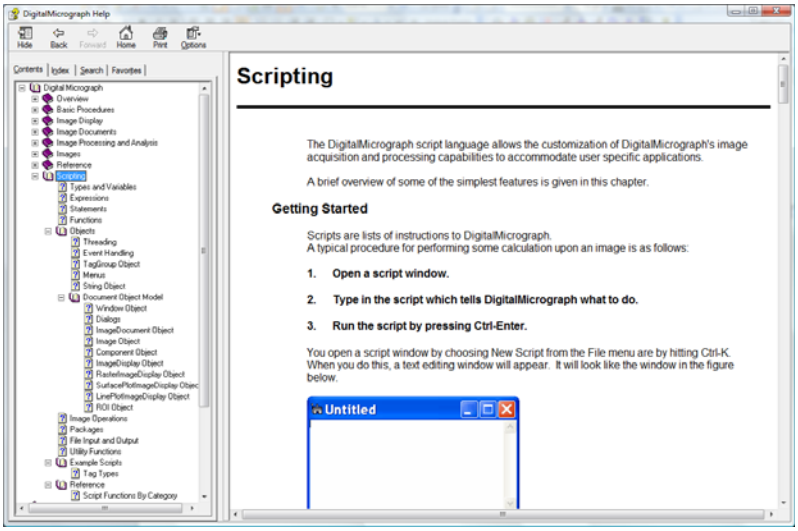
How to start

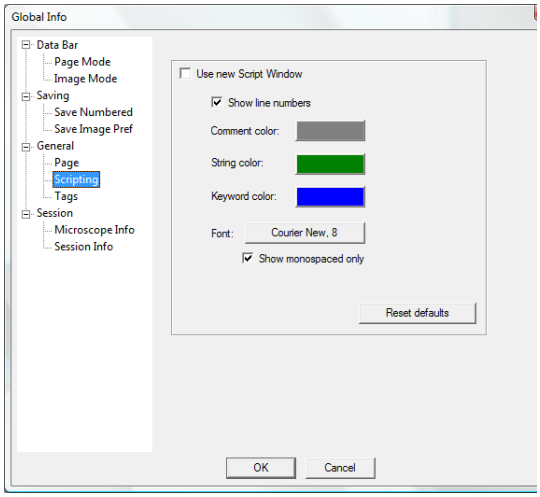
This tutorial is meant for the impatient. If you have not scripted in DigitalMicrgraph before, this should give you a quick feeling of what is possible and explain some of the basics.

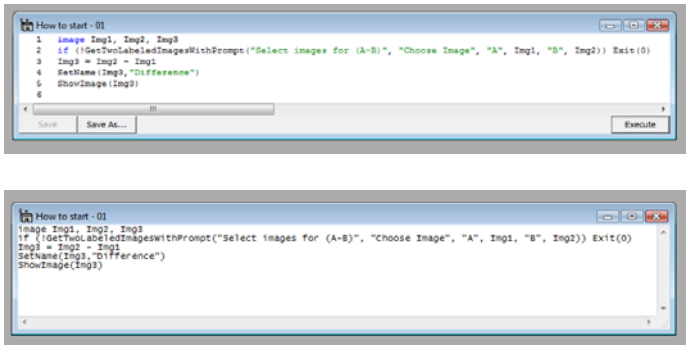
```
ShowImage(); SetName(); GetTwoLabeledImagesWithPrompt(); Exit();
```

It is not the intention of this tutorial to be a 'manual' to DigitalMicrograph scripting, but as with all things, one has to start at one point sooner or later. A lot of what is said here can also be re-read in the official documentation of DigitalMicrograph which is accessed by the *F1* key from within the application. In fact, I strongly recommend reading that documentation sooner or later.



Scripts are simple plain text files which can be written in any text-editor of your choice. Usually, they are stored with the *.s file extension, but that is not a requirement. For your convenience, DigitialMicrograph itself allows text-windows, and has a dedicated *script-editor* window with some code-visualization features. The script-editor feature can be enabled/disabled and customized in the *Global Info* dialog:





All scripts are written in a syntax loosely based on the C++ language syntax, but with a lot of simplifications. While strict C++ syntax will be mostly accepted, the constraints are less strict. The code is case insensitive, and individual statements do not need to be terminated with the semi-colon sign as long as there is only one statement per line. White-spaces and tabulator-jumps are generally ignored, and too long lines can be split into multiple lines using a single backslash \ sign. The biggest simplifications concerns variables. Integer and floating-point numbers are handled within a single 'number' type, number arrays (in up to 5 dimensions) are handled as simple 'image' type. In many simple cases, images can be directly addressed with *implicitly defined variables.*

While the syntax is similar to C++ code, DM scripting is not a compiler language like C++. Instead, the code is interpreted line-by-line on execution, and some coding errors may only then become apparent. However, to avoid small mistakes spoiling large scripts (-usually horribly difficult to debug-), DigitialMicrograph performs a quick syntax-check before executing the code. If an obvious error is found, a (sometimes) helpful error-message is presented and the script is not executed at all.

