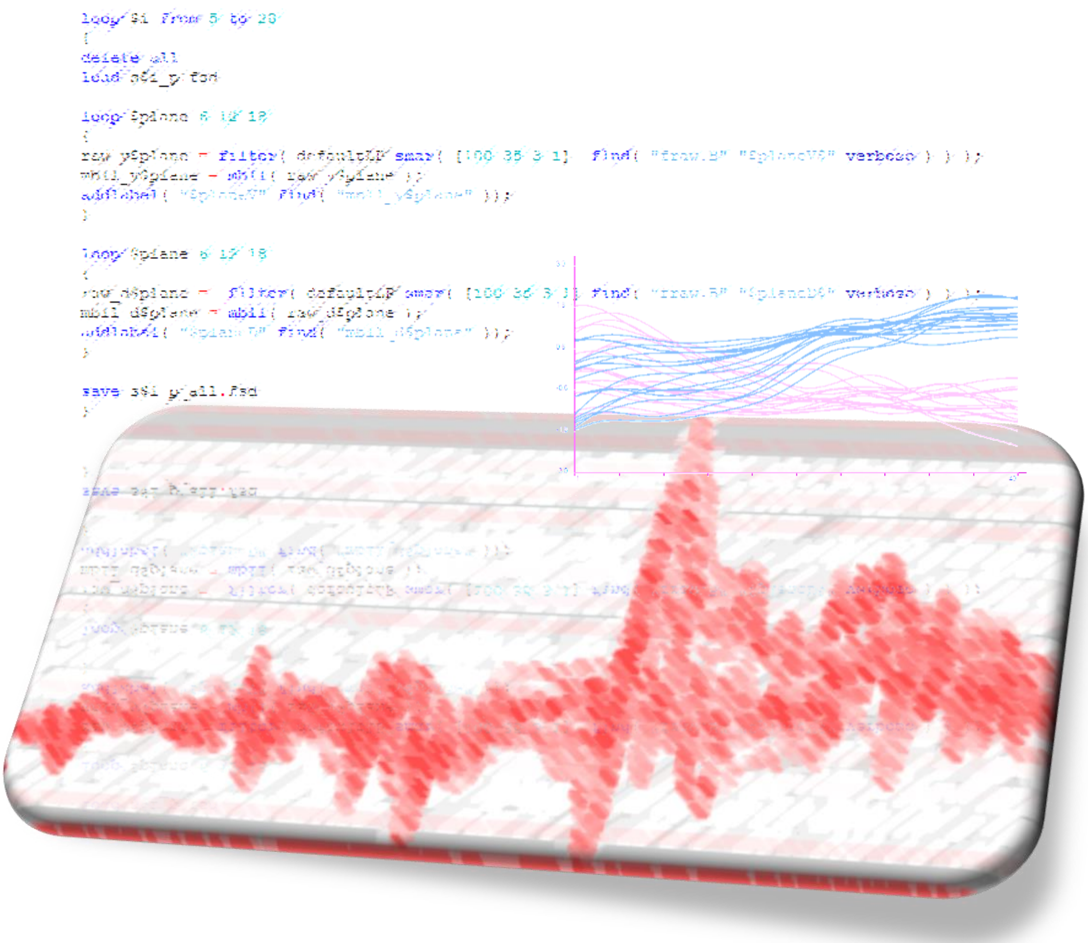
fnirSoft Scripting

A Quick Start Guide



fnirSoft

Please refer to fnirSoft in your publications with the following:

Ayaz, H. (2010). "*Functional Near Infrared Spectroscopy based Brain Computer Interface". *PhD Thesis, Drexel University, Philadelphia, PA.

Disclaimer

THIS SOFTWARE IS PROVIDED ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

1. Introduction 6

1.1. Overview 6

2. Quick Start 7

2.1. Hello World! 7

2.2. Mathematical Expressions 7

2.3. Creating and Using Variables 8

2.3.1. Creating a Single Value – Numeric Variable 8

2.3.2. Creating a Vector – Numeric Variable 8

2.3.3. Creating an Array – Numeric Variable 9

2.3.4. Creating a String Variable 10

2.3.5. Creating a List Variable 10

2.4. Intellisense 11

2.5. Command Popup Help Documentation 12

2.6. Help Explorer 12

2.7. Editor Tool 12

2.7.1. Coding Pane 15

2.7.2. Debug Pane 16

2.8. Output Window 18

2.9. Execution Control 18

2.9.1. Loops 18

2.9.2. Nested loops 19

2.9.3. Conditionals 20

2.9.4. Warning and Error Handling 20

2.9.5. Debugging 23

2.9.6. Try/Catch/Finally 24

2.9.7. Current directory 24

2.10. Functions 26

3. Common programming/use patterns 27

3.1. Loading Light Intensity data (*.nir) files 27

3.2. Loading Oxygenation/Hemoglobin concentration changes data (*.oxy) files 27

3.3. Loading Marker/Event data (*.mrk) files 28

3.4. Loading fnirSoft data (*.fsd) files 28

3.5. Loading multiple files simultaneously 29

3.6. Calculating Oxygenation from Light Intensity data 30

3.7. Using Find command 30

3.8. Calculating Averages 32

3.9. Saving variables to fnirSoft data (*.fsd) files 33

3.10. Applying FIR Filter 34

3.11. Defining Block Times 36

3.12. Standardizing 38

3.13. Splitting Variables 39

3.14. Exporting variables to Matlab 40

3.15. Exporting variables to Text Files 40

4. Naming Conventions 42

4.1 Categories 42

4.2 Variable Associations 43

5. Indexing 45

5.1. Get Numeric Variable Indexing 45

5.2. Set Numeric Variable Indexing 46

5.3. List Variable Indexing 47

6. List of Commands 49

Console 49

DAQ 49

Dataspace 49

DateTime 50

Execution 50

Functions 50

Lightgraph 50

Math 51

Oxygraph 51

Spatial 51

System 52
Temporal 52
Topograph 54
Utility 54
Variable 55

1. Introduction

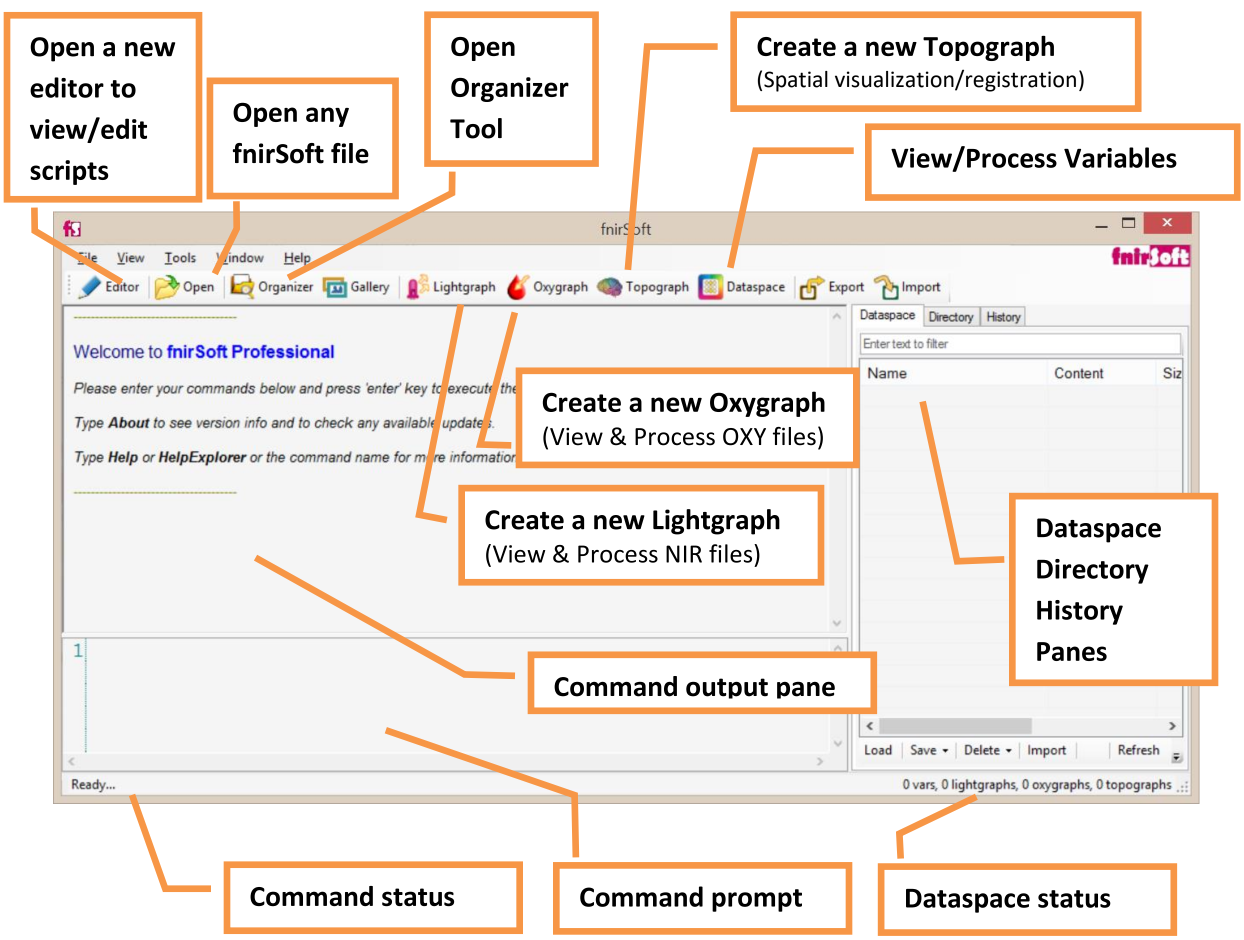
fnirSoft (fS) is a stand-alone software package designed to process, analyze and visualize functional near infrared spectroscopy signals through both graphical user interface and/or scripting. This document provides an introductory tutorial for fnirSoft scripting (programming).

fS script is a data-driven functional programming language for neuroimaging data and was designed to do more with less code and be both readable by humans and computers. The idea is to write simple procedural text to easily manage and automate data processing for scientific computing. fS Script takes advantage of neuroimaging data structure and using intrinsic language features and commands to eliminate repetitive and error prone elements that would be necessary to process data in other languages. fS Script is also aimed to help repeatable research by standardizing functional neuroimaging signal processing steps and help establishing conventions that can be replicated over time on different datasets.

1.1. Overview

Below is the main window of fnirSoft with common user elements identified. For more information, please refer to fnirSoft User Manual (2017).

Below is the main window of fnirSoft with common user elements and tools identified.



fnirSoft has a scripting engine that interprets written procedural and descriptive commands at run-time. Commands can be entered to the "command prompt" at the bottom of the window and executed by pressing "Enter". Algebraic expressions can be evaluated. Overview of the syntax and step by step tutorials will be provided in the next chapters. Longer scripts can be composed in Editor Tool and saved for future use. History Tools keeps track of all executed commands through the command prompt and can be accessed either by executing History command or through top menu View>History dialog.

TIP
Type "2 +3" (without quotes) at the command prompt and press enter. You should see the result 5 at the output pane.

2. Quick Start

Scripts are series of commands written in text. They can be executed either by typing at the command prompt or by saving them in separate files and executing the file.

Parentheses are used to prioritize processing and identify parameters for a command.

Space is the basic separator between functions, names, numbers and everything…

2.1.Hello World!

Here's your first fnirSoft program. Type in the following at command prompt and hit enter. The code will be repeated at the command output pane and output will be displayed below it as shown below.

```
WriteLine("Hello world!");
```

```
Hello world!
```

2.2.Mathematical Expressions

Binary arithmetic operators + (addition), - (subtraction), * (multiplication), / (division); and unary operators ^ (power), ' (transpose) are available.

```
12 + 5
```

```
12 - 5
```

```
0.5 * Log( Exp( 10 ) ) + ( 16.2 * 5 - 1)
```

```
Sin( pi / 2 )
```

```
Atan( Tan( pi / 2 ) ) / pi
```

```
17

7

85

1

0.5
```

TIP

If you add ';' (semicolon) after the command entry, no output will be displayed at the output pane!

2.3.Creating and Using Variables

2.3.1. Creating a Single Value – Numeric Variable

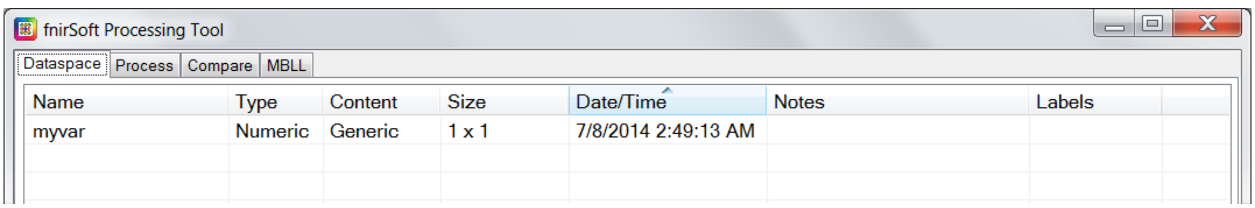
To create a new variable, just type a valid name (that starts with a letter and not a reserved word) and use '**=**' (equals sign) to assign value to the variable.

```
myvar = 5
```

```
myvar =
5
```

With this single assignment, a new variable called 'myvar' is created. A number is assigned as value and the size of the variable is 1x1 (single scalar). All available variables can be seen at the 'Dataspace' window and can also be listed by typing '**Variables**' or '**Dataspace.Variables**' at the command prompt and hitting enter or, by using the toolbar button at the main window (See figure in page 3).

```
>> Variables
    fs.Dataspace.Variables:
    ans                             1x1
    myvar                           1x1
    Total number: 2
```



Note that there's another variable called 'ans' which is short for answer and always contains the results from last operation.

A variable can be single scalar value, a vector or an array. After creating the variable, the name can be used in subsequent commands.

```
myvar + 2
```

```
7
```

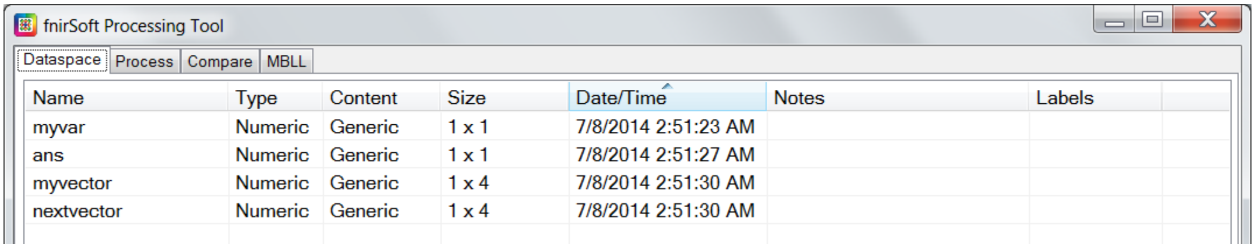
2.3.2. Creating a Vector – Numeric Variable

To create a vector, type values separated by space between squared parenthesis: '[' and ']' as shown below.

```
myvector = [ 1 2 3 4];
nextvector = myvector * 2
```

```
nextvector =
2 4 6 8
```

After the two commands are executed, the new variables are also available in the list.



Name	Type	Content	Size	Date/Time	Notes	Labels
myvar	Numeric	Generic	1 x 1	7/8/2014 2:51:23 AM		
ans	Numeric	Generic	1 x 1	7/8/2014 2:51:27 AM		
myvector	Numeric	Generic	1 x 4	7/8/2014 2:51:30 AM		
nextvector	Numeric	Generic	1 x 4	7/8/2014 2:51:30 AM		

TIP

Use ‘**Variables**’ command to list all variables in Dataspace (with name and sizes) within the command pane.

```
>> Variables
    fs.Dataspace.Variables:
    myvar                       1x1
    ans                         1x1
    myvector                    1x4
    nextvector                  1x4
    Total number: 4
```

TIP

Use ‘**Delete all’** command to delete all variables and objects in Dataspace.

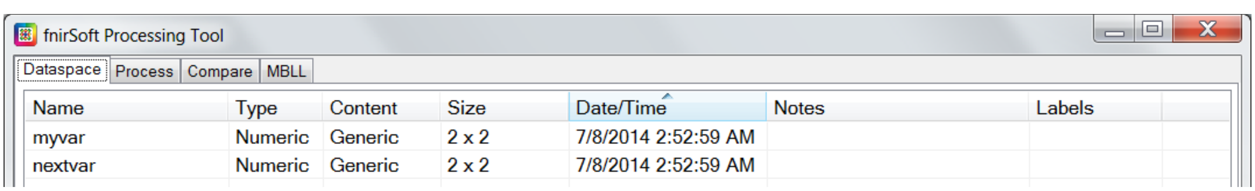
```
>> Delete all
    fs.Dataspace.Delete: All variables and objects are deleted!
```

2.3.3. Creating an Array – Numeric Variable

The process is similar to creating a vector as described above. Use ‘**,**’ (comma) to indicate end of row/new row. To create an array/matrix, type values separated by space between squared parenthesis: ‘[‘ and ‘]’ as shown below, and use ‘,’ (comma) to separate rows.

```
myvar = [ 5 6, 7 8];
nextvar = myvar * 10
```

```
    nextvar =
    50  60
    70  80
```



Name	Type	Content	Size	Date/Time	Notes	Labels
myvar	Numeric	Generic	2 x 2	7/8/2014 2:52:59 AM		
nextvar	Numeric	Generic	2 x 2	7/8/2014 2:52:59 AM		

2.3.4. Creating a String Variable

A string is defined as text with in double quote (“) characters. To create a new string variable, similar to numeric assignments, just type a valid name (that starts with a letter and not a reserved word) and use ‘=’ (equals sign) to assign string value to the variable.

```
mystring = "This is a string!"
```

```
mystring =
"This is a string!"
```

The string variable can be accessed by name. Also, “+” and “-“ operators can be used to concatenate multiple string and to remove string from one another.

