

Version

8

NEUROSTIM

User Guide

© Bart Krekelberg, Jessica Wright
Rutgers University
197 University Avenue • Suite 220
Newark, NJ 07102
Phone 973.353.3602 • Fax 973.353.1272

Table of Contents

Table of Contents	i
1 Introduction	3
Conventions	3
2 Getting Started	4
Defining an Experiment	4
Basic Experimental Design	8
Controllers	18
Controlling the Controller	20
Exercises	22
3 Using the Visual Studio Stimulus Wizard	23
4 Program your own Stimulus	26
The nsStimulus Class	26
Step-by-Step	29
Declare Your Class	29
The Constructor	30
The setup() Function	30
The Stimulus File	30
The draw() Function	31
The move() function	32
5 Interactive Stimuli	33
Keyboard	33
Mouse	33
Eye Movements	35
Fixation	35
On/Off	35
Complex Eye Movements	36
Reward	38
Writing Responses to File	38
6 Experiment Design	40
N-Alternative Forced Choice	Error! Bookmark not defined.
Exercise	Error! Bookmark not defined.
Complete Control	40
Staircase	41
Jitter	43
Global Variables	44
Background Stimuli	44

7	Installation.....	46
	System Requirements.....	46
	User Requirements	46
	Download	46
	Installing Neurostim.....	47
	Developers.....	47
8	History.....	48
	Background.....	Error! Bookmark not defined.
	What's New	Error! Bookmark not defined.
9	Answers to Selected Exercises	49
	Chapter 2 – Getting Started	49
10	Appendices	51
	Windows Registry Script	51
	File Formats	51
	Experiment Files (.nml)	51
	Stimulus Files (.nml)	52
	Data Files (.nmld).....	52
	Monitor Calibration Files (.xml)	53
	Learning C++ and OpenGL.....	54
	C++	54
	OpenGL	55

1 Introduction

What is Neurostim?

Neurostim is a program to design and run experiments in visual neuroscience. The user creates stimuli by using a minimal number of drawing commands from the OpenGL libraries. Behind the scenes, the Neurostim program makes sure that variables get initialized from configuration files, that multiple stimuli can be loaded, that the communication with external devices (eye trackers, spike recording) is synched and that timing data are written to an output file.

This User Guide describes how to install Neurostim, how to create your own stimuli and - for advanced users - how to modify Neurostim so that it can interact with other computers or programs. Depending on what you want to do, some chapters of this User Guide may not be relevant; check the “Should I read this?” notes at the beginning of chapter. As you read, certain sections will have exercises so you can immediately put the ideas into action. At the end of each chapter there are final exercises for you to put ideas together in a more realistic format. To complete the exercises, copy the directory nsLLDots to a temporary directory or your home directory so that you can change the experiment and stimulus configuration files.

CONVENTIONS


While Neurostim can in principle be run under any operating system, and can be compiled with different compilers, this User Guide assumes that you did a standard installation (see section Installation):

1. You work with Microsoft Windows
2. You use Visual Studio C++ (Express) as your development environment and know some C++ and OpenGL. (Some pointers to learn about this are given on page 54)
3. The .nml and .nmld file extensions are associated with Neurostim to run and an XML editor to view/edit. (See page 51 for a Windows registry script that does this).

2 Getting Started

SHOULD I

READ THIS?

 Yes; anyone working
with Neurostim

First, let's see Neurostim in action.

Go to the start menu and select Start | All Programs | Neurostim | Neurostim.

For this experiment, the Neurostim window will initially be maximized and brought to the foreground. The user will be prompted to press the Enter key to begin the experiment. To minimize the Neurostim screen, Ctrl-f can be pressed. To return the Neurostim window to full screen simply press Ctrl-f again.

When you minimized the Neurostim screen you may have noticed an additional window present; this is the command prompt. When Neurostim is initialized and during the experiment this command prompt will also show up with messages. These messages can be important now and certainly in a real experiment, so keep an eye on them. To begin the experiment press the Enter key and after a short delay you should see some moving dots on the screen.