Now let us start! We assume in the following examples that two (arbitrary but equally sized) images have been opened in DigitialMicrograph. If not, do so now. All image-windows have titles with one or two letters separated by a colon from the image's name. These letters do not belong to the image and are reassigned each time a new image is created or loaded. We will hence refer to them as *image-letter* and we call images by this letter directly, i.e. "Image XY" refers to the image which currently displays "XY: …" in its window title. Now the same thing can be done in a script. Even better, if an *image-letter* is used and the according image does not yet exist, DigitalMicrograph will create and display it. This makes "on-the-fly" image processing really simple, as shown in the following script. You will likely have to replace the *image-letters* used in the examples (and coloured **blue**) by appropriate ones. First, open a new script window: Press CTRL+K or use *File/New Script…* from the menu. Next, type the example below in the script window. Finally, execute your first script: Press CTRL+ENTER

```
ZZ = A - B
SetName(ZZ,"Difference")
```

If you haven't got an error message, you should see a new image with image-letter ZZ and the title "Difference". You can do all sorts of simple (or complex) image manipulation using such short scripts, and they are far more powerful than the cumbersome *Process/Simple Math…* manipulations from the menu.
What, you did get an error message? What could have gone wrong?

- You did not replace **A** and **B** by the according image letters of <u>your</u> displayed images. (Yes, I really mean your screen right in front of you now.)
- There was already an image with letter **ZZ** open when you ran the script. (Just use any other unused letter…).
- Images **A** and **B** were not of same size.
- You misspelled any of the lines above.

Implicit image variables – the use of image-letters to address the images – are a nice and quick way of doing simple tasks on-the-fly, but they become rather useless in scripts which you want to use repeatedly, unless you like replacing image-letters in your code over and over again. Let us thus expand our script into a more typical form:

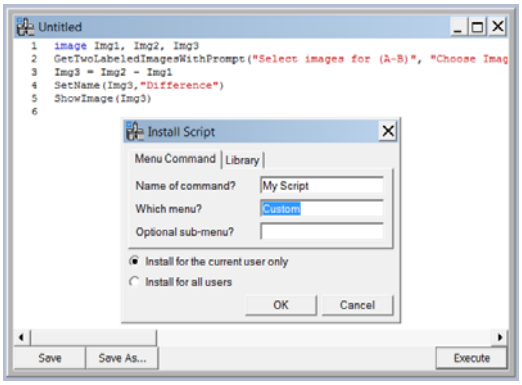
```
image Img1 := A             // We define and assign an image variable in one line here, but
image Img2, Img3            // we could define more variables – comma separated – at once, and
Img2 := B                   // do the assignment in a separate line.
Img3 = Img2 - Img1          // While := assigns an image, = copies the values only. We will come back to this later
SetName(Img3,"Difference")
ShowImage(Img3)             // This command makes a new image visible (and gives it an image-letter)
```

What have we done? First, we've demonstrated that everything right of a double-backslash // is considered comment-text and will be ignored in a script. (This is very useful, if you want to temporarily remove a code line. Don't delete it, just put a // in front!) If you are an inexperienced programmer, let me give you one important advice: **<u>Always comment your scripts!</u>** In a year's time you will have forgotten what you are thinking now, and your future-you will appreciate any help it can get - not to speak of all *other* people who might want to understand your code! If you are an experienced programmer: Remember that you have been told to comment your scripts? Do it!
As for the remaining code: We have started using real variables of the type *image*. In the first three lines, we have defined three image variables and *assigned* the images A and B to it. In fact, Img1 *is* image A and Img2 *is* image B after these lines. We've just given the two images new names which we have used from there on. Even if it is going to be a long script with a lot of usage of the two images, only the two assignments will need changes later on, when you want to reuse your script with different images. Img3 is a bit different. We have defined (but not created!) it in the second line. In the 4th line, we have calculated the difference between Img2 and Img1 and then copied the values into Img3. Because Img3 doesn't exist at this moment, DigitalMicrograph creates it for us! (This is one of the nice simplifications of scripting over strict programming languages like C++.) However, although Img3 is now created, it is not yet displayed. Consequently, after the script finishes, it will be deleted from memory again. We are thus using the command ShowImage() to display it. Displayed images do not get destroyed when a script finishes. If you use ShowImage() on an image which is already displayed, it will not display it again, but force an immediate update of its display.
The script above minimized the need for "adaptation" considerably, but we can still do better: Remove the need for "adaptation" and get user-interaction instead!

```
image Img1, Img2, Img3
GetTwoLabeledImagesWithPrompt("Select images for (A-B)", "Choose Image", "A", Img1, "B", Img2)
Img3 = Img2 - Img1
SetName(Img3,"Difference")
ShowImage(Img3)
```

