Kota Miura

EMBL-CMCI course II

Macro Programming in ImageJ

ver 2.0.3

Centre for Molecular & Cellular Imaging
EMBL Heidelberg

EMBL /CMCI

Abstract

Aim: Students acquire ImageJ macro programming technique to ease their work loads with image processing / analysis.

Note: This textbook was written using Fiji (ImageJ 1.44e). When you want to distribute, please ask Kota as this textbook is progressively edited.
Compiled on May 15, 2014

© 2006 - 2012, Kota Miura (http://cmci.embl.de)

Contents

2.1 Why do we write macro?

. . . to make your life easier by:

2.1.1 Automate your mouse clicking!

ImageJ commands can be listed in a text file as a sequence of event and then executed as a single task. Writing such a text file is called scripting, or writing a "macro".

For example, if you want to do
`[Process -> Contrast Enhancement],`
apply `[Process -> Gaussian blurring]`
and then `[Process -> threshold]`
to an image, it takes three times of mouse operation choosing these commands from ImageJ menu. If you need to do this for many images, the work load would become pretty heavy. Writing a macro assembles such works into a single task. Macro becomes even more powerful when you need to process/analyze stacks.

2.1.2 Generate new functions which do not exist in the menu or as a PlugIn.

Automation of a series of command is very convenient, but the macro programming is not only limited to such automation. ImageJ macro language enables you to access single pixels in digital images. This means that when you know an algorithm for processing/analysis but the function is not implemented in the ImageJ itself or as a PlugIn, one could write a custom function using the macro language.

2.1.3 Running your job in remote server

ImageJ could be used from command line, and macro file could be executed by adding the file name to the java command. This enables you to do the calculation with faster machines like the EMBL cluster. If you use job array

many jobs could be done in parallel so you could finish your calculation much faster.

2.1.4 Limitations compared to Plugins written in Java

In addition to Macro programming, ImageJ has plugin-writing capability. This enables one to add new functions to ImageJ by coding in Java programming language. This capability affords almost infinite possibility to process and analyze images; you could create any kind of processing / analysis functions you could imagine. Compared to plugins, ImageJ Macro language has some limitations:

1. If you need to process large images or stacks with many steps, you might recognize that it is slow. Some benchmarks indicate that a plugin would be about 40 times faster than Macro.

2. Macro cannot be used as a library [1]. In Java, once a class is written, this could be used later again for another class.

3. Macro is not efficient in implementing real-time interactive input during when the macro function is executed; *e.g.* If you want to design a program that requires real-time user input to select a ROI interactively. Macro could only do such interactive tasks by closely related macro set with each macro doing each step of interaction.

4. Macro is tightly coupled to GUI (Image Window), so that when you want to process images without showing them on desktop, macros are not really an optimal solution.

If you become unsatisfied with these limitations, learning more complicated but more flexible Java plugin development is recommended.

---

[1] It is possible to write a macro in a library fashion using the function `eval` and use it from another macro, but this is not as robust and as clear as it is in Java, which is a language designed to be so.

2.1.5   Comparison with Other scripting languages

Besides ImageJ macro, there are several scripting languages that could be used for programming with ImageJ. The bare ImageJ supports Javascript (Rhino). Recent version of ImageJ (> 1.47m, since 6 March 2013), Jython became included in the menu as well. In the Fiji distribution, you could use the following languages off the shelf:

- Javascript

- BeanShell

- Jython (Java implemented Python)

- JRuby (Java implemented Ruby)

- Clojure

If you set up an environment by yourself, other languages such as Scala and Groovy are also available. Compared to the ImageJ macro language, all these languages are more general scripting languages also used outside ImageJ.

There are pros and cons. Merits of using the ImageJ Macro compared to these scripting languages are:

- Easy to learn. ImageJ macro build-in functions are mirrors of ImageJ menu, so scripting is intuitive if you know ImageJ already. Macro recorder is a handy tool for finding out the macro function you need.

- A significant hurdle for coding with general scripting languages is that one must know the **ImageJ Java API** well, meaning that you basically have to know fundamentals of Java programming language for using these scripting languages.

- You could have multiple macros in one file (called 'Macro-set"). This is useful for packaging complex processing tasks.

Thus, ImageJ macro language is the easiest way to access the scripting capability of ImageJ.

There are several disadvantages of ImageJ macro compared to other scripting languages. First is its generality. Since others are based on major scripting languages, you do not need to learn a lot if you know one of them already. For example, if you know Python already, it should be easy for you to start writing codes in Jython (note: but you also need to know about Java).

The second disadvantage of ImageJ macro is its extendability. Codes you wrote could only be recycled by copying and pasting [2]. With other scripting languages, once you write a code, it could be used from other programs [3].

Lastly, although ImageJ Macro processes with a speed comparable to Javascript and Jython, it is slow compared to Clojure and Scala.

2.1.6 Summary

By programming ImageJ Macro, your work load radically decreases by automating sequence of commands - less clicking. In addition, capability to access single pixels allows you to analyze details of images. The potential of the macro is similar to PlugIn, both adding capability for tailoring your original functions. But for interactive functions PlugIn works better and macro cannot be used as a library. Image processing by Macro is slower than by PlugIn.

How to learn Macro programming: In this course, you will encounter many example codes. You will write example codes using your own computer and run those macros. Modifying these examples by yourself is a very important exercise since in most cases, you start writing your macro by modifying some parts of the macro that are available already [4].

---

[2]One could also use getArgument() and File related functions to pass arguments from a macro file to the other, but ImageJ macro is not designed to construct a library of functions.

[3]Calling other Javascript file from another Javascript file had been difficult but became easily possible in the Fiji distribution from March 2012.

[4]200+ macros are available in ImageJ web site. http://rsb.info.nih.gov/ij/macros/

2.2 Basics

2.2.1 "Hello World!"

We first try writing a simple macro that prints "Hello World!". ImageJ has a macro editor with simple debugger function [5]. To open the editor, select `[PlugIns -> New -> Macro]` from the menu. This will create a new window where you can write macro (we call this "macro editor", fig**??**).

> Fiji: You could use more advanced interface called "script editor" by `[File -> New -> Script]`. It should look like **??**. From its menu, select `[Language -> ImageJ Macro]`. Syntax high lighter offers automatic coloring of ImageJ functions.

Then write your first macro as shown below. In the second line DON'T forget to indent the line using tab or spaces[6]. Omit the line numbers! These numbers were added just for explanation.

```
1 macro "print_out" {
2   print("Hello World!");
3 }
```

<div align="center">code/code01.ijm</div>

From the macro editor menu, running the code by `[Macros-> Run Macro]`[7]. You could also run the code by shortcut keys (Windows: ctrl-r, OSX command-r) as well.

> Fiji: Use `[Run -> Run]` from the script editor menu. Shortcut keys are same as in ImageJ. You could also use "run" button in the script editor.

This will create a new window "Log". Within the log window, "Hello World" will be printed.

Explanation for the Code 1:

---

[5]Debugger assists you to correct mistakes in the code. This is convenient when the code becomes long. Macro can be written in any text editor such as "Notepad" in Windows but
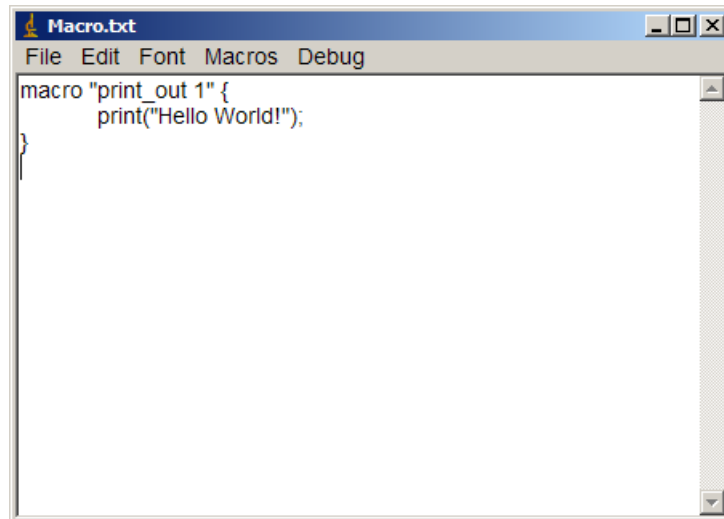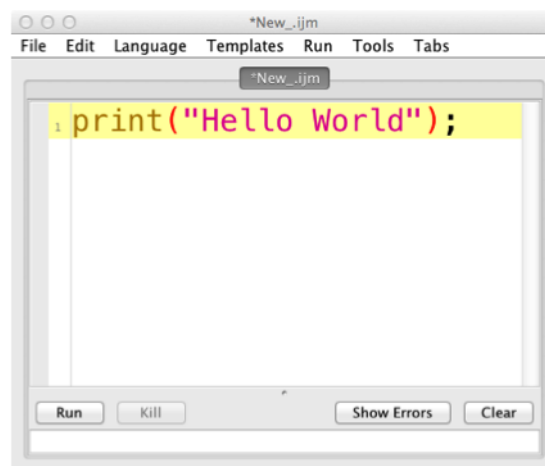
Figure 2.1: Macro Editor of ImageJ

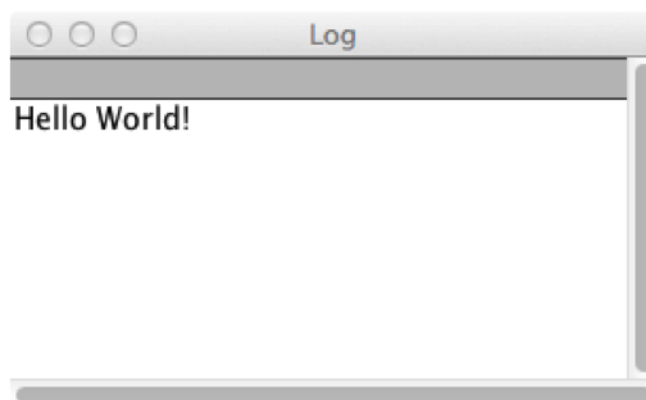Figure 2.2: Script Editor of the Fiji distribution

Figure 2.3: Hello World Output

- line 1: You are declaring that a macro code starts and the code is contained between curly braces {}. "print_out" will be the name of macro.

- line2: print() function orders ImageJ to print out the content within the parenthesis in the "Log" window. The text to be printed must be contained within the double quotes (""). The best reference for ImageJ macro functions is in the ImageJ web site [8]. For example, you could find definition of print("") function on the web site as quoted below:

> **print(string)**
>
> Outputs a string to the "Log" window. Numeric arguments are automatically converted to strings. The print() function accepts multiple arguments. For example, you can use print(x,y,width, height) instead of print(x+" "+y+" "+width+" "+height). If the first argument is a file handle returned by File.open(path), then the second is saved in the referred file (see SaveTextFileDemo).
>
> Numeric expressions are automatically converted to strings using four decimal places, or use the `d2s` function to specify the decimal places. For example, print(2/3) outputs "0.6667" but print(d2s(2/3,1))

---

of course there is no debugger function available in this case.

[6]In ImageJ macro, indenting is not a required syntax for writing macros but doing this will be very very helpful afterward. You will understand it as the macro you write becomes longer

[7]"Macros" in the menu appears only when the macro editing window is active

[8]http://rsbweb.nih.gov/ij/developer/macro/functions.html

outputs "0.7".

- line 3: a brace tells ImageJ that the code "print_out" finishes at this line.

So that was the very basic of how you use a macro. To integrate the macro into the ImageJ Menu bar, the macro must be "installed". To do so, in the editor menu, `[Macros -> Install Macros]`

Fiji: [Run -> Install Macro]).

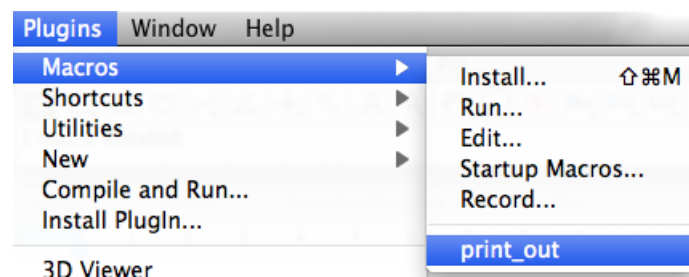Check IJ menu `[Macros -> ]` to see that the macro is now in the menu.

Figure 2.4: Macro Now in ImageJ menu

Macro can be saved as a file and can be directly installed also. In the editor, do `[File -> Save]`. Saving dialogue window appears, and just save the file wherever you can remember afterwards . To install the macro, do `[PlugIns -> Macro -> Install...]` Select the macro file you want to install.

Exercise 2.2.1-1

Add another line `"print("\\Clear");"` after the second line (below, code 1.5. don't forget the semi-colon at the end!).

```
1 //code 1.5
2 macro "print_out 1.5" {
3   print("\\Clear");
```

```
4   print("Hello World!");
5 }
```

<div align="center">code/code01_5.ijm</div>

Then test also another macro when you insert the same function in the third line (code 1.75). What happened?

```
1 //Code 1.75
2 macro "print_out 1.75" {
3   print("Hello World!");
4   print("\\Clear");
5 }
```

<div align="center">code/code01_75.ijm</div>

Exercise 2.2.1-2

Try modifying the third line in code 1.5 and check that the modified text will be printed in the "Log" window.

Exercise 2.2.1-3

Multiple macros can exist in a single file. We call this **"macro sets"**. Duplicate the code you wrote by copying and pasting it under the first macro. The second macro should have a different name. In the example shown in fig. **??**, the second macro is named "pirnt_out2".

When macro is properly declared in this way, you could install the macro to have it as a menu item. To do so, in the editor menu select:

  [Run -> Install Macro]).

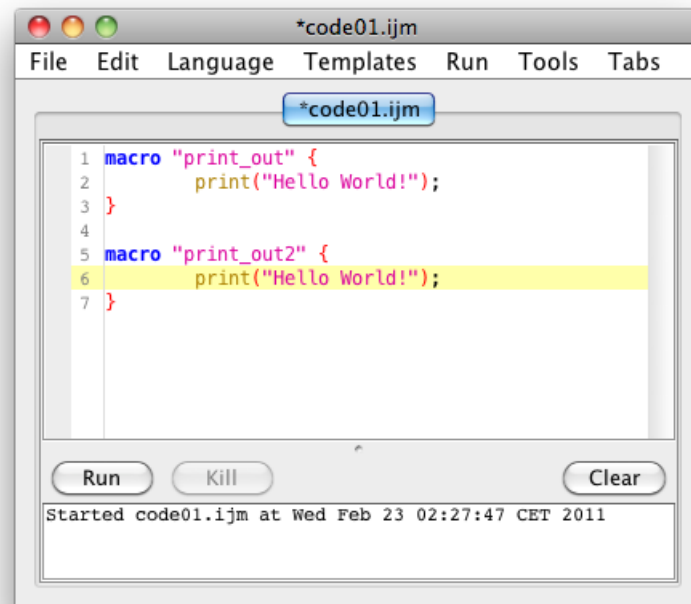In the main menu you should no be able to see the macro names under
`[Plugins > Macros > ]`.

Figure 2.5: Macro Set

2.2.2 Variables and Strings

Texts such as "Hello World!" can be represented by a variable [9]. Let's understand this by examining a short macro below.

```
1 //Code 2
2 macro "print_out 2" {
3   text = "Hello World";
4   print( text );
5   text = "Bye World";
6   print( text );
7 }
```

code/code02.ijm

`text` is a "String Variable" or simply a "String". ImageJ prepares a memory space for this variable, and you can change the content by re-defining the

---

[9]there is no declaration of types, such as number or string, in ImageJ macro.

11

content. Two (or maybe more) variables could be used to construct another variable.

```
1 //Code 3
2 macro "print_out 3" {
3   text1 = "Hello";
4   text2 = " World!";
5   text3 = text1 + text2;
6   print(text3);
7 }
```

<div align="center">code/code03.ijm</div>

The above operation concatenates content of `text2` to the content of `text1` and produces a third variable `text3` that holds the result of concatenation.

Exercise 2.2.2-1

> Add more string variables and make a longer sentence.

It is also possible to store a number in a variable. For example,

```
text = 256;
```

With this assignment, the variable is now a "numerical variable" or simply "variable". In other programming languages such as C or Java, difference between numbers and characters matters a lot. In ImageJ macro you do not have to care whether the variable is number or string, but in some cases this may cause problem so it's better to just keep this difference in your mind. We will see an example of such confusion, and also a way to avoid the confusion.

Test the following macro to see how the numerical variable works.

```
1 //Code 4
2 macro "print_out_calc" {
3   a = 1;
4   b = 2;
5   c = a + b;
6   print(c);
```

```
 7    print(a + "+"+ b + "="+c);
 8    txt=""+a + "+"+ b + "="+c;
 9    print(txt);
10  }
```

<p align="center">code/code04.ijm</p>

Did you get some results printed out? It should, but you should read the code carefully.

You might have noticed a strange expression at line 8, in the way it assigns the variable txt. It starts with double quotation marks.

```
txt= "" + a + "+" + b + "=" + c;
```

Seemingly this looks like meaningless. If you define txt without the first "useless" quotation marks, then it will be like

```
txt= a + "+"+ b + "=" + c;
```

Theoretically this should work also, since the double quote does not have any content so it's presence should be meaningless. But if you try to run this what it seems to be straight-forward assignment, ImageJ returns an error message.

This is because when ImageJ scans through the macro from top to bottom, line by line, it reaches the line for the assignment of the variable `txt` and first sees the variable `a` and interprets that `txt` should be a numerical variable (or function), since `a` is known to be a number as it was defined so in one of the lines above. Then ImageJ goes on interpreting rightward thinking that this is math. Then finding a "+" which surprisingly is a character ImageJ cannot interpret string variable within a numerical function, so it returns an error message. The macro aborts.

To overcome this problem, the programmer can tell ImageJ that *txt* is a string function at the beginning of the assignment by putting a set of dou-
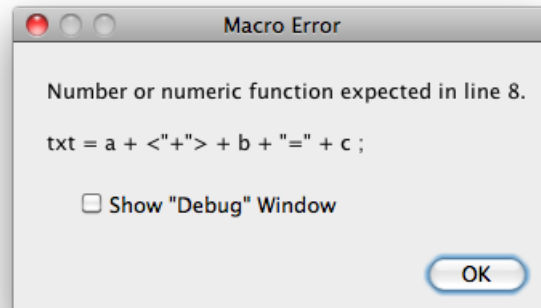
<p align="center">13</p>

Figure 2.6: Error with Variable Assignment

ble quote. This tells the interpreter that this assignment is a string concatenation assignment and not a numerical assignment. ImageJ does handle numerical values within string function, so the line is interpreted without problem and prints out the result successfully.

Exercise 2.2.2-2

>   Modify the code 4, so that the calculation involves subtraction (-), multiplication (*) and division (/).

2.2.3   Parameter Input by User

At some point you might want to make a macro to ask the user to input numerical values or file names. We now learn how to do that, by first examining the following code. Run the code first.

```
1  //Code 5
2  macro "input_print_out_calc" {
3    a = getNumber("a?", 10);
4    b = getNumber("b?", 5);
5    c = a*b;
6    print("\\Clear");
7    print(c);
8  }
```

code/code05.ijm
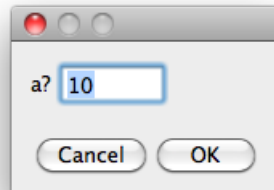
Running this macro, a dialogue window pops-up.

Figure 2.7: getNumber Dialog

The function *getNumber* consists of two parameters (programmers call such parameters "arguments" so we use this word in the reminder of this textbook).

getNumber(message string, default number)

The first argument is a string wrapped by double quotes (see code 5, line 3 and 4). This string will appear in the dialog window such as shown above. Default number will appear in the input field in the dialog window, and the user is expected to modifies this default number. When OK button is clicked, the number given by the user will be returned to the macro and then substituted to a variable. In the above case, this could be either *a* or *b*.

To ask a user for providing a string in dialog, following is an example.

```
1 //Code 6
2 macro "input_print_out_str" {
3   a = getString("a?", 10);
4   b = getString("b?", 5);
5   c = a+b;
6   print("\\Clear");
7   print(c);
8 }
```

code/code06.ijm

The function **getString** also has two arguments, and only the difference is that the user input will be treated as a string.

Exercise 2.2.3-1

> Run the code 6 and input 1 for a and 2 for b. What happened? Explain the reason.

2.2.4 Including ImageJ commands into a macro

There are many functions in ImageJ as you could see them by exploring the menu. Almost all of these commands can be accessed from macro[10].

We will make a macro that creates a new image, add noise, blurs this image by Gaussian blurring, and then thresholding the image. To know macro functions mirroring the menu commands there is a very convenient tool called **Command Recorder**. Do `[PlugIns -> Macros -> Record...]`. A window like figure **??** opens.
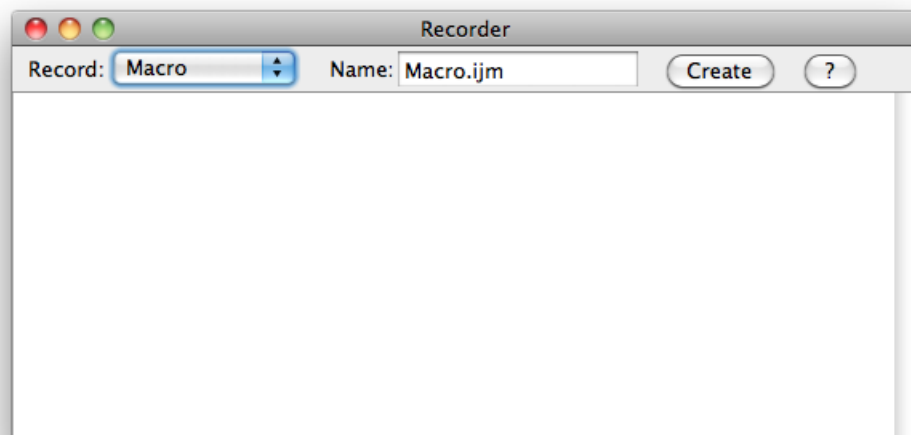

Figure 2.8: Macro Recorder

All the menu commands that you execute will be printed out as a history of macro functions in this window. For composing a macro using this recorder, we first do the processing manually from the menu as follows.

---

[10]Some plugins are not macro-ready.

- Prepare a new image using `[File -> New]` command. The size of the image can be anything.

- Then do `[Process -> Noise -> Salt and Pepper]` (Fig. **??**).

- `[Process -> Filters -> Gaussian Blur]` (use Sigma = 2.0).

