height value of 0.2
range rin=(1,2);
range routx = 3, routy=4;
pkfind iy:=rin dir:=p method:=max npts:=5 filter:=h value:=0.2
       ocenter:=<none> ocenter_x:=routx ocenter_y:=routy;
```

Now graph the data as line plot and the peak x,y as scatter:

```
plotxy iy:=rin plot:=200;
// Set x output column as type X and plot the Y column
routx.type = 4;
plotxy iy:=routy plot:=201 color:=2 o:=1;
```

14.5.4 Integrating and Fitting Peaks

X-Functions specific to the goals of directly integrating peaks, or fitting multiple peaks, do not exist. Therefore, to perform peak fitting or integration, one must first use the **Peak Analyzer**

dialog to create and save a theme file. Once a theme file has been saved, the **pa** or **paMultiY** X-Functions can be utilized to perform integration and peak fitting from script.

14.6 Image Processing

Origin 8 offers enhanced image processing capabilities compared with earlier versions of Origin. A few examples of basic image processing are shown below, along with LabTalk scripts for performing the necessary tasks.

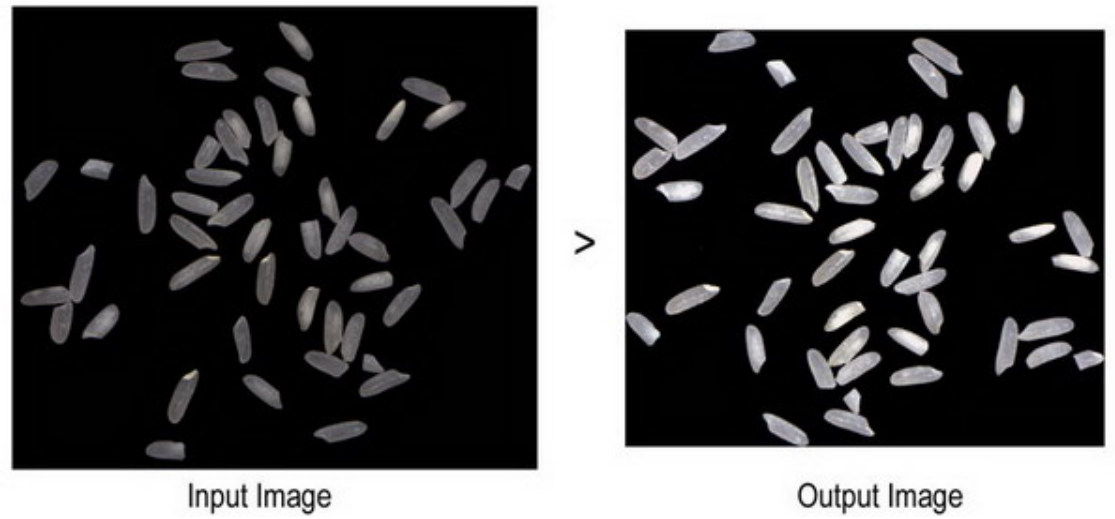
To view a list of all X-Functions available for image processing, please type the following command:

```
lx cat:="image*";
```

Some of the X-Functions are only available in OriginPro.

14.6.1 Rotate and Make Image Compact

This example rotates, trims the margins, and applies an auto-level to make the image more compact and clear.,

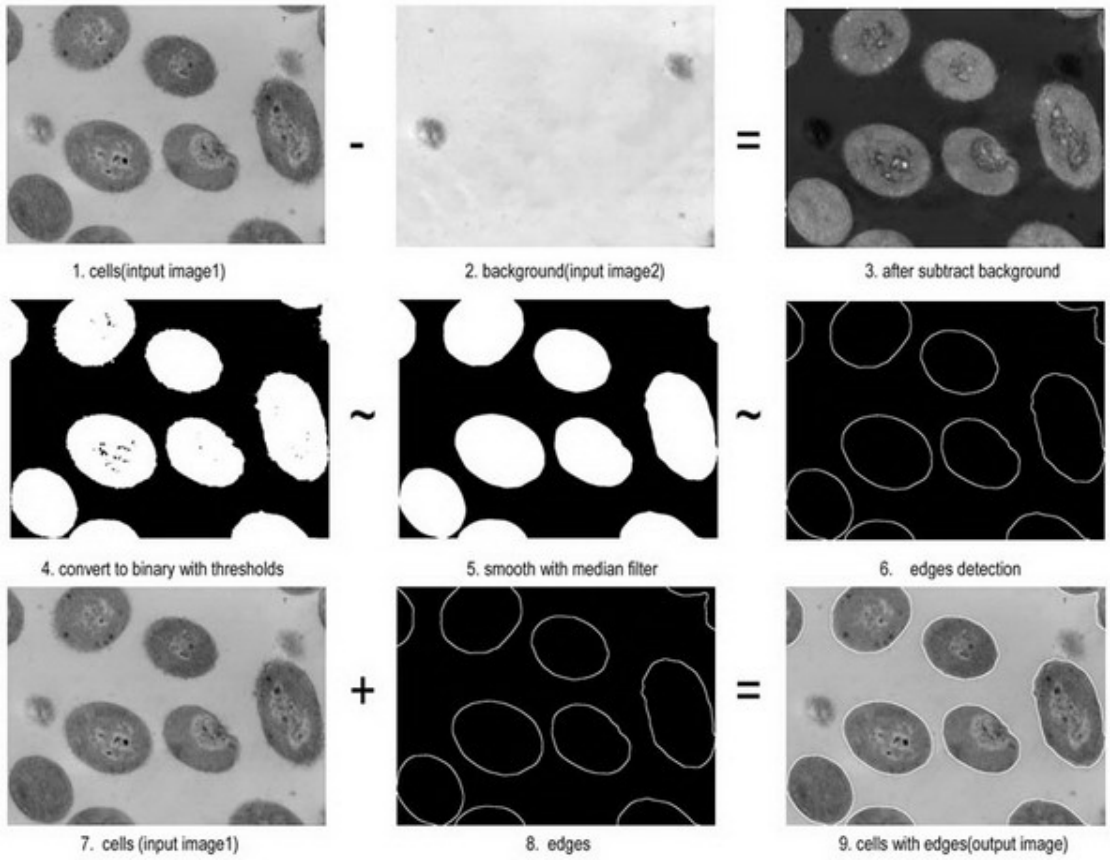



```
//Create a new folder in the Project Explorer
pe_mkdir RotateTrim path:=aa$;
pe_cd aa$;

//Create a matrix and import an image into it
window -t m;
string fpath$ = "samples\Image Processing and Analysis\rice.bmp";
string fname$ = System.path.program$ + fpath$;
```

```
impimage;
window -r %h Original;

//Get the dimension of the original image
matrix -pg DIM nCol1 nRow1;

window -d;        //Duplicate the image
window -r %h Modified;

imgRotate angle:=42;
imgTrim t:=17;

matrix -pg DIM nCol2 nRow2; //Get the dimension of the modified iamge

imgAutoLevel;// Apply auto leveling to image

window -s T;      //Tile the windows horizontally

//Report
window -n n Report;
old = type.redirection;
type.redirection = 2;
type.notes$=Report;
type "Dimension of the original image: ";
type "   $(nCol1) * $(nRow1)\r\n";   //  "754 * 668"
type "Dimension of the modified image: ";   //  "688 * 601"
type "   $(nCol2) * $(nRow2)\r\n";
type.redirection = old;
}
```

14.6.2 Edge Detection

Subtract background from Cells image then detect the edges.

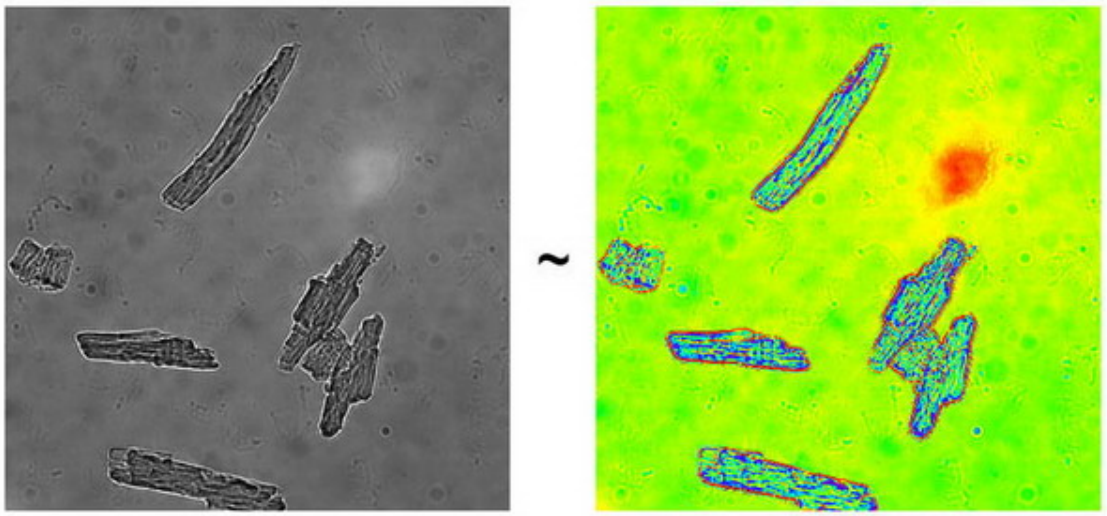



```
//Create a new folder in the Project Explorer
pe_mkdir EdgeDetection path:=aa$;
pe_cd aa$;

//Create a matrix and import the cell image into it
window -t m;
string fpath$ = "samples\Image Processing and Analysis\cell.jpg";
string fname$ = System.path.program$ + fpath$;
impimage;
cell$ = %h;

//Create a matrix and import the background image into it
window -t m;
string fpath$ = "samples\Image Processing and Analysis\bgnd.jpg";
```

```
string fname$ = System.path.program$ + fpath$;
cellbk$ = %h;
impimage;

//Subtract background and pre-processing
//x, y is the offset of Image2
imgSimpleMath img1:=cellbk$ img2:=cell$ func:=sub12 x:=7 y:=13 crop:=1;
//specify the lowest and highest intensity to be convert to binary 0 or 1.
imgBinary t1:=65 t2:=255;
// the dimensions of median filter is 18
imgMedian d:=18;

//Edge detection
// the threshold value 12 used to determine edge pixels,
// and shv(Sobel horizontal & vertical) Edge detection filter is applied.
imgEdge t:=12 f:=shv;
edge$ = %h;

//Add the edges back to the cell image
imgSimpleMath img1:=edge$ img2:=cell$ func:=add;
window -z;
```

14.6.3 Apply Rainbow Palette to Gray Image

This example shows how to convert a gray image to rainbow color.




