

Getting Started Guide for scripting

```
/* This code is executed when clicking 'Run' */
MakeProtocol();
CreateDesignMatrix();
CreateMultiStudyDesignMatrix();

/* These functions can be invoked */
function MakeProtocol()
{
    BrainVoyagerQX.PrintToLog("Create stimulation protocol...");
    var doc = BrainVoyagerQX.OpenDocument(ObjectsRawDataPath + "CC_OBJECTS_FROMSCRIPT.ph7");
    doc.ClearSimulationProtocol();

    doc.StimulationProtocol.ExperimentName = "01";
    doc.StimulationProtocol.Resolution = 1;

    doc.AddCondition("T1wation");
    doc.AddCondition("Objects in LVF");
    doc.AddCondition("Objects in RVF");
    doc.AddCondition("Objects in BVP");

    forInterval = 0; interval
    {
        intervalstart = int
        intervalend = int

        // first interval
        if(interval == 0)
        {
            interval
        }
        doc.Addi
    }

    forInterval =
    {
        intervalst
        intervale
    }
    doc.Addi
    }

    forInterval = 1
    {
        intervalstart
        intervalend
    }
    doc.AddInterval
    }

    forInterval = 0; interval
    {
        intervalstart = 62 + inter
        intervalend = intervalstart
    }
    doc.AddInterval("Objects in BVP");
    }

    doc.SetConditionColor("T1wation", 100, 100, 100);
    doc.SetConditionColor("Objects in LVF", 255, 0, 0);
    doc.SetConditionColor("Objects in RVF", 0, 255, 0);
    doc.SetConditionColor("Objects in BVP", 0, 0, 255);

    doc.StimulationProtocol.BackgroundColorR = 0;
    doc.StimulationProtocol.BackgroundColorG = 0;
    doc.StimulationProtocol.BackgroundColorB = 0;

    doc.StimulationProtocol.TimeCourseColorR = 255;
    doc.StimulationProtocol.TimeCourseColorG = 255;
    doc.StimulationProtocol.TimeCourseColorB = 255;
    doc.StimulationProtocol.TimeCourseThickness = 4;

    doc.SaveSimulationProtocol(ObjectsRawDataPath + "CC_OBJECTS_FROMSCRIPT.ph7");
}
```

in Brain Voyager 21.4

Hester Breman, Joost Mulders and Rainer Goebel

Copyright 2019 © Brain Innovation B.V.

Contents

1	Familiarizing with the scripting environment	3
1.1	Open the BrainVoyager Script Editor	3
1.2	Executing code	5
1.2.1	Run a script	5
1.2.2	Run a command	6
1.3	Creating a new script	7
1.4	Appendix: functions and scripts	8
2	The first script	9
2.1	Script to open an FMR project	9
2.2	Appendix: objects	11
2.2.1	Actions: member-specific functions	11
2.2.2	Properties: member-specific properties	12
3	Using filenames	13
3.1	Write the script	13
3.2	Run the script	14
3.3	Script	14
3.4	Appendix: variables	15
3.4.1	Variables and values of variables	15
3.4.2	Variables and objects	16
4	Following the processing in Brain Voyager	17
4.1	Print text literally	17
4.1.1	Script	17
4.2	Print text with filename	17
4.2.1	Script	18
4.3	Print text with parameter value	19
4.3.1	Script	19
4.4	Save log text in text file	20
4.4.1	Script	21
4.5	Appendix: arguments and return values	22
5	Preprocessing several files	24
5.1	Adding the array	24
5.2	Adding the loop	24
5.3	Scripts	25
5.3.1	Array and loop script 1	25
5.3.2	Array and loop script 2	26
5.3.3	Array and loop script 3	27
5.3.4	Array manipulation script	28
5.4	Appendix: arrays and loops	29
5.4.1	Arrays	29
5.4.2	Loops	32

6	Creating a function within the script: read a text file	33
6.1	Writing a function	33
6.1.1	Setup of the function	33
6.1.2	Communicating with the function	33
6.1.3	Working with the text file	34
6.1.4	Script	35
6.2	Adding the function to our script	36
6.2.1	Script	36
7	Scripting the Getting Started Guide	37
7.1	Introduction of the script(s)	37

Introduction

Welcome to the Getting Started Guide for Scripting in BrainVoyager! Scripting in BrainVoyager has many advantages. The most obvious advantage is saving time by running processes overnight via scripts. It is easy to learn how to perform this via scripting. After practising only the first 6 steps of the Getting Scripted Guide it is already possible to achieve this.

Another advantage is that the scripts clearly document which operations were applied to the data with which parameters.

Note: Since BrainVoyager 20 the project creation, preprocessing and statistics in volumetric space can be processed via the Data Analysis Manager, which does not require scripting.

The Getting Started Guide for Scripting is created to gain familiarity with scripting in BrainVoyager in a practical and goal-directed way. An attempt is made to make scripting in BrainVoyager accessible also for users with little prior knowledge about scripting by the appendices which contain explanations about programming concepts (like loops and variables). In case the scripting concepts are known already, the appendices can be skipped.

Data

In the following examples we will be using three functional data documents (*.fmr) that have been created of the BrainVoyager sample data, but it should not be too difficult to apply the principles to your own data. Please note that in the Guide, both concepts 'project' and 'document' refer to FMR, VMR, DMR or AMR data in BrainVoyager. Since 2019, chapter 7 of this guide has been updated to process the data of the new Getting Started Guide for BrainVoyager 21.4 according to the steps in that Guide; this means that the data are transformed to MNI space instead of Talairach space.

Other sources of information about scripting in BrainVoyager

The BrainVoyager Scripting Reference that contains all BrainVoyager scripting functions can be downloaded from the BrainVoyager support page:

[BrainVoyager Scripting Reference Manual on support page.](#)

Example scripts can be obtained from the '[Scripts](#)' page on the same support website.

In the BrainVoyager Developer's Guide is described how to create GUI scripts, which can be used together with plugins but one can also create dialogs for scripts only. The Developer's Guide can be found in the BrainVoyager directory → UsersGuide. Example GUI scripts are available on the above mentioned 'Scripts' website.

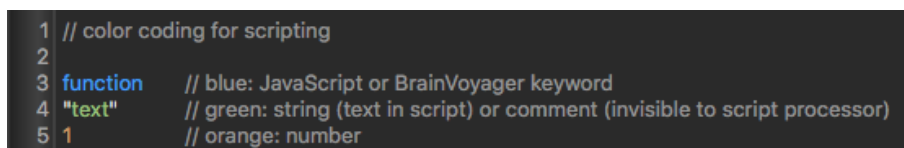
*Hester Breman, Joost Mulders and Rainer Goebel
With thanks to Quentin Noirhomme
Latest update: 19-09-2019*

Chapter 1

Familiarizing with the scripting environment

1.1 Open the BrainVoyager Script Editor

The Script Editor can be used to create or edit a BrainVoyager script, and to run a script. In fact, it is also possible to edit a script in a text editor, since it is just a text file.



```
1 // color coding for scripting
2
3 function // blue: JavaScript or BrainVoyager keyword
4 "text" // green: string (text in script) or comment (invisible to script processor)
5 1 // orange: number
```

Figure 1.1: Autocoloring and line numbers

However, the BrainVoyager Script Editor contains some advantages, since it recognises the script language and the BrainVoyager commands, which is indicated by giving the words different colors (see table 1.1 and figure 1.1).

color	meaning
blue	JavaScript or BrainVoyager keyword
green	string (text in script) or comment (invisible to script processor)
orange	number

Table 1.1: Autocoloring of JavaScript language and BrainVoyager API keywords

Also, the Script Editor has line numbers (see figure 1.1), so that it is easier to find a bug in the script (also the Debugger in the Script Editor might help to find bugs).

The Script Editor can be found via the option “Edit and Run Scripts” of the “Scripts” menu option in BrainVoyager (see figure 1.2).

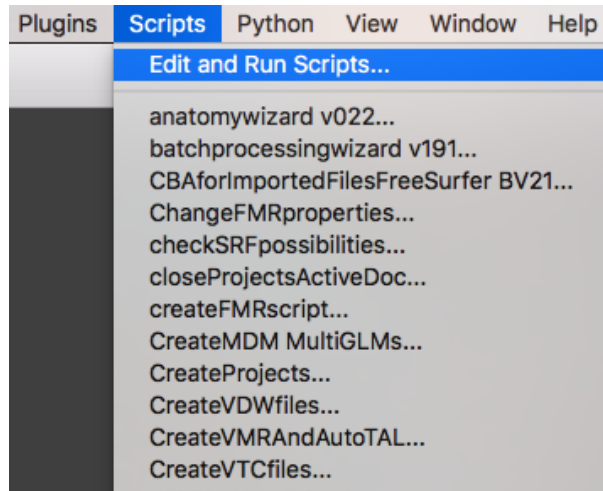


Figure 1.2: Starting the Script Editor

A separate window will open, with an empty script called “untitled.js”. When starting a new script, this can be used to add commands. Then it can be saved via the “Save” or “Save As...” buttons.

1.2 Executing code

1.2.1 Run a script

There are two ways to run a script.

The first is to run a script directly from the BrainVoyager "Scripts" menu. This does not require an opening of the Script Editor, since the script names in the /Documents/BVExtensions/Scripts/ directory are visible in the "Scripts" menu.

The second way to run a script is to open the Script Editor, select the script by doubleclicking the sidebar of the Script Editor and pressing the "Run" button (figure 1.3).

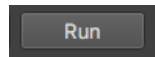
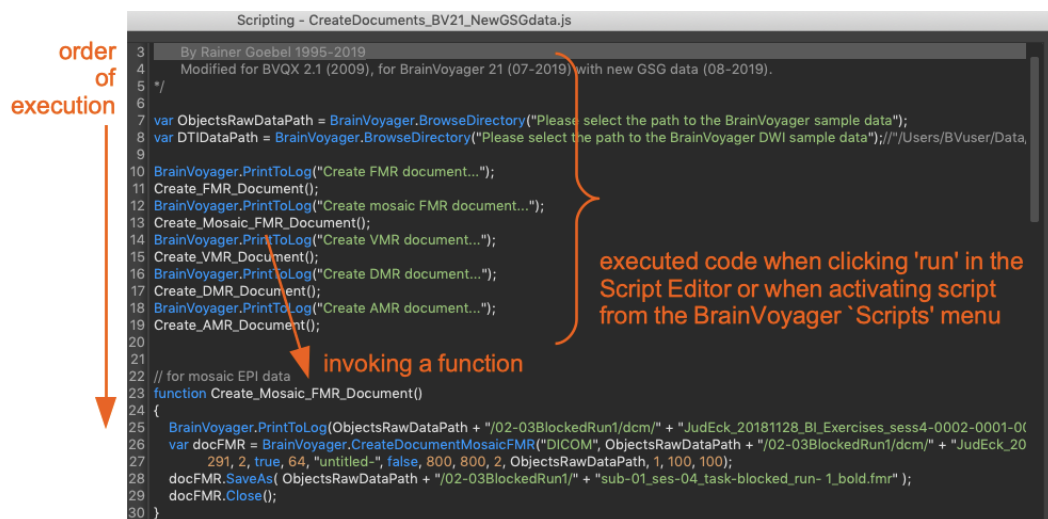


Figure 1.3: The "Run" button of the Script Editor

All code that is not within a function (see first red bracket in figure 1.4), is executed directly and sequentially from top to bottom. In this case, the executed code consists of two variable declarations and the use of two functions, `LoadVMRandsRF()` and `SurfaceViewpoints()`.

Code within a function block is only executed when the function is called (see second red bracket in figure 1.4).



```
Scripting - CreateDocuments_BV21_NewGSGdata.js
3  By Rainer Goebel 1995-2019
4  Modified for BVQX 2.1 (2009), for BrainVoyager 21 (07-2019) with new GSG data (08-2019).
5  */
6
7  var ObjectsRawDataPath = BrainVoyager.BrowseDirectory("Please select the path to the BrainVoyager sample data");
8  var DTIDataPath = BrainVoyager.BrowseDirectory("Please select the path to the BrainVoyager DWI sample data");//"/Users/BVuser/Data,
9
10 BrainVoyager.PrintToLog("Create FMR document...");
11 Create_FMR_Document();
12 BrainVoyager.PrintToLog("Create mosaic FMR document...");
13 Create_Mosaic_FMR_Document();
14 BrainVoyager.PrintToLog("Create VMR document...");
15 Create_VMR_Document();
16 BrainVoyager.PrintToLog("Create DMR document...");
17 Create_DMR_Document();
18 BrainVoyager.PrintToLog("Create AMR document...");
19 Create_AMR_Document();
20
21
22 // for mosaic EPI data
23 function Create_Mosaic_FMR_Document()
24 {
25   BrainVoyager.PrintToLog(ObjectsRawDataPath + "/02-03BlockedRun1/dcm/" + "JudEck_20181128_BI_Exercises_sess4-0002-0001-01
26   var docFMR = BrainVoyager.CreateDocumentMosaicFMR("DICOM", ObjectsRawDataPath + "/02-03BlockedRun1/dcm/" + "JudEck_20
27   291, 2, true, 64, "untitled-", false, 800, 800, 2, ObjectsRawDataPath, 1, 100, 100);
28   docFMR.SaveAs( ObjectsRawDataPath + "/02-03BlockedRun1/" + "sub-01_ses-04_task-blocked_run-1_bold.fmr" );
29   docFMR.Close();
30 }
```

Figure 1.4: Scripts and functions in the script editor

1.2.2 Run a command

There is also a third way to run some code, which is via the direct evaluation. The advantage is that it is very easy to check the effect of a function. The direct evaluation will accept one command at a time, and has no memory.

