

EngLab

0.3

Generated by Doxygen 1.5.5

Sun May 31 13:01:51 2009

Contents

1	Installing EngLab	1
1.1	[Installation Steps]	1
1.2	-Installation Options-	1
2	EngLab Commands	3
2.1	Command "!"	4
2.2	Command "about"	5
2.3	Command "clear"	5
2.4	Command "delete"	5
2.5	Command "exec"	6
2.6	Command "exit"	6
2.7	Command "help"	7
2.8	Command history	7
2.9	Command "lsfunc"	7
2.10	Command "lsmem"	7
2.11	Command "quit"	8
3	EngLab Language	9
3.1	General syntax information	10
3.2	Constants in Englab	20
3.3	Operators	22
3.4	Functions	26
4	User's Englab folder	29
4.1	Recent files	30
4.2	"History" folder	30
4.3	"autoexec.eng" file	31
4.4	Libraries list (libs.conf file)	31
4.5	"initial_code" file	31

5	Englab Toolboxes	33
5.1	General Description	34
5.2	Dynamic toolboxes	34
5.3	General Description of Analog Filters Toolbox	56
5.4	inputs_of_Analog_Filters	56
5.5	circuits_of_Analog_Filters	57
5.6	Description of Low Pass Notch circuit	58
6	Englab GUI (Graphical User's Interface)	61
6.1	General Description	63
6.2	Main window's components	63
6.3	Menu bar	68
6.4	Toolbars	70
6.5	About box	71
6.6	Variable's Edit Tab	71
6.7	Information window	73
6.8	Englab Editor	74
6.9	Help-Search tool	74
7	TODO	75
7.1	TODO	76
8	Bug List	77
9	Todo List	79

Chapter 1

Installing EngLab

1.1 [Installation Steps]

These are generic installation instructions.

1. Extract EngLab's source code by using command 'tar -xzf englab-0.2a.tar.gz'
2. cd to EngLab's source directory by using command 'cd englab-0.2a'
3. Type './configure' in order to configure EngLab
4. Type 'make' to build EngLab
5. Type './widgets/englabgui' in order to run EngLab. You will see some errors since you have not installed englabgui yet. This should be considered a test and not normal operation.
6. Type 'make install' in order to install EngLab in system directories. (/usr/local/bin/englab) You must be root!

1.2 –Installation Options–

In step 3) type './configure --prefix=/usr' in order to configure Englab for installation under /usr/bin

Chapter 2

EngLab Commands

With the term "EngLab command" we refer to a command the user can issue after invoking EngLab in order to display information, view or alter settings controlling the behavior of EngLab and display documentation. All commands are defined using a single word and a number of space divided arguments.

Index:

- [Command "!"](#)
- [Command "about"](#)
- [Command "clear"](#)
- [Command "delete"](#)
- [Command "exec"](#)
- [Command "exit"](#)
- [Command "help"](#)
- [Command history](#)
- [Command "lsfunc"](#)
- [Command "lsmem"](#)
- [Command "quit"](#)

2.1 Command "!"

Displays a history of previous inputs to EngLab and allows the user to re-enter previous command or code as an input to EngLab.

Syntax:

```
! [inputId]
```

Parameters:

inputId Optional argument. Re-enter input with an ID of inputId. If argument isn't specified a list of previous inputs is displayed along with their corresponding Id's

Bug

Let "id" be the inputId of the last input given to EngLab. If you execute command "! id+1" the program falls into an infinite loop and crashes

Bug

Command "!" with no arguments doesn't display history

2.2 Command "about"

Display a list of everyone that has developed code for EngLab program and their aliases

```
about
```

2.3 Command "clear"

Allows the user to clear the output box of englabgui, the history box or both

```
clear [history|output]
```

Parameters:

history Optional argument. Clears the contents of the history box

output Optional argument. Clears the contents of the output box

If no argument is supplied both the history box and the output box are cleared

Bug

The history box is not cleared permanently. If you close and re-run englabgui the history appears again. Workaround: Remove the history folder.

See also:

history

2.4 Command "delete"

Allows the user to delete allocated variables or constants from memory

```
delete [-f] [-a] [varName] ...
```

Parameters:

-f Optional argument. Forcibly delete variables even if they are defined as constants

-a Optional argument. Delete all variables. If provided alone command has the same effect as a "delete" command with no arguments.

varName Optional and repeatable argument. Names of variables or constants that will be deleted. If you enter a constant's name you must specify the *-f* argument

Attention:

If no argument is specified **all** allocated variables will be deleted. Constants will **not** be affected. In order to delete a constant you must explicitly declare it using the force *-f* argument and the constant's name.

In order to delete all variables and constants use both the *-f* and *-a* parameters

Examples:

```
delete x
delete x y
delete
delete -f pi
delete -f -a
```

Todo

add reference to constants

Bug

Deleting a variable that isn't allocated doesn't give out a warning

Bug

Argument -f in "delete" command is not implemented yet and is ignored if specified

Bug

Argument -a in "delete" command is not implemented yet and is ignored if specified

2.5 Command "exec"

Execute code from a specified file

```
exec fileName [-v]
```

Parameters:

fileName Required argument. The relative or absolute path to the file.

-v Optional argument. Display the commands that are executed (contained in the file)

Bug

In command "exec" *fileName* and *-v* arguments are not exchangeable. If specified with a different order *-v* is interpreted as a filename.

2.6 Command "exit"

Terminates 'englab' process with an exit status of 0 or 'status'.

```
exit [status]
```

Parameters:

status Optional argument. EngLab exits with status "status". Please see the bugs section

Bug

Command Exit always exits with status 0. The "status" argument is not implemented and ignored if supplied.

2.7 Command "help"

Display documentation, index and help for commands, toolboxes or functions

```
help [command|toolbox [function]]
```

Parameters:

command Optional argument. Display documentation about command with name "command"

toolbox Optional argument. List functions in toolbox with name "toolbox"

function Optional argument. Display documentation about function "function" that exists in "toolbox"

If no input argument is specified an documentation index is displayed

Bug

Command "help toolbox functionName" crashes if "functionName" doesn't exist in toolbox "toolbox"

2.8 Command history

Displays the history of user input to EngLab

```
history
```

See also:

[Command "!"](#)

2.9 Command "lsfunc"

List all available functions

```
lsfunc
```

See also:

[p1_help](#)

2.10 Command "lsmem"

List all allocated variables and/or constants

```
lsmem [-c|-a]
```

Parameters:

- c Optional argument. Only list constants
- a Optional argument. List both variables and constants

If no argument is specified only variables are listed

Bug

Command "lsmem" -c and -a arguments have not been implemented yet and are ignored if specified

2.11 Command "quit"

Similar to exit but always exits with status 0

```
quit
```

See also:

p1_exit

Documentation info**Author:**

Harry Serenis <bobomastoras@sf.net>

Chapter 3

EngLab Language

- General syntax information
 - Symbol ; (semicolon)
 - Command blocks
 - * for
 - * while
 - * do - while
 - * if
 - Returned ans variable
- Variables
 - Variable types
 - Variable declaration
 - Assign values to a variable - Implicit Variable declaration
 - * Value assignment to variable
 - * Value assignment in a matrix variable
 - * Assign variable to another variable
 - * Typecasting in assignment
 - Constants in Englab
 - * Constant types
 - * Constants included in Englab
 - * Overloading constants
- Operators
 - Arithmetic Operators
 - Logical Operators
 - Binary Operators
 - Assignment Operators
 - Comparison Operators
 - Special Operators
- Functions
 - Inline user functions
 - Toolbox functions
 - File functions

3.1 General syntax information

EngLab has an interpreted language with a C like syntax.

3.1.1 Symbol ; (semicolon)

Such as C, EngLab commands can be terminated with a semicolon (;). The use of a semicolon is optional and controls the output of the results. If provided the result is not printed to the output stream.

3.1.2 Command blocks

In EngLag there are 4 block that can be used in order to implement standard conditional execution structures.

3.1.3 for

The block **for** allows user to repeatedly execute a number of commands, until the termination condition becomes true. The syntax of for command is the following:

```
for(initial command ; termination condition ; repeatable command)
{
#commands
}
```

The first time a for loop is parsed the initial command is executed. This command is only executed once and is usually used to initialize variables that will be used inside the loop. After that the commands inside the for block are executed sequentially. If there is only one command to be executed the brackets ({,}) can be omitted.

3.1.4 while

The block **while** allows users to repeatedly execute commands, as long as a certain recursion condition remains true. The syntax of **while** command is the following:

```
while(repetition condition)
{
#commands
}
```

The recursion condition is checked, when the block is initially parsed. The command is executed and if the result is non-zero the commands within the loop are executed. When all commands have been executed, the recursion condition command is evaluated again. If it remains non-zero, we have another repetition otherwise the program steps out of the while block. If there is only one command to be executed the brackets ({,}) can be omitted.

Example :

```
while(k == 3 && i < 10)
{
#commands
}
```

In this example, the block of commands will be executed, as long as k equals to 3 and i is smaller than 10. If at the start k was equal with 4, the program would not step into the while block. Of course, in the recursion condition, the user can place whichever command the user wants. Recursions will stop, when the condition will have as a result **0**.

Bug

The while block is not implemented yet.

3.1.5 do - while

The **do - while** block is similar to the [while](#) block. The only difference is, that the recursion condition is checked in the end of the block of the commands. That is, there is at least one recursion, and then the recursion condition is checked. The syntax of **do - while** command is

```
do
{
#commands
}while(repetition condition) ;
```

The repetition condition has a similar usage with the repetition condition in a [while](#) block. The only difference is that the condition is evaluated after the commands inside the block are executed. If there is only one command to be executed the brackets ({,}) can be omitted.

Note:

The commands inside a do while block are always executed at least once

Bug

The do while block is not implemented yet.