- `[Image -> Adjust -> Threshold...]`. Toggle the slider to make signals red. Check "Dark Background", then click "Apply".
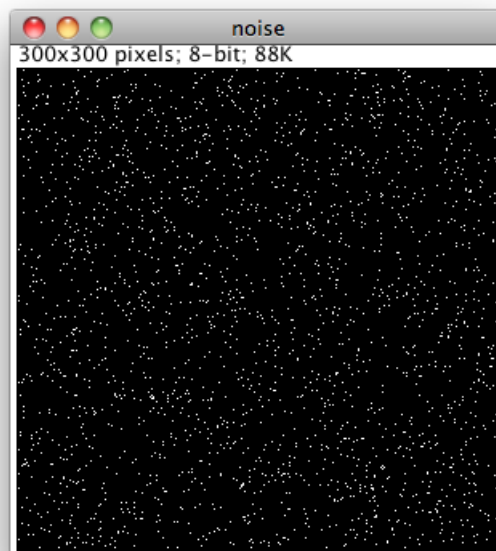

Figure 2.9: A demo image for Recording Macro

Now, check the Command Recorder window. It should now look like Fig. **??**. Each line is a macro function that corresponds to a menu command you selected.

These texts generated in in the recorder can be used as it is in your macro. You could copy and paste them[11]. Compose a macro like below by copying

---

[11]In case of OSX, you might probably need to click "Create" button to generate a duplicate of macro functions in a new script window. Then you could copy the macro functions from there.
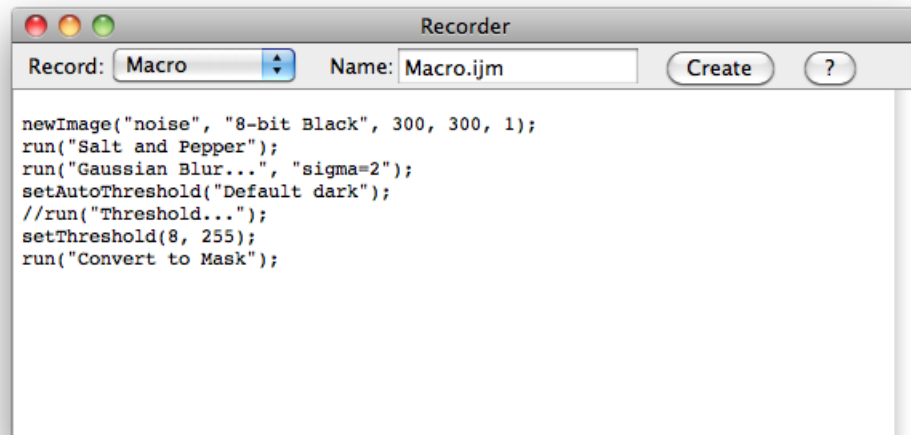
Figure 2.10: Macro Recorder after some lines Recorded

and pasting the macro functions in the recorder. Delete the lines that are commented out (lines that begin with "//" are lines that are skipped by the macro interpreter).

```
1  //Code 6.9
2    newImage("test", "8-bit Black", 300, 300, 1);
3    run("Salt and Pepper");
4    run("Gaussian Blur...", "radius=2");
5    setThreshold(32, 100);
6    run("Convert to Mask");
```

code/code06_9.ijm

Run the macro! ...I hope that you are amazed by now with the power of Macro Recorder! Now, you could simple add a line at the top and bottom to package this in a named macro. This is optional in the current case, but it's always good to keep your macro like this since boundary of the macro becomes clear.

```
1  //Code 7
2  macro "GB2_Thr" {
3    newImage("test", "8-bit Black", 300, 300, 1);
4    run("Salt and Pepper");
5    run("Gaussian Blur...", "radius=2");
```

```
6   setThreshold(32, 100);
7   run("Convert to Mask");
8 }
```

code/code07.ijm

The second line in the above macro has a function `newImage()`. This function creates a new image. It has five arguments (in coding jargon, we say there are "five arguments"). To know what these arguments are, the quickest way is to read the Build-In Macro Function page in ImageJ web site[12]. In case of the function `newImage`, the description looks like this.

---

newImage(title, type, width, height, depth)
Opens a new image or stack using the name title. The string type should contain "8-bit", "16-bit", "32-bit" or "RGB". In addition, it can contain "white", "black" or "ramp" (the default is "white"). As an example, use "16-bit ramp" to create a 16-bit image containing a grayscale ramp. Width and height specify the width and height of the image in pixels. Depth specifies the number of stack slices.

Using this information, you can modify the macro to change the size of the image.

Exercise 2.2.4-1

> Modify the code 8, so that user can input the desired Gaussian sigma.

Other optional lines you could add to the macro are "comments". This does not affect the macro but adding some comment about what the macro does helps you to understand what the macro is doing when you open the file some time later. There are two ways to add comment. One is the **block comment**. Texts bounded by `/*` and `*/` will be ignored by interpreter. Another is the line comment. Texts in a line starting with double slash `//`

---

[12]http://rsbweb.nih.gov/ij/developer/macro/functions.html

will be ignored by the interpreter. Below is an example of commenting code 07.

```
 1 //Code 7.1
 2 /*
 3 This macro creates binary image with randomly positioned
       dots.
 4 */
 5 macro "GB2_Thr" {
 6   //creates a new image window
 7   newImage("test", "8-bit Black", 300, 300, 1);
 8   //add noise
 9   run("Salt and Pepper");
10   //blur the image
11   run("Gaussian Blur...", "radius=2");
12   //binarize the image
13   setThreshold(32, 100);
14   run("Convert to Mask");
15 }
```

code/code07_1.ijm

2.2.5   Batch Processing using "batch macro" function

In above macro, list of functions were wrapped inside macro "title"{ code } so that these macro functions could be executed by single command from menu. To apply such a sequence of macro functions for many images in a single folder (say you have one-thousand images you want to contrast enhance and also to Gaussian-blur), there are two ways. One way is to further extend the macro by adding file-accessing macro functions and looping those functions (you will learn this later). Another way is to do such "batch processing" by copy and pasting list of macro functions to batch-processing interface. This interface could be used by `[Process -> Batch -> Macro]`

In "Input" field, select the folder where image files are stored. In output field, select a destination folder where processed images will be stored. You then copy and paste the list of macro functions in the code field such as
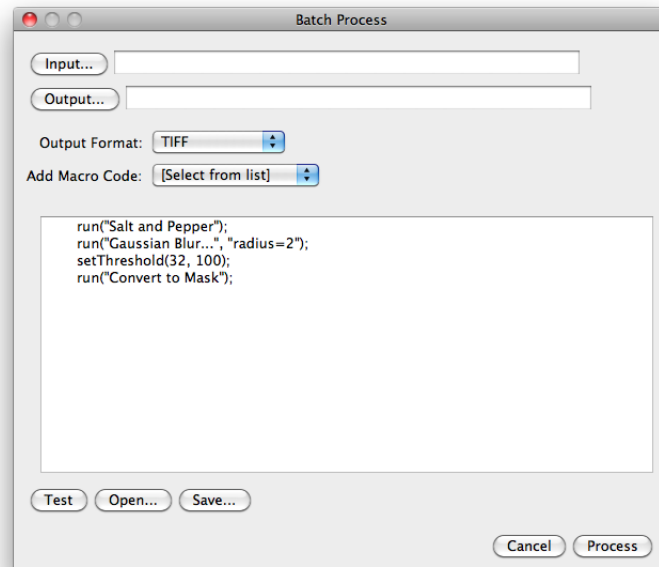
Figure 2.11: Batch Processing Dialog

shown in Fig. **??**. In the case shown in this figure, line 6 to 9 was copied and pasted. Clicking "Process" button will start the processing.

2.3 Conditions and Loops

In many cases, we want to iterate certain processing many times ("Loops": see middle in the figure **??**), or we want to limit some of the process in the program only for certain situations ("Conditions": see right of the figure **??**). In this section we learn how to include these loops and conditional behaviors into macro.
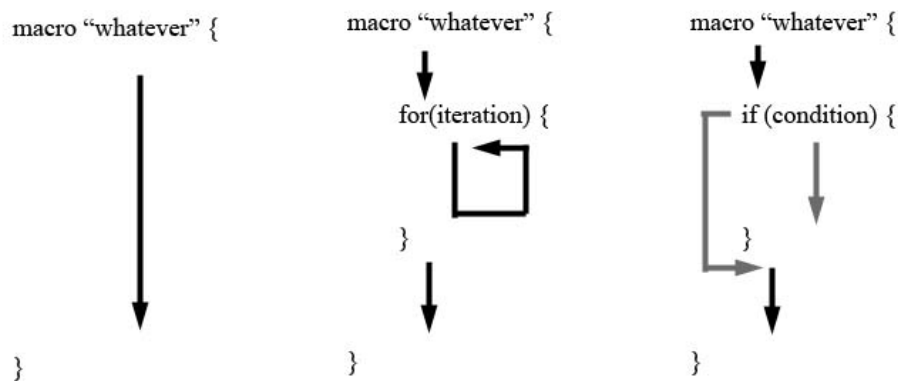
Figure 2.12: Schematic view of conditions and loops. Straightly top to bottom, line by line processing (left) and macro with loops (middle) and with a condition (right).

2.3.1 Loop: for-looping

Here is a simple example macro using for-loop. Write the macro in your editor and run it.

```
1 //Code 9
2 macro "loop1" {
3   message_txt = getString("message to loop?", "whatever");
4   for(i=0; i<5; i+=1) {
5     print(i + ": " + message_txt);
6   }
7 }
```

code/code09.ijm

The result should look like:

Figure 2.13: Code 9 output in Log Window

- Line 3 asks the user to input a string (we did this already). If user does not change the default text ("whatever") and click "OK", then the macro interpreter proceeds to line 4.
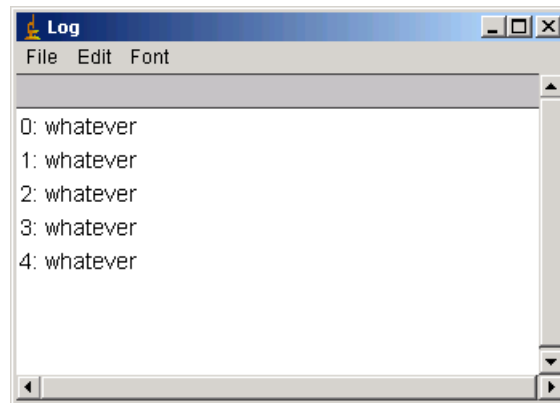
- Line 4 `for( i = 0 ; i < 5 ; i+= 1)` sets the number of loops. Three parameters are required for "for" loop. The first parameter defines the variable used for the counting loop and its initial value (`i = 0`). The second parameter sets the condition for exiting from the loop (`i < 5`). Third parameter sets the step size of i, meaning that how much value is added per loop (`i += 1`, could also be subtraction, multiplication, division e.g. `i -= 1`).

- After this `for(...;...;...)` statement, there is a brace (`{`) at the end of line 4 and the second one in the line 6. These curly braces tell ImageJ to loop macro functions in between so the functions in line 5 will be iterated according to the parameters defined in the parenthesis of `for`. Between braces, you could add more lines of macro functions as many as you want.

So when the macro interpreter reaches line 4 and sees `for(`, it starts looking inside the parenthesis and defines that the counting starts with 0 using a variable `i`, and then line 5 is executed. The macro prints out "0 :whatever" using the content of `i`, string `:` and the string variable `txt`. Then in line 6 interpreter sees the boundary `}` and goes back to line 4 and adds 1 to i (because of `i+=1`). i = 1 then, so `i<5` is true. The interpreter proceeds to line 5

23

and executes the macro function and prints out "1:whatever". Such looping will continue until i = 5, since only by then `i<5` is no longer true so interpreter exits from the for-loop.

Exercise 2.3.1-1

> (1) Change the first parameter in `for(i=0;i<5;i+=1)` so that the macro prints out only 1 line.

> (2) Change the second parameter in `for(i=0;i<5;i+=1)` so that the macro prints out 10 lines.

> (3) Change the third parameter in `for(i=0;i<5;i+=1)` so that the macro prints out 10 lines.

Stack Analysis by for-looping

One of frequently encountered task is image stack management, such as measuring dynamics or multi-frame processing. Many ImageJ functions work with only single frame within a stack. Without macro programming, you need to execute the command while you flip the frame manually. Macro programming enables you to automate this process. Here is an example of measuring intensity change over time[13].

```
1  //Code 10
2  macro "Measure Ave Intensity Stack" {
3    frames=nSlices;
4    run("Set Measurements...", "  mean  min integrated
        redirect=None decimal=4");
5    run("Clear Results");
6    for(i=0; i<frames; i++) {
7      currentslice=i+1;
8      setSlice(currentslice);
9      run("Measure");
10   }
11 }
```

code/code10.ijm

---

[13]What we write as macro here could be done with a single command `[Image > Stacks > Plot Z-Profile]` but this only measures intensity. If you want to measure other values such as the minimum intensity, a macro should be written.

- Line 3: `nSlices` is a macro function that returns the number of slices in the active stack.

- Line 4: Sets measurement parameters, from the menu would be `[Analyze > Set measurements...]`. In this case "mean min integrated" is added as part of the second argument. "mean" is the mean intensity, "min" is the minimum intensity and "integrated" is integrated density (total intensity). These keys for measured parameters could be known by using the command recorder. You do not have to care for now about the "redirect" argument. "decimal" This is the number of digits to the right of the decimal point in real numbers displayed in the results table.

- Line 5: clears the results table.

- Line 6 to 9 is the loop. Loop starts from count i=0, and ends at i=frame-1. Increment is 1.

- Line 7: calculates the current frame number.

- Line 8: `setSlice` function sets the frame according to the frame number calculated in line 6.

- Line 9: actual measurement is done. Result will be recorded in the memory and will be displayed in the Results table window.

Open an example stack **1703-2(3s-20s).stk** [14]. This is a short sequence of FRAP analysis, so the edge of the one of the cells is bleached and then fluorescence signal at that bleached position recovers by time. Select the frapped region by ROI tool (such as in the figure below). Execute the macro. Results will be printed in the Results window (see the table in the figure left).

Measurement parameters can be added as argument by modifying the line 4 in the code 10. "Set Measurement" could be added with more parameters

---

[14]Some of you may realize that you used this sequence in the Image Processing / Analysis Course for learning stack measurements using Z-profiler `[Image > Stacks > Plot Z-Profile]`. Now, you can program similar device in macro. Good thing about a custom macro is that you will be able to modify the program further to add more functions. For example, You could measure the time course of standard deviation of intensity within the selected ROI.

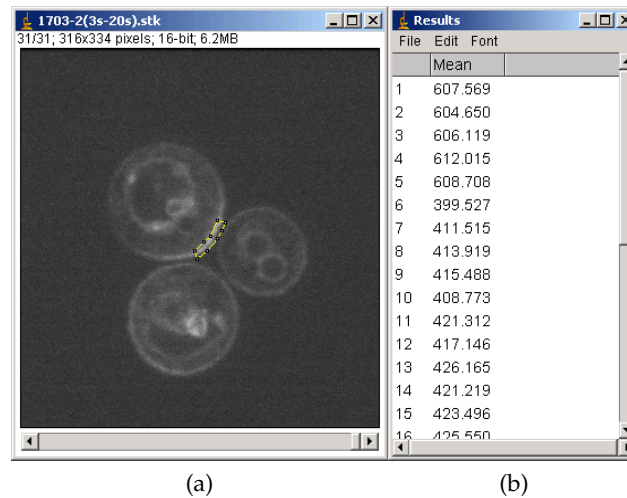(a)   (b)

Figure 2.14: Measuring Stack Intensity Series. (a) Setting a Segmented ROI at the FRAPped area. (b) Results of Measuring Mean Intensity Dynamics.

to be measured, and the digits after the decimal point could be increased by increasing the number after "decimal=". For example,

```
run("Set Measurements...", "area mean standard modal min
    centroid center perimeter bounding integrated median
    stack redirect=None decimal=5");
```

Exercise 2.3.1-1

> Modify code 10 to include more measurement parameters (whatever you like), and test the macro. Check the results.

2.3.2   Loop: while-looping

Another way of letting a part of macro to loop is **while**-statement. In this case, iteration is not defined strictly. Looping continues until certain condition is met. As soon as the condition is full-filled, macro interpreter goes out from the loop.

Figure 2.15: An example result after adding more measurement parameters.

Basics of while statement

Here is a simple example macro using `while`.

```
1 //Code 11
2 macro "while looping1" {
3   counter=0;
4   while (counter<=90) {
5     print(counter);
6     counter = counter + 10;
7   }
8 }
```

code/code11.ijm

This macro prints out characters 0 to 90 with a 10 increment.

- line 3: The macro interpreter first assigns 0 to the counter.

- line 4: The interpreter evaluates if the counter value is less than or equal to 90. Since counter is initially 0...

- line 5 Printing function is executed.

- line 6: counter is added with 10.

- line 7: the interpreter realizes the end of "while" boundary and goes back to line 4. Since counter= 10 <= 90, line 5 is again executed...and so on. When counter becomes 100 in line 6 after several more loops,
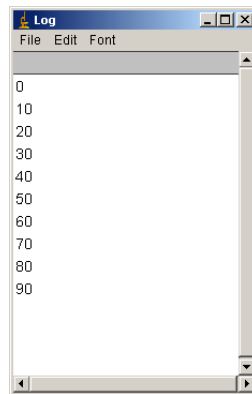
Figure 2.16: Output of code 11

counter is no longer <=90. So the interpreter goes out from the loop, moves to line 8. Then the macro is terminated.

Line 5 could be written in the following way as well.

```
counter += 10;
```

This means that "counter" is added with 10. Similarly, subtracting 10 from counter is

```
counter -= 10;
```

Multiplication is

```
counter *= 10;
```

Division is

```
counter /= 10;
```

If the increment is 1 or -1, (counter +=1 or counter-=1), then one could also write them as

```
counter++;
 or
counter--;
```

These two last macro functions are said to work faster than +=1 or -=1, but I myself do not see much difference. Computers are fast enough these days.

Exercise 2.3.2-1

(1) Try changing code 11 so that it uses "+=" sign.
(2) Change code 11 so that it uses "++" sign, and prints out integers from 0 to 9.

Evaluation of `while` condition could also be at the end of loop. In this case, `do` should be stated at the beginning of the loop. With do-while combination, the loop is always executed at least once, regardless of the condition defined by `while` since macro interpreter reads lines from top to bottom. Try with the following exercise.

Exercise 2.3.2-2

Change line 4 of code 11 to `while (counter <0)` and check the effect (see below).

```
1 //Code 11.5
2 macro "while looping2" {
3   counter=0;
4   do {
5     print(counter);
6     counter += 10;
7   } while (counter<0);
8 }
```

code/code11_5.ijm

Condition for the while-statement could be various. Here is a small list of comparison operators.

<	less than
<=	less than or equal
>	greater than
>=	greater than or equal to
==	equal
!=	not equal

Exercise 2.3.2-3

> Modify code 11 so that the macro prints out numbers from 200 to 100, with an increment of -10.

Why is there while-loop?

An often raised question with the while-loop is why do we have two types of loops, the for-loop and the while-loop. Answering to this question, they have different flexibility. The for-loop is rather solid and the while-loop is more flexible. In the example code below, the user is asked for a correct number and if the answer is wrong, the question is asked 5 times repeatedly. Number of loop is not determined by the programmer, but interactively when the code is running. We will study the branching of the program based on if-else in the next section.

```
1  macro "flexible loop by while" {
2    answer_is_wrong = true;
3    imagej_first_release = 1997;
4    trial = 5;
5    while (answer_is_wrong) {
6      answer = getNumber("In which year did the first version
           of ImageJ released?", 1900);
7      if (answer == imagej_first_release )}
8        answer_is_wrong = false;
9            showMessage("CORRECT! The year" +
                 imagej_first_release);
10         } else {
11            showMessage("NO. try again: trials left:" +
                 trial);
12            trial--;
13         }
```

```
14          if (trial < 1)
15              answer_is_wrong = false;
16    }
17 }
```

code/code11_6.ijm

Writing a similar code using the for-loop is possible but the code becomes tricky. Below is the for-loop version of the above code.

```
1 macro "flexible loop by for" {
2   imagej_first_release = 1997;
3   trial = 10;
4   for (correct = 0; correct < 1; ) {
5     answer = getNumber("In which year did the first version
           of ImageJ released?", 1900);
6     if (answer == imagej_first_release ){
7       showMessage("CORRECT! The year" +
             imagej_first_release);
8       correct++;
9     } else {
10      showMessage("NO. try again: trials left:" + trial);
11      trial--;
12    }
13    if (trial < 1)
14            correct++;
15  }
16 }
```

code/code11_7.ijm

Note that the third argument of for-loop is missing. Since the variable `correct` does not change as long as the answer is wrong, we leave it not incrementing nor decrementing. In such case we can leave the third argument vacant.

2.3.3 Conditions: if-else statements

Introducing if-else

A macro program could have parts which are executed depending on some conditions. Here is an example of macro with conditions.

```
1 //Code 12
2 macro "Condition_if_else 1"{
3   input_num = getNumber("Input a number", 5);
4   if (input_num == 5) {
5     print(input_num+ ": The number is 5 ");
6   }
7 }
```

<div align="center">code/code12.ijm</div>

- Line 3 The macro asks user to input a number and the number is substituted to the variable input_num.

- Line 4 Content of input_num is evaluated. If input_num is equal to 5, line 5 is executed and prints out the message in the Log window. Otherwise macro interpreter jumps to line 7, and ends the operation. By adding "else" which will be executed if input_num is not 5, the macro prints out message in all cases (see code 12.5 for this if - else case).

- Line 4 We used double equal signs for evaluating the value in the right side and the left side (e.g. `if (a==5)`). Note that the role of the sign = is different from assignments, or substitution (e.g. `a = b + c`).

Now, we examine the content between parenthesis after "if" in more detail. Write the following code in your script editor and run it.

```
1 a = (5==5);
2 print(a);
```

<div align="center">code/code12_1.ijm</div>

The output in the log window should be *1* indicating that "`(5 == 5)`" is *1*. Next, modify the code like below and run it.
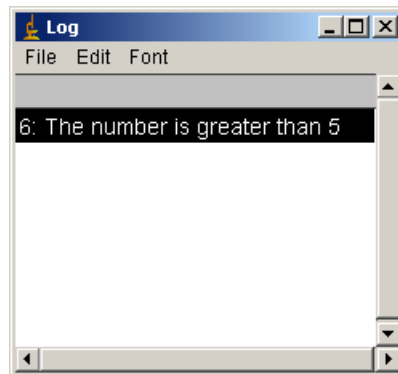
Figure 2.17: Output of code 12

```
1 a = (5 == 4);
2 print(a);
```

code/code12_2.ijm

The output is now 0, indicating that "(5 == 4)" is 0. What double equal signs == are doing in these examples are comparison of numbers in the left and the right side, and if the numbers are the same, it returns 1 and if they are not the same, it returns 0. 1 and 0 actually are representing **true** (= 1) or **false** (= 0), the **boolean values**.

