Digital Micrograph
BASIC & ADVACNED SCRIPTING

Bernhard Schaffer *
Bernd Kraus +

*** FELMI**
Research Institute for Electron Microscopy
Graz University of Technology
Steyrergasse 17, A-8010 Graz, Austria
Austrian Centre for Electron Microscopy and Nanoanalysis
www.felmi-zfe.at

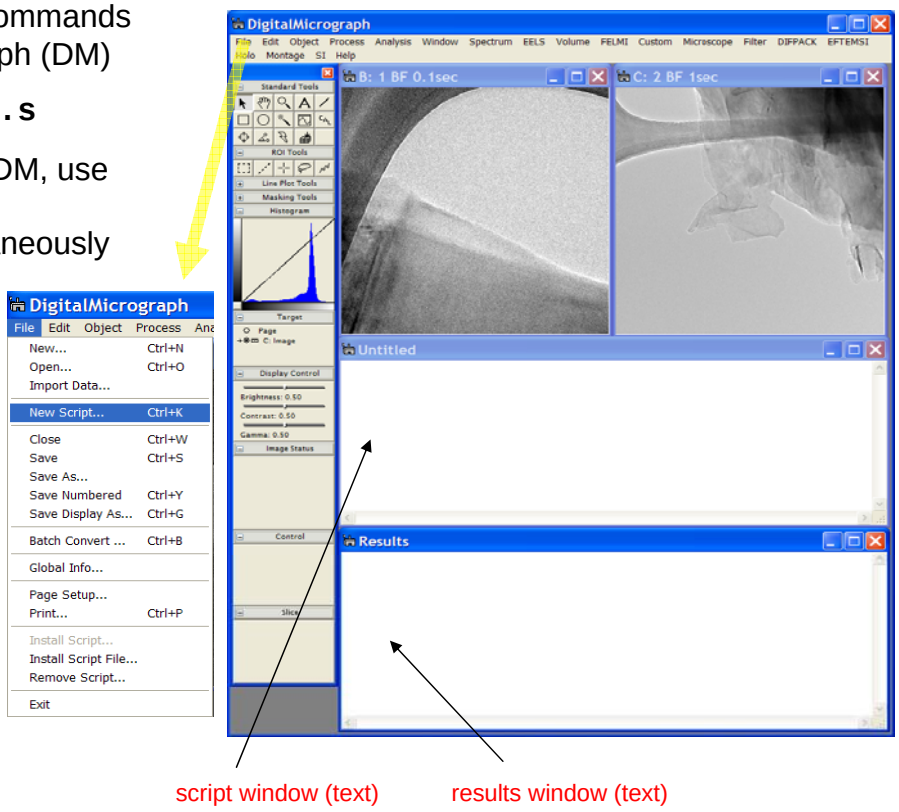
+ GATAN GmbH
Ingoldstädter Straße 12, D-80807 Munich, Germany
www.gatan.com

Scripting – WHY ?

- Scripting allows easy manipulation and evaluation of your data within DigitalMicrograph
- Scripting allows automation of regularly performed tasks within DigitalMicrograph
- Scripting allows customization and expansion of some tasks (e.g. acquisition!) within DigitalMicrograph.

First scripting window

- A script is a simple text file containing commands which are interpreted by DigitalMicrograph (DM)
- The files are saved with the extension **`*.s`**
- To create a new script window (text) in DM, use the menu: **File / New Script…**
  or press the buttons **`Ctrl`** and **`K`** simultaneously
- The script is written in the new *Untitled* text window.
- The script is executed by 'activating' the script window (click on window) and pressing the buttons **`Ctrl`** and **`Enter`** simultaneously
- The script is first checked for syntax errors. If none is found, the script is interpreted line by line



First script – 'Hello world!'

- While a script is running, no other tasks can be performed in DM. The mouse cursor becomes an hour-glass. After the script finishes, the mouse cursor becomes an arrow again.
- A running script can be interrupted by pressing the keys **Ctrl** and **Break** simultaneously
- Any script output (also error-messages!) prints to the results-window.

  If no results window is shown, display it using the menu: **Window / Show Results Window**
- The following script command prints output to the results window:

  `result(`*`string`***`)`**

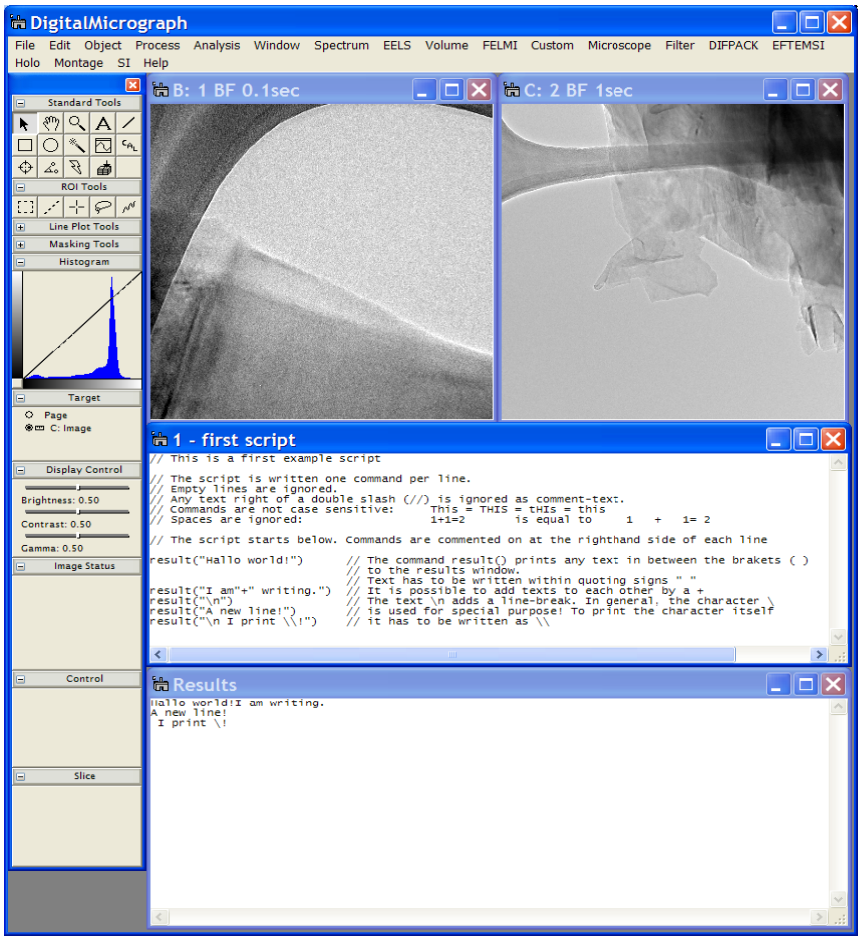
  where *string* can be a variable or any text within quotes, e.g.:

  `result("Hello word!\n")`

  prints to the results window:

  Hello world!

  The two characters **\n** add a line break



TASK T01

Do the following to test your first script:

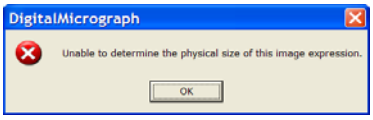
1. Make the *results window* visible (if not already shown)
2. Create a new *script* window (*text window*)
3. Write a script with at least one comment line on top, telling the purpose of the script
4. Write a script to print “My name is *YOUR NAME*” in the results window, followed by three empty lines, and “And who are you?” in the fifth line.
5. Right of one command (choose any), write the comment: “I will always comment my scripts!” (Keep this pledge!)
6. RUN your script. The output in the results window should look like below:
7. Save your script as **`S01.s`**

Directly addressing images

- Images can be directly addressed in a script by using their *image-letter*, shown in the title of the image window.
- The following script command inverses the image contrast of the image with the letter B by multiplying its values by minus 1.

```
B = B * (-1)
```

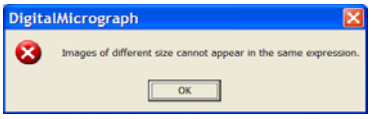
If no image with letter B is shown, the script will result an error message:

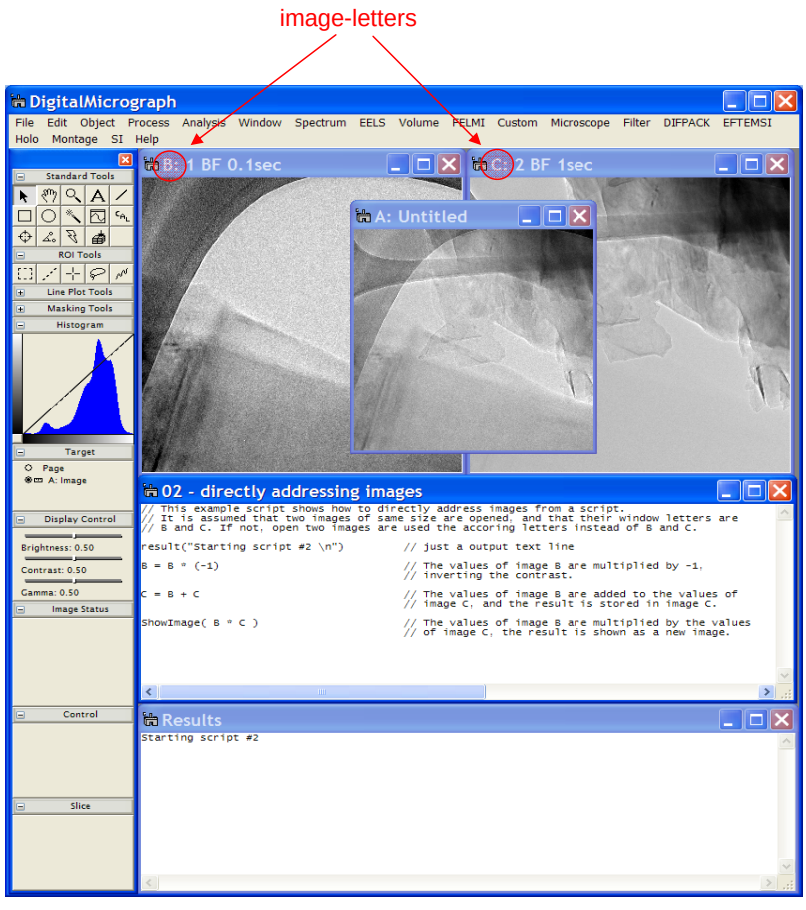


- The following script multiplies image B and C with each other (pixel by pixel). The result is shown as new image with name *Untitled*.

```
ShowImage (B * C)
```