2.3.5. Creating a List Variable

List variable is a collection of multiple variables that are either numeric or string. List variables help organizing related variable by placing together and can still access each member item (variable) separately. In the following example, three previously defined variables are input to “ToList” command to create a new list variable.

```
mylist = ToList( myvar myvector mystring)
```

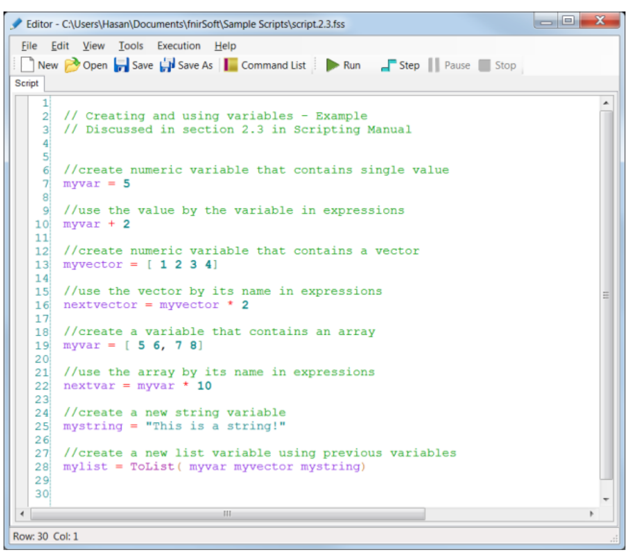
```
mylist =
(1) :
5 6
7 8

(2) :
1 2 3 4

(3) :
"This is a string!"
```

TIP

Sample script file “script2.3.fss” contains all expressions used here with comments, you can open this file and execute from the fnirSoft Editor.

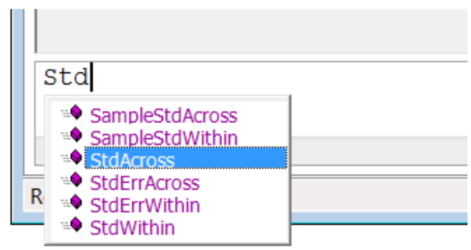


2.4.Intellisense

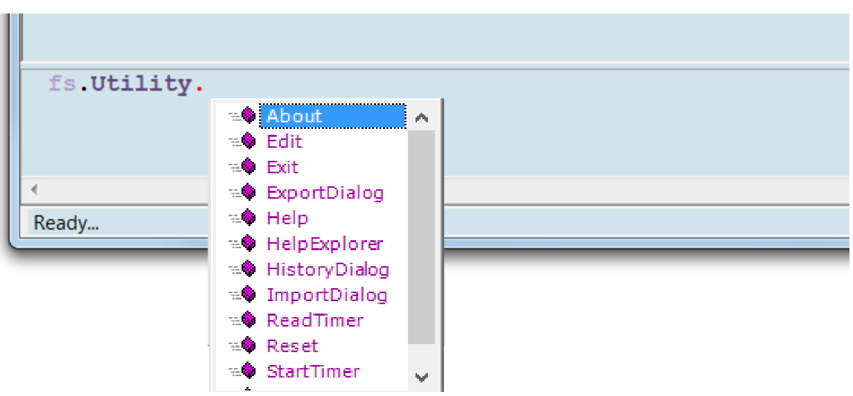
When typing variable names or command names, intellisense pop-up dialog displays suggestions and auto-complete options to help typing your scripts. After you type a few letters, intellisense will pop-up automatically with a list that displays commands and variables that include that partial text. The list is updated as you continue typing. You can hit enter to select one (highlighted option will be inserted), or click on any option with mouse cursor or hit 'Escape' button (on the keyboard) to cancel and close intellisense. If intellisense is not open or closed, use ctrl+space button combination to re-launch it.

For example, type 'std' in the command prompt. These three characters will list the following popup-window with the matching command names (variables created in previous examples) will be shown as candidates.

```
std
```



Also, the '.' (dot) character is a trigger for intellisense. When you type collection and category names followed by '.' (dot) all subitems are listed. The default collection for fnirSoft commands is "fs" and "Utility" is one of the categories. When you type "Utility" followed by '.' All commands in that category is listed. You can see additional information for each command if you click on one of them (as tooltip) or hover the mouse cursor over one of the written commands in the command prompt.

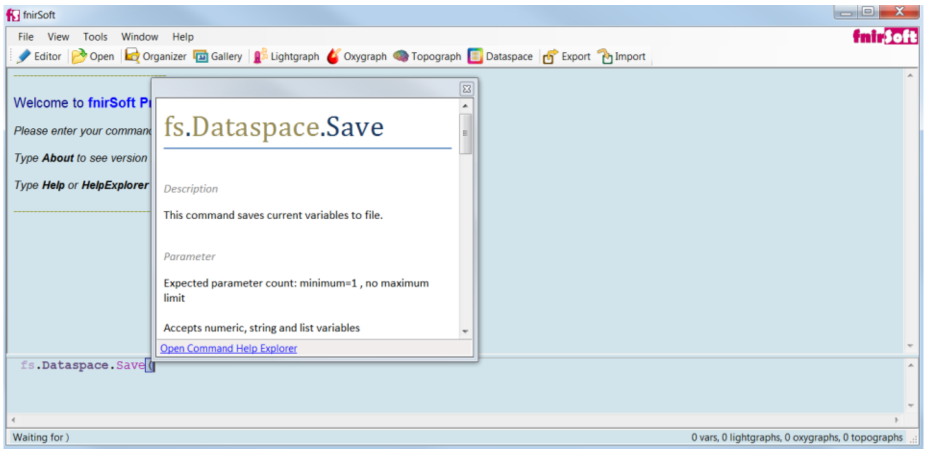


TIP

Use '**up/down**' cursor keys (on the keyboard) to navigate between command history. You can immediately recall the previous executed entries by using up and down keyboard buttons.

2.5.Command Popup Help Documentation

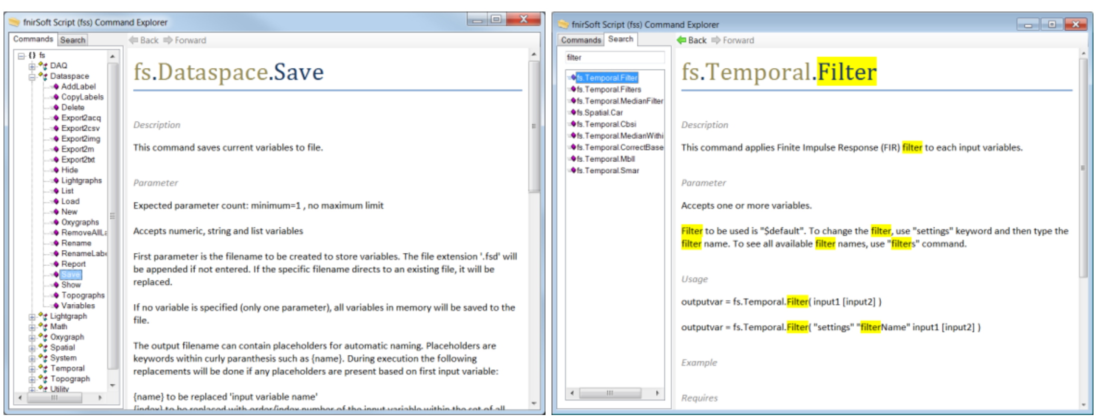
When you type a valid command name followed by '(', open parenthesis character, a popup window with description of that command is displayed. You can mouse cursor to scroll up/down, or click "Open Command Help Explorer" button to read more. Otherwise, just continue typing and the popup window will disappear automatically.



2.6.Help Explorer

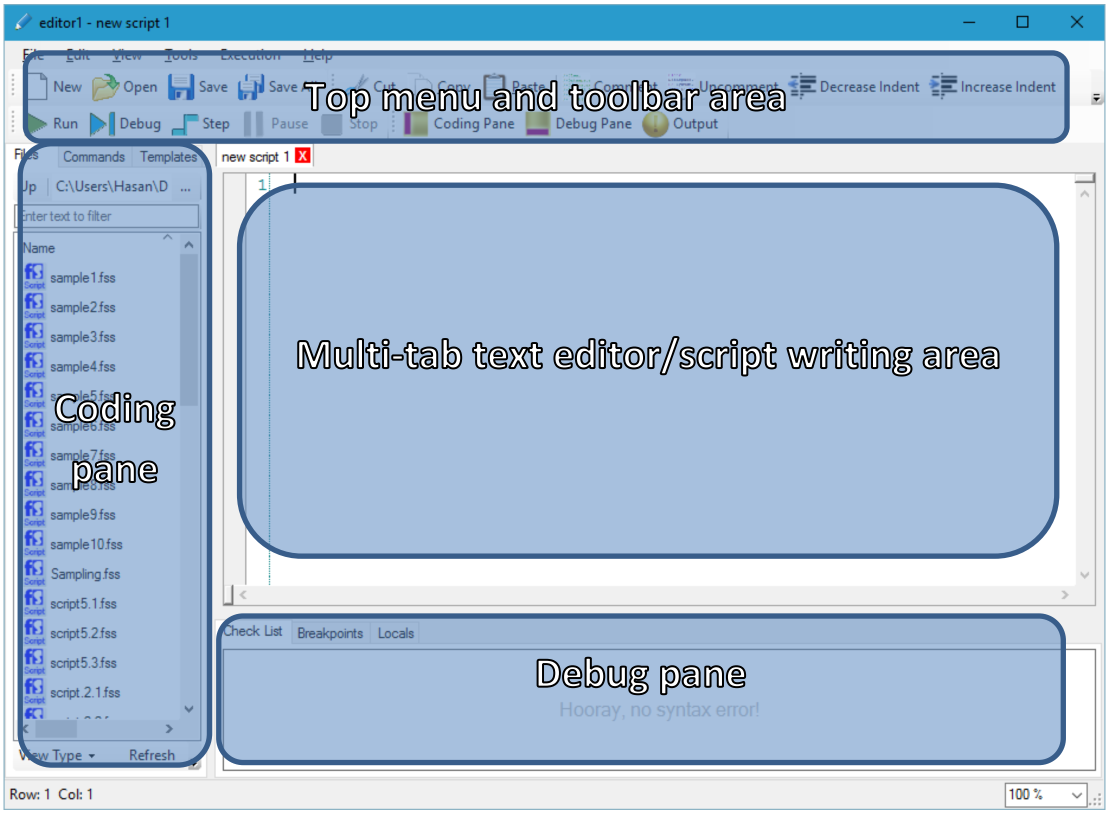
Command help explorer provides displays help documentation for all fnirSoft Script commands. To launch use "HelpExplorer" command, or the link in popup help window (See above) or at the main window, top menu, Help>fnirSoft Command Explorer menu item.

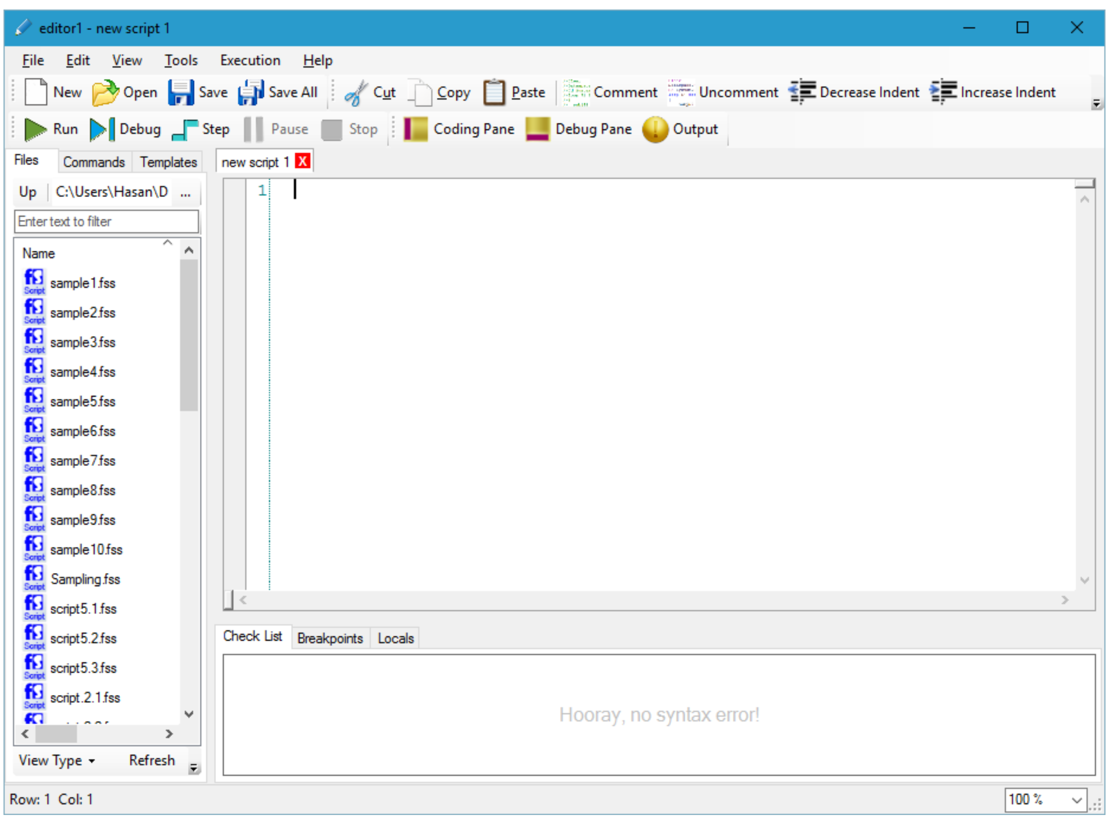
The left pane at Help Explorer lists all available items in the first tab hierarchically and in the second tab, allows search within all documentation.

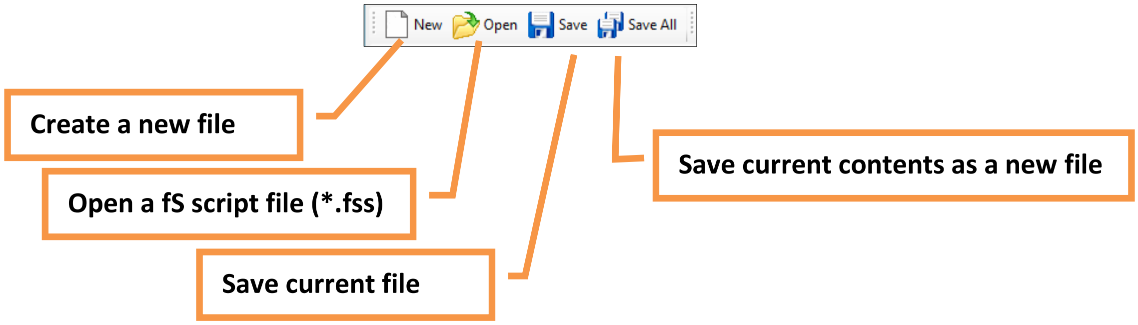


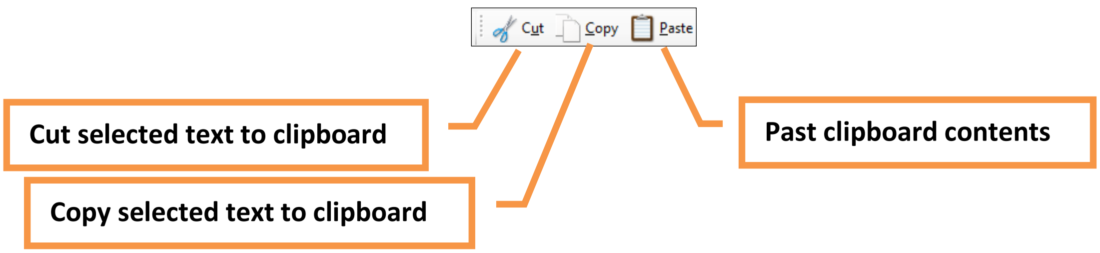
2.7.Editor Tool

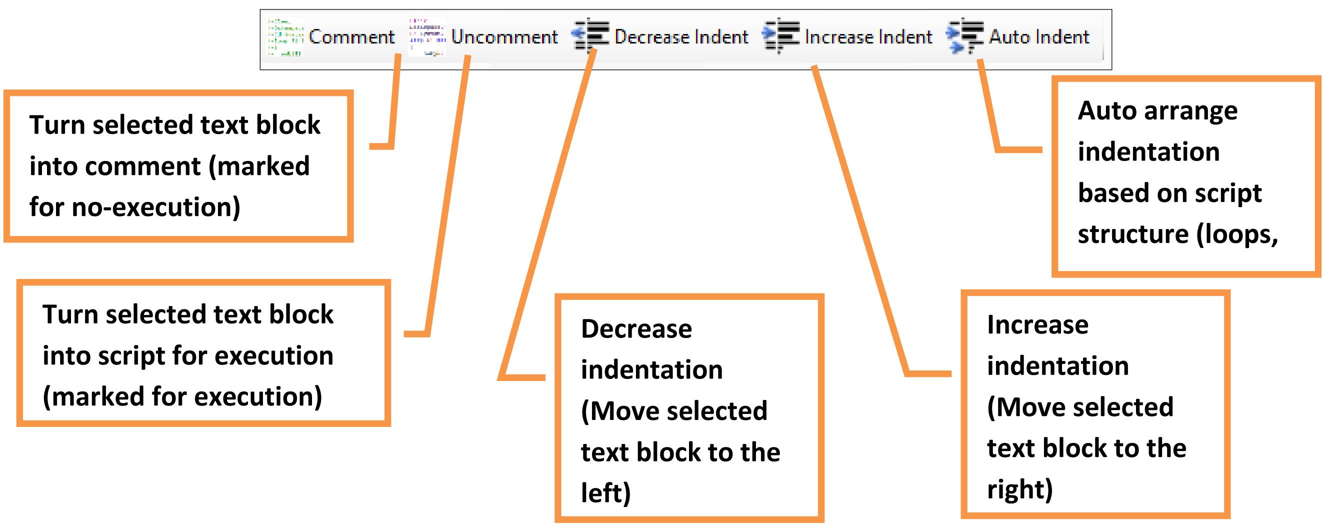
Editor Tool is for composing fS Scripts with all syntax highlighting and intellisense available same as on the command prompt. Editor Tool allows loading/saving/executing fS Programs. Editors can be opened by clicking the 'Editor' button at the toolbar of main window or just typing '**editor**' at the command prompt. Below is an empty editor window.