We could also test if they are NOT equal. For this, replace == by !=.

```
1 a = (5 != 4);
2 print(a);
```

code/code12_3.ijm

Run the code above, and it returns 1, because 5 is NOT 4 and that is true. Now, you could introduce the *if* again as follows.

```
1 if (5 != 4){
2   print("true");
3 }
```

code/code12_35.ijm

In the parenthesis after "if", there is obvious TRUE statement (5 is not 4). This is true, so the macro function bounded by curly braces is executed, which is to print out "true!" in the log window.

33

Try changing the line two to `if (5 == 4)`. Running this prints nothing in the log window, because 5 is not 4 (FALSE!) so that the macro function in line 3 is ignored. To avoid such ignorant no-output behavior, you could add "else" as follows.

```
1 if (5 == 4){
2   print("true");
3 } else {
4   print("false!");
5 }
```

<div align="center">code/code12_4.ijm</div>

The code works also with the direct true or false declaration inside the if parenthesis. Try the following code.

```
1 if (0){
2   print("true");
3 } else {
4   print("false!");
5 }
6
7 if (false){
8   print("true");
9 } else {
10  print("false!");
11 }
```

<div align="center">code/code12_5.ijm</div>

The above prints two lines of "false!" in the log window. You could replace the if parenthesis values to 1 and true to check that it works as well.

By now, it is probably pretty clear to you wi what is going on in the code below.

```
1 macro "Condition_if_else 2"{
2   input_num = getNumber("Input a number", 5);
3   if (input_num == 5) {
4     print(input_num+ ": The number is 5 ");
5   } else {
6     print(input_num+ ": The number is not 5 ");
7   }
```

```
8   print("-------------");
9 }
```

code/code12_6.ijm

Complex Conditions

In many cases, you might need to evaluate the condition of multiple variables at once. For such demands, several different comparisons can be combined by using following Boolean operators.

&&   boolean AND
|| boolean OR

Let's first test what they do with very a simple example directly using `true` and `false`.

```
1 a = true;
2 b = true;
3 if (a && b){
4   print("&& both true")
5 }
6
7 if (a || b){
8   print("|| one of them or both is true")
9 }
```

code/code12_65.ijm

When you run this code as it is, line 4 and line 8 are both executed and prints the messages. For the first `if` parenthesis, `&&` operator tests if both sides are true. If both are indeed true, it returns true (1), and that is the case above. If one of them or both are false, then `&&` operator returns false(0).

On the other hand, in the second if parenthesis, `||` operator tests if one of the two sides is true. Since both are true in the above code, OR operator returns true because at least one of them is true. Only when both sides are false, the returned value becomes false (0).

Exercise 2.3.3-1

Change the values of `a` and `b` in code 12_65 to `false` and compose other three possible combinations (e.g. a = true, b = false will print only one line). Check the output. Change the values of `a` and `b` also to 0 and/or 1 and check the results.

Here is a bit more realistic example (though very useless), an extended version of code 16_6.

```
//Code 12.75----------------------------

macro "Condition_if_else 3"{
  input_num1 = getNumber("Input a number 1", 5);
  input_num2 = getNumber("Input a number 2", 6);
  message0 = ""+input_num1 + ","+input_num2;  //use this
      string four times
  if ( (input_num1==5) && (input_num2==6) ) {
    print(message0+ ": The parameter1 is 5 and the
        parameter2 is 6");
  } else {
    if (input_num1!=5) && (input_num2!=6) {
        print(message0 + ": The parameter1 is not 5 and the
            parameter2 is not 6");
    } else {
      if (input_num2==6) {
        print(message0 + ": The parameter1 is NOT 5 but the
            parameter2 is 6");
      } else {
        print(message0 + ": The parameter1 is 5 but the
            parameter2 is NOT 6");
      }
    }
}
```

code/code12_75.ijm

- Line 4 and 5 Ask user to input two parameters.

- Line 6 is for setting a string variable, to abbreviate a long string assignment that appears four times in the macro.

- Line 7 evaluates these input parameters by comparing each of them separately, but the decision is made by associating two decisions by "&&".

- Text after "//" is called comment. Text after this double slash will not be evaluated by the macro interpreter. Comments helps programmers later for remembering (or letting other programmer to understand) the purpose of the line.

- Line 10, != compares left and right sides of the operators and returns true if they are NOT equal.

From line 10 to 17, there are several layers of conditions. Macro programmer should use tab-shifting for deeper condition layers as above for the visibility of code. Easy-to-understand code helps the programmer oneself to debug afterward, and also for other programmers who might reuse the code.

Application of if-statement

As an application of looping and conditions, we write a macro that produces an animation of moving dot. User inputs the speed of the dot, and then the animation is generated. In the animation (which actually is a stack) the dot moves horizontally and bounces back from the edge of the frame. (if) operator is used to switch the movement direction.

```
1  //Code 13
2  macro "Generate Dot Animation back and forth" {
3
4  // **** initial values ****
5    sizenum=10; //dot size in pixel
6    int=255;  //dot intensity in 8bit grayscale
7    frames=50;  //frames in stack
8    w=200;    //width of frame
9    h=50;   //height of frame
10   x_position = sizenum; //starting x position:
11   y_position= (h/2)-(sizenum/2);  //y positon of the oval
       top-left corner: constant
12
13  //**** set colors *****
14   setForegroundColor(int, int, int);
15   setBackgroundColor(0, 0, 0);
16
```

```
17  //**** ask speed *****
18    speed=getNumber("Speed [pix/frame]?",10)
19
20  //**** prepare stack ****
21    stackname="dotanimation"+speed;
22    newImage(stackname, "8-bit Black", w, h, frames);
23
24  //**** drawing oval in the stack ****
25    for(i=0; i<frames; i++) {
26      setSlice(i+1);
27      x_position += speed;
28      if ((x_position > (w-sizenum)) || (x_position < 0) ) {
29        speed*=-1;
30        x_position += speed*2;  //avoids penetrating boundary
31      }
32      makeOval(x_position, y_position, sizenum, sizenum);
33      run("Fill", "slice");
34    }
35    run("Select None");
36  }
```

code/code13.ijm

- Lines 4 to 11: Set parameters for drawing a dot. It is also possible to directly use numerical values in the later lines, but for the sake of readability of the code, and also for possible later extension of the code, it is always better to use easy-to-understand variable names and explicitly define them before the main part starts.

A short note on the x-y coordinate system in digital images: Since digital image is a matrix of numbers, each pixel position is represented as coordinates. The top left corner of image is the position (x, y) = (0, 0). X increases horizontally towards right side of the image. Y increases vertically towards the bottom of the image. In line 9, y-position of the dot is defined to be placed in the middle of the vertical axis.

- Lines 14, 15: These lines set the drawing and background color. Three arguments are for each RGB components. Here the image is in grayscale

so all the RGB components are set to the same value. 0 is black, and int = 255 = white.

- Line 14 asks the user to input the speed of the dot movement.

- Lines 16, 17 prepares a new stack with parameters defined in lines 7, 8 and 9.

- Lines 21 to 34 is the loop for drawing moving dot. Loop will be iterated from the starting frame until the last frame. Line 21 creates an oval ROI, which will be filled in line 22 with the foreground color that was already set in the line 14. `makeOval` function is explained in the Built-on function page as follows.

    makeOval(x, y, width, height)
    Creates an elliptical selection, where (x,y) define the upper left corner of the bounding rectangle of the ellipse.

- Line 27: Shifts the x position of the dot by "speed" distance.

- Line 28: if the position calculated in the line 27 exceeds the boundary, either left (`x_position < 0`) OR right (`x_position > (w-sizenum)`), then the direction of movement is switched by multiplying -1.

Exercise 2.3.3-2

Modify code 13 that the dot moves up and down vertically. Change the stack width and height as well.

If you are successful with this, try further on to extend the code so that the dot moves both in x and y directions. For this, you need to have two independent speed `xspeed` and `yspeed` since change in the direction by bouncing should be independent in x and y.

Application of "while" and "if" in image processing.

Now, we try solving a problem with image thresholding by an application of `while` loop in a macro. Open image **mt_darkening.tif** in the sample image you downloaded. This is a stack, so you could slide the bar at the bottom of the window to see what is happening: the image gets darker and

darker, as frame number increases. When you study fluorescence images, you will find such effect very often, because fluorescence bleaches due to the irradiated excitation light for the acquisition. When you want to segment this structure (a microtubule), you might use image-thresholding as follows.
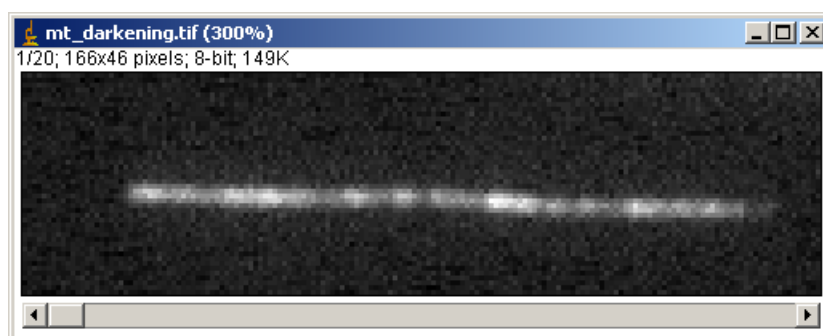
Figure 2.18: A stack with darkening microtubule

Go back to the first frame and do `[Image -> Adjust -> Thresholding...]`. The image is then automatically adjusted with threshold level. and it seems Ok that the structure is well segmented. But the problem appears as you slid the bar at the bottom. Since image is darkening, area where highlighted decreases.

(a)   (b)

Figure 2.19: Adjusted with threshold level first frame (a) and the last frame (b)

This is because the threshold minimum and the maximum is kept constant while the intensity of the image is decreasing. To segment the structure while the image darkening is occurring, we must adjust the threshold intensity range as the frame progresses.

The macro below finds the minimum value for the thresholding, that the highlighted area in each frame in a stack is approximately similar to the

first frame. `while` is used to loop the adjustment until the highlighted area is constant. Then the threshold is applied to the image to convert the stack to a binary stack.

```
1  //Code 14
2  macro "Automatic Threshold Adjustment" {
3    if (nSlices==1) {
4      exit("Active window is not a stack");
5    }
6    getThreshold(lower, upper);
7    if ((lower==-1) && (upper==-1)) {
8      exit("Image must be thresholded");
9    }
10   w=getWidth();
11   h=getHeight();
12   frames=nSlices;
13   ref_slice=getSliceNumber(); //reference frame
14   originalStackID=getImageID();
15
16   run("Clear Results");
17   run("Set Measurements...", "area limit redirect=None
         decimal=0");
18   run("Measure");
19   ref_area=getResult("Area", 0);  //reference area
20   temp_area=0;
21   tol=0.03; // tolerence for the area difference +-3%
22
23   newImage("bin stack", "8-bit White", w, h, frames);
24   binStackID=getImageID();
25   for(i=0;i<frames;i++) {   //flipping frames
26     selectImage(originalStackID);
27     setSlice(i+1);
28     run("Copy");
29     newImage("stack", "8-bit White", w, h, 1);
30     run("Paste");
31     while ((ref_area*(1-tol)>temp_area) || (temp_area>
          ref_area*(1+tol))){
32       setThreshold(lower,upper);
33       run("Clear Results");
34       run("Measure");
35       temp_area=getResult("Area", 0);
```

41

```
36      lower--;
37    }
38    run("Convert to Mask");
39    run("Copy");
40    close();
41    selectImage(binStackID);
42    setSlice(i+1);
43    run("Paste");
44  }
45 }
```

code/code14.ijm

(a)   (b)

Figure 2.20: Binarized last frame without threshold adjustment (a) and with adjustment using macro (b).

- Lines 3 to 5 Check if the active window is a stack. If `nSlices==1` (meaning that the image is not a stack), macro is terminated.

- Lines 6 to 9: Get the threshold parameter from image and check if the image is adjusted with threshold level. If not, both upper and lower values are -1. In this case, macro is terminated.

- Lines 10 to 14: Get stack information. `getSliceNumber()` returns the current frame in the stack. Adjusted with threshold level area in this frame (first frame) will be used as the reference area. `getImageID()` returns a number that specifically identifies the active window. This ImageID will be used later,by `selectImageID(ImageID)` to re-activate the window.

  getImageID()
  Returns the unique ID (a negative number) of the active image. Use the selectImage(id), isOpen(id) and isActive(id) functions to activate an image or to determine if it is open or active.

42

- Line 16: clears the results table without saving.

- Line 17: sets the measurement parameter Area, and limits the measurement to the adjusted with threshold level region.

- Line 18: Do the measurements. Result is recorded in the first row of the Results table.

- Line 19: The measured area is stored in the variable ref_area.

- Line 20: temp_area will be used later in the while loop.

- Line 21: the variable ilcomtol is a tolerance ratio of error against the reference area. So the adjusted with threshold level area in each frame should be between 97 and 103% of the reference area.

- Line 22: Create a destination stack, where adjusted with threshold level images will be pasted.

- Line 23: get the Image ID of newly created image.

- Line 25: Loop for the frames starts.

- Lines 26, 7: Select the original stack and sets the frame number according to the loop number. `selectImageID` works with `getImageID` function in line 14.

  selectImage(id)
  Activates the image with the specified ID (a negative number). If id is greater than zero, activates the idth image listed in the Window menu. With ImageJ 1.33n and later, id can be an image title (a string).

- Line 28: Copy the full frame.

- Lines 29, 30: creates a temporally single frame image and the image copied in line 28 is pasted.

- Lines 31 to 37: While loop. temp_area is evaluated if the area is outside 97 and 103% of the reference area. If true, then loop continues.

Initial temp_area value is 0 so the loop is at least one time. Set Threshold with lower and upper (line 32). Measure the adjusted with threshold level area, and then lower is incremented -1. The area is evaluated, and if it does not meet the criteria set in line 31, then the loop continues with wider threshold range.

- Lines 38 to 40: The adjusted with threshold level image will be converted to black & white image and then copied. The single frame temporary image is closed.

- Line 41, 42: destination stack is activated and the same frame as the source stack is set.

- Line 43: Binarized image in the clipboard is pasted into the destination stack.

- Line 44: returns to Line 25 until all stack frames are processed.

- Line 45: Terminates the macro.

2.4 Advanced Macro Programming

This section could be a bit boring for you in terms of biology, but try to be patient. All these knowledge are required for advanced programming. Ability to do complex image processing using macro widens your view on planning experiments also.

2.4.1 User-defined Functions

As your code becomes longer, you will start to realize that similar processing or calculation appears several times in a macro or through macro sets. To simplify such redundancy, one could write a separate `function` that works as a module for macros. For example, if you have a simple code like:

```
1 //Code 15
2 macro "addition" {
3   a = 1;
4   b = 2;
5   c = a + b;
6   print(c);
7 }
```

code/code15.ijm

It should be easy for you to expect that this macro will print out "3" in the Log window. From this macro, we could extract part of it and make a separate function.

```
1 //Code 15.1
2 function ReturnAdd(n, m) {
3   p = n + m;
4   return p;
5 }
```

code/code15_1.ijm

This is not a macro, but is a program that works as a unit. Functions can be embedded in macro. `ReturnAdd` (code 15.1) is the name of the function, and the following (n, m) are the variables that will be used in the function. Within the function, n and m will be added and the result of which is substituted in to a new variable p. `return p` in line 4 will return a value as an

output of the function. In a sense, this is a custom made Macro function. Using this function, code 15 can be rewritten as

```
1  //Code 15.2
2  macro "addition with function1" {
3    a = 1;
4    b = 2;
5    c = ReturnAdd(a, b);
6    print(c);
7  }
8  //Code 15.1
9  function ReturnAdd(n, m) {
10   p = n + m;
11   return p;
12 }
```

code/code15_2.ijm

or more simply, by nesting the custom made function inside ImageJ native function `print()`,

```
1  //Code 15.3
2  macro "addition with function2" {
3    a = 1;
4    b = 2;
5    print(ReturnAdd(a, b));
6  }
7  //Code 15.1
8  function ReturnAdd(n, m) {
9    p = n + m;
10   return p;
11 }
```

code/code15_3.ijm

Macro interpreter reads the macro line by line. When the interpreter sees `ReturnAdd(a, b)`, the interpreter first tries to find the function within the ImageJ Build-in function. If its not there, the interpreter looks for the function within the same macro file... (user-defined function (e.g. `ReturnAdd(a, b)` must be written in the same macro file. Here is how it looks like: a macro that uses a function.
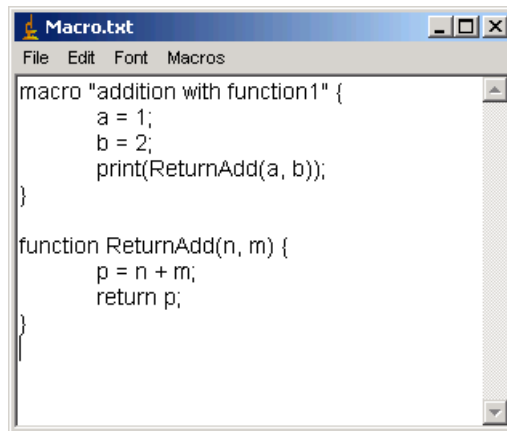
Figure 2.21: A macro file with function

In this simple case, you might not feel the convenience of the User-defined function, but you will start to feel its power as you start writing longer codes. Advantages of using `function` are

1. Once written in a macro file, it could be used as a single line function as many times as you want in the macro file. This also means that if there is a bug, fixing the function solves the problem in all places where the function is used.

2. Long codes could be simplified to an explicit outline of events. Such as:

```
macro "whatever" {
      function1;
    function2;
    function3;
}
```

Let's go back to the code 14, the automatic threshold adjusting macro.

At the beginning of the code, we check if the active image if it is a stack. There is another check after that, to see if the image is adjusted with threshold level.

```
1 //Code 14
```

```
2  macro "Automatic Threshold Adjustment" {
3    if (nSlices==1) {
4      exit("Active window is not a stack");
5    }
6    getThreshold(lower, upper);
7    if ((lower==-1) && (upper==-1)) {
8      exit("Image must be thresholded");
9    }
10   w=getWidth();
```

code/code14.ijm

We can make a function for checking stack (line 3 to 5) and another function that checks if the stack is adjusted with threshold level (from line 6 to 9) as below.

```
1  //Code 16
2  function CheckStack() {
3    if (nSlices==1) {
4      exit("Active window is not a stack");
5    }
6  }
7
8  //Code 17
9  function CheckThreshold() {
10   getThreshold(lower, upper);
11   if ((lower==-1) && (upper==-1)) {
12     exit("Image must be thresholded");
13   }
14 }
```

code/code16_17functions.ijm

Then the initial part of code 14 (line 3 to 9) can now be replaced with these two functions[15].

```
1  //Code 14.1
2  macro "Automatic Threshold Adjustment with function" {
3    CheckStack();
4    CheckThreshold();
```

---

[15]For a complete coding of 14.1, getThreshold(lower, upper) should appear again in line 8 to get lower and upper threshold value of the reference image.

```
5   getThreshold(lower, upper);
```

<div align="center">code/code14_1.ijm</div>

Exercise 2.4.1-1

The following macro asks the user to input x and y coordinates of two points, calculate the distance between those points and prints out the distance. Modify the code so that the distance calculation is done in a separate function.

```
1  //Code 18
2  macro "Calculate Distance" {
3    p1x = getNumber("point 1 x coordinate", 0);
4    p1y = getNumber("point 1 y coordinate",0);
5    p2x = getNumber("point 2 x coordinate",2);
6    p2y = getNumber("point 2 y coordinate",2);
7
8    sum_difference_squared = pow((p2x - p1x),2) + pow((
        p2y - p1y),2);
9    distance = pow(sum_difference_squared, 0.5);
10
11   print("p1:", p1x, ",", p1y);
12   print("p2:", p2x, ",", p2y);
13   print("distance:" + distance);
14 }
```

<div align="center">code/code18.ijm</div>

Note that function `pow()` in the code is defined as

pow(base, exponent)
Returns the value of base raised to the power of exponent.

For example, `pow(4, 2)` returns 16.

2.4.2   Multi-parameter dialogue

In code 18 we examined above, user-interface is very poor since before calculation the user must input...click...input...click...for total of four times. To ease this exhausting series of input process, you could create a

dialog box that asks the user to input several parameters at once. We use `Dialog` functions.

```
1  //Code 18.5
2  macro "Calculate Distance 2" {
3    Dialog.create("Calculate Distance");
4    Dialog.addMessage("Calculates distance between two points
         ");
5
6    Dialog.addNumber("point1 x:", 0);    //number 1
7    Dialog.addNumber("point1 y:", 0);    //number 2
8    Dialog.addNumber("point2 x:", 2);    //number 3
9    Dialog.addNumber("point2 y:", 2);    //number 4
10   Dialog.addNumber("Scale [um/pixel]:", 0.1); //number 5
11   Dialog.addCheckbox("scale?", true);    //check 1
12
13   Dialog.show();
14
15   p1x = Dialog.getNumber(); //1
16   p1y = Dialog.getNumber(); //2
17   p2x = Dialog.getNumber(); //3
18   p2y = Dialog.getNumber(); //4
19   scale = Dialog.getNumber(); //5
20   scaleswitch = Dialog.getCheckbox();
21
22   distance = CalcDistance(p1x, p1y, p2x, p2y);
23
24   if (scaleswitch) distance *= scale;
25
26   print("p1:" + p1x + "," + p1y);
27   print("p2:" + p2x + "," + p2y);
28   if (scaleswitch) {
29     print("distance:" + distance + " [um]");
30   } else {
31     print("distance:" + distance + " [pixels]");
32   }
33 }
```
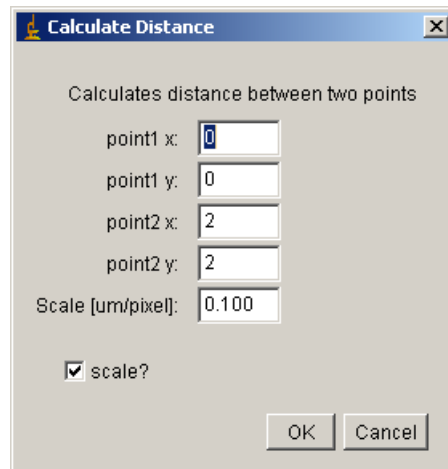
code/code18_5.ijm

Figure 2.22: Custom Parameter Input Dialog

Line 2 to 9 creates a dialog box that has multiple input boxes that looks like Fig. **??**.

- Line 3 defines the title of the dialog window.

- Line 4 texts will be shown within the window.

- Line 5 to 11 defines the parameter input fields. Fields appear in the dialog box in the order of lines with Dialog.addNumber function in the macro. When you press OK button in the dialog box, parameter will be stored in the same order.