If the images have different sizes, the script will result an error message:





Some useful commands for image evaluation

INPUT: image OUTPUT: single value

Name	Summary
max	Returns the maximum value
mean	Returns the mean value
min	Returns the minimum value
variance	Returns the variance value
sum	Returns the sum
GetName	Returns the image name
GetLabel	Returns the image-letter

This script outputs the important values of an image (image-letter B):

```
result("\n NAME     :"+GetName(B))
result("\n LETTER   :"+GetLabel(B))
result("\n SUM      :"+sum(B))
result("\n MEAN     :"+mean(B))
result("\n VARIANCE :"+variance(B))
result("\n MINIMUM  :"+min(B))
result("\n MAXIMUM  :"+max(B))
```

INPUT: image OUTPUT: image
(pixel-by-pixel operation)

Name	Summary
abs	Calculates absolute value
acos	Calculates the arccosine
asin	Calculates the arcsine
atan	Calculates the arctangent
atanh	Calculates the hyperbolic arctangent
cos	Calculates the cosine
cosh	Calculates the hyperbolic cosine
exp	Calculates the exponential
exp10	Calculates 10 raised to x
exp2	Calculates 2 raised to x
Factorial	Calculates the factorial
SGN	Gives the sign (+/-)
SQRT	Calcualtes the square-root

This script squares the absolute value of an image (image-letter B), keeping the sign:

```
ShowImage(SGN(B)*B**2)
```


tert () – a powerful command

- The powerful command *tert( )* evaluates an image pixel-by-pixel in parallel. For each pixel, a *condition* is evaluated, and depending on the result, one of two values is chosen as output. The command syntax is:

```
tert(condition,true-value,false-value)
```

- *Conditions* are built using the logical operators:

Operator	Meaning
==	Equality
!=	Inequality
<	Less than
<=	Less than or Equal
>	Greater than
>=	Greater than or Equal
!	Logical NOT
&&	Logical AND
||	Logical OR

- The following script creates a binary mask from image B. Each point, where B was positive, becomes 1, each other point 0.

```
B = tert( B < 0 , 0 , 1 )
```

- It is possible to use images as result values The following script shows an image which has the values of image B where the image A is one, while it has the values of image C elsewhere. All three images have to be of the same size!

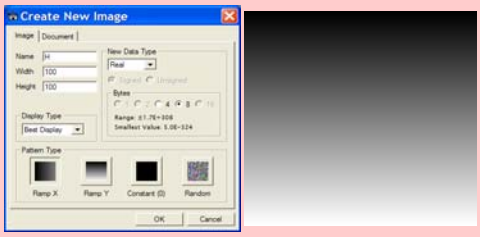
```
ShowImage( tert( A==1 , B , C ) )
```

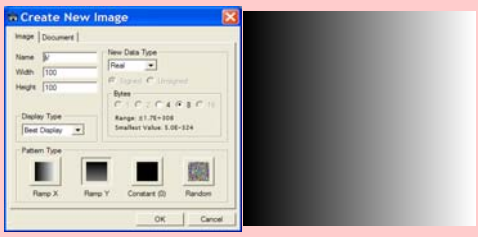
- The following script shows an binary image which is one, only where both A and B are positive and non-zero.

```
ShowImage( tert( (A>0)&&(B>0) , 1 , 0 ) )
```

TASK T02

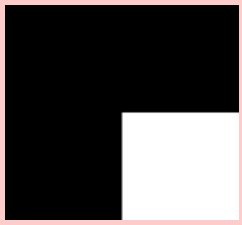
1. Use File/New... **to create two new images of 100x100 pixel size. One with horizontal, one with vertical gradient.**





2. **Write and test a script which does the following:**

1. Display a new image which is 1, where both images have intensities above their mean-value.



2. Display a new image which has the values of A, where A has values within one standard deviation of A's total mean value.

Directly addressing image areas

- Selections (rectangular ROIs) in images can be directly addressed in a script by using the *image-letter* followed by empty square brackets. e.g.:

```
B[] = B[] * (-1)
```

If no ROI is in the image, the whole image will be addressed.

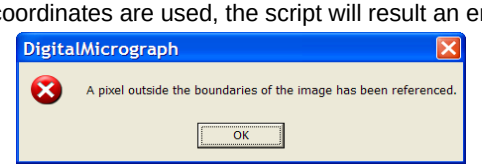
- Image areas are addressed by giving the coordinates of two points: Top-Left and Bottom-Right in square brackets.
  e.g.: `A[ top , left , bottom , right ]`

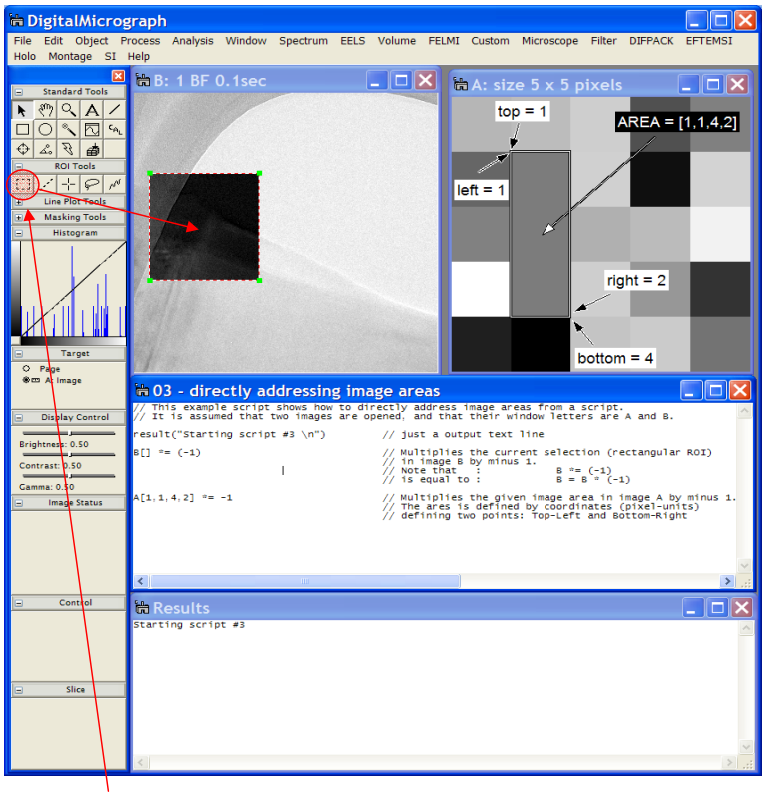
```
A[1,1,4,2] = A[1,1,4,2] * (-1)
```

Note that area coordinates address the grid *in between* the pixels, and not the pixel-centers! The top-left point of an image is the origin, and coordinates increase towards the bottom-right.

The topmost-left pixel is therefore addressed by the area [0,0,1,1]. The bottommost-right pixel of an (5x5) image by: [4,4,5,5]

If incorrect coordinates are used, the script will result an error message:





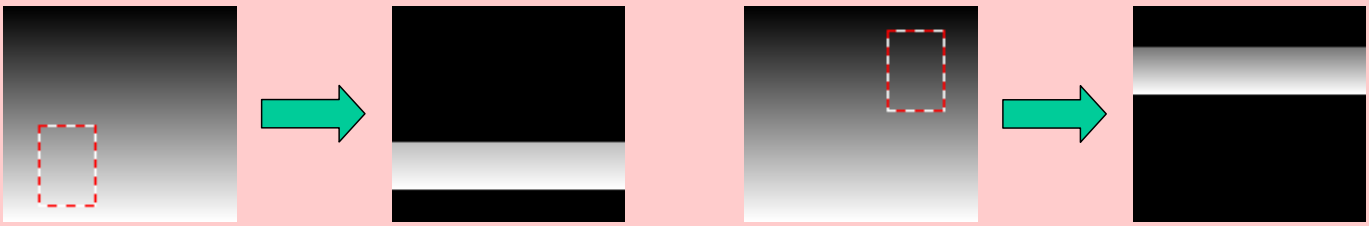
Rectangular selection (**ROI** = **R**egion **O**f **I**nterest)

TASK T03

Expand the scripts of task T02 and test them:

1. Display a new image which is 1, where both images have intensities above the mean-value of their top-left quarter, respectively.
(Remember, the images are 100x100 pixels in size)

2. Display a new image which has the values of A, where A has values within one standard deviation of the mean value of a user-drawn ROI.