In this version we removed all image-letters and use the rather long command GetTwoLabeledImagesWithPrompt() instead. This command pops up a user dialog allowing the selection of two images which will then be assigned to the given image variables. The dialog has two buttons, an *OK* and a *Cancel* button, but in both cases the remaining code

will be executed (We will soon show how to 'catch' the cancellation event.). All remaining code stays the same. Now that we do no longer need to alter our script for different images, we can become even more user-friendly, and install our script as a menu command. This is done by having the script window front-most and selecting the menu *File/Install Script…* . The dialog asks for a menu name, an optional sub-menu name, and a command name, and after clicking *okay*, our script can be accessed from a newly created menu entry! To remove this menu entry, use the menu *File/Remove script…* and select the according entry from the dialog. It is as easy as this.



Before finishing this tutorial, let us make the code a tiny bit better, by introducing an if-statement. As the name implies, the if-statement is only executed *if* some condition is fulfilled. This condition has to be given as the argument of the if() command, and – when evaluated – has to result in a single value. Only if this value is exactly 0 – the numerical equivalent of *false* – will the conditioned statement *not* be executed. Most dialog commands such as GetTwoLabeledImagesWithPrompt() return a value depending on the button pressed by the user. OK usually returns 1, and Cancel returns 0. We can thus directly use the command as a condition for the if-statement. We now have two options: We can write the script such that all the code is only evaluated *if* the user clicks OK, or we can immediately stop the script, *if* the user clicks Cancel. The second choice is easier to understand in large scripts, when multiple breaking conditions would produces a cascade of 'encapsulated' code – one for each condition. The command to immediately stop a script is called Exit(), and we are going to use it in the if-statement. No we have one final hurdle: The command will only be executed if the dialog returns anything but *not* 0 – and 0 is exactly the value we will get once the user clicks on Cancel! The way out is the logical operator *NOT* which turns *true* into *false* and vice versa. Numerically wise, it turns 0 into 1 and *all* other numbers into 0. This operator is abbreviated by a single exclamation mark (!) immediately before the expression, such that our final script is:

```
image Img1, Img2, Img3
if (!GetTwoLabeledImagesWithPrompt("Select images for (A-B)", "Choose Image", "A", Img1, "B", Img2)) Exit(0)
Img3 = Img2 - Img1
SetName(Img3,"Difference")
ShowImage(Img3)
```

The if-statement might read complicated, but if you read it out aloud, it is rather simple: "If Not Get-Two-Labeled-Images-With-Prompt Exit".
Oh, and if you wonder about the 0 in Exit(). The command also returns a *true*/*false* value – which could be caught by the program which called the script, i.e. DigitalMicrograph. It is good coding practise to ensure that such *exit codes* are logically correct. Our script would work exactly the same with a 1 instead of the 0 in the Exit(), but as our script is "not successful" when we stop it, it is sensible to return 0 or *false*.

How to perform a task on all open images

For work automation and batch processing, scripts are needed which perform automatically on all currently opened and/or visible images. The main idea is to build a loop which takes one image after the other and then performs some tasks with it. This loop should 'circle' through all images which are currently opened in the program.

batch processing; image management; loop; While(); FindNextImage(); ImageIsValid(); SpaceDown(); ImageGetID();

The first script is very simple. It takes the first visible image – the one which is the front most – and then enters a while loop. It relies on the fact that an image variable may either contain (or point to) an image or nothing. The latter is equivalent to containing an *invalid* image. Within the loop, it performs a task (– adding * to the image name –) and then gets the 'next' *visible* image after the currently selected one, following a list of all displayed images (It is the same list as could be seen in the *Window* menu of DM, but *hidden* images are ignored.) If the list is at its end, the 'next' image does not exist and the variable points to an *invalid* image, thus stopping the loop.

```
Image img
img.GetFrontImage()
While(img.ImageIsValid())
   {
   img.SetName(img.GetName()+"*")
   img := FindNextImage(img)
   }
```