- Line 15 to 19 These values then are assigned to each variable by Dialog.getNumber().

- Line 20 Checkbox is independent from these number fields and the value is returned by `Dialog.getCheckbox()`. When you check the check box, the return value is 1. If not, the return value is 0. We use this Boolean value (true or false) to decide if the scale will be multiplied to the distance [pixel] in Line 24.

- Line 24 This if-statement does not have braces. Such simplification is possible if there is one line when "if" is true.

You may also realize that the if statement in line 24 (and also in Line 28) does not have comparison like == or < or so on. This is because `switchscale` takes only 0 or 1 (**boolean**), which are interpreted as true (`switchscale = 1`) or false (`switchscale = 0`). So even without comparison, `switchscale` is already a decision.

Exercise 2.4.2-1

Modify Code 5 so that two parameters are asked in a single dialog box.

2.4.3   Global Variables

"Global variables" are variables that are defined outside macro or function within the same macro file. So what is good about Global variables? For instance in Code 18.5, we had a variable called `scale`. `scale` had to be typed every time when you execute the macro. One way to avoid such tedious interaction with the program is forget about the line 10 and 19, where the user input is asked for the `scale`, and instead place something like

```
scale = 0.1
```

somewhere at the beginning of the macro. This works OK, but the problem appears when there are many macros in the file, since it will be a loads of work to find the variable `scale` in the file and change the value. It could also be that the name of variable is not `scale` and something like `pixelsize`, which then you have to check what this variable is doing. Furthermore, it becomes redundant if you need to calculate the scale in every macro. For this reason, you could define the scale only once in the macro file such that:

```
1 //Code 18.75
2 var Gscale = 0.1;
3 macro "Calculate Distance 2" {
4   Dialog.create("Calculate Distance");
5   Dialog.addMessage("Calculates distance between two points
       ");
```

```
6
7   Dialog.addNumber("point1 x:", 0);    //number 1
8   Dialog.addNumber("point1 y:", 0);    //number 2
9   Dialog.addNumber("point2 x:", 2);    //number 3
10  Dialog.addNumber("point2 y:", 2);    //number 4
11  Dialog.addNumber("Scale [um/pixel]:", Gscale);  //number
        5
12  Dialog.addCheckbox("scale?", true);    //check 1
13
14  Dialog.show();
15
16  p1x = Dialog.getNumber(); //1
17  p1y = Dialog.getNumber(); //2
18  p2x = Dialog.getNumber(); //3
19  p2y = Dialog.getNumber(); //4
20   scale = Dialog.getNumber(); //5
```

code/code18_75.ijm

`var` is a statement that tells macro interpreter to treat the variable as a global variable. It should be always outside the scope (braces) of macro or function. I replaced the default value in the scale input field of the `dialog.addnumber()` at line 11 to `Gscale`, so that the initial value defined in line 2 appears in the dialog box. The value in the field could be modified by the user, but this does not affect the `Gscale` value defined in Line 2. This is because the flow of information is:

`Gscale`

> default value for the `Dialog.addNumber` field 5

> user changes the value

> stored in the `Dialog.addNumber` field 5

> scale = `Dialog.getNumber` (field 5)

So `Gscale` is referenced, but not modified. If you want to change the Global value from inside the macro, you must redefine by such as

```
Gscale = scale;
```

In the macro set below, we test the use of global variable (+ function!). The macro is for the conversion of pixel length into micrometer. The second macro changes the scale value. I usually put G for all global variable. This is not necessary, but in a file with many macros this is convenient.

```
1  //Code 19 ************* Global variable ************
2  var G_scale=0.2;
3
4  macro "convert pixel to um" {
5    length_pix = getNumber("Length? [pixel]", 10);
6    print(length_pix+" [pixel] --> " + Conv_pix2um(length_pix
         ) + " [um]");
7  }
8
9  function Conv_pix2um(in_pix){
10   in_um = in_pix * G_scale;
11   return in_um;
12 }
13
14 macro "Change Scale" {
15   new_scale = getNumber("Length? [pixel]", G_scale);
16   G_scale = new_scale;
17   print("scale chaged to " + G_scale + " [um/pixel]");
18 }
```

code/code19_globalVariable.ijm

Exercise 2.4.3-1

> Add another global variable G_scale_z ([μm]) for storing spacing in z-axis. Change the first macro, that it calculates the size of Voxel in um3. Then add another macro for changing the scale in Z axis.

2.4.4　String Arrays

Array is a powerful tool. before going into how to use it, here is an easy explanation. Imagine that an array is a stack of boxes. Boxes could contain either numbers or strings. For instance, if you have a following list of strings:

Heidelberg, Hamburg, Hixton, Grenoble, Monterotondo

An array "EMBL" could be prepared and each array element could contain one of these five strings.

Figure 2.23: EMBL array

Then when you want to retrieve some name from the array, you refer to the address within the array. So EMBL[0] will be Heidelberg, EMBL[4] will be Monterotondo, and so on. In such a way, files names contained in a folder could be listed and stored, or x-y coordinates of free-hand ROI could be stored for further use.

Here is a macro using the EMBL array example.

```
1  //Code 20
2  macro "EMBL array" {
3    EMBL = newArray(5);
4    EMBL[0] = "Heidelberg";
5    EMBL[1] = "Hamburg";
6    EMBL[2] = "Hixton";
7    EMBL[3] = "Grenoble";
8    EMBL[4] = "Monterotondo";
9    address = getNumber("which address [0-4]?", 0);
10   if ((0<=address) && (address<4)) {
11     print("address"+address+" -> "+EMBL[address]);
12   } else {
13     print("That address is somewhere else not EMBL");
14   }
15 }
```

code/code20.ijm

- Line 3 uses a function that creates a new array (`newArray()`), defined by a parameter for number of array elements (in the example case its 5) and its name `EMBL`.

- From line 4 to 8, each array from position 0 to 4 will be filled with names (Array starts with 0th element).

- Line 9 asks the user to input the address (position) within the array. Then this input address is examined if the address exists within the `EMBL` array in line 10. `EMBL.length` returns the number of "boxes" within the array. If this is satisfied, then line 10 prints out the string in that address.

Array could be created and initialized with actual values at the same time, so line 3 to 8 could be written in a single line like this:

```
EMBL = newArray("Heidelberg","Hamburg","Hixton","Grenoble",
    "Monterotondo");
for (i = 0; i < EMBL.length; i++)
    print(EMBL[i]);
```

2.4.5 Numerical Array

Array could also contain numerical values, and this way of usage is more common when you do image analysis. Here is a simple example of numerical array that prints out intensity profile along selected line ROI.

```
1  //code 20.5
2  macro "get profile and printout" {
3    if (selectionType() !=5) exit("selection type must be a
         straight line ROI");
4    tempProfile=getProfile();
5    output_results(tempProfile);
6  }
7  function output_results(rA) {
8    run("Clear Results");
9    for(i = 0; i < rA.length; i++) {
10     setResult("n", i, i);
```

```
11    setResult("intensity", i, rA[i]);
12   }
13   updateResults();
14 }
```

<div align="center">code/code20_5.ijm</div>

- Line 3: Check if the selection type is a straight line ROI. If not, macro terminates leaving a message.

    selectionType()
    Returns the selection type, where 0=rectangle, 1=oval, 2=polygon, 3=freehand, 4=traced, 5=straight line, 6=segmented line, 7=freehand line, 8=angle, 9=composite and 10=point. Returns -1 if there is no selection.

- Line 4: Empty array `tempProfile` is loaded with the intensity profile along the line ROI by `getProfile`().

-   **getProfile**()
    Runs Analyze/Plot Profile (without displaying the plot) and returns the intensity values as an array.

- Line 5: Passing the array `tempProfile` to function "output_results", which prints the content of array in Results window.

- Line 7 to 14: A function for outputting the profile array in the result table. It takes an argument `rA`, which is supposed to be an array.

- Line 8: Clears the result table.

- Line 9 to 12: for-loop to go through the array and to print out each element.

- Line 10: Sets the pixel position along the segment in the column labeled "n".

- Line 11: Sets the content of the array (pixel intensity) in the column labeled "intensity".

<div align="center">57</div>

setResult("Column", row, value) Adds an entry to the ImageJ results table or modifies an existing entry. The first argument specifies a column in the table. If the specified column does not exist, it is added. The second argument specifies the row, where 0<=row<=nResults. (nResults is a predefined variable.) A row is added to the table if row=nResults. The third argument is the value to be added or modified.

- Line 13: Updates the result table, so that above changes are actually reflected in the Result window.

  updateResults() Call this function to update the "Results" window after the results table has been modified by calls to the setResult() function.

Exercise 2.4.5-1

Modify code 20.5 that the macro calculates the sum of all intensity.

1. You do not need the function anymore.
2. for-loop should be used.
3. Hint: use `tempProfile.length`

2.4.6 Array Functions

Arrays could be directly treated using array functions. These functions are:

Array.concat(array1,array2) Returns a new array created by joining two or more arrays or values.

Array.copy(array) Returns a copy of array.

Array.fill(array, value) Assigns the specified numeric value to each element of array.

Array.getStatistics(array, min, max, mean, stdDev) Returns the min, max, mean, and stdDev of array, which must contain all numbers.

Array.print(array) Prints the array on a single line.

Array.rankPositions(array) Returns, as an array, the rank positions of array, which must contain all numbers or all strings.

Array.reverse(array) Reverses (inverts) the order of the elements in array.

Array.slice(array,start,end) Extracts a part of an array and returns it.

Array.sort(array) Sorts array, which must contain all numbers or all strings. String sorts are case-insensitive in v1.44i or later.

Array.trim(array, n) Returns an array that contains the first n elements of array.

For example, array could be sorted and reversed.

```
EMBL = newArray("Heidelberg","Hamburg","Hixton","Grenoble",
    "Monterotondo");
Array.print(EMBL);
Array.sort(EMBL);
Array.print(EMBL);
Array.reverse(EMBL);
Array.print(EMBL);
```

The output of this code is:

```
1 Heidelberg,Hamburg,Hixton,Grenoble,Monterotondo
2 Grenoble,Hamburg,Heidelberg,Hixton,Monterotondo
3 Monterotondo,Hixton,Heidelberg,Hamburg,Grenoble
```

The first line is printed in the order when the array was initialized. After sorting, names are in alphabetical order. Third line shows the reversed elements.

2.4.7   Build-in Macro Functions using Array

Array is used in many built-in macro functions, especially as the array of numerical values. Here is a list of functions which use array.

```
Dialog.addChoice("Label", items)

Dialog.addChoice("Label", items, default)

Fit.doFit(equation, xpoints, ypoints)

Fit.doFit(equation, xpoints, ypoints, initialGuesses)

getFileList(directory)

getHistogram(values, counts, nBins[, histMin, histMax])

getList("window.titles")

getList("java.properties")

getLut(reds, greens, blues)

getProfile()

getRawStatistics(nPixels, mean, min, max, std, histogram)

getSelectionCoordinates(xCoordinates, yCoordinates)

getStatistics(area, mean, min, max, std, histogram)

makeSelection(type, xcoord, ycoord)

newArray(size)

newMenu(macroName, stringArray)

Plot.create("Title", "X-axis Label", "Y-axis Label", xValues,
yValues)

Plot.add("circles", xValues, yValues)

Plot.getValues(xpoints, ypoints)
```

```
setLut(reds, greens, blues)

split(string, delimiters)
```

2.4.8   Application of Array in Image Analysis

To learn the actual use of Array in Image analysis, I will introduce several example applications.

Extending Stack Analysis by Direct Measurements

We studied how to use for-loops to measure each frame/slice within a stack (**??**). There we did measurements using `run("Set Measurements...")` and `run("Measure")` and then results were shown in the Results Table. To use those values in the table for such as to calculate statistics or to plot the results, one needs to access the results table and parse all the values. Writing all this in a macro is possible but it will be indirect as the values we measured are only accessible via the Results Table. We could now try to do measurement directly and store results in arrays. There are two ways.

1. `getRawStatistics(nPixels, mean, min, max, std, histogram)`

2. `List.setMeasurement`

The function `getRawStatistics` will measure statistical parameters from the image and returns those values in the variables declared as arguments. In other words after having this command, variable `mean` will have the mean intensity of the image. If a ROI is selected, mean intensity of that ROI will be the value of `mean`. We could loop each slices/frames within a stack and for each loop we could do `getRawStatistics` and store measured values in arrays. This is doable, but has a demerit of using this function: the available parameters to measure is limited.

The second method using `List.setMeasurement` does not have this demerit. One could measure many more parameters because all the available parameters in `[Analyze > Set Measurements...]` are accessible with this function. The basic usage is like shown in below.

```
1 List.setMeasurements;
2 mean = List.getValue("Mean");
3 print(mean)
```

This code measures currently active image, extracts specific measurement value (in the above case "Mean" intensity) and then prints out that value in the log window. We could do this measurement for every loop for stack slices/frames and store the results in arrays. Here is the code, a modified version of code 10.

```
1  //Code 10.1
2  requires("1.42i");
3  macro "Measure Ave Intensity Stack" {
4    frames=nSlices;
5    meanA = newArray(frames)
6    sdA = newArray(frames)
7    for(i=0; i<frames; i++) {
8      currentslice=i+1;
9      setSlice(currentslice);
10     List.setMeasurements;
11     meanA[i] = List.getValue("Mean");
12     sdA[i] = List.getValue("StdDev");
13
14   }
15   Array.print(meanA)
16   Array.print(sdA)
17 }
```

code/code10_1.ijm

- Line 2: Checks the ImageJ version, since `List.setMeasurements` function is only available after version 1.42i.

- Line 5, 6: Create new arrays with a lenght same as the number of frames within the stack. These arrays will be uses to store measurement results.

- Line7: for-looping to go through each frames in the stack.

- Line 10: Do measurement. All the parameters will be stored in the List.

- Line 11, 12: Retrieve the results, mean intensity and standard deviation.

- Line 15, 16: Prints out results in the log window.

Acquiring intensity profile from segmented line ROI

In recent version of ImageJ, selection thickness controls the width of segmented line ROI when you do [Analyze > Plot Profile]). We try to mimick this behavior in macro, and instead of choosing the line ROI thickness using GUI, the macro asks the user to input the thickness.

In the code below, there is only one macro. Two functions are added at the bottom. One is for profile plotting and the last one is for listing intensity profile data in the result table. Strategy of this macro is to use straight line selection for each segment, measure that segment and then profiles are concatenated to the total profile array.

```
1  //code 20.75 Array application
2  macro "get segmented line profile wide" {
3    if (selectionType() !=6) exit("selection type must be
        segmented line ROI");
4    getSelectionCoordinates(xCA, yCA);
5    width = getNumber("ROI Width?", 9);
6    op = "line=" + width;
7    run("Line Width...", op);
8    totalprofile = newArray(0);
9    for (i = 0; i < xCA.length-1; i++) {
10     makeLine(xCA[i], yCA[i], xCA[i+1], yCA[i+1]);
11     thisprofile = getProfile();
12     totalprofile = Array.concat(totalprofile, thisprofile);
13   }
14   K_createThickProfilePlot(totalprofile);
15   output_results(totalprofile);
16 }
17
18 //*********Graph Plotting *******
```

```
19
20 function K_createThickProfilePlot(pA) {
21   Array.getStatistics(pA, min, max, mean, sdev);
22   Plot.create("Intensity profile", "pixels", "intensity");
23   Plot.setLimits(0, pA.length, min * 0.95, max * 1.05);
24   Plot.setColor("black");
25   Plot.add("line", pA);
26   Plot.show();
27 }
28
29 //results output to a table
30 function output_results(rA) {
31   run("Clear Results");
32   for(i = 0; i < rA.length; i++) {
33     setResult("n", i, i);
34     setResult("intensity", i, rA[i]);
35   }
36   updateResults();
37 }
```

<div align="center">code/code20_76.ijm</div>

- Lines 2 - 16: Main part, macro for the segmented line ROI measurement.

- Line 3: Check if the selection type is a segmented line ROI. If not, macro terminates leaving a message.

- Line 4: Reads the x and y coordinates of the segmented line and store them in two arrays xCA and yCA.

  getSelectionCoordinates(xCoordinates, yCoordinates)
  Returns two arrays containing the X and Y coordinates of the points that define the current selection.

- Line 5 - 7: Asks the user to input width of the segmented ROI. The ROI line width is set to that value.

- Line 8: A new array totalprofile is created, initialized without any element. This new array will store the profile data of full ROI.

- Line 9 - 13: Profile measurement by placing straight line ROI, for wach segment of the original ROI. `makeLine` function is used for this purpose, and `getProfile` returns intensity profile of the corresponding line ROI. Profile data in `thisprofile` array are concatenated to `totalprofile` array using `Array.concat`.

  makeLine(x1, y1, x2, y2)
  Creates a new straight line selection. The origin (0,0) is assumed to be the upper left corner of the image. Coordinates are in pixels. With ImageJ 1.35b and letter, you can create segmented line selections by specifying more than two coordinate, for example make-Line(25,34,44,19,69,30,71,56).

- Line 14: Call graph plotting function (Line 20 - 27), passing `totalprofile` array as an argument.

- Line 15: call function to printout the profile array in the results window (Lines 32 - 39).

- Line 20 - 27: Function for plotting the intensity profile.

- Line 21 : Use `Array.getStatistics` function to know the minimum and the maximum value of the array that was given as argument.

- Line 22: Creates the window and axes of the plot.

- Line 23: Set the range for x and y axis using the results of line 21 `min` and `max`. 5% of offset is added to both values for some margins below and above.

- Line 24: Sets the color of the plot.

- Line 25: Plot the profile.

- Line 26: Show the plot on the screen (lot is hidden until this show() function).

- Line 30 - 37: Function for outputting the profile array in the result table. This function is exactly the same function you already used in the previous chapter (code 20.5).

2.5 File I/O

Analysis of images requires both input and out put: input is to load images, and output is to save either processed images or numerical data. If number of image files or quantity data is manageable by manual loading and saving, we do not have to automate. But in some cases you need to process a huge number of files. This often happens especially after you establish a protocol and you want to get statistically sufficient amount of data. Then you need to automate file input and output using macro. Once you learn how to write File I/O program, you can process as much files as you want, as long as your memory space allows.

2.5.1 Saving the Measurement Results Automatically

When you have a time series sequence and you want to measure multiple signals with multiple parameters in each frame, measurement results in each frame needs to be somehow saved. Here, we learn how to export measurement results in your hard disk automatically using macro.

Open the sample image `Nucseq001.tif`. Cell nucleus shows that they divide and increase their number over time. We want to count the number of nucleus in each frame to know the dynamics of increase. At the same time, we may also want to see changes in the signal intensity and shape. For this measurement Particle Analysis function works best. Do the following:

1. `[Image -> Adjust -> Threshold]`. Threshold the image and check the threshold lower and upper value that segments the nucleus optimally.

2. Set the measurement parameters. `[Analysis -> set measurements...]`

   (a) Check Area, Mean intensity, centroid, Circularity and Slice number.

   (b) Check "limit to threshold"

   (c) Digits after decimal point: 2

3. `[Analyze -> Analyze Particles...]`

   (a) Size: 10 - Infinity

   (b) Circularity = 0.5 - 1.0

          (c) Show: Outline

          (d) Check Display Results

          (e) Check Exclude on Edges

          (f) Check Clear Results

    4. Then click "OK".

After these steps, you will find outline image showing detected cells and a result table.

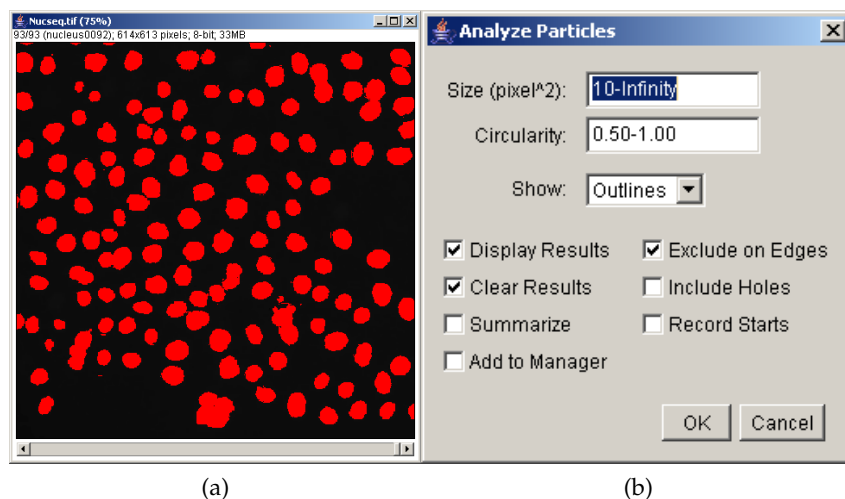<center>(a)   (b)</center>

Figure 2.24: (a) Thresholded cell image and (b) Particle Analysis parameter input dialog.

Using macro recorder, its easy to write a macro set as following.

```
1  // Code 21
2  var G_Ddir = "D:\\_Kota\\CMCI\\course_macro\\"; //Windows
      path
3  //var G_Ddir = "/Users/Kota/CMCI/course_macro/"; //OSX path
4
5  macro "Set Directory to save Results" {
6    G_Ddir = getDirectory("Choose Destination Directory");
7    print(G_Ddir);
8  }
9
10 macro "auto save results" {
11   getThreshold(lower, upper);
```
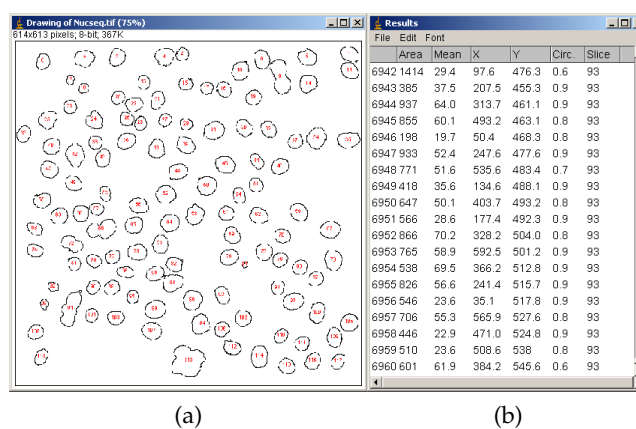
Figure 2.25: After particle analysis is done, (a) outlined cell image and (b) results table listing measurement results.

```
12  if ((lower == -1) && (upper == -1)) {
13    exit("Image must be thresholded");
14  }
15  setThreshold(lower, 255);
16  img_title = getTitle();
17  run("Set Measurements...", "area mean centroid shape
        circularity slice limit redirect=None decimal=2");
18  run("Analyze Particles...", "size=10-Infinity circularity
        =0.50-1.00 show=Outlines display exclude clear stack")
        ;
19  dest_filename = img_title+"_measure.xls";
20  fullpath = G_Ddir + dest_filename;
21  saveAs("Measurements", fullpath);
22 }
```

code/code21.ijm

Two macros and a global variable consists this macro set. Global variable is a string variable that stores the path to the location where file will be saved. (note: path is differently written in MacOS. It uses slash instead of backslash). The first macro `Set Directory to save Results` is for setting the path to the folder (or directory) where the file will be saved. We use a macro function `getDirectory(title)` to get user choice of a destination folder.

getDirectory(title)

Returns the path to a specified directory. If title is "startup", returns the path to the directory that ImageJ was launched from (usually the ImageJ directory). If it is "plugins" or "macros", returns the path to the plugins or macros folder. If it is "image", returns the path to the directory that the active image was loaded from. If it is "home", returns the path to users home directory. If it is "temp", returns the path to the /tmp directory. Otherwise, displays a dialog (with title as the title), and returns the path to the directory selected by the user. Note that the path returned by get-Directory() ends with a file separator, either "̈(Windows) or "/". Returns an empty string if the specified directory is not found or aborts the macro if the user cancels the dialog box.