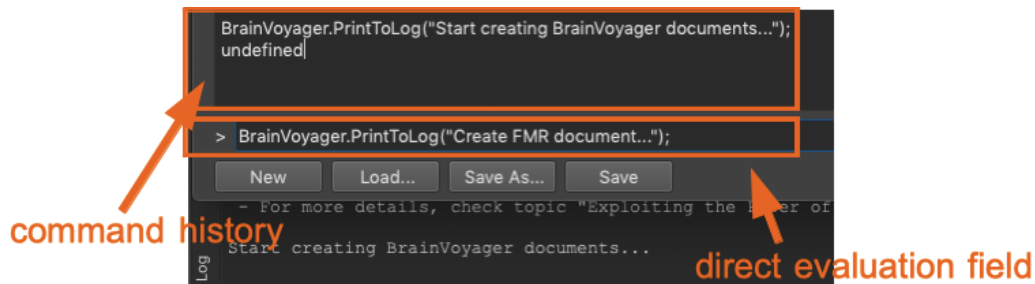


Figure 1.5: The direct evaluation field of the Script Editor and result on BrainVoyager Log tab

Just type the command in the text edit field and then press “Enter” to start the evaluation (see figure 1.5). the command we have used is displayed in the command window (front) and the information has been printed to the BrainVoyager Log tab (back). In this case, we use the command `BrainVoyager.PrintToLog()` to display the text that the project creation has started. Of course we could execute any command, as long as it is only one line. The command and the result are saved in the command history field. If there is no return value of the command, the command history displays `undefined`.

1.3 Creating a new script

The commands that can be send to BrainVoyager are collected in scripts (*.js). One script can be opened in the BrainVoyager Script Editor at the time. All scripts in the /Documents/BVExtensions/Scripts/ directory are visible in the sidebar of the Script Editor. Open the scripting editor via the “Edit and Run Scripts...” option of the ‘Scripts’ menu option in the BrainVoyager menu (see figure 1.2).

By default, an empty script is available, so one can start typing directly. Otherwise, click the “New” button (figure 1.6) to create a new script with the name “untitled.js”.

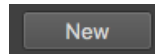


Figure 1.6: Creating a new script

Save the script with the name ‘EasyScript.js’ in the default directory for scripts and script projects, which is /(My) Documents/BVExtensions/Scripts/ (see figure 1.7).

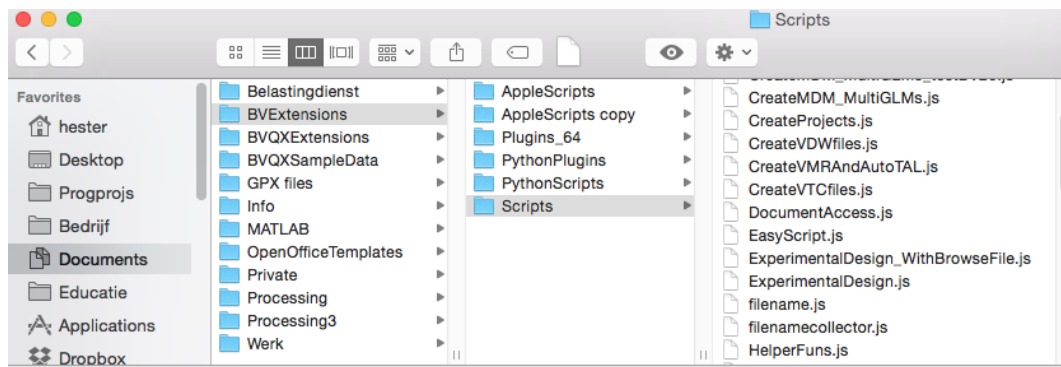


Figure 1.7: Location of BrainVoyager scripts

1.4 Appendix: functions and scripts

Functions are a set of instructions to the computer that belong together. All instructions that should be executed sequentially can be stored in a script (see figure 1.4). In BrainVoyager, the language for the scripts is JavaScript (there is also an option for Python scripting in BrainVoyager). A difference between computer scripts and programs is that scripts are converted to machine code when they start running, while a program has to be compiled (= converted to machine code) before it can be used.

It is possible to store several functions in the same script. The script is just a text file with the extension *.js. In the BrainVoyager script editor, scripts appear in the side bar (see figure 1.8). The name of the script that is currently opened is shown in the title of the Script Editor. Double-click on the top of the side bar to undock (on Mac OS X) and click the X to make the file list disappear.

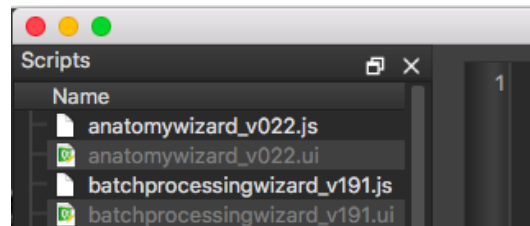


Figure 1.8: Scripts in the side bar of the Script Editor

The script will only run once, beginning with the commands at the top of the script and running each command line until the bottom of the script has been reached, while a function in the script can be invoked from within the script as often as one likes.

Chapter 2

The first script

2.1 Script to open an FMR project

When a script (*.js) is saved, it is shown in the scripts bar on the left side of the script editor. Also, the scripts are shown in the BrainVoyager 'Scripts' menu.

One script (*.js) can contain several functions. A function is a collection of commands that the script editor can execute. It can be recognized via the keyword `function` and the parentheses after the name of the function: `()`. The begin and end of a function is indicated via the brackets `{` and `}`.

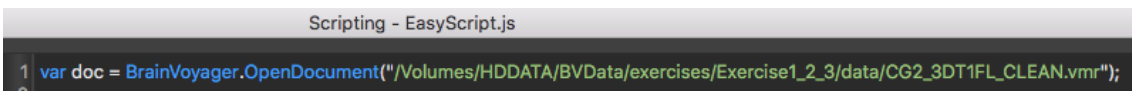
It is possible to create a script without creating new functions. BrainVoyager will just execute the script line by line from top to bottom.

So in this first script, we will not create a function, but only use an existing command. It is a BrainVoyager scripting function called `OpenDocument()`. The `Document` in `OpenDocument()` refers to a BrainVoyager project. With the command `OpenDocument()` we can open VMR, AMR, DMR and FMR documents. In this case we open an anatomical project (*.vmr).

In the script editor space, type: `BrainVoyager.OpenDocument();`

To open the document, we need to provide the function with text: the path to the file and the name of the file as argument (via the parentheses). Text is always written within single `'...'` or double quotes `"..."` and will appear green in the script editor.

So we just write the filename including the path in the parentheses. The filename is called an *argument* to the function `OpenDocument()`. In programming, the direction is just opposite to reading. So we start on the right, with the `BrainVoyager` object, which performs some action (opening a document). The result value - the VMR document - appears on the left of the `'='` sign (Figure 2.1).



```
Scripting - EasyScript.js
1 var doc = BrainVoyager.OpenDocument('/Volumes/HDDATA/BVData/exercises/Exercise1_2_3/data/CG2_3DT1FL_CLEAN.vmr');
```

Figure 2.1: EasyScript.js: script consisting of one instruction

Now save the script with the name “EasyScript.js” using the “Save As...” button (see figure 2.2).

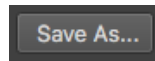


Figure 2.2: Save EasyScript.js via the Save As... button

If the file exists in the directory that was provided with the filename, run the script via the “Run” button (see figure 1.3) or directly from the “Scripts” menu in BrainVoyager (see figure 2.3).

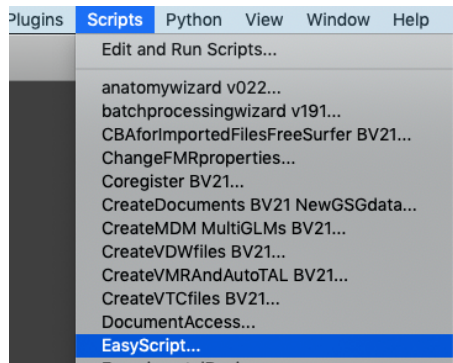


Figure 2.3: Start EasyScript.js from the Scripts ... menu

The VMR document will now be opened by BrainVoyager.

2.2 Appendix: objects

The above mentioned example shows that BrainVoyager, the application object, is actually opening the document. The action, opening the document, follows after the dot ('.').

In object-oriented programming, the program is concentrated around objects that do things. Objects can perform *actions* and they can have *properties*. The collection of actions that an object can perform is called 'member-specific functions'.

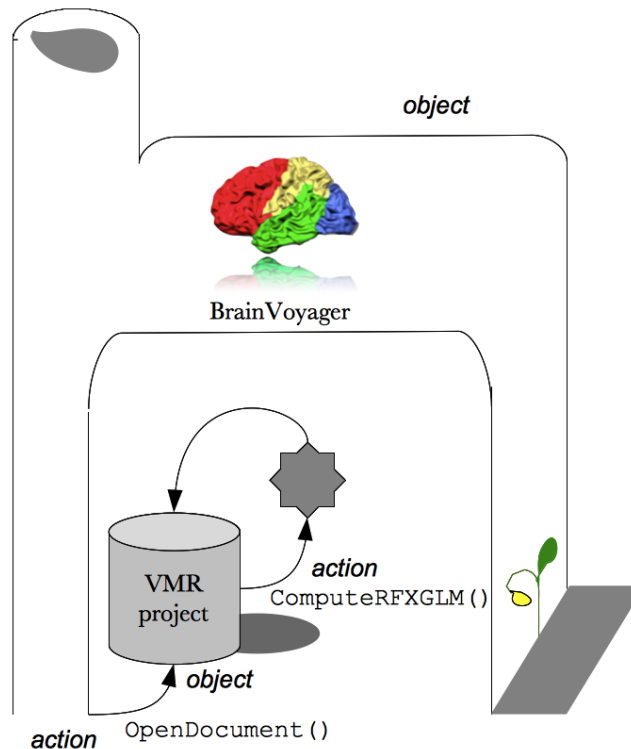


Figure 2.4: The BrainVoyager and project objects and the `OpenDocument()` and `ComputeRFXGLM()` actions

2.2.1 Actions: member-specific functions

This is the general syntax to perform actions:

```
object.doSomething()
```

which is in our case

```
BrainVoyager.OpenDocument()
```

In BrainVoyager scripting, there are only two types of objects that can perform actions: the "BrainVoyager" object and the BrainVoyager document (VMR, FMR, DMR, AMR).

In general, the BrainVoyager object performs generic actions, and can open and create projects. The BrainVoyager object has performed the action of opening a document via the command `OpenDocument()`.

The BrainVoyager project object, which can be a VMR, FMR, DMR or AMR project, can perform everything else. The FMR project object can preprocess, the VMR project object can create VTCs and VDWs and show and manipulate meshes in the surface window, and so on.

In figure 2.4, the BrainVoyager object is depicted as performing the action to open a project via the `OpenDocument()` command, and the BrainVoyager project is instructed to compute a random effects general linear model via the command `ComputeRFXGLM()`. Each instruction line is usually followed

by a sign so that the computer knows that this was the end of the line. In JavaScript, the line ends with a semicolon (;). An object in the JavaScript language that we will use later on in the Guide, is for example `Array`, because in the `Array` object we can store a number of values at the same time, very useful!

2.2.2 Properties: member-specific properties

Properties are the values associated with an object. An example of one of the properties of the `BrainVoyager` object is `CurrentDirectory`. An object property can be accessed or modified in almost the same way as an action: the property, follows after the dot (.).

Hence the notation is `BrainVoyager.CurrentDirectory`. When this combination is used on the right hand side of the equal sign, the information is transferred to the variable on the left hand side. This is shown in the example below, where the name of the `fmr` (`fmrname`) is constructed by combining the `CurrentDirectory` property with the name of a specific FMR project “`CG_Objects.fmr`”:

```
var fmrname = BrainVoyager.CurrentDirectory + "CG_Objects.fmr";
```

Sometimes the properties can be modified. In that case the property appears on the left hand side of the equal sign, and the value that should be assigned to the property is placed on the right hand side. If we, for example, would want to create a VMR project in Talairach space with a bounding box that includes the cerebellum, we set the VMR property `ExtendedTALSpaceForVTCCreation` to `true`. This can be performed in the following way:

```
vmrproject.ExtendedTALSpaceForVTCCreation =true;
```

After setting this property, we can safely create our VTC project with cerebellum via the function `CreateVTCInVMRSpace()`. (In `BrainVoyager QX 2.4.1` we also need to set `UseBoundingBoxForVTCCreation` to `true` or `false`.)

Chapter 3

Using filenames

There are several ways to use filenames in scripts, and the most frequently used way is to write the name explicitly in the script. However, this is prone to errors since it is easy to make spelling errors, the file separators `'/'` can be confusing for Windows users, and when files are moved or renamed the script has to be changed as well.

Therefore, we will print the filename to the BrainVoyager Log tab so that we are sure to have used the proper name. In one of the later chapters we will read a text file that contains the filenames.

3.1 Write the script

First, open the script editor via `Scripts → Edit and Run scripts`. A new script will be present with the name `"untitled.js"`.