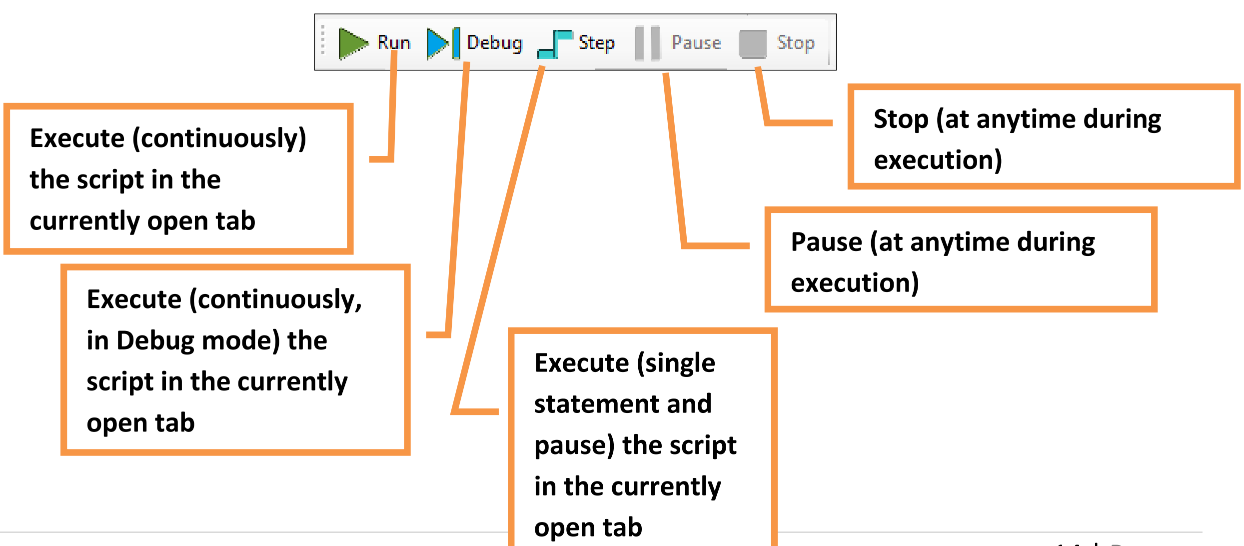

editor1 - new script 1
Top menu and toolbar area
New Open Save Save
Decrease Indent Increase Indent
Run Debug Step Pause Stop Coding Pane Debug Pane Output
Commands Templates
new script 1
C:\Users\Hasan\D
Coding
pane
Multi-tab text editor/script writing area
Check List Breakpoints Locals
Debug pane
Hooray, no syntax error!
Refresh
Row: 1 Col: 1
100 %

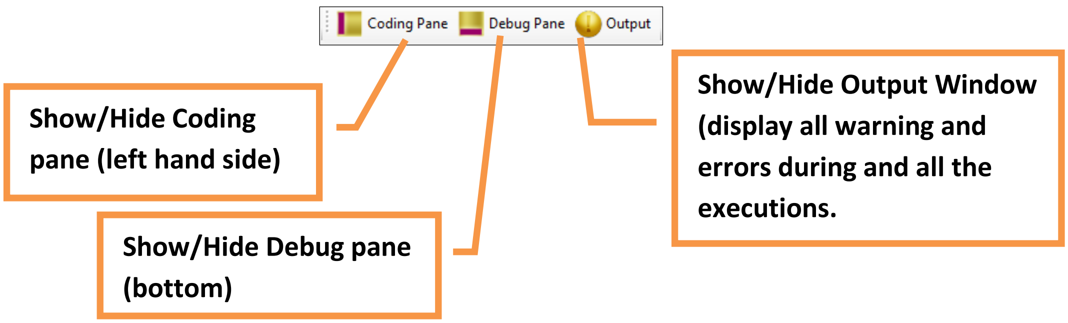

editor1 - new script 1
File Edit View Tools Execution Help
New Open Save Save All Cut Copy Paste Comment Uncomment Decrease Indent Increase Indent
Run Debug Step Pause Stop Coding Pane Debug Pane Output
Files Commands Templates
new script 1
Up C:\Users\Hasan\D ...
Enter text to filter
Name
sample1.fss
sample2.fss
sample3.fss
sample4.fss
sample5.fss
sample6.fss
sample7.fss
sample8.fss
sample9.fss
sample10.fss
Sampling.fss
script5.1.fss
script5.2.fss
script5.3.fss
script.2.1.fss
Check List Breakpoints Locals
Hooray, no syntax error!
View Type Refresh
Row: 1 Col: 1
100 %


New
Open
Save
Save All
Create a new file
Open a fS script file (*.fss)
Save current file
Save current contents as a new file


Cut
Copy
Paste
Cut selected text to clipboard
Copy selected text to clipboard
Past clipboard contents

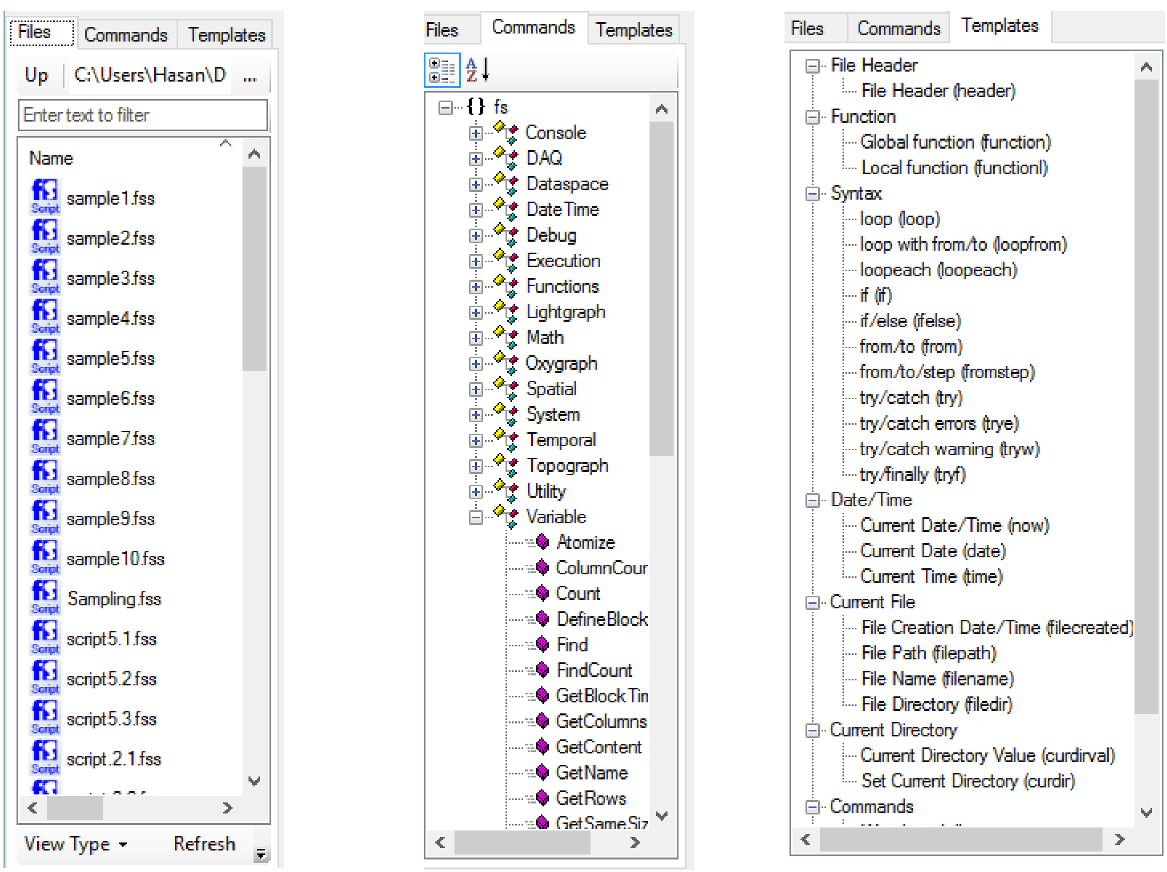

Comment
Uncomment
Decrease Indent
Increase Indent
Auto Indent
Turn selected text block into comment (marked for no-execution)
Turn selected text block into script for execution (marked for execution)
Decrease indentation (Move selected text block to the left)
Increase indentation (Move selected text block to the right)
Auto arrange indentation based on script structure (loops,


Run
Debug
Step
Pause
Stop
Execute (continuously) the script in the currently open tab
Execute (continuously, in Debug mode) the script in the currently open tab
Execute (single statement and pause) the script in the currently open tab
Pause (at anytime during execution)
Stop (at anytime during execution)



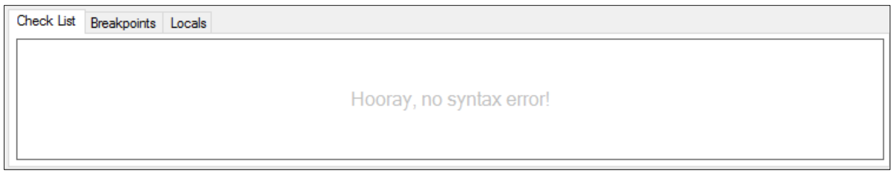
2.7.1. Coding Pane

There are three tab for the coding page: files (lists all script and text files in the current directory, commands (lists all fs category and commands) and templates (script statements for fast typing). Use the coding pane to quickly perform tasks, for example, clicking on a file will open it in a new tab, clicking on a command, will enter the full command name into the text editing area, or clicking on a template, will enter the respective script block into the text editing area.

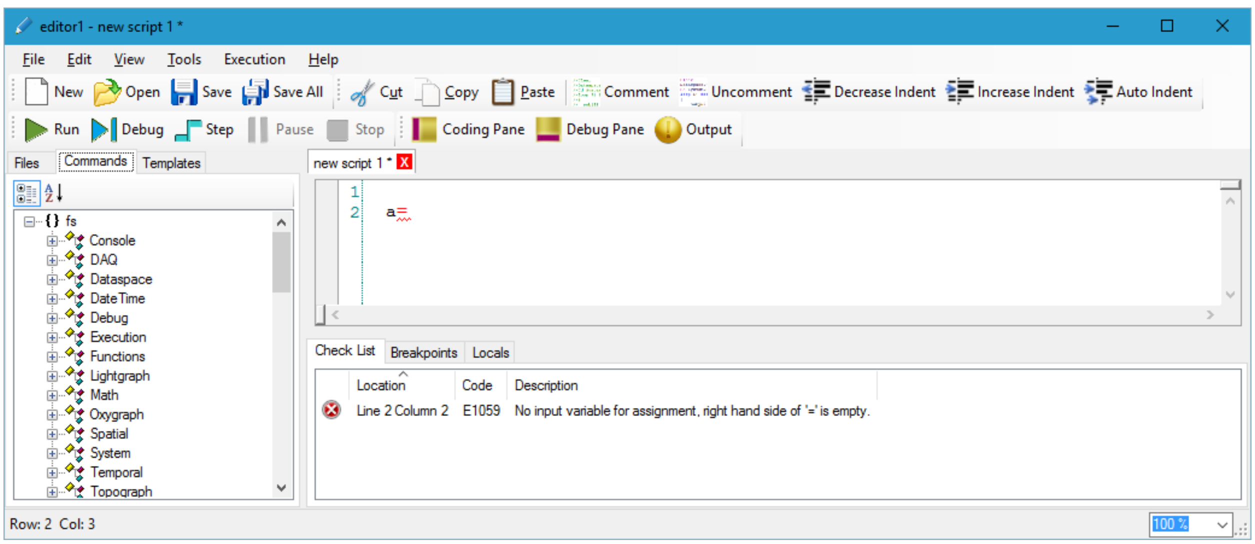


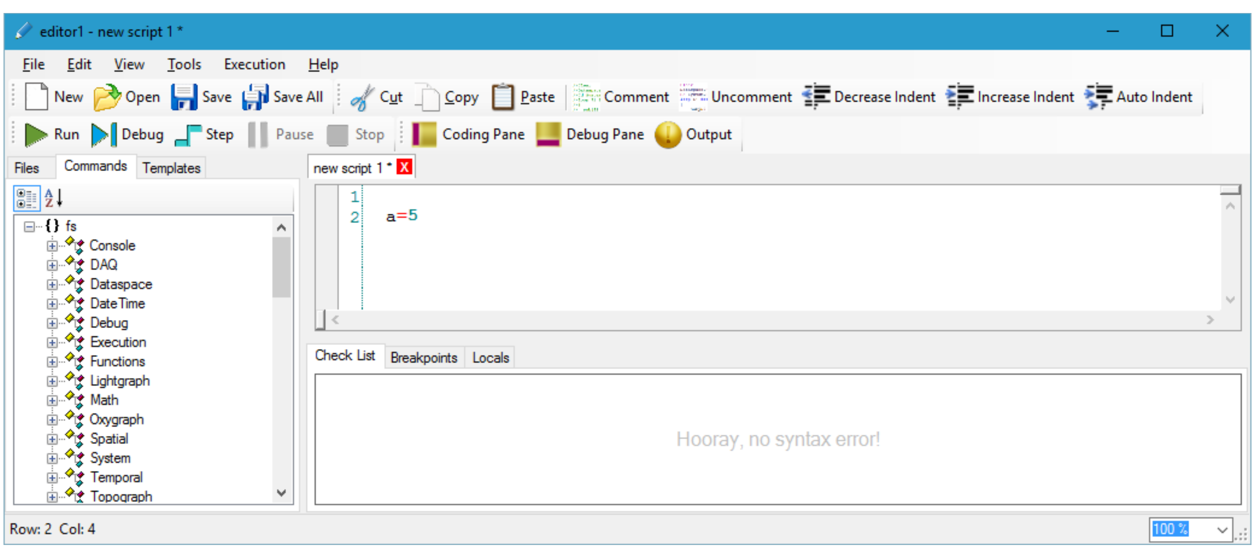
2.7.2. Debug Pane

The Debug pane is at the bottom of the editor window and contains three tabs: Check List (lists errors and warning while you are typing the script), Breakpoints (lists breakpoints marked by clicking the beginning of the row, row number area, that execution will pause during debug mode), Locals (lists local/temporary variables that do not appear in dataspacse for debuging)



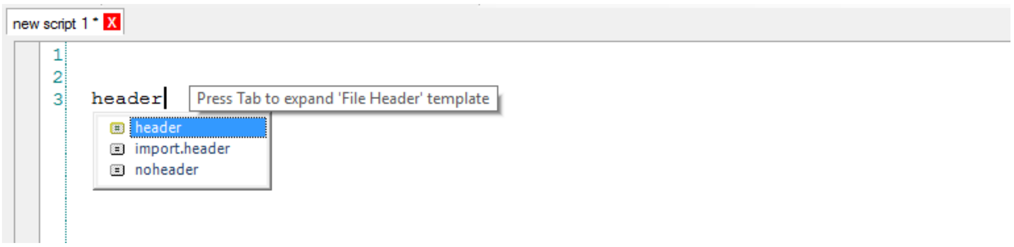
The check list pane lists any run-time syntax error that are detected as you type. To test is type "a=" since no assigment value is detected an error will immediately appears in the check list pane. When you complete it by adding a number 'a=5' the error will dissappear. See the following figures.