```
pe_mkdir Conversion path:=aa$;
```

```
pe_cd aa$;

//Create a matrix and import a sample image
window -t m;
path$ = System.path.program$;
fname$ = path$ + "samples\Image Processing and Analysis\myocyte8.tif";
impimage;
window -r %h Original;

window -d;        //Duplicate the image
window -r %h newimage;

imgC2gray;        //Convert to gray

//Apply pallete
fname$ = System.path.program$ + "palettes\Rainbow.PAL";
imgpalette palfile:=fname$;

window -s T;      //Tile the windows horizontally
```

14.6.4 Converting Image to Data

When an image is imported into a matrix object, it is kept as type **Image**, indicated by the icon **I** on the top right corner of the window. For certain mathematical operations such as **2D FFT** the type needs to be converted to **Data**, which would then be indicated by the icon **D** at the top right corner.

This script example shows importing multiple images into a matrix book and converting them to type **data**:

```
// Find files using wildcard
string path$=system.path.program$+"Samples\Image Processing and Analysis";
findFiles ext:="*tif*";

// Create a new matrix book and import all images as new sheets
newbook mat:=1;
impImage options.FirstMode:=0 options.Mode:=4;
// Loop over all sheets and convert image to byte data
doc -e LW {
   img2m om:=<input> type:=1;
}
```

15 User Interaction

There may be times when you would like to provide a specific type of input to your script that would be difficult to automate. For instance, you wish to specify a particular data point on a graph, or a certain cell in a worksheet as input to one or more functions called from script. To do this, LabTalk supports ways of prompting the user for input while running a script.

In general, consecutive lines of a script are executed until such a user prompt is encountered. Execution of the script then halts until the user has entered the desired information, and then proceeds. The following sections demonstrate several examples of programming for this type of user interaction:

15.1 Getting Numeric and String Input

This section gives examples of prompting for three types of user input during script execution:

1. Yes/No response
2. Single String
3. Multi-Type Input (GetN)

15.1.1 Get a Yes/No Response

The GetYesNo command can be used to get a Yes or No response from the user. The command takes three arguments:

Syntax: **getyesno** *stringMessageToUser numericVariableName windowTitle*

For example, entering the following line in the Script Window will generate a pop-up window titled **Check Sign of X** and ask the user the Yes/No question **Should X be positive?** with the options **Yes**, **No**, and **Cancel** as clickable buttons. If **Yes** is selected, **xpos** will be assigned a value of 1. If **No** is selected, **xpos** will be assigned the value 0. If **Cancel** is selected, **xpos** will be assigned the value 0, **#Command Error!** will be printed, and script execution will stop.

```
getyesno "Should X be positive?" xpos "Check Sign of X"
```

If additional script processing is required in any event, this command should be called from elsewhere and the numeric value can be tested. In the following example, **getyesno** is called from its own section of code and the two string inputs are passed as arguments to the section(note, a multi-section LabTalk script will not work if simply pasted to the script window; save to file and run):

```
[Main]
// Here is the calling code
```

```
int iVal = -1;
run.section(,myGetYesNo,"Create a Graph of results?" "Graphing Option");
if( iVal > 0 )
{
   type "Graph generated";        // Yes response
}
else
{
   type "Graph NOT generated";    // No or Cancel response
}

// 'myGetYesNo' section
[myGetYesNo]
getyesno (%1) iVal (%2);
```

15.1.2 Get a String

GetString can be used for user entry of a single string.

```
%B = "";
GetString (Enter as Last, First) Last (Your Name);
// Cancel stops here unless using technique as in GetYesNo
if("%B"!="Last")
{
    type User entered %B.;
}
else
{
    type User clicked OK, but did not modify text;
}
```

15.1.3 Get Multiple Values

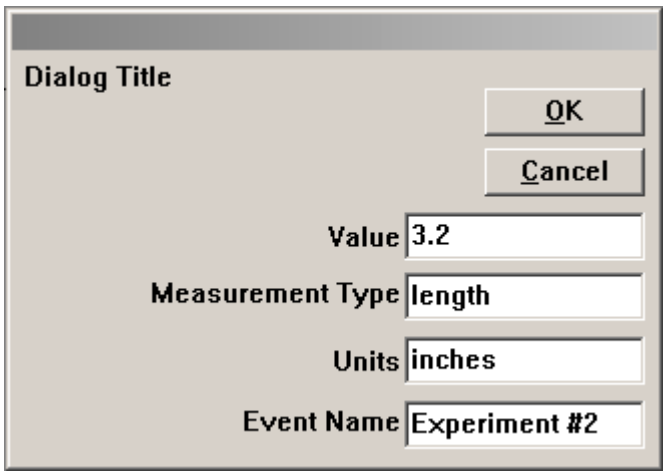
The GetN or GetNumber dialog prompts a user for a number, a string or a list entry (in previous versions of Origin only numeric values were possible, hence the name). Starting with Origin 8.1, ***GetNumber*** will accept both string variables (i.e., string str1$) and string registers (i.e., %A) for string input. Previous versions support string registers only. **GetN** currently accepts up to 7 variables in addition to the dialog title.

With the increased functionality of ***GetN*** in Origin 8.1, string variables can be used in the command call. In this case, the strings must first be declared. It is always a good practice to create variables by declaration rather than by assignment alone; for more see Scope of (String) Variables. For example:

```
// First, declare the variables to be used:
double nn = 3.2;
```

```
string measurement$="length", units$="inches", event$="Experiment #2";

// Use GetN dialog to collect user data:
getn
(Value) nn
(Measurement Type) measurement$
(Units) units$
(Event Name) event$
(Dialog Title);
```

brings up the following dialog, prompting the user for input:



The values entered in this dialog will be assigned to the declared variables. If the variables have an initial value (before **GetN** is called), that value will show up in the input box, otherwise the input box will appear blank. In either case, the initial value can be changed or kept.

To check the data entered, run the following line of script:

```
// Output the data:
type In %(event$), the %(measurement$) was $(nn) %(units$);
```

This next example script assumes a Graph is the active window and prompts for information then draws a line and labels it. The call to **GetN** uses string registers and pre-defined lists as inputs.

```
%A=Minimum;
iColor = 15;
dVal = 2.75;
iStyle = 2;

// Opens the GetN dialog ...
// The extra %-sign in front of %A interprets the string register
// literally, instead of treating it as a variable name.
getn (Label Quantile) %%A
```

```
(Color) iColor:@C
(Style) iStyle:@D
(Value) dVal
(Set Quantile);

draw -n %A -l -h dVal;     // Draws a horzontal, named line
%A.color = iColor;         // Sets the line color
%A.linetype = iStyle;      // Sets the line style

// Creates a text label named QLabel at the right end of the
// line
label -s -a x2 dVal -n QLabel %A;

%A.Connect(QLabel,1);       // Connects the two objects
```

Note : The script requires that %A should be a single word and that object *QLabel* does not exist.

The following character sequences, beginning with the **@** character, access pre-defined lists for **GetN** arguments:

List	Description
@B	List of Object Background attributes
@C	Basic Color List
@D	Line Style List
@P	Pattern List
@S	Font Size List
@T	Font List
@W	Line Width List
@Z	Symbol Size List

Note that the value returned when a list item is selected within the **GetN** dialog is the index of the item in the list. For instance, if one of your **GetN** entries is:

```
(Font Size) fs:@S
```

and you select **18** from the drop-down list in the dialog, the variable **fs** will hold the value **8**, since 18 is the 8th item in the list.

Below is another example script that allows a user to change a Symbol Plot to a Line + Symbol Plot or the reverse:

```
get %C -z iSymbolSize;  // Get current Symbol Size
get %C -cl iLineColor;  // Get current Line color
iUseLine = 0;
// Now open the dialog to the user
getn (Symbol Size) iSymbolSize
     (Use Line) iUseLine:2s
     (Line Color) iLineColor:@C
     (Set Plot Style);
// If User asked for Line
if(iUseLine == 1)
{
    set %C -l 1;           // Turn on the line
    set %C -cl iLineColor; // Set the line color
}
// .. if not
else
    set %C -l 0;           // Turn off line
set %C -z iSymbolSize;     // Set Symbol size
```

15.2 Getting Points from Graph

Any of the Tools in the Origin **Tools** Toolbar can be initiated from script, but three can be linked to macros and programmed to do more.

To program tools, define the **pointproc** macro to execute appropriate code. The **pointproc** macro runs when the user double-clicks or single-clicks and presses the Enter key.

15.2.1 Screen Reader

This script puts a label on a graph using the Screen Reader tool.

```
dotool 2;  // Start the Screen Reader Tool
dotool -d; // Allow a single click to act as double click
// Here we define our '''pointproc''' macro
def pointproc {
    label -a x y -n MyLabel Hello;
    dotool 0;  // Reset the tool to Pointer
```

```
    done = 1;  // Set the variable to allow infinite loop to end
}
// Script does not stop when using a tool,
// so further execution needs to be prevented.
// This infinite loop waits for the user to select the point
for( done = 0 ; done == 0; ) sec -p .1;
// A .1 second delay gives our loop something to do:
type Continuing script ...;
// Once the macro has run, our infinite loop is released
```

15.2.2 Data Reader

The Data Reader tool is similar to the Screen Reader, but the cursor locks on to actual data points. If defined, a **quittoolbox** macro runs if user presses **Esc** key or clicks the Pointer Tool to stop the Data Reader.

This example assumes a graph window is active and expects the user to select three points on their graph.