Example :

```
do
{
#commands
}while(t != u) ;
```

The commands inside this block will be executed until t becomes equal to u.

3.1.6 if

The if block can be used by the user to conditionally execute a number commands. After the commands have been executed the remaining if block is skipped. If the if condition is not true then the interpreter will check any conditions specified by an else if block sequentially and execute the corresponding commands. Again after a set of commands is executed the rest of the if block is skipped. Finally if no condition has been met the commands inside the else block are executed. The else if clause is optional and repeatable. The else clause is optional but not repeatable.

Example :

```
if(k*i < 1000)
{
#commands
}
```

In this example, when $k*i$ becomes bigger than 1000, the commands inside the block will be executed.

The commands inside the block will be executed if the condition will return a non-zero result. If it doesn't, the block of commands will be ignored.

Example :

```
if(0 < i && i <= 10)
{
#commands1
}
else if(10 < i && i <= 20)
{
#commands2
}
else if(20 < i && i <= 30)
{
#commands3
}
else
{
#commands4
}
```

In this example, if i lies in space $(0,10]$ commands1 are executed, else if it lies in space $(10,20]$ then commands2 are executed, else if it lies in space $(20,30]$ commands3 are executed. If it doesn't lie in any of these spaces, then commands4 are executed.

Bug

The if-else block is not implemented yet.

3.1.7 Returned ans variable

Any operation that is conducted in Englab has a result. If this result **is not assigned in a user-defined variable, the result is stored in variable ans**. When two consecutive operations are conducted, the first result will be lost and substituted by the result of the second operation. A user should avoid using a variable named ans since any data stored in that variable will be lost if a command that doesn't specify an assignment variable is executed. When a .eng file is executed, the result of the final command is also saved in the ans variable and therefore can be used for error handling. Of course, . ans variable has no difference from the variables defined from the user and therefore can be used in operations.

Examples:

```
>> 1+2
ans =
3

>> a=1+2
a =
3
//Result is stored in variable a, so ans still equals 3
>> ans=5
ans =
5
```

3.1.8 Variables

In Englab, all variables are scalar (matrixes). A simple variable is declared as a 1x1 vector. There is not to limitation to the number of dimensions a matrix may have. The only limitations that exist are the total amount of memory available in the host system, and that the size of each dimension must be smaller than the maximum value of an unsigned long int of the host architecture.

Englab, supports 16 variable types. These data types can be categorized in three groups.

-Integral variables -Floating point variables -Complex variables

Variables can be declared explicitly or implicitly. Explicit declaration is done by using a specific keyword. The user can assign a value to a variable that has not been defined yet. Depending on the the type of the value (integral, floating point, complex) the variable is defined and assigned the correct value. This is called implicit declaration. When defining a variable in such a way, a default type will be used. It is recommended that users define variables explicitly before using them in order to control the data type the wish to use.

Example:

```
>> a=3;
>> b=2.7;
>> c=1+2.5*i;
>> signed long int d;
>> lsmem
Name Size Data Type
-----
a 1x1 signed long int
b 1x1 double
c 1x1 complex<float>
d 1x1 signed long int
```

See also:

[Variable types](#)
[Assign values to a variable - Implicit Variable declaration](#)

Note:

In future releases non-scalar variables will be supported.

3.1.9 Variable types

Englab support most C data types:

```
- bool
- unsigned char
- signed char
- unsigned short int
- signed short int
- unsigned long int
- signed long int
- float
- double
- long double
- complex<float>
- complex<double>
- complex<long double>
```

All data types have a direct correspondence to C types. Therefore most of them are architecture specific.

Data type **unsigned char** declares a variable, whose elements have a size of one byte and are interpreted as unsigned values, therefore their range is [0,255].

Data type **signed char** declares a variable, whose elements have a size of one byte and are interpreted as unsigned values, therefore their range is [-128,127].

All char variables when printed in the output stream are interpreted as characters.

```
>> char a=48;
```

```
>> lsmem
Name Size Data Type
-----
a 1x1 signed char
>> a
ans =
0
//ASCII character 48 is '0'
```

Data type **unsigned short int** declares a variable, whose elements are short integers, interpreted as unsigned values. This type is architecture dependent. It usually uses 2 bytes to represent a variable and therefore their range is $[0, 2^{16}-1]$

Data type **signed short int** declares a variable, whose elements are short integers, interpreted as signed values. This type is architecture dependent. It usually uses 2 bytes to represent a variable and therefore their range is $[-2^8, 2^8-1]$

Data type **unsigned long int** declares a variable, whose elements are long integers, interpreted as unsigned values. This type is architecture dependent. It usually uses 4 bytes to represent a variable and therefore their range is $[0, 2^{32}-1]$

Data type **signed long int** declares a variable, whose elements are short integers, interpreted as signed values. This type is architecture dependent. It usually uses 4 bytes to represent a variable and therefore their range is $[-2^{16}, 2^{16}-1]$

Data type **float** declares a variable, whose elements are single precision floating point variables. This type is architecture dependent. It usually uses 4 bytes to represent a floating point variable with range $[-3.4e-38, 3.4e+38]$ (7 digits mantissa)

Data type **double** declares a variable, whose elements are double precision floating point variables. This type is architecture dependent. It usually uses 8 bytes to represent a floating point variable with range $[-1.7e-380, 1.7e+380]$ (15 digits mantissa)

Data type **long double** declares a variable, whose elements are long double precision floating point variables. This type is architecture dependent.

As www.cplusplus.com states:

The values of the Size and Range depend on the system the program is compiled for. The values shown above are those found on most 32-bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one "word") and the four integer types char, short, int and long must each one be at least as large as the one preceding it, with char being always 1 byte in size. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

3.1.10 Matrix syntax and access

A matrix is defined by using brackets ([.]). The elements of each dimension are enclosed between brackets, as a comma separated list. A multidimensional matrix is a matrix whose elements are matrixes.

The general definition of a table is

```
first dimension
|      last dimension
|      |
V      V
[ ... [value0, ...], ... ]
```

Example:

```
>> [1,2,3]
ans =
1 2 3

>> [[1,2,3],[4,5,6]]
ans =
1 2 3
4 5 6

>> [[2,3],[4,5]],[[2,3],[4,5]]
ans =
table:0
2 3
4 5
table:1
2 3
4 5
```

In order to access an element of the table you can provide a comma separated list of indexes inside brackets ((),)

```
varName[index0, ...]
```

In EngLab all indexes start from zero (0) as in C.

Examples :

```
>> int a[2,3]=[[1,2,3],[4,5,6]]
a =
1 2 3
4 5 6

>> lsmem
Name Size Data Type
-----
a 2x3 signed long int
>> a[1,2]
ans =
6
```

3.1.11 Variable declaration

Explicit variable declaration can be done by using a keyword defining a data type, the variable name and an optional value assignment.

```
datatype varName=[value];
```

Parameters:

datatype A keyword used to define an EngLab data type. Can be one of "bool", "int", "signed int", "long int", "signed long int", "long", "float", "double", "unsigned char", "char", "signed char", "unsigned int", "unsigned long int", "unsigned long", "short int", "signed short int", "short", "signed short", "unsigned short int", "unsigned short", "long double", "complex", "float complex", "double complex", "long double complex"

varName The name of the variable to define. The variable must not be the same as an [EngLab Commands](#), and can only contain characters [a-z,A-Z,0-9,_]. A variable name must start cannot start with an underscore '_'

value The value the variable must be initialize to. If not provided variables are initialized to zero.

Declaring a matrix can be done with the following syntax:

```
datatype varName[size0,size1, ...];
```

Example :

```
>> int a[2,3,4]
a =
table:0
0 0 0 0
0 0 0 0
0 0 0 0
table:1
0 0 0 0
0 0 0 0
0 0 0 0
>> lsmem
Name Size Data Type
-----
a 2x3x4 signed long int
```

You can explicit declare a variable and initialize it by providing values after an equal operator (=)

```
datatype varName[size0,size1, ...]=[ ... [value0, ...], ... ];
```

Example:

```
>> int a[2,3]=[ [1,2,3], [4,5,6] ]
a =
1 2 3
4 5 6
>> lsmem
Name Size Data Type
-----
a 2x3 signed long int
```

The user can declare multiple variables in a single line, if they have the same type as a list of comma separated variable names.

```
dataType varName1,varName2,...
```

Example:

```
>> int a,b,c
>> lsmem
Name Size Data Type
-----
a 1x1 signed long int
b 1x1 signed long int
c 1x1 signed long int
```

Multiple variable declaration can combined with an assignment.

```
dataType varName1[size0,size1, ...]=[ ... [value0, ...], ... ],varName2[size0,size1, ...]=[ ... [value0, ...], ... ];
```

Example:

```
>> int a,b,c
>> lsmem
Name Size Data Type
-----
a 1x1 signed long int
b 1x1 signed long int
c 1x1 signed long int
>> delete
>> int a[2,3]=[1,2,3],[4,5,6]],b,c=10
>> lsmem
Name Size Data Type
-----
a 2x3 signed long int
b 1x1 signed long int
c 1x1 signed long int
>> a
ans =
1 2 3
4 5 6

>> c
ans =
10

>> b
ans =
0
```

3.1.12 Assign values to a variable - Implicit Variable declaration

In this section variable assignment will be described. Variable assignment includes simple variable assignment, multidimensional variable assignment and declaration with assignment (Implicit declaration)

3.1.12.1 Value assignment to variable

The user can:

- Assign a new value to a simple variable
- Assign a new value to an element of a matrix
- Declare a new variable by assigning a value to it

Assign a new value to an existing variable is pretty straightforward and can be explained with an example.

```
>> int a
a =
0

>> a=2
a =
2
```

Assigning a new value to an element of a matrix can be done by combining an element access and an assignment.