Variables (numbers and strings)

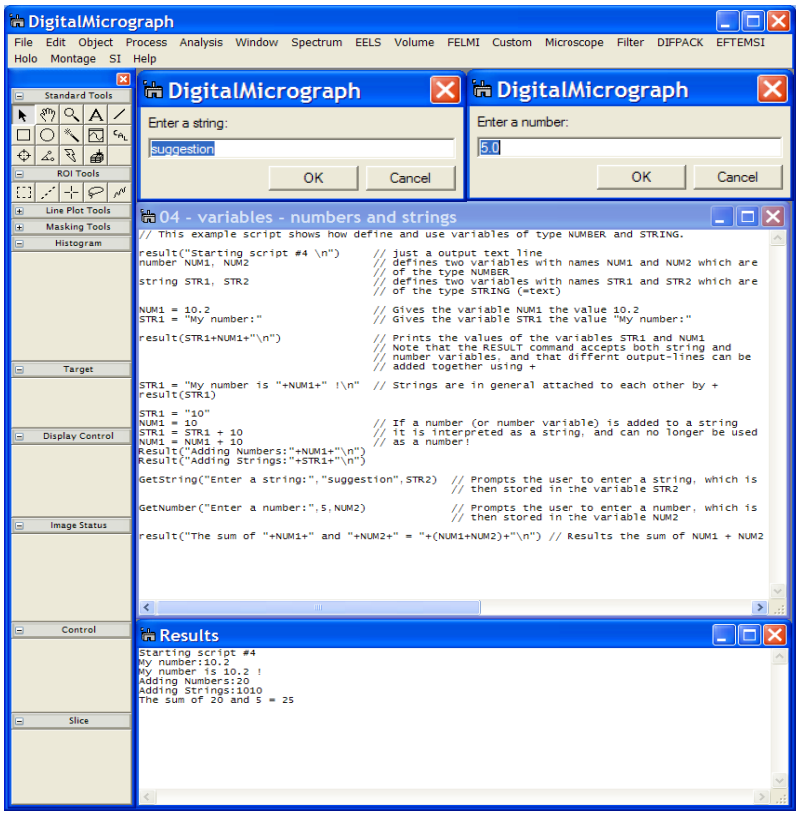
- Variables are *containers*. They posses:
  - a ***name*** (has to start with a letter)
  - a ***type*** (e.g. *number* / *string*)
  - a ***value*** (of that type)

- Variables need to be defined before they can be used. This script defines a variable of the ***type*** number with the ***name*** NUM1, and gives it the ***value*** 10.2. Then it results the double of this value:

```
number NUM1
NUM1 = 10.2
result("Input was: "+NUM1+"\n")
NUM1 = NUM1 * 2
result("Doubled is: "+NUM1+"\n")
```

- The following commands prompt the user to enter values for a variable:

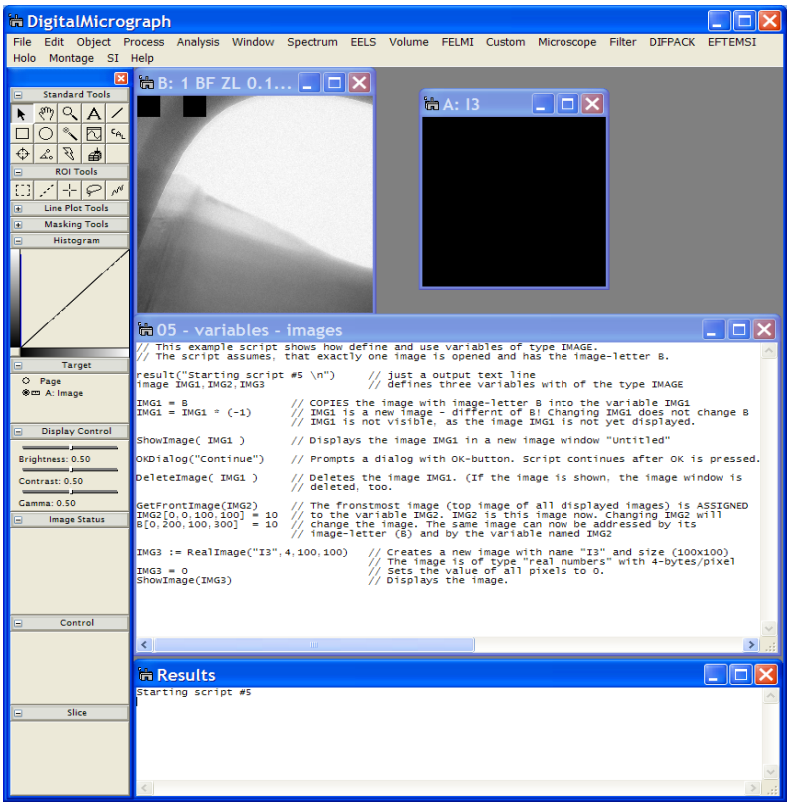
```
number NUM2
string STR2
GetString("Enter a string:","suggestion",STR2)
GetNumber("Enter a number:",5,NUM2)
```



Variables (images)

- A third type of variables is the *image* type.

- The following script *assigns* an image variable named IMG to the front-most displayed image. The image can then be addressed by IMG.
  A second image variable COPYIMG is created and contains a *copy* of the *values* of IMG.
  Changing COPYIMG does not change IMG.
  It is then displayed.
  A third image variable NEWIMG is created as 100x100 pixel image (of type: real 4-bit)




```
image IMG , COPYIMG , NEWIMG
GetFrontImage(IMG)
IMG = IMG * (-1)
COPYIMG = IMG
COPYIMG = COPYIMG + 100
ShowImage( COPYIMG )
NEWIMG := RealImage("New",4,100,100)
NEWIMG = 0
ShowImage( NEWIMG )
```

- The following command deletes the image variable IMG (and its assigned image, if displayed)

```
DeleteImage(IMG)
```


TASK T04

Write and test a script which does the following:

1. Ask the user to enter a x-dimension and a y-dimension of an image
2. Ask the user to enter an image name
3. Create an image of size (X x Y) with the given name (Type: real 4-bit)
4. The upper half of the image should get the intensity 100
   The lower half should get the intensity 200
5. The left half of the image should then be multiplied by -1
6. Display the image (which should look like the example below)

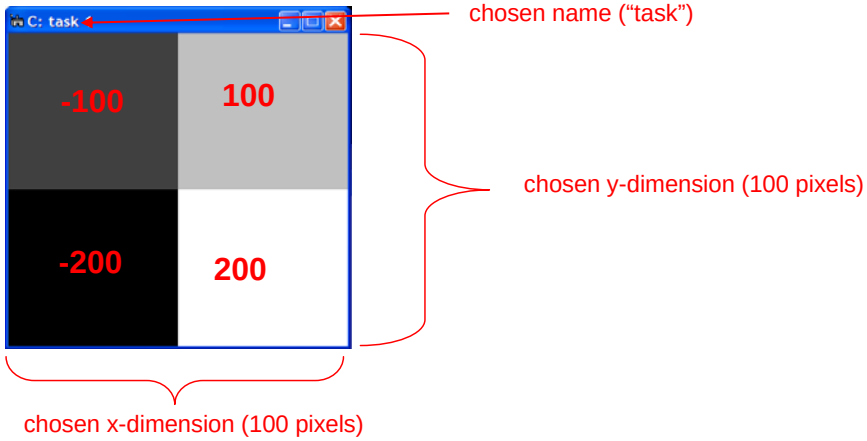
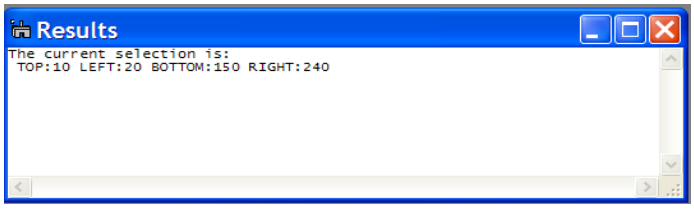


Image selections (ROIs)

- It is possible to read, set, and change an image selection (rectangular ROI) in an image. The following script demonstrates the according commands.

```
image IMG
IMG := RealImage("Test",4,256,256)
ShowImage(IMG)

ClearSelection(IMG)            // Deletes a selection in the image

SetSelection(IMG,10,20,150,240) // Sets a new ROI in the image. The areas is defined by
                                // the coordinates of two points: top-left , bottom-right
                                // Existing ROI(s) are automatically deleted.

number t,l,b,r
GetSelection(IMG,t,l,b,r)      // Reads the coordinates of the current ROI in the image.
                               // If no ROI is present, it reads the coordinates of the
                               // whole image ( 0, 0 , sizeY , sizeX)

result("The current selection is:\n")
result(" TOP:"+t+" LEFT:"+l+" BOTTOM:"+b+" RIGHT:"+r+"\n")
```