```
@global = 1;
dataset dsx, dsy;  // Create two datasets to hold the X and Y values
dotool 3;          // Start the tool
// Define the macro that runs for each point selection
def pointproc {
    dsx[count] = x;  // Get the X coordinate
    dsy[count] = y;  // Get the Y coordinate
    count++;         // Increment count
    if(count == 4) dotool 0;  // Check to see if we have three points
    else type -a Select next point;
}
// Define a macro that runs if user presses Esc key,
// or clicks the Pointer Tool:
def quittoolbox {
    // Error : Not enough points
    if(count < 4) ty -b You did not specify three datapoints;
    else
    {
        draw -l {dsx[1],dsy[1],dsx[2],dsy[2]};
        draw -l {dsx[2],dsy[2],dsx[3],dsy[3]};
        draw -l {dsx[3],dsy[3],dsx[1],dsy[1]};
        double ds12 = dsx[1]*dsy[2] - dsy[1]*dsx[2];
        double ds13 = dsy[1]*dsx[3] - dsx[1]*dsy[3];
        double ds23 = dsy[3]*dsx[2] - dsy[2]*dsx[3];
        area = abs(.5*(ds12 + ds13 + ds23));
        type -b Area is $(area);
    }
```

```
}
count = 1;  // Initial point
type DoubleClick your first point (or SingleClick and press Enter);
```

The following example allows user to select arbitrary number of points until Esc key is pressed or user clicks on the Pointer tool in the Tools toolbar.

```
@global = 1;
dataset dsx, dsy;  // Create two datasets to hold the X and Y values
dotool 3;          // Start the tool
// Define the macro that runs for each point selection
def pointproc {
    count++;         // Increment count
    dsx[count] = x;  // Get the X coordinate
    dsy[count] = y;  // Get the Y coordinate
}

// Define a macro that runs if user presses Esc key,
// or clicks the Pointer Tool:
def quittoolbox {
   count=;
   for(int ii=1; ii<=count; ii++)
   {
      type $(ii), $(dsx[ii]), $(dsy[ii]);
   }
}
count = 0;  // Initial point
type "Click to select point, then press Enter";
type "Press Esc or click on Pointer tool to stop";
```

Pressing Enter key to select a point works more reliably than double-clicking on the point.

You can also use the **getpts** command to gather data values from a graph.

15.2.3 Data Selector

The Data Selector tool is used to set a Range for a dataset. A range is defined by a beginning row number (index) and an ending row. You can define multiple ranges in a dataset and Origin analysis routines will use these ranges as input, excluding data outside these ranges.

Here is a script that lets the user select a range on a graph.

```
    // Start the tool
    dotool 4;
    // Define macro that runs when user is done
```

```
    def pointproc {
        done = 1;
        dotool 0;
    }
    // Wait in a loop for user to finish by pressing ...
    // (1) Enter key or (2) double-clicking
    for( done = 0 ; done == 0 ; )
    {
        sec -p .1;
    }
    // Additional script will run once user completes tool.
    ty continuing ..;
```

When using the Regional Data Selector or the Regional Mask Tool you can hook into the quittoolbox macro which triggers when a user presses Esc key:

```
    // Start the Regional Data Selector tool with a graph active
    dotool 17;
    // Define macro that runs when user is done
    def quittoolbox {
        done = 1;
    }
    // Wait in a loop for user to finish by pressing ...
    // (1) Esc key or (2) clicking Pointer tool:
    for( done = 0 ; done == 0 ; )
    {
        sec -p .1;
    }
    // Additional script will run once user completes tool.
    ty continuing ..;
```

And we can use an X-Function to find and use these ranges:

```
// Get the ranges into datasets
dataset dsB, dsE;
mks ob:=dsB oe:=dsE;
// For each range
for(idx = 1 ; idx <= dsB.GetSize() ; idx++ )
{
    // Get the integral under the curve for that range
    integ %C -b dsB[idx] -e dsE[idx];
    type Area of %C from $(dsB[idx]) to $(dsE[idx]) is $(integ.area);
}
```

List of Tools in Origin Tools Toolbar. Those in **bold** are useful in programming.

Tool Number	Description
0	Pointer - The Pointer is the default condition for the mouse and makes the mouse act as a selector.
1	ZoomIn - A rectangular selection on a graph will rescale the axes to the rectangle. (Graph only)
2	**Screen Reader** - Reads the location of a point on a page.
3	**Data Reader** - Reads the location of a data point on a graph. (Graph only)
4	**Data Selector** - Sets a pair of Data Markers indicating a data range. (Graph only)
5	Draw Data - Allows user the draw data points on a graph. (Graph only)
6	Text - Allows text annotation to be added to a page.
7	Arrow - Allows arrow annotation to be added to a page.
8	Curved Line - Allows curved line annotation to be added to a page.
9	Line - Allows line annotation to be added to a page.
10	Rectangle - Allows rectangle annotation to be added to a page.
11	Circle - Allows circle annotation to be added to a page.
12	Closed Polygon - Allows closed polygon annotation to be added to a page.
13	Open Polygon - Allows open polygon annotation to be added to a page.

14	Closed Region - Allows closed region annotation to be added to a page.
15	Open Region - Allows open region annotation to be added to a page.
16	ZoomOut - Zooms out (one level) when clicking anywhere in a graph. (Graph only)
17	**Regional Data Selector** - Allows selection of a data range. (Graph only)
18	**Regional Mask Tool** - Allows masking a points in a data range. (Graph only)

15.3 Bringing Up a Dialog

X-Functions whose names begin with **dlg** may be called in your scripts to facilitate dialog-based interaction.

Name	Brief Description
dlgChkList	Prompt to select from a list
dlgFile	Prompt with an Open File dialog
dlgPath	Prompt with an Open Path dialog
dlgRowColGoto	Go to specified row and column
dlgSave	Prompt with a Save As dialog
dlgTheme	Select a theme from a dialog

Possibly the most common such operation is to select a file from a directory. The following line of script brings up a dialog that pre-selects the PDF file extension (group), and starts at the given path location (init):

```
dlgfile group:=*.pdf init:="C:\MyData\MyPdfFiles";
type %(fname$);  // Outputs the selected file path to Script Window
```

The complete filename of the file selected in the dialog, including path, is stored in the variable **fname$**. If **init** is left out of the X-Function call or cannot be found, the dialog will start in the User Files folder.

The **dlgsave** X-Function works for saving a file using a dialog.

```
dlgsave ext:=*.ogs;
type %(fname$);  // Outputs the saved file path to Script Window
```

16 Working with Excel

Origin can use Excel Workbooks directly within the Origin Workspace. The Excel Workbooks can be stored within the project or linked to an external Excel file (*.xls, *.xlsx). An external Excel Workbook which was opened in Origin can be converted to internal, and a an Excel Workbook created within Origin can be saved to an external Excel file.

To create a new Excel Workbook within Origin ..

```
window -tx;
```

The titlebar will include the text **[Internal]** to indicate the Excel Workbook will be saved in the Origin Project file.

To open an external Excel file ..

```
document -append D:\Test1.xls;
```

The titlebar will include the file path and name to indicate the Excel file is saved external to the Origin Project file.

You can save an internal Excel Workbook as an external file at which point it becomes a linked external file ..

```
// The Excel window must be active. win -o can temporarily make it active
window -o Book5 {
    // You must include the file path and the .xls extension
    save -i D:\Test2.xls;
}
```

You can re-save an external Excel Workbook to a new location creating a new file and link ..

```
// Assume the Excel Workbook is active
// %X holds the path of an opened Origin Project file
save -i %XNewBook.xls;
```

17 Automation and Batch Processing

This chapter demonstrates using LabTalk script to automate analysis in Origin by creating Analysis Templates, and using these templates to perform batch processing of your data:

17.1 Analysis Templates

Analysis Templates are pre-configured workbooks which can contain multiple sheets including data sheets, report sheets from analysis operations, and optional custom report sheets for presenting results. The analysis operations can be set to recalculate on data change, thus allowing repeat use of the analysis template for batch processing or manual processing of multiple data files.

The following script example opens a built-in Analysis Template, *Dose Response Analysis.ogw*, and imports a data file into the data sheet. The results are automatically updated based on the new data.

```
string fPath$ = system.path.program$ + "Samples\Curve Fitting\";
string fname$ = fPath$ + "Dose Response Analysis.ogw";
// Append/open the analsys template to current project
doc -a %(fname$);
string bn$ = %H;
win -o bn$ {
   // Import no inhibitor data
   fname$ = fPath$ + "Dose Response - No Inhibitor.dat";
   impASC options.Names.FNameToSht:=0
        options.Names.FNameToBk:=0
        options.Names.FNameToBkComm:=0
        orng:=[bn$]"Dose Response - No Inhibitor";
   // Import inhibitor data
   fname$ = fPath$ + "Dose Response - Inhibitor.dat";
   impASC options.Names.FNameToSht:=0
        options.Names.FNameToBk:=0
        options.Names.FNameToBkComm:=0
        orng:=[bn$]"Dose Response - Inhibitor";
   // Active the result worksheet
   page.active$ = result;
}
```

To learn how to create Analysis Templates, please refer to the Origin tutorial: Creating and Using Analysis Templates.

17.2 Using Set Column Values to Create an Analysis Template

Many analysis tools in Origin provide a Recalculate option, allowing for results to update when source data is modified, such as when importing new data to replace existing data. A workbook containing such operations can be saved as an **Analysis Template** for repeated use with **Batch Processing**.

The **Set Column Values** feature can also be used to create such Analysis Templates when custom script is needed for your analysis.

In order to create Analysis Templates using the Set Column Values feature, the following steps are recommended:

1. Set up your data sheet, such as importing a representative data file.
2. Add an extra column to the data sheet, or to a new sheet in the same workbook.
3. Open the Set Column Values dialog from this newly added column.
4. Enter the desired analysis script in the Before Formula Scripts panel. Note that your script can call X-Functions to perform multiple operations on the data.
5. In you script, make sure to reference at least one column or cell of your data sheet that will get replaced with new data. You can do this by defining a range variable that points to a data column and then use that range variable in your script for computing your custom analysis output.
6. Set the **Recalculate** drop-down in the dialog to either *Manual* or *Auto*, and press OK.
7. Use the **File: Save Workbook as Analysis Template...** menu item to save the Analysis Template.

For an example on setting up such a template using script, please refer to the Origin tutorial: Creating Analysis Templates using Set Column Value.

17.3 Batch Processing

One may often encounter the need to perform batch processing of multiple sets of data files or datasets in Origin, repeating the same analysis procedure on each set of data. This can be achieved in three different ways, and the following sections provide information and examples of each.

17.3.1 Processing Each Dataset in a Loop

One way to achieve batch processing is to loop over multiple files or datasets, and within the loop process each dataset by calling appropriate X-Functions and other script commands to perform the necessary data processing.

The following example shows how to import 10 files and perform a curve fit operation and print out the fitting results:

```
// Find all files using wild card
string path$ = system.path.program$ + "Samples\Batch Processing";  // Path to find
files
// Find the files in the folder specified by path$ variable (default)
// The result file names are stored in the string variable fname$
// Separated by CRLF (default). Here wild card * is used, which means
// all files start with "T", and with the extension "csv"
findFiles ext:="T*.csv";

// Start a new book with no sheets
newbook sheet:=0;
// Loop over all files
for(int iFile = 1; iFile <= fname.GetNumTokens(CRLF); iFile++)
{
   // Get file name
   string file$ = fname.GetToken(iFile, CRLF)$;
   // Import file into a new sheet
   newsheet;
   impasc file$;
   // Perform gaussian fitting to col 2 of the current data
   nlbegin iy:=2 func:=gaussamp nltree:=myfitresult;
   // Just fit and end with no report
   nlfit;
   nlend;
   // Print out file name and results
   type "File Name: %(file$)";
   type "   Peak Center= $(myfitresult.xc)";
   type "   Peak Height= $(myfitresult.A)";
   type "   Peak Width=  $(myfitresult.w)";
}
```

17.3.2 Using Analysis Template in a Loop

Custom templates for analysis can be created in Origin by performing the necessary data processing from the GUI on a representative dataset and then saving the workbook, or the entire project, as an **Analysis Template**. The following example shows how to make use of an existing analysis template to perform curve fitting on 10 files:

```
// Find all files using wild card
string fpath$ = "Samples\Batch Processing\";
string path$ = system.path.program$ + fpath$;  // Path to find files
// Find the files in the folder specified by path$ variable (default)
// The result file names are stored in the string variable fname$
```

```
// Separated by CRLF (default). Here wild card * is used, which means
// all files start with "T", and with the extension "csv"
findFiles ext:="T*.csv";

// Set path of Analysis Template
string templ$ = path$ + "Peak Analysis.OGW";
// Loop over all files
for(int iFile = 1; iFile <= fname.GetNumTokens(CRLF); iFile++)
{
   // Open an instance of the analysis template
   doc -a %(templ$);
   // Import current file into first sheet
   page.active = 1;
   impasc fname.GetToken(iFile, CRLF)$
}

// Issue a command to update all pending operations
// in case the operations were set to manual recalculate in the template
run -p au;
```

17.3.3 Using Batch Processing X-Functions

Origin provides script-accessible X-Functions to perform batch processing, where there is no need to loop over files or datsets. One simply creates a list of desired data to be processed and calls the relevant X-Function. The X-Function then either uses a template or a theme to process all of the specified data. Some of these X-Functions can also create an optional summary report that contains results from each file/dataset that were marked for reporting by the user, in their custom analysis template.

The table below lists X-Functions available for batch analysis:

Name	Brief Description
batchProcess	Perform batch processing of multiple files or datasets using Analysis Template, with optional summary report sheet
paMultiY	Perform peak analysis of multiple Y datasets using Peak Analyzer theme

The following script shows how to use the batchProcess X-Function to perform curve fitting of data from 10 files using an analysis template, with a summary report created at the end of the process.

```
// Find all files using wild card
string path$ = system.path.program$ + "Samples\Batch Processing\";  // Path to find
files
```

```
// Find the files in the folder specified by path$ variable (default)
// The result file names are stored in the string variable fname$
// Separated by CRLF (default). Here wild card * is used, which means
// all files start with "T", and with the extension "csv"
findFiles ext:="T*.csv";

// Set path of Analysis Template
string templ$ = path$ + "Peak Analysis.OGW";