We create a variable for the filename with the keyword `var`:

```
var fmrfilename =
```

A variable is an entity that can change value (see section 3.4). In this case, we assign the value of the variable `fmrfilename` by selection of the name via a file dialog on the right hand side of the equal sign:

```
var fmrfilename = BrainVoyager.BrowseFile("Please select an FMR", "*.fmr");
```

The paths are separated by a forward slash `"/"` instead of a backslash `"\"`. This is the internal representation of the path. It is also possible to use the backslash as file separator.

The name consists of text, text is always written within double quotes: `" "`. In the script editor, text can be recognized easily, because it is shown in green.

To prevent problems in scripting, please take care that the path and filenames do not contain spaces.

Now we ask the BrainVoyager object to print the text to the Log tab:

```
BrainVoyager.PrintToLog(fmrfilename);
```

Since the value of our variable `'fmrfilename'` can change, what BrainVoyager will print depends of the current value of the FMR file name that was selected.

The script is now complete (see figure 3.2). Save the script via the `"Save As..."` button with the name `"print.filename.js"`.

3.2 Run the script

Now close the Script Editor using the “Close” button. Select the script from the list in the “Scripts” menu. The name that was saved in the script is now printed to the BrainVoyager Log tab (3.1):

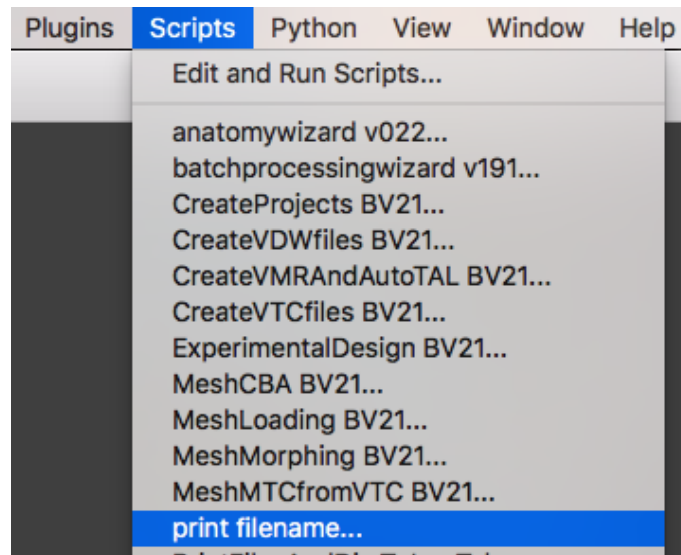


Figure 3.1: The selected filename has been written to the BrainVoyager Log tab

3.3 Script

```
var fmrfilename = BrainVoyager.BrowseFile("Please select an FMR", "*.fmr");  
BrainVoyager.PrintToLog(fmrfilename);
```

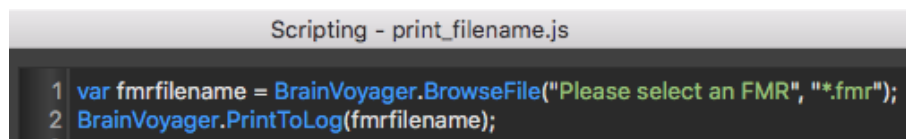


Figure 3.2: Script to print filename to the BrainVoyager Log tab

3.4 Appendix: variables

3.4.1 Variables and values of variables

A variable is an entity that can hold a value. It can be compared to an empty basket, where a text or number can be stored (see left side of figure 3.3).

In step 1, the existence of the variable is declared by the keyword `var` (variable)¹. In step 2, the variable is given the label `fmrname`. This is the name one refers to in the program. The actual value of the variable is assigned in step 3, when variable 'fmrname' is assigned the value `c:/studdata/run1/run1.fmr` using the equal sign (=).

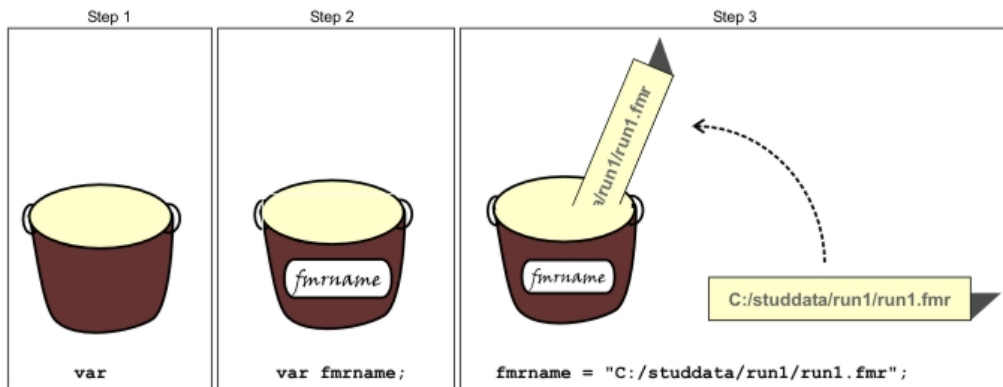


Figure 3.3: Three steps to use a variable. Step 1: bring variable into existence (using the keyword 'var'). Step 2: give the variable a label (in this case 'fmrname'). Step 3: assign a value to the variable (in this case "C:/studdata/run1/run1.fmr").

A value that should be assigned to a variable is always placed on the right hand side of the equal sign. The variable that receives the value is placed on the left hand side of the equal sign.

¹All types of variables, with either text values, whole numbers or rational numbers, or boolean values (true and false) use the same keyword `var` in JavaScript. This feature of the programming language is called *weakly typed*. The compiler infers which type it is meant to be. For example, when a number is concatenated with a string of text, the compiler thinks the variable containing the value of number 2 should be converted to text, while when a number is added via the + sign to another number, the compiler takes the numerical value of the variable.

3.4.2 Variables and objects

Sometimes the variable is a new object (for objects, see section 2.2), and therefore needs to be constructed using the keyword `new` and by invoking a function with the same name as of the object², before the value can be assigned. And sometimes a value is assigned while constructing the object, via the parentheses.

In BrainVoyager scripting, there can be three types of objects: BrainVoyager objects, JavaScript objects and self-created objects.

BrainVoyager objects

The BrainVoyager objects are either the `BrainVoyager` application object or the `BrainVoyager document (=project)` object. The `BrainVoyager` object does not have to be created with the keyword `new`. The document objects also do not need to be created explicitly, because the `BrainVoyager` object will create this for us when we invoke the functions `OpenDocument ()` or `Create [FMR/DMR/VMR/AMR] Project ()`.

JavaScript objects

The JavaScript objects are predefined in the JavaScript language. They are all mentioned in the BrainVoyager Scripting Reference Manual (and of course in JavaScript books and on websites). Examples of JavaScript objects are the date object `Date`, the multiple values object `Array`, the text object `String`, the regular expression object `RegExp` and the mathematics object `Math`. Only the `Math` object cannot be constructed via the keyword `new`, because its functions and properties can be used directly.

Self-created objects

When we are creating our own objects, we need to provide a function with the name of the object, which creates the object. This function is called the constructor of the object. The constructor can be considered as a factory which creates objects of one kind. Each time the `new` keyword is then used together with the object name, the constructor factory creates a new copy.

Then, we also need functions for the object, otherwise it is not very necessary to create an object (except if the objects needs to store lots of values, like the `Array` object). In JavaScript for example, complex numbers are not defined. Therefore we can create a `Complex` object, which keeps the real and imaginary value in the member variables `this.real` and `this.imag`³.

Then, we can create functions for the `Complex` object, for example to compute the polar coordinates of the complex number (see figure 3.4). We can create these functions `phase ()` and `magnitude ()` outside of the constructor function. But in the constructor function, we create a reference to these functions, via the command lines `this.magnitude = magnitude;` and `this.phase = phase;`. Then, each time an `Complex` object invokes the function `phase ()` or `magnitude ()`, the function takes its member variables `this.real` and `this.imag` into phase or magnitude values and returns this to the `Complex` object that invoked the function.

²A function with the same name as the object itself, which creates the object, is known as *constructor*.

³This function was created for the JavaScript accompanying the *anatabacus* plugin.

```

771 /*~~~~~*/
772 /* constructor of complex number object */
773 function Complex(x, y, isPolar) {
774
775     BrainVoyagerQX.PrintToLog("Create complex number...");
776     if (isPolar) {
777         this.real = x * Math.cos(y);
778         this.imag = x * Math.sin(y);
779     } else {
780         this.real = x;
781         this.imag = y;
782     }
783     BrainVoyagerQX.PrintToLog(this.real + " + " + this.imag + "i");
784
785     // declaration of member functions
786     this.magnitude = magnitude;
787     this.phase = phase;
788 }
789
790 function magnitude() {
791     var magnitude = Math.sqrt(Math.pow(this.real,2) + Math.pow(this.imag,2));
792     return magnitude;
793 }
794
795 function phase() {
796     var phase = Math.atan2(this.imag, this.real);
797     return phase;
798 }
799 /*~~~~~*/
800

```

Figure 3.4: The Complex() object constructor function, and the phase() and magnitude() functions

Chapter 4

Following the processing in BrainVoyager

While the scripting commands are being executed automatically, it would be nice to keep track of the process. Because, if we let BrainVoyager execute just one action, then we know exactly what it is doing currently, but if BrainVoyager is requested to execute a long list of commands, then we would like to keep a record of the processed filenames and the parameter values.

One could, for example, write a log file to the hard disk. Another, easier, possibility is to write information to the BrainVoyager Log tab (everything that has been written to the BrainVoyager Log tab can also be saved to a text file by right-clicking on the text, and press CTRL+C on Windows or CMD+C on Mac OS X for copying the text). Both possibilities, to write text to the BrainVoyager Log tab and to save text to a file are described below.

4.1 Print text literally

One can send a message to BrainVoyager to display text in the log tab via the command `PrintToLog()`:

```
BrainVoyager.PrintToLog("your text goes here");
```

Since the Log tab is not always open, first open the Log tab via the instruction

```
BrainVoyager.ShowLogTab();
```

The function `ShowLogTab()` does not ask for extra parameters. After the Log tab has been made visible, we can print the text, for example to indicate that the preprocessing of the experiment data is about to start:

```
BrainVoyager.PrintToLog("Start preprocessing of experiment data...");
```

4.1.1 Script

Run the following script, or save first as “printtext.js”:

```
BrainVoyager.ShowLogTab();  
BrainVoyager.PrintToLog("Start preprocessing of experiment data...");
```

4.2 Print text with filename

If we use different FMR projects, then it would be nice to know the name of the project that is currently preprocessed.

Therefore, we give the variable `fmrprojectname` on the left hand side the value `"/Users/home/Data/experiment/run1.fmr"` (write here instead the path and name of your own FMR project) on the right hand side of the command line and close the command line with a semicolon (;). If you are a Windows user, you might be used to having the file path separators as backslash ('\')

instead of forward slash ('/'), but to prevent problems it is safer to always use the Unix convention of writing the file path separators as forward slash, like we just did.

Then we combine the text "Currently preprocessing FMR project: " with the `fmrprojectname` variable and send it to the `PrintToLog()` function by placing it between the parentheses. The result is shown in figure 4.1, which indicates that the value of the `fmrprojectname` variable has been properly recognised and combined with our hardcoded text.

4.2.1 Script

Run the following script, or save first as "logtextWithName.js" in the directory `/Documents/BVExtensions/Scripts/`:

```
BrainVoyager.ShowLogTab();
BrainVoyager.PrintToLog("Start preprocessing of experiment data...");
var fmrprojectname = "/Users/home/Data/experiment/run1.fmr";
BrainVoyager.PrintToLog("Currently preprocessing FMR project: " + fmrprojectname);
```

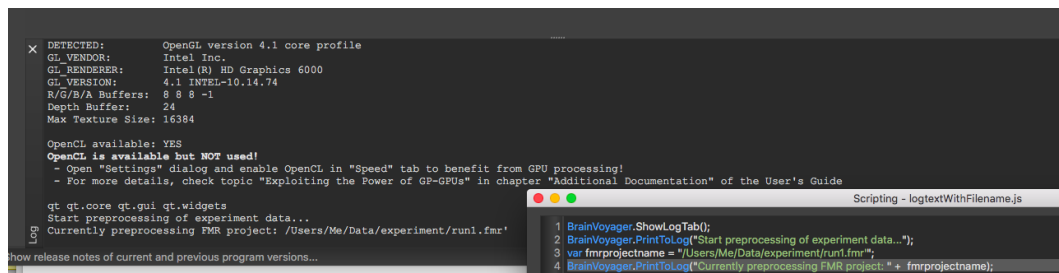


Figure 4.1: The script (front) and the result in the BrainVoyager Log tab (behind script window)

4.3 Print text with parameter value

To add parameter values to our script, we use the same principle as for text values: just use the '+' sign and the value to concatenate the text with the value. The script processor will then implicitly convert the parameter from number to text. The result is shown in figure 4.2.