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

>> a[0,0]=5
>> a
ans =
5 0 0
0 0 0
```

Variable declaration can be done by implicit declaration

```
>> lsmem
Name Size Data Type
-----
a 3x5 signed long int
b 1x1 double
c 1x50 signed char
>> implicit=10
implicit =
10

>> lsmem
Name Size Data Type
-----
a 3x5 signed long int
b 1x1 double
c 1x50 signed char
implicit 1x1 signed long int
```

If implicit declaration is used there is no way the user can control the type of the define variable. The type is one of the three defaults for each variable group type (integral, floating point, complex)

The defaults are:

signed long int for integral variables double for floating point variables complex<float> for complex variables

3.1.12.2 Value assignment in a matrix variable

In order to assign a values to all of the elements of a matrix you have to use a syntax that was already defined in variable explicit declaration with assignment.

```
varName=[ ... [value0, ...], ... ];
```

Example:

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

>> a=[[1,2,3],[4,5,6]]
a =
1 2 3
4 5 6
```

3.1.12.3 Assign variable to another variable

The Englab user can assign values in any variable that is stored in Englab memory. The only limitation is that the variable that is assigned to another variable must be of the same dimensions as well as sizes of dimensions. If a variable is assigned to another variable, the first variable's values are copied one by one to the second. For example, if the following code is written:

```
int a[3,4]
int b[3,4]
...
a=b
```

the b's elements will be copied to a. On the other hand if the following code is written:

```
int a[3,4]
int b[3,5]
...
a=b
```

the assignment will not be done, since the variables a and b have different dimension sizes.

3.1.12.4 Typecasting in assignment

The most important fact in assignment that user must have in mind, is that when there is an assignment to a variable, a change of type is done in the copied elements. Specifically, in the following code:

```
int a[2,2]
float b[2,2]=[[2,3.1],[1.1,7]]
a=b
```

the elements of 'b' will be copied one by one in the variable 'a', but will first be typecasted to integers, because variable 'a' is stored as integer. Conclusively, 'a' will contain [[2,3],[1,7]]. Of course not all the typecasting are allowed. The only category that results to error is when the assignment involves a complex variable. For example, in the following code:

```
int a
complex b=2+i
a=b
```

there will be an error, since a complex value cannot be assigned to an integer variable.

3.2 Constants in Englab

Englab provides a special type of variables, which is the constant. Constant variables stay permanently in Englab memory (for the current session) and cannot be deleted.

3.2.1 Constant types

The various types of constants are the following:

1. Character constants are declared like this:

```
char const a="r"
```

2. Integer constants are declared like this:

```
int const a=3
```

3. Floating point constants are declared like this:

```
float const a=3.4
```

4. Double floating point constants are declared like this:

```
double const a=3.4
```

or like this:

```
const a=4.5
```

5. Complex constants are declared like this:

```
complex const a=i-2
```

3.2.2 Constants included in Englab

The constants that are loaded when Englab starts, are the following:

- `const omega=0.56714329040978387299996866221035555`
- `const pi=3.14159265358979323846264338327950288`
- `const e=2.71828182845904523536028747135266249`
- `const gama=0.57721566490153286060651209008240243`
- `const phi=1.61803398874989484820458683436563811`
- `const true=1`
- `const false=0`
- `complex const i=complex(0,1)`

3.2.3 Overloading constants

As we have said in the previous section, constant variables cannot be erased. But what happens if the user wants to create a variable whose name is the same as one of the constants? For example what will happen in the execution of the following code?

```
pi=3
b=pi
delete pi
a=pi
delete pi
```

The answer is: In the first line the constant pi is "overloaded", meaning that a new variable is saved by the name of "pi", with integer type. Despite this, constant pi is still saved in EngLab memory, but cannot be referenced. The command in the second line assigns the value 3 to the variable 'b'.

In the third line the integer variable pi is deleted from the memory. Once again, the initial constant pi is still saved, and can be retrieved. So, in the fourth line, the value 3.14... is assigned to the variable 'a'. Finally, the last line will throw an exception, since the constant variables cannot be deleted.

3.3 Operators

EngLab is a mathematical/computational language. Therefore operators are an important part of it. To ensure correct and efficient expression parsing, EngLab strictly defines the use of operators and precedence of a wide variety of operators.

The precedence of the operators in ascending priority is the following:

1. Unary Operators

- ++
- -
- ~
- !

2. Unary Sign Operators

- +
- -

3. Multiplicative Operators

- *
- /
- %
- .*
- ./
- .%

4. Additive Operators

- +
- -

5. Shift Operators

- <<
- >>

6. Relational Operators

- <
- >
- <=
- >=

7. Equality Operators

- ==
- !=

8. Bitwise AND

- &

9. Bitwise XOR

- ^

10. Bitwise OR

- |

11. Logical AND

- &&

12. Logical OR

- ||

13. Assignment Operators

- =
- *=type
- /=
- =
- +=
- -=
- >>=
- <<=
- &=
- ^=
- |=

Operators that have the same priority are evaluated sequentially within the expression.

Another thing to consider is that EngLab has a number of different data types. When operations between different types are parsed, both operands are type casted to the larger data type. The result depends both on the operator precedence and data type priority.

The data type priority in ascending order is the following:

1. bool
2. unsigned char
3. signed char
4. unsigned short
5. signed short
6. unsigned long
7. signed long
8. float
9. double
10. long double
11. complex<float>
12. complex<double>
13. complex<long double>

3.3.1 Arithmetic Operators

Arithmetic operators are the simplest and most understandable operators in mathematics and programming. Nothing is different in EngLab. The arithmetic operators supported by EngLab are:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)
- Unary (+,-)

3.3.2 Logical Operators

Following the rules of Boolean algebra, logical operators in EngLab are defined as such:

- Logical AND (&&)
- Logical OR (||)
- Logical NOT (!)

3.3.3 Binary Operators

In similarity to the logical operators, EngLab also supports binary operators. The latter suggest bitwise logical operations to be performed between two variables/constants. To understand entirely the way binary operators work you should also take into consideration that EngLab supports multiple variable types. For more information on variable types please refer to [Variable types](#) .

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise XOR (^)
- Bitwise One's Complementary (~)
- Bitwise Left Shift (<<)
- Bitwise Right Shift (>>)

Bug

Not yet Implemented

3.3.4 Assignment Operators

This subsection deals with assignment operators. The simplest assignment operator is the equal sign (=) and is used as always.

See also:

[Value assignment to variable](#)

Apart from the equal sign (=), EngLab supports a variety of complex operators that include post-computational assignment. All operators previously discussed except for Logical operators and the arithmetic unary operator have a respective assignment operator.

- Assignment by addition (+=)
- Assignment by subtraction (-=)
- Assignment by multiplication (*=)
- Assignment by division (/=)
- Assignment by modulus (=)
- Assignment by Bitwise AND (&=)
- Assignment by Bitwise OR (|=)
- Assignment by Bitwise XOR (^=)
- Assignment by Bitwise Left Shift (<<=)
- Assignment by Bitwise Right Shift (>>=)

3.3.5 Comparison Operators

Comparison operators are used in conditional statements and repetition loops and all result in boolean expressions. The comparison operators supported by EngLab are:

- Equal to (==)
- Greater than (>)
- Greater than or equal to (>=)
- Less than (<)
- Less than or equal to (<=)
- Not equal to (!=)

Comparison operators cannot be used for complex variables.

Generally, a comparison expression only stands meaningfully within a conditional statement ([if](#)) or a loop ([for](#), [while](#) and [do - while](#)).

3.3.6 Special Operators

Last but not least EngLab uses some special operators.

- Character string (") Encloses an alphanumeric phrase. All characters between the " " are the characters of the alphanumeric expression produced.

```
>> a="test"
a =
t e s t

>> lsmem
Name Size Data Type
-----
a 1x4 signed char
```

3.4 Functions

EngLab supports three kinds of functions: the inline user functions, the toolbox functions and the file functions.

3.4.1 Inline user functions

By "inline user functions" we mean straightforward declaration of functions from the user in the command window. It has one line syntax and has a mathematical form. For example an inline user function can be

```
f(x,y) = sin(x+2*cos(y)) ;
```

3.4.2 Toolbox functions

Toolbox functions are a group of functions that have common characteristics or refer to the same scientific field. Reference of the toolbox functions can be found here : [Englab Toolboxes](#)

Toolbox functions can be implemented only in C++ code using the Toolbox interface of Englab, thus a user cannot implement them with EngLab language.

3.4.3 File functions

File functions are independent functions that exist in *.eng files. This feature has not been implemented yet.

Chapter 4

User's Englab folder

- [Recent files](#)
- ["History" folder](#)
- ["autoexec.eng" file](#)
- [Libraries list \(libs.conf file\)](#)
- ["initial_code" file](#)

The user's Englab folder contains information about the program's operation. The folder's name is '.englab' and it is installed with the program's installation in the user's home folder. The folder contains the following:

1. The file "recent_files"
2. The folder "history"
3. The file "autoexec.eng"
4. The file "libs.conf"
5. The file "initial_code"

4.1 Recent files

The file "recent_files" contains the recent files that the user has opened in the [Englab Editor](#) or executed directly from the [The recent files list](#). or the [The working directory's list](#). . A file is automatically added when opened from the editor, or executed from the recent files or the working directory box. In order to add a file, user has two options: The first option is to open it from the editor and the second to add the file's full path in the [Recent files](#) . Each file's path must be in one line. If more than one paths is in the same line, the program will malfunction.

4.2 "History" folder

The folder named "history" contains three files, which contain the five last sessions's history. Each file has a name of this type:

YYYYMMDDHHmmSS

- YYYY is the year.
- MM is the month.
- DD is the day.
- HH is the hour.
- mm is the minute.
- SS is the seconds.

For example if a session began in 23:39:18 22/5/2008 the file's name will be: 20080522233918

As said before, the "history" folder contains the last five sessions's history. When a session begins, and there are already five session files in the folder, the older is being erased and the current is saved when Englab exits.