// Call the Batch Processing X-Function
// Keep only the final summary sheet, delete intermediate books
batchProcess batch:=1 name:=templ$ data:=0 fill:="Raw Data"
             append:="Summary" remove:=1 method:=impASC;
```

Batch processing using X-Functions can also be performed by calling Origin from an external console; for more see Running Scripts From Console.

18 Function Reference

This section provides reference lists of functions, X-Functions and Origin C Functions that are supported in LabTalk scripting:

18.1 LabTalk-Supported Functions

Below is a tabular listing of functions supported by the LabTalk scripting language, broken down by category.

An Alphabetical Listing of All LabTalk-Supported Functions is also available (CHM and Wiki only!).

Key to Function Arguments

The datatypes of the arguments in the function tables are given by the following naming convention:

Name	Datatype
ds	dataset
m or **n**	integer
str$	string
v	vector

An argument with any other name is a numeric of type **double**.

Multiple arguments of type **double** will be given different names, as in **Histogram(ds, inc, min, max)**, or numbered, as in **Cov(ds1, ds2, ave1, ave2)**.

Multiple arguments of a datatype other than double will be numbered, as in **Corr(ds1, ds2)**.

18.1.1 Statistical Functions

General Statistics

Name	Brief Description
Ave(ds, n)	Breaks dataset into groups of size *n*, finds the average for each group, and returns a range containing these values.
Count(v [,n])	Counts elements in a vector *v*; *n* is an integer parameter specifying different options.
Cov(ds1, ds2, ave1, ave2)	Returns the covariance between two datasets, where *ave1* and *ave2* are the respective means of datasets *ds1* and *ds2*.
Diff(ds)	Returns a dataset that contains the difference between adjacent elements in *dataset*.
Histogram(ds, inc, min, max)	Generates data bins from *dataset* in the specified range from *min* to *max*.
Max(v)	This function returns the maximum value from a set of values.
Mean(v)	Returns the average of a vector.
Median(v [,n])	This function is used to return median of vector *v*, with parameter *n* specifying the type of interpolation.
Min(v)	This function is used to return the minimum value from a vector *v*.
Percentile(ds1, ds2)	Returns a range comprised of the percentile values for *ds1* at each percent value specified in *ds2*.
QCD2(n)	Returns Quality Control D2 Factor
QCD3(n)	Returns Quality Control D3 Factor
QCD4(n)	Returns Quality Control D4 Factor

Ss(ds [,ref])	Returns the sum of the squares of dataset *ds*. The optional *ref* defaults to the mean of *ds* as the reference value.
StdDev(v)	Calculates the standard deviation based on a sample.
StdDevP(v)	Return the standard deviation based on the entire population given as arguments.
Sum(ds)	Returns a range whose ***i*** th element is the sum of the first ***i*** elements of the dataset dataset.
Total(v)	Returns the sum of a vector.

Cumulative Distribution Functions

Name	Brief Description
Betacdf(x,a,b)	Computes beta cumulative distribution function at x, with parameters a and b.
Erf(x)	The error function (or normal error integral).
InvF(value, m, n)	The inverse F distribution function with m and n degrees of freedom.
Ncchi2cdf(x,f,lambda)	Computes the probability associated with the lower tail of the non-central χ^2 distribution.
Poisscdf(n,rlamda)	Computes the lower tail probabilities in given value k, associated with a Poisson distribution using the corresponding parameters in λ.
Binocdf(m,n,p)	Computes the lower tail, upper tail and point probabilities in given value k, associated with a Binomial distribution using the corresponding parameters in n, p.
Fcdf(f,df1,df2)	Computes F cumulative distribution function at x, with parameters a and b, and lower tail.
Invprob(x)	The Inverse Probability Density function.
Ncfcdf(f,df1,df2,lambda)	Computes the probability associated with the lower tail of the

	non-central F or variance-ratio distribution.
Srangecdf(q,v,n)	Computes the probability associated with the lower tail of the distribution of the Studentized range statistic.
Bivarnormcdf(x,y,rho)	Computes the lower tail probability for the bivariate Normal distribution.
Gamcdf(g,a,b)	Computes the lower tail probability for the gamma distribution with real degrees of freedom, with parameters α and β
Invt(value, n)	The inverse t distribution function with n degrees of freedom.
Nctcdf(t,df,delta)	Computes the lower tail probability for the non-central Student's t-distribution.
Tcdf(t,df)	Computes the cumulative distribution function of Student's t-distribution.
Chi2cdf(x,df)	Computes the lower tail probability for the χ^2 distribution with real degrees of freedom.
Hygecdf(m1, m2, n1, n2)	Computes the lower tail probabilities in a given value, associated with a hypergeometric distribution using the corresponding parameters.

Inverse Cumulative Distribution Functions

Name	Brief Description
Chi2inv(p,df)	Computes the inverse of the χ^2 cdf for the corresponding probabilities in X with parameters specified by ν.
Ftable(x, m, n)	The F distribution function with m and n degrees of freedom.
Finv(p,df1,df2)	Computes the inverse of F cdf at x, with parameters ν_1 and ν_2
Gaminv(p,a,b)	Computes the inverse of Gamma cdf at g_p , with parameters a and b.
IncF(x, m, n)	The incomplete F-table function.

Inverf(x)	Computes inverse error function fnction at x.
Norminv(p)	Computes the deviate, x_p, associated with the given lower tail probabilip, p, of the standardized normal distribution.
Srangeinv(p,v,n)	Computes the deviate, x_p, associated with the lower tail probability of the distribution of the Studentized range statistic.
Ttable(x, n)	The Student's t distribution with n degrees of freedom.
Tinv(p,df)	Computes the deviate associated with the lower tail probability of Student's t-distribution with real degrees of freedom.
Wblinv(p,a,b)	Computes the inverse Weibull cumulative distribution function for the given probability using the parameters a and b.
Betainv(p,a,b)	Returns the inverse of the cumulative distribution function for a specified beta distribution.

Probability Density Functions

Name	Brief Description
Betapdf(x,a,b)	Returns the probability density function of the beta distribution with parameters a and b.
Wblpdf(x,a,b) (8.6 SR0)	Returns the probability density function of the Weibull distribution with parameters a and b.
lognpdf(x,mu,sigma) (8.6 SR0)	Returns values at X of the lognormal pdf with distribution parameters mu and sigma.
normpdf(x,mu,sigma) (8.6 SR0)	computes the pdf at each of the values in X using the normal distribution with mean mu and standard deviation sigma.
poisspdf(x,lambda) (8.6 SR0)	computes the Poisson pdf at each of the values in X using mean parameters in lambda.

exppdf(x,lambda) (8.6 SR0)	returns the pdf of the exponential distribution with mean parameter lambda, evaluated at the values in X.
lappdf(x,a,b) (8.6 SR0)	Laplace probability density function
cauchypdf(x,a,b) (8.6 SR0)	Cauchy probability density function (also called Lorentz distribution)
gampdf(x,a,b) (8.6 SR0)	Returns the Gamma probability density with parameters a and b.

18.1.2 Mathematical Functions

Basic Mathematics

Name	Brief Description
Abs(x)	Returns the absolute value of x
Acos(x)	Returns the inverse of the corresponding trigonometric function.
Acot(x)	Returns the inverse of the corresponding trigonometric function.
Acoth(x)	Returns the inverse hyperbolic cotangent.
Acsc(x)	Returns the inverse of the corresponding trigonometric function csc(x)=1/sin(x).
Acsch(x)	Returns the inverse hyperbolic cosecant.
Asec(x)	Returns the inverse of the corresponding trigonometric function sec(x)=1/cos(x).
Asech(x)	Returns the inverse hyperbolic secant.
Angle(x, y)	Returns the angle in radians measured between the positive X axis and the line joining the origin (0,0) with the point given by (x, y).

Asin(x)	Returns the inverse of the corresponding trigonometric function.
Atan(x)	Returns the inverse of the corresponding trigonometric function.
Asinh(x)	Returns the inverse hyperbolic sine.
Acosh(x)	Return the inverse hyperbolic cosin.
Atanh(x)	Return the inverse hyperbolic tangent.
Cos(x)	Return value of cosine for each value of the given x.
Cosh(x)	Returns the hyperbolic form of cos(x) .
Cot(x)	Returns value of cotangent for each value of the given x.
Coth(x)	Returns value of the hyperbolic cotangent of x.
Csc(x)	Returns value of cosecant for each value of the given x.
Csch(x)	Returns value of hyperbolic cosecant of x.
Degrees(angle)	Converts the radians into degrees.
Derivative(vd[,n])	Returns the derivative of the data list in a given vector.
Exp(x)	Returns the exponential value of x.
Int(x)	Return the truncated integer of x.
Ln(x)	Return the natural logarithm value of x.
Log(x)	Return the base 10 logarithm value of x.
Mod(n, m)	Return the integer modulus (the remainder from division) of integer x divided by integer y.
Nint(x)	Return value of the nint(x) function is identical to round(x, 0).
Prec(x, n)	Returns the input value *x* to *n* significant figures.
Rmod(x, y)	Returns the real modulus (the remainder from division) of double x divided by double y.

Round(x, n)	Returns the value (or dataset) *x* to *n* decimal places.
Secant(x)	Returns value of secant for each value of the given x.
Sech(x)	Returns hyperbolic secant of x.
Sign(x)	Returns the sign of real number x.
Sin(x)	Returns value of sine for each value of the given x.
Sinh(x)	Returns the hyperbolic form of sin(x).
Sqrt(x)	Returns the square root of x.
Tan(x)	Returns value of tangent for each value of the given x.
Tanh(x)	Returns the hyperbolic form of and tan(x).
Radians(angle)	Returns radians given input *angle* in degrees.
Distance(x1, y1, x2, y2)	Returns the distance with two points.
Distance3D(x1, y1, z1, x2, y2, z2)	Returns the distance with two points in 3D.
Angleint1(x1, y1, x2, y2 [, n, m])	Returns the angle between a line with endpoints (x1, y1) and (x2, y2) and the X axis. Returns degrees if *n*=1 or radians if *n*=0, default is radians. Constrains the returned angle value to the first (+x,+y) and fourth (+x,-y) quadrant if *m*=0. If *m*=1, returns values from 0–2pi radians or 0–360 degrees.
Angleint2(x1, y1, x2, y2, x3, y3, x4, y4 [, n, m])	Returns the angle between two lines with endpoints (x1, y1) and (x2, y2) for one line and (x3, y3) and (x4, y4) for the other. Returns degrees if *n*=1 or radians if *n*=0, default is radians. Constrains the returned angle value to the first (+x,+y) and fourth (+x,-y) quadrant if *m*=0. If *m*=1, returns values from 0–2pi radians or 0–360 degrees.

Multi-parameter Functions

Multi-parameter functions are used as built-in functions for Origin's nonlinear fitter. You can view the equation, a sample curve, and the function details for each multi-parameter function by opening the NLFit (Analysis:Fitting:Nonlinear Curve Fit). Then select the function of interest from the Function selection page.

For additional documentation on all the multi-parameter functions available from Origin's nonlinear curve fit, see this PDF on the OriginLab website. This document includes the mathematical description, a sample curve, a discussion of the parameters, and the LabTalk function syntax for each multi-parameter function.

Name	Brief Description
Boltzmann(x, A1, A2, x0, dx)	Boltzmann Function
Dhyperbl(x, P1, P2, P3, P4, P5)	Double Rectangular Hyperbola Function
ExpAssoc(x, y0, A1, t1, A2, t2)	Exponential Associate Function
ExpDecay2(x, y0, x0, A1, t1, A2, t2)	Exponential Decay 2 with Offset Function