Neurostim started this experiment at a random condition, but you can choose other conditions by pressing the number keys. For instance, to run condition number one, simply press the '1' key or to run condition number two, simply press the '2' key and so on (in this nsLLDots example there are four conditions). Please note, this feature only works for up to nine conditions. Another way to select different conditions is to press the 'N' key to advance to the next condition or the 'P' key to move to the previous condition. Unlike using the number keys, these letter keys will work for conditions past condition nine, for example if condition 9 is being presented and the 'N' key is pressed, the experiment will then advance to condition 10, providing there are at least 10 conditions in the experiment. If you choose a condition that does not exist, a warning message will appear in the Command Prompt window and you will need to press any key in order for the experiment to continue.

Another keyboard command that you may want to test is the Ctrl-e key; it shows the stimuli without erasing previous stimuli. This can be useful to test whether your moving stimulus really does appear in every position it is supposed to appear.

Pressing Ctrl-c makes a cursor visible that you can move with the mouse. A left click moves all suitable stimuli to that position providing that <ALLOWRFCHANGE> is enabled, i.e. it is set to 1, in your experiment file (the nsLLDots example has this enabled). We will discuss this file in the next section.

Ctrl-q closes the window and ends the experiment. (For a full list of keyboard commands, see page 20)

DEFINING AN EXPERIMENT

Two types of files were involved to run the example Neurostim experiment in the previous section. An experiment file and a stimulus configuration file.

📄 ***The experiment file.*** This is the main Neurostim file that is used when running an experiment, such as, nsLLDots. This file contains information to be able to run the experiment, as well as, for the overall design of the experiment. (For additional information on Advanced Experimental Design, see Chapter 6). These files have the *.nml* extension (Neurostim Markup Language) and contain the following type of information:

- Basic information concerning your computer, i.e. how big your monitor is, how fast its refresh rate is, etc.
- Where information is stored, i.e. where you want to store output files or where it should access information concerning the stimuli.
- Which stimuli to use and it may also contain default parameters for each stimulus. These parameters may also be stored in a stimulus configuration file, which will be discussed later in this section.
- Information on the overall design of the experiment, i.e. the conditions to be used in the experiment and parameters necessary for specific conditions, intertrial interval (ITI) and how the conditions will be presented.

The structure of these files is simple: At the top level you find the <Experiment> tag which contains all other tags and parameters for the experiment including one <Controller> tag that specifies how Neurostim will run, and a separate <DESIGN> tag containing one or more <STIMULUS> tags that specify which stimuli will be displayed.

Let's take a look at the experiment file, C:\Program Files\Neurostim\Samples\nsLLDots\experiment.nml, for the nsLLDots experiment. Initially, we will just do a brief overview and then go into more specifics about how the different elements are used for basic experimental design.

```

<?xml version="1.0" encoding="utf-8"?>
<Experiment>
  <Controller Name="Keyboard">
    <DESIGN>
      <ITIDURATION>250</ITIDURATION>
      <TRIALDURATION>2000</TRIALDURATION>
    </DESIGN>
    <WINDOW>
      <XPIXELS>1024</XPIXELS>
      <YPIXELS>768</YPIXELS>
      <XORIGIN>0</XORIGIN>
      <YORIGIN>0</YORIGIN>
      <WIDTH>30</WIDTH>
      <HEIGHT>20</HEIGHT>
      <BACKGROUND>
        <XCIE>0.33</XCIE>
        <YCIE>0.33</YCIE>
        <LUMINANCE>0</LUMINANCE>
      </BACKGROUND>
    </WINDOW>
    <DIRECTORIES>
      <CONFIG>./configs</CONFIG>
      <OUTPUT>c:\temp</OUTPUT>
      <DLL>./dlls</DLL>
    </DIRECTORIES>
    <ALLOWRFCHANGE>1</ALLOWRFCHANGE> <!--Comment: Allow the stimulus to follow mouse clicks-->
  </Controller>
  <DESIGN>
    <STIMULI>
      <STIMULUS Name="nsFixation" Background="0" Config="Fixation">
        <Default>
          <X>0</X>
          <Y>0</Y>
          <Enable>1</Enable>
          <shape>LEFT</shape>
          <R>1</R>
          <G Jitter="0.5">0.5</G>
          <B>0</B>
          <size>0.25</size>
          <off>2000</off>
          <on>0</on>
        </Default>
      </STIMULUS>
      <STIMULUS Name="nsLLDots" Config="oblique" />
    </STIMULI>
    <CONDITIONS>
      <EVERY>
        <nsFixation/>
      </EVERY>
      <CONDITION Name="left">
        <nsLLDots>
          <X>-10</X>
          <coherence>1</coherence>
        </nsLLDots>
      </CONDITION>
      <FACTORIAL Name="coherence" Levels="3" >
        <FACTOR1>
          <nsLLDots>
            <X>0,0,0</X>
            <coherence>0,0.5,1</coherence>
          </nsLLDots>
        </FACTOR1>
      </FACTORIAL>
    </CONDITIONS>
    <BLOCKS Repeats="5">
      <BLOCK Randomize ="WithoutReplacement" Retry ="RANDOMINBLOCK">
        <coherence Repeats="5"/>
      </BLOCK>
      <BLOCK Retry ="IGNORE">
        <left Repeats="2"/>
      </BLOCK>
    </BLOCKS>
  </DESIGN>
</Experiment>