In later versions, user will be able to define the number of sessions to be saved.

4.3 "autoexec.eng" file

The file "autoexec.eng" contains the Englab commands which are executed in the beginning of each session in order to set the constants in the memory. User can add or delete the constants which are specified in this file.

4.4 Libraries list (libs.conf file)

In this list the preferred dynamic libraries are to be denoted, in order to be loaded when Englab starts.

The libraries list has to have 4 strings per library which must be separated by a space. The first and the second are the libraries's names which are used from the program. The third is the path of the dynamic library. The fourth is the number 0, if user doesn't want the library to be loaded when executing Englab, and 1 if he does. The libraries must be denoted inside the <libs> and </libs> tags. When user installs a library, the above elements are automatically added to the libs.conf file

4.5 "initial_code" file

The file "initial_code" contains the Englab script that is going to be executed when Englab starts. So if user wants to have specific variables or constants in the program memory every time Englab begins, he can denote them in the "initial_code" file. If the file contains nothing, there is going to be no effect.

Documentation info

Author:

Manos Tsardoulias <etsardou@users.sf.net>

Chapter 5

Englab Toolboxes

- [General Description](#)
- [Dynamic toolboxes](#)
 - [CImg toolbox](#)
 - [Plot Toolbox](#)
 - [General Description of Analog Filters Toolbox](#)
 - [inputs_of_Analog_Filters](#)
 - * [Description of low pass inputs](#)
 - * [Description of band pass inputs](#)
 - * [Description of high pass inputs](#)
 - * [Description of band stop inputs](#)
 - [circuits_of_Analog_Filters](#)
 - * [Description of RC-circuit](#)
 - * [Description of Sallen-Key circuit](#)
 - * [Description of circuit Low-pass Notch](#)
 - * [Description of Low-pass Notch_2 circuit](#)
 - * [Description of Delliyiannis-Fried circuit](#)
 - * [Description of band-pass Dellyianis-Fred\(with Q consideration\) circuit](#)
 - * [Description of High Pass Notch circuit](#)
 - * [Description of Low Pass Notch circuit](#)
 - * [Description of Notch circuit](#)
 - * [Description of High Pass Notch Fried circuit](#)
 - * [Description of Low Pass Notch Boctor circuit](#)
 - * [Description of High Pass Notch Boctor circuit](#)
 - * [Description of High Pass Notch Boctor circuit](#)

5.1 General Description

A "toolbox" is a group of functions that have common usage, or generally correspond to one scientific field. Englab contains two types of toolboxes: the static and the dynamic ones. The static toolboxes are included in the basic englab's package, and they contain basic mathematical functions. On the contrary, dynamic toolboxes are given separately from the basic Englab's installation. So, in order to be installed by the user, he (she) must download and install them separately. We choose to give the dynamic toolboxes separately, because they have dependencies over other open source libraries, libraries that user must have installed in his system.

5.2 Dynamic toolboxes

Englab's dynamic toolboxes are the following:

5.2.1 CImg toolbox

"Cimg" toolbox contains basic functions for importing, exporting and manipulating 2-dimensional images.

Dependencies: CImg toolbox was implemented with the help of the open-source libraries **CImg** and **ImageMagick**. User must have these two libraries installed to his system in order to install and use the CImg library.

Note:

: Some function's descriptions of the CImg's toolbox, are copied from the **CImg** manual.

Note:

: If an additional argument (whatever that is; proposed 0 or 1) is denoted in any of the toolbox functions, the result of the function will appear in the bottom left corner of the EnglabGUI. Exceptions are the functions: savep, saveg, resize, resizeg, showp, showg.

CImg's functions are the following, presented with alphabetical order:

- [addnoise](#)
- [addnoiseg](#)
- [blur](#)
- [blurg](#)
- [crop](#)
- [cropg](#)
- [cutimg](#)
- [cutimgg](#)
- [equalhist](#)
- [equalhistg](#)
- [loadg](#)
- [loadp](#)
- [normal](#)
- [normalg](#)
- [quant](#)
- [quantg](#)

- [resize](#)
- [resizeg](#)
- [rotate](#)
- [rotateg](#)
- [saveg](#)
- [savep](#)
- [showg](#)
- [showp](#)
- [thres](#)
- [thresg](#)

5.2.1.1 addnoise

- **Syntax:**

```
addnoise(a,b,c,[1])
```

- **Description:** Add noise to the image.
- **Input parameters:**
 - **a** must be a matrix of dimensions 3xNxM (RGB image)
 - **b** must be a number of type double. Represents the power of the noise. If $b < 0$ it corresponds to the percentage of the maximum image value.
 - **c** must be an integer number between 0 and two. It corresponds to the type of noise. Insert 0 for Gaussian, 1 for uniform and 2 for Salt and Pepper.
- **Exceptions:**
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if first input's dimensions are not 3xNxM.
 - `hException::hError` if second and third input's dimensions are not 1x1 .
- **Returns:** an noisy image with the same dimensions as "a".

5.2.1.2 addnoiseg

- **Syntax:**

```
addnoiseg(a,b,c,[1])
```

- **Description:** Similar to [addnoise](#), but "a" must be a gray-scale image (with one channel) of dimensions NxM. Can be applied to one of the RGB channels of an RGB image.

5.2.1.3 blur

- **Syntax:**

```
blur(a,b,c,[1])
```

- **Description:** Blurs the image.

- **Input parameters:**

- **a** must be a matrix of dimensions $3 \times N \times M$ (RGB image)
- **b** must be a positive number of type float. Represents the power of blurring in the X axis.
- **c** must be a positive number of type float. Represents the power of blurring in the Y axis.

- **Exceptions:**

- `hException::hError` if input argument's number is not 3.
- `hException::hError` if first input's dimensions are not $3 \times N \times M$.
- `hException::hError` if second and third input's dimensions are not 1×1 .

- **Returns:** an blurred image with the same dimensions as "a".

5.2.1.4 blurg

- **Syntax:**

```
blurg(a,b,c,[1])
```

- **Description:** Similar to [blur](#), but "a" must be a gray-scale image (with one channel) of dimensions $N \times M$. Can be applied to one of the RGB channels of an RGB image.

5.2.1.5 crop

- **Syntax:**

```
crop(a,b,c,d,e,[1])
```

- **Description:** Crops the image (Returns a rectangular section of the image).

- **Input parameters:**

- **a** must be a matrix of dimensions $3 \times N \times M$ (RGB image)
- **b** must be a positive number of type integer. Represents the top-left X pixel.
- **c** must be a positive number of type integer. Represents the top-left Y pixel.
- **d** must be a positive number of type integer. Represents the lower-right X pixel.
- **e** must be a positive number of type integer. Represents the lower-right Y pixel.

- **Exceptions:**

- `hException::hError` if input argument's number is not 3.
- `hException::hError` if first input's dimensions are not $3 \times N \times M$.
- `hException::hError` if second,third,fourth and fifth input's dimensions are not 1×1 .

- **Returns:** a rectangular piece of the initial image with dimension $(c-b) \times (e-d)$.

5.2.1.6 cropg

- **Syntax:**

```
cropg(a,b,c,d,e,[1])
```

- **Description:** Similar to [crop](#), but "a" must be a gray-scale image (with one channel) of dimensions NxM. Can be applied to one of the RGB channels of an RGB image.

5.2.1.7 cutimg

- **Syntax:**

```
cutimg(a,b,c,[1])
```

- **Description:** Cut pixel values which are smaller than b and greater than c.
- **Input parameters:**
 - a must be a matrix of dimensions 3xNxM (RGB image)
 - b must be a positive number of type unsigned char. Represents the minimum pixel value after cut.
 - c must be a positive number of type unsigned char. Represents the maximum pixel value after cut.
- **Exceptions:**
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if first input's dimensions are not 3xNxM.
 - `hException::hError` if second and third input's dimensions are not 1x1 .
 - `hException::hError` if $b > c$.
- **Returns:** a cut image with the same dimensions as "a".

5.2.1.8 cutimgg

- **Syntax:**

```
cutimgg(a,b,c,[1])
```

- **Description:** Similar to [cutimg](#), but "a" must be a gray-scale image (with one channel) of dimensions NxM. Can be applied to one of the RGB channels of an RGB image.

5.2.1.9 equalhist

- **Syntax:**

```
equalhist(a,b,c,d,[1])
```

- **Description:** Return the histogram-equalized version of the current image. The histogram equalization is a classical image processing algorithm that enhances the image contrast by expanding its histogram.

- **Input parameters:**

- **a** must be a matrix of dimensions $3 \times N \times M$ (RGB image)
- **b** must be a positive number of type int. Number of different levels of the computed histogram. For classical images, this value is 256 (default value).
- **c** must be a positive number of type unsigned char. Minimum value considered for the histogram computation. All pixel values lower than "c" won't be changed.
- **d** must be a positive number of type unsigned char. Maximum value considered for the histogram computation. All pixel values higher than "d" won't be changed.

- **Exceptions:**

- `hException::hError` if input argument's number is not 3.
- `hException::hError` if first input's dimensions are not $3 \times N \times M$.
- `hException::hError` if second, third and fourth input's dimensions are not 1×1 .
- `hException::hError` if $c > d$.

- **Returns:** an image with the same dimensions as "a" and equalized histogram.

5.2.1.10 `equalhistg`

- **Syntax:**

```
equalhistg(a, b, c, d, [1])
```

- **Description:** Similar to `equalhist`, but "a" must be a gray-scale image (with one channel) of dimensions $N \times M$. Can be applied to one of the RGB channels of an RGB image.

5.2.1.11 `loadg`

- **Syntax:**

```
loadg(a, [1])
```

- **Description:** Loads the grayscale image, defined by the path 'a'.

- **Input parameters:**

- **a** must be an $1 \times N$ character vector, which contains the path of the image to be loaded.