4.3.1 Script

Run the following script, or save first as "motionCorrectionWithLogtext.js":

```
BrainVoyager.ShowLogTab();
BrainVoyager.PrintToLog("Start preprocessing of experiment data...");
var fmrprojectname = BrainVoyager.BrowseFile("Please select the functional data", "*.fmr");
BrainVoyager.PrintToLog("Currently preprocessing FMR project: " + fmrprojectname);
var fmrproject = BrainVoyager.OpenDocument(fmrprojectname);
BrainVoyager.PrintToLog("Reference value for motion correction: " + 1);
var success = fmrproject.CorrectMotion(1);
```

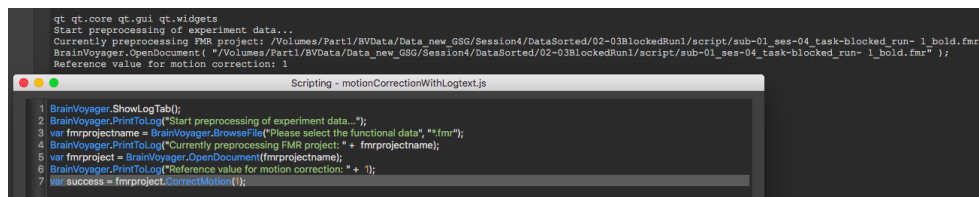


Figure 4.2: The script (front) and the result in the BrainVoyager Log tab (behind script window)

4.4 Save log text in text file

To save the text in a file, we need to go through five steps.

1. At first, a new file object is created via `new QFile()`. The name and extension for the new file, "log_preprocessing_parameters.txt" are immediately passed to the function via the parentheses:

```
var logfile = new QFile("log_preprocessing_parameters.txt");
```
2. Then, we need to open the file by invoking the function `open()`. When opening a file, usually we have to indicate in which mode the file is opened: writing mode, reading mode, or appending mode. In this case, we choose the writing mode: `WriteOnly`:

```
logfile.open(new QIODevice.OpenMode(QIODevice.WriteOnly));
```
3. In this case, we need to create a text stream object; the text stream object decides when to write the text to the file:

```
var textoutput = new QTextStream(logfile);
```
4. Now we write the preprocessing parameters to the file. The first parameter for slice scan time correction is slice order, we take 'ascending-interleaved', which is parameter 1 (see table 4.4). The second is the interpolation method, where we choose parameter '2', which is SINC (although cubic spline, parameter '1', would have been perfect, too).
5. Finally, we close the file via the function `close()`:

```
logfile.close();
```

Scan order	Parameter for scripting
ascending	0
ascending interleaved	1
ascending-interleaved 2	2
descending	10
descending-interleaved	11
descending-interleaved 2	12

Table 4.1: CorrectSliceTiming() parameter 1: scan order

Interpolation	Parameter	Resulting addition to filename
Linear	0	SCLAI
Cubic spline	1	SCCAI
SINC	2	SCSAI
Linear	3	SCLAI

Table 4.2: CorrectSliceTiming() parameter 2: interpolation options

Remarks:

To start writing on a new line, we have used the character '\n' (on Mac OS X); on Windows, one could also try '\r\n'.

When the textfile did not receive a pathname, it is written to the current directory. The current directory can be retrieved using

`BrainVoyager.PrintToLog(BrainVoyager.CurrentDirectory)` in a script or in the direct evaluation field (see section 1.2.2).

4.4.1 Script

Run the following script, or save first as “preprocessingWithLogfile.js”:

```
BrainVoyager.ShowLogTab();
BrainVoyager.PrintToLog("Start preprocessing of experiment data...");
var fmrprojectname = "/Users/home/Data/run1/CG_Objects.fmr"; // please change this filename
BrainVoyager.PrintToLog("Currently preprocessing FMR project: " + fmrprojectname);

var logfile = new QFile("log_preprocessing_parameters.txt");
logfile.open(new QIODevice.OpenMode(QIODevice.WriteOnly));
var textoutput = new QTextStream(logfile);
textoutput.writeString("Filename: " + fmrprojectname + "\n");
textoutput.writeString("Preprocessing parameters for slice scan time correction:\n");
textoutput.writeString("Scan order (ascending-interleaved): ");
textoutput.writeInt(1);
textoutput.writeString("\n");
textoutput.writeString("Interpolation (): ");
textoutput.writeInt(2);
textoutput.writeString("\n");

var fmrproject = BrainVoyager.OpenDocument(fmrprojectname);
var success = fmrproject.CorrectSliceTiming(1,2); // please check whether the parameters for this FMR project are correct
textoutput.writeString("New name: " + fmrproject.FileNameOfPreprocessdFMR + "\n");
// until QX 2.3: FileNameOfPreprocessdFMR without pathname
logfile.close();

BrainVoyager.PrintToLog("Finished preprocessing.");
```

The script in the BrainVoyager script editor and the resulting log file are shown in figure 4.3.

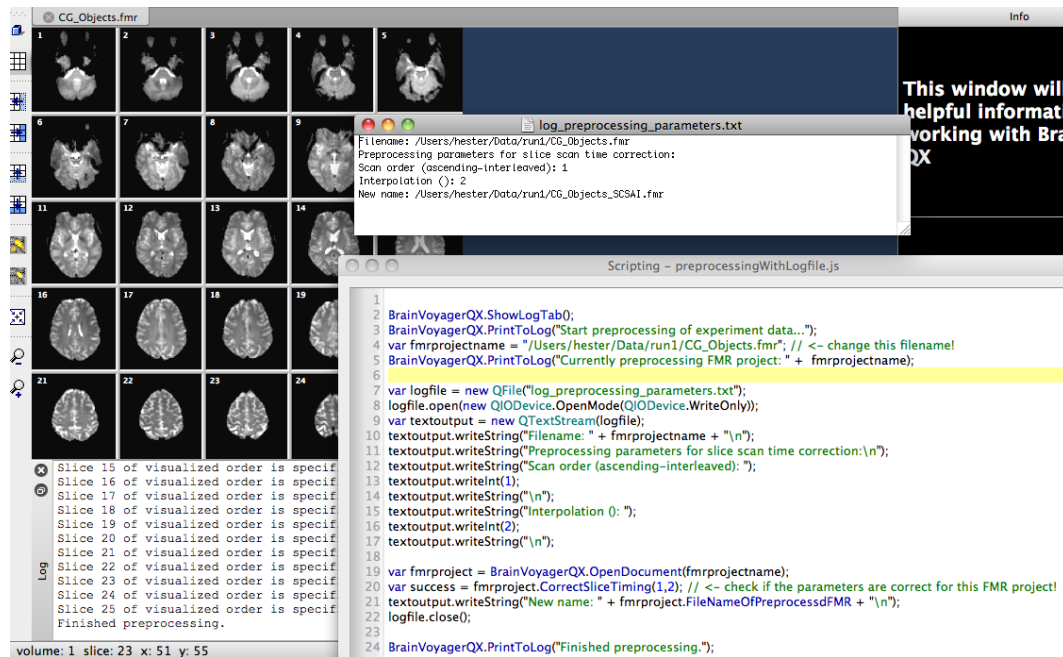


Figure 4.3: Script for preprocessing including saving a log file

4.5 Appendix: arguments and return values

The argument(s) are the input for a function that can be entered via the values between parentheses '()'. To use a function without arguments, the parentheses to invoke the function can be kept empty: `function()`. Some functions cannot process without further information. For these functions, the extra information can be passed as arguments. They can be written between parentheses: `function(information)`, for example

```
var success = fmrproject.CorrectMotion(1);
```

In this command line, we have provided the function `CorrectMotion()` with the information that it should use volume 1 as reference volume. The number 1 is the argument here.

In the figure 4.4 below is illustrated that an FMR project can invoke a function to correct motion. The result of the `CorrectMotion()` function is that it corrects the motion in the FMR project that invokes the function. This is a new, motion corrected, FMR project (not shown in illustration), which has been written to the harddisk. The FMR project object sends as information to the `CorrectMotion()` function the number of the volume that serves as reference volume.

The FMR project function also returns a boolean value `success`, which can be true or false. This value indicates whether the motion correction of the FMR file succeeded. This value is sent to the variable `success` that was written on the left hand side in our example command line above.

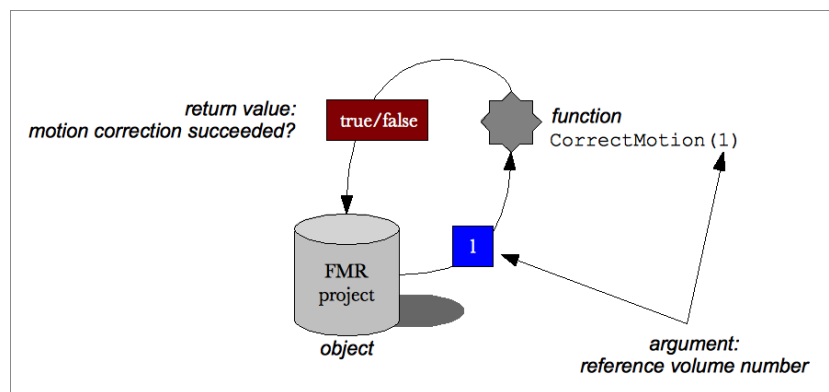


Figure 4.4: Input (arguments) and output of the function `CorrectMotion()`

Invoking the function `var success = fmrproject.CorrectMotion(1);` via scripting (see figure 4.4) is identical to using the motion correction functionality of the BrainVoyager user interface with reference volume 1 (see figure 4.5). This reference volume parameter can be selected via the 3D Motion Correction Options dialog. In case we would like to set more parameters via scripting, like it is possible via the user interface, for example the type of interpolation, the use of reduced data, etcetera, we could use the extended method `CorrectMotionEx()` instead of `CorrectMotion()`, which accepts more parameters.

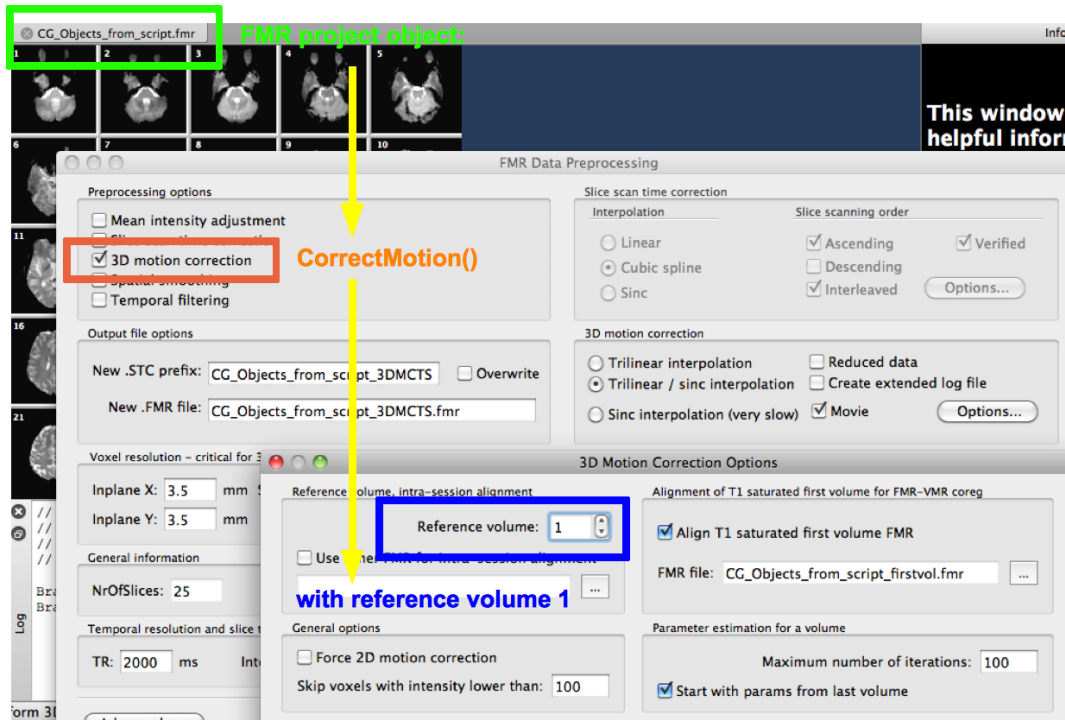


Figure 4.5: Motion correction via BrainVoyager user interface

Chapter 5

Preprocessing several files

In this chapter we will learn to use arrays and loops for applying slice scan time correction to several files. To do this, we extend the script that we made in the previous chapter with an array and a loop. We would like to apply the same procedure to several variables, that is why we need the loop. The procedure is the same for all the variables, and the loop should be indifferent about which variable it is currently using. Therefore we store the variables in an array, then the values can be accessed using a number.

5.1 Adding the array

In an array we can store multiple values. As shown in figure 3.3, a variable can be depicted as a basket to store a value. An array could be depicted as bookshelves, where on each shelf we can store a value. In figure 5.5 an array is depicted to have three shelves: level 0, level 1 and level 2. So in this array we could store 3 values, because there are 3 shelves. We can bring an array into existence in three ways:

- Declare the array while immediately initialising it (initialising = assigning value(s)) (see the example script in section 5.3.1). In this case, we place three values (words/strings, in this case) between square brackets and separate the values with a comma (',') (see the first example script). The array has now immediately three storage shelves filled with three values. The syntax is:

```
var fmrprojectnames = ["name1.fmr", "name2.fmr", "name3.fmr"];
```
- Declare the array with indication of the size (see the example script in section 5.3.2). The array has now immediately three storage shelves but no values yet. We use the keyword `new` here because we would like to obtain a new instance of the object type `Array`. The syntax is:

```
var fmrprojectnames = new Array(3);
```

In this case we can assign one value at a time using a number in square brackets as indication of the place in the array where it should be stored. We start counting at 0:

```
fmrprojectnames[0] = "/Users/hester/Data/run1/CG_Objects.fmr";
```
- Declare the array without indication of the size (see the example script in section 5.3.3). The array has now no storage space yet and no values. `var fmrprojectnames = new Array()`. We add values by using the `Array`'s function `push()`:

```
fmrprojectnames.push("/Users/hester/Data/run1/CG_Objects.fmr");
```

5.2 Adding the loop

When we use a loop (a "for" block), we can apply the same procedure to many different values. The values should be available to the loop in a standardised way. Therefore we have used an array. If we want to change a procedure in a repeated procedure, we simply add the `for () {}` keyword and the conditions around the procedure. Also, we need a counter. In the `for` block is defined at which value the counter should start, at which value it should stop iterating, and how big the steps should be. The conditions we have to define within the `for` block are:

- begin value for the iteration, for example

```
counter = 0;
```

- stop value for the iteration, for example
`counter < fmrprojectnames.length;`
- step size for going through the array, mostly 1 ('+1' = '++')
`counter++;`

Which gives then our loop:

```
for (<counter> = begin value; stop value; step size) {
< the procedure comes here>
}
```

And we add a counter with the name `projectnr` before the loop, and the procedure that we already had, so now we have a complete loop with counter, begin value, stop value and step size:

```
var projectnr;
for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) {
var fmrproject = BrainVoyager.OpenDocument(fmrprojectnames[projectnr]);
var success = fmrproject.CorrectSliceTiming(1,1);
fmrproject.Close();
}
```

If we join the array part of the script which we created in section 5.1 with the loop part of the script which we created in this section, our script to preprocess multiple FMR projects is ready.