ExpGrow2(x, y0, x0, A1, t1, A2, t2)	Exponential Growth 2 with Offset Function
Gauss(x, y0, xc, w, A)	Gaussian Function
Hyperbl(x, P1, P2)	Hyperbola Function
Logistic(x, A1, A2, x0, p)	Logistic Dose Response Function
Lorentz(x, y0, xc, w, A)	Lorentzian Function
Poly(x, a0, a1, a2, a3, a4, a5, a6, a7, a8, a9)	Polynomial Function
Pulse(x, y0, x0, A, t1, P, t2)	Pulse Function

Random Number Generators

Two functions in this category, **rnd() and ran()** and **grnd()**, return a value. All the other functions in this category return a range.

Name	Brief Description
grnd()	Returns a value from a normally (Gaussian) distributed sample,with zero mean and unit standard deviation.
normal(npts, seed)	Returns a range with **npts** number of values.
Poisson(n, mean [, seed])	Returns *n* random integers having a Poisson distribution with mean *mean*. Optional *seed* provides a seed for the number generator.
rnd() and ran()	Return a value between 0 and 1 from a uniformly distributed sample.
uniform(npts, seed)	Returns a range with **npts** number of values.

Bessel, Beta, and Gamma Functions

Bessel Functions

Name	Brief Description
Jn(x, n)	Bessel function of order *n*
Yn(x, n)	Bessel Function of Second Kind
J1(x)	First Order Bessel Function
Y1(x)	First order Bessel function of second kind has the following form: Y1(x)
J0(x)	Zero Order Bessel Function
Y0(x)	Zero Order Bessel Function of Second Kind

Beta Functions

Name	Brief Description
beta(a, b)	Beta Function with parameters *a* and *b*

incbeta(x, a, b)	Incomplete Beta Function with parameters *x*, *a*, *b*

Gamma Functions

Name	Brief Description
incomplete_gamma(a, x)	Incomplete gamma functions
gammaln(x)	Natural Log of the Gamma Function
Incgamma	

Approximations of NAG Functions

Name	Brief Description
bessel_i_nu(x,n)	Evaluates an approximation to the modified Bessel function of the first kind Iν/4 (x)
bessel_i_nu_scaled(x,n)	Evaluates an approximation to the modified Bessel function of the first kind $e^{-x}I_{\frac{\nu}{4}}(x)$
bessel_i0(x)	Evaluates an approximation to the modified Bessel function of the first kind, I0(x).
bessel_i0_scaled(x)	Evaluates an approximation to $e^{-\lvert x\rvert}I_0(x)$
bessel_i1(x)	Evaluates an approximation to the modified Bessel function of the first kind,$I_1(x)$.
bessel_i1_scaled(x)	Evaluates an approximation to $e^{-\lvert x\rvert}I_1(x)$
bessel_j0(x)	Evaluates the Bessel function of the first kind,$J_0(x)$
bessel_j1(x)	Evaluates an approximation to the Bessel function of the first kind $J_1(x)$
bessel_k_nu(x,n)	Evaluates an approximation to the modified Bessel function of the second kind $K_{\upsilon/4}(x)$

bessel_k_nu_scaled(x,n)	Evaluates an approximation to the modified Bessel function of the second kind $e^{-x}K_{\upsilon/4}(x)$
bessel_k0(x)	Evaluates an approximation to the modified Bessel function of the second kind, $K_0(x)$
bessel_k0_scaled(x)	Evaluates an approximation to $e^{x}K_0(x)$
Bessel_k1(x)	Evaluates an approximation to the modified Bessel function of the second kind, $K_1(x)$
bessel_k1_scaled(x)	Evaluates an approximation to $e^{x}K_1(x)$
Gamma(x)	Evaluates $\Gamma(x) = \int_0^\infty t^{x-1}e^{-t}dt$

18.1.3 Origin Worksheet and Dataset Functions

Worksheet Functions

Name	Brief Description
Cell(n,m)	Gets or sets values in the active worksheet or matrix. Indicate the row number *n* and column number *m* in parentheses.
Col(ds)	Refers to the dataset in a worksheet column, to a cell in the column, or to the column headers.
Wcol(ds)	Can be used either on the left side or on the right side of an assignment.

Dataset Information Functions

Name	Brief Description
Errof(ds)	Returns the dataset (error column) containing the error values of *dataset*.

Findmasks(ds)	Returns a dataset that contains the indexes of the masked data points.
hasx(ds)	Returns 1 if *dataset* is plotted against an X dataset in the active layer. If not, this function returns 0.
Index(d,vd[,n])	Returns the index of x, which is controlled by ctrl, in a strictly monotonic dataset.
IsMasked(n, ds)	Returns the number of masked points in *dataset* if *index* = 0.
List(val, ds)	Returns the index number in dataset *ds* where value *val* first occurs.
Xindex(x, ds)	Returns the index number of the cell in the X dataset associated with *dataset*, where the cell value is closest to x.
Xof(ds)	Returns a string containing the X values of *dataset*.
Xvalue(n, ds)	Returns the corresponding X value for *dataset* at row number *i* in the active worksheet.

Dataset Manipulation Functions

Name	Brief Description
asc(str$)	Returns the ASCII value of the uppercase character in parentheses.
corr(ds1, ds2, k [,n])	Returns the correlation between two datasets using a lag size *k* and an optional number of points *n*.
peaks(ds, width, minht)	Returns a dataset containing indices of peaks found using *width* and *minHt* as a criteria.
sort(ds)	Returns a dataset that contains *dataset*, sorted in ascending order.

treplace(ds, val1, val2 [, cnd)]	Returns a dataset. Each value in dataset is compared to *value1* according to the condition *cnd*.

Dataset Generation Functions

Name	Brief Description
Data(x1, x2, inc)	Create a dataset with values ranging from *x1–x2* with an increment, *inc*.
{v1, v2, ...vn}, {v1:vn}, {v1:vstep:vn}	Create a dataset of either discrete values, a range from *v1–vn* with an implied increment equal to 1, or a range from *v1–vn* with an increment equal to *vstep*.
Fit(Xds,n)	Create a dataset based on a fit of the data in *Xdataset*. If more than one fit curve was produced in the last fitting session, *n* indicates the index of the dataset to use (default = 1).
Table(ds1, ds2, ds3)	

String and Character Functions

Note: All of the following functions are available only in the Origin 8 SR6 or later version!

Name	Brief Description
Char(n)	Return the character specified by the code number.
Code(str$)	Return a numeric code for the first character in input string.
Compare(str1$, str2$ [, n])	Compare str1 with str2, identical will return 1.
Exact(str1$, str2$)	Return TRUE if both strings are an exact match (case and length).

Find(str1$, str2$ [, n])	Finds a string (**str2**) within another string (**str1**) starting from the specific positoin (**StartPos**), and returns the starting position of **str2** in **str1**.
Format(data, str$)	Convert double to string with LabTalk formatting option.
Left(str$, n)	Returns the leftmost **n** characters from the string.
Len(str$)	Returns the number of characters of a string (**str**).
Lower(str$)	Converts the string to lowercase.
MakeCSV(str1$[,n1,n2,str2$])$	Converts a string which has an identical delimiter into CSV format.
MatchBegin(str1$, str2$ [, n, m])	Finds a string pattern (**str2**) within another string **str1** starting from the specified positoin **StartPos**, and returns the starting position of **str2** in **str1**.
MatchEnd(str1$, str2$ [, n, m)]	Finds a string pattern (**str2**) within another string **str1** from the specified positoin **StartPos**, and returns the ending position of **str2** in **str1**.
Mid(str$, n1, n2)	Returns a specific number of characters (**n2**) from the string (**str**), starting at the specific position (**n1**).
Replace(str1$, n1, n2, str2$)	Replace **n2** characters in string1 starting at **n1**th position with string2.
Right(str$, n)	Returns the rightmost **n** characters from the string.
Search(str1$, str2$ [, n])	Finds a string (**str2**) within another string (**str1**) starting from the specific positoin (**StartPos**), and returns the starting position of **str2** in **str1**.
Substitute(str1$, str2$, str3$ [,n])	Substitute string3 with string2 when found in string1.

Token(str$,n1[,n2])$	Get the Nth token using specified delimiter from a string.
Trim(str$, n)	Removes spaces from string.
Upper(str$)	Converts the string to uppercase.

Date and Time Functions

Name	Brief Description
WeekDay(d, n)	Returns the day of the week according to calculate a date. By default, the day is ranging from 0 (Sunday) to 6 (Saturday).
WeekNum(d, n)	Return a number that indicates the calendar week number of the year.
Year(d)	Return the year as an integer in the range 0100-9999.
Month(d)	Return the month as an integer from 1 (January) to 12 (December).
MonthName(d, n)	Returns the Month name for specified month by index of 1 to 12, or as a Date value.
YearName(d, n)	Returns the year in string form with input of year or date, with option n.
Day(d, n)	Returns the day number of a given date.
Hour(d), Hour(t)	Returns the hour as an integer, ranging from 0 (12:00 A.M.) to 23 (11:00 P.M.).
Minute(d), Minute(t)	Returns the minutes as an integer, ranging from 0 to 59.
Second(d), Second(t)	Returns the seconds as a real value in the range 0 (zero) to 59.9999...
Now()	Returns the current date-time as a date(Julian days) value.
Today()	Returns the current date as a date(Julian days) value.

Quarter(date)	Returns current quarter of time.
Date(MM/dd/yy HH:mm,[format])	Returns the Julian-day value which Origin uses internally to represent dates.
Time(n1, n2, n3)	Returns the **Julian-day value** which Origin uses internally to represent time.
WeekDayName(d[,n1,n2])$	Returns the name of the weekday according to index of the day of the week or date in Julian format.

Utility Functions

Name	Brief Description
BitAND(n, m)	Returns bitwise AND operation of two intergers.
BitOR(n, m)	Returns bitwise OR operation of two intergers.
BitXOR(n, m)	Returns bitwise XOR operation of two intergers.
colnum(colNameOrDs)	Returns the column position number of the column specified by colName.
color(name)	Returns a number corresponding to the index in the color list of the color specified by the name or by the RGB value.
color(name, 0)	Similar to color(name) which returns a number corresponding to the *zero-based index* in the color list.
color(R, G, B)	Returns a integer color value. This value stores additional info in the highest byte. R, G, and B correspond to Red, Green, and Blue in RGB color scheme, and each component value ranges from 0 to 255.
exist(name)	Returns a single value indicating what 'object type' the given name is associated with string value.
font(name)	Returns a number corresponding to the font list index of the font specified by name.

hex(str$)	Returns the base 10 equivalent to the hexadecimal value represented by the given string.
ISNA(dd)	Determines whether the number is a NANUM.
NA()	Returns NANUM.
nlf_*name*(ds, p1, p2, ..., p*n*)	Returns Y values using the user-defined fitting function *name*, using the dataset **ds** as X values, and the parameters **p1-p*n***.
xf_get_last_error_code()	Get the last error code value of XFunction engine.
xf_get_last_error_message()$	Get the last error string message of XFunction engine.

18.1.4 Notes on Use

Each function returns either a single value or a range of values (a dataset), depending on the type of function and the arguments supplied. Unless otherwise specified, all functions will return a range if the first argument passed to the function is a range, and all functions will return a value if a value is passed.

18.2 LabTalk-Supported X-Functions

Below are several X-Functions, arranged by category, that are used frequently in LabTalk script.

This is not a complete list of X-Functions in Origin, but only those supported by LabTalk! For a complete listing of *all* X-Functions, arranged by category and alphabetical, see the X-Function Reference.

18.2.1 Data Exploration

Name	Brief Description
addtool_curve_deriv	Place a rectangle on the plot to perform differentiation