- **Exceptions:**

- `hException::hError` if input argument's dimensions is not $1 \times N$.
- `hException::hError` if the input path does not exist, if the file cannot be opened.

- **Returns:** 1.

5.2.1.12 loadp

- **Syntax:**

```
loadp(a, [1])
```

- **Description:** Loads the RGB image, defined by the path 'a'.
- **Input parameters:**
 - a must be an 1xN character vector, which contains the path of the image to be loaded.
- **Exceptions:**
 - `hException::hError` if input argument's dimensions is not 1xN.
 - `hException::hError` if the input path does not exist, if the file cannot be opened.
- **Returns:** 1.

5.2.1.13 normal

- **Syntax:**

```
normal(a,b,c, [1])
```

- **Description:** Linear normalization of the pixel values between b and c.
- **Input parameters:**
 - a must be must be a matrix of dimensions 3xNxM (RBG image)
 - b must be must be an integer value. Minimum pixel value after normalization.
 - c must be must be an integer value. Maximum pixel value after normalization.
- **Exceptions:**
 - `hException::hError` if input argument's dimensions is not 3xNxM.
 - `hException::hError` if b and c's dimensions are not 1x1
 - `hException::hError` if b>c
- **Returns:** a 3xNxM matrix, containing the normalized 'a' image.

5.2.1.14 normalg

- **Syntax:**

```
normalg(a,b,c, [1])
```

- **Description:** Similar to [normal](#), but "a" must be a gray-scale image (with one channel) of dimensions NxM. Can be applied to one of the RGB channels of an RGB image.

5.2.1.15 quant

- **Syntax:**

```
quant(a, b, [1])
```

- **Description:** Quantize pixel values into 'b' levels.

- **Input parameters:**

- **a** must be must be a matrix of dimensions 3xNxM (RGB image)
- **b** must be must be an integer value. Number of quantification levels

- **Exceptions:**

- `hException::hError` if input argument's dimensions is not 3xNxM.
- `hException::hError` if b's dimensions are not 1x1

- **Returns:** a 3xNxM matrix, containing the quantified 'a' image.

5.2.1.16 quantg

- **Syntax:**

```
quantg(a, b, [1])
```

- **Description:** Similar to [quant](#), but "a" must be a gray-scale image (with one channel) of dimensions NxM. Can be applied to one of the RGB channels of an RGB image.

5.2.1.17 resize

- **Syntax:**

```
resize(a, b, c, d)
```

- **Description:** Resize image 'a'.

- **Input parameters:**

- **a** must be must be a matrix of dimensions 3xNxM (RGB image)
- **b** must be must be an integer value. The new image's X dimension.
- **c** must be must be an integer value. The new image's Y dimension.
- **d** must be must be an integer value between -1 and 5. The method of interpolation
 - * -1 = no interpolation : raw memory resizing.
 - * 0 = no interpolation
 - * 1 = bloc interpolation (nearest point).
 - * 2 = moving average interpolation.
 - * 3 = linear interpolation.
 - * 4 = grid interpolation.
 - * 5 = bi-cubic interpolation.

- **Exceptions:**

- `hException::hError` if input argument's dimensions is not 3xNxM.
- `hException::hError` if b,c and d's dimensions are not 1x1

- **Returns:** a 3xbxc matrix, containing the resized 'a' image.

5.2.1.18 `resizeg`

- **Syntax:**

```
resizeg(a,b,c,d)
```

- **Description:** Similar to `resize`, but "a" must be a gray-scale image (with one channel) of dimensions $N \times M$. Can be applied to one of the RGB channels of an RGB image.

5.2.1.19 `rotate`

- **Syntax:**

```
rotate(a,b,[1])
```

- **Description:** Rotates image by an angle of 'b' degrees .
- **Input parameters:**
 - **a** must be must be a matrix of dimensions $3 \times N \times M$ (RBG image)
 - **b** must be must be a float value. Number of rotation degrees.
- **Exceptions:**
 - `hException::hError` if input argument's dimensions is not $3 \times N \times M$.
 - `hException::hError` if b's dimensions are not 1×1
- **Returns:** a $3 \times G \times G$ matrix, containing the rotated 'a' image. G is the length of the main diagonal of the initial image.

- **Note:**

: There might be an image duplication in the returned image's border.

5.2.1.20 `rotateg`

- **Syntax:**

```
rotateg(a,b,[1])
```

- **Description:** Similar to `rotate`, but "a" must be a gray-scale image (with one channel) of dimensions $N \times M$. Can be applied to one of the RGB channels of an RGB image.

5.2.1.21 `saveg`

- **Syntax:**

```
saveg(a,b)
```

- **Description:** Saves the grayscale image 'a', to the path 'b'.
- **Input parameters:**
 - **a** must be an $M \times N$ grayscale image.

- **b** must be an 1xN character vector, which contains the path of the image to be saved.

- **Exceptions:**

- `hException::hError` if **a**'s dimensions is not MxN.
- `hException::hError` if **b**'s dimensions is not 1xN.
- `hException::hError` if the type of saved image is not supported.

- **Returns:** 1.

5.2.1.22 savep

- **Syntax:**

```
savep(a,b)
```

- **Description:** Saves the RGB image 'a' to the path 'b'.

- **Input parameters:**

- **a** must be an RGB image of dimensions 3xNxM
- **b** must be an 1xN character vector, which contains the path of the image to be saved.

- **Exceptions:**

- `hException::hError` if **a**'s dimensions is not 3xNxM.
- `hException::hError` if **b**'s dimensions is not 1xN.
- `hException::hError` if the type of saved image is not supported.

- **Returns:** 1.

5.2.1.23 showg

- **Syntax:**

```
showg(a)
```

- **Description:** Shows the grayscale 'a' image. Englab is stalled until the window is closed. Used for previewing the image. User cannot open two images at the same time. Will probably be fixed in the next versions.

- **Input parameters:**

- **a** must be a grayscale image.

- **Exceptions:**

- **Returns:** 1.

5.2.1.24 showp

- **Syntax:**

```
showp(a)
```

- **Description:** Shows the RB 'a' image. Englab is stalled until the window is closed. Used for previewing the image. User cannot open two images at the same time. Will probably be fixed in the next versions.

- **Input parameters:**

- **a** must be an RGB image of dimensions 3xNxM.

- **Exceptions:**

- **Returns:** 1.

5.2.1.25 thres

- **Syntax:**

```
thres(a,b,[1])
```

- **Description:** Thresholds image a. If pixel value is under b, it's value turns to 0. If pixel value is over b, it's value turns to 1.

- **Input parameters:**

- **a** must be a matrix of dimensions 3xNxM (RBG image)
- **b** must be an unsigned char value, which represents the threshold

- **Exceptions:**

- hException::hError if a's argument's number is not 3.
- hException::hError if b's argument's number is not 2.

- **Returns:** a 3xNxM image, containing the the thresholded 'a'.

5.2.1.26 thresg

- **Syntax:**

```
thresg(a,b,[1])
```

- **Description:** Similar to [thres](#), but "a" must be a gray-scale image (with one channel) of dimensions NxM. Can be applied to one of the RGB channels of an RGB image.

5.2.2 Plot Toolbox

This toolbox implements various plotting functions. The Plot toolbox uses the MathGL library and the FLTK tool, which are its dependencies. Also another dependency is the library "pthread". The plot functions with alphabetical order are the following:

Note:

The definitions of the functions were taken from the documentation of MathGL v.1.6 and were enriched by the author. Also this documentation is old. The names of the following functions are changed.

- [Function areaplot](#)
- [Function axial](#)
- [Function beltplot](#)
- [Function boxs](#)
- [Function contour](#)
- [Function contourf](#)
- [Function density](#)
- [Function mesh](#)
- [Function plot](#)
- [Function plot3d](#)
- [Function prodist](#)
- [Function stem](#)
- [Function stem3d](#)
- [Function stepplot](#)
- [Function stepplot3d](#)
- [Function surface](#)
- [Function torus](#)
- [Plot examples](#)

5.2.2.1 Function areaplot

- Syntax: `areaplot(x)`
- Description: The function draws continuous lines between points X_i and Y_i in plane $Z=0$ and fills the area below the plot line.
- Input parameters:

- x must be of dimensions $2 \times N$ or $N \times 2$. The first dimension is the X points and the second dimension is the Y points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $2 \times N$ or $N \times 2$.
- Returns: 1
- Example:

5.2.2.2 Function axial

- Syntax: `axial(x,y,z,[u])`
- Description: The function draws surface which is result of contour plot rotation for the surface specified parametrically $\{X_i, Y_i, Z_i\}$.
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
 - u is an optional argument. It must be an integer number. It indicates the number of contours to be painted equidistantly between Z_{min} and Z_{max} . If not denoted the default is 10.
- Exceptions:
 - `hException::hError` if input argument's number is not 3 or 4.
 - `hException::hError` if the three first input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.3 Function beltplot

- Syntax: `beltplot(x,y,z)`
- Description: The function draws belts (types) for surface specified parametrically by $\{X_i, Y_i, Z_i\}$
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
- Exceptions:
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if the input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.4 Function boxes

- Syntax: `boxes(x,y,z)`
- Description: The function draws vertical boxes for surface specified parametrically by $\{X_i, Y_i, Z_i\}$
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
- Exceptions:
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if the input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.5 Function contour

- Syntax: `contour(x,y,z,[u])`
- Description: The function draws contour lines for the surface specified parametrically $\{X_i, Y_i, Z_i\}$.
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
 - u is an optional argument. It must be an integer number. It indicates the number of contours to be painted equidistantly between Z_{min} and Z_{max} . If not denoted the default is 10.
- Exceptions:
 - `hException::hError` if input argument's number is not 3 or 4.
 - `hException::hError` if the three first input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.6 Function contourf

- Syntax: `contourf(x,y,z,[u])`
- Description: The function draws solid (or filled) contours for the surface specified parametrically $\{X_i, Y_i, Z_i\}$.
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$