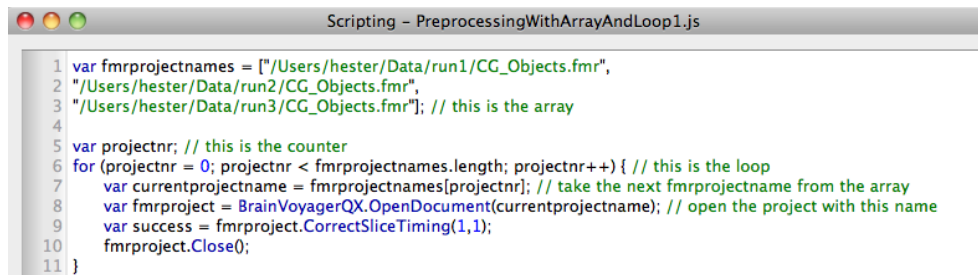
In the next section, section 5.3 “Scripts”, three variants of the array and loop script are shown. In “Scripts” section 5.3.4 is an alternative way to accessing values via `var retrievedvalue = array[index]` in the array demonstrated using the function `pop()`.

5.3 Scripts

5.3.1 Array and loop script 1

In this script, the array is declared while immediately initialising it. To use the script, replace the filenames by your own filenames.

```
var fmrprojectnames = ["/Users/hester/Data/run1/CG_Objects.fmr",  
"/Users/hester/Data/run2/CG_Objects.fmr",  
"/Users/hester/Data/run3/CG_Objects.fmr"];  
  
var projectnr;  
for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) {  
var currentprojectname = fmrprojectnames[projectnr];  
var fmrproject = BrainVoyager.OpenDocument(currentprojectname);  
var success = fmrproject.CorrectSliceTiming(1,1);  
fmrproject.Close();  
}
```



```
1 var fmrprojectnames = ["/Users/hester/Data/run1/CG_Objects.fmr",  
2 "/Users/hester/Data/run2/CG_Objects.fmr",  
3 "/Users/hester/Data/run3/CG_Objects.fmr"]; // this is the array  
4  
5 var projectnr; // this is the counter  
6 for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) { // this is the loop  
7   var currentprojectname = fmrprojectnames[projectnr]; // take the next fmrprojectname from the array  
8   var fmrproject = BrainVoyagerQX.OpenDocument(currentprojectname); // open the project with this name  
9   var success = fmrproject.CorrectSliceTiming(1,1);  
10  fmrproject.Close();  
11 }
```

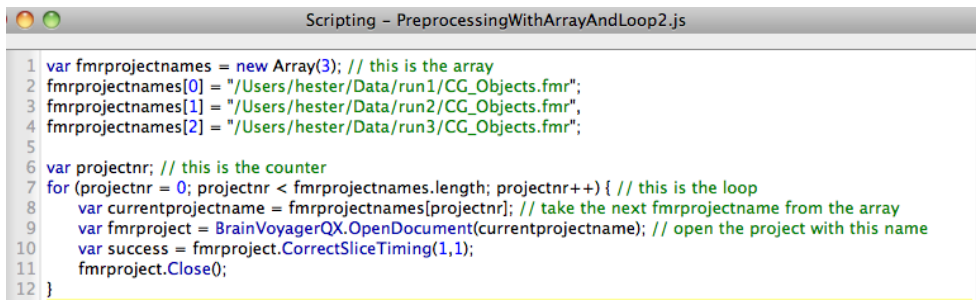
Figure 5.1: Array and loop script 1

5.3.2 Array and loop script 2

In this script, we have declared the array with indication of the size but without initial values. To use the script, replace the filenames by your own filenames. If you use less or more filenames, change the number in Array (3) to the proper number of filenames.

```
var fmrprojectnames = new Array(3);
fmrprojectnames[0] = "/Users/hester/Data/run1/CG_Objects.fmr";
fmrprojectnames[1] = "/Users/hester/Data/run2/CG_Objects.fmr";
fmrprojectnames[2] = "/Users/hester/Data/run3/CG_Objects.fmr";

var projectnr; // this is the counter
for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) {
var currentprojectname = fmrprojectnames[projectnr];
var fmrproject = BrainVoyager.OpenDocument(currentprojectname);
var success = fmrproject.CorrectSliceTiming(1,1);
fmrproject.Close();
}
```



```
Scripting - PreprocessingWithArrayAndLoop2.js
1 var fmrprojectnames = new Array(3); // this is the array
2 fmrprojectnames[0] = "/Users/hester/Data/run1/CG_Objects.fmr";
3 fmrprojectnames[1] = "/Users/hester/Data/run2/CG_Objects.fmr";
4 fmrprojectnames[2] = "/Users/hester/Data/run3/CG_Objects.fmr";
5
6 var projectnr; // this is the counter
7 for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) { // this is the loop
8   var currentprojectname = fmrprojectnames[projectnr]; // take the next fmrprojectname from the array
9   var fmrproject = BrainVoyagerQX.OpenDocument(currentprojectname); // open the project with this name
10  var success = fmrproject.CorrectSliceTiming(1,1);
11  fmrproject.Close();
12 }
```

Figure 5.2: Array and loop script 2

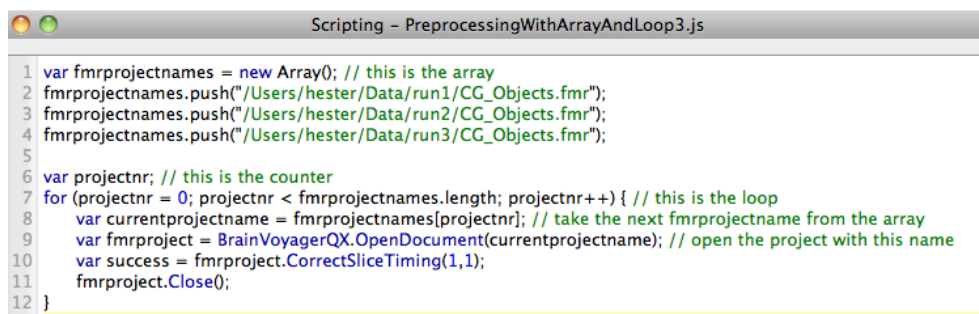
5.3.3 Array and loop script 3

In this example script, we declare the array without indication of the size. The array has no storage space when it is declared and no initial values. Storage space is created once at a time when adding a filename via the function `push()`.

To use the script, replace the filenames by your own filenames.

```
var fmrprojectnames = new Array();
fmrprojectnames.push("/Users/hester/Data/run1/CG_Objects.fmr");
fmrprojectnames.push("/Users/hester/Data/run2/CG_Objects.fmr");
fmrprojectnames.push("/Users/hester/Data/run3/CG_Objects.fmr");

var projectnr; // this is the counter
for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) {
var currentprojectname = fmrprojectnames[projectnr];
var fmrproject = BrainVoyager.OpenDocument(currentprojectname);
var success = fmrproject.CorrectSliceTiming(1,1);
fmrproject.Close();
}
```



```
Scripting - PreprocessingWithArrayAndLoop3.js
1 var fmrprojectnames = new Array(); // this is the array
2 fmrprojectnames.push("/Users/hester/Data/run1/CG_Objects.fmr");
3 fmrprojectnames.push("/Users/hester/Data/run2/CG_Objects.fmr");
4 fmrprojectnames.push("/Users/hester/Data/run3/CG_Objects.fmr");
5
6 var projectnr; // this is the counter
7 for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) { // this is the loop
8   var currentprojectname = fmrprojectnames[projectnr]; // take the next fmrprojectname from the array
9   var fmrproject = BrainVoyagerQX.OpenDocument(currentprojectname); // open the project with this name
10  var success = fmrproject.CorrectSliceTiming(1,1);
11  fmrproject.Close();
12 }
```

Figure 5.3: Array and loop script 3

5.3.4 Array manipulation script

The script uses the array function `push()` to add values (elements) to the array. To retrieve the values, we use the array name and the place number (index) of the value between the square brackets; we could have done this in another way, namely by using the array function `pop()`. This is shown in the script and figure 5.4 below. We add three elements to the empty array via `push()`, then removing the value at the top of the array via `pop()`. This removes the last added element (the largest index value). In the meantime, we check the size of the array using the array property `length`.

```
var fmrprojectnames = new Array(); // this is the array
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);
fmrprojectnames.push("/Users/hester/Data/run1/CG_Objects.fmr");
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);
fmrprojectnames.push("/Users/hester/Data/run2/CG_Objects.fmr");
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);
fmrprojectnames.push("/Users/hester/Data/run3/CG_Objects.fmr");
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);

BrainVoyager.PrintToLog("Removing: " + fmrprojectnames.pop());
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);
BrainVoyager.PrintToLog("Removing: " + fmrprojectnames.pop());
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);
BrainVoyager.PrintToLog("Removing: " + fmrprojectnames.pop());
BrainVoyager.PrintToLog("Number of values in the array: " + fmrprojectnames.length);
```

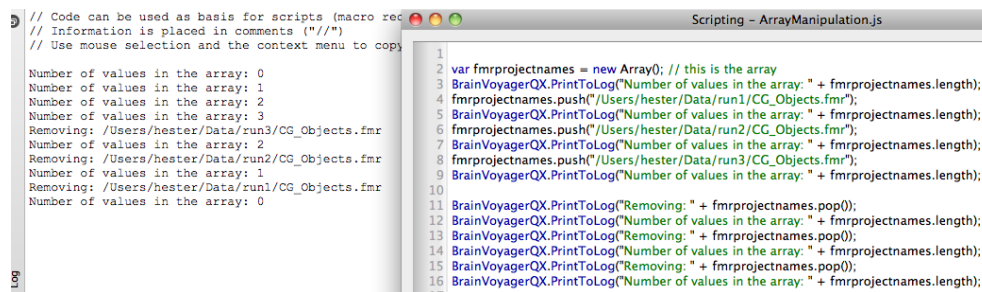


Figure 5.4: Array manipulation script

5.4 Appendix: arrays and loops

5.4.1 Arrays

Creating the array

An array is a stack of values. It can be considered as shelves, where one can store several values of the same kind. Just like in one-valued variables, first declare its existence with the keyword 'var'; for arrays, also 'new Array()' has to be added on the right hand side. The name given to this array is 'fmrnames'. In the example below, the size of the array is immediately provided (by the number 3). This means that there are 3 levels.

In JavaScript, one starts counting at 0 (see step one of figure 5.5). One can only access one level of the array at a time. The current level of array 'fmrnames' will be determined by the value of the variable 'counter'. In step 2, the value `c:/studdata/run1/run1.fmr` is assigned to level 0 of the array. In step 3, the value `c:/studdata/run2/run2.fmr` is assigned to level 1 of the array. In step 4, the value `c:/studdata/run3/run3.fmr` is assigned to level 2 of the array.

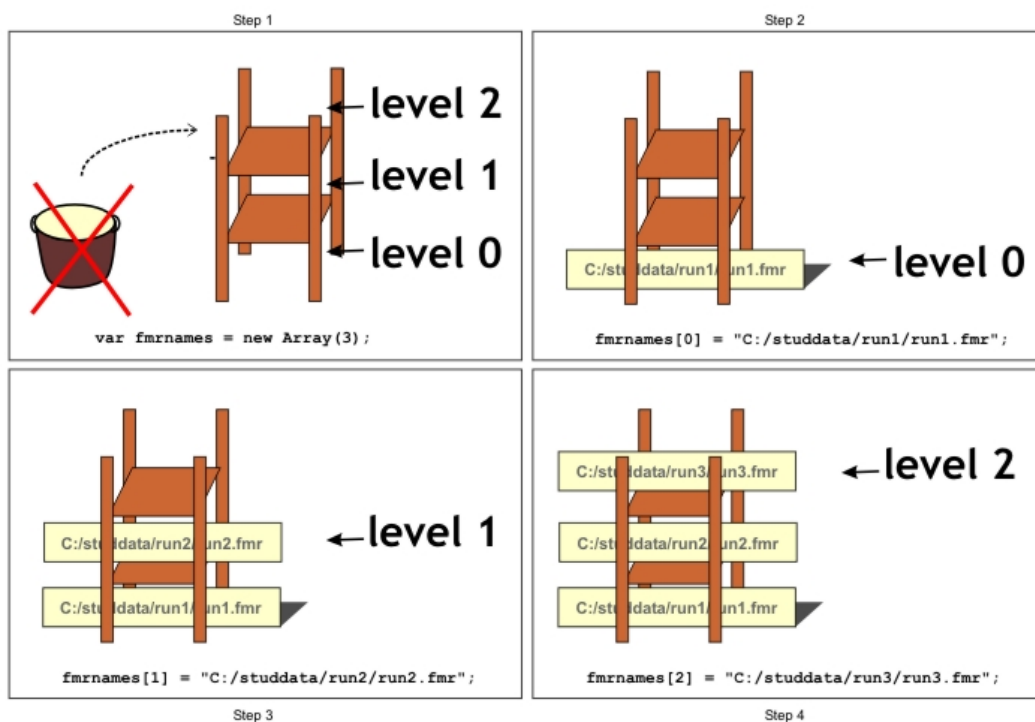


Figure 5.5: Creating an array in 4 steps

Using the array

We use the array in the loop, by retrieving one value at a time. Values in the array can be retrieved via the square brackets []. The counting starts at 0. For further information, see section 5.4.2 about loops.