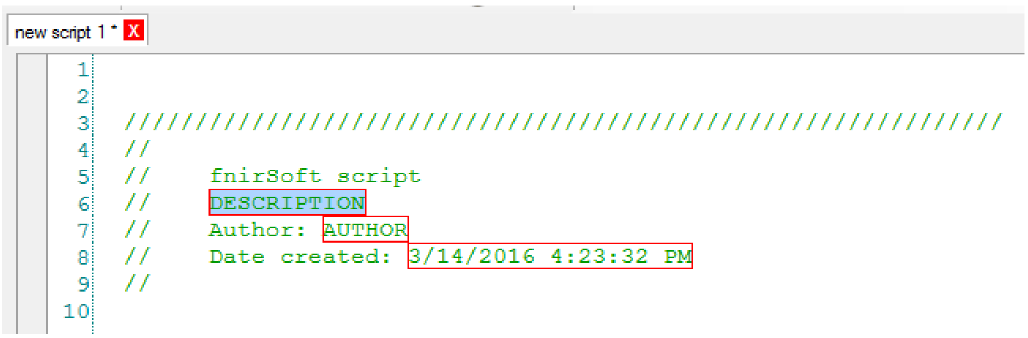


TIP

Use *Templates* by typing template identifiers and pressing <tab> key to expand:



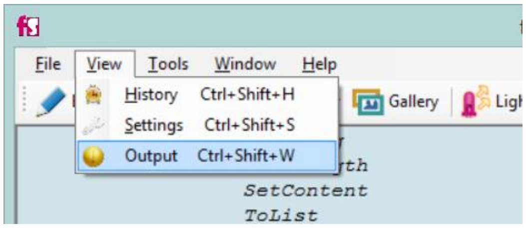
When <tab> button on the keyboard is pressed the following is automatically inserted:



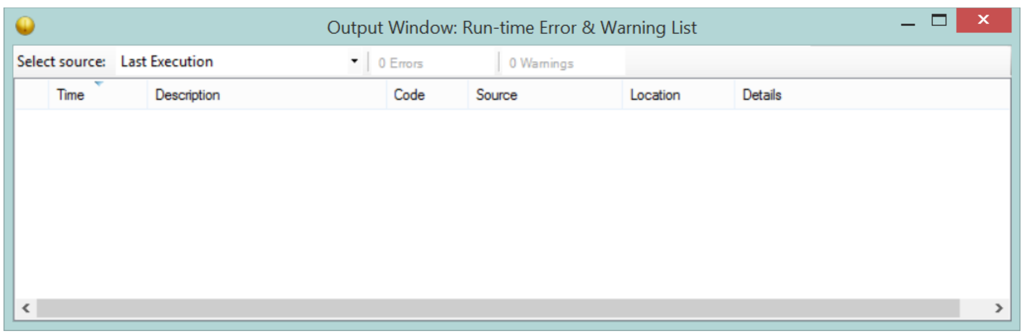
See template tab (at the coding pane) to see list of template identifiers (which are listed in parenthesis next to the template item.

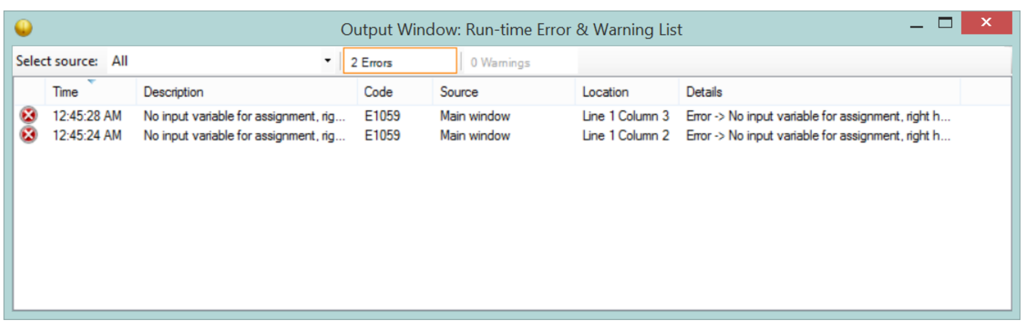
2.8.Output Window

Output window provides a single location to see warnings and errors generated throughout the execution of scripts. It can be access from main window, under View top menu item.



When the output window is launched it will display list of errors from last execution. Also, all errors can be selected from the “select source” combo box. Also number of errors and warnings are provided at the toolbar.





2.9.Execution Control

2.9.1. Loops

Iterations and repetitions are critical in performing same task on large amount of data. To create a loop, use “**loop”** keyword followed by an **iterator** name (i.e. any valid variable name that starts with $ sign) and use, “**from**” and “**to**” keywords to indicate start and end counts for the helper variable to go through. After typing the code, hit ‘Run’ button (at the editor toolbar) to execute the script. Output will be shown at the command output pane of the main window.

```
// count from 1 to 5

loop $i from 1 to 5
{
      WriteLine "$i"
}
```

```
Executing file C:\Users
1
2
3
4
5
```

```
// count from 1 to 5
// with increments of 2

loop $i from 1 to 5 step 2
{
      WriteLine "$i"
}
```

```
Executing file C:\Users
1
3
5
```

```
// count from 1 to 5
// with increments of 2
// with exceptions
loop $i from 1 to 5 step 2 except 3
{
      WriteLine "$i"
}
```

```
Executing file C:\Users\
1
5
```

```
// count free-form, identify each

loop $i 5 2 3 1
{
      WriteLine "$i"
}
```

```
Executing file C:\Users\
5
2
3
1
```

2.9.2. Nested loops

Loops within loops can be used, below is an example code that has nested-loops and the output indicates the order of execution.

```
// nested loops

loop $i from 1 to 3
{

      loop $j from 10 to 7
      {
            WriteLine( "$i $j" );
      }

}
```

```
Executing file C:\Users\
1 10
1 9
1 8
1 7
2 10
2 9
2 8
2 7
3 10
3 9
3 8
3 7
```

2.9.3. Conditionals

If-else statements can be used to create operation branches. The syntax is as follows.

If *logical-statement*
    Operation1
else
    Operation2

The logical statement is composed of two values/variables separated by a comparison operator ('<', '>' or '='). Else statement is optional. Both operation1 and operation2 should be single operations that is on single line without any ';'.

```
testvar = 1

if testvar  > 0
       WriteLine ( "SUCCESS!" )
else
       WriteLine ( "FAILED!")
```

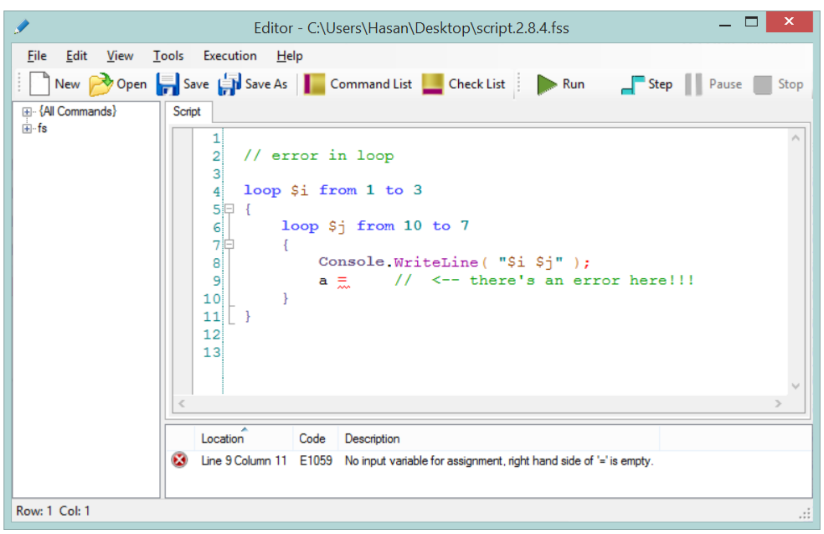
```
Executing file C:\Users\
testvar =
1

SUCCESS!
```

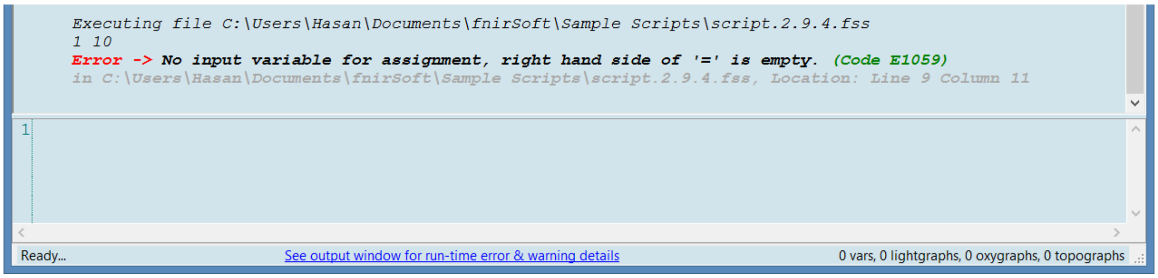
2.9.4. Warning and Error Handling

2.9.4.1. Errors

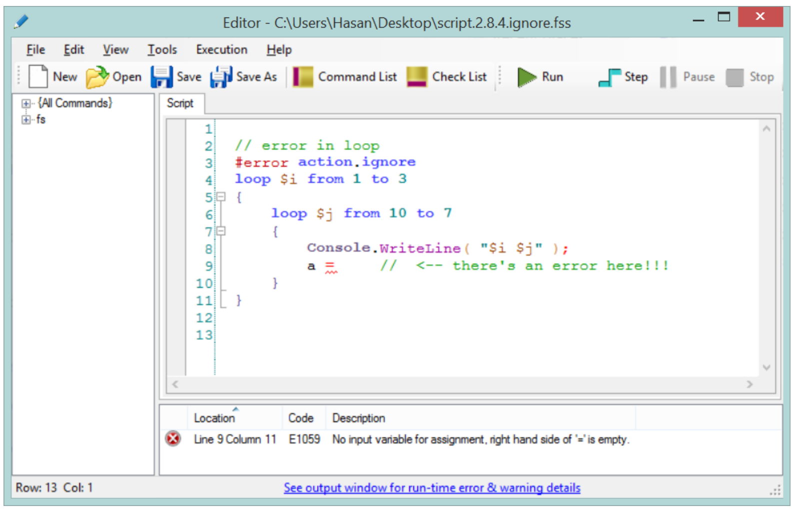
Any identified error during execution is reported on command output pane and stops execution by default. To see an example, consider the following example, in which, on line 10 in the code, a variable name (a) is entered without any assignment value. When the code is executed, an error message is displayed on command output pane with the type and location of error and the execution is stopped.



And, here's the command pane output, listing the error. The message includes the identified message and the script file. Note also that error location (line number and column number) is listed at the end.



To change behavior after an error, use '#error' directive and one of the following: action.pause, action.ignore or action.end



This time, execution continues after the error. However, error is still reported each time. See below:

Also, specific error and warning numbers can be incorporated, for example, for the above example, the following could be used to change the action for just this specific error by providing error number. Note that error numbers are provided at the end of the error message as (Code Exxxx) where xxxx is the number: `#error action.ignore 1059`

2.9.4.2. Warnings

Similarly, warnings generated during an execution will not stop the execution and only reported in the console and output pane. Hence, the default action for warning is ‘action.ignore’. To change behavior after an error, use ‘#warning’ directive and one of the following: action.pause, action.ignore or action.end

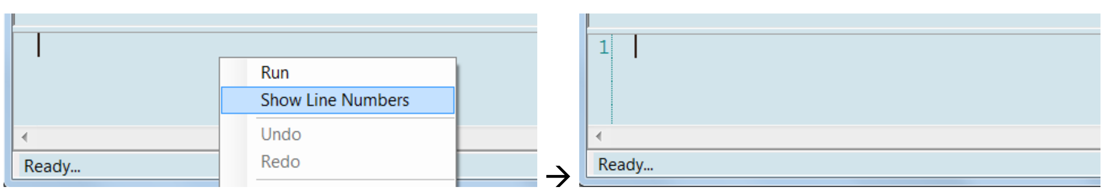
`#warning action.ignore`(default), `#warning action.pause`, `#warning action.end`
`#error action.ignore`, `#error action.pause`, `#error action.end` (default)

Description of the action keywords:

`action.ignore`	Ignores warning/error and continues the execution. This is the default action for warnings.
`action.pause`	Pauses the execution at the line where warning/error is given. See Debug section for more information
`action.end`	Ends the execution at the line where warning/error is generated. This is the default action for errors.

It is important to note that warning and error directives take into effect from and on the line they are, so multiple directives can be used at different parts of a script to have different effects at these different parts. Also, just warning numbers can be used to change default action for one or more warning types.

Note that, in editor, line numbers are also provided on the left of each line. Line numbers can also be enabled on main window command entry. To do that, right click and from drop down menu, select show line numbers.



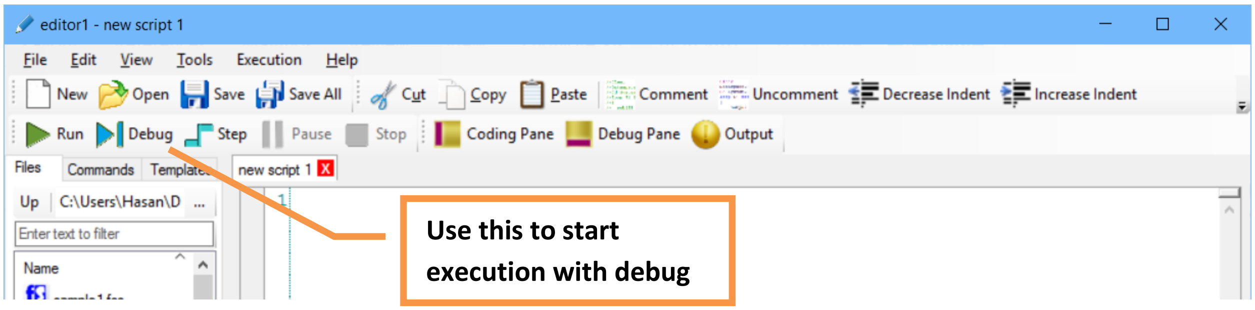
TIP

Use ‘**Delete all**’ or ‘**Dataspace.Clear**’ command to delete all variables in memory.

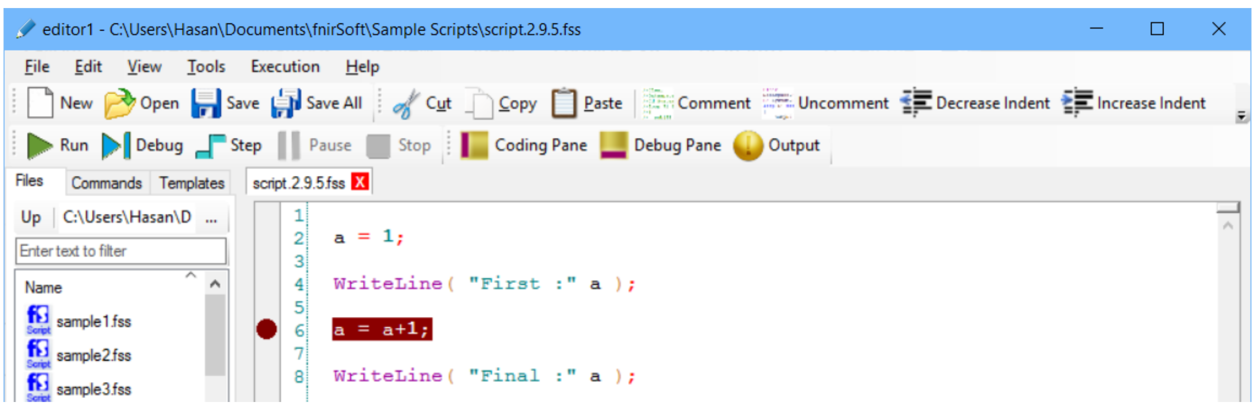
Or use “**Delete <variable name>**” to delete specific variables.

2.9.5. Debugging

When working with long script files, a way to identify problematic code sections can be challenging. Use, step to pause execution after execution of each step. When execution is paused, editor displays current execution line and all memory (variables in Dataspace) is available to investigate.

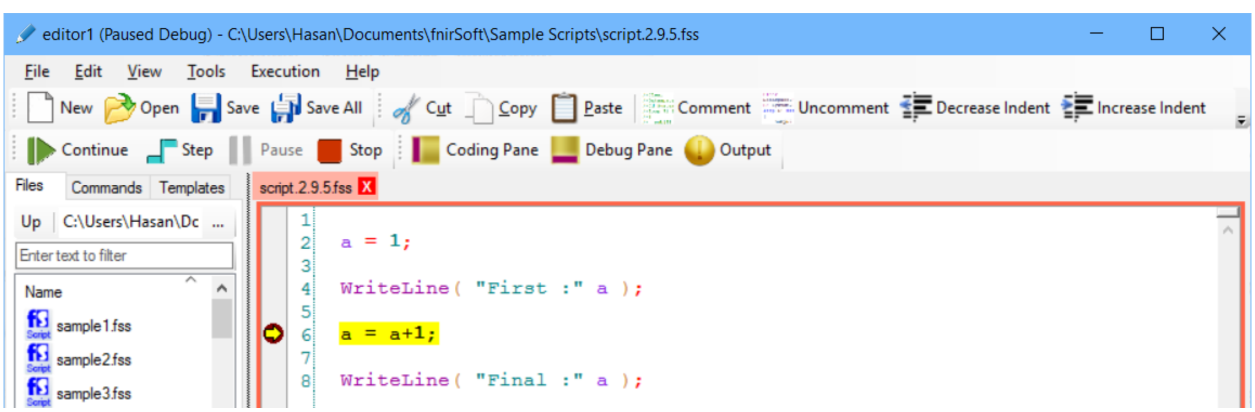


Also, break points in a script can be placed, where execution pauses if 'Debug' is used. Below in the example, line 6 is marked with a break point (a red circle before the line number). Clicking on this circle again clears the break point.