- z must be of dimensions $M \times N$
- u is an optional argument. It must be an integer number. It indicates the number of contours to be painted equidistantly between Z_{\min} and Z_{\max} . If not denoted the default is 10.
- Exceptions:
 - `hException::hError` if input argument's number is not 3 or 4.
 - `hException::hError` if the three first input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.7 Function density

- Syntax: `density(x,y,z)`
- Description: The function draws density plot for surface specified parametrically by $\{X_i, Y_i, Z_i\}$
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
- Exceptions:
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if the input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.8 Function mesh

- Syntax: `mesh(x,y,z)`
- Description: The function draws mesh lines for surface specified parametrically by $\{X_i, Y_i, Z_i\}$
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
- Exceptions:
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if the input argument's dimensions are not the same.
- Returns: 1
- Example:

5.2.2.9 Function plot

- Syntax: `plot(x)`
- Description: The function draws continuous lines between points X_i and Y_i in plane $Z=0$. In other words the area below the plot line gets filled with color.
- Input parameters:
 - x must be of dimensions $2 \times N$ or $N \times 2$. The first dimension is the X points and the second dimension is the Y points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $2 \times N$ or $N \times 2$.
- Returns: 1
- Example:

5.2.2.10 Function plot3d

- Syntax: `plot3d(x)`
- Description: The function draws continuous lines between points X_i, Y_i and Z_i .
- Input parameters:
 - x must be of dimensions $3 \times N$. The first dimension is the X points, the second dimension is the Y points, and the third the Z points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $3 \times N$.
- Returns: 1
- Example:

5.2.2.11 Function prodist

- Syntax: `prodist(dist,m,s)`
- Description: The function draws various probability distributions. Since now the gauss probability distribution can be plotted.
- Input parameters:
 - $dist$ is a unsigned char vector that contains the distribution's name. For now the only valid name is "gauss"
 - m is the mean of the gauss distribution.

- s is the variance of the gauss distribution.
- Exceptions:
 - `hException::hError` if input argument's number is not 3.
- Returns: 1
- Example:

5.2.2.12 Function stem

- Syntax: `stem(x)`
- Description: The function draws vertical lines from points $\{X_i, Y_i\}$ to $Y=0$.
- Input parameters:
 - x must be of dimensions $2 \times N$ or $N \times 2$. The first dimension is the X points and the second dimensions is the Y points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $2 \times N$ or $N \times 2$.
- Returns: 1
- Example:

5.2.2.13 Function stem3d

- Syntax: `stem(x)`
- Description: The function draws vertical lines from points $\{X_i, Y_i, Z_i\}$ to $Z=0$.
- Input parameters:
 - x must be of dimensions $3 \times N$. The first dimension is the X points, the second dimensions is the Y points, and the third the Z points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $3 \times N$.
- Returns: 1
- Example:

5.2.2.14 Function stepplot

- Syntax: `stepplot(x)`
- Description: The function draws continuous stairs for points $\{X_i, Y_i\}$.
- Input parameters:
 - x must be of dimensions $2 \times N$ or $N \times 2$. The first dimension is the X points and the second dimensions is the Y points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $2 \times N$ or $N \times 2$.
- Returns: 1
- Example:

5.2.2.15 Function stepplot3d

- Syntax: `stepplot3d(x)`
- Description: The function draws continuous stairs for points $\{X_i, Y_i, Z_i\}$.
- Input parameters:
 - x must be of dimensions $3 \times N$. The first dimension is the X points, the second dimensions is the Y points, and the third the Z points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $3 \times N$.
- Returns: 1
- Example:

5.2.2.16 Function surface

- Syntax: `surface(x,y,z)`
- Description: The function draws surface specified parametrically by $\{X_i, Y_i, Z_i\}$
- Input parameters:
 - x must be of dimensions $M \times N$
 - y must be of dimensions $M \times N$
 - z must be of dimensions $M \times N$
- Exceptions:
 - `hException::hError` if input argument's number is not 3.
 - `hException::hError` if the input argument's dimensions are not the same.

- Returns: 1
- Example:

5.2.2.17 Function torus

- Syntax: `stepplot(x)`
- Description: The function draws surface which is result of curve $\{r,z\}$ rotation around Z axis.
- Input parameters:
 - x must be of dimensions $2 \times N$. The first dimension is the R points and the second dimensions is the Z points.
- Exceptions:
 - `hException::hError` if input argument's number is not 1.
 - `hException::hError` if input argument's dimensions number is not $2 \times N$.
- Returns: 1
- Example:

5.2.2.18 Plot examples

Example of `areaplot`:

```
delete
float a[2,100];
int i;
for (i=1;i<100;i+=1)
{
a[0,i]=i;
a[1,i]=sin(i*0.1)+cos(0.31*i);
}
areaplot(a);
delete i
delete ans
```

Example of `axial`:

```
delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i,j]=i;
b[i,j]=j;
c[i,j]=i-j;
}
}
axial(a,b,c,3);
```

Example of beltplot:

```
delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i,j]=i-25;
b[i,j]=j-25;
c[i,j]=3-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25);
}
}
beltplot(a,b,c);
```

Example of boxes:

```
delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i,j]=i-25;
b[i,j]=j-25;
c[i,j]=4-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25)-cos(i*0.3)*sin(j*0.4)*0.5;
}
}
boxes(a,b,c);
```

Example of contourf:

```
delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i,j]=i-25;
b[i,j]=j-25;
c[i,j]=3-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25)+i/3;
}
}
contourf(a,b,c,20);
```

Example of contour:

```
delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i,j]=i-25;
```

```

b[i, j]=j-25;
c[i, j]=3-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25);
}
}
contour(a,b,c,15);

```

Example of **density**:

```

delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i, j]=i-25;
b[i, j]=j-25;
c[i, j]=4-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25)-cos(i*0.3)*sin(j*0.4)*0.5;
}
}
density(a,b,c);

```

Example of **prodist**:

```

delete
prodist("gauss",15,2);

```

Example of **prodist**:

```

delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i, j]=i-25;
b[i, j]=j-25;
c[i, j]=3-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25);
}
}
mesh(a,b,c);

```

Example of **plot3d**:

```

delete
float a[3,1000];
int i;
for (i=1;i<1000;i+=1)
{
a[0,i]=0.8*sin(0.2*pi*i)*sin(0.1*pi*i);
a[1,i]=0.8*sin(0.2*pi*i)*cos(0.1*pi*i);
a[2,i]=cos(i)*sin(i);
}
plot3d(a);

```

Example of **plot3d**:

```
delete
float a[2,100];
int i;
for (i=1;i<100;i+=1)
{
a[0,i]=i;
a[1,i]=sin(i*0.1)*cos(i*0.23);
}
plot(a);
delete i
delete ans
```

Example of stem3d:

```
delete
float a[3,1000];
int i;
for (i=1;i<1000;i+=1)
{
a[0,i]=0.1*sin(0.04*i);
a[1,i]=0.1*sqrt(i);
a[2,i]=cos(0.003*i);
}
stem3d(a);
```

Example of stem:

```
delete
float a[2,100];
int i;
for (i=1;i<100;i+=1)
{
a[0,i]=i;
a[1,i]=sin(i*0.1);
}
stem(a);
delete i
delete ans
```

Example of stepplot3d:

```
delete
float a[3,1000];
int i;
for (i=1;i<1000;i+=1)
{
a[0,i]=0.01*i;
a[1,i]=0.01*i;
a[2,i]=cos(0.003*i);
}
stepplot3d(a);
```

Example of stepplot:

```
delete
float a[2,100];
int i;
for (i=1;i<100;i+=1)
{
a[0,i]=i;
a[1,i]=sin(i*0.1);
}
```

```

}
stepplot(a);
delete i
delete ans

```

Example of **surface**:

```

delete
float a[50,50];
float b[50,50];
float c[50,50];
for(i=0;i<50;i++)
{
for(j=0;j<50;j++)
{
a[i,j]=i-25;
b[i,j]=j-25;
c[i,j]=3-0.01*(i-25)*(i-25)-0.01*(j-25)*(j-25)-cos(i*0.3)*sin(j*0.4)*0.5;
}
}
surface(a,b,c);

```

Example of **torus**:

```

delete
float a[2,10];
int i;
for (i=1;i<10;i+=1)
{
a[0,i]=cos(i);
a[1,i]=sin(i);
}
torus(a);
delete i
delete ans

```

5.3 General Description of Analog Filters Toolbox

Analog Filters toolbox contains functions that implement algorithms, such as butterworth, chebychev and inverse chebychev, in order to create circuits that meet the desired behaviour {low_pass, band_pass, high_pass, bandstop} specified by the user. The inputs of a function is specified later on the documentation. As an output you can have a n-grade circuit. Each circuit is identified by an ID, by this ID you can interpret the elements following this. ID's are negative integers, so that they can differ from circuit's elements.

5.4 inputs_of_Analog_Filters

5.4.1 Description of low pass inputs

1. amax, is the maximum decrement of the signal, in dB, for frequencies ranging from 0 to wp.
2. amin, is the minimum decrement of the signal, in dB, for frequencies ranging from ws to nan.

5.4.2 Description of band pass inputs

1. amin, is the minimum decrement of the signal, in dB, for frequencies ranging from 0 to w3 and w4 to nan.
2. amax, is the maximum decrement of the signal, in dB, for frequencies ranging from w1 to w.

5.4.3 Description of high pass inputs

1. amin, is the minimum decrement of the signal, in dB, for frequencies ranging from 0 to ws.
2. amax, is the maximum decrement of the signal, in dB, for frequencies ranging from wp to nan.

5.4.4 Description of band stop inputs

1. amax, is the maximum decrement of the signal, in dB, for frequencies ranging from 0 to w1 and w2 to nan
2. amin, is the minimum decrement of the signal, in dB, for frequencies ranging from w3 to w4.