Though functional, there is one problem with this script. As the FindNextImage() command relies on the 'internal sorting list', any change in image sorting will screw up this script. This is illustrated in the script below. Within the loop, the selected image will be brought to front by using the ShowImage() command. At the same time, this means that the image becomes the first one in the 'internal sorting list', so that the 'next' image is the second, which again becomes automatically the first due to the script... This makes this loop a never ending one, so we add another breaking condition with the logical AND (&&) to the script, the SpaceDown(). This function returns 1 as soon as the space-bar is pressed on the keyboard, so we make the condition with the logical NOT (!) to have the loop being quit on pressing the key.

```
Image img
img.GetFrontImage()
While(img.ImageIsValid() && !SpaceDown())
   {
   img.SetName(img.GetName()+"*")
   img.ShowImage()
   img := FindNextImage(img)
   }
```

So what can we do to avoid this problem? Instead of building a simple loop around our tasks we do the script in two steps: First, we register all images into our own list, and then we go through this list one by one performing our tasks. This has the additional advantage that we can filter or expand our list as we wish. We use the ImageGetID() command, which returns a unique number for each image in memory, by which the image can then be addressed. Note however, that this ID is not permanently attached to an image, but only to the image in memory. Whenever a new image in DM is created (or loaded) it gets a new, unique ID number.
A good way of keeping structured information is to make use of the *Tag* concept of DM. As we will quite likely use our 'list-management' more often, it is practical to put this functionality into a set of functions. The first one creates a TagList containing the image IDs of all images as list entries.

```
/*****************************************************************/
/* ImageList Functions – a set of functions to manage open images */
/*****************************************************************/

/* IL_Create() creates a list (as a TagList) of all images. */
TagGroup IL_Create()
   {
   TagGroup img_list = NewTagList()
   Image  img
   img.GetFrontImage()
   While(img.ImageIsValid())
      {
      img_list.TagGroupInsertTagAsLong(Infinity(),img.ImageGetID())
      img := FindNextImage(img)
      }
   return img_list
   }
```

The following functions all take the created list as input and return information derived from it: The size of the list, the ID of an image at given position, the image at a given position. Note, that the image is passed as a reference (with leading &).

```
/* IL_Size() returns the size of the image list. */
Number IL_Size(TagGroup img_list)
   {
   return TagGroupCountTags(img_list)
   }

/* IL_GetID returns the ID of the image stored at list position list_pos. It returns 0 for invalid positions */
Number IL_GetID(TagGroup img_list, Number list_pos)
   {
   Number ID
   If (img_list.TagGroupGetIndexedTagAsLong(list_pos, ID)) Return ID
   Else Return 0
   }

/* IL_GetImage gives the image stored at list position list_pos. It returns 1 on success and 0 on failure */
Number IL_GetImage(TagGroup img_list, Number list_pos, Image &img)
   {
   Number ID
   ID = IL_GetID(img_list, list_pos)
   If (ID) If (GetImageFromID(img, ID)) Return 1
   Else Return 0
   }
```

This set of functions can be easily expanded by list-managing functions, for example filters, as shown next. The function takes the image list and returns a new image list containing only ‘true’ images which are 2D. This is done by copying only those entries which do match the criterion of having two dimensions with a length greater than one pixel.

```
/* IL_Filter2D() returns a reduced image list of 2D images */
TagGroup IL_Filter2D(TagGroup img_list)
   {
   Number count, img_list_size, tg_entry, size_x, size_y
   Image img
   TagGroup img_list_filtered
   img_list_size = IL_Size(img_list)
   If (img_list_size == 0) Return img_list
   img_list_filtered = NewTagList()
   For (count = 0 ; count < img_list_size ; count++)
      {
      If (IL_GetImage(img_list, count, img))
         {
         img.GetSize(size_x, size_y)
         If ((img.ImageGetNumDimensions()==2) && (size_x>1) && (size_y>1) )
            {
            img_list.TagGroupGetIndexedTagAsNumber(count, tg_entry)
            img_list_filtered.TagGroupInsertTagAsNumber(Infinity(), tg_entry)
            }
         }
      }
   return img_list_filtered
   }
```

The last script shows how to use the ImageList functions to perform task on all images. It simply prints the name of all currently visible images, and then another list of really 2D images. Please note that all loops count beginning with 0, as TagLists start with index 0 for the first entry.

```
TagGroup my_list
Number   count, list_size
Image img
my_list = IL_Create()
my_list = IL_Filter2D(my_list)
list_size = IL_Size(my_list)
Result("\n 2D images:")
For (count = 0; count < list_size; count++)
   {
   If (!IL_GetImage(my_list, count, img)) result("\n Unexpected error: image not found")
   Else Result("\n -->: "+img.GetName())
   }
```