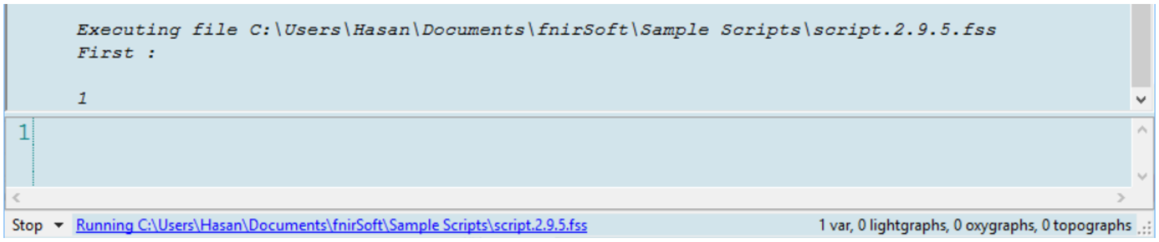


If the script is executed as is (with the break-point enabled) by clicking 'Run', execution continues until line 6 and pauses there as shown below. Stop, Step and Continue buttons are now enabled.

Hit 'Continue' to resume execution, or hit 'Stop' to end execution.



Execution can also be stopped from main window, lower left corner status button with text ‘Stop’.



2.9.6. Try/Catch/Finally

Try/Catch statements can be used to perform warning and error handling at run-time and control execution with additional branches.

The following describes the usage:

Try
{
    //Some script that might generate warning and error. Try statement is a mandatory section.
}
Catch
{
    //Script to run if and when error happens. Catch statement is mandatory if finally is not used.
}
Finally
{
    //Additional script to run <u>always</u> at the end of a try/catch statement. Finally section is optional.
}

Use `catch #error` to just catch error events, and `catch #warning` to just catch warning events. Also, specific error and warning numbers can be used to catch only specific warning or error events, for example `catch #error 1085`

See Sample scripts 2.9.6.1 through 2.9.6.6 for various examples and use cases.

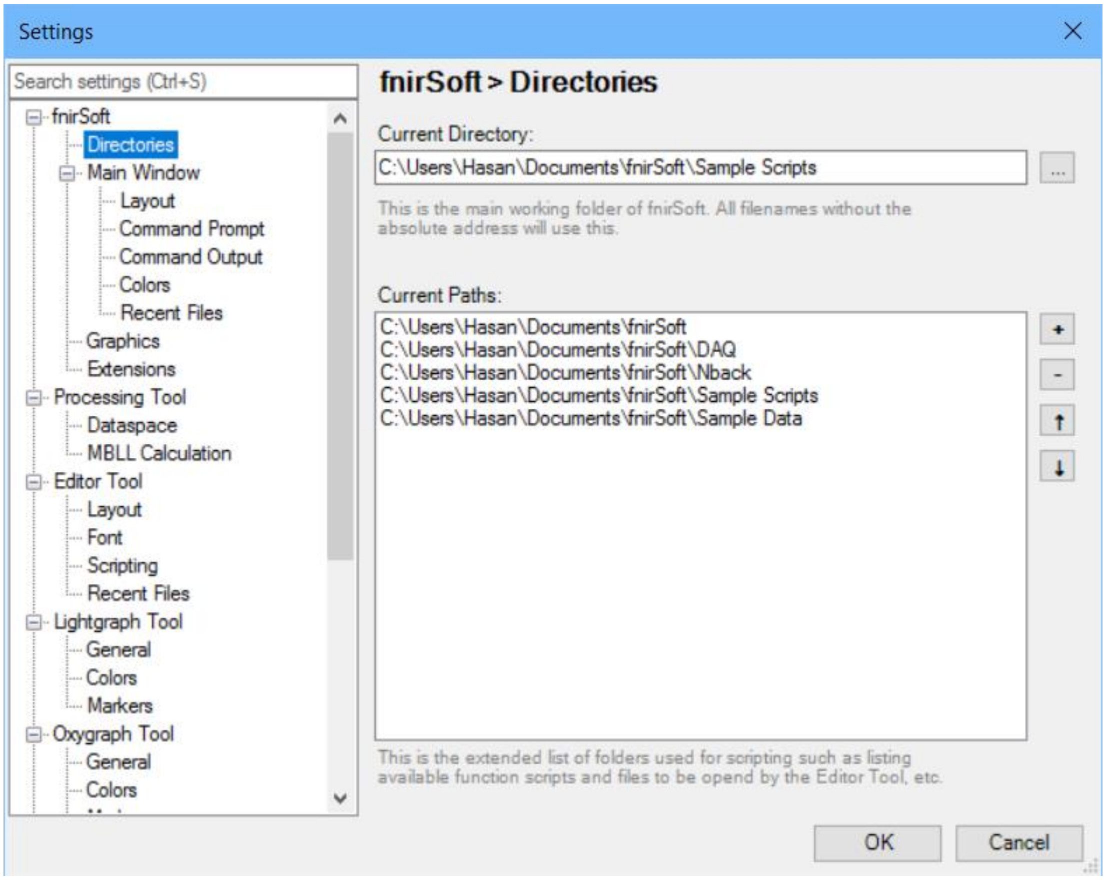
2.9.7. Current directory

Current directory is a setting that can be changed. When using commands to do file operations (read/write files) if only file names are provided, they are expected to be in the current directory.

To check current directory, use ‘CurrentDirectory’ command.

```
CurrentDirectory
```

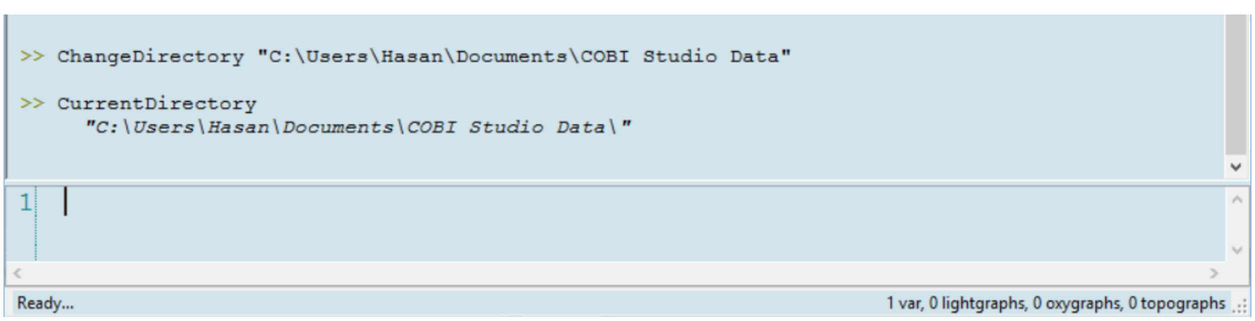
Alternatively, current directory can be checked and changed by clicking “View > Settings” from the top menu of the main window, and at the ‘Directories’ section.



Current directory can be edited or the “…” button can be used to select a folder. When a folder is saved, the setting will persist beyond a restart.

Also, current directory can be changed in scripting by using the ‘ChangeDirectory’ command.

```
ChangeDirectory "C:\Users\Hasan\Documents\COBI Studio Data"
```



2.10. Functions

Functions or subroutines can be created to keep commonly used code pieces re-used and called from multiples places without re-writing all code block.

fnirSoft will search current directory and all paths and look into fss (script) files and check if any contain function definition. If any function definition is found, it will be available for use anywhere (from main window or from other fss files).

To define a new function use the `function` keyword followed by a valid name and then use curly parenthesis to indicate function body. The following sample function returns two times of any input variable.

```
function myfunction291
{
	return Multiply(2 GetAllParameters)
}
```

Next, this function can be used anywhere within the same script.

```
function myfunction291
{
	return Multiply(2 GetAllParameters)
}

//function call
myfunction291(3 4)    //when you run this file, returns 6 and 8
```

Also, if you save this fss file in current directory or any path folder, this function can be used from

TIP

If `global` keyword is used before function keyword in the definition, that function can be called from other files and main window. To see all available global functions, type fs.Functions followed with a dot ('.') that opens the intellisense popup window.

3. Common programming/use patterns

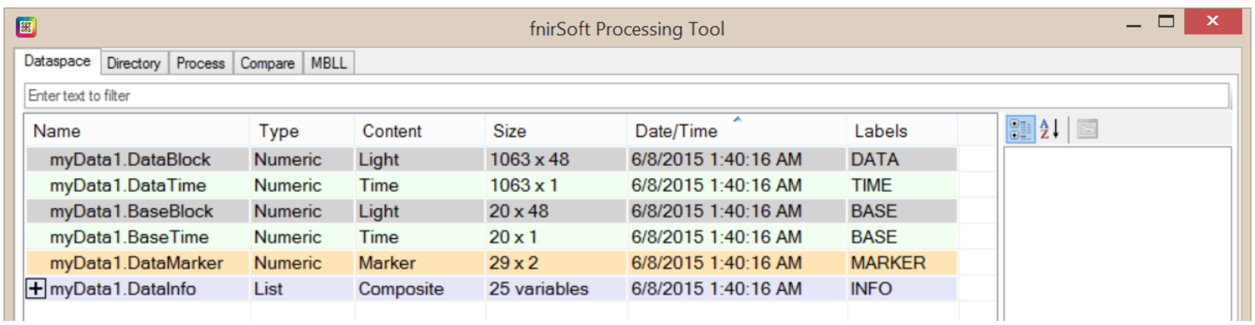
In this chapter, sample usage patterns are listed with codes.

3.1. Loading Light Intensity data (*.nir) files

This section describes loading COBI Studio light intensity data (*.nir) file to Dataspace as variables. All associated marker files are also loaded automatically, see the single line below that searches for an nir file that starts with “HA_25”. See sample script file “script.3.1.fss” for more information.

```
myData# = Load( "Sample Data\HA_25.*.nir" );
```

After the operation, the following varibles are created for each loaded file. Main data is in “#.DataBlock” variable. All marker data is loaded in “#.DataMarker” variable. Baseline information is loaded to “#BaseBlock” variable. And, time variables for Data and Baseline are also created seperately.



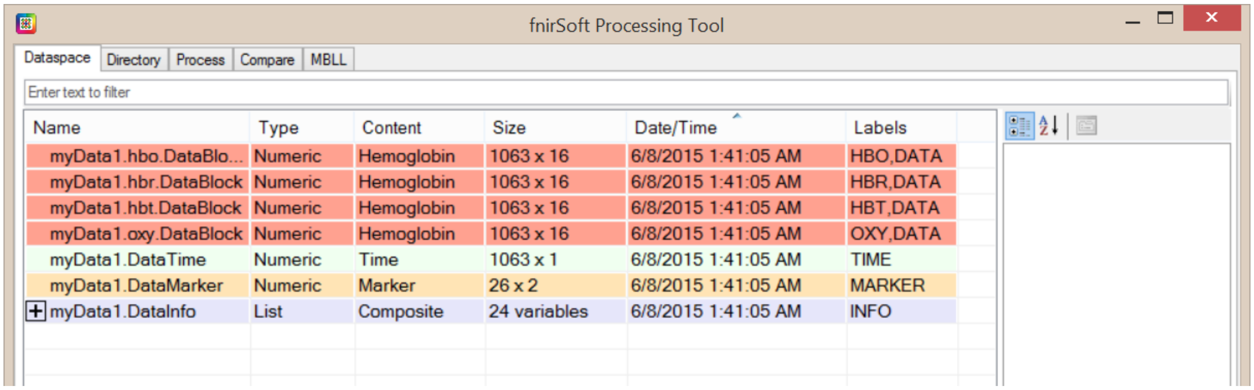
Name	Type	Content	Size	Date/Time	Labels
myData1.DataBlock	Numeric	Light	1063 x 48	6/8/2015 1:40:16 AM	DATA
myData1.DataTime	Numeric	Time	1063 x 1	6/8/2015 1:40:16 AM	TIME
myData1.BaseBlock	Numeric	Light	20 x 48	6/8/2015 1:40:16 AM	BASE
myData1.BaseTime	Numeric	Time	20 x 1	6/8/2015 1:40:16 AM	BASE
myData1.DataMarker	Numeric	Marker	29 x 2	6/8/2015 1:40:16 AM	MARKER
myData1.DataInfo	List	Composite	25 variables	6/8/2015 1:40:16 AM	INFO

3.2. Loading Oxygenation/Hemoglobin concentration changes data (*.oxy) files

This section describes loading COBI Studio oxygenation/hemoglobin concentration changes data (*.oxy) file to Dataspace as variables. All associated marker files are also loaded automatically, see the single line below that searches for an oxy file that starts with “HA_25”. See sample script file “script.3.2.fss” for more information.

```
myData# = Load( "Sample Data\HA_25.*.oxy" );
```

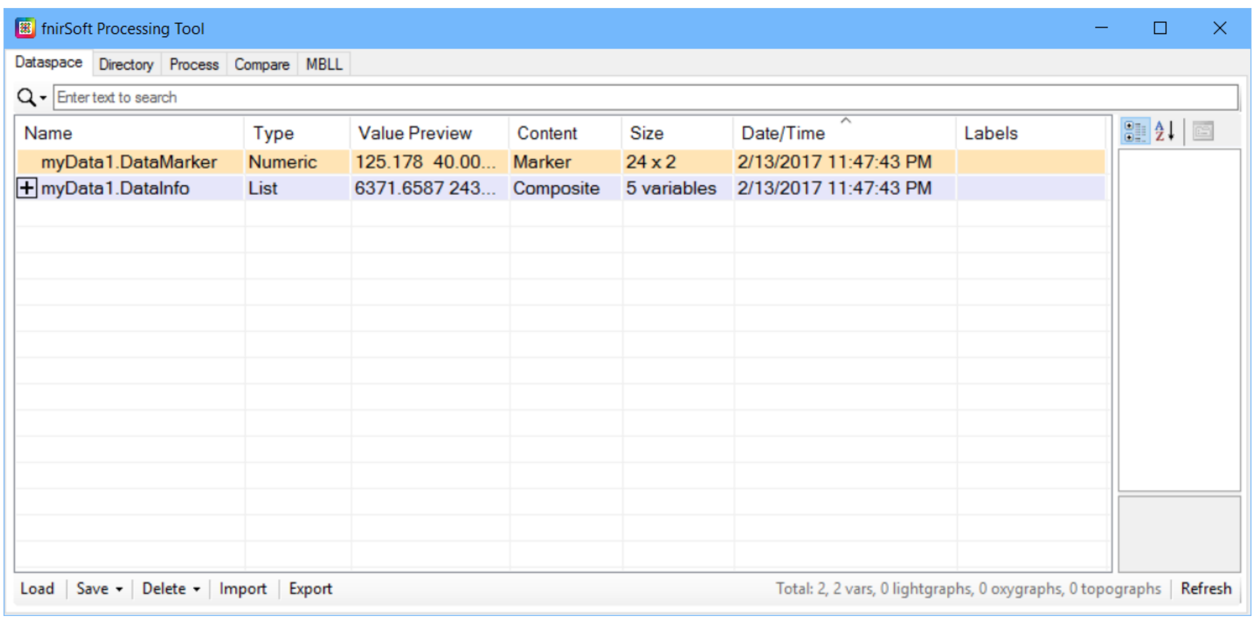
After the operation, the following varibles are created for each loaded file. Main data for all four biomarks are created in separate variables “#.Hbo.DataBlock” for oxygenated-Hemoglobin, “#.Hbr.DataBlock” for deoxyganed-Hemogblobin, “#.Hbt.DataBlock” for total-hemoglobin concentartion and “#.oxy.DataBlock” for difference in Hemoglobin, oxygenation. All marker data is loaded in “#.DataMarker” variable and time variable is loaded in Datatime variable.