addtool_curve_fft	Add a rectangle onto the plot to perform FFT

addtool_curve_integ	Attach a rectangle on the plot to perform integration
addtool_curve_interp	Place a rectangle on the plot to perform interpolation
addtool_curve_stats	Place a rectangle onto the plot to calculate basic statistics
addtool_quickfit	Place a rectangle onto the plot to do fitting
addtool_region_stats	Region Statistics:Place a rectangle or circle onto the plot to calculate basic statistics
dlgRowColGoto	Go to specified row and column
imageprofile	Open the Image Profile dialog.
vinc	Calculate the average increment in a vector
vinc_check	Calculate the average increment in a vector

18.2.2 Data Manipulation

Name	Brief Description
addsheet	Set up data format and fitting function for Assays Template
assays	Assays Template Configuration:Set up data format and fitting function for Assays Template
copydata	Copy numeric data
cxt	Shift the x values of the active curve with different mode
levelcrossing	Get x coordinate crossing the given level
m2v	Convert a matrix to a vector
map2c	Combine an amplitude matrix and a phase matrix to a complex matrix.
mc2ap	Convert complex numbers in a matrix to amplitudes and phases.
mc2ri	Convert complex numbers in a matrix to their real and imaginary parts.

mcopy	Copy a matrix
mks	Get data markers in data plot
mo2s	Convert a matrix layer with multiple matrix objects to a matrix page with multiple matrix layers.
mri2c	Combine real numbers in two matrices into a complex matrix.
ms2o	Merge (move) multiple matrix sheets into one single matrix sheet with multiple matrixobjects.
newbook	Create a new workbook or matrix book
newsheet	Create new worksheet.
rank	Decide whether data points are within specified ranges
reducedup	Reduce Duplicate X Data
reduce_ex	Average data points to reduce data size and make even spaced X
reducerows	Reduce every N points of data with basic statistics
reducexy	Reduce XY data by sub-group statistics according to X's distribution
subtract_line	Subtract the active plot from a straight line formed with two points picked on the graph page
subtract_ref	Subtract on one dataset with another
trimright	Remove missing values from the right end of Y columns
v2m	Convert a vector to matrix
vap2c	Combine amplitude vector and phase vector to form a complex vector.
vc2ap	Convert a complex vector into a vector for the amplitudes and a vector for the phases.
vc2ri	Convert complex numbers in a vector into their real parts and imaginary parts.

vfind	Find all vector elements whose values are equal to a specified value
vri2c	Construct a complex vector from the real parts and imaginary parts of the complex numbers
vshift	Shift a vector
xy_resample	Mesh within a given polygon to resample data.
xyz_resample	Resample XYZ data by meshing and gridding

Gridding

Name	Brief Description
m2w	Convert the Matrix data into a Worksheet
r2m	Convert a range of worksheet data directly into a matrix
w2m	Convert the worksheet data directly into a matrix, whose coordinates can be specified by first column/row and row labels in the worksheet.
wexpand2m	Convert Worksheet to Matrix by expand for columns or rows
XYZ2Mat	Convert XYZ worksheet data into matrix
xyz_regular	Regular Gridding
xyz_renka	Renka-Cline Gridding Method
xyz_renka_nag	NAG Renka-Cline Gridding Method
xyz_shep	Modified Shepard Gridding Method
xyz_shep_nag	NAG Modified Shepard Gridding Method
xyz_sparse	Sparse Gridding
xyz_tps	Thin Plane Spline interpolation

Matrix

Name	Brief Description
mCrop	Crop matrix to a rectangle area
mdim	Set the dimensions and values of XY coordinates for the active matrix
mexpand	Expand for every cell in the active matrix according to the column and row factors
mflip	Flip the matrix horizontally or vertically
mproperty	Set properties of the active matrix
mreplace	Replace cells in the active matrix with specified datamreplace
mrotate90	Rotates the matrix 90/180 degreesmrotate90
msetvalue	Assign each cell in the active matrix from the user definited formula
mshrink	Shrink matrix according shrinkage factors
mtranspose	Transpose the active matrix

Plotting

Name	Brief Description
plotbylabel	Plot a multiple-layers graph by grouping on column labels
plotgroup	Plot by page group, layer group, and data group
plotmatrix	Plot scatter matrix of the dataset
plotmyaxes	Customize Multi-Axes plot
plotstack	Plot stacked graph
plotxy	Plot XY data with specific properties

plotms	Plot color fill surfaces or colormap surfaces for all matrix objects in the specified matrix sheet.
plotvm	Plot from a range of cells in worksheet as a virtual matrix

Worksheet

Name	Brief Description
colcopy	Copy columns with format & headers
colint	Set Sampling Interval (Implicit X) for selected Y columns
colmask	Mask a range of columns based on some condition
colmove	Move selected columns
colshowx	Show X column (extract Sampling Interval) for the selected Y column(s)
colswap	Swap the position of two selected columns
filltext	Fill the cell in the specified range with random letters
getresults	Get the result tree
insertArrow	Insert arrow
insertGraph	Insert a graph into a worksheet cell
insertImg	Insert images from files
insertNotes	Embed a Notes page into a worksheet cell
insertSparklines	Insert sparklines into worksheet cells
insertVar	Insert Variables into cells
merge_book	Merge the workbooks to a new workbook.
sparklines	Add thumbnail size plots of each Y column above the data.

updateEmbedGraphs	Update the embedded Graphs in the worksheet.
updateSparklines	Add thumbnail size plots of each Y column above the data
w2xyz	Convert formatted data into XYZ form
wautofill	Worksheet selection auto fill
wautosize	Resize the worksheet by the column maximal string length.
wcellcolor	Set cell(s) color to fill color or set the selected character font color to Font color.
wcellformat	Format the selected cells
wcellmask	Set cell(s) mask in specified range
wcellsel	Select cell(s) with specified condition
wclear	Worksheet Clear
wcolwidth	Update the width of columns in worksheet
wcopy	Create a copy of the specified worksheet
wdeldup	Remove Duplicated Rows:Remove rows in a worksheet based on duplications in one column
wdelrows	Delete specified worksheet rows
wkeepdup	Hold Duplicated Rows:Hold rows in a worksheet based on duplications in one column
wks_update_link_table	Update the contents in the worksheet to the linked table on graph
wmergexy	Copy XY data from one worksheet to another and merge mismatching X by inserting empty rows when needed
wmove_sheet	Move the specified worksheet to the destination workbook
wmvsn	Reset short names for all columns in worksheet

wpivot	Pivot Table:Create a pivot table to visualize data summarization
wproperties	Get or set the worksheet property through a tree from script
wrcopy	Worksheet Range Copy with options to copy labels
wreplace	Find and replace cell value in a worksheet
wrow2label	Set Label Value
wrowheight	Set row(s) height
wsort	Sort an entire worksheet or selected columns
wsplit_book	Split specific workbooks into multiple workbooks with single sheet
wtranspose	Transpose the active worksheet
wunstackcol	UnStack grouped data into multiple columns
wxt	Worksheet Extraction

18.2.3 Database Access

Name	Brief Description
dbEdit	Create/Edit/Remove/Load Query
dbImport	Import data from database through the query
dbInfo	Show database connection information
dbPreview	Import to certain top rows for previewing the data from the query

18.2.4 Fitting

Name	Brief Description
findBase	Find Baseline region in XY data
fitcmpdata	Compare two datasets to the same fit model
fitcmpmodel	Compare two fit models to the same dataset
fitLR	Simple Linear Regression for LabTalk usage
fitpoly	Polynomial fit for LabTalk usage
getnlr	Get NLFit tree from a fitting report sheet
nlbegin	Start a LabTalk nlfit session
nlbeginm	Start a LabTalk nlfit session on matrix data
nlbeginr	Start a LabTalk nlfit session and fit multiple dependent/independent variables function.
nlbeginz	Start a LabTalk nlfit session on xyz data
nlend	Terminate an nlfit session
nlfit	Iterate the nl fit session
nlfn	Set Automatic Parameter Initialization option
nlgui	Control NLFIT output quantities and destination.
nlpara	Open the Fitting Parameter dialog.

18.2.5 Graph Manipulation

Name	Brief Description
add_graph_to_graph	Paste a graph from existing graphs as an EMF object onto a layout window
add_table_to_graph	Add a linked table to graph
add_wks_to_graph	Paste a worksheet from existing worksheets onto a layout window
add_xyscale_obj	Add a new XY Scale object to the layer
axis_scrollbar	Add a scrollbar object to graph to allow easy zooming and panning
axis_scroller	Add a pair of inverted triangles to the bottom X-Axis that allows easy rescaling
g2w	Move graphs into worksheet
gxy2w	For a given X value, find all Y values from all curves and add them as a row to a worksheet
layadd	Create a new layer on the active graph
layalign	Align some destination layers according to the source layer.
layarrange	Arrange the layers on the graph.
laycolor	Fill layer background color
laycopyscale	Copy scale from one layer to another layer
layextract	Extract specified layers to separate graph windows
laylink	Link several layers to a layer.
laymanage	Manage the organization of layers in the active graph

laysetfont	Fix the display scaling of text in the layer(s) to one.
laysetpos	Set position of one or more graph layers.
laysetratio	Set ratio of layer width to layer height.
laysetscale	Set axes scales for graph layers.
laysetunit	Set unit for graph layers.
layswap	Swap the positions of two graph layers.
laytoggle	Toggle the left axis and bottom axis on and off.
layzoom	Center zooms on layer
legendupdate	Update or reconstruct legend on the graph page/layer
merge_graph	Merge selected graph windows into one graph
newinset	Create a new graph page with insets
newlayer	Add a new layer to graph
newpanel	Create a new graph with panels
palApply	Apply Palette to &Color Map:Apply palette to the specified graph with an existing palette file
pickpts	Pick XY data points from a graph
speedmode	Set speed mode properties

18.2.6 Image

Adjustments

Name	Brief Description
imgAutoLevel	Apply auto leveling to image

imgBalance	Balance the color of image
imgBrightness	Adjust the brightness of Image
imgColorlevel	Apply user-defined color leveling to image
imgColorReplace	Replace color within pre-defined color range
imgContrast	Adjust contrast of image
imgFuncLUT	Apply lookup table function to image
imgGamma	Apply gamma correction to image
imgHistcontrast	Adjust the contrast of image, using histogram to calculate the median.
imgHisteq	Apply histogram equalization
imgHue	Adjust hue of image
imgInvert	Invert image color
imgLevel	Adjust the levels of image
imgSaturation	Adjust Saturation of image

Analysis

Name	Brief Description
imgHistogram	Image histogram

Arithmetic Transform

Name	Brief Description
imgBlend	Blend two images into a combined image
imgMathfun	Perform math function on image pixel values with a factor

imgMorph	Apply morphological filter to numeric Matrix or grayscale/binary image
imgPixlog	Perform logic operation on pixels
imgReplaceBg	Replace background color
imgSimpleMath	Simple Math operation between two Images
imgSubtractBg	Subtract image background

Conversion

Name	Brief Description
img2m	Convert a grayscale image to a numeric data matrix
imgAutoBinary	Auto convert to binary
imgBinary	Convert to binary
imgC2gray	Convert to a grayscale image
imgDynamicBinary	Convert to binary using dynamic threshold