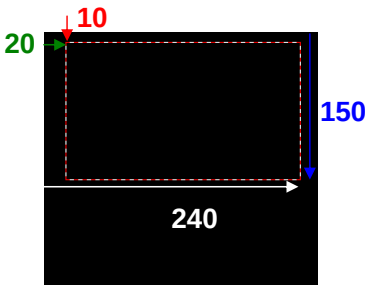


Image related commands

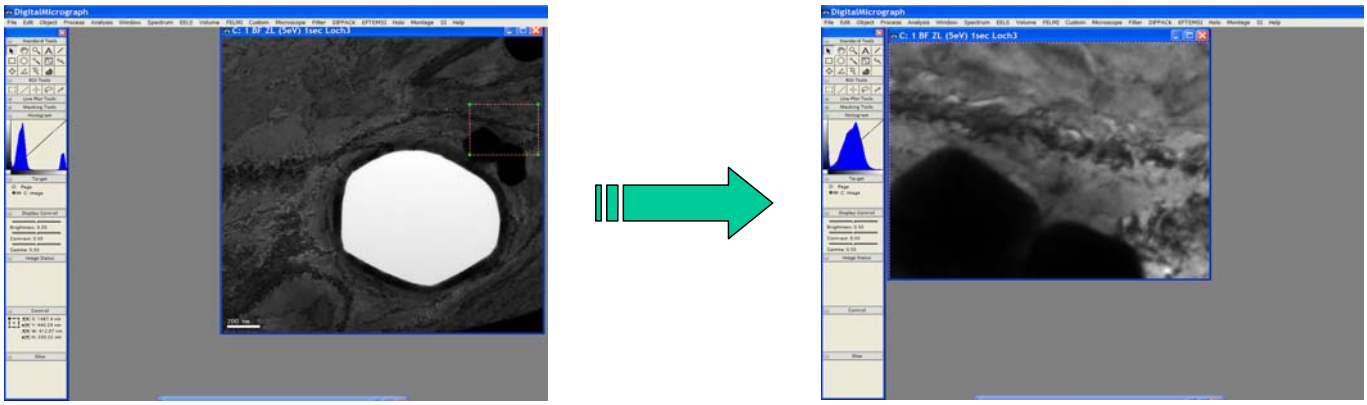
- Images in DM are more than an array (pixels) of numbers (pixel values). The following commands can read/alter some attributes of images (e.g. name or calibration) and how they are displayed:

```
image  IMG
IMG := RealImage("Test",4,100,100)
ShowImage(IMG)

string NAME
NAME = GetName(IMG)          // reads the name of an image
SetName(IMG,"My Image")      // sets the name of an image

number SCx,SCy,Ox,Oy
GetScale(IMG,SCx,SCy)        // Gets the scale (units/pixel) for X and Y dimension
SetScale(IMG,0.1,0.1)        // Sets the scale for X and Y
GetOrigin(IMG,Ox,Oy)         // Gets the coordinates of the origin
SetOrigin(IMG,0,0)           // Sets the origin (in this case to the top/left corner)

SetSurvey(IMG,0)             // turns off the "auto-survey" of the image display limits
SetSurvey(IMG,1)             // turns on the "auto-Survey"
SetSurveyTechnique(IMG,2)    // Sets the "auto-survey" technique.
                             // 0=cross wire , 1=whole image , 2=sparse , 3=reduction
number HIGH,LOW
GetLimits(IMG,LOW,HIGH)      // Gets the current high & low limits of the image display,
SetLimits(IMG,0,100)         // Sets high & low limits for the image display, if "auto-survey" is off

SetInversionMode(IMG,1)      // Turns on "inverted contrast" (0 to turn it off)
SetZoom(IMG,2)               // Sets the ZOOM of an image to 200% (Does not change size of window)

number SX, SY, WX, WY, IWX, IWY
GetWindowSize(IMG,SX,SY)     // Gets the size of the image window (in pixels).
SetWindowSize(IMG,150,150) // Sets the size of the image window (in pixels). The ZOOM is automatically
                           // adjusted, just as when resizing the window with the mouse.
GetWindowPosition(IMG,WX,WY)          // Gets the position of the image window within the DM application.
SetWindowPosition(IMG,200,30)         // Sets the position of the image window with respect to the
                                      // application window. Be careful not to use y<30 or the window
                                      // will be "hidden" behind the menubar.
SetImagePositionWithinWindow(IMG,0,0) // Sets the position of the top-left corner of the image with
                                      // respect to the image window. Remember: Image ZOOM plays a role.
```

TASK T04b

Write and test a script which does the following:

1. Get the front most image. (Which should have a selection.)
2. Read the selection of the image, and set the display limits to the minimum and maximum value of this image area.
3. Resize the image window to the aspect ratio of the selection. Keep the window dimension which belongs to the larger dimension of the selection (either X or Y) constant.
4. Place the image window in the top-left corner of the DM application window, but below the menu-line and right of floating tool windows. (Usually this is x=145 y=35.)
5. Zoom and place the content of the image window, so that the selected area is shown with maximum zoom fitting to the window.



IF cases

- The IF statement is used to make decisions in a program. Based on a condition, commands are executed or not. It has the syntax:

```
IF (condition) action
```

```
IF (condition) action
ELSE alternative-action
```

```
IF (condition)
  {
  action1
  action2
  …
  }
```

```
IF (condition)
  {
  action1
  …
  }
ELSE
  {
  alternative-action1
  …
  }
```

Note that the { } brackets are used to create blocks of commands. Indenting those blocks helps reading script-code, especially if several nested blocks are used.

- Several commands have a return value of: 1 = success
  0 = failure

They can be directly used as condition:

```
number NUM1
IF (GetNumber("Enter x",0,NUM1)) result("You entered:"+NUM1)
ELSE result("You have pressed the CANCEL button")
```

Or, using the logical NOT operator !:

```
string STR1
IF (!GetString("Enter text","",STR1)) STR1 = "default string value"
```

WHILE loops

- The WHILE statement is used to create loops in a program. The actions are repeated as long as the *condition* is fulfilled. The WHILE statement has the syntax:

```
WHILE (condition) action
```

```
WHILE (condition)
  {
  action1
  action2
  …
  }
```

- The following input dialog will reappear until a positive number is entered:

```
number NUM1
NUM1 = -1
While( NUM1<0 ) GetNumber("Enter a POSITIVE number:",NUM1,NUM1)
```

- The *break* command can be used to exit the current loop at once. Most often it is used in combination with an IF statement:

```
number NUM1
NUM1 = -1
While( NUM1<0 )
  {
  If (!GetNumber("Enter a POSITIVE number:",NUM1,NUM1)) BREAK
  result("You entered:"+NUM1+"\n")
  }
```


FOR loops

- The FOR statement is used to create counting loops in a program. The actions are repeated until a counting variable reaches a limit. The FOR statement has the syntax:

```
FOR (initialize ; condition ; action) loop-action
```

```
FOR (initialize ; condition ; action)
  {
  loop-action1
  …
  }
```

- This script simply counts from 1 to 20 and results 20 lines:
  (The line `COUNT++` is equal to `COUNT = COUNT +1`)

```
number COUNT
For (COUNT=1 ; COUNT<=20 ; COUNT++) result("STEP#"+COUNT+"\n")
```

- The next script counts from MAX to 0 with a given STEPSIZE:
  (The line `COUNT -= STEPSIZE` is equal to `COUNT = COUNT - STEPSIZE`)

```
number COUNT, LIMIT, STEPSIZE
STEPSIZE = 5
LIMIT = 100
For (COUNT=LIMIT ; COUNT<=0 ; COUNT-=STEPSIZE) result("STEP#"+COUNT+"\n")
```

- Be aware of never-ending loops, especially when using the counting variable within the loop:
  (The line `COUNT *= -1` is equal to `COUNT = COUNT * (-1)`)

```
number COUNT
for (COUNT=1; COUNT<10; COUNT++)
          {
          result(" COUNT = "+COUNT+"\n")
          COUNT *= -1
          }
```

COUNT = 1
COUNT = 0
COUNT = 1
COUNT = 0
…

TASK T05

Write and test a script which does the following:

1. Take the front most image and determine the *dynamic range* of its values (**`min`** & **`max`**)
2. Successively decrease a **`limit`** from **`max`** to **`min`** in 2% steps of the dynamic range, until (at least) 50% of all image pixels have values below the **`limit`**.
3. Use this **`limit`** to calculate the mean value of the remaining (at maximum) 50% 'brightest' pixels in an image.
4. Output the mean value, the **`limit`** and the source image with all excluded pixels having the value 0.

Hint: Either use a `for`-loop and the `break` statement, or use a `while`-loop.

Hint: Use the `tert()` and `sum()` commands to determine how many pixels of an image are below the given limit, and to calculate the mean value of the brightest pixels.

Hint: Use the command `GetSize()` to determine the dimensions of an image:

```
number sizeX , sizeY
image Image
GetSize(Image , sizeX , sizeY)
```

Note: Such a script can for example be used to automatically determine a CCD exposure time, avoiding over-exposure at "bright" image parts.

icol, irow, iradius, ...

- There are several *intrinsic* variables which can be used in calculations of images. Their value depends on the position within the image.

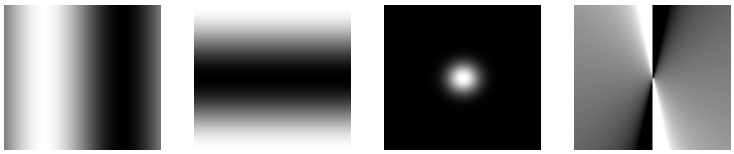
  (e.g.: `icol` becomes 5 for all points in an image, which have x=5 as coordinate. It becomes 6 for x=6 and so on..)

Name	Description
icol	column of the image
iheight	height of the image
ipoints	number of points in the image
iradius	distance from the center of the image
irow	row of the image
itheta	angle with respect to the center of the image
iwidth	width of the image
iplane	plane of the image (3D images)

- The following script creates some examples:

  (The function `Pi()` returns the value of Pi.)