3.3.Loading Marker/Event data (*.mrk) files

Marker files can also be loaded programmatically to Dataspace separately, however, it is strongly suggested that either nir or oxy data file is used to load the markers indirectly since, the start time stamp is needed to align event markers, specifically manual marker data files. See sample script file 'script.3.3.fss

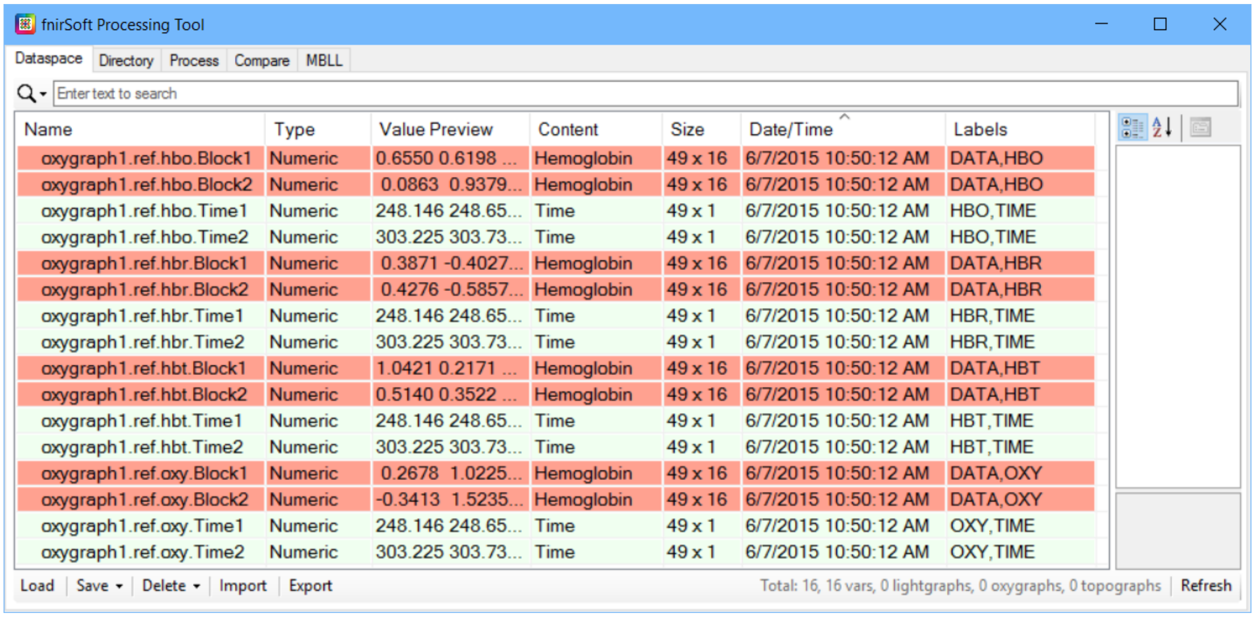
```
myData# = Load( "Sample Data\HA_25.*.mrk" );
```



3.4.Loading fnirSoft data (*.fsd) files

Fsd data files contain all types of variables and load command can be used to load them back to Dataspace. Note that there's no assignment after this as fsd variables are all loaded to Dataspace directly. The following example searches and load fsd files whose name ends with 'refined'. See sample script file 'script.3.4.fss".

```
Load( "Sample Data\.*refined.fsd" );
```

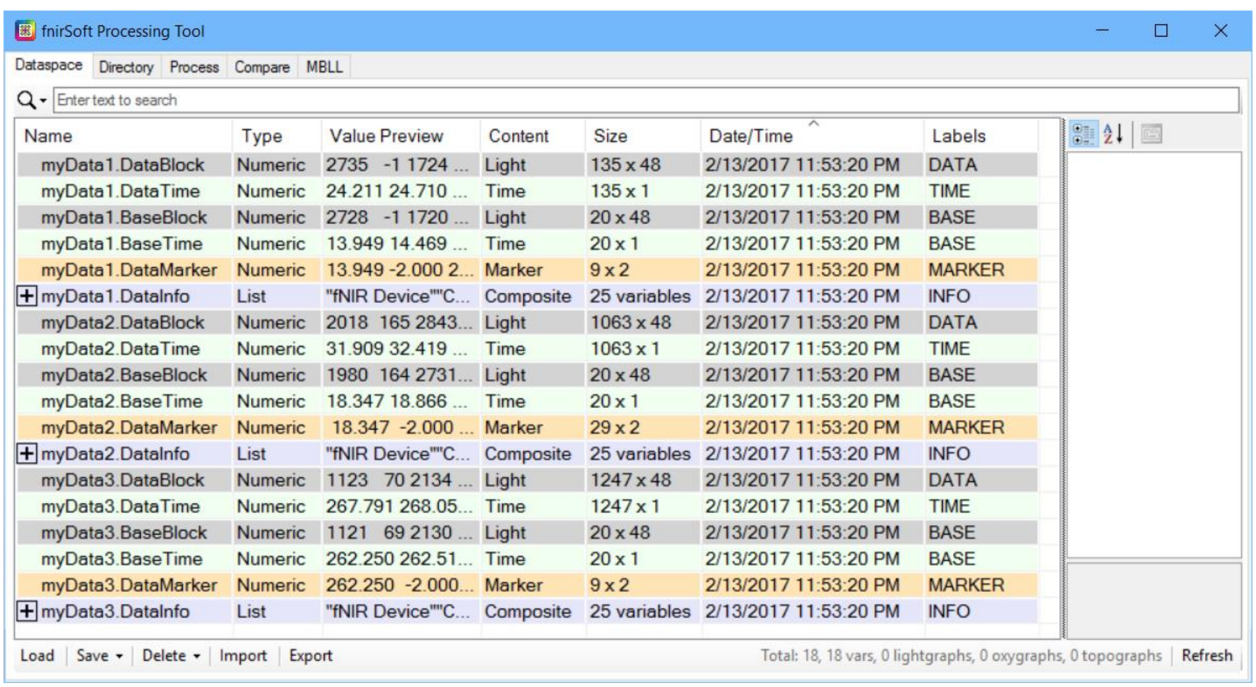


Name	Type	Value Preview	Content	Size	Date/Time	Labels
oxygraph1.ref.hbo.Block1	Numeric	0.6550 0.6198 ...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,HBO
oxygraph1.ref.hbo.Block2	Numeric	0.0863 0.9379...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,HBO
oxygraph1.ref.hbo.Time1	Numeric	248.146 248.65...	Time	49 x 1	6/7/2015 10:50:12 AM	HBO,TIME
oxygraph1.ref.hbo.Time2	Numeric	303.225 303.73...	Time	49 x 1	6/7/2015 10:50:12 AM	HBO,TIME
oxygraph1.ref.hbr.Block1	Numeric	0.3871 -0.4027...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,HBR
oxygraph1.ref.hbr.Block2	Numeric	0.4276 -0.5857...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,HBR
oxygraph1.ref.hbr.Time1	Numeric	248.146 248.65...	Time	49 x 1	6/7/2015 10:50:12 AM	HBR,TIME
oxygraph1.ref.hbr.Time2	Numeric	303.225 303.73...	Time	49 x 1	6/7/2015 10:50:12 AM	HBR,TIME
oxygraph1.ref.hbt.Block1	Numeric	1.0421 0.2171 ...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,HBT
oxygraph1.ref.hbt.Block2	Numeric	0.5140 0.3522 ...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,HBT
oxygraph1.ref.hbt.Time1	Numeric	248.146 248.65...	Time	49 x 1	6/7/2015 10:50:12 AM	HBT,TIME
oxygraph1.ref.hbt.Time2	Numeric	303.225 303.73...	Time	49 x 1	6/7/2015 10:50:12 AM	HBT,TIME
oxygraph1.ref.oxy.Block1	Numeric	0.2678 1.0225...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,OXY
oxygraph1.ref.oxy.Block2	Numeric	-0.3413 1.5235...	Hemoglobin	49 x 16	6/7/2015 10:50:12 AM	DATA,OXY
oxygraph1.ref.oxy.Time1	Numeric	248.146 248.65...	Time	49 x 1	6/7/2015 10:50:12 AM	OXY,TIME
oxygraph1.ref.oxy.Time2	Numeric	303.225 303.73...	Time	49 x 1	6/7/2015 10:50:12 AM	OXY,TIME

3.5.Loading multiple files simultaneously

Multiple files can be loaded at once by expanding the search pattern (regular expression) given as input to the load command. In the example below, any nir file that starts with "HA" is searched in the 'Sample Data' folder (within the current directory) and two files matched. Compare the number of name of created variables with those in section 3.1. See sample script file 'script.3.5.fss' for more information.

```
myData# = Load( "Sample Data\HA.*.nir" );
```



Name	Type	Value Preview	Content	Size	Date/Time	Labels
myData1.DataBlock	Numeric	2735 -1 1724 ...	Light	135 x 48	2/13/2017 11:53:20 PM	DATA
myData1.DataTime	Numeric	24.211 24.710 ...	Time	135 x 1	2/13/2017 11:53:20 PM	TIME
myData1.BaseBlock	Numeric	2728 -1 1720 ...	Light	20 x 48	2/13/2017 11:53:20 PM	BASE
myData1.BaseTime	Numeric	13.949 14.469 ...	Time	20 x 1	2/13/2017 11:53:20 PM	BASE
myData1.DataMarker	Numeric	13.949 -2.000 2...	Marker	9 x 2	2/13/2017 11:53:20 PM	MARKER
+ myData1.DataInfo	List	"fNIR Device""C...	Composite	25 variables	2/13/2017 11:53:20 PM	INFO
myData2.DataBlock	Numeric	2018 165 2843...	Light	1063 x 48	2/13/2017 11:53:20 PM	DATA
myData2.DataTime	Numeric	31.909 32.419 ...	Time	1063 x 1	2/13/2017 11:53:20 PM	TIME
myData2.BaseBlock	Numeric	1980 164 2731...	Light	20 x 48	2/13/2017 11:53:20 PM	BASE
myData2.BaseTime	Numeric	18.347 18.866 ...	Time	20 x 1	2/13/2017 11:53:20 PM	BASE
myData2.DataMarker	Numeric	18.347 -2.000 ...	Marker	29 x 2	2/13/2017 11:53:20 PM	MARKER
+ myData2.DataInfo	List	"fNIR Device""C...	Composite	25 variables	2/13/2017 11:53:20 PM	INFO
myData3.DataBlock	Numeric	1123 70 2134 ...	Light	1247 x 48	2/13/2017 11:53:20 PM	DATA
myData3.DataTime	Numeric	267.791 268.05...	Time	1247 x 1	2/13/2017 11:53:20 PM	TIME
myData3.BaseBlock	Numeric	1121 69 2130 ...	Light	20 x 48	2/13/2017 11:53:20 PM	BASE
myData3.BaseTime	Numeric	262.250 262.51...	Time	20 x 1	2/13/2017 11:53:20 PM	BASE
myData3.DataMarker	Numeric	262.250 -2.000...	Marker	9 x 2	2/13/2017 11:53:20 PM	MARKER
+ myData3.DataInfo	List	"fNIR Device""C...	Composite	25 variables	2/13/2017 11:53:20 PM	INFO

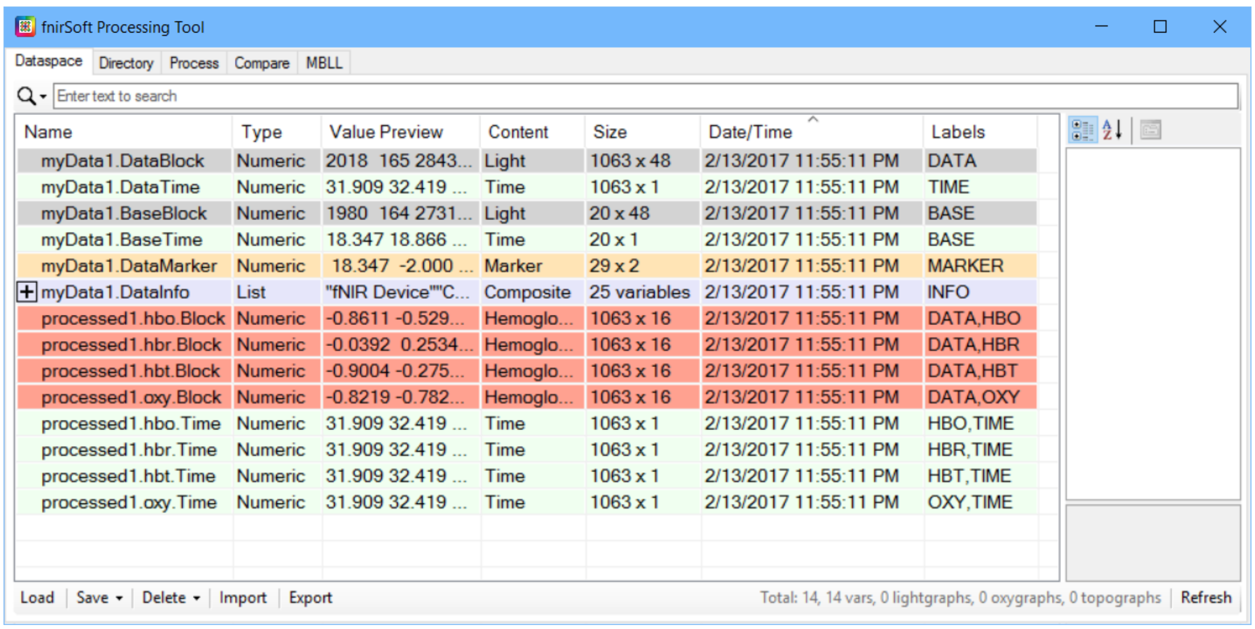
3.6.Calculating Oxygenation from Light Intensity data

To apply modified beer lambert law, use mbll command. See the command documentation by typing "help mbll" or in command explorer window under help. Below example uses a Block in Dataspace and a given baseline to calculate all four biomarkers. The baseline keyword identifies the following variable to be the baseline. But both are optional and if not used, automated baseline from the DataBlock (beginning) part is extracted and used. See sample script file 'script.3.6.fss".

```
processed# = Mbll( Find( "DataBlock") baseline Find("BaseBlock") );
```

or (Same as)

```
processed# = Mbll( myData1.DataBlock baseline myData1.BaseBlock);
```



3.7.Using Find command

The Find command can get one more of variables by name, label or type. Find command can apply search through variables in Dataspace and use string matching for name, label and/or type information to select the variables. Regular expressions (RegEx) can be used for string search. An optional parameter '`echo.commandmessages.simple`' can be added to print out a summary at command messages on the console.

The following examples use the sample data_3.2.2_refined.fsd loaded as described in section 3.4.

```
Find( "hbo"
echo.commandmessages.simple);
```

```
>> Find( "hbo" verbose);
   fs.Variable.Find:
   Numeric     oxygraph1.ref.hbo.Block1   HBO DATA
   Numeric     oxygraph1.ref.hbo.Block2   HBO DATA
   Numeric     oxygraph1.ref.hbo.Time1    HBO TIME
   Numeric     oxygraph1.ref.hbo.Time2    HBO TIME
   Found 4 matching variables
```

```
Find( "hbo.B"
echo.commandmessages.simple);
```

```
>> Find( "hbo.B" verbose);
   fs.Variable.Find:
   Numeric     oxygraph1.ref.hbo.Block1   HBO DATA
   Numeric     oxygraph1.ref.hbo.Block2   HBO DATA
   Found 2 matching variables
```

```
Find("Block1"
echo.commandmessages.simple
);
```

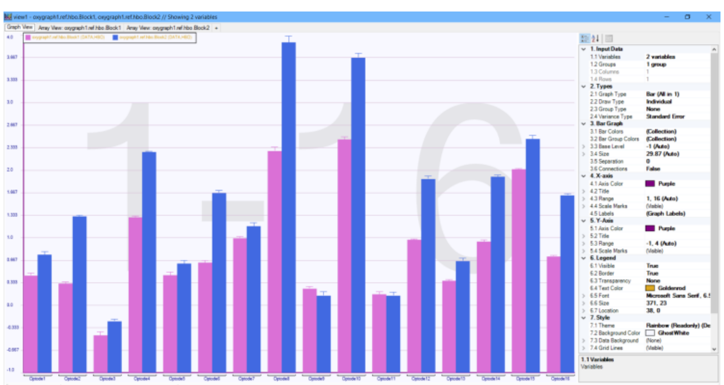
```
>> Find("Block1" verbose);
   fs.Variable.Find:
   Numeric     oxygraph1.ref.hbo.Block1   HBO DATA
   Numeric     oxygraph1.ref.hbr.Block1   HBR DATA
   Numeric     oxygraph1.ref.hbt.Block1   HBT DATA
   Numeric     oxygraph1.ref.oxy.Block1   OXY DATA
   Found 4 matching variables
```

More information about regular expression syntax can be found elsewhere[1]. In addition to name, label and type information can be searched. Use name, label or type keywords. The following string input after the keywords are the search patterns to be used for that.

TIP

Use 'View' command along with find command to display all variables together.

Executing `View Find("hbo.Block")` will display the following window:



[1] http://en.wikipedia.org/wiki/Regular_expression

3.8.Calculating Averages

A common operation is to calculate temporal and/or spatial mean values for a large number of blocks. There are multiple ways to perform the same operation and their advantages vary depending on the condition. This section demonstrates calculating the mean value across time and optodes, for oxy1.fhbo.Block0 and oxy1.fbho.Block1.

The following two methods perform the same overall operation in two different ways. The first methods simply uses find function to get all variables and perform the calculation in one line. The second method uses iterative approach to get each variable one by one. See sample script file 'script.3.8.fss".

Method 1

```
res1 = Spatial.MeanAcross( Temporal.MeanAcross(Find("hbo.Block"
echo.commandmessages.simple)))
```

```
>> res1 = Spatial.MeanAcross( Temporal.MeanAcross(Find("hbo.Block" verbose)))
     fs.Variable.Find:
     Numeric     oxygraph1.ref.hbo.Block1    HBO DATA
     Numeric     oxygraph1.ref.hbo.Block2    HBO DATA
     Found 2 matching variables
     res1 =
     1.1814
```

Method 2

```
loop $i from 1 to 2
{
  temp = Append( @temp Temporal.MeanWithin(oxygraph1.ref.hbo.Block$i ));
}

res2 = Spatial.MeanWithin( Temporal.MeanWithin( temp))
```

```
     res2 =
     1.1814
```

Note that, in the loop, we have used '@' prefix in-front of temp variable, which relieved us from initializing the variable before iteration. It merely, disregards the variable if it is the first iteration.