When you run this first macro, global string variable `G_Ddir` will be set to a folder where user will select, and line 6 prints out the path to a folder (or directory). It might be convenient for you to change the default directory path in the code above (line 2), by copying the results in the log window and pasting it in the macro.

The measurement macro starts from the Line 9.

- Line 10 to 13: Checks if the image is thresholded.

- Line 14: Sets the threshold level.

- Line 15: Gets the title of the image window for later use. We use this for generating name of the results file.

- Line 16: to 17: Sets the measurement parameter and does the actual particle analysis. Macro functions are direct copies from the recorder.

After the analysis, we want to save the results.

- Line 18: Generates the file name using the image file name stored in the line 15 by concatenating concatenate image title with `_measure.xls`.

- Line 19: The full path file name is constructed by adding the result filename generated in line 18 with path stored in the global variable.

69

- Line 20: Saving the result table as an excel-readable file uses a new macro function `saveAs`:

  saveAs(format, path) Saves the active image, lookup table, selection, measurement results, selection XY coordinates or text window to the specified file path. The format argument must be "tiff", "jpeg", "gif", "zip", "raw", "avi", "bmp", "fits", "png", "pgm", "text image", "lut", "selection", "measurements", "xy Coordinates" or "text". Use saveAs(format) to have a "Save As" dialog displayed.

  Path in line 20 is a full path constructed in the previous line 19.

Exercise 2.5.1-1

Create a new macro file and write the code 21. If it works, save the macro as "macro_fileIO.ijm". We use it in the next section (.ijm is the extension for imageJ macro). Then modify the code so that user can change the size-range for the particle analysis. Save the file separately.

2.5.2 Batch Processing of Files

What should we do if we have more stacks that should be analyzed? Should we open each of the stack and execute the macro? A better idea is to automate the loading process also. For this, we modify and extend the code written in the previous section. The tasks are:

- task a: List files in a folder.

- task b: Open a file, do analysis, save results and close the file.

- task c: Do this until all files are analyzed.

A very useful function for task a is `getFileList(path)`.

getFileList(directory)
Returns an array containing the names of the files in the specified directory path. The names of subdirectories have a "/" appended.

You need to set the path to the source image containing folder (directory). We learn this macro function in the following short macro.

```
// Code 22
macro "List files in a folder" {
  dir = getDirectory("Choose a Directory ");
  list = getFileList(dir);
  for(i = 0; i < list.length; i++) {
    print(list[i]);
  }
}
```

<div align="center">code/code22.ijm</div>

Run this macro and if you choose a folder in the sample image folder containing four stacks, macro prints out texts in Log window. It should then look like figure **??**.
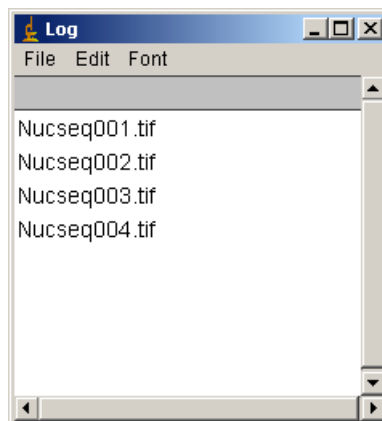

Figure 2.26: Output of code 22

- Line 2: Asks the user to select a folder. Variable `dir` is then stored with the full path to the folder.

- Line3 uses the `getFileList` function, and reads out the file names as a string array and stored in List (array).

- Line 4 to 6 is a loop. `list.length` returns the length of the list. In this way, all the contents are printed out in the window. We use this `getFileList` function to process multiple files automatically.

Let's modify code 22 so we can measure multiple stacks automatically. Code 23 (see below) works like this: You must first set two things:

1. Full path to the destination folder where the results will be saved. Macro `Set Directory to save Results`

2. Threshold level for the particle analysis. Macro `set the threshold lower level`

The first setting task is the same as you did in code 22. The second setting is done by manually opening a stack (may be the first one in the files) and manually setting the threshold as you like, then execute the second macro below. In both these settings, parameters will be saved in global variables and will be used in the main program.

When you run the main macro (`Multiple measurement`) after these two settings, the program asks you where the files are. As soon as you select a folder where files are contained, then processing and saving just proceeds automatically.

```
1  // Code 23
2
3  var G_Sdir = "D:\\_Kota\\CMCI\\";
4  var G_Ddir = "D:\\_Kota\\CMCI\\";
5  var G_threshold_lower = 20;
6  var G_threshold_upper = 255;
7
8
9  macro "Set Directory to save Results" {
10   G_Ddir = getDirectory("Choose Destination Directory");
11   print(G_Ddir);
12 }
13
14 macro "set the threshold lower level" {
15   getThreshold(lower, upper);
16   if ((lower == -1) && (upper == -1)) {
17     exit("Image must be thresholded");
18   }
19   G_threshold_lower = lower;
20 }
```

```
21
22 macro "Multiple measurement" {
23   G_Sdir = getDirectory("Choose the Directory where the
         file is");
24   list = getFileList(G_Sdir);
25   for(i = 0; i<list.length; i++) {
26     NucAnalysis(list[i]);
27   }
28 }
29
30 function NucAnalysis(img_filename) {
31   fullpath_image = G_Sdir + img_filename;
32   open(fullpath_image);
33   sourceID = getImageID();
34   setThreshold(G_threshold_lower, G_threshold_upper);
35   img_title = getTitle();
36
37   run("Set Measurements...", "area mean centroid
         circularity slice limit redirect=None decimal=2");
38   run("Analyze Particles...", "size=10-Infinity circularity
         =0.50-1.00 show=Outlines display exclude clear stack")
         ;
39   selectImage(sourceID);
40   close();
41
42   dest_outlinename = img_title + "_outline.tif";
43   fullpath = G_Ddir + dest_outlinename;
44   saveAs("tiff", fullpath);
45   close();
46
47   dest_filename = img_title+"_measure.xls";
48   fullpath = G_Ddir + dest_filename;
49   print(fullpath );
50   saveAs("Measurements", fullpath);
51 }
```

code/code23_FileIO_02.ijm

Now we have three macros and four global variables.

- Line 3: global string variable for storing path to the source folder.

- Line 4: global string variable for storing path to the destination folder, where results will be saved.

- Line5 and 6: Global numerical variables for storing threshold upper and lower values.

- Macro `Set Directory to save Results`: Line 9 to 12: First macro. This is used to set the path to the destination folder. The function is same as code 22.

- Macro `set the threshold lower level`: Line 14 to 20: This macro is for storing the lower value of the threshold in global variables, the values of which will be used in the main macro. You might have seen a similar code already: code 17, function for checking if the image is thresholded. Only difference is that in this code 23, lower threshold value is stored in the global variable defined in line 5. Upper value is not touched, kept to 255.

- Macro `Multiple measurement`: Line 22 to 28: This is the main macro (third one in this macro set). Line 23 asks the user where the files are. This path to the source file is stored in the global variable defined in line 3. Then in line 24, a list of files contained in source folder is generated and stored in the array `list`. From line 25 to 27 is a small for-loop, the number of loop is same as the length of the list. In this loop, name of file is passed one by one to the function `NucAnalysis()` that does the actual analysis...

- function `NucAnalysis(img_filename)`: Line 30 to 51 is the core of analysis. `img_filename` is string variable for file names in the list array, given as argument.

  – Line 31: Using `img_filename`, the full path name is constructed by combining two strings.

  – Line 32: A file is opened by `open(path)` function.

    open(path)
    Opens and displays a tiff, dicom, fits, pgm, jpeg, bmp, gif, lut, roi, or text file. Displays an error message and aborts the macro if the specified file is not in one of the supported formats, or if the file is not found. Displays a file open dialog box

if path is an empty string or if there is no argument. Use the File>Open command with the command recorder running to generate calls to this function. With 1.41k or later, opens images specified by a URL.

– Line 33: After opening image, its `ImageID` is stored in the variable `sourceID`.

– Line 34, the image is thresholded according to the global variables for the lower and the upper values.

– Line 35: Title of the image, which actually is the file name, is retrieved and stored in the variable `img_title`.

– Line 38: The particle analysis is then applied to the image (Lines 34 to 37 are same as lines 14 to 17 in code 21).

– Line 38: Source image you opened from the hard disk is activated and then closed in line 39. Activation of the image by `selectImage()` is required, because there is already a new stack (outline stack!), so that original stack is already behind. Therefore to close the original, one must activate the image by using `selectImage()` function.

– After all these processing and measurement, results will be saved. Lines 42 to 45 saves the outline stack. Outline stack is also closed after saved (line 45). Line 47 to 50 is exactly same as the Result table saving you did in code 21 (lines 18 to 20). Line 49 is added, just for an additional information printout in Log window.

2.6 Working with Texts

With some advanced macro programming, you might need to deal in more fine-tuned manner with texts. For example, let's think about a title of an image "exp13_C0_Z10_T3.tif". Such naming occurs often to indicate that this image is from third time point, at 11th slice (imagine that the Z slice numbering starts from 0) and its first channel.

We might be lucky enough to know such information from header, but in many cases such information is only available from the file name (the title of the image). To extract dimensional information from such file name, we need to know how to work with texts (or Strings) to decompose them and get information that we need. Build-in macro functions which are related to such tasks with texts are the following.

- lengthOf(str)

- substring(string, index1, index2)

- indexOf(string, substring)

- indexOf(string, substring, fromIndex)

- lastIndexOf(string, substring)

- startsWith(string, prefix)

- endsWith(string, suffix)

- matches(string, regex)

- replace(string, old, new)

Let's go back to the example file name "exp13_C0_Z10_T3.tif" again. If we need to get the file name without file extension, what should we do? Several ways are there, but lets start with the simplest one. We assume that we already know that all the file names we are dealing with are TIFF format, so we know that they all ends with ".tif". So we could replace the ".tif" with a 0 length string. we could do this by using `replace`.

76

```
1 name = "exp13_C0_Z10_T3.tif";
2 newname = replace(name, ".tif", "");
3 print(newname);
```

This will print out "exp13_C0_Z10_T3" in the log window. In the second line, the function `replace` is used. The old string ".tif" is replaced by a new 0 length string "". So it works!

But what if our lucky assumption that all files ends with ".tif" is not true and it could be anything? To work with this, we now need to use different strategy to now the file extension. By definition, file extension and the file name is separated by a dot. Length of the extension could be various, as some extension such as python file is ".py" and C code is ".c". Thus, we cannot assume that the length of the file extension is constant, but we know that there is a dot.

For such case with variable length of file extension being expected, we first need to know about the **index** of the dot within file name. Each character within file name is positioned at certain index from the beginning of the name. In the example we are now dealing with, The index 0 is "e". The index 1 is "x". The index starts from 0. Since the index starts from 0, the last index will be total length of the file name minus one. You could modify the code above like below to try getting the length of the file name.

```
1 name = "exp13_C0_Z10_T3.tif";
2 tlength = lengthOf(name);
3 print(tlength);
```

You should see "19" in the log window. That is the length of this file name. So in this example string, index starts from 0 and the last index is 18.

Next, you need to know is about the function `substring(string, index1, index2)`. With this function, you could extract part of the `string` by the start index and the end index. We could just try this by again modifying the code above.

```
1 name = "exp13_C0_Z10_T3.tif";
2 subname = substring(name, 0, 3);
```

```
3 print(subname);
```

In the log window, there should be "exp". The second argument in the code above is 0, and the third is 3. This tells the substring function to extract characters from the index 0 to the index 2 should be extracted (so the third argument will be the index just after the last index that would be included in the substring).

Exercise 2.6.0-1

> Test changing the second and the third argument so that different part of the file name is extracted.

How could we know the index of the dot? For this we use the `indexOf(string, substring)`. Try the following code.

```
1 name = "exp13_C0_Z10_T3.tif";
2 dotindex = indexOf(name, ".");
3 print(dotindex);
```

Now you know that the index of dot is "15". We could then combine the knowledge we have now to compose a single macro that extracts the filename without file extension.

```
1 name = "exp13_C0_Z10_T3.tif";
2 dotindex = indexOf(name, ".");
3 filename = substring(name, 0, dotindex);
4 print(filename);
```

Let's make the problem a bit more complicated. If the file name contains multiple dots, what should we do? In the example below, I added two more dots.

```
1 name = "exp13._C0._Z10_T3.tif";
2 dotindex = indexOf(name, ".");
3 filename = substring(name, 0, dotindex);
4 print(filename);
```

Output is now "exp13". Far from what we need. To treat such case, we use `lastIndexOf`, which returns the index of the last appearance of the given character. Let's slightly modify the code.

```
1 name = "exp13._C0._Z10_T3.tif";
2 dotindex = lastIndexOf(name, ".");
3 filename = substring(name, 0, dotindex);
4 print(filename);
```

It should then working again as we want.

Let's change our task: We now want to know the time point that this image was taken. How should we do that? Examining the file name again, we realize that the time point number appears after "T". The number could be any length of digits, but currently is 0. Then the dot comes right after the number. We then just need to know the index of "T" ... but wait, we might have "T" anywhere, as this is a single character alphabet that could easily be a file name. Therefore we find the index of "_T" that looks like more specific.

```
1 name = "exp13._C0._Z10_T3.tif";
2 timeindex = indexOf(name, "_T");
3 print(timeindex);
```

Now we know that "_T" is at index 14, so the number should start from the index 16 (because index 15 will be "T"). Taken this into account, we could extract the time point.

```
1 name = "exp13._C0._Z10_T3.tif";
2 timeindex = indexOf(name, "_T");
3 dotindex = lastIndexOf(name, ".");
4 timepoint = substring(name, timeindex + 2, dotindex);
5 print(timepoint);
```

The time point that you have just now captured is a string. You can not pass this to mathematical assignments. To do so, you need to convert this to a number. For doing so, you could use `parseInt(string)`.

```
1 name = "exp13._C0._Z10_T3.tif";
2 timeindex = indexOf(name, "_T");
3 dotindex = lastIndexOf(name, ".");
4 timepoint = substring(name, timeindex + 2, dotindex);
5 timepoint = parseInt(timepoint);
6 print(timepoint * 2);
```

An example case where conversion of string to number (in this case an integer) required would be when you need to compare such file names and get the maximum time point from all the file names. Usage is diverse, but at some point you need to use this. If you need a Float number (numbers with decimal point), use `parseFloat(string)`.

2.7   Secondary Measurement

In this section we learn a macro usage which you may often encounter in actual situations: We do certain measurement first. We then use results from this first measurement for setting parameters of second measurement.

We take following example of secondary measurement:

1. We first measure XY coordinates of moving particles by particle tracking.

2. Using these XY data, we measure changes in pixel intensity of the particle.

There could be two cases of how you get the data out and load it into currently running macro. First is to do so directly from data table within ImageJ, and the other is to access data file saved in hard disk. We learn both.

2.7.1 Using Values in Results Window

ParticleTracker is an excellent plugin for automated tracking of spherical particles[16]. We use this plugin first to get tracked data.

Exercise 2.7.1-1

> Open sample image stack **TransportOfEndosomalVirus.tif**. Then do
> `[Plugins > Particle Detector & Tracker> Particle Tracker].`
> A parameter input dialog window appears. Fill in parameters as follows:
>
> - radius: 3
> - cutoff: 0
> - percentile: 0.3
> - link range: 1
> - distance: 20

Now, you should see a results window that looks like figure **??**. In there, it should be reported that over 100 trajectories were detected. You could see how they look like by clicking "Visualize All Trajectories". Another window overlaid with colorful tracks appears (Fig. **??**). Click "Filter Options" and input 10, so that short trajectories become invisible in the window. Use your mouse and select one of trajectory by clicking. Rectangle ROI is created in the surrounding of the selected track. Go back to the Results window (Fig. **??**) and click "Focus on Selected Trajectory". Then you will see another window is created with only the track you chose (Fig. **??**). Check carefully if the tracking was done properly. If you are satisfied, go back to the result window (Fig. **??**) again then click "selected trajectory to Table". You will then find the trajectory data is transferred to the Results table of ImageJ (Fig. **??**).

---

[16]As of Nov. 2010, we have a largely updated version of ParticleTracker plugin available at the ETH site. This 2D/3D implemented version could be downloaded from ETH site http://www.mosaic.ethz.ch/Downloads/ParticleTracker.This plugin is added with many new features but there is some bugs still. With some measurement conditions, the new plugin returns error and crashes. For this reason, please download the plugin
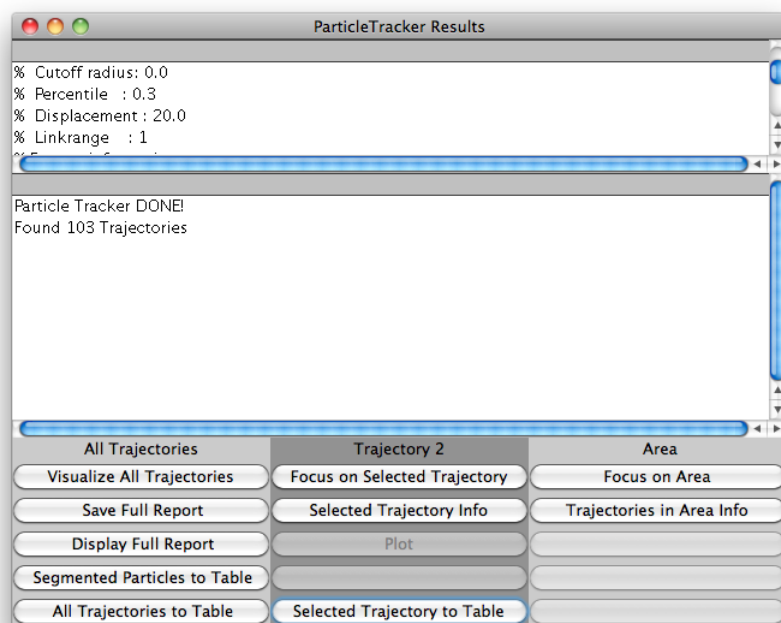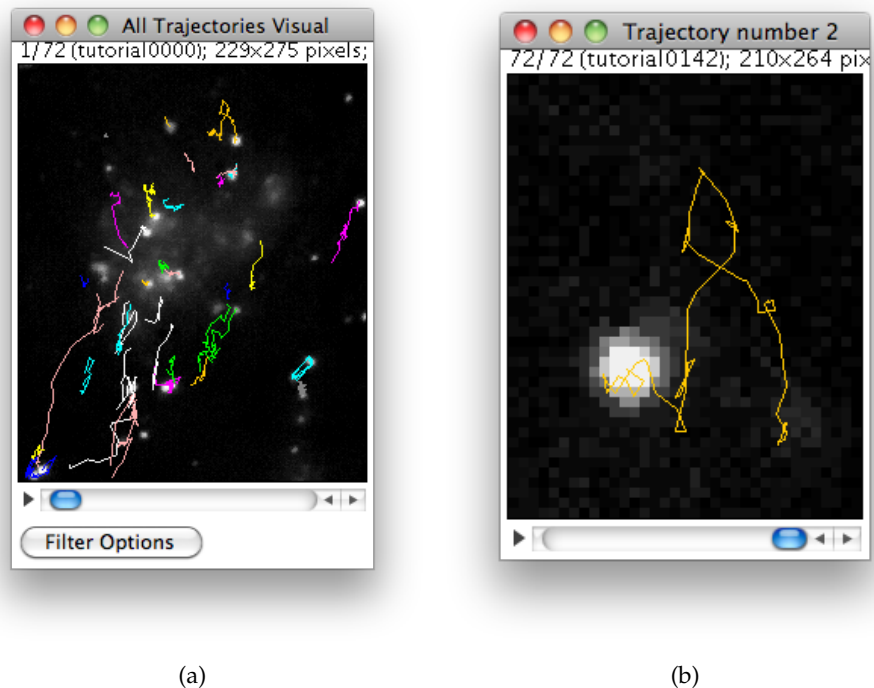
Figure 2.27: ParticleTracking Results

(a)   (b)

Figure 2.28: (a) ParticleTracking Trajectories, all, and (b) Focus on Single Track.

	Trajectory	Frame	x	y	m0	m2	NPscore
52	2	51	46.609	132.039	13.905	3.937	15.492
53	2	52	42.795	133.876	15.038	3.871	6.008
54	2	53	43.087	133.184	15.608	3.884	16.217
55	2	54	45.806	133.147	14.768	3.897	2.692
56	2	55	49.981	132.031	15.070	3.829	8.713
57	2	56	49.843	133.016	15.042	3.802	0.093
58	2	57	47.672	132.205	14.882	3.549	0.073
59	2	58	46.056	130.025	14.728	3.496	0.003
60	2	59	43.120	129.261	14.736	3.317	0.914
61	2	60	42.823	128.953	14.891	3.338	0.134
62	2	61	43.277	128.036	15.054	3.566	0.017
63	2	62	46.010	126.042	15.080	3.802	8.385
64	2	63	45.139	125.207	13.525	3.878	6.621
65	2	64	44.265	124.955	13.109	3.869	0.432
66	2	65	46.040	125.262	13.023	3.705	15.256
67	2	66	46.130	125.253	13.938	3.677	25.290
68	2	67	44.433	126.455	11.515	3.716	6.773
69	2	68	44.876	127.177	14.116	3.785	10.278
70	2	69	46.446	127.986	12.617	3.837	5.565
71	2	70	44.853	128.845	14.350	3.781	5.046
72	2	71	43.778	127.255	14.643	3.660	0.280

Figure 2.29: ParticleTracking Results transferred to ImageJ Results Table

83

So now, what we have to do is access results table, get XY coordinates from there and do intensity measurements at corresponding positions. To get data out of results table, we use the following macro function:

getResult("Column", row)
Returns a measurement from the ImageJ results table or NaN if the specified column is not found. The first argument specifies a column in the table. It must be a "Results" window column label, such as "Area", "Mean" or "Circ.". The second argument specifies the row, where 0<=row<nResults. nResults is a predefined variable that contains the current measurement count. (Actually, it's a built-in function with the "()" optional.) Omit the second argument and the row defaults to nResults-1 (the last row in the results table). See also: nResults, setResult, isNaN, getResultLabel.