5.5 circuits_of_Analog_Filters

5.5.1 Description of RC-circuit

Reference ID: -1 Order of values Returned: R,C

5.5.2 Description of Sallen-Key circuit

Reference ID: -2 Order of values Returned: R1,R2,C1,C2

5.5.3 Description of circuit Low-pass Notch

Reference ID: -3 Order of values Returned: R1,R2,R3,R4,R5,C1,C2

5.5.4 Description of Low-pass Notch_2 circuit

Reference ID: -4 Order of values Returned: R1,R2,R3,R4,C1,C2

5.5.5 Description of Delliyiannis-Fried circuit

Reference ID: -5 Order of values Returned: R1,R2,C1,C2

5.5.6 Description of Dellyianis-Fred With Gain Reconfiguration Circuit

Reference ID: -51 Order of values Returned: R1(used internally),R2,RA,RB,C1,C2

5.5.7 Description of band-pass Dellyianis-Fred(with Q consideration) circuit

Reference ID: -6 Order of values Returned: R1(used internally),R2,Ra,Rb,Z22,Z23,C1,C2

5.5.8 Description of High Pass Notch circuit

Reference ID: -7 Order of values Returned: R1,R2,R3,R4,C

5.6 Description of Low Pass Notch circuit

Reference ID: -8 Order of values Returned: R1,R2,R3,R4,R5,C,C

5.6.1 Description of Notch circuit

Reference ID: -9 Order of values Returned: R1,R2,R3,R4,C

5.6.2 Description of High Pass Notch Fried circuit

Reference ID: -10 Order of values Returned: R1,R2,R3,R4,C,C1

5.6.3 Description of Low Pass Notch Boctor circuit

Reference ID: -11 Order of values Returned: R1,R2,R3,R4,R5,R6,C1,C2

5.6.4 Description of High Pass Notch Boctor circuit

Reference ID: -12 Order of values Returned: R1,R2,R3,R4,R5,R6,C1,C2

5.6.5 Description of , another, Low Pass Notch Boctor circuit

Reference ID: -13 Order of values Returned: R1,R2,R3,R4,R5,R6,C1,C2

Documentation info

Author:

Manos Tsardoulias <etsardou@users.sf.net>, Stratis Gavves <sg_aurelius@sf.net>

Chapter 6

Englab GUI (Graphical User's Interface)

- General Description
- Main window's components
 - The recent files list.
 - * Description
 - * Mouse Functions
 - The working directory's list.
 - * Description
 - * Mouse Functions
 - History List
 - * Description
 - * Mouse Functions
 - EngLab's Main Window
 - * Description
 - * EngLab's session
 - * EngLab's editor
 - * EngLab's editor for variables
 - * EngLab's search tool
 - Variable's Tree
 - * Description
 - * Variables description and Mouse Functions
 - * Constants description and Mouse Functions
 - * Functions description and Mouse Functions
 - Labels's Mouse Functions
- Menu bar
 - File
 - * New File
 - * Open File
 - * Open Toolbox
 - * Recent Files
 - * Load Workspace
 - * Save Workspace
 - * Save Workspace As
 - * Exit
 - Edit
 - * Cut
 - * Copy
 - * Paste
 - * Preferences
 - View
 - * Toolbars:
 - * Widgets:
 - Help
 - * About

- Toolbars
 - File toolbar
 - Workspace toolbar
 - Edit toolbar
 - Editor toolbar
- About box
- Variable's Edit Tab
 - Variable's Tree
 - Variable's Grid
- Information window
 - Variables and Constants
 - Functions
- Englab Editor
 - General
- Help-Search tool

6.1 General Description

Englab's GUI is made using Qt4. It is a graphical user's interface for Englab, which gives user many more functionalities than the console version. When Englab begins, the main window is shown. Through it, user can manage the variables, see the sessions history, open recent files and many other things, which will be described further down.

6.2 Main window's components

As said in paragraph [General Description](#) , main window is the window that shows up when a session of Englab GUI begins. Main window contains the following:

1. The menu bar placed top-side.
2. The icon bar placed below menu bar
3. A list containing the recent files placed in the right middle side of the main window.
4. A list representing the working directory of Englab in the right upper side of the main window
5. A list which contains the session's history placed in the bottom right corner of the main window.
6. A multi-tab area which contains the main features of EngLab:
 - (a) The main session of EngLab.
 - (b) The EngLab's editor.
 - (c) The edit variable.
 - (d) Help - Search
7. The variable's tree which is placed in the left side of main window.

Next, the above sections will be explained.

6.2.1 The recent files list.

6.2.1.1 Description

The recent files list contains the recent files opened in the [Englab Editor](#) from the user, or the recent files that were executed. The files are divided in many types: .eng files, image files, video files, sound files and others. If a file is an englab file (*.eng) an .eng icon appears next to it's name, and the same goes with the other file types. If the file is of other type a default icon is appeared.

6.2.1.2 Mouse Functions

1. Simple Left Click

No special function. Just selects the file.

2. Double Left Click

Executes the file. By the term "executes" we mean that the Englab script contained in the file is executed line by line.

3. Simple Right Click

When a simple right click occurs, a popup menu is appeared with the following elements:

- Execute: Executes the file. Same as being double clicked.
- Open In Editor : Opens the clicked file in the [Englab Editor](#) , if the file is an .eng file or of no type.
- Remove from recent : Removes the clicked file from the list.

6.2.2 The working directory's list.

6.2.2.1 Description

This list represents the working directory of Englab. By default the initial folder is the home folder of the user. The wd's first folder is the ".." folder which returns the parent folder of the wd. The files are indicated with different icons, depending on their type. Above wd list there are some helpful things:

1. An address bar for easier navigation
2. A "Home" button
3. A "Go Back" button
4. A "Go Up" button
5. A "Create New Folder" button

6.2.2.2 Mouse Functions

If the item clicked is not a folder:

1. Simple Left Click

No special function. Just selects the file.

2. Double Left Click

Executes the file. By the term "executes" we mean that the Englab script contained in the file is executed line by line, or the image/video/sound is loaded in Englab. Also the file is added in the recent files.

3. Simple Right Click

When a simple right click occurs, a popup menu is appeared with the following elements:

- Execute : Executes the file. Same as being double clicked.
- Open In Editor : Opens the clicked file in the [Englab Editor](#) .
- Add to recent : Adds this file in recent files.
- Go up : Browses the parent folder of wd.

If the item clicked is a folder:

1. Simple Left Click

No special function. Just selects the folder.

2. Double Left Click

Opens the folder.

3. Simple Right Click

When a simple right click occurs, a popup menu is appeared with the following elements:

- Go up : Opens the parent directory of WD.
- Go into : Opens the folder.

6.2.3 History List

6.2.3.1 Description

History list is a list that contains the current and the last four session's history of commands. The last command given, is appeared at the bottom of the list. Each session (except the current) is denoted with a name of the form "- → 11:42:41 - 30/05/2008", where the date and the time of the session is shown.

6.2.3.2 Mouse Functions

1. Simple Left Click

If an item is left clicked, it appears in the session area in order to be edited.

2. Double Left Click

If an item is double left clicked, the clicked command is executed.

3. Simple Right Click

Same as simple left click.

4. Double Right Click

Same as double left click.

6.2.4 EngLab's Main Window

6.2.4.1 Description

This section contains the main functionalities of EngLab which are:

6.2.4.2 EngLab's session

In EngLab's session the user is able to execute commands and see the results. The text box containing the session is command - like.

6.2.4.3 EngLab's editor

EngLab's editor has the capability of showing many tabs which contain a file editor each. The editor supports highlighting.

6.2.4.4 EngLab's editor for variables

In this section, user can edit graphically a variable (not constant). This tool has no limitation concerning the size, the type or the dimensions of the variables. In order to edit a variable, the user must has focus to this tab and click to a declared variable from the [Variable's Tree](#) .

6.2.4.5 EngLab's search tool

In general each function contained in EngLab's toolboxes has three distinct elements: It's name, it's short description and it's full description. In search tool, user can search for a phrase of his choise in any of these elements (or all together). The results appear on a list below the search box. If a result is clicked, the full description of the function is shown at a text box below the list.

6.2.5 Variable's Tree

6.2.5.1 Description

Variable's tree contains the current workspace. By the word "workspace" we mean the variables and the functions that are saved in Englab memory and can be used by the user. The root item of the variable's tree has the name "Workspace". In next versions, an option of importing many workspaces will exist. Workspace has three children: Variables, Constants and Functions.

6.2.5.2 Variables description and Mouse Functions

Node Variables contain the local variables that were denoted by the user. The variables are listed with alphabetical order, except if one variable's name is the same as the name of a constant. In addition, next to variable's name the dimensions adn the type of the variable are shown. Finally, in order to understand visually the type of the variable, each variable is colored according to it's type.

1. Simple Left Click

The [Information window](#) for the variable is updated.

2. Double Left Click

Nothing happens.

3. Simple Right Click

A popup menu is shown with the following options:

- **Delete** : Deletes the clicked variable from the program's memory.
- **Edit** : TODO Has to be implemented
- **Extract** : Saves the variable in an engv file.
- **Rename** : Renames the variable.

6.2.5.3 Constants description and Mouse Functions

Node Constants's children are the constants which were saved in memory when Englab's session began. New items can be added if declared with the type "const". In contrast to that, the constants cannot be deleted, but their values can be overwritten.

1. Simple Left Click

The [Information window](#) for the variable is appeared.

2. Double Left Click

No operation.

3. Simple Right Click

A popup menu is shown with the following options:

- **Extract** : Saves the constant in an engv file.
- **Rename** : Renames the constant.

4. Double Right Click

Same as Double Left Click.