> **TIP**
>
> All commands that end with "**within**" means the operation is performed for each input variable. So it creates the same number of output variables as the number of input variables.
>
> All commands that end with "**across**" means the operation is performed using all input variables at once and creates only one output variables with any number of input variables (with a few exceptions).

Otherwise, we would need to create vector of the same size (columns) and append all other variables and trim the first row (initial all zero). The ‘@’ prefix saved us from additional work.

3.9.Saving variables to fnirSoft data (*.fsd) files

fnirSoft saves all variables (numeric, string and list) to a fsd data file. These files can be loaded to memory using “load” command followed by filename (relative or absolute directory address) as parameter, as shown in section 3.4. This section discusses creating fsd files by saving variables.

All current variables in Dataspace can be saved to a file by using save command. The first parameter is the new file name. The s will create a new file “data1.fsd” in current directory. If a file already exists, it is overwritten.

```
Save "data1"
```

or

```
Save( "data1.fsd" )
```

In addition, you can only save a subset of variables instead of variables all current variables. To do that, add the names of the variables to save after the file name.

```
Save "data1.fsd" oxy1.hbo.Block1 oxy1.hbo.Block2
```

or

```
Save( "data1.fsd" Find("hbo.Block"))
```

After executing the “save” command, a brief report (file name and number of variables saved) will appear on the command output pane.

```
>> Save(  "data1.fsd" Find("hbo.Block"))
    fs.Dataspace.Save: Saved 2 variables to C:\Users\Hasan\Documents\fnirSoft\data1.fsd
```

The output filename can contain placeholders for automatic naming. Placeholders are keywords within curly parenthesis such as {name}. During execution the following replacements will be done if any placeholders are present based on first input variable:

{name} to be replaced 'input variable name'
{index} to be replaced with order/index number of the input variable within the set of all inputs
{count} to be replaced with the number of all input variables
{type} to be replaced with the variable type such as Numeric, String or List
{size} to be replaced with variable size such as row x col that is height x width for numeric (array)
{label} to be replaced with available labels of each input variable
{width} to be replaced with number of columns in the input variables
{height} to be replaced with the number of rows in the input variables
{date} to be replaced with current date/time during export
{time}, {year}, {month}, {day}, {hour}, {minute}, {second} similar to date but only respective parts.

The same placeholders are used for Export functions. If more than one variable is used, the first variable's properties are used for replacing placeholders as shown below.

```
Save( "data_{date}.fsd" Find("hbo.Block"))
```

```
fs.Dataspace.Save: Saved 2 variables to C:\Users\Hasan\Documents\fnirSoft\data1_2014-04-27.fsd
```

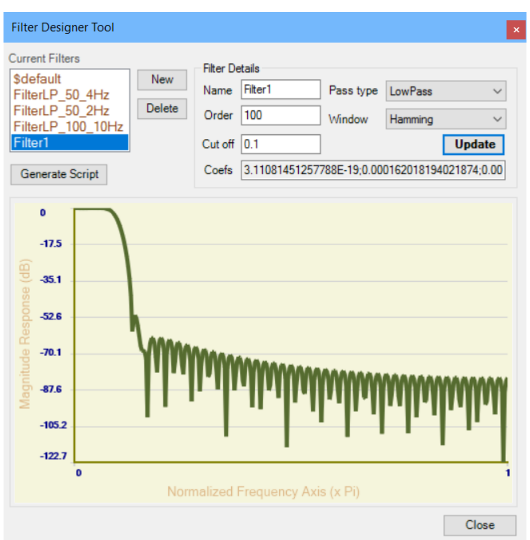
3.10. Applying FIR Filter

Use Filter command to apply FIR filter to select variables. The syntax is as follows

Filter(<variable name(s)>)

Filter(<variable name(s)> settings <filter name to use>)

To see the available filter names or design new filters type `FilterList`. Also, use `FilterDesignDialog` can be used to view, edit and create all available filters as shown below.



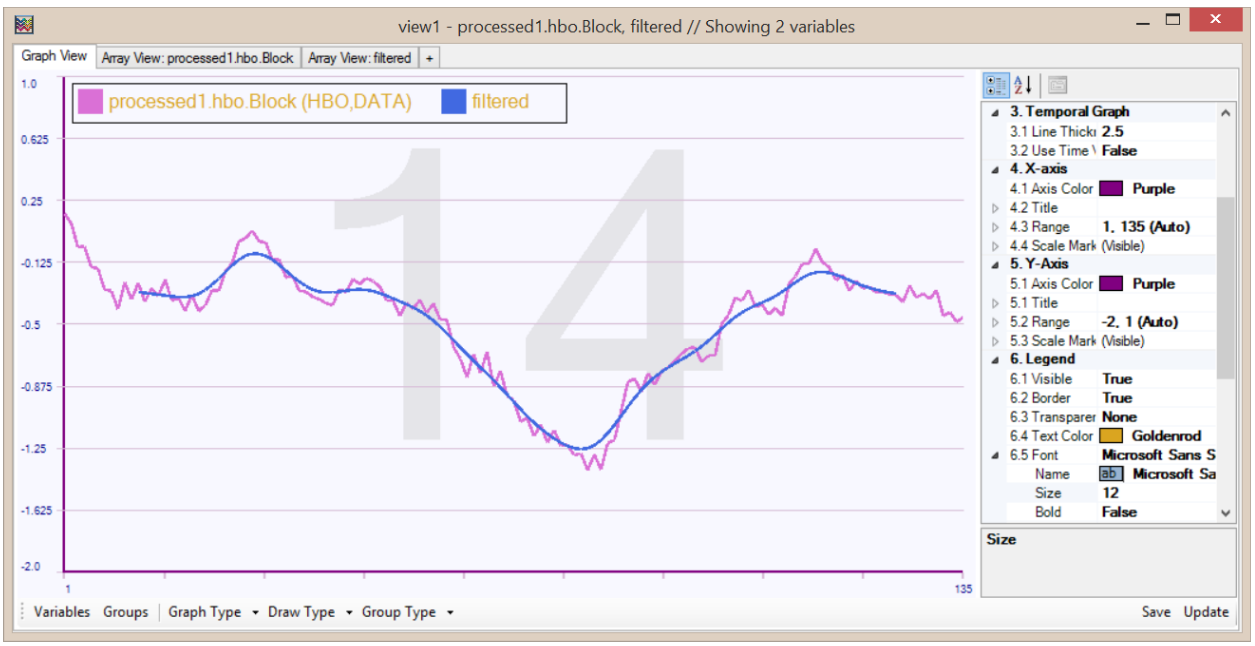
Here's the code for applying the default filter (which is a low-pass FIR filter) to the 2 variables:

```
filtered = Filter( Find("processed.*hbo.B" echo.commandmessages.simple) );
```

Same operation can be performed by specifying the filter to use, the following code performs the same operation as the one above.

```
filtered=Filter( Find("processed.*hbo.B" echo.commandmessages.simple)
settings "System1200S_2Hz");
```

A new variable is created named "filtered". Below is graphing just optode14 for the original (processed1.hbo) and its filtered version "filtered" variables. See sample script file 'script.3.10.fss'



TIP

Use '**#**' operator to define indexing within names. If it is not used, by default indexing numbers are added such as <name>_1, <name>_2. However, you can define where the numbers should be added by placing '#' within the name.

3.11. Defining Block Times

Use `DefineBlockTimes` command to identify time(s) for a given start and end pattern.

Usage is as follows:

outputvar = DefineBlockTimes (blockDefinitionVariable inputvar1 [inputvar2])

Block definition variable is a set of pattern that describes start and end times. Use `BlockDefinitionDesign` to create the block definition variable.

Use is as follows:

outputvar = BlockDefinitionDesign(keyword1 input1 keyword2 input2 [keyword3 input3] ...)

Each keyword is a specific definition followed by parameters. Below are most used keywords

starttype.markers **endtype.markers**	(for using marker values for start or end), a variable with one or more marker values follows the keyword.
starttype.fixedtime **endtype.fixedtime**	(for absolute time of start or end), a variable with the absolute time value in seconds and optionally period value follows the keyword.
starttype.relativetime **endtype.relativetime**	(for using relative time, requires the other end of block to be a fixed time or marker based): relative time value as second item.
label	(for assigning label to output block): a string variable that contains selected label follows the keyword
withintimerange	(to use only specified time range of the input variable instead of using entire variable): A variable with start and end times of the time range follows.

An example is to use marker value '45' as start and marker value '50' to end, both definitions are as follows:

```
definition1=BlockDefinitionDesign( starttype.markers 45 endtype.markers 50);
```

Here starttype and endtype are both markers and marker values follow each relevant keyword. You can check the contents of the block definition variable by typing its name at the command prompt and contents will be printed as below:

```
>> definition1
    definition1("Start Type") =
    "Markers"

    definition1("Start Markers") =
    45

    definition1("End Type") =
    "Markers"

    definition1("End Markers") =
    50

    definition1("Block Production Style") =
    "Isolated"
```

And, here's another example in which 3 successive markers indicate start the pattern. There shouldn't be any other markers in between the given successive markers.

For start, use marker values 40 45 and 90 one after another' as a single pattern for start and for end marker value '50'.

```
definition2=BlockDefinitionDesign( starttype.markers [40 45 90]
            endtype.markers [50] );
```

Note that if needed border inclusion can be enabled by using `start.includeborder` keyword. With this, start time results would be the beginning of the multi-marker pattern instead of the default inner border side. For single marker patterns this is the same. Similarly, if border inclusion is enabled for end pattern (by `end.includeborder`), then end time would be the outer border/end of multi-marker pattern. For single markers this is the same.

Another example is to use marker for start and use relative end time based on start of a block. In the following, use marker value '92' as start and end time relative to start with 40 seconds, both definitions are as follows:

```
definition3 = BlockDefinitionDesign( starttype.markers 92
                  endtype.relativetime 40 );
```

Here are executions for all pairs using the sample raw data HA_25_1_07301658.nir loaded as in section 3.1. Note that the results are identical for Pair 1 and 2, just there are more command print out.

Add `echo.commandmessages.simple` keyword to receive a report about found patterns on the console.

See sample script file 'script.3.11.fss' for all examples described in this section.

Below are use of the block definition variables.

Pair 1

```
times1=DefineBlockTimes(definition1 myData1.DataBlock)
```

```
times1 =
136.655 180.697   1.000
247.677 291.286   1.000
358.299 401.908   1.000
```

The times variable is of type blocktime, each row is another block definition and first column is start time, second column is end time. Optional last column indicates when this definition is used for extraction of data, time information along with the respective data information is also extracted.

Pair 2

```
times2=DefineBlockTimes(definition2 myData1.DataBlock
echo.commandmessages.simple)
```

```
fs.Variable.DefineBlockTimes: Start Pattern Count: 2, End Pattern Count: 6
fs.Variable.DefineBlockTimes: Run: (136.66,180.7) (358.3,401.91)  Found 2 blocks
times2 =
136.655 180.697   1.000
358.299 401.908   1.000
```

Pair 3

```
times3=DefineBlockTimes(definition3 myData1.DataBlock)
```

```
times3 =
247.677 287.677   1.000
302.972 342.972   1.000
```

3.12. Standardizing

Z-scores are one method of normalization.

The standard z-score is defined as follows:

$$z = \frac{x-\mu}{\sigma}$$

where x is a raw value to be standardized, µ is the mean of the dataset where x came from, and σ is the standard deviation of the same dataset.

Use `Temporal.ZScoresAcross` command to calculate z-scores across input variables. In the following sample, two variables (g1.oxy.Block0 and g1.oxy.Block1) are used as input

```
res = TemporalZScoresAcross( Find("hbo.B" echo.commandmessages.simple));
```

```
>> res = TemporalZScoresAcross( Find("hbo.B" echo.commandmessages.simple));
    fs.Variable.Find:
    Numeric     oxygraph1.ref.hbo.Block1   DATA HBO
    Numeric     oxygraph1.ref.hbo.Block2   DATA HBO
    Found 2 matching variables
```

res_1	Numeric	Statistics	49 x 16	6/8/201
res_2	Numeric	Statistics	49 x 16	6/8/201

Load | Save ▾ | Delete ▾ | Import | Export

3.13. Splitting Variables

Use `Temporal.Split` command to create smaller output variables from input variables. Splitting can be done either by specifying the number of output variables (each with equal size that depends on the size of the input variables) or by specifying the size of the output variables and the number of outputs depend on the size of the input variables.

Usage is as follows:

outputs = Split(splitDefinition inputVar1 [inputVar2] {inputVar3] …);

splitdefinition is a vector:

For Fixed Number of Output Blocks: [(Using-Row-Number) (Using-Column-Number)]

For Fixed Length of Output Blocks: [0 0 (Using-Row-Number) (Using-Column-Number)]

The following example creates 10 blocks out of g1.raw.Block0. Input block has 55 rows so each output block has the rounded factor that is 5 rows. Hence, the last rows has been eliminated for the split. To change that, you can use `Temporal.TrimFirst` or `Temporal.TrimLast` commands to eliminate as much rows as required.

```
res = Split( [ 10 ] myData1.DataBlock );
```

res_1	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 0 of myData1.DataBlock
res_2	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 1 of myData1.DataBlock
res_3	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 2 of myData1.DataBlock
res_4	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 3 of myData1.DataBlock
res_5	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 4 of myData1.DataBlock
res_6	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 5 of myData1.DataBlock
res_7	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 6 of myData1.DataBlock
res_8	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 7 of myData1.DataBlock
res_9	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 8 of myData1.DataBlock
res_10	Numeric	106 x 48	4/28/2014 1:14:57 ...	Split 9 of myData1.DataBlock

Load | Save | Delete ▾ | Import | Export   Count: 17 | Refresh

3.14. Exporting variables to Matlab

Use `Export2m` command to save select variables to a single matlab file. If the selected filename doesn't have an extension of ".m", it will be appended.

Export2m("<filename>.m" <variable1> <variable2> …)

```
Export2m("data.m" oxygraph1.ref.hbo.Block1 oxygraph1.ref.hbo.Block2)
```

The output filename can contain placeholders for automatic naming. Placeholders are keywords within curly parenthesis such as {name}. During execution the following replacements will be done if any placeholders are present based on first input variable:

{name} to be replaced 'input variable name'
{index} to be replaced with order/index number of the input variable within the set of all inputs
{count} to be replaced with the number of all input variables
{type} to be replaced with the variable type such as Numeric, String or List
{size} to be replaced with variable size such as row x col that is height x width for numeric (array)
{label} to be replaced with available labels of each input variable
{width} to be replaced with number of columns in the input variables
{height} to be replaced with the number of rows in the input variables
{date} to be replaced with current date/time during export
{time}, {year}, {month}, {day}, {hour}, {minute}, {second} similar to date but only respective parts.

For example, the following will include type and date info in the output filename.

```
Export2m( "data_{type}_{year}_{month}_{day}.m" oxy1.fhbo.Block1 )
```

```
>> Export2m( "data_{type}_{year}_{month}_{day}.m" oxy1.fhbo.Block1  )
    fs.Dataspace.Export2m: Exported 1 variables to C:\Users\Hasan\Documents\fnirSoft\data_Numeric_2015_02_08.m
```

Note that, if there are more than one variable, the first variables' properties will be used to replace placeholders.

3.15. Exporting variables to Text Files

Use `Export2txt` command to save select variables to a single tab separated file that can be opened by many data processing applications. If the selected filename doesn't have an extension of ".txt", it will be appended.

```
Export2txt( "data.txt" oxy1.fhbo.Block1 oxy1.fhbo.Block2 )
```

Similar to Export2m and save commands, the output filename can contain placeholders for automatic naming. Placeholders are keywords within curly parenthesis such as {name}. During execution the following replacements will be done if any placeholders are present based on first input variable:

{name} to be replaced 'input variable name'
{index} to be replaced with order/index number of the input variable within the set of all inputs
{count} to be replaced with the number of all input variables
{type} to be replaced with the variable type such as Numeric, String or List
{size} to be replaced with variable size such as row x col that is height x width for numeric (array)
{label} to be replaced with available labels of each input variable
{width} to be replaced with number of columns in the input variables
{height} to be replaced with the number of rows in the input variables
{date} to be replaced with current date/time during export
{time}, {year}, {month}, {day}, {hour}, {minute}, {second} similar to date but only respective parts.