Let's first test with a short macro that reads data from Results table and print out XY coordinates in the Log window.

```
1  for (i = 0; i < nResults; i++){
2    frame = getResult("Frame", i) + 1;
3    ypos = getResult("x", i);
4    xpos = getResult("y", i);
5    print(frame + ", " + xpos + ", " + ypos);
6  }
```

<div align="center">code/code24.ijm</div>

At line 1, `nResults` is a function that returns number of rows in the Results table. Frame number is added with 1 in the line 2 because frame number in ParticleTracker plugin starts from 0, while it starts from 1 in ImageJ. In line 3 and 4, xpos and ypos is inverted, because ParticleTracker program was originally wrote in Matlab and for that convention (in Matlab, vertical diretion is called "X" and horizontal direction is called "Y", and this is common to matrix calculation software since row = X and column = Y), XY data should be inverted for use in in ImageJ.

If you check the log window and if you are confident with data read out from Results window, we could now add the code with lines to measure

---

from CMCI site for the exercise in this textbook. http://cmci.embl.de/downloads/particletracker2d.

intensity by placing circular ROI at XY coordinates of trajectory. Here we go.

```
1  stacktitle = "TransportOfEndosomalVirus.tif";
2  diam = 9;
3  offset = floor(diam/2);
4
5  for (i = 0; i < nResults; i++){
6    frame = getResult("Frame", i) + 1;
7    ypos = getResult("x", i);
8    xpos = getResult("y", i);
9    print(frame + ", " + xpos + ", " + ypos);
10   selectWindow(stacktitle);
11   setSlice(frame);
12   makeOval(xpos-offset, ypos-offset, diam, diam);
13   getRawStatistics(nPixels, mean, min, max, std);
14   setResult("RoiInt", i, mean);
15
16 }
```
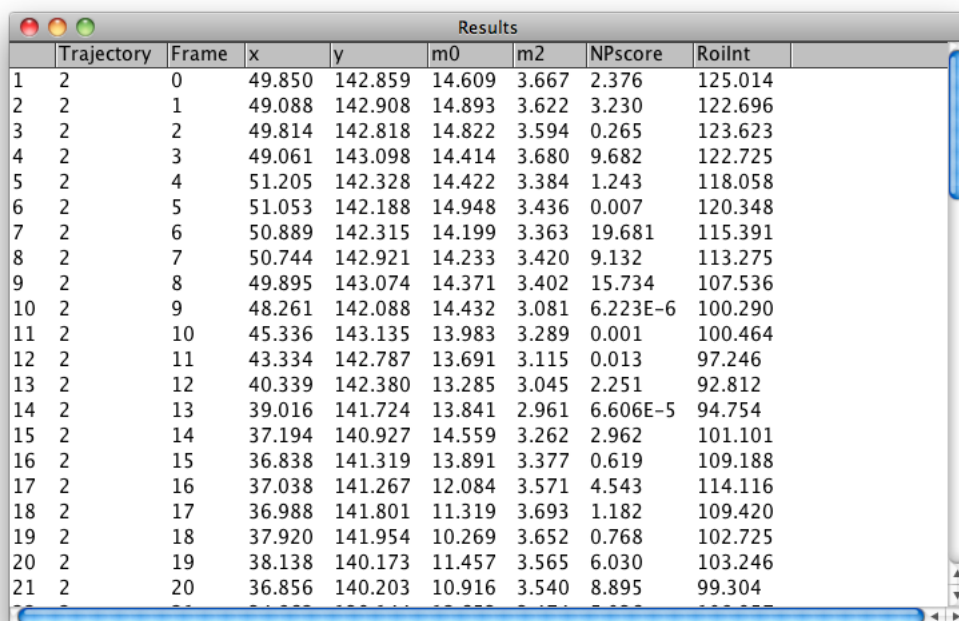
<div align="center">code/code24_5.ijm</div>

Running this macro, you should see a new column in Results window with header title "RoiInt", where measured intensity is listed (Fig. **??**).

Explanation of the code: In the first line, we set the name of the image stack so that we are sure with which window to be measured with intensity. Line 2 and 3 are for setting the size of oval ROI.

The way oval ROI is created is what you have learned already in detail in the section **??**. `getRawStatistics()` returns basic parameters of the selected ROI, and is more convenient than using `run("Measure")`.

getRawStatistics(nPixels, mean, min, max, std, histogram)
This function is similar to getStatistics except that the values returned are uncalibrated and the histogram of 16-bit images has a bin width of one and is returned as a max+1 element array. For examples, refer to the ShowStatistics macro set. See also: calibrate and List.setMeasurements

	Trajectory	Frame	x	y	m0	m2	NPscore	RoiInt
1	2	0	49.850	142.859	14.609	3.667	2.376	125.014
2	2	1	49.088	142.908	14.893	3.622	3.230	122.696
3	2	2	49.814	142.818	14.822	3.594	0.265	123.623
4	2	3	49.061	143.098	14.414	3.680	9.682	122.725
5	2	4	51.205	142.328	14.422	3.384	1.243	118.058
6	2	5	51.053	142.188	14.948	3.436	0.007	120.348
7	2	6	50.889	142.315	14.199	3.363	19.681	115.391
8	2	7	50.744	142.921	14.233	3.420	9.132	113.275
9	2	8	49.895	143.074	14.371	3.402	15.734	107.536
10	2	9	48.261	142.088	14.432	3.081	6.223E-6	100.290
11	2	10	45.336	143.135	13.983	3.289	0.001	100.464
12	2	11	43.334	142.787	13.691	3.115	0.013	97.246
13	2	12	40.339	142.380	13.285	3.045	2.251	92.812
14	2	13	39.016	141.724	13.841	2.961	6.606E-5	94.754
15	2	14	37.194	140.927	14.559	3.262	2.962	101.101
16	2	15	36.838	141.319	13.891	3.377	0.619	109.188
17	2	16	37.038	141.267	12.084	3.571	4.543	114.116
18	2	17	36.988	141.801	11.319	3.693	1.182	109.420
19	2	18	37.920	141.954	10.269	3.652	0.768	102.725
20	2	19	38.138	140.173	11.457	3.565	6.030	103.246
21	2	20	36.856	140.203	10.916	3.540	8.895	99.304

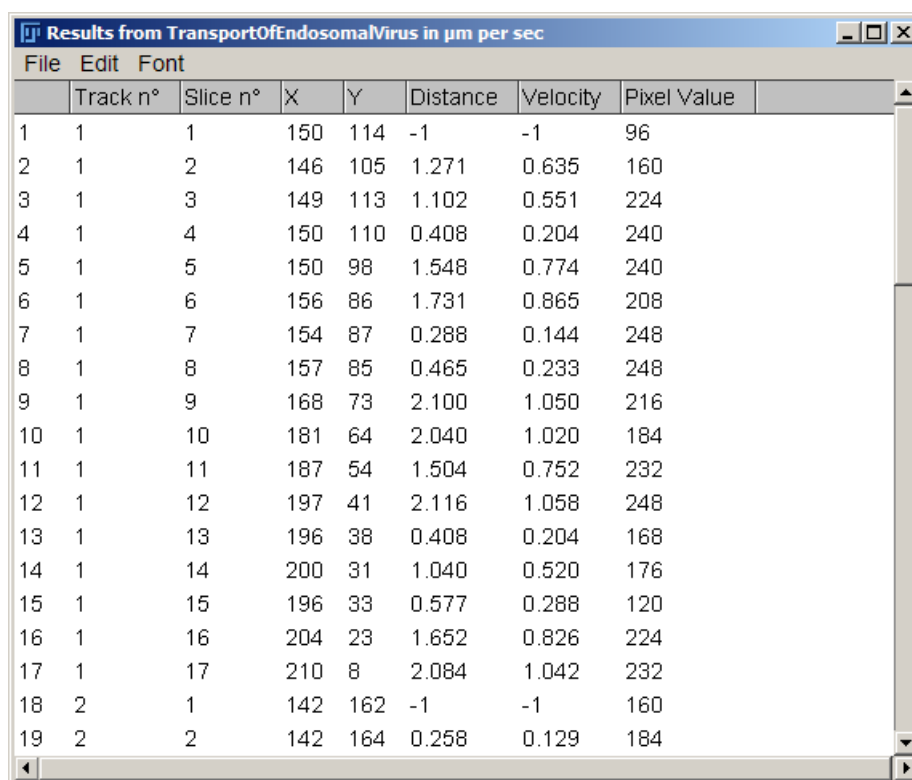Figure 2.30: Results Table with Intensity Measurement
Column "RoiInt"

2.7.2 Using values in non-Results table

Next, we study a case when data are shown in non-Results Table. Manual Tracking plugin is another way of measuring particle movement, and utilizes non-Results table.

Exercise 2.7.2-1

> Open sample image stack **TransportOfEndosomalVirus.tif** and track
> at least two virus manually[17]. In ImageJ, you should install this plu-
> gin by yourself. In Fiji, Manual Tracking plugin could be found at
> `[Plugins > tracking >]`.

After the tracking, we have a results table that looks like figure **??**.

	Track n°	Slice n°	X	Y	Distance	Velocity	Pixel Value	
1	1	1	150	114	-1	-1	96	
2	1	2	146	105	1.271	0.635	160	
3	1	3	149	113	1.102	0.551	224	
4	1	4	150	110	0.408	0.204	240	
5	1	5	150	98	1.548	0.774	240	
6	1	6	156	86	1.731	0.865	208	
7	1	7	154	87	0.288	0.144	248	
8	1	8	157	85	0.465	0.233	248	
9	1	9	168	73	2.100	1.050	216	
10	1	10	181	64	2.040	1.020	184	
11	1	11	187	54	1.504	0.752	232	
12	1	12	197	41	2.116	1.058	248	
13	1	13	196	38	0.408	0.204	168	
14	1	14	200	31	1.040	0.520	176	
15	1	15	196	33	0.577	0.288	120	
16	1	16	204	23	1.652	0.826	224	
17	1	17	210	8	2.084	1.042	232	
18	2	1	142	162	-1	-1	160	
19	2	2	142	164	0.258	0.129	184	

Figure 2.31: Manual Tracking Results

---

[17]For detailed instruction on how to use Manual tracker, see corresponding section in CMCI Image Processing and Analysis Course I Basic.

To extract these data and use it for the secondary measurement, you might immediately think of using `getResult(column header, row)` as we did in the previous subsection.

That should then be pretty straight forward...But if you try this, you would see that this function returns error and does not work in case of Manual Tracking plugin. This is because result table created by Manual Tracking plugin is not the genuine ImageJ Results window. For such non-genuine results window, we retrieve data using the following function.

> **getInfo("window.contents")**
> If the front window is a text window, returns the contents of that window. If the front window is an image, returns a string containing the text that would be displayed by Image>Show Info. Note that `getImageInfo()` is a more reliable way to retrieve information about an image. Use
> `split(getInfo(),"\n")`
> to break the string returned by this function into separate lines. Replaces the `getInfo()` function.

This function returns a string with the content of the table. Try the following two lines to see how it works.

```
str = getInfo("window.contents");
print(str);
```

If you run these two lines, you will see data printed out in the Log window (Fig. **??**).

So far so good, we succeeded in getting data out of the results table. Then what we need to do now is to play around with `str` variable, where all the data is now contained as a chunk. Since this chunk of data is not usable directly, we first split the `str` to single lines of string array. For this we use `split` function, the definition of which is

> **split(string, delimiters)**
> Breaks a string into an array of substrings. Delimiters is a string contain-
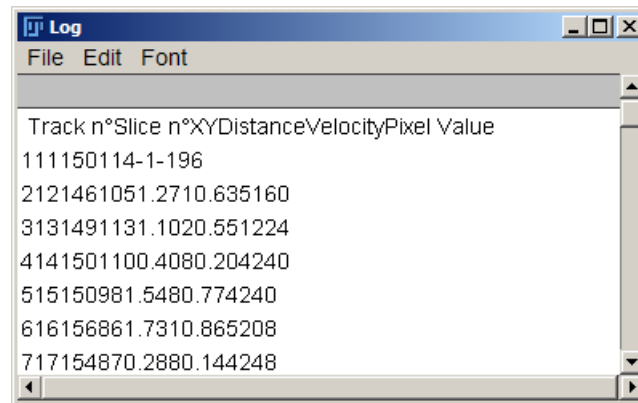
Figure 2.32: Manual Tracking Results in Log window

ing one or more delimiter characters. The default delimiter set " \t\n\r" (space, tab, newline, return) is used if delimiters is an empty string or split is called with only one argument. Returns a one element array if no delimiter is found.

Using this function to convert the string to a string array and adding two more lines to check the content of array, code now looks like this:

```
str = getInfo("window.contents");
//print(str);
strA = split(str, "\n");
print(strA[0]);
print(strA[1]);
```

We use delimiter \n which means "new line", a hidden character in str that feeds new line to form a table. When we run above code we will see two data shown in the log window (Fig. **??**).

We then still need to split each single line to individual data for each column. We modify the code as follows:

```
str = getInfo("window.contents");
strA = split(str, "\n");
lineA = split(strA[1], "\t");
```
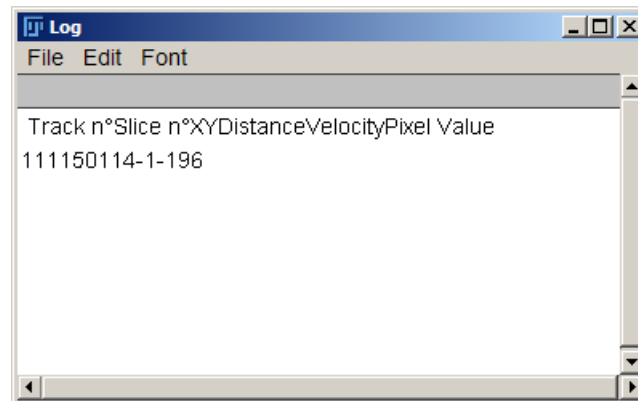
Figure 2.33: Two Lines from data in the Log window

```
print(lineA[3]);
```

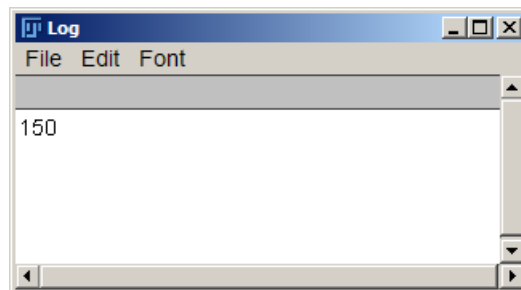We use delimiter \t, which means "tab", to convert single line to an array of data. each for a column.

Figure 2.34: Elementary data in the Log window

The value shown in the Log window (Fig. **??**) should be the same as X value in the first row of the original table (Fig. **??**).

We now know how to access individual data values within str, by first splitting it with delimiter \n and then by \t. We can print all XY coordinates in Log window by following code.

```
1 //code 25
2 str = getInfo("window.contents");
```

```
3 strA = split(str, "\n");
4 trackA = newArray(strA.length);
5 frameA = newArray(strA.length);
6 xA = newArray(strA.length);
7 yA = newArray(strA.length);
8 for (i = 0; i < strA.length; i++){
9   lineA = split(strA[i], "\t");
10   trackA[i] = lineA[1];
11   frameA[i] = lineA[2];
12   xA[i] = lineA[3];
13   yA[i] = lineA[4];
14 }
15 // checking
16 for (i = 0; i < trackA.length; i++)
17   print(xA[i] + ", " + yA[i]);
```

<div align="center">code/code25.ijm</div>

If you encounter error message with `lineA[]` such as shown in fig.**??**, this is just because there is another text widow opened and `getInfo()` function worked on that window rather than the results table of the Manual Tracking. To avoid such error, close extra text windows and then try running the macro again.
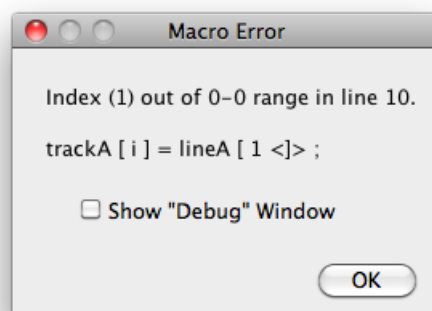

Figure 2.35: Possible Error Message with Code25 and Code25.5

From line 4 to 7, new arrays are generated to store data from four columns
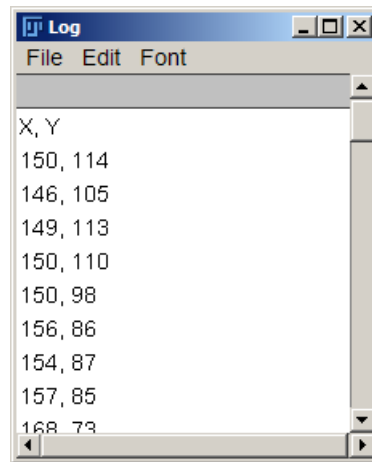
Figure 2.36: Elementary data in the Log window

in the for-loop from line 8 to 14.

Check the log window (Fig. **??**), compare the output with manual tracker results table, and if you are confident that you are accessing data in table, you could then use XY coordinates to place a circular ROI, measure average intensity in that area and list them in ImageJ Results window.
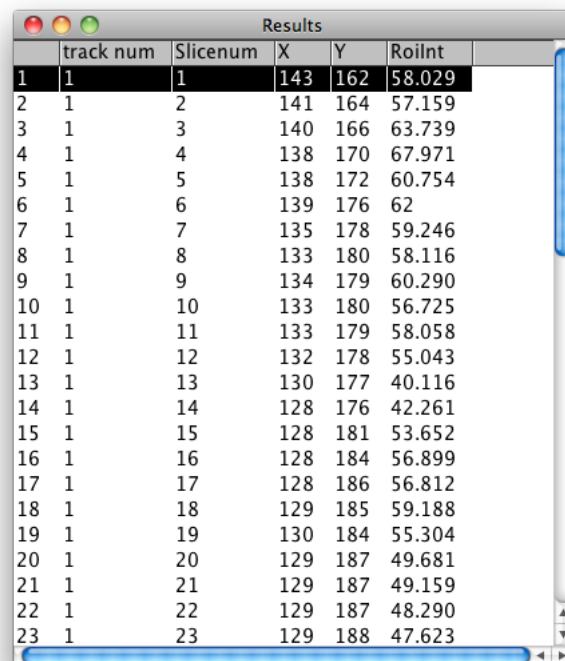
```
1  //code 25.5
2  stacktitle = "TransportOfEndosomalVirus.tif";
3  str = getInfo("window.contents");
4  strA = split(str, "\n");
5  trackA = newArray(strA.length);
6  frameA = newArray(strA.length);
7  xA = newArray(strA.length);
8  yA = newArray(strA.length);
9  for (i = 0; i < strA.length; i++){
10    lineA = split(strA[i], "\t");
11    trackA[i] = lineA[1];
12    frameA[i] = lineA[2];
13    xA[i] = lineA[3];
14    yA[i] = lineA[4];
15  }
16  diam = 9;
17  offset = floor(diam/2);
18  intA = newArray(xA.length);
19  for (i = 1; i< xA.length; i++){
```

```
20   selectWindow(stacktitle);
21   setSlice(frameA[i]);
22   makeOval(xA[i]-offset, yA[i]-offset, diam, diam);
23   getRawStatistics(nPixels, mean, min, max, std);
24   intA[i] = mean;
25   print(xA[i] + " , " + yA[i] + ":  mean int="+ intA[i]);
26   setResult("track num", i-1, trackA[i]);
27   setResult("Slicenum", i-1, frameA[i]);
28   setResult("X", i-1, xA[i]);
29   setResult("Y", i-1, yA[i]);
30   setResult("RoiInt", i-1, intA[i]);
31 }
```

<div align="center">code/code25_5.ijm</div>

Running this code, you should see Results window that looks like figure **??**, tracking data plus measured intensity is shown in column titled "RoiInt". The way oval ROI is used to measure mean intensity is similar to what we have coded in the previous subsection. A difference is that this time, we use arrays that store data extracted by splitting the chunk of string data.

We then successfully measured the intensity dynamics of moving object again.

	track num	Slicenum	X	Y	RoiInt
1	1	1	143	162	58.029
2	1	2	141	164	57.159
3	1	3	140	166	63.739
4	1	4	138	170	67.971
5	1	5	138	172	60.754
6	1	6	139	176	62
7	1	7	135	178	59.246
8	1	8	133	180	58.116
9	1	9	134	179	60.290
10	1	10	133	180	56.725
11	1	11	133	179	58.058
12	1	12	132	178	55.043
13	1	13	130	177	40.116
14	1	14	128	176	42.261
15	1	15	128	181	53.652
16	1	16	128	184	56.899
17	1	17	128	186	56.812
18	1	18	129	185	59.188
19	1	19	130	184	55.304
20	1	20	129	187	49.681
21	1	21	129	187	49.159
22	1	22	129	187	48.290
23	1	23	129	188	47.623

Figure 2.37: Manual Tracker Results in IJ Results table now with measured intensity column RoiInt

2.7.3 Accessing Data File: Simple Case

In this section and in the next section, we study how to access data in saved file for secondary measurements.

Two cases we studied so far were both accessing data listed in a table that is already loaded in ImageJ (data is already an instance of ImageJ). What if we want to use data that is saved as a file? For example, we want to use results of Manual Tracking (previous section) that was saved as an .xls file and you want to use its data for secondary measurement. More generally, you did particle tracking using different software such as Imaris Track, and you want to use coordinate data from that analysis for intensity measurement to be done in ImageJ. In such cases, we should access tabulated data in file accessing from ImageJ.

In this section, we try accessing data saved by Manual Tracking. Loading data file could be done by a small modification of the code we studied in the previous section. Instead of the function `getInfo("window.content")` in line 3 of code 25.5, we use `FileOpenAsString(path)` to retrieve file content into a string variable. By leaving the argument `path` blank (`""`), user will be asked for choosing a file. Since this is a simple one-line replacement of line 3 of code 25.5, below is the new code but only showing its first 10 lines.

```
1  //code 26
2  stacktitle = "TransportOfEndosomalVirus.tif";
3  str = File.openAsString("");
4  strA = split(str, "\n");
5  trackA = newArray(strA.length);
6  frameA = newArray(strA.length);
7  xA = newArray(strA.length);
8  yA = newArray(strA.length);
9  for (i = 0; i < strA.length; i++){
10   lineA = split(strA[i], "\t");
```

code/code26.ijm

To run this macro, be sure that you have your stack already opened, as it is required for measuring intensity.