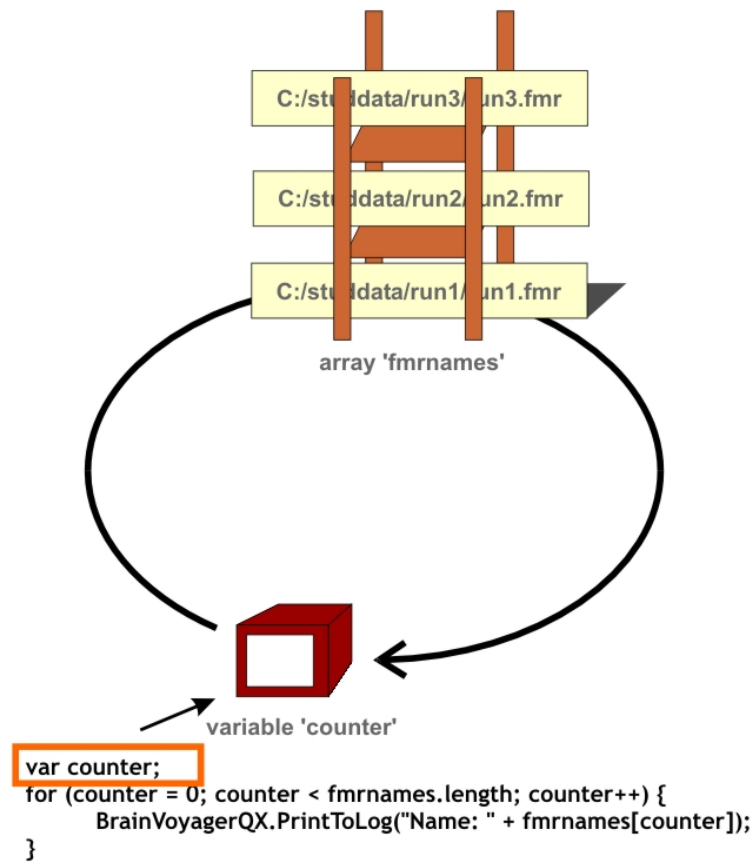


Figure 5.6: Indicate that a variable with the name 'counter' will be used

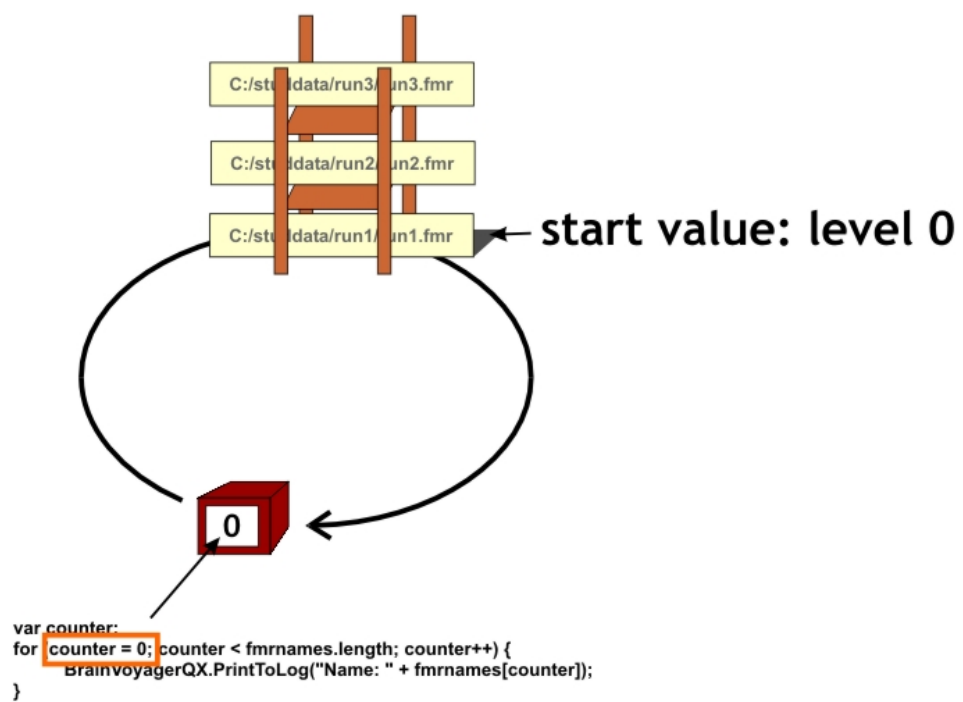


Figure 5.7: The initial value of the variable 'counter' is set to 0

5.4.2 Loops

To access the values that are stored in an array, a loop can be used. The counter, which was introduced in step 1, starts counting at 0 (step 2). Each time a walk is made to the shelves to pick the value of the layer that the counter indicates, the value of the counter is incremented by 1. This is indicated by '++'. In the first walk, the value at level 0 is retrieved by asking the value of `fmnames[0]`, which is `c:/studdata/run1/run1.fmr`. When this also has been done for `fmnames[1]` and `fmnames[2]`, the stop value is reached. The stop value, in step 4, is the length of the array, which numerical value is 3, because there are 3 layers.

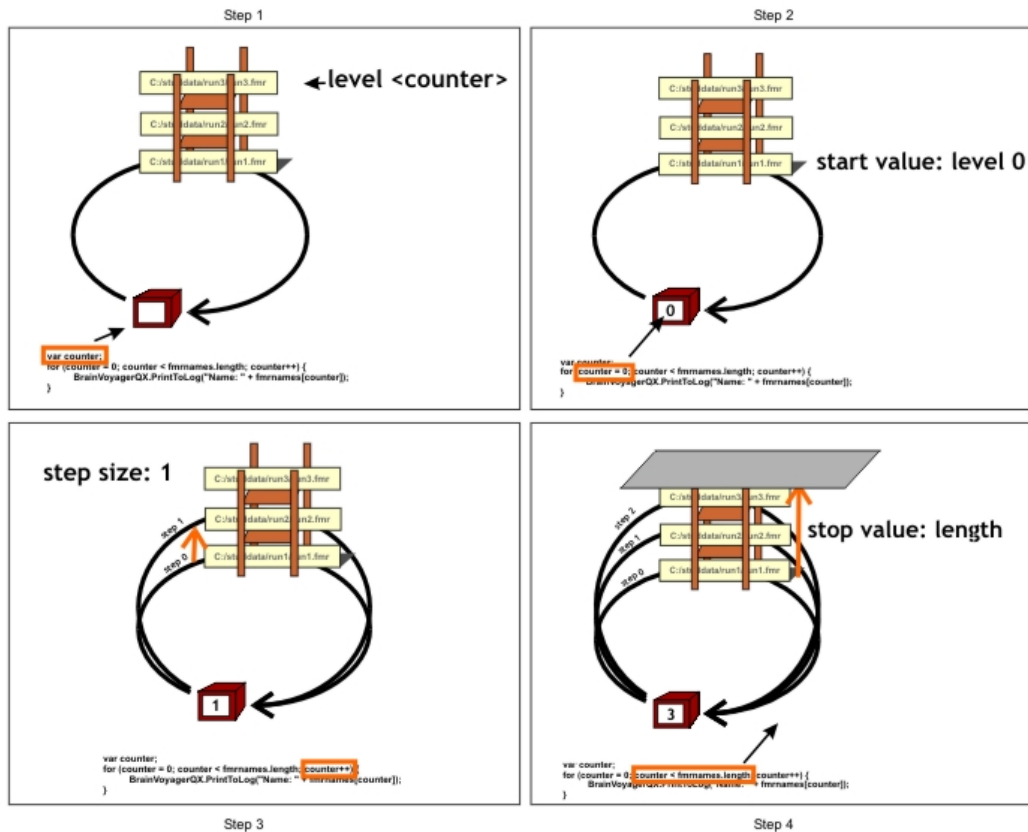


Figure 5.8: Looping through the levels of the array

Chapter 6

Creating a function within the script: read a text file

Thus far, we did not need to create our own functions yet, while we can write a nice script that uses arrays and loops to perform preprocessing on several files.

However, because the script becomes dense with all these filenames, it might be useful to have a function that reads a text file containing the filenames. So then we don't have to change the filenames in the script each time we would like to use the script, only the text file needs to be adapted.

6.1 Writing a function

6.1.1 Setup of the function

In JavaScript, a function is created within a block that starts with the keyword `function`, followed by the name of the function (which shall not contain spaces), round parentheses for arguments, and brackets to identify all code that belongs to the function (see figure 6.1).

```
4 function readTextFile() {  
5 |  
6 }
```

Figure 6.1: The function block

6.1.2 Communicating with the function

However, as we may recall from section 4.5 about arguments and return values, if we want to communicate with our function, which is to provide information to the function via an argument and receive information via the return value, we need to adapt our function block.

Because our function should read a text file, we will have to provide the function information about which text file to read. Therefore, we send it a variable containing the value of the name of the text file to our function. Within the parentheses, we write some descriptive name, which indicates to the function that it can expect a value to arrive from the caller. Because we send the name of the textfile, we call the argument here 'name'. This does not mean that the variable that we send to the `readTextFile()` function has to be called `name`, but inside the function we have indeed to work with this variable name (unless we rename the variable name).

6.1.3 Working with the text file

Just like in the example where we wrote a text file, we need to perform certain actions to work with a file:

```

4 function readTextFile(name) {
5
6     var projfilenames = new Array();
7
8     return projfilenames;
9 }

```

Figure 6.2: Argument and return value in the function block

- Create a file
- Open the file using an opening mode
- Create a textstream
- Read the text
- Close the file
- Return the information to the caller

Create a file: We create a file by creating a new instance of the Qt object `QFile`:

```
var filenamesfile = new QFile;
```

Open the file using an opening mode: To obtain access to the file, it needs to be opened. Therefore, we use the keyword `open()`, with a flag that indicates in which mode the file should be opened. The most often used flags are `read` and `write`. In this case, the flags belong to some object called `QIODevice`, which is not important to remember, therefore we just use it as given:

```
filenamesfile.open(new QIODevice.OpenMode(QIODevice.ReadOnly));
```

Create a textstream: Then, we create a textstream which reads the file for us; generally in reading files, a text stream is not always required, therefore we just use it without further explanation:

```
var textstr = new QTextStream(filenamesfile);
```

Read the text: We know that the text is a line with a number (the number of file names) and then that number of lines with filenames, which are all regular text.