```


<Controller> element:

The <Controller> element contains parameter settings for various properties of the controller. Most importantly, it specifies the type of controller (Keyboard in this case; a controller used to inspect your stimuli by hand –see the [Controllers](#) section.). Additionally, the <Controller> element provides information on the size of the monitor, the intertrial interval, the duration of a single trial, as well as, the directories to use for in and output. Note that <DIRECTORIES><DLL> tells Neurostim to search DLL's in the ./dlls directory; i.e. it tells Neurostim that every DLL can be found in the dlls folder, which is within the same directory as the experiment file. For a complete list of all Controller properties that can be set, open the file c:\program files\neurostim\settings\global.nml (To view this file simply right click on it and choose Edit); it contains the default values for each property and some information on what the parameters mean. This file is also helpful to provide information on how certain parameters should be formatted.


When making updates to the <Controller> element, you should view the global.nml file for reference.

 Try it out

- A. In a copy of the experiment file, experiment with the <DESIGN> <TRIALDURATION> parameter.
- B. In the same file, modify the pixels to be consistent with the resolution of your monitor, if needed. Also, adjust the width and height to reflect your monitor, if needed.

<DESIGN> element:

The <DESIGN> element¹ contains information concerning which stimuli will be used, the location of the stimulus configuration file, all the different conditions in the experiment and how the conditions should be presented, i.e. blocking. The two <STIMULUS> elements specify that two stimuli will be used in the nsLLDots experiment. The Name attribute of the <STIMULUS> element will be used as a reference to this stimulus throughout the remainder of the experiment file. The Config attribute refers to a stimulus configuration file or stimulus file for short.

 ***The stimulus configuration file.*** This file also has an .nml extension, but it contains information on just one specific stimulus. For instance, its size and color.² For each stimulus there is a separate directory with stimulus files. As mentioned

¹ Please note, this <DESIGN> element is separate from the <DESIGN> element that is within the <Controller> element. The <DESIGN> tag within the <Controller> element is used to indicate the intertrial interval and trial duration. Historically, the <DESIGN> tag within the <Controller> element was also used to indicate the order conditions would be presented, how the conditions would be randomized and the number of repeats. These last 3 properties can now be included within the <BLOCKS> element.

² Historically, the stimulus file also used to contain all conditions with parameters specific to a particular stimulus. These files are no longer necessary to run an experiment since all parameter values can be listed in the experiment file, as well as, all conditions.

previously, the Config attribute of the <Stimulus> element in the experiment file tells Neurostim which stimulus file to use for a specific stimulus. In our nsLLDots example, the second <Stimulus> element refers to the ‘oblique’ stimulus file. Neurostim will combine this with the <Directories><Config> setting (./configs), and with the name of the dll corresponding to the stimulus (nsLLDots) into the following filename: ./configs/nsLLDots/oblique.nml. The stimulus file is shown here:

```
<?xml version="1.0" ?>
<Stimulus AllowRfChange =1>
  <Default>
    <motionMode>      0      </motionMode>
    <nrDots>          250    </nrDots>
    <coherence>        1     </coherence>
    <lifetime>         250   </lifetime>
    <syncLife>         0     </syncLife>
    <dwelTime>         1     </dwelTime>
    <pointSize>        6     </pointSize>
    <maxRadius>        0.05  </maxRadius>
    <xSpeed>          -0.005 </xSpeed>
    <ySpeed>           0.005 </ySpeed>
    <on>              100   </on>
    <off>             1500  </off>
    <X>                0.0   </X>
    <Y>                0.0   </Y>
  </Default>
</Stimulus>
```

This file contains one <Stimulus> element, which has one <Default> element. There are no <Condition> elements since all conditions are included in the experiment file.