2.7.4 Accessing Data File: Complex Case

We now try making use of data file with more complex format. In the case we studied with Manual Tracking plugin in the previous section, data format was pretty straight forward so we did not have to do much work for dealing with the data format. In general, things are not so simple and one must figure out some way to decode the format for using it in secondary measurement.

Automatic tracking plugin "ParticelTracker" allows you to save trajectory data by "Save Full Report" button in the results interface (see Fig. **??**). We try to access this data. [18]

First, you must prepare the data file.

Exercise 2.7.4-1

> Open sample image stack **TransportOfEndosomalVirus.tif**. Then do
> `[Plugins > Particle Detextor & Tracker> Particle Tracker]`.
> An interface appears, so fill in the parameters as follows:
>
> - radius: 3
> - cutoff: 0
> - percentile: 0.3
> - link range: 1
> - distance: 20
>
> Click "Save Full Report" and save the data file in the folder where sample image stack "**TransportOfEndosomalVirus.tif**" is. Saving dialog will come up with a proposal of file name to be "Traj_<filename>.txt" so do not change that and simply click Save (this file name will be important). Close the Results interface. Do not close the image stack **TransportOfEndosomalVirus.tif**, as we will use it still in the following.

---

[18]This technique is especially important if you want to do automated particle tracking of many data. A new feature in ParticleTracker plugin released in Nov. 2010 is that when ParticleTracker is called from macro, it automatically saves data in folder where the image file is located. We then are able to process many files using macro, even recursively, and get tracking data automatically. For this reason technique for accessing ParticleTracker data file is valuable.

Now you are ready with data, so we try loading data into ImageJ/Fiji. Run the following code.

```
1 //code27
2 path = getDirectory("image");
3 filename = getTitle();
4 txtfile = "Traj_" + substring(filename, 0, lengthOf(
      filename)-4) + ".txt";
5 fullpath = path+txtfile;
6 print(fullpath);
7
8 if (!File.exists(fullpath))
9   exit("data file not found");
10
11 str = File.openAsString(fullpath);
12 print(str);
```

<div align="center">code/code27_PTfileaccess.ijm</div>

You now have all the values in the log window, that should look like fig. **??**.
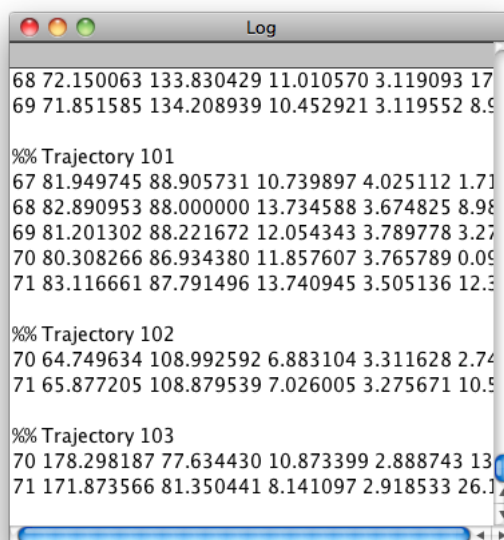
Figure 2.38: ParticleTracker data loaded to Log Window

Before getting into data file structure, let's look at what we have done in the code above. You might then study about how to deal with string, extracting parts of it.

- Line 2: `getDirectory` function with option `("image")` will return a path of the last-opened image location. This will be where the file "**TransportOfEndosomalVirus.tif**", inside the sample image folder.

- Line 3: Filename of the image "**TransportOfEndosomalVirus.tif**" is acquired.

- Line 4: Generating the file name of the data file. "Traj_" is the prefix that was automatically added to the data file name. Function `substring` extracts part of the string variable, and in our case, we try to get the image file name without ".tif". Definition of `substring` function is as follows.

    substring(string, index1, index2)
    Returns a new string that is a substring of string. The substring begins at index1 and extends to the character at index2 - 1. See also: indexOf, startsWith, endsWith, replace.

    index1 should be 0, as we want from the beginning of the file name, and index2 should be 4 strings before the last string so we need the total length of the string. For this we use `lengthOf` function.

    lengthOf(str)
    Returns the length of a string or array.

    In this way, we construct the file name we want to access (the name of which originally is automatically generated when saving the data in Particle Tracker interface) the file further setting the full path to the file in line 5.

- Line 8: This line checks if the file full path generated above is valid. `File.exists(full-path-to-file)` returns false if there is no such file. In that case, we should not proceed more so macro terminates at line 8.

98

- Line 11: If everything is ok, then the file is opened as a string, and the string will be printed in Log window by line 12.

ParticleTracker data file consists of three parts.

1. Header: contains information on the condition of tracking.

2. Detected Particles: Detected particles in each frame is listed.

3. Trajectories: Trajectories are listed, one by one.

Since we need to access trajectory information, we need to go through the data string to reach the third part of the data structure. To do so we split the file by lines (\n), then loop through the array to find the position where the trajectory information is contained.

We examine the data file in detail, to see what could be the marker for the starting and end point of each trajectory. Here is a part of data, that is a directly copy and pasted below.

```
24 154.894485 137.063614 7.338140 3.316167 15.683273
25 154.217377 138.368927 7.087145 3.417947 0.188002


%% Trajectory 37
15 22.111439 204.511826 9.726052 3.180977 28.252821
16 8.837964 209.618210 13.082743 3.273177 29.163294
17 0.432002 208.377045 3.574241 1.552869 0.000000
18 2.150804 209.609573 11.131773 2.939974 13.455443
19 10.542578 206.151169 14.173851 3.202391 14.258223
20 9.727753 206.960999 14.360144 3.250930 6.493425
21 15.708366 207.058640 14.943080 3.121640 0.053831
22 26.715679 208.680145 14.648912 3.091611 1.570746
23 29.143650 208.706314 13.616717 2.975144 21.279413
24 28.148304 202.200867 12.116886 2.862307 6.292521


%% Trajectory 38
15 142.326370 81.088326 5.813046 2.967546 13.995440
```

Single trajectory data start with a line with "%%Trajectory " plus numbering, and ends with a blank space. We use these information as markers

for determining the location of trajectory data within file. Each line of data consists of 6 numbers:

1. Frame number

2. Y coordinate

3. X coordinate

4. image moment m0

5. image moment m2

6. Non-Particle Discrimination Criteria

These values are separated by space. So splitting single line to an array of elementary data needs to be done by `split(line, " ")`.

Here is a macro, that uses `File.openAsString("")` to load the file and then loads the trajectory information into Results table with the strategy explained above.

```
1 //code 28
2 macro "Load Track File to Results (trackwise)"{
3   print("\\Clear");
4   run("Clear Results");
5   tempstr = File.openAsString("");
6   openedFile = File.name();
7   print(openedFile);
8   openedDirectory = File.directory;
9   Load2ResultsV3(openedDirectory, openedFile);
10 }
11
12 function Load2ResultsV3(openpath,openedFile) {
13   fullpathname = openpath + openedFile;
14   print(fullpathname);
15   tempstr = File.openAsString(fullpathname);
16   linesA = split(tempstr,"\n");
17   trajectoryCount = 1;
18   rowcounter = 0;
19   for (i = 0; i < linesA.length; i++) {
20     tempstr = linesA[i];
```

```
21    comparestr = "%% Trajectory " + trajectoryCount;
22    if (tempstr == comparestr) {
23      traj_startline = i;
24      do {
25        i++;
26        paramA = split(linesA[i], " ");
27        tempstr2="";
28        for (j = 0; j<paramA.length; j++) {
29          tempstr2 = tempstr2 + paramA[j] + "\t";
30        }
31        tempstr = "" + trajectoryCount + "\t" + tempstr2;
32        finalstr = CommaEliminator(tempstr);
33        linecontentA = split(finalstr, "\t");
34        if (linecontentA.length>1) {
35          setResult("TrackNo", rowcounter , linecontentA
                [0]);
36          setResult("Frame", rowcounter , linecontentA[1]);
37          setResult("x", rowcounter , parseFloat(
                linecontentA[3]));
38          setResult("y", rowcounter , parseFloat(
                linecontentA[2]));
39          rowcounter++;
40        }
41      } while (linesA[i]!= "")
42      trajectoryCount++;
43    }
44  }
45  updateResults();
46 }
47
48 function CommaEliminator(strval) {
49   while (indexOf(strval, ",")>0) {
50       delindex = indexOf(strval, ",");
51       returnstr = substring(strval, 0, delindex) +
              substring(strval, delindex+1, lengthOf(strval));
52       strval = returnstr ;
53   }
54   return strval;
55 }
```

code/code28.ijm

In the code 28, core of the processing resides within the function `Load2ResultsV3`. It takes path to the data file and file name as arguments, and first opens the file as string at line 15. The chunk of string is then split by lines and for-loop starts to go through the array of lines (line 19).

In every line in the array, if the line is a starting marker "%% Trajectory <number>" is tested (line 22) . If that is the case, then do-while loop is started, that loops until all the trajectory points are read out (line 24 to 41). While this trajectory read out is done, counter for the for-loop `i` is also incremented that when the do-while loop ends (line 25), the for-loop starts again from the line after the data of that trajectory. Exit from do-loop occurs when blank line is found (line 42).

Inside the do-while loop, space character is replaced with tab delimiter (line 26 to 32). This is required, since ImageJ results importing only recognized tab-delimited file as a table.

There is a small adjustment by another function `CommaEliminator` at line 32. This is required for removing comma character in some cases (this happens when the data file was opened and saved in excel or so). You probably do not need this line and function as we are using the data file directly after saving them. I left it in the code for your future usage.

Just to be sure with data content, line 33 and 34 double-check that the line indeed contains several data. Line 35 to 38 writes trajectory ID, Frame number and XY coordinates into Results window by `setResult` function.

Exercise 2.7.4-2

> Finally, we can combine several macros and functions we studied so far to make a new macro that loads the ParticleTracker data file to Results table, and read out intensity. This could be done by combining following codes, with a bit of modifications with each.
>
> - code 27 (check the current image stack name and loads its track data file as string)
> - code 28 (data in string format is converted to Result table)
> - code 24.5 (measure intensity according to coordinates in Results table)

Bits and pieces are already there. Please try completing a macro that loads track data file according to the title of the image stack that is already opened, place them in Results table, and then measure the intensity in corresponding frame and position in the image stack.

2.8   Using Javascript

As you become experienced with coding in ImageJ macros, you might start to find out that for whatever you want to do with ImageJ, corresponding macro function does not exist in the Build-in Macro Functions page [19]. One way to supplement the missing function is to create your own user function. Another way is to find a function directly from ImageJ Java code and use that function in macro. Javascript is a convenient way to access ImageJ API (Application Programming Interface), and since Javascript could be called from within ImageJ macro, you could use ImageJ API in your macro code. This is done by the macro function shown below:

eval("script", javascript)
Evaluates the JavaScript code contained in the string javascript.

This would be the simplest way to use Javascript if you are already comfortable with ImageJ macro language.

But there is more to it. You could also run Javascript as it is in ImageJ and Fiji. Syntax of Javascript is not same as ImageJ macro, but if you are used to write ImageJ macro, it should not be too difficult to learn Javascript.

. . . then how could we code Javascript?

In this section, we learn basic know-how of Javascript with ImageJ [20]. Experience with Java programming is largely helpful but if not, there is also some way around to learn quickly.

When we are programming ImageJ macro, we often refer to the web site listing ImageJ macro language functions to look for a macro function. In similar way, we access so called API (Application Programming Interface)

---

[19]see http://rsb.info.nih.gov/ij/developer/macro/functions.html

[20]If you want to learn in more detail, you could also visit http://pacific.mpi-cbg.de/wiki/index.php/Javascript_Scripting for learning more about Javascript.

for coding with Javascript. ImageJ API is in the following page:

http://rsb.info.nih.gov/ij/developer/api/index.html

At this moment, you might be puzzled with these pages, but don't worry. Major aim of this section is to learn how to use this resource to code your Javascript.

OK, let's start.

2.8.1   A trial with Javascript

Let us first try using Javascript (JS).

From menu, do `[Plugins > Scripting > Javascript Interpreter]`. You will then see a new interface that looks like Fig **??**. This interface provides Javascripting in an interactive mode and is useful for a quick testing of codes. There is a input field at the bottom, where you could type in JS code. Then by pressing return key, the code is executed.
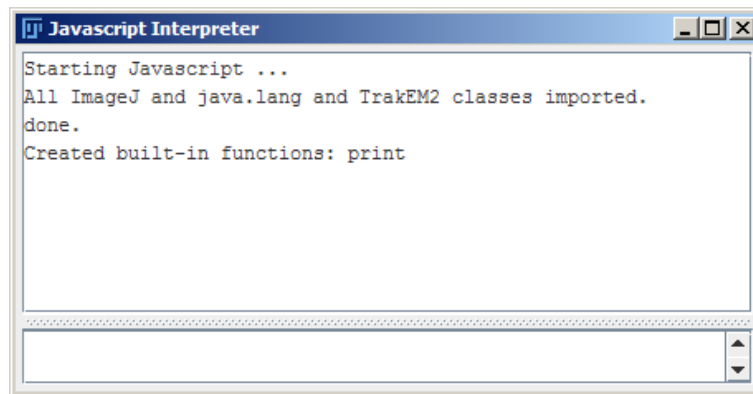

Figure 2.39: Javascript interpreter on start up.

Type the following command and execute by return key.

```
IJ.log("Hello JS")
```

This JS command will print "Hello JS" in the Log window of ImageJ.

Same command could be written in the Script editor. To start up the editor, do `[File > New > Script]` (then select `[Language > Javascript]`).

```
IJ.log("Hello JS");
```

This should do the same thing, but be careful! Do not forget adding semi-colon (;) at the end of the line. In case you write your code in Script Editor, you need to explicitly mark the end of line, just like you do when you write a macro.

To run the code, `[Run > Run]` will execute the command (you could also use ctrl-r or cmd-r).

What this JS code does is the same as the Macro code below.

```
print();
```

In the following, you could use either Javascript interpreter or Script editor. Just choose the one you like. If code become multiple lines, I recommend you to use the Script Editor… and in this case, this is a redundant warning, place a semi-colon ";" at the end of each line.

Now, we could try some commands that is not present in ImageJ macro.

For example, what would you do if you want to convert angle in degrees to radian? In macro, you could do calculation by first dividing the value by 360, then multiply by 2π. But a function actually is already there in Java, so you could simply use that as well.

```
IJ.log(java.lang.Math.toDegrees(3.1415))
```

Running this line should print a number close to 180. You could also do the other way around:

```
IJ.log(java.lang.Math.toRadians(180))
```

should print out 3.1415….

Next, we try to retrieve a column of data from table in Results window. In macro, you could do this by `getResult` function, with which by specifying the column label and row number you could retrieve a value in that cell.

But what should we do if we want to retrieve all data in a row at once, not a single value in a specific column at specific row? If you want to do this in macro, we could write a user defined function that loops for all the columns and get data one-by-one.

With Javascript, this could be done in just a single step, one command.

Exercise 2.8.1-1

    Preparation of Results table

    Open image by `[File > Open Sample > blobs (25k)]`.

    Check measurement parameters by

    `[Analyze > Set Measurements...]`

    that some measurement parameters are checked. Be sure that "Limit to Threshold" is checked.

    Then Threshold the blob image by

    `[Image > Adjust > Threshold]`.

    Since the background of this image is bright, Dark Background' should be unchecked. If you see the thresholded image like fig. **??**, do

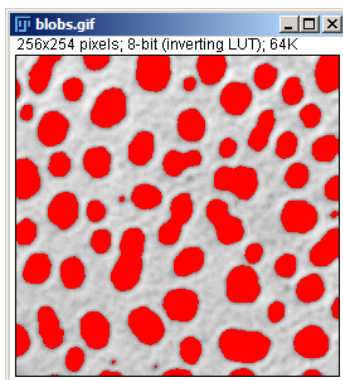   `[Analyze > Analyze Particles...]`.

Figure 2.40: Thresholded image of "blob"

In the Analyze Particles dialog, just be sure that **Display Results** is checked. Then click OK button. When the analysis is done, you will see 64 or so particles detected and listed in the Results window.

Testing Javascript Code

We now use the following command to extract data from single Row. In Javascript interpreter, type the following command .

```
IJ.log(ResultsTable.getResultsTable().getRowAsString
    (10));
```

If you execute this, you will see that all data that is in row 11 in Results table is now printed in the log window.

. . . that's the end of this exercise but don't close the Results window yet.

Above is a single line pure JS code. We can use this code within macro by using **eval** function mentioned already. Here is an exercise to test the function **eval**.

Exercise 2.8.1-2

Test running the following code, and check that any row in the table could be extracted and printed in Log window.

```
1 rownum = getNumber("Row?", 0);
2 jscom = "IJ.log(ResultsTable.getResultsTable().
      getRowAsString("+rownum+"));";
3 eval("script", jscom);
```

In the second line, JS code is constructed as a single string `jscom` using the variable `rownum` from line 1. Line 3 executes this JS code using **eval**.

So far, I have not yet explained where these commands came from. I will give more detailed explanation in later sections.

Exercise 2.8.1-2

This exercise is optional:

Try the following Javascript commands using **eval**, from within an ImageJ macro.

...lists all ImageIDs. There should be at least one image opened.

```
a = WindowManager.getIDList();
for(i in a) IJ.log(a[i]);
```

...zooms current image centered at top-left corner.

```
IJ.getImage().getCanvas().zoomIn(0, 0);
```

...print souts statistics of current image in log window.

```
eval("script", "IJ.log(IJ.getImage().getStatistics().
   toString());");
```

...print outs used memory in log window.

```
eval("script", "print(IJ.currentMemory())");
```

...moves current image window to top-left corner of the monitor with offset of 10 by to, and resizes the window.

```
eval("script", "IJ.log(IJ.getImage().getWindow().
   setLocationAndSize(10, 10, 100, 100))");
```

In all the example codes, we placed Javascript commands in the second argument for the function *eval*. You could also write a full path to a Javascript file. Here is the syntax.

eval('script', **File.openAsString**("<pathpath>/name.js")) ;

2.8.2 Using Macro Recorder and ImageJ API

Javascript is a scripting language, so it has its own build-in functions. I will not explain about this since you could find many Javascript tutorials on the web. For example, following site is a place where I go and look for Javascript commands and usages:

[Javascript Reference @ w3chools.com]

How do we find Javascript commands to interact and control ImageJ? The easiest way is to use the macro recorder. We have already learned and used macro recorder in previous chapters. We could used the same interface for recording JS codes. Recorded lines of JS codes could be copy & pasted into Script Editor and can be directly executed.

Exercise 2.8.2-1

First, we should start the recorder. `[Plugins > Macros > Record...]`. Then in the recorder window at the top-left corner, choose Javascript as the code to be recorded (Fig. **??**).

Figure 2.41: Setting Up Macro Recorder ready for Javascript

Then do the following sequence of commands.

```
[File > Open Sample > Blobs]
```

- Select rectangular ROI tool and set a ROI to select about 1/4th of the image (can be any place within the image. This is just a test.).

- `[File > Transform > Flip Horizontally]`

- `[Process > Filters > Gaussian Blurr...]`

After all these operations, there should be JS codes printed in the Recorder window. Copy them all, and paste it to Script Editor (`[File`

> New > Scripts] and then paste, Fig. **??**) . After pasting, set lan-
guage to JS by ([Language > Javascript] from the menu bar of
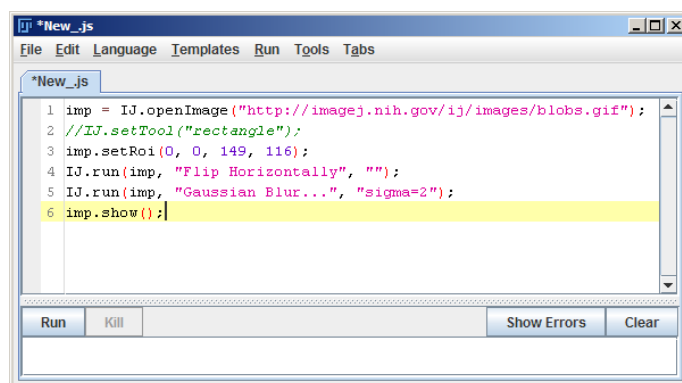script editor.

Figure 2.42: Javascript by recorder commands.

Then do [Run > Run] from the menu of script editor, the script is
executed. You should then see a new window of "blob" with some
part of image processed.

If you are successful in running the code, let's see the code. Here is how
the code should look like.

```
1 imp = IJ.openImage("http://imagej.nih.gov/ij/images/blobs.
    gif");
2 imp.setRoi(0, 0, 149, 116);
3 IJ.run(imp, "Flip Horizontally", "");
4 IJ.run(imp, "Gaussian Blur...", "sigma=2");
5 imp.show();
```

code/code29.js

We first examine line 3 and line 4, focusing on the method IJ.run (in the
follwoing, we use word "method" instead of "command", as this is more
conventional way of calling it in Java). This method has three arguments
for it.

```
IJ.run(argument1, argument2, argument3)
```

Just by looking at each of them you could realize that the second argument
is a descriptive explanation of what the method does. This is because these

strings are exactly the phrase of the menu item you see when you choose that function from ImageJ menu.

`IJ.run` is a method that uses second argument as a keyword to search for all the ImageJ menu items to find which of them is the one that the method intends to invoke [21]. Third argument of `IJ.run` in line 3 is an empty string, but in line 4, the third argument is `sigma=2`. This is a value that you normally input when you select Gaussian blur from the menu bar for the size of blurring kernel.

Then what is the first argument in `IJ.run`? In both line 3 and 4, we have a variable **imp**. To see what this is, we go back to line 1. **imp** appears for the first time in the code at line 1, and **imp** is the returned value of a method `IJ.openImage`. If we think back what we were actually doing for this first line when recording, we accessed an item in the menu tree `[File > Open Sample > blobs]`. By choosing this item from menu, ImageJ downloads blobs.gif file from NIH web site and then shows it on your desktop. Single method that does the download action is the method `IJ.openImage`. Argument for this command is the URL of the image.

To know the definition of the method `IJ.openImage`, we look up a reference called [ImageJ API](#) [22]. In this web page, there is side bar in left side, with upper part for a list of "All Packages" (These packages are same as those listed in the table shown in the top page) and the bottom part for "All Classes".

Each package contains several classes. We currently do not know which package does `IJ.openImage` belong to, so we look for it in the bottom part "All Classes". There, you will find "IJ"[23]. Click the link, and in the right side of the page, a page titled "Class IJ" appears (Fig. **??**).