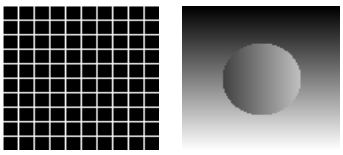
```
image  TestImage
TestImage := RealImage("Test",4,100,100)
ShowImage(TestImage)

TestImage = sin(2*Pi()/iwidth*icol)
TestImage = cos(2*Pi()/iheight*irow)
TestImage = exp(-iradius**2/(iheight/10)**2)
TestImage = tan(itheta)
```



- Often, the intrinsic variables are used in the `tert()`command:

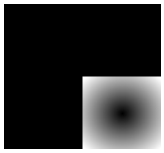
  (The function `mod(a,b)` returns the modulo, e.g. `mod(14,3)`=2 as 14 = 4*3 + 2)

```
TestImage = tert( mod(icol,10)==0 || mod(irow,10)==0,1,0)
TestImage = tert( iradius<iwidth/4 , icol , irow)
```



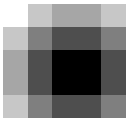
- Note that the variables check the actual image expression, not the image itself. If an area of an image is used, the top-left pixel of this area is (0/0):

```
TestImage = 0
TestImage[50,50,100,100] = iradius  // the center is now at 75/75!
```



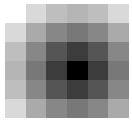
Center of “iradius”

- Be aware that the value of a pixel belongs to the top-left corner of its pixel-area. Using iradius, the origin (0/0) from which the radius is calculated may be either in the center of a pixel (even dimensions), in the center of an edge (one even, one odd dimension) or in the corner (odd dimensions). Especially the one-dimensional case may cause troubles.



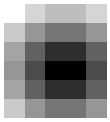
(5x5)

3.54	2.92	2.55	2.55	2.92
2.92	2.12	1.58	1.58	2.12
2.55	1.58	0.71	0.71	1.58
2.55	1.58	0.71	0.71	1.58
2.92	2.12	1.58	1.58	2.12



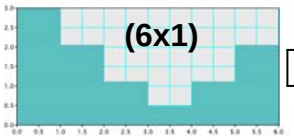
(6x6)

4.24	3.61	3.16	3.00	3.16	3.61
3.61	2.83	2.24	2.00	2.24	2.83
3.16	2.24	1.41	1.00	1.41	2.24
3.00	2.00	1.00	0.00	1.00	2.00
3.16	2.24	1.41	1.00	1.41	2.24
3.61	2.83	2.24	2.00	2.24	2.83



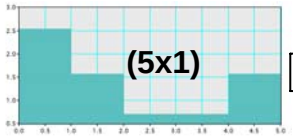
(5x6)

3.91	3.35	3.04	3.04	3.35
3.20	2.50	2.06	2.06	2.50
2.69	1.80	1.12	1.12	1.80
2.50	1.50	0.50	0.50	1.50
2.69	1.80	1.12	1.12	1.80
3.20	2.50	2.06	2.06	2.50



(6x1)

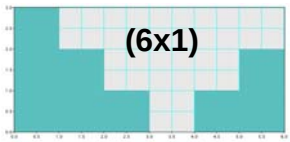
3.04	2.06	1.12	0.5	1.12	2.06



(5x1)

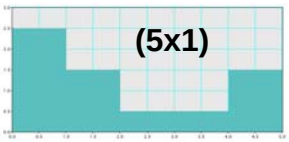
2.55	1.58	0.71	0.71	1.58

- To get the proper behavior for the one-dimensional case, one should therefore replace `iradius` by the expression: `(iwidth/2-icol)`



(6x1)

3	2	1	0	1	2



(5x1)

2.5	1.5	0.5	0.5	1.5

TASK T06

Write and test a script which does the following:

1. Create an image of size 2x6 with predefined values (see example below):
2. Think of this image as 6 points with coordinates X/Y. Perform a linear fit through these points, calculating k,d for `y = k*x + d`
3. Use the linear equation to produce a graph (1D-image) with 200 points from x=-10 to x=10. Display this graph as lineplot.

Hint: The code at right defines the image from above directly:

```
Image POINTS := [2,6] :
 {
 {2,12.5},
 {4,23.8},
 {7,43.2},
 {9,52.6},
 {11,67.7},
 {12,71.8}
 }
```

X	Y
2	12.5
4	23.8
7	43.2
9	52.6
11	67.7
12	71.8

Hint: The linear fit can be calculated, using the equations:
(n = 6, X=x value Y=y value, sums are calculated over all points)

$$k = \frac{n\cdot\left(\sum X\cdot Y\right)-\left(\sum X\right)\cdot\left(\sum Y\right)}{n\cdot\left(\sum X^2\right)-\left(\sum X\right)^2} \qquad d = \frac{\left(\sum Y\right)-k\cdot\left(\sum X\right)}{n}$$

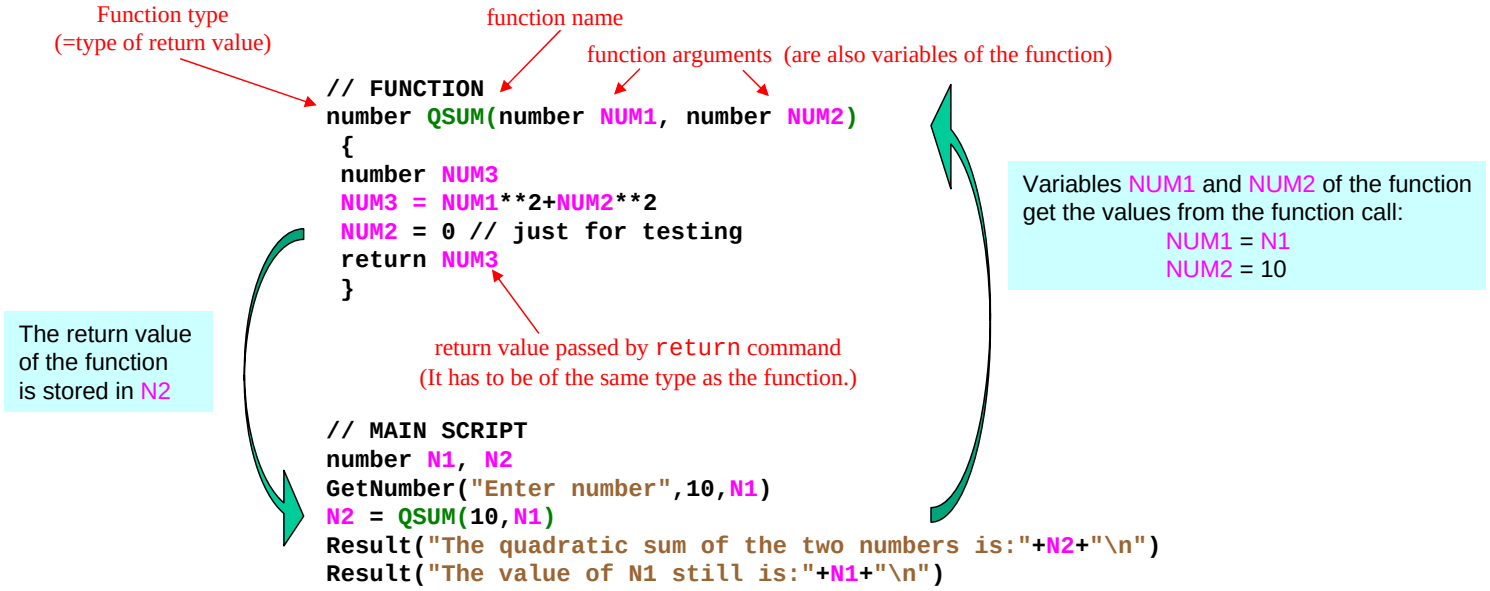
Hint: To calibrate the lineplot, use the following commands:

`SetScale(IMAGE,ScX,ScY)` Sets the scale of an image in X and Y.
The distance between two neighbouring pixels is then ScX units along X.
The distance between two neighbouring pixels is then ScY units along Y.

`SetOrigin(IMAGE,X,Y)` Places the image origin at the (pixel-)coordinates (X,Y).
The pixel at (X,Y) has then the calibrated coordinates (0,0).

Functions

- DM scripts can be structured by using functions. A function is a set of commands performing a certain task. A function has a *name*, a set of *arguments*, a *return value*, and a *body* consisting of commands. Functions can be called in a script like a command.



- Functions have to be defined *before* they are used for the first time.
- Variables are *local*. The main program does not know NUM1, NUM2, and NUM3. The function does not know N1 and N2. Only the *values* are passed, and only into one direction. Therefore, changing NUM2 in the function did not change the value of N1.

Functions: arguments and optional arguments

- Function can have an arbitrary number of arguments (also zero!).
  Functions may be of the same types as variables (number, string, image,…).
  An additional type called *void* allows functions which return no value (=procedures).

```
void PrintHallo()
 {
 result("\n HALLO \n")
 return              // the return command at the end of a procedure can be omitted.
 }

// Main Script
PrintHallo()
```

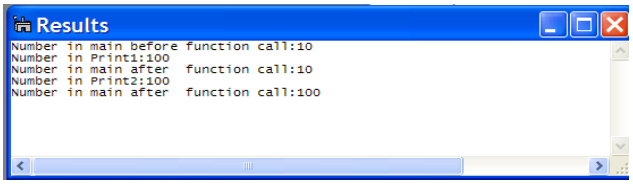
- It is possible to define the same function with different arguments. This can be used to create *optional* arguments. (Note that functions can be called from within a function.)

```
void PrintLine(number NUM)
 {
 number count
 for (count=1;count<=NUM;count++) result("\n")
 }

void PrintLine() PrintLine(2) // This procedure has only one line. Blocks { } can therefore be omitted.

// Main Script
PrintLine(5)        // 5 empty lines
result("HALLO\n")
PrintLine()         // 2 empty lines
```

Functions: assigned variables

- Normally, argument *values* are passed from the main program to the function. However, it is also possible to pass *assign* variables instead. Changing the variable in the function then does change the variable in the main-program. *Assigned* arguments are defined, using a leading ampersand (&) sign.

```
void Print1(number NUM1)        // The argument just gets the value of the passed variable.
 {
 NUM1 = NUM1 * 10               // This will only change the internal value of NUM1
 result("Number in Print1:"+NUM1+"\n")
 }
void Print2(number &NUM1)       // The argument is now assigned to the passed variable!
 {
 NUM1 = NUM1 * 10               // This will also change the value of the passed (assigned) variable.
 result("Number in Print2:"+NUM1+"\n")
 }

number N1
N1 = 10
result("Number in main before function call:"+N1+"\n")
Print1(N1)
result("Number in main after function call:"+N1+"\n")
Print2(N1)
result("Number in main after function call:"+N1+"\n")
```



- Note: While it is possible to call `Print1` with a parameter, e.g. `Print1(10)`, this is no longer possible for `Print2`, because the argument now needs to be a variable! `Print2(20)` results in an error.