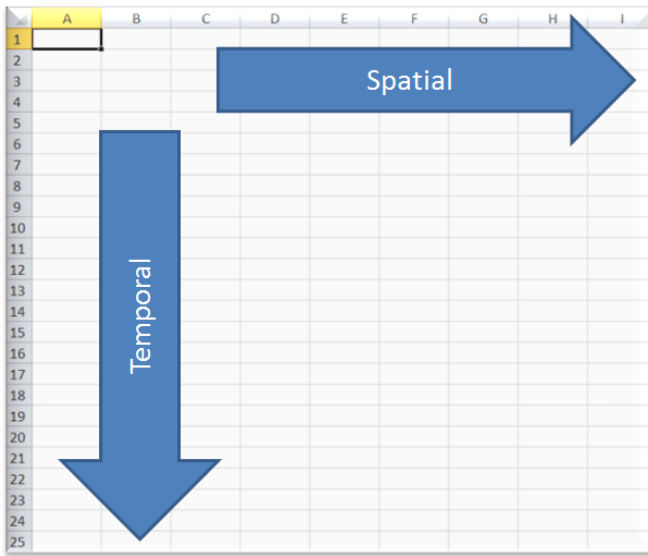
```
Export2txt( "data_{size}.txt" oxy1.fhbo.Block1  )
```

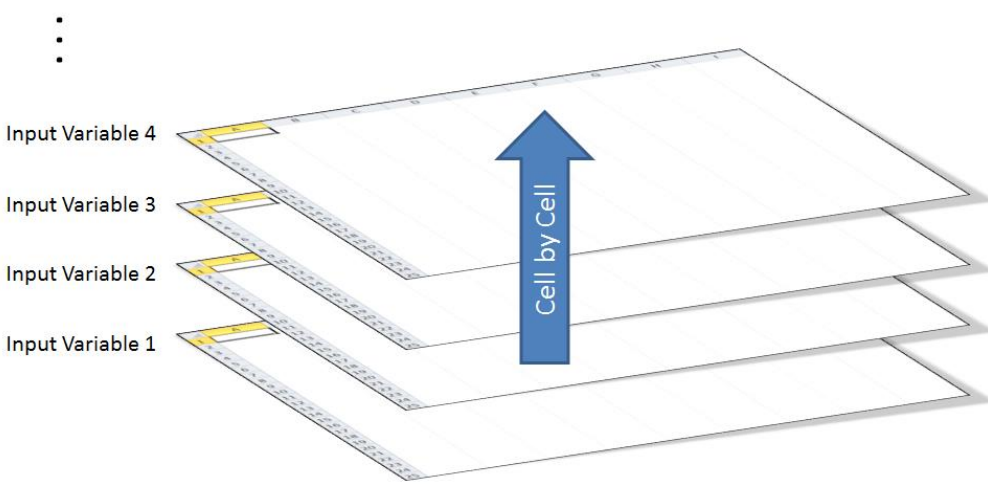
```
>> Export2txt( "data_{size}.txt" oxy1.fhbo.Block1  )
     fs.Dataspace.Export2txt: Exported 1 variable to 1 file in C:\Users\Hasan\Documents\fnirSoft
```

4. Naming Conventions

4.1 Categories

The categories of in fS command collection indicates processing features of the commands. 'Temporal', 'Spatial' or 'Math' (Cell by Cell), specify the direction of the operation. For example, Temporal.MeanWithin command, calculates means of all columns (of input variable) and the output variable has the same number of columns with the input variable. Similarly, Spatial.MeanWithin variable calculates means of all rows (of input variable) and the output variable has the same number of rows with the input variable. 'Cell by Cell' specifies use of each cell separately across multiple input variables. Below is an illustration of the direction of commands.





If the command name ends with either 'within', this indicates operation is performed on each input variable separately and if command name ends with 'across', this indicates operation is performed on all input variables together. For example, Temporal.MeanWithin calculates means on each input variable separately, and Temporal.MeanAcross calculates one global mean from all input variables.

4.2 Variable Associations

Variables in fS Dataspace can be associated or linked using variable names. Associated variables contain complementary information such as time and data variables. Linking variable names is achieved using specific strings within the names and keeping all else the same. For data variables 'Block' is the substring that variable name should contain. And for time variables, 'Time' is the substring that variable name should have. So, as an example, "a.Block" and "a.Time" would be linked. Other linked variables are marker variables and

Relation Type	Variable name should contain	Variable type should be	Variable content should be
Data	Block	Numeric	Light or Hemoglobin
Marker	Marker	Numeric	Marker
Time	Time	Numeric	Time
Info	Info	List	Composite

Various commands can be used to get the associated variable name or the associated variable itself, such as `GetDataVariable`, `GetDataVariableName`, `GetTimeVariable`, `GetTimeVariable`, `GetMarkerVariableName`, `GetMarkerVariableName`, `GetInfoVariable`, `GetInfoVariableName`.

Additional information about how these functions operate is below:

- First, identifies what kind of variable is input variable from name, type and content;
    - example: GetMarkerVariable(a.Block) -> a.Block is a Data variable
- Replaces the relation name part with the desired relation name part;
    - example: GetMarkerVariable(a.Block) -> name to search is 'a.Marker'
- If the input relation type and searched relation type are same and that is not "Data", it returns the input variable;
    - example: GetMarkerVariable(a.Marker) -> a.Marker EXIT
- If not found, searches for exatch match and returns it if found.
    - example: GetMarkerVariable(a.Block) -> a.Marker EXIT
- If not found, checks if the searched name contains a bio part (hbo,hbr,hbt,oxy); example : a.hbo.Block
    - If the input relation type and searched relation type both "Data", it searches for other bio parts;

    - example: GetDataVariable(a.hbo.Block) -> a.hbo.Block, a.hbr.Block,a.hbtBlock, a.oxy.Block EXIT
  - Otherwise searches for the variable with name searched minus bio part and returns if found it;
    - example: GetMarkerVariable(a.hbo.Block) -> a.Marker EXIT
- If not found, searches for variables whose names contain a bio part in searched name;
  - example: GetMarkerVariable(a.Block) -> a.hbo.Marker EXIT
  - example: GetDataVariable(a.Marker) -> a.hbo.Block, a.hbr.Block,a.hbtBlock, a.oxy.Block EXIT

5. Indexing

fnirSoft Script allows multiple data types variables (as matrices) and lists (sets of variables and other types). Hence, there are different types of indexing is necessary to access different elements.

5.1. Get Numeric Variable Indexing

First, let's create an array to access its cell values.

```
a = [10 20 30, 40 50 60, 70 80 90]
```

```
>> a = [10 20 30, 40 50 60, 70 80 90]
   a =
   10 20 30
   40 50 60
   70 80 90
```

Use squared parenthesis after variable name to access cells. Usage is as follows:

variable_name[row(s) , column(s)]

```
// get single cell
a[2,3]      // second row, third column
```

```
>> a[2,3]
    60
```

```
// get row(s)
a[1]   // first row
a[1 3]      // first and third rows
```

```
>> // get row(s)
   a[1]    // first row
   a[1 3]  // first and third rows
     10 20 30

     10 20 30
     70 80 90
```

```
// get column(s)
a[,1] // first column
a[,2 3]     // second and third columns
```

```
>> // get column(s)
   a[,1]   // first column
   a[,2 3] // second and third columns
     10
     40
     70

     20 30
     50 60
     80 90
```

```
// get mixed
a[1,2 3] //first row, second and third columns
a[1 3, 2]// first and third rows, second column
```

```
>> // get mixed
   a[1,2 3]     // first row, second ar
   a[1 3, 2]    // first and third rows
     20 30

     20
     80
```

```
// none existing rows/columns returns NaN
a[4, 5]
a[3 4, 3 4]
```

```
>> // none existing rows/columns returns NaN
   a[4, 5]
   a[3 4, 3 4]
     NaN

     90 NaN
     NaN NaN
```

```
// invalid indices (error)
a[0]
a[-1]
a["b"]
```

```
Error -> Array index 0 in a[0] is invalid, index should be a positive integer value. (Code E1182)
in main window: console entry, Location: Line 2 Columns 1 to 4

Error -> Array index -1 in a[-1] is invalid, index should be a positive integer value. (Code E1182)
in main window: console entry, Location: Line 1 Columns 1 to 5

Error -> Invalid value '"b"' for array index in '[]', only numeric values are allowed. (Code E1101)
in main window: console entry, Location: Line 1 Columns 3 to 5
```

5.2.Set Numeric Variable Indexing

Using the same indexing as described in previous section, numeric variable cell values can be re-assigned. Usage is as follows:

variable_name[row(s) , column(s)] = new_value

```
// set single cell
a[2,3] = 0;
a
```

```
>> a[2,3] = 0;
   a

     a =
     10 20 30
     40 50 0
     70 80 90
```

```
// reset a and set row
a[1] = 0;

// first row -> 0 NaN NaN

a[2] = [from 3 to 9 step 3, 5];
// second row -> 3 6 9

a[3] = [5, 6, 7];
// third row -> 5 NaN NaN

a
```

```
>> a[1] = 0;
   a[2] = [from 3 to 9 step 3, 5];
   a[3] = [5, 6, 7];
   a
     a =
     0 NaN NaN
     3 6 9
     5 NaN NaN
```

```
// reset a and set column
a[,1] = 0;
// first column -> 0, NaN, NaN

a[,2] = [1 2, 3 4];
// second column -> 1, 3, NaN

a[,3] = [5 6 7 8];
// third column -> 5, NaN, NaN

a
```

```
>> a[,1] = 0;
   a[,2] = [1 2, 3 4];
   a[,3] = [5 6 7 8];
   a
     a =
     0 1 5
     NaN 3 NaN
     NaN NaN NaN
```

```
// reset a and set mixed
a[1 4 2, 3 4] = [100 200, 300, 400 500];
// non existing indices expand the
// numeric variable
a
```

```
>> // reset a and set mixed
   a[1 4 2, 3 4] = [100 200, 300, 400 500];
   // non existing indices expand the numeric variable
   a

     a =
     10 20 100 200
     40 50 400 500
     70 80 90 NaN
     NaN NaN 300 NaN
```

5.3. List Variable Indexing

First, let's create a list variable to access its items

```
a = ToList (5 "test" [2 3, 4 5]);
Rename a(3) "third";
a
```

```
>> a
      a(1) =
      5

      a(2) =
      "test"

      a("third") =
      2 3
      4 5
```

Use round parenthesis after variable name to access items. Usage is as follows:

list_name(index_number) where index_number is a positive integer

list_name("index_name") where index_name is a string

```
// get var with list indices
a(1)
a("third")
a(2 "third")
a(from 1 to 3)
```

```
>> // get var with list indices
   a(1)
   a("third")
   a(2 "third")
   a(from 1 to 3)
     a(1) =
     5

     a("third") =
     2 3
     4 5

     a(2) =
     "test"

     a("third") =
     2 3
     4 5

     a(1) =
     5

     a(2) =
     "test"

     a("third") =
     2 3
     4 5
```

```
// set var with list indices
a(1) = 6
a("third") = 2 * a("third")
a(2 "third") = 5 "test2"
a(5) = 99 // expands the list to 5
//variables and fills the missing
//indices with NaN
a(from 1 to 5) = (from 2 to 10 step 2)
```

```
>> a(1) = 6
   a("third") = 2 * a("third")
   a(2 "third") = 5 "test2"
   a(5) = 99 // expands the list to 5 //v
   a(from 1 to 5) = (from 2 to 10 step 2)

     a(1) =
     6

     a("third") =
     4 6
     8 10

     a(2) =
     5

     a("third") =
     "test2"

     a(5) =
     99

     a(1) =
     2

     a(2) =
     4

     a("third") =
     6

     a(4) =
     8

     a(5) =
     10
```

```
// non existing indices warning
a(20)
a("none")
```

```
>> // non existing indices (warning)
   a(20)
   a("none")

     Warning -> List index 20 in a(20) is out of bounds of a. (Code W2079)
     in main window: console entry, Location: Line 2 Columns 1 to 5

     NaN

     Warning -> List index none in a("none") is out of bounds of a. (Code W2080)
     in main window: console entry, Location: Line 3 Columns 1 to 9

     NaN
```

```
// invalid indices (error)
a(-1)
a(0)
a("")
```

```
Error -> List index -1 in a(-1) is invalid, numeric index should be a positive integer value. (Code E1178)
in main window: console entry, Location: Line 2 Columns 2 to 5
```

```
Error -> List index 0 in a(0) is invalid, numeric index should be a positive integer value. (Code E1178)
in main window: console entry, Location: Line 1 Columns 2 to 4
```

```
Error -> List index "" in a("") is invalid, string index should not be empty. (Code E1179)
in main window: console entry, Location: Line 1 Columns 2 to 5
```

6. List of Commands

Here's list of current variables classified by the type. To get more information about these command, type their name in the command prompt and hit enter. Description and usage will appear in the command output pane. Or, use Command HelpExplorer that is accessible from main window, top menu, under help>command help explorer.

fs

Console

Clear
Echo
ReadNumeric
ReadNumericMultiple
ReadString
Write
WriteLine

DAQ

BaseStation
Remote

Dataspace

AddLabel
Clear
CopyInfo
CopyLabels
Delete
Export2acq
Export2csv
Export2img
Export2m
Export2txt
GetDataVariable
GetDataVariableName
GetInfoVariable
GetInfoVariableName
GetLabels
GetMarkerVariable
GetMarkerVariableName
GetTimeVariable
GetTimeVariableName
GetVariable
Hide
ImportData
Lightgraphs
List
Load
New

Oxygraphs
RemoveAllLabels
RemoveLabel
Rename
RenameLabel
Report
Save
SetNameTemplate
Show
Topographs
VariableExists
Variables

DateTime

CurrentDateTime
CurrentDay
CurrentDayOfYear
CurrentHour
CurrentMillisecond
CurrentMinute
CurrentMonth
CurrentSecond
CurrentTicks
CurrentYear

Execution

Error
LastResult
Run
RunDebug
RunStep
Warning

Functions

CurrentFunctionName
GetAllParameters
GetParameter
ParameterCount

Lightgraph

Channel2Optode
Load
Optode2AmbientChannel
Optode2Channel
Optode2Channels
Optode2WavelengthChannels
Organizer
Refine
Save
SaveImage

SaveLayoutImage
Show

Math

Abs
Acos
Add
Asin
Atan
Base
Ceiling
ConfidenceIntervalAcross
Cos
Cosh
CountAcross
Divide
Exp
Floor
Gamma
IsFalse
IsTrue
Log
Log10
LogGamma
MeanAcross
Multiply
Random
RandomInteger
Round
Sin
Sinh
Sqrt
StdAcross
StdErrAcross
Subtract
Tan
Truncate

Oxygraph

Load
Refine
Save
SaveImage
SaveLayoutImage
Show

Spatial

Append
Car
FixMissingWithin

IsSameSize
MaxAcross
MaxWithin
MeanAcross
MeanWithin
MedianWithin
MinAcross
MinWithin
Reject
SampleStdWithin
SizeAcross
SizeWithin
SlopeWithin
SortWithin
StdAcross
StdWithin
SumAcross
SumWithin
Trim
TrimFirst
TrimLast

System

ChangeDirectory
CopyDirectory
CopyFile
CreateDirectory
CurrentDirectory
CurrentScriptFileDirectory
DefaultCobiDirectory
DefaultDaqDirectory
DefaultDirectory
DeleteDirectory
DeleteFile
DirectoryContents
DirectoryNames
FileExists
FileNames
LatestFile
MoveDirectory
MoveFile
ReadFile
RenameDirectory
RenameFile
Version
WriteFile

Temporal

Append
AppendIfLabelsMatch

AppendMeans
AppendMeansIfLabelsMatch
Cbsi
ConfidenceIntervalRangeWithin
ConfidenceIntervalWithin
CorrectBaseline
CorrWithin
CountWithin
DecomposeWithin
DetrendWithin
DLLAcross
DULAcross
EffectSizeDAcross
EffectSizeRAcross
ExtractWithIndex
ExtractWithTime
Filter
FilterCheck
FilterDelete
FilterDesign
FilterDesignDialog
FilterExists
FilterGet
FilterGetDefault
FilterGetDefaultName
FilterList
FilterSave
FilterUpdate
FixMissingAcross
FixMissingWithin
InterceptWithin
IsSameSize
MaxAcross
MaxWithin
Mbll
MeanAcross
MeanWithin
MedianFilter
MedianWithin
MinAcross
MinWithin
PeakIndexWithin
Reject
RejectStdAway
RemoveAmbient
RLLAcross
RULAcross
SampleDown
SampleStdAcross
SampleStdWithin
SampleUp

SizeAcross
SizeWithin
SkewnessWithin
SlopeWithin
Smar
SortWithin
Split
StdAcross
StdErrAcross
StdErrWithin
StdWithin
SumAcross
SumWithin
Trim
TrimFirst
TrimLast
TrimNanPivotWithin
TrimNanWithin
TTest
TTestPaired
ZScoresAcross
ZScoresWithin

Topograph

Load
Pause
Play
SaveBrainImage
SaveBrainVideo
SaveData
SaveImage
SaveVideo
SetRange
SetThreshold
SetView
Show
ShowBrain
ShowFrame
Stop

Utility

About
Edit
Exit
ExportDialog
Help
HelpExplorer
HistoryDialog
ImportDialog
ReadTimer

Reset
StartTimer
Wait

Variable

AppendColumns
AppendRows
Atomize
BlockDefinitionCheck
BlockDefinitionDesign
BlockDefinitionPresetDelete
BlockDefinitionPresetExists
BlockDefinitionPresetGet
BlockDefinitionPresetList
BlockDefinitionPresetSave
BlockDefinitionUpdate
ColumnCount
Count
DefineBlockTimes
Find
FindCount
GetBlockTimes
GetColumns
GetContent
GetName
GetRows
GetSameSized
GetViewLabels
IsContent
IsList
IsNumeric
IsSameSize
IsString
ListContains
ListLength
Match
Refine
RejectColumns
RejectRows
RowCount
SetContent
SetName
Size
SortWithin
SplitRows
StringContains
StringEndsWith
StringIndexOf
StringJoin
StringLastIndexOf
StringLength

StringReplace
StringSplit
StringStartsWith
StringTrim
Substring
ToList
ToNumeric
ToString
View
ViewSave