The page might look cryptic to you, if you scroll down the page, there is a table titled "Method Summary", listing all the methods that class IJ contains in alphabetical order. Within this list, you will find (Fig. **??**)

---

[21]In ImageJ macro, a function similar to `IJ.run` method is `run(arg1, arg2)`.
[22]ImageJ API: http://rsb.info.nih.gov/ij/developer/api/index.html
[23]Unlike ImageJ macro, Java and Javascript are case sensitive
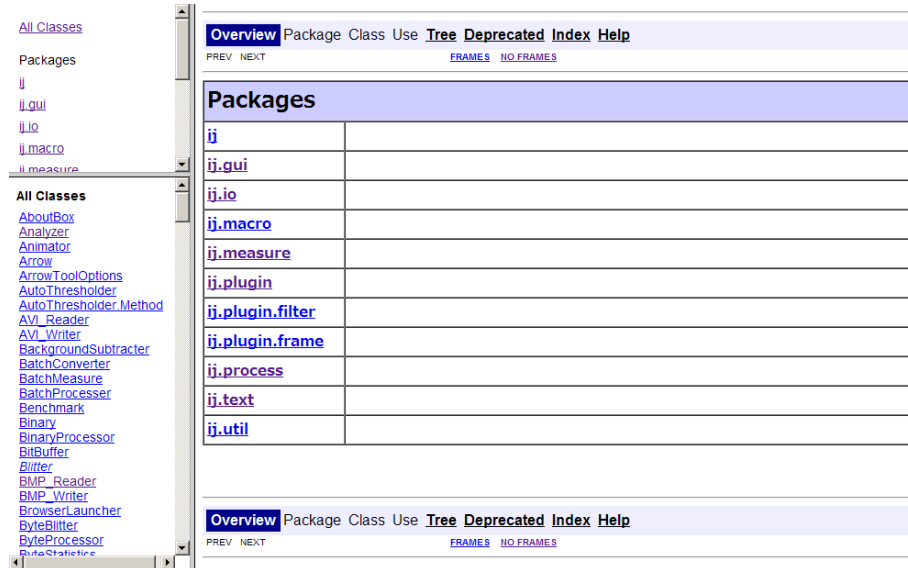
Figure 2.43: ImageJ API
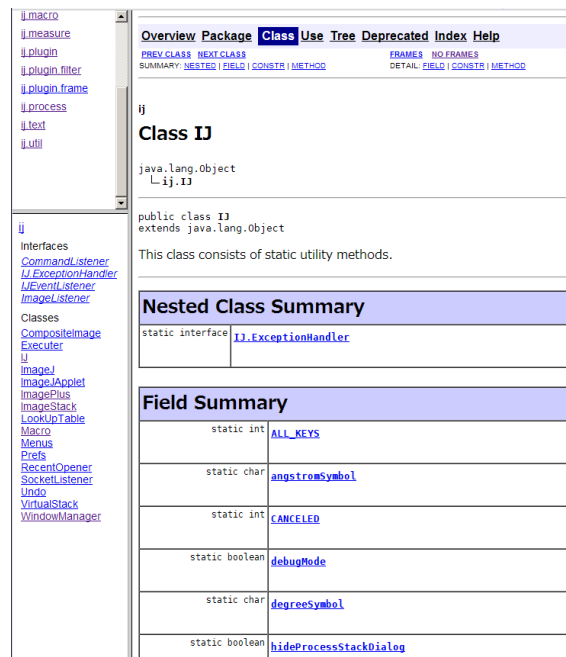
Figure 2.44: ImageJ API Class IJ

```
openImage(java.lang.String path)
```


Figure 2.45: ImageJ API Class IJ, openImage method

There are three `openImage` methods, with difference in number and types of arguments. The first one is without any argument, the second one has only one argument, the third having two arguments. When openImage method is called, one of these three are called depending on the number and types of the argument in the call.

More Explanation: So what are "Classes?" A class consists of two major components. One is Field and the other is Method. The former is like variable and the latter is similar to function in ImageJ macro (this is not a precise analogy, but for now let's think like that). Fields are values. Methods are actions. Class is a group of various field values and methods. By utilizing Class, we can have a single unit of assembled functions, which would be an advantage for letting an application to have high modularity. You could access (or use) fields and methods by appending them behind the name of the class. For example. <class name>.<field name> or <class name>.<method name>(<arguments>) e.g. `IJ.openImage(pathname)`.

A bit more on API page At the top of the Class IJ page (Fig.**??**), "ij" is written above the class name "Class IJ". This is the name of package that is

114

containing this class. You could see all classes within the package ij (note: case sensitive), click "ij" in the left top panel listing "all packages". You then will see all the classes within the package ij, such as compositeImage, Executor, IJ, ImageJ...and so on. Package is used to organize classes in hierarchical tree. For example, there are packages ij.plugin, ij.plugin.filter and ij.plugin.frame. Likewise, package is a tree-like structure (remember how folders are organized in your laptop) that organizes many classes in a structure. In Java, such tree-like structure is organized separated by dots ("."). For example, see Package ij.plugin.filter. The package filter is under the plugin package, that is under the ij package.

Go back to the API description, in the left side of the table column where

```
openImage(java.lang.String path)
```

is listed (Fig. **??**), there is a phrase

```
static ImagePlus
```

This tells you that if you invoke method `openImage(java.lang.String path)` of class `IJ`, a object called ImagePlus is returned (forget about the notion "static" for now). ImagePlus is another class, which you could also look up using the list of "All Classes".

The class ImagePlus consists of image data and metadata fields, and methods to access these values. In ImagePlus API page, you could see that many fields and methods are associated with this class. Note that many of field values are "protected".

For example, `nChannels` is one of the filed values of class ImagePlus but having a notion "protected" means that you cannot simply access from outside the class itself. Instead, there is a method named `getNChannels()` for accessing the value from outside which by invoking it "Returns the number of channels" of ImagePlus object. `getNChannels()` returns a value indicated as "int". This tells you that the type of returned object is "int", which means that returned value is a number, and specifically an integer, not a number with decimal points. Unlike ImageJ macro **object** with any type of class could be returned, not limited to number, string and array. This flexibility affords higher potential in modularity with Javascript compared to ImageJ macro and hence we call returned values as "object" not "variables".

A draw back is that Javascript coding becomes a bit more complicated then macro coding. One should always be careful about which class or type is returned, and one way to do so is to check ImageJ API every time you wonder what is the returned value of certain method.

Let's go back to the code again. The method `IJ.openImage` returns an **object "ImagePlus"**, so in line 1 of the recorded Javascript code, an instance of ImagePlus object (which actually is "blob.png" downloaded from the NIH website) is stored in the variable "imp". Then after line 1, `imp` behaves as an ImagePlus object. `imp` is repeatedly used from line 2 to 5. In line 2, an method of class ImagePlus is invoked in the form <class>.<method>(arguments). Since method name is `setROI` we look for it in the ImaegJ API ImagePlus page, and you will find the following description:

> void | setRoi(int x, int y, int width, int height)
> Creates a rectangular selection.

"void" means that this method does not return any value. There are four arguments and all of them are "int", integer. Description tells you that this methods creates an ROI in the image with position and size defined by arguments.

In line 3 and 4, the first argument of `run` method is `imp`, telling the command `IJ.run` to do the operation specified by the second argument on `imp`.

Note that in case of ImageJ macro, target image could only be specified by activating the window using `selectImage(imageID)`. In Javascript, selection of image could be more explicit by the direct use of ImagePlus object.

Down to line 4, blob image actually is not shown on the desktop. ImagePlus object of "blobs.tif" is in the memory, but still is not displayed. To show it on the desktop, we do line 5.

```
imp.show();
```

`show()` is a method of ImagePlus class to show the actual image of ImagePlus object.

Grabbing Image What if you want to capture already opened Image as a
ImagePlus object? This could be done by using a method in Class IJ called
`getImage()`. You could replace the first line in the code we just studied
with `IJ.getImage()` to grab currently active image rather than download-
ing and opening an image file.

Exercise 2.8.2-1

Modify code 29 so that this JS code grabs currently active image and
do the same processing.

Exercise 2.8.2-2

We study the nature of ImagePlus object in this exercise. We start
with a simple code to open an image, then add more lines to see how
ImagePlus instance behaves. Type in the code below to start up.

```
1 imp = IJ.openImage("http://imagej.nih.gov/ij/images/
     blobs.gif");
2 imp.show();
```

<div align="center">code/code30_1.js</div>

As we have done already, this will show a blobs.gif image on your
desktop. We can have another window with blobs.gif by adding two
more lines.

```
1 imp = IJ.openImage("http://imagej.nih.gov/ij/images/
     blobs.gif");
2 imp.show();
3 imp2 = IJ.openImage("http://imagej.nih.gov/ij/images/
     blobs.gif");
4 imp2.show();
```

<div align="center">code/code30_2.js</div>

We now have two instances of ImagePlus object. These are indepen-
dent. Whatever you do to imp, that does not affect imp2. We could
do a small trick that two windows could share the the same image,
that the same image appearing in two windows. Close two windows
of blobs, and then modify the code as shown below.

```
1 imp = IJ.openImage("http://imagej.nih.gov/ij/images/
     blobs.gif");
```

<div align="center">117</div>

```
2 imp.show();
3 ip = imp.getProcessor();
4 imp2 = new ImagePlus("SecondWindow with same IP", ip);
5 imp2.show();
```

<div align="center">code/code30_3.js</div>

Run this code, and you would see two windows with same image, which seemingly are same as before.

Take one image and try to select some region using a ROI tool, and from Fiji menu do *[Edit > delete]*. Then the selected region blacks out or whites out. Click another blobs window. Then you would see that the second window, which you did not process anything, also is processed.

This is because both windows are sharing the same image, shown in two windows. In other way of saying, there is a single instance of image with two ImagePlus instances. In the modified code, an extra line is added in line 3, and line 4 is changed. Line 3 is generating a pointer (ip) to the image that is contained in the ImagePlus instance *imp*. In the 4th line, a new instance of ImagePlus is created using what we call "constructor" (see ImagePlus API for a list of constructor), using the ip that is actually associated with the preexisting ImagePlus *imp*. Finally, the 5th line shows the second window *imp2*, the image content of which is shared with *imp*.

Summary

- Object should be either created (initialized, of "Constructed") or Grabbed.

  - `IJ.openImage(path)` creates and image object from file.
  - `IJ.getImage()` grabs already existing object.

- Object is constructed taking a class as template. A class has field values and methods. Hence, object has those values and classes.

  - Field values are in most cases accessed via methods.
  - Once an object is constructed, its public method could be used

- Exception: So called "static" methods could be accessed any time.

– Most of methods in Class IJ are static, so you do not need to construct it.

2.8.3 Example Codes

In this section, I will just show Javascript example using ImageJ API[24].

Curve Fitting example.

```javascript
1  //Curve fitting example
2  // see class CurveFitter
3  // http://rsb.info.nih.gov/ij/developer/api/ij/measure/
       CurveFitter.html
4
5  //creat example data arrays
6  var xa = [1, 2, 3, 4];
7  var ya = [3, 3.5, 4, 4.5];
8
9  //construct a CurveFitter instance
10 cf = CurveFitter(xa, ya);
11
12 //actual fitting
13 //fit models: see http://rsb.info.nih.gov/ij/developer/api/
       constant-values.html#ij.measure.CurveFitter.
       STRAIGHT_LINE
14 cf.doFit(0);
15
16 //print out fitted parameters.
17 IJ.log(cf.getParams()[0]+ " : " + cf.getParams()[1]);
```

code/codeCurveFitting.js

2.8.4 Using none-ImageJ libraries in Fiji

Besides access to ImageJ API, power of Javascript is in importing external packages written in Java to use their functions.

---

[24]Javascript cookbook is also available in the CMCI website for various coding examples. Visit http://cmci.embl.de/documents/110822jsip_cooking/javascript_imagej_cookbook

In Fiji, many java libraries (packages) are included besides ImageJ itself. You could see them listed in Fiji API. Here are some picks among them, which might be interesting to use for math and statistics in Javascript[25]. In the next section, we will study several examples to know how to import these libraries in Javascript.

Java Matrix Package (JAMA)

JAMA is comprised of six Java classes: Matrix, CholeskyDecomposition, LUDecomposition, QRDecomposition, SingularValueDecomposition and EigenvalueDecomposition.

The Matrix class provides the fundamental operations of numerical linear algebra. Various constructors create Matrices from two dimensional arrays of double precision floating point numbers. Various gets and sets provide access to submatrices and matrix elements. The basic arithmetic operations include matrix addition and multiplication, matrix norms and selected element-by-element array operations. A convenient matrix print method is also included.

Five fundamental matrix decompositions, which consist of pairs or triples of matrices, permutation vectors, and the like, produce results in five decomposition classes. These decompositions are accessed by the Matrix class to compute solutions of simultaneous linear equations, determinants, inverses and other matrix functions. The five decompositions are

- Cholesky Decomposition of symmetric, positive definite matrices

- LU Decomposition (Gaussian elimination) of rectangular matrices

- QR Decomposition of rectangular matrices

- Eigenvalue Decomposition of both symmetric and non-symmetric square matrices

---

[25]If you want to use these packages in ImageJ, you could download the package from its website and configure ImageJ to include that package on start up.

- Singular Value Decomposition of rectangular matrices

Apache Commons Math Package

Commons Math is divided into fourteen subpackages, based on functionality provided.

- org.apache.commons.math.stat - statistics, statistical tests
- org.apache.commons.math.analysis - root finding, integration, interpolation, polynomials
- org.apache.commons.math.random - random numbers, strings and data generation
- org.apache.commons.math.special - special functions (Gamma, Beta)
- org.apache.commons.math.linear - matrices, solving linear systems
- org.apache.commons.math.util - common math/stat functions extending java.lang.Math
- org.apache.commons.math.complex - complex numbers
- org.apache.commons.math.distribution - probability distributions
- org.apache.commons.math.fraction - rational numbers
- org.apache.commons.math.transform - transform methods (Fast Fourier)
- org.apache.commons.math.geometry - 3D geometry (vectors and rotations)
- org.apache.commons.math.optimization - function maximization or minimization
- org.apache.commons.math.ode - Ordinary Differential Equations integration
- org.apache.commons.math.genetics - Genetic Algorithms

Batic SVG Tool kit

Batik is a Java-based toolkit for applications or applets that want to use images in the Scalable Vector Graphics (SVG) format for various purposes, such as display, generation or manipulation.

The project's ambition is to give developers a set of core modules that can be used together or individually to support specific SVG solutions. Examples of modules are the SVG Parser, the SVG Generator and the SVG DOM. Another ambition for the Batik project is to make it highly extensibleÃćâĆňâĂİfor example, Batik allows the developer to handle custom SVG elements. Even though the goal of the project is to provide a set of core modules, one of the deliverables is a full fledged SVG browser implementation which validates the various modules and their inter-operability.

Mantissa (Mathematical Algorithms for Numerical Tasks In Space System Applications)

Mantissa is a collection of various mathematical tools aimed towards for simulation. It is not a complete mathematical library like GSL, NAG or IMSL, but it contains various algorithms useful for dynamics simulation and 3D geometry computation.

Weka

Weka is a collection of machine learning algorithms for data mining tasks. The algorithms can either be applied directly to

> a dataset or called from your own Java code. Weka contains
> tools for data pre-processing, classification, regression, cluster-
> ing, association rules, and visualization. It is also well-suited
> for developing new machine learning schemes.

Some other packages...

- ImageScience - Implemented as plugins for filtering but valuable for use as a library.

- ImageJ3D - JRenderer3D -3D renderer.

- kfschmidt.* - Numerical analysis tools: Simplex, GLM analyzer, Matrix calculations

- math3D -Matrix calculation, 3D tools.

2.8.5   Example Use of Library

Following will be some examples of using Apache Commons Math library.

Descriptive statistics

```
1 importPackage(Packages.org.apache.commons.math.stat.
     descriptive);
2
3 //prparation of data
4 stats = new DescriptiveStatistics();
5 var exA = [0, 1, 2, 3, 8, 9, 10];
6 for (i in exA) stats.addValue(exA[i]);
7
8 // Compute some statistics
9 mean = stats.getMean();
10 std = stats.getStandardDeviation();
11 npnts = stats.getN();
```

```
12 IJ.log("mean: "+ mean + "\nsd: " + std + "\npnts:" + npnts)
   ;
```

<div align="center">code/code32.js</div>

When you use a package, you should first import it. The first line is doing this by using a method `importPackage`. One could also import a single class, using `importClass` method. In line 4, a new object of Class DescriptiveStatistics is created. Line 5 and 6 stores data in this object, and calculates statistics in line 9, 10 and 11.

Solving Linear System

We could solve linear equation system below, in the form AX = B, by LU decomposition.

$$2x + 3y - 2z = 1$$

$$-x + 7y + 6x = -2$$

$$4x - 3y - 5z = 1$$

```
1 importPackage(Packages.org.apache.commons.math.linear);
2 //preparing matrix and data
3 matA = [[ 2, 3, -2 ], [ -1, 7, 6 ], [ 4, -3, -5 ]];
4 vecB = [ 1, -2, 1 ]
5 //LU decomposition
6 coefficients = new Array2DRowRealMatrix(matA, false);
7 solver = new LUDecompositionImpl(coefficients).getSolver();
8 ans = solver.solve(vecB);
9 for (i in ans) IJ.log(ans[i]);
```

<div align="center">code/code33.js</div>

2.9   Actual Macro programming

The biggest tip for Macro coding: Don't try to code everything from scratch. Refer to the downloadable macros linked in the ImageJ web site [26], and there should be something you could copy some parts to full fill the task you want to achieve.

Exercise 2.9.0-1

Think about your daily work with image processing / analysis, and design a macro that helps your task.

1. Present your idea. Similar macro might already exists, which could be modified for your task.

2. Write the macro after discussion with your instructor.

3. Debug the macro. If you could not finish, do it as homework. Turn it in, regardless of whether its working or not.

---

[26]see http://rsb.info.nih.gov/ij/macros/

2.10 Homework!

2.10.1 Homework for basics

Assignment 1

Change code 12.75 so that

- Does not use "&&"(AND)

- Instead, uses "||" (OR).

Comment: This is also a test if you can think things logically... Thanks to Prof. Boole.

Assignment 2

Write a macro that draws gird (lattice) in a image (see example, attached). If you have time, modify the macro so that the macro plots diagonal lattice. Steps should be something like:

1. creat a new image

2. loop in x direction and draw vertical line ... for this, use command `drawLine(x1, y1, x2, y2)` ... see http://rsb.info.nih.gov/ij/developer/macro/functions.html#drawLine

3. loop in y direction and draw horizontal line

Hints: if you want to draw white lines on black image

- you need to select black background when you make a new image

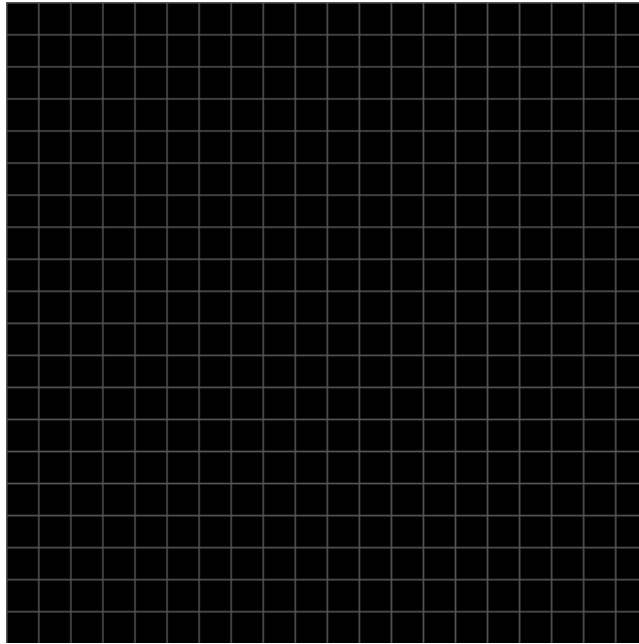- you need to set the drawing color using `setColor()`

- see http://rsb.info.nih.gov/ij/developer/macro/functions.html#setColor
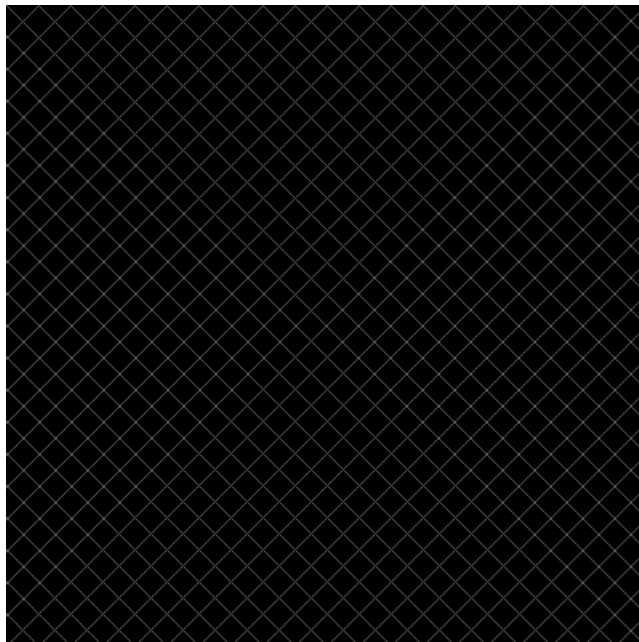
Figure 2.46: Composing grid image

Figure 2.47: Composing grid image

Assignment 3

Write a macro that deletes every second frame (even-numbered frames) in a stack.

Hint: use `run("Delete Slice");` to delete a single slice.

Comment: it might be tricky.

Assignment 4

Write a time stamping macro for t-stacks. You should implement following functions.

- User inputs the time resolution of the recording (how many seconds per frame).

- The time point of each frame appears at the top-left corner of each frame.

- If possible, time should be in the following format:
  mm:ss
  (two-digits minutes and two digits seconds)

Hint: Use following: for-statement, `nSlices`, `setSlice`, `getNumber`, `setForegroundcolor`, `setBackgroundColor`, `drawString`, `IJ.pad`. (refer to the Build-in Macro Function page in ImageJ web site!)

Assignment 5

Modify code 14 so that the macro does not use "while" loop. For example with the following way.

- Macro measures the integrated density of all area in the first frame (= ref_int).

- In the next frame, full integrated intensity is measured again (temp_int).

- Decrease the lower for the thresholding by temp_int/ref_int.

2.10.2 Homework for a bit advanced

Assignment 6

Write an elementary calculator macro with single dialog box that does:

- user input two numbers

- user selects either one of addition, subtraction, multiplication or division.

- answer appears in the Log window.

Hint: use `Dialog.addChoice Dialog.getChoice` command.

Assignment 7

Write a macro that does pseudo high-pass filtering by Gaussian blurred image (duplicate an image, do Gaussian blurring with a large kernel to create background and subtract it from the original). If you could successfully write a macro, then convert it to a function and use it from a macro. Hint: use `getImageID(), selectImage(id)` command.