We can extract numbers in the text by first reading the line using the `readLine()` function, and looking in the text that is returned by the `readLine()` function via the function `parseInt()`. Output that is returned from `readLine()` is used as input by a second function by just placing parentheses of this function around it: *secondfunction(firstfunction(input))*:

```
var nroffiles = parseInt(textstr.readLine());
```

Then we read all the lines into an array. We make an empty array on beforehand

(`var projfilenames = new Array();`), then in the loop, read each line and add it to the array using the `push()` function.

```

for (filecounter = 0; filecounter < nroffiles; filecounter++) {
var filename = textstr.readLine();
BrainVoyager.PrintToLog(filename);
projfilenames.push(filename);
}

```

Close the file: To close the file, we use the `close()` command, which is similar to BrainVoyager's `close()` command:

```
<file object instance>.close();
```

```
filenamesfile.close();
```

Return the information to the caller: We would like to send the `filenames` to the caller. We always can just send one variable via the `return` keyword. But, because this is object-oriented programming, this variable can be an object containing many values. And that is what we have here, an array called `projfilenames` with several filenames in it:

```
return projfilenames;
```

6.1.4 Script

The script below reads a text file that consists of the number of files on the first line, and filenames on all following lines. The text file with the three filenames and the number of files, in the format

```
<number of filenames>
<filename 1>
<filename 2>
...
<filename n>
```

should be saved in the directory where `BrainVoyager.PathToData` points to (to find which directory this is, place `BrainVoyager.PathToData` in the direct evaluation field of the script editor, see section 1.2.2).

```
var filename = String(BrainVoyager.PathToData + "filenamesfile.txt");
var fmrprojectnames = this.readTextFile(filename);

function readTextFile(name) {

var projfilenames = new Array();
var filecounter; BrainVoyager.PrintToLog("Reading: " + name);
var filenamesfile = new QFile(name);
filenamesfile.open(new QIODevice.OpenMode(QIODevice.ReadOnly));
var textstr = new QTextStream(filenamesfile);
var nroffiles = parseInt(textstr.readLine());
for (filecounter = 0; filecounter < nroffiles; filecounter++) {
var filename = textstr.readLine();
BrainVoyager.PrintToLog(filename);
projfilenames.push(filename);
}
filenamesfile.close();
return projfilenames;
}
```

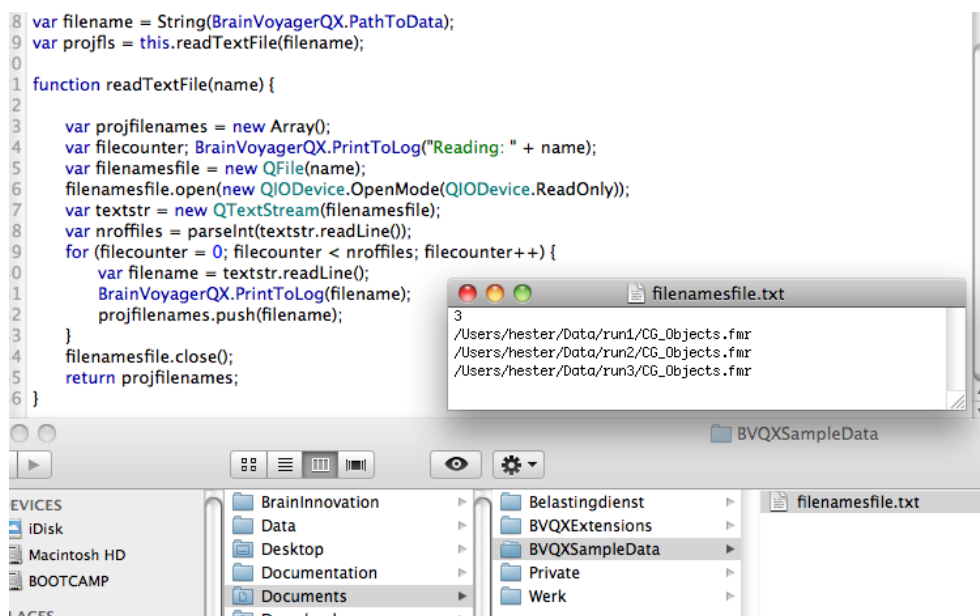


Figure 6.3: The function to read a text file, the text file and the PathToData directory

6.2 Adding the function to our script

We take the `PreprocessingWithArrayAndLoop1.js` script to modify this for our purposes. At first, save it with the name `PreprocessingWithArrayLoopAndTextFile.js`. Because the filenames are now saved in a text file, the names can be cut from the script and pasted into a text file (see figure 6.3).

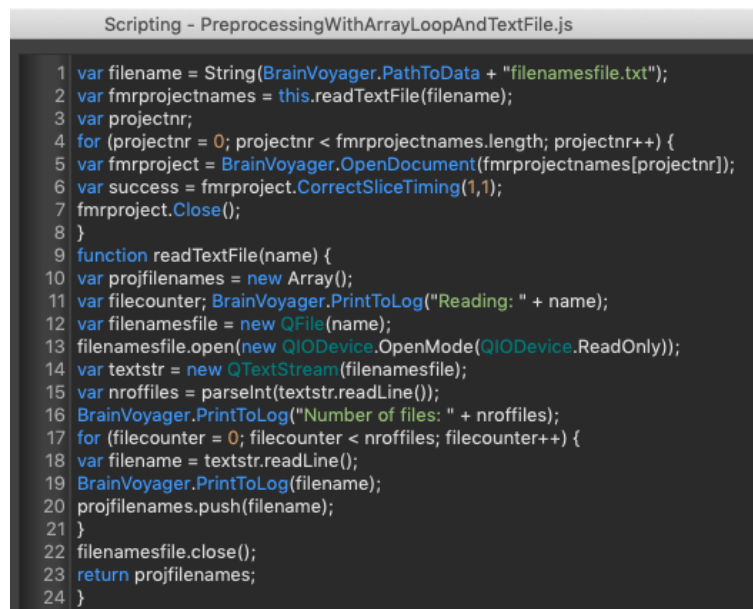
The information about the FMR filenames is now returned from the function `readTextFile()`, so we receive this from our new function using the variable `var fmrprojectnames`. Then, we simply use the loop that was already in the script. The new function we paste at the bottom of the script. Now we're done. The function is just invoked once, to read the text file, then the loop is invoked just as often as there are filenames.

6.2.1 Script

```
var filename = String(BrainVoyager.PathToData + "filenamesfile.txt");
var fmrprojectnames = this.readTextFile(filename);
var projectnr;
for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) {
var fmrproject = BrainVoyager.OpenDocument(fmrprojectnames[projectnr]);
var success = fmrproject.CorrectSliceTiming(1,1);
fmrproject.Close();
}

function readTextFile(name) {

var projfilenames = new Array();
var filecounter; BrainVoyager.PrintToLog("Reading: " + name);
var filenamesfile = new QFile(name);
filenamesfile.open(new QIODevice.OpenMode(QIODevice.ReadOnly));
var textstr = new QTextStream(filenamesfile);
var nroffiles = parseInt(textstr.readLine());
BrainVoyager.PrintToLog("Number of files: " + nroffiles);
for (filecounter = 0; filecounter < nroffiles; filecounter++) {
var filename = textstr.readLine();
BrainVoyager.PrintToLog(filename);
projfilenames.push(filename);
}
filenamesfile.close();
return projfilenames;
}
```



```
Scripting - PreprocessingWithArrayLoopAndTextFile.js
1 var filename = String(BrainVoyager.PathToData + "filenamesfile.txt");
2 var fmrprojectnames = this.readTextFile(filename);
3 var projectnr;
4 for (projectnr = 0; projectnr < fmrprojectnames.length; projectnr++) {
5 var fmrproject = BrainVoyager.OpenDocument(fmrprojectnames[projectnr]);
6 var success = fmrproject.CorrectSliceTiming(1,1);
7 fmrproject.Close();
8 }
9 function readTextFile(name) {
10 var projfilenames = new Array();
11 var filecounter; BrainVoyager.PrintToLog("Reading: " + name);
12 var filenamesfile = new QFile(name);
13 filenamesfile.open(new QIODevice.OpenMode(QIODevice.ReadOnly));
14 var textstr = new QTextStream(filenamesfile);
15 var nroffiles = parseInt(textstr.readLine());
16 BrainVoyager.PrintToLog("Number of files: " + nroffiles);
17 for (filecounter = 0; filecounter < nroffiles; filecounter++) {
18 var filename = textstr.readLine();
19 BrainVoyager.PrintToLog(filename);
20 projfilenames.push(filename);
21 }
22 filenamesfile.close();
23 return projfilenames;
24 }
25
```

Figure 6.4: The final script `PreprocessingWithArrayLoopAndTextFile.js`

Chapter 7

Scripting the Getting Started Guide

In this chapter is shown how the steps in the Getting Started Guide can be scripted. The scripts are here shown separately for each step, but the complete scripts are distributed with this BrainVoyager Getting Started Guide for Scripting (names: GSGpart1.js, GSGpart2.js and GSGpart3.js).

7.1 Introduction of the script(s)

This script shows all functions of the getting started guide that can be scripted. Variables are used often to make the script more understandable and also easily allow the user to change something. Writing own functions is avoided to make the script easily readable in sequential order. All text that belongs to the scripts are printed in Courier New font.

```
/*
GSG part 1
Script for BrainVoyager
By Joost Mulders, 06-2013
    Modified for BrainVoyager 20.4 (14-12-16) and for BrainVoyager 21.4 (July 2019) by HB
- changed 'BrainVoyagerQX' to 'BrainVoyager'
- added SavingPath to *.fmr, *.prt, *.sdm and *.glm
    - changed hardcoding of path to use of BrowseDirectory()
    - added file separator to path and filename concatenation
    - changed comment for step 7 (ROI analysis)
```

This script shows all functions of the getting started guide that can be scripted. Variables are used often to make the script more understandable and also easily allow the user to change something. Writing own functions is avoided to make the script easily readable in sequential order.

Part 1: Analysis in original space (FMR-STC) and normalised space (VMR with VTC)

```
Files Needed:
- Functional and anatomical DICOM files, starting with
JudEck_20181128_BI_Exercises_sess4-0002-0001-00001.dcm and
JudEck_20181128_BI_Exercises_sess4-0008-0001-00001.dcm
*/
```

Step 1: Creation of a functional project

```
/**
//Step 1: Creation of a functional project
//*****
BrainVoyager.PrintToLog("Step 1: Creation of a functional project");
var DicomPath = BrainVoyager.BrowseDirectory("Please select the directory with the DCM files");
var SavingPath = BrainVoyager.BrowseDirectory("Please select the directory to save the data");
var FilePath = SavingPath;

BrainVoyager.PrintToLog("Step 1: Creation of a functional project");
var DataFormat = "DICOM";
var FirstSourceFile = "JudEck_20181128_BI_Exercises_sess4-0002-0001-00001.dcm";
var NrVolumes = 291;
var NrVolumesSkip = 0;
var CreateAMR = true;
var NrSlices = 64;
var STCname = "sub-01_ses-04_task-blocked_run- 1_bold";
```

```

var SwapBytes = false;
var MosaicResolutionX = 800;
var MosaicResolutionY = 800;
var NrBytes = 2;
var NrVolumesPerFile = 1;
var ResolutionX = 100;
var ResolutionY = 100;

var FMRname = "sub-01_ses-04_task-blocked_run- 1_bold";
var SavingName = SavingPath + "/" + ".fmr";

var VMRname = "sub-01_ses-04_acq-nondistorted_T1w";

BrainVoyager.PrintToLog("DicomPath + FirstSourceFile: " + DicomPath + FirstSourceFile);
var FMR = BrainVoyager.CreateProjectMosaicFMR(DataFormat, DicomPath + "/" + FirstSourceFile,
NrVolumes, NrVolumesSkip, CreateAMR, NrSlices, STCname, SwapBytes, MosaicResolutionX,
MosaicResolutionY, NrBytes, SavingPath, NrVolumesPerFile, ResolutionX, ResolutionY);
FMR.SaveAs(SavingName);

```

Step 2: Layout and zooming

Not Scriptable.

Step 3: Creation of the stimulation protocol

```

//*****
//Step 3: Creation of the stimulation protocol
//*****
BrainVoyager.PrintToLog("Step 3: Creation of the stimulation protocol");

FMR.ClearStimulationProtocol();
FMR.StimulationProtocolExperimentName = "Faces Houses in LVF, CVF, RVF";
FMR.StimulationProtocolResolution = 1;
FMR.AddCondition("Faces_LVF");
FMR.AddCondition("Houses_RVF");
FMR.AddCondition("Faces_CVF");
FMR.AddCondition("Houses_LVF");
FMR.AddCondition("Faces_RVF");
FMR.AddCondition("Houses_CVF");

FMR.AddInterval("Faces_LVF", 4, 11);
FMR.AddInterval("Houses_RVF", 20, 27);
FMR.AddInterval("Faces_CVF", 36, 43);
FMR.AddInterval("Houses_LVF", 52, 59);
FMR.AddInterval("Faces_RVF", 68, 75);
FMR.AddInterval("Houses_CVF", 84, 91);
FMR.AddInterval("Faces_LVF", 100, 107);
FMR.AddInterval("Houses_RVF", 116, 123);
FMR.AddInterval("Faces_CVF", 132, 139);
FMR.AddInterval("Houses_LVF", 148, 155);
FMR.AddInterval("Faces_RVF", 164, 171);
FMR.AddInterval("Houses_CVF", 180, 187);
FMR.AddInterval("Faces_LVF", 196, 203);
FMR.AddInterval("Houses_RVF", 212, 219);
FMR.AddInterval("Faces_CVF", 228, 235);
FMR.AddInterval("Houses_LVF", 244, 251);
FMR.AddInterval("Faces_RVF", 260, 267);
FMR.AddInterval("Houses_CVF", 276, 283);

FMR.SetConditionColor("Faces_LVF", 200, 43, 43);
FMR.SetConditionColor("Houses_RVF", 200, 200, 43);
FMR.SetConditionColor("Faces_CVF", 43, 200, 43);
FMR.SetConditionColor("Houses_LVF", 43, 200, 200);
FMR.SetConditionColor("Faces_RVF", 43, 43, 200);
FMR.SetConditionColor("Houses_CVF", 200, 43, 200);

FMR.StimulationProtocolBackgroundColorR = 0;
FMR.StimulationProtocolBackgroundColorG = 0;
FMR.StimulationProtocolBackgroundColorB = 0;
FMR.StimulationProtocolTimeCourseColorR = 255;
FMR.StimulationProtocolTimeCourseColorG = 255;
FMR.StimulationProtocolTimeCourseColorB = 255;
FMR.StimulationProtocolTimeCourseThickness = 2;
FMR.SaveStimulationProtocol(SavingPath + "/s01_sess4_blocked_vol.prt");
FMR.Save(); // to save link to protocol permanently

```

Step 4: Statistical tests and time courses

```

//*****
//Step 4: Statistical tests and time courses
//*****
BrainVoyager.PrintToLog("Step 4: Statistical tests and time courses");

```

```
FMR.TR = 2000;
FMR.InterSliceTime = 80;
FMR.TimeResolutionVerified = true;
FMR.Save();
```

```
//Correlation calculation is not scriptable
```