Functions: image arguments

- Function arguments can also be images. However, image arguments are *always, automatically* passed as *assigned arguments* not as *values*. Therefore, changing the image within the function will change the image of the main program! In the following script, the image TEST will become zero everywhere afte the call of ImageManipulate().

```
void ImageManipulate(image IMG1) IMG1=0

image TEST
TEST := RealImage("Test",4,256,256)
TEST = icol*irow
ShowImage(TEST)
ImageManipulate(TEST)
```

- In order to pass an identical copy of TEST one can use the command `ImageClone()`:

```
void ImageManipulate(image IMG1) IMG1=0

image TEST2
TEST2 := RealImage("Test2",4,256,256)
TEST2 = icol*irow
ShowImage(TEST2)
ImageManipulate(ImageClone(TEST2))
```

However, remember that using image clones for large and/or many images will be both slow (as the data is copied) and memory consuming (there are now twice as many images!). This is the reason, why the “standard” option is to *assign* the image instead.

Functions: some remarks

- Using functions is good programming style. It makes the code more readable and forces the programmer to break the problem into several tasks. Additional, function can be used more than once in code, making it smaller. Finally, functions can be collected and stored in “libraries” which are then available for all further scripting.
  The message is: Whenever writing anything but a very simple script - use functions!
- It *is* possible to use ‘global variables’ which are valid in both the function and the main program (Simply define the variable above of the function.), but this is bad programming style and should be avoided.
- Always document your functions! It takes you just a few minutes to write down, what exactly the function is doing, but it will save you *hours* if you are going to understand/reuse functions later!
- Use suggestive function names and argument names!
  This helps to quickly realize the syntax of the function later, e.g. use :

```
number ChangeBoxSettings(number BOXid, string name, number width, numer height, number color)
```

  instead of:

```
number CBS2(number n1, string s1, number n2, numer n3, number n4)
```

- Each function name might only be defined *once* (including all loaded libraries). To avoid problems name functions with a prefix which makes it unlikely that the same function name has already been used. Use prefixes to somehow group functions! , e.g. use :

```
number myBOX_ChangeBoxSize(number BOXid, number size)
number myBOX_ChangeBoxName(number BOXid, string name)
number myBOX_CreateBox()
```


TASK T07

Write and test a script (using functions!) which does the following:

1. Take the front most image.
2. Place a *random* selection in it.
3. Ask the user to either *flip* the content horizontally or vertically.
4. Within the flipped area, set a random selection to zero.
5. Repeat 2 to 4 until the user wants to exit the loop. (Use some sort of dialog.)

Hint: The following command prompts a dialog with a question and two buttons of given text. It returns 1 if the first button is pressed, and 0 if the second is pressed:

```
TwoButtonDialog(“Question”,”Choice 1”,”Choice 0”)
```

Hint: The following commands flip an image vertically / horizontally:

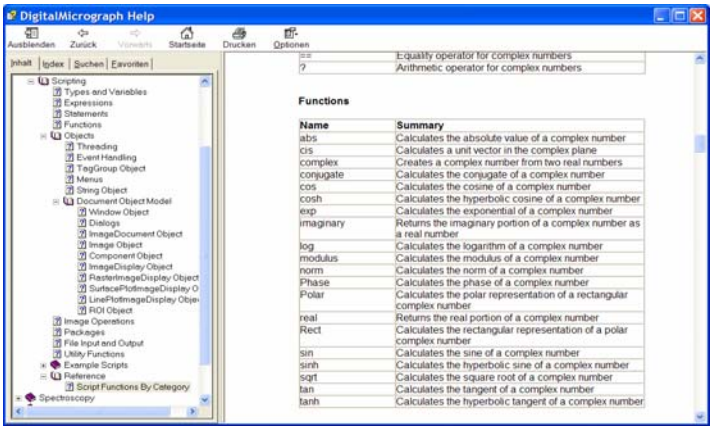
```
FlipHorizontal(IMAGE)
FlipVertical(IMAGE)
```

They can be used on areas only, too:

```
FlipHorizontal(IMAGE[])
FlipVertical(IMAGE[])
```

Where to go from here?

- The help-documentation of DM is a *the* starting point and the only “official” documentation listing most of the available commands. (You launch the documentation by pressing F1 within DM.)
- The DM script database hosted at HTTP://www.felmi-zfe.at has lots of scripts both as simple examples and as useful tools shared by other DM users.
  The scripts and manuscript of this DM course can be downloaded from the site, too.
- At HTTP://lists.asu.edu/archives/dmsug.html you can sign up a mailing list about DM scripting. Please also check the *archives* of the list prior to sending your questions. The answer might already be there!



“Pipeline” syntax

- There exists an alternate syntax for using DM commands which looks strange at first but helps structured programming. The *first* argument of a function can be written as prefix to the command, separated by a dot. The following lines are each equivalent:

```
SetName(IMAGE,"myname")                    <=>  IMAGE.SetName("myname")
GetFrontImage().SetName("myname")          <=>  SetName(GetFrontImage(),"myname")
IF (!IMAGE.ImageIsValid()) exit(0)         <=>  If (!ImageIsValid(IMAGE)) exit(0)
IMAGE.GetSize(sx,sy)                       <=>  GetSize(IMAGE,sx,sy)
```

- The “pipeline” syntax becomes especially useful if results of a function are at the same time the first argument of another function, as it is often the case for more complex commands, as illustrated below. The first line is hardly understandable whereas the second can be easily read and understood. Both lines do exactly the same thing: Take the front most image. Get the according (first) ImageDisplay. In this ImageDisplay get the (first) ROI. Set the label of this ROI to “My ROI”.

```
ROISetLabel(ImageDisplayGetROI(ImageGetImageDisplay(GetFrontImage(),0),0),"My ROI")
                                  <=>
GetFrontImage().ImageGetImageDisplay(0).ImageDisplayGetROI(0).ROISetLabel("My ROI")
```

The difference of == and := and =

- A common source of errors in scripts is the incorrect usage of the operators == , := and =. Though similar in appearance, they have completely different meanings:

  == compares two expressions and results either 1 (equal) or 0 (not equal).
  = *copies the value* of the right expression into the *variable* left of the = operator.
  := *assigns* the *image* right of the := operator to the *variable* left of the := operator.
  (Note that more than one variable can be assigned to the same image!)

- The following examples are typical scripting mistakes (IMGx/NUMx are image/number variables):

```
IF (NUM1=NUM2) result("Equal!")
```
→ This will always return „Equal!“, because the value of NUM2 is copied into NUM1 instead of comparison.

```
IMG1.GetFrontImage()
IMG1 := 2
```
→ This will result an error, because the right hand side is not an image.

```
IMG1 = RealImage("MyImage",4,10,10)
IMG1.Showimage()
```
→ The shown image will have the name “untitled”, because first an image with name “MyImage” is created, but then only the image *values* are copied into a (newly created) image IMG1.

```
IMG2 := IMG1
IMG2 = MedianFilter(IMG2,3,1)
Showimage(IMG1-IMG2)
```
→ The shown image will be exactly zero, because IMG2 and IMG1 are two different names for the *same* image! Changing IMG2 also changes IMG1. Their difference will therefore always be zero.

```
IMG1.GetFrontimage()
IMG2 = IMG1
Result(„Intensity scale of IMG1:")
Result(IMG2.ImageGetIntensityScale())
```
→ IMG2 is *not* an identical copy of IMG1! Just the *values* have been copied, *but not* things like *calibrations*, *image name*, *information tags*… Therefore, this script will not show the proper intensity scale of IMG1.

```
IMG2 := IMG1.ImageClone()
```
→ This line creates an identical copy of IMG1 and assigns IMG2 to this copy.

The difference of == and := and =

- Another typical mistake based on the incorrect use of = and := is shown below. Instead of adding some random noise with each step, and updating the image by `ShowImage()`, each time a new image is created and displayed!