6.2.5.4 Functions description and Mouse Functions

Node Functions contains the functions that can be used by the user. The children of the node Functions are nodes, each one of which represents a toolbox that was loaded in Englab during startup, except for the node 'userbox' that is designed to hold the functions that are defined by the user during runtime. The rest of the nodes are the [Englab Toolboxes](#) . If a toolbox is loaded it will appear as a node. It's children will be the functions that are part of the loaded toolbox.

The following mouse functions concern the toolbox functions.

1. Simple Left Click

The info for the function is displayed in the [Information window](#).

2. Double Left Click

No operation.

3. Simple Right Click

A popup menu is shown with the following options:

- **Use** : Inserts the function at the current position for the cursor in the command box

4. Double Right Click

No operation.

6.2.6 Labels's Mouse Functions

The following mouse functions concern the labels of the tree (parent-nodes).

1. Simple Left Click

No operation. Just selection of the label.

2. Double Left Click

Expands or collapses the node.

3. Simple Right Click

No operation.

4. Double Right Click

Same as Double Left Click.

6.3 Menu bar

The menu bar has four menu elements: "File", "Edit", "View", "Help", which are described below.

6.3.1 File

Contains the following:

6.3.1.1 New File

Creates an empty .eng file (appears in the editor tab)

6.3.1.2 Open File

A dialog appears to choose an .eng file to open in the editor

6.3.1.3 Open Toolbox

Not yet implemented

6.3.1.4 Recent Files

Not yet implemented

6.3.1.5 Load Workspace

A dialog appears to choose an .engw file (Englab workspace) to load into memory

6.3.1.6 Save Workspace

The current Englab workspace (all the variables currently in the workspace) will be saved to a file of the user's choice.

6.3.1.7 Save Workspace As

The current Englab workspace (all the variables currently in the workspace) will be saved to a file of the user's choice.

6.3.1.8 Exit

Exits the program.

6.3.2 Edit

Contains the following:

6.3.2.1 Cut

Cuts the selected text to the clipboard in the current session tab.

6.3.2.2 Copy

Copies the selected text to the clipboard in the current session tab.

6.3.2.3 Paste

Pastes text from the clipboard to the current session tab.

6.3.2.4 Preferences

Not implemented yet.

6.3.3 View

Contains the following:

6.3.3.1 Toolbars:

Under this submenu all the toolbar objects in the main window are listed and can be chosen to appear or not.

6.3.3.2 Widgets:

Under this submenu all the widget objects in the main window are listed and can be chosen to appear or not.

6.3.4 Help

Contains the following:

6.3.4.1 About

The [About box](#) is appeared.

6.4 Toolbars

6.4.1 File toolbar

The File toolbar contains the following buttons:

1. **"New file in editor"** : If pressed the [Englab Editor](#) opens with a blank and unsaved page ready to be written.

2. **"Open file in editor"** : As with the respective [File](#) action, a dialog appears for the user to choose an .eng file to open.

6.4.2 Workspace toolbar

The Workspace toolbar contains the following buttons:

1. **"Load Workspace"** : If pressed, a dialog appears for the user to select an existing workspace to load into memory.
2. **"Save Workspace"** : If pressed, the current workspace is saved. (Currently works as save as)
3. **"Save Workspace As"** : If pressed, the current workspace is saved with the selected filename.

6.4.3 Edit toolbar

The Edit toolbar contains the following buttons:

1. **"Cut"** : Cuts the selected text from input text box, since the output box is not editable.
2. **"Copy"** : Copies in clipboard the selected text from output or input text box.
3. **"Paste"** : Pastes the copied text from clipboard to input text box.

6.4.4 Editor toolbar

The Editor toolbar becomes available when the window focus is on the "Editor" tab. It contains the following buttons:

1. **"Run script"** : Executes the Englab script in the current file in the editor.
2. **"Save file"** : Saves the current file in the editor.
3. **"New file"** : Creates a new tab in the editor with a blank file.
4. **"Close file"** : Closes the current tab in the editor. If the file has not been saved the user will be asked whether to save changes or not.
5. **"Reload"** : Reloads the file in the current tab.

6.5 About box

About window is accessible from Help->About ([About](#)). It contains a brief description of Englab and the names of its main developers.

6.6 Variable's Edit Tab

Variable's Edit Box is accessible from the popup menu which appears when a variable is selected from the workspace widget.

6.6.1 Variable's Tree

The variable's tree is located in the left side of variable's edit window. The depth of the tree is N , where N stands for the number of the variable's dimensions. The root symbolizes the variable.

For example, if the variable is a matrix of dimensions $[2,3,4]$, the tree depth will be 3. Root's children will be 2, equal to the first dimension's size. Each of the two children stands for a matrix of dimensions $[3,4]$. Same as above, each of the two children will have 3 children, each of which will be a matrix of dimensions $[4]$. Similarly each of the above children will have 4 children which are one-dimensional values.

The last children have their value printed next to their names. For example, the above variable (with name a and all values equal to zero) will have a tree like this:

```

• a
  - a[0]
    * a[0,0]
      · a[0,0,0] 0
      · a[0,0,1] 0
      · a[0,0,2] 0
      · a[0,0,3] 0
    * a[0,1]
      · a[0,1,0] 0
      · a[0,1,1] 0
      · a[0,1,2] 0
      · a[0,1,3] 0
    * a[0,2]
      · a[0,2,0] 0
      · a[0,2,1] 0
      · a[0,2,2] 0
      · a[0,2,3] 0
  - a[1]
    * a[1,0]
      · a[1,0,0] 0
      · a[1,0,1] 0
      · a[1,0,2] 0
      · a[1,0,3] 0
    * a[1,1]
      · a[1,1,0] 0
      · a[1,1,1] 0
      · a[1,1,2] 0
      · a[1,1,3] 0
    * a[1,2]
      · a[1,2,0] 0
      · a[1,2,1] 0

```

- a[1,2,2] 0
- a[1,2,3] 0

With this way, we have an easy and comprehensible way of visualizing the variables, whatever are their dimensions.

6.6.2 Variable's Grid

The variable's grid is a 2-dimensional grid of dimensions $A \times B$, where A and B are the two last variable's dimension's sizes. For the example above, the grid has 3x4 dimensions. The grid can visualize the values of two dimensional matrixes. In our example, two dimensional matrixes are the tree nodes a[0] and a[1]. If any of them is selected, their contents will show up in the grid.

It is understandable that the root of the tree, represents a three dimensional matrix, so if the root is selected, the grid will show nothing.

According to the above, a node of the form a[x,y] represents a line vector. So, if such a node is selected the appropriate line in the grid will be highlighted. Finally in a leaf of the tree is selected, the appropriate cell of the grid will be highlighted.

Of course, the grid's cells are editable. User can easily change each cell's value. When a change is done, a command is executed in main window, so that the user can see what changes he made. If the change cannot be done, the cell's value remains as it was (for example if user tries to put a complex value in an integer variable).

Finally, when a change is made, the changed node in the tree becomes visible and highlighted.

6.7 Information window

Information window is accessible from the selection "Info" from the popup menu, when a variable, a constant or a function is left clicked in the [Variable's Tree](#) .

The information window shows different things when a variable or constant is clicked than the case when a function is clicked.

6.7.1 Variables and Constants

When a variable's or constant's information window is shown, it contains the following information:

1. **Name:** The name of the variable
2. **Size:** The dimensions of the variable and their sizes
3. **Data:** The data of the variable (it's values in form of two dimensional matrixes)

6.7.2 Functions

When a function's information window is shown, it contains the following:

1. **Name** : What the function's name is
2. **Syntax** : What the function's syntax is
3. **Description** : What the function does
4. **Input arguments**
5. **Returns** : The result of the function

6.8 Englab Editor

6.8.1 General

Englab's editor is a build-in script editor to edit and execute Englab scripts. The Editor tab appears in the main window's tab widget. It is possible to edit many files at the same time because each open file appears in a separate tab. Note that if a script is executed the current workspace will be altered according to the scripts commands.

The usual actions (open file, new file etc.) can be performed either by choosing the respective menu items in the [Menu bar](#) or by using the buttons in the [Editor toolbar](#) .

6.9 Help-Search tool

The Help-Search tool offers a method to search within the documentation included with the functions of the toolboxes loaded. The user can enter keyword(s) and is able to choose the parts of each function documentation (Function name, Function short description and Function full description) in which the keywords will be searched.

Author:

Manos Tsardoulis <etsardou@users.sf.net>

Chapter 7

TODO

- [TODO](#)

7.1 TODO

- Implementation of file functions
- Structures
- correction of if, do-while blocks
- Implementation and enrichment of existing toolboxes (opencv etc)

Chapter 8

Bug List

Page EngLab Commands Let "id" be the inputId of the last input given to EngLab. If you execute command "! id+1" the program falls into an infinite loop and crashes

Page EngLab Commands Command "!" with no arguments doesn't display history

Page EngLab Commands The history box is not cleared permanently. If you close and re-run englabgui the history appears again. Workaround: Remove the history folder.

Page EngLab Commands Deleting a variable that isn't allocated doesn't give out a warning

Page EngLab Commands Argument -f in "delete" command is not implemented yet and is ignored if specified

Page EngLab Commands Argument -a in "delete" command is not implemented yet and is ignored if specified

Page EngLab Commands In command "exec" fileName and -v arguments are not exchangeable. If specified with a different order -v is interpreted as a filename.

Page EngLab Commands Command Exit always exits with status 0. The "status" argument is not implemented and ignored if supplied.

Page EngLab Commands Command "help toolbox functionName" crashes if "functionName" doesn't exist in toolbox "toolbox"

Page EngLab Commands Command "ismem" -c and -a arguments have not been implemented yet and are ignored if specified

Page EngLab Language The while block is not implemented yet.

Page EngLab Language The do while block is not implemented yet.

Page EngLab Language The if-else block is not implemented yet.

Page EngLab Language Not yet Implemented

Chapter 9

Todo List

Page [EngLab Commands](#) add reference to constants