Step 5: Preprocessing of FMR projects

```
//*****
//Step 5: Preprocessing of FMR projects
//*****
BrainVoyager.PrintToLog("Step 5: Preprocessing of FMR projects");

// <<<< Slice Scan Time Correction >>>>
var ScanOrder = 2; //0 = Ascending, 1 = Asc-Interleaved, 2 = Asc-Int2, 10 = Descending, 11 = Desc-Int, 12 = Desc-Int2
var InterpolationMethod = 1; //0 = trilinear, 1 = cubic spline, 2 = windowed SINC.
if (FMR.HasSliceTimeTable) {
    FMR.CorrectSliceTimingUsingTimeTable(InterpolationMethod);
} else {
    FMR.CorrectSliceTiming(ScanOrder, InterpolationMethod);
}
var ResultFileName = FMR.FileNameOfPreprocessdFMR;
FMR.Close(); //FMR.Remove();
FMR = BrainVoyager.OpenDocument( ResultFileName );

// <<<< Motion correction >>>>
var TargetVolume = 1;
var InterpolationMethod = 2; //0 and 1 = trilinear, 2 = tri-linear detection and sinc apply, 3 = sinc.
var UseFullData = true;
var NrIterations = 100;
var CreateMovie = false;
var GenerateExtendedLogFile = false;
FMR.CorrectMotionEx(TargetVolume, InterpolationMethod, UseFullData, NrIterations, CreateMovie, GenerateExtendedLogFile);

var ResultFileName = FMR.FileNameOfPreprocessdFMR;
FMR.Remove(); //FMR.Close();
FMR = BrainVoyager.OpenDocument( ResultFileName );

// <<<< Temporal Filtering >>>>
var NrCycles = 2;
FMR.TemporalHighPassFilterGLMFourier(NrCycles);

var ResultFileName = FMR.FileNameOfPreprocessdFMR;
FMR.Remove(); //FMR.Close();
FMR = BrainVoyager.OpenDocument( ResultFileName );
```

Step 6: Statistical analysis with the General Linear Model (GLM)

```
//*****
//Step 6: Statistical analysis with the General Linear Model (GLM)
//*****
BrainVoyager.PrintToLog("Step 6: Statistical analysis with the General Linear Model (GLM)");

FMR.ClearDesignMatrix();
FMR.AddPredictor("Faces_LVF");
FMR.SetPredictorValuesFromCondition("Faces_LVF", "Faces_LVF", 1.0);
FMR.ApplyHemodynamicResponseFunctionToPredictor("Faces_LVF");
FMR.AddPredictor("Houses_RVF");
FMR.SetPredictorValuesFromCondition("Houses_RVF", "Houses_RVF", 1.0);
FMR.ApplyHemodynamicResponseFunctionToPredictor("Houses_RVF");
FMR.AddPredictor("Faces_CVF");
FMR.SetPredictorValuesFromCondition("Faces_CVF", "Faces_CVF", 1.0);
FMR.ApplyHemodynamicResponseFunctionToPredictor("Faces_CVF");
FMR.AddPredictor("Houses_LVF");
FMR.SetPredictorValuesFromCondition("Houses_LVF", "Houses_LVF", 1.0);
FMR.ApplyHemodynamicResponseFunctionToPredictor("Houses_LVF");
FMR.AddPredictor("Faces_RVF");
FMR.SetPredictorValuesFromCondition("Faces_RVF", "Faces_RVF", 1.0);
FMR.ApplyHemodynamicResponseFunctionToPredictor("Faces_RVF");
FMR.AddPredictor("Houses_CVF");
FMR.SetPredictorValuesFromCondition("Houses_CVF", "Houses_CVF", 1.0);
FMR.ApplyHemodynamicResponseFunctionToPredictor("Houses_CVF");
FMR.SaveSingleStudyGLMDesignMatrix(SavingPath + "/FacesHousesDesignMatrix.sdm");

FMR.ComputeSingleStudyGLM();
FMR.ClearContrasts();
FMR.AddContrast("1 0 0 0 -1 0");
FMR.SetContrastValue("Faces_LVF", +1);
FMR.SetContrastValue("Faces_RVF", -1);
FMR.ShowGLM();
FMR.SaveGLM(SavingPath + "/FacesHouses_FMR.glm");
FMR.Close();
```

Step 7: Detailed GLM analysis of regions-of-interest (ROIs)

// Scriptable for VMR-VTC VOIs

Step 8: Event-related averaging

Not Scriptable.

Step 9: Creation of a 3D anatomical project and inhomogeneity correction

```
BrainVoyager.PrintToLog("Step 9: Creation of a 3D anatomical project");
var DicomPathAnatomical = BrainVoyager.BrowseDirectory("Please select the directory of the anatomical data");
var DataFormat = "DICOM";
var FirstSourceFile = "JudEck_20181128_BI_Exercises_sess4-0008-0001-00001.dcm";
var NrSlices = 192;
var IsLittleEndian = false; //specification of the byte order in the file, almost always false
var XSize = 256;
var YSize = 256;
var NrBytes = 2; //Number of bytes for each pixel, almost always 2
var VMRname = "sub-01_ses-04_acq-nondistorted_T1w";
var VMR = BrainVoyager.CreateProjectVMR( DataFormat, DicomPathAnatomical
+ "/" + FirstSourceFile, NrSlices, IsLittleEndian, XSize, YSize, NrBytes);
VMR.SaveAs( FilePath + "/" + VMRname + ".vmr");
VMR.CorrectIntensityInhomogeneities();
VMR.Close();
```

Step 10: Coregistration of functional and anatomical data

```
//*****
//Step 10: Coregistration of functional and anatomical data
//*****

VMRname = VMRname + "_IIHC";
var VMR = BrainVoyager.OpenDocument(FilePath + "/" + VMRname + ".vmr");
var FMRname = FMRname + "_SCCTBL_3DMCTS_LTR_THPGLMF2c";
var success = VMR.CoregisterFMRToVMRUsingBBR(FilePath + "/" + FMRname + ".fmr");
```

Step 11: MNI transformation of anatomical data

```
//*****
//Step 11: MNI Normalization of Anatomical Data
//*****

VMR.NormalizeToMNISpace();
VMR.Close();
```

Step 12: MNI transformation of functional data

```
//*****
//Step 12: MNI transformation of functional data
//*****

var MNIVMRfilename = FilePath + "/" + VMRname + "_MNI.vmr";
var IAfilename = FilePath + "/" + FMRname + "-TO-" + VMRname + "_IA.trf";
var FAfilename = FilePath + "/" + FMRname + "-TO-" + VMRname + "_BBR_FA.trf";
var MNIfilename = FilePath + "/" + VMRname + "_TO_MNI_a12.trf";
var VTCfilename = FilePath + "/" + FMRname + "_MNI.vtc";
var DataType = 2; // 1: int16, 2: float32
var Resolution = 3; // one of 1, 2 or 3 mm^2
var Interpolation = 1; //0 = nearest neighbor, 1 = trilinear, 2 = sinc;
var Threshold = 100; //Intensity treshold to avoid interpolation calculations in non-brain voxels
var UseExtendedBoundingBox = false;

VMR = BrainVoyager.OpenDocument(MNIVMRfilename);
VMR.ExtendedTALSpaceForVTCCreation = UseExtendedBoundingBox;
VMR.CreateVTCInMNISpace(FilePath + "/" + FMRname + ".fmr", IAfilename,
FAfilename, MNIfilename, VTCfilename, DataType, Resolution, Interpolation, Threshold);
```

Step 13: Statistical analysis of 3D functional data

```
//*****  
//Step 13: Statistical analysis of 3D functional data  
//*****  
  
var VMR = BrainVoyager.ActiveDocument;  
VMR.LinkVTC(VTCfilename);  
VMR.LoadSingleStudyGLMDesignMatrix(SavingPath + "/FacesHousesDesignMatrix.sdm");  
VMR.ComputeSingleStudyGLM();  
VMR.SaveGLM(SavingPath + "/FacesHouses_VTC.glm");  
VMR.ClearContrasts();  
VMR.AddContrast("Faces > Houses");  
VMR.SetContrastString("1 -1 1 -1 1 -1");  
VMR.SetContrastValue("Faces_LVF", +1);  
VMR.SetContrastValue("Faces_RVF", +1);  
VMR.SetContrastValue("Faces_CVF", +1);  
VMR.SetContrastValue("Houses_LVF", -1);  
VMR.SetContrastValue("Houses_RVF", -1);  
VMR.SetContrastValue("Houses_CVF", -1);  
VMR.ShowGLM();
```

Step 14: Preprocessing of VTC Documents

```
//*****  
//Step 14: Preprocessing of VTC Documents  
//*****  
  
VMR.SpatialGaussianSmoothing(4, "mm");
```

Step 15: Event-related averaging of 3D functional data

Not Scriptable.

```
/*  
GSG Part 2: Analysis in surface space - SRF-MTC projects  
Script for BrainVoyager  
By Joost Mulders, 06-2013  
    Modified for BrainVoyager 20.4 by HB, 14-12-16  
    - changed 'BrainVoyagerQX' to 'BrainVoyager'  
    - changed hardcoding of path to use of BrowseDirectory()  
    - added file separator to path and filename concatenation  
    - removed references to the file CG2_3DT1FL_CLEAN.vmr (not used in GSG 2.13 any more)  
    - added step 22  
Modified for BrainVoyager 21 by HB, July 2019  
    - changed example data and GSG steps
```

This script shows all functions of the getting started guide that can be scripted. Variables are used often to make the script more understandable and also easily allow the user to change something. Writing own functions is avoided to make the script easily readable in sequential order.

```
Files Needed:  
- VMRs: - sub-01_ses-04_acq-nondistorted_T1w_IIHC_MNI.vmr  
- SRFs: - sub-01_ses-04_acq-nondistorted_T1w_IIHC_MNI_WM_LH_RECOSM.srf  
*/  
  
var SavingPath = BrainVoyager.BrowseDirectory("Please select the directory to save the data");  
var FilePath = SavingPath; //Path of all files mentioned above  
  
var FMRname = "sub-01_ses-04_task-blocked_run- 1_bold_SCCTBL_3DMCTS_LTR_THPGLMF2c";  
var VMRname = "sub-01_ses-04_acq-nondistorted_T1w_IIHC_MNI";
```

Step 16: Surface reconstruction of the head

Not Scriptable.

Step 17: Navigation in the surface module window

```
var VMR = BrainVoyager.OpenDocument(FilePath + "/" + VMRname + ".vmr");  
if (VMR == undefined) return;  
var meshscene = VMR.CreateMeshScene();  
var SRFname = BrainVoyager.BrowseFile("Please select the surface file", "*.srf");  
var success = meshscene.LoadMesh(SRFname);  
var mesh = meshscene.CurrentMesh;  
//mesh.ViewPointTranslationX  
VMR.ViewpointTranslationX = -500;  
VMR.ViewpointTranslationY = 0;
```

```

VMR.ViewpointTranslationZ = 0;
VMR.ViewpointRotationX = 0;
VMR.ViewpointRotationY = 180;
VMR.ViewpointRotationZ = 180;
mesh.UpdateAppearance();

```

Step 18: Slicing a polygon mesh

Not Scriptable.

Step 19: Automatic cortex segmentation

Not Scriptable.

Step 20: Cortex inflation

```

//*****
//Step 20: Cortex inflation
//*****
var NrOfIterations = 500;
var SmoothingForce = 0.8;
mesh.MorphingUpdateInterval = 10;
mesh.InflateMesh(NrOfIterations, SmoothingForce, "");
// if "" used for 3rd param (area reference mesh), the current mesh
// at the moment the fn is called is used to calculate ref mesh area
var NrOfIterations = 300;
mesh.InflateMesh(NrOfIterations, SmoothingForce, "");
mesh.SaveAs(FilePath + "/"
+ "sub-01_ses- 04_acq-nondistorted_T1w_IIHC_MNI_WM_LH_RECOSM_INFL.srf");

```

Step 21: Changing mesh colors

Not Scriptable.

Step 22: Statistical maps on cortex meshes

```

//*****
//Step 22: Statistical maps on cortex meshes
// We run statistics via a GLM on the mesh time course (*.mtc)
// instead of creating a surface map (*.smp) from the volume map (*.vmp) in the Getting Started Guide
//*****

var VTCfilename = FilePath + "/" + FMRname + "_MNI.vtc";
VMR.LinkVTC(VTCfilename);
if (mesh == undefined) return;
var ok = mesh.CreateMTCFromVTC(-1.0, 2.0, SavingPath + "/FacesHouses_MNI_LH.mtc" );
if (!ok) {
    BrainVoyager.PrintToLog("Could not create MTC data from VTC data.");
} else {
    ok = mesh.LinkMTC(SavingPath + "/FacesHouses_MNI_LH.mtc" );
    if (!ok) return;
    mesh.ClearDesignMatrix();
    ok = mesh.LoadSingleStudyGLMDesignMatrix(SavingPath + "/FacesHousesDesignMatrix.sdm");
    if (!ok) return;
    mesh.CorrectForSerialCorrelations = 2; // 1 -> AR(1), 2 -> AR(2)
    mesh.ComputeSingleStudyGLM();
    mesh.ShowGLM();
    mesh.SaveGLM(SavingPath + "/FacesHouses_MNI_LH.glm");
}

// alternatively: use mesh.CreateSurfaceMapFromVolumeMap(interpolation, nonZeroValues);
// with the following parameters - Parameter 1: Interpolation method (integer),
// Parameter 2: Sample only non zero values (boolean) for example
// mesh.CreateSurfaceMapFromVolumeMap(1, 0);

```