The right hand side of the operator creates a new image without changing IMG1.
Instead of copying these new values into IMG1 and thus changing IMG1 (=),
IMG1 is assigned to this new image (:=).
The old image (still displayed) can no longer be addressed by IMG1,
and `ShowImage()` displays the new image.

```
image IMG1
IMG1 := RealImage("1",4,250,250)
IMG1 = (icol+irow)*itheta
IMG1.ShowImage()
While(OKCancelDialog("Continue adding noise?"))
        {
        IMG1 := IMG1 + mean(IMG1)*Random()
        IMG1.ShowImage()
        }
```

How to create function libraries

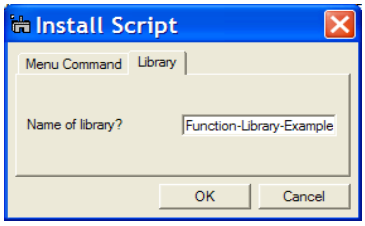
- Functions which are often used should be collected in *libraries*. *libraries* are automatically loaded when DM is started. To install a set of functions as *library*, do the following:

1.) Write a script with the functions, but without a main script:

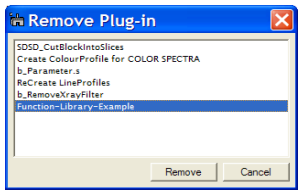
2.) Use the menu **File…/Install Script…**

3.) In the dialog switch to the **Library** tab and enter a name.

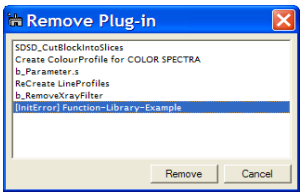
```
void MY_PrintImgType(image IMG)
    {
    IF (IMG.ImageGetDataType()==1)  result("INTEGER 2 Signed")
    IF (IMG.ImageGetDataType()==2)  result("REAL 4")
    IF (IMG.ImageGetDataType()==3)  result("COMPLEX 8")
    IF (IMG.ImageGetDataType()==6)  result("INTEGER 1 UnSigned")
    IF (IMG.ImageGetDataType()==7)  result("INTEGER 4 Signed")
    IF (IMG.ImageGetDataType()==9)  result("INTEGER 1 Signed")
    IF (IMG.ImageGetDataType()==10) result("INTEGER 2 UnSigned")
    IF (IMG.ImageGetDataType()==11) result("INTEGER 4 UnSigned")
    IF (IMG.ImageGetDataType()==12) result("REAL 8")
    IF (IMG.ImageGetDataType()==13) result("COMPLEX 16")
    IF (IMG.ImageGetDataType()==14) result("BINARY")
    IF (IMG.ImageGetDataType()==23) result("RGB 4")
    }

void MY_PrintImgSelection(image IMG)
    {
    number t,l,b,r
    IMG.GetSelection(t,l,b,r)
    result("["+t+"/"+l+"/"+b+"/"+r+"]")
    }
```



4.) The functions are now stored in the preference files of DM (`DigitalMicrographCF.8.prf`) and are automatically loaded on launching DM. The functions are then available in all scripts.



- To deinstall a set of functions, use the menu **File…/Remove Script…** , then select the library you want to deinstall and press “Remove”.



- Note: If the scripts return an error message on installing, you have to first remove them prior to reinstall a (corrected) version. Libraries which failed on installing are marked with an [IntitError].
- Note: You can not install functions with names of functions which already have been installed.

How to allow user action during a script

- The following script shows how to built a script, which runs “in the background” so that DM can be used during the execution of a script. The important step is to put the following first line in a script (It is *not* a comment and needs to be written exactly like this!):

```
// $BACKGROUND$
number count
For (count=0;count<100;count++)
 {
 delay(100)          // wait a second
 OpenAndSetProgressWindow("Waiting script","count "+count+"/100","")
 }
CloseProgressWindow()
```

- The following function can be used to produce a dialog with one button. The calling script halts until this button is pressed, but the user is allowed to work with DM in the meantime. Note, that all scripts using this function need the have the `// $BACKGROUND$` line as first line.

```
// $BACKGROUND$
number T_WaitOnUserDialog(string prompt, string buttonName)
          {
          number sem = NewSemaphore()
          ModelessDialog(prompt, buttonName, sem)
          try GrabSemaphore(sem)
          catch return 0
          return 1
          }

image IMG:=RealImage("Test",4,100,100)
IMG = icol+irow
IMG.ShowImage()
T_WaitOnUserDialog("Please draw selection","OK")
IMG[]=0
```

How to perform a task on all open images

- Scripts used for automatization often need to perform the same task on all shown images. The following script demonstrates how to “circle” through all open (and shown) images.

```
image  current
current.GetFrontImage()
While(current.ImageIsValid())
          {
          current.Setname(current.Getname()+"*")  // Simply a task: Add * to image name
          current := FindNextImage(current)       // Get the next image
          }
```

- `FindNextImage(IMG)` returns the image after IMG in the internal list of displayed images (as shown in the menu Window). If IMG is the last image of the list, the returned image will be *invalid* and can thus be used to find the end of the list.
- Be careful not to change the sorting of the visible images during the script, or you will get unexpected results! The following script will run endlessly, because the 2nd image of the stack will be brought to the front (by `showimage()`) and thus, the next image is again the 2nd of the stack, which will be brought to the front again, etc.

```
image  current
current.GetFrontImage()
While(current.ImageIsValid())
          {
          current.Setname(current.Getname()+"*")  // Simply a task: Add * to image name
          current.ShowImage()                     // This brings the image to the front.
          current := FindNextImage(current)       // Get the next image
          }
```


How to perform a task on all open images

- A safer way to manage “all images” is to create a list of image IDs, and then always use this list to call images by their ID. Newly created images will not be in this list, nor will sorting the images change the list. The following functions are given without explanation and could be installed as library. The script to the right shows how to use these functions.

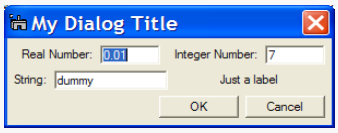
```
/**** ImageList functions ****/

taggroup IL_Create()
  {
  taggroup IMGlist = NewTagList()
  Image  current
  current.GetFrontImage()
  While(current.ImageIsValid())
    {
    IMGlist.TagGroupInsertTagAsLong(Infinity(),current.ImageGetID())
    current := FindNextImage(current)
    }
    return IMGlist
  }

number IL_Size(taggroup IMGlist) return TagGroupCountTags(IMGlist)

number IL_GetID(taggroup IMGlist, number position)
  {
  number ID
  IF (IMGlist.TagGroupGetIndexedTagAsLong(position,ID)) return ID
  return 0
  }

image IL_GetImage(taggroup IMGlist, number position)
  {
  number ID
  image test
  ID = IL_GetID(IMGlist,position)
  If (ID) IF (GetImageFromID(Test,ID)) return test
  result("\n WARNING: Image {"+position+"} not found!\n")
  return RealImage("UNKNOWN IMAGE",4,2,2)
 }
```

```
/**** MAIN PROGRAM ****/
// This short program shows how to use
// the „ImageList“ functions.

taggroup LIST
number   NRimg, i
image    IMG
  // Create a list of all
  // currently displayed images
     LIST  = IL_Create()

  // Get the number of images
  // in the list.
     NRimg = IL_Size(LIST)

  // Always count from 0 to SIZE-1
     for (i=0;i<NRimg;i++)
       {
       IMG := IL_GetImage(LIST,i)
       IMG.Setname(IMG.Getname()+"*")
       }
```

How to create a customized dialog

- The following script shows -without explanation- how to create a simple customized dialog.

```
TagGroup MyDialog, MyDialogItems
Object   MyDialogWindow
taggroup rNUM, iNUM, STR               // These variables will contain the values (RealNumber, IntegerNumber, String)
taggroup f_rNUM, f_iNUM, f_STR         // These variables will contain the dialog-items (including a label)
taggroup f_label                       // A label-field.

// Create Dialog
    MyDialog = DLGCreateDialog( "My Dialog Title", MyDialogItems )
    MyDialogWindow = alloc(UIframe).Init(MyDialog)

// Create items
    f_rNUM = DLGCreateRealField("Real Number:",rNUM,0.01,10,3)
    f_iNUM = DLGCreateIntegerField("Integer Number:",iNUM,7,10)
    f_STR  = DLGCreateStringField("String:",STR,"dummy",20)
    f_label= DLGCreateLabel("Just a label")

// Add items to dialog & define the layout (2 columns, 2 rows)
    MyDialog.DLGAddElement(f_rNUM)
    MyDialog.DLGAddElement(f_iNUM)
    MyDialog.DLGAddElement(f_STR)
    MyDialog.DLGAddElement(f_label)
    MyDialog.DLGTableLayout(2,2,0)


// Show the dialog
    IF (MyDialogWindow.Pose())  result("pressed OK\n")            // Pose() returns 1 on pressing OK
    else                        result("pressed CANCEL\n")        // Pose() returns 0 on pressing CANCEL

// Read the field values
    result("real number   : "+rNUM.DLGGetValue()+"\n")            // instead of using the field values directly
    result("integer number: "+iNUM.DLGGetValue()+"\n")            // you can of course also read the values into
    result("string        : "+STR.DLGGetValue()+"\n")             // variables, e.g. number myReal = rNUM.DLGGetValue()

// Set the field values
    rNUM.DLGValue(12.12)
    iNUM.DLGValue(666)
    STR.DLGValue("NEW DEFAULT")
    f_label.DLGValue("New label")

// Show dialog again!
    MyDialogWindow.Pose()
```



How to open/save a file

- The following script shows how to open an image from within a script.

```
image        IMG
string       filename

IF (!OpenDialog(filename)) exit(0)      // exit program if dialog is “cancelled”
IMG := OpenImage(filename)
IMG.ShowImage()
```

- The following script shows how to save an image from within a script to the hard disk.

```
image        IMG
string       filename

IMG.GetFrontImage()
IF (SaveAsDialog("Please select destination","defaultname.dm3",filename)) exit(0) // exit program if dialog is “cancelled”
IMG.SaveImage(filename)
```