imgInfo	Print out the given image's basic parameters in script window
imgPalette	Apply palette to image
imgRGBmerge	Merge RGB channels to recombine a color image
imgRGBsplit	Split color image into R,G, B channels
imgThreshold	Convert part of an image to black and white using threshold
m2img	Convert a numeric matrix to a grayscale image

Geometric Transform

Name	Brief Description
imgCrop	Crop image to a rectangle area

imgFlip	Flip the image horizontally or vertically
imgResize	Resize image
imgRotate	Rotates an image by a specified degree
imgShear	Shear the image horizontally or vertically
imgTrim	Trim image with auto threshold settings

Spatial Filters

Name	Brief Description
imgAverage	Apply average filter to image
imgClear	Clear the image
imgEdge	Detecting edges
imgGaussian	Apply Gaussian filter
imgMedian	Apply median filter
imgNoise	Add random noise to image
imgSharpen	Increase or decrease image sharpness
imgUnsharpmask	Apply unsharp mask
imgUserfilter	Apply user defined filter

18.2.7 Import and Export

Name	Brief Description
batchProcess	Batch processing with Analysis Template to generate summary report
expASC	Export worksheet data as ASCII file

expGraph	Export graph(s) to graphics file(s)
expImage	Export the active Image into a graphics file
expMatASC	Export matrix data as ASCII file
expNITDM	Export workbook data as National Instruments TDM and TDMS files
expPDFw	Export worksheet as multipage PDF file
expWAV	Export data as Microsoft PCM wave file
expWks	Export the active sheet as raster or vector image file
img2GIF	Export the active Image into a gif file
impASC	Import ASCII file/files
impBin2d	Import binary 2d array file
impCDF	Import CDF file. It supports the file version lower than 3.0
impCSV	Import csv file
impDT	Import Data Translation Version 1.0 files
impEDF	Import EDF file
impEP	Import EarthProbe (EPA) file. Now only EPA file is supported for EarthProbe data.
impExcel	Import Microsoft Excel 97-2007 files
impFamos	Import Famos Version 2 files
impFile	Import file with pre-defined filter.
impHDF5	Import HDF5 file. It supports the file version lower than 1.8.2
impHEKA	Import HEKA (dat) files
impIgorPro	Import WaveMetrics IgorPro (pxp, ibw) files

impImage	Import a graphics file
impinfo	Read information related to import files.
impJCAMP	Import JCAMP-DX Version 6 files
impJNB	Import SigmaPlot (JNB) file. It supports version lower than SigmaPlot 8.0.
impKG	Import KaleidaGraph file
impMatlab	Import Matlab files
impMDF	Import ETAS INCA MDF (DAT, MDF) files. It supports INCA 5.4 (file version 3.0).
impMNTB	Import Minitab file (MTW) or project (MPJ). It supports the version prior to Minitab 13.
impNetCDF	Import netCDF file. It supports the file version lower than 3.1.
impNIDIAdem	Import National Instruments DIAdem 10.0 dat files
impNITDM	Import National Instruments TDM and TDMS files(TDMS does not support data/time format)
impODQ	Import *.ODQ files.
imppClamp	Import pCLAMP file. It supports pClamp 9 (ABF 1.8 file format) and pClamp 10 (ABF 2.0 file format).
impSIE	Import nCode Somat SIE 0.92 file
impSPC	Import Thermo File
impSPE	Import Princeton Instruments (SPE) file. It supports the version prior to 2.5.
impWav	Import waveform audio file
insertImg2g	Insert Images From Files:Insert graphic file(s) into Graph Window
iwfilter	Make an X-Function import filter

plotpClamp	Plot pClamp data
reimport	Re-import current file

18.2.8 Mathematics

Name	Brief Description
avecurves	Average or concatenate multiple curves
averagexy	Average or concatenate multiple curves
bspline	Perform cubic B-Spline interpolation and extrapolation
csetvalue	Setting column value
differentiate	Calculate derivative of the input data
filter2	Apply customized filter to a Matrix
integ1	Perform integration on input data
integ2	Calculate the volume beneath the matrix surface from zero panel.
interp1	Perform 1D interpolation or extrapolation on a group of XY data to find Y at given X values using 3 alternative methods.
interp1q	Perform linear interpolation and extrapolation
interp1trace	Perform trace/periodic interpolation on the data
interp1xy	Perform 1D interpolation/extrapolation on a group of XY data to generate a set of interpolated data with uniformly-spaced X values using 3 alternative methods.
interp3	Perform 3D interpolation
interpxyz	Perform trace interpolation on the XYZ data
marea	Calculate the area of the matrix surface

mathtool	Perform simple arithmetic on data
medianflt2	Apply median filter to a matrix
minterp2	2D Interpolate/Extrapolate on the matrix
minverse	Generate (pseudo) inverse of a matrix
normalize	Normalize the input data
polyarea	Calculate the area of an enclosed plot region
reflection	Reflect a range of data to certain interval
rnormalize	Normalize Columns:Normalize the input range column by column
specialflt2	Apply predefined special filter to a matrix
spline	Perform spline interpolation and extrapolation
vcmath1	Perform simple arithmetic on one complex number
vcmath2	Perform simple arithmetic on two complex numbers
vmathtool	Perform simple arithmetic on input data
vnormalize	Normalize the input vector
white_noise	Add white (Gaussian) noise to data
xyzarea	Calculate the area of the XYZ surface

18.2.9 Signal Processing

Name	Brief Description
cohere	Perform coherence
conv	Compute the convolution of two signals
corr1	Compute 1D correlation of two signals

corr2	2D correlation.
deconv	Compute the deconvolution
envelope	Get envelope of the data
fft_filter2	Perform 2D FFT filtering
fft_filters	Perform FFT Filtering
hilbert	Perform Hilbert transform or calculate analytic signal
msmooth	Smooth the matrix by expanding and shrinking
smooth	Perform smoothing to irregular and noisy data.

FFT

Name	Brief Description
fft1	Fast Fourier transform on input vector (discrete Fourier transforms)
fft2	Two-dimensional fast Fourier transform
ifft1	Perform inverse Fourier transform
ifft2	Inverse two-dimensional discrete Fourier transform
stft	Perform Short Time Fourier Transform
unwrap	Transfer phase angles into smoother phase

Wavelet

Name	Brief Description
cw_evaluate	Evaluation of continuous wavelet functions
cwt	Computes the real, one-dimensional, continuous wavelet transform coefficients

dwt	1D discrete wavelet transform
dwt2	Decompose matrix data with wavelet transform
idwt	Inverted 1D Wavelet Transform from its approximation coefficients and detail coefficients.
idwt2	Reconstruct 2D signal from coefficients matrix
mdwt	Multilevel 1-D wavelet decomposition
wtdenoise	Remove noise using wavelet transform
wtsmooth	Smooth signal by cutting off detailed coefficients

18.2.10 Spectroscopy

Name	Brief Description
blauto	Create baseline automatically
fitpeaks	Pick multiple peaks from a curve to fit Guassian or Lorentzian peak functions
pa	Open Peak Analyzer
paMultiY	Peak Analysis batch processing using Analysis Theme to generate summary report
pkFind	Pick peaks on the curve.

18.2.11 Statistics

Descriptive Statistics

Name	Brief Description
colstats	Perform statistics on columns
corrcoef	Calculate correlation coefficients of the selected data

discfreqs	Calculate Frequency for discrete/categorical data
freqcounts	Calculate frequency counts
kstest	One sample Kolmogorov-Smirnov test for normality
lillietest	Lilliefors normality test
mmoments	Calculate moments on selected data
moments	Calculate moments on selected data
mquantiles	Calculate quantiles on selected data
mstats	Calculate descriptive statistics on selected data
quantiles	Calculate quantiles on selected data
rowquantiles	Calculate quantiles on row(s)
rowstats	Descriptive statistics on row(s)
stats	Calculate descriptive statistics on selected data
swtest	Shapiro-Wilk test for normality:Shapiro-Wilk Normality test

Hypothesis Testing

Name	Brief Description
rowttest2	Perform a two-sample t-test on rows
ttest1	One-Sample t-test
ttest2	Two-Sample t-test
ttestpair	Pair-Sample t test
vartest1	Chi-squared variance test
vartest2	Perform a F-test.

Nonparametric Tests

Name	Brief Description
friedman	Perform a Friedman ANOVA
kstest2	Perform a two-sample KS-test on the input data.
kwanova	Perform Kruskal-Wallis ANOVA
mediantest	Perform median test
mwtest	Preform Mann-Whitney test
sign2	Perform paired sample sign test
signrank1	Perform a one-sample Wilcoxon signed rank test
signrank2	Preform paired sample Wilcoxon signed rank test

Survival Analysis

Name	Brief Description
kaplanmeier	Perform a Kaplan-Meier (product-limit) analysis
phm_Cox	Perform a Cox Proportional Hazards Model analysis
weibullfit	Perform a Weibull fit on survival data

18.2.12 Utility

Name	Brief Description
customMenu	Open Custom Menu Editor Dialog.
get_plot_sel	Get plot selections in data plot
get_wks_sel	Get selections in worksheet

themeApply2g	Apply a theme to a graph or some graphs.
themeApply2w	Apply a theme to a worksheet or some worksheets.
themeEdit	Edit the specific theme file using **Theme Editing** tool.
xop	X-Function to run the operation framework based classes.

File

Name	Brief Description
cmpfile	Compare two binary files and print out comparison results
dlgFile	Prompt user to select a file with an Open file dialog.
dlgPath	Prompt user to select a path with an Open Path dialog.
dlgSave	Prompt user with an Save as dialog.
filelog	Create a .txt file that contains notes or records of the user's work through a string
findFiles	Searches for a file or files.
findFolders	Searches for a folder or folders.
imgFile	Prompt user to select an image with an Open file dialog.
template_saveas	Save a graph/workbook/matrix window to a template
web2file	Copy a web page to a local file

System

Name	Brief Description
cd	Change or show working directory
cdset	Assigns a specified index to the current working directory, or lists all assigned indices and associated paths.

debug_log	Used to create a debug log file. Turn on only if you have a problem to report to OriginLab.
dir	list script (ogs) and x-functions (oxf) in current working directory.
dlgChkList	Open a dialog with check boxes and return each check box's selected status when the dialog is closed.
group_server	Set up the Group Folder location for both group leader and members
groupmgr	Group Leader's tool to manage Group Folder files
instOPX	Install an Origin XML Package
language	Change Origin Display Language
lc	Lists x-function categories, or all x-functions in a specified category.
lic	Update Module License:Add module license file into Origin
lx	Lists x-functions (by name, keyword, location etc)
mkdir	Create a new folder in the current working directory
op_change	Get and set tree stored in operation object
pb	Open the Project Browser
pe_cd	Change project explorer directory
pe_dir	Lists current project explorer folders and workbooks
pe_load	Load an Origin project into an existing folder in the current project
pe_mkdir	Create new folder
pe_move	Move specified page of folder to specified folder
pe_path	Find Project Explorer path
pe_rename	Rename Page or subfolder

pe_rmdir	Delete a subfolder under the active folder in PE
pe_save	Save a folder from the current project to an Origin project file
pef_pptslide	Export all graphs in folder to PowerPoint Slides
pef_slideshow	Slide Show (full screen view) of all graphs in folder
pemp_pptslide	Export selected graphs to PowerPoint Slides
pemp_slideshow	Slide Show (full screen view) of selected graphs
pep_addshortcuts	Create shortcuts for selected windows in Favorites folder
pesp_gotofolder	Go to the original folder where this page locates
updateUFF	Transfer user files in Origin75 to Origin8
ux	Update x-function list in specified location

Index