- Sometimes, one wants to save the image “as displayed” in TIFF format rather than as DM file. The following script does this:

```
image        IMG
string       filename

IMG.GetFrontImage()
IMG.SetZoom(1)                                                      // To ensure that 1 pixel remains 1 pixel.
IF (SaveAsDialog("Please select destination","defaultname",filename)) exit(0)   // exit program if dialog is “cancelled”
IMG.CreateImageFromDisplay().SaveAsTiff(filename)
```

Advanced area indexing: more on icol and irow

- **icol** and **irow** can also be used to address subareas of an image. The syntax is different then. Assume you have an image IMG of size 6x6. You want to create an image SUB which is a subset of IMG. Then you *first* create the image SUB of wanted size, and *then* copy the values using **icol** and **irow** as shown by the examples below.

```
image SUB
SUB := Realimage("",4,1,6)
SUB = IMG[3,irow]
```

3,0
3,1
3,2
3,3
3,4
3,5
3,6

```
image SUB
SUB := Realimage("",4,3,1)
SUB = IMG[icol,2]
```

0,2	1,2	2,2

Image IMG, values are coordinates

0,0	1,0	2,0	3,0	4,0	5,0	6,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5
0,6	1,6	2,6	3,6	4,6	6,5	6,6

```
image SUB
SUB := Realimage("",4,5,3)
SUB = IMG[icol,0]
```

0,0	1,0	2,0	3,0	4,0
0,0	1,0	2,0	3,0	4,0
0,0	1,0	2,0	3,0	4,0

The values are automatically contiued to fill the image!

```
image SUB
SUB := Realimage("",4,3,3)
SUB = IMG[5+irow,3+icol]
```

5,3	5,4
6,3	6,4

Note that the image has also been transposed by changing irow ←→icol

- Note: **icol** and **irow** can be used on either side of the = operator, but not on both at the same time, as the other side is used to determine the expression size (and therefore the range of **icol** and **irow**)

Advanced area indexing: slices

- Instead of addressing image areas as simple rectangular regions, it is possible to define areas by a *starting point*, and a *step-number* plus *step-size* <u>for each</u> dimension. This is possible for one, two or three dimensions. One first defines a slice-object and can then apply it to images. The following two examples should illustrate this:

Image IMG, values are coordinates

0,0	1,0	2,0	3,0	4,0	5,0	6,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5
0,6	1,6	2,6	3,6	4,6	6,5	6,6

```
object sliceIt = NewImageDataSlice(2,1,3,3,0,3,1)
IMG[sliceIt].ShowImage()
```

position in source is given 2-dimensional by these (two) coordinates

1st-dim of result

direction (0=x)

step-size

result is 1-dimensional

step-numer (= size of result!)

3,3	4,3	5,3

Image IMG, values are coordinates

0,0	1,0	2,0	3,0	4,0	5,0	6,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5
0,6	1,6	2,6	3,6	4,6	6,5	6,6

```
object sliceIt = NewImageDataSlice(2,2,1,3,1,-1,0,3,2)
IMG[sliceIt].ShowImage()
```

position in source is given 2-dimensional by these (two) coordinates

1st-dim of result

2nd-dim of result

direction (1=y)

step-size

direction (0=x)

step-size

result is 2-dimensional

step-numer (= size of result!)

1,3	1,2	1,1
3,3	3,2	3,1
5,3	5,2	5,1

TASK T01 - SOLUTION

```
// This is the first line of my script! Just a comment.
// The purpose of the script is to demonstrate the RESULT command.
result("My name is Bernhard Schaffer")
result("\n\n\n\n")
result("And who are you?")    // I will always comment my scripts!
```


TASK T02 - SOLUTION

```
ShowImage( tert( A>mean(A) && B>mean(B), 1 , 0 ) )

ShowImage( tert( A>=(mean(A)-SQRT(variance(A)))&& A<=(mean(A)+SQRT(variance(A))) , A , 0 ) )
```

TASK T03 - SOLUTION

```
ShowImage(tert(A>mean(A[0,0,50,50]) && B>mean(B[0,0,50,50]), 1 , 0 ))

ShowImage(tert(A>=(mean(A[])-SQRT(variance(A[])))&& A<=(mean(A[])+SQRT(variance(A[]))) , A , 0 ))
```


TASK T04 - SOLUTION

```
number SIZEX,SIZEY
string NAME
image  MYIMAGE

GetNumber("Enter size X:",100,SIZEX)
GetNumber("Enter size Y:",100,SIZEY)
GetString("Enter name:","task",NAME)

MYIMAGE := RealImage(NAME,4,SIZEX,SIZEY)
MYIMAGE[0,0,SIZEY/2,SIZEX]              = 100
MYIMAGE[SIZEY/2,0,SIZEY,SIZEX]          = 200
MYIMAGE[0,0,SIZEY,SIZEX/2]              *= -1
ShowImage(MYIMAGE)
```

TASK T04b - SOLUTION

```
image  IMG
number t,l,b,r,ratio
number wx,wy,zoom

GetFrontImage(IMG)
GetSelection(IMG,t,l,b,r)
SetSurvey(IMG,0)
SetLimits(IMG,min(IMG[]),max(IMG[]))

ratio = (r-l)/(b-t)
GetWindowSize(IMG,wx,wy)

IF (ratio<1) SetWindowSize(IMG,ratio*wy,wy)
ELSE         SetWindowSize(IMG,wx,wx/ratio)

SetWindowPosition(IMG,145,35)
GetWindowSize(IMG,wx,wy)

zoom = min(wy/(b-t),wx/(r-l))
SetZoom(IMG,zoom)

SetImagePositionWithinWindow(IMG,-l*zoom,-t*zoom)
```


TASK T05 - SOLUTION

```
image  MYIMAGE
number MINVALUE, MAXVALUE, LIMIT, RANGE
number COUNT, SIZEX, SIZEY, MEANVALUE, BRIGHTPIXEL

GetFrontImage(MYIMAGE)
GetSize(MYIMAGE,SIZEX,SIZEY)
MINVALUE = min(MYIMAGE)
MAXVALUE = max(MYIMAGE)
RANGE = MAXVALUE-MINVALUE
```

```
For (COUNT=2;COUNT<=100;COUNT+=2)
 {
 LIMIT = MAXVALUE - RANGE*COUNT/100
 BRIGHTPIXEL = sum(tert(MYIMAGE>=LIMIT,1,0))
 IF ( BRIGHTPIXEL >= 0.5 * SIZEX*SIZEY ) break
 }
```

```
BRIGHTPIXEL = 0
LIMIT = MAXVALUE
while(BRIGHTPIXEL<0.5*SIZEX*SIZEY)
 {
 LIMIT -= RANGE*2/100
 BRIGHTPIXEL = sum(tert(MYIMAGE>=LIMIT,1,0))
 }
```

```
MEANVALUE = sum(tert(MYIMAGE>=LIMIT,MYIMAGE,0)) / BRIGHTPIXEL

result(" LIMIT value:"+limit+"\n")
result(" MEAN  value:"+MEANVALUE+"\n")
showimage(tert(MYIMAGE>=LIMIT,MYIMAGE,0))
```

TASK T06 - SOLUTION

```
image     POINTS, LINEPLOT
number    k, d, scale, origin
number    s_x, s_y, s_xx, s_xy

POINTS := [2,6] :
          {
          {2,12.5},
          {4,23.8},
          {7,43.2},
          {9,52.6},
          {11,67.7},
          {12,71.8}
          }

s_x       = sum(POINTS[0,0,5,1])
s_y       = sum(POINTS[0,1,5,2])
s_xy      = sum(POINTS[0,0,5,1]*POINTS[0,1,5,2])
s_xx      = sum(POINTS[0,0,5,1]*POINTS[0,0,5,1])
k         = (6*s_xy - s_x*s_y ) / (6*s_xx-s_x*s_x)
d         = (s_y-k*s_x)/6

result("Linear regression yields: k:"+k+" d:"+d+"\n")

LINEPLOT := RealImage("Graph",4,200,1)
scale     = 0.1
origin    = -10
LINEPLOT  = (origin+scale*icol)*k+d
Showimage(LINEPLOT)

SetScale(LINEPLOT,scale,1)
SetOrigin(LINEPLOT,-origin/scale,1)
```


TASK T07 - SOLUTION

```
void T_PlaceRandomSelection(image IMG)
  {
  // This function places a random selection anywhere within the current selection of the image
  number t,l,b,r
  number t2,l2,b2,r2
  GetSelection(IMG,t,l,b,r)
  t2 = t + (b-t)*Random()
  l2 = l + (r-l)*Random()
  b2 = b - (b-t)*Random()
  r2 = r - (r-l)*Random()
  SetSelection(IMG,t2,l2,b2,r2)
  }

void T_FlipHorizontal(image IMG)
  {
  // This function flips the selected area horizontally and sets a random areas within to zero
  FlipHorizontal(IMG[])
  T_PlaceRandomSelection(IMG)
  IMG[]=0
  ClearSelection(IMG)
  }

void T_FlipVertical(image IMG)
  {
  // This function flips the selected area vertically and sets a random areas within to zero
  FlipVertical(IMG[])
  T_PlaceRandomSelection(IMG)
  IMG[]=0
  ClearSelection(IMG)
  }

image FRONT
GetFrontImage(FRONT)
ClearSelection(FRONT)
While (OkCancelDialog("Another step?"))
  {
  T_PlaceRandomSelection(FRONT)
  IF (TwoButtonDialog("Flip the area","horizontally","vertically")) T_FlipHorizontal(FRONT)
  Else T_FlipVertical(FRONT)
  }
```