As you may guess, these files are not completely necessary for an experiment since all default parameter values can be listed directly in the experiment file as is done for the nsFixation stimulus. Any default parameters directly in the experiment file supersede values defined in the stimulus file. It may still be helpful to have a separate stimulus file especially to document comments for specific stimulus parameters³.

BASIC EXPERIMENTAL DESIGN

Let’s take a moment to walk through the <DESIGN> element in more detail and discuss how a basic experiment is implemented given that the stimuli have already been created, please use the nsLLDots experiment file for reference.

1. *Defining the stimuli to use*

Initially, you will define all stimuli that will be used within an experiment. Each stimulus will be listed separately using a <STIMULUS> tag and all <STIMULUS>


³ Comments can be added by typing the following: <!--Insert text here-->. See the nsLLDots experiment file for an example.

tags will be contained within a <STIMULI> element. For convenience, the stimuli section of nsLLDots has been copied here;

```
<STIMULI>
  <STIMULUS Name="nsFixation" Background="0" Config="Fixation">
    <Default>
      <X>0</X>
      <Y>0</Y>
      <Enable>1</Enable>
      <shape>LEFT</shape>
      <R>1</R>
      <G Jitter="0.5">0.5</G>
      <B>0</B>
      <size>0.25</size>
      <off>2000</off>
      <on>0</on>
    </Default>
  </STIMULUS>
  <STIMULUS Name="nsLLDots" Config="oblique" />
</STIMULI>
```

The first stimulus is named nsFixation. This particular stimulus controls the fixation object that appears in the center of the screen. The second stimulus is nsLLDots and this controls the moving dots that are presented on the screen.

Let's review what parameters are being set within the <STIMULUS> tag. Since a separate .dll file is not mentioned in either case, Neurostim will assume the name of the .dll file matches the stimulus name, i.e. the nsFixation.dll (which is built-in into Neurostim) and the nsLLDots.dll (which you'll find in the C:\Program Files\Neurostim\Samples\nsLLDots\dlls directory). If you prefer to use a stimulus name other than the name of the .dll file, just add in a Dll attribute, <STIMULUS Name="MovingDots" Dll="nsLLDots" Config="oblique" /> to the <STIMULUS> tag. Also note that whenever you use the same stimulus .dll file twice in the same experiment, you will need to use separate <STIMULUS> tags with different Name attributes and the same Dll attribute (the referenced stimulus file can be the same or different). An example of when this may be needed is if you want a particular stimulus to change color at some point in the experiment.


 Try it out

- C. Create a new <STIMULUS> tag after the one you modified in Part D. In this stimulus tag be sure to use a unique name, use the nsLLDots DLL and reference the "spiral" stimulus file, which is located in the same directory as the oblique.nml stimulus file.

The second attribute in nsFixation states that it is not background stimulus. Any stimulus can be set to a background stimulus, by setting the Background attribute to 1, please review page 44 for further details. As seen with the nsLLDots stimulus, the Background attribute is not necessary and will be defaulted to 0 if not present. As mentioned under the Stimulus Configuration File section, the last shown attribute is Config, which states the name of the stimulus file corresponding to this stimulus.

Since Neurostim will work with or without a stimulus file, you must decide whether or not one will be included. As shown with nsFixation, all default parameters can be set directly within the <STIMULUS> element. These default parameters in the experiment file would be used instead of any parameters defined within the stimulus file. Since nsLLDots only has a stimulus file, this file will be used to retrieve information on parameters applicable to that stimulus.

Parameters that are constant for all conditions only need to be included in the <Default> element in whichever file, i.e. experiment or stimulus file, is being used to contain this data. The parameters set in the stimulus file or within the <STIMULUS> element are specific to that stimulus; another stimulus will have quite different parameters that can be set, but the structure will be the same.

 Try it out

- D. Review the oblique.nml stimulus file. What parameters differ between this file and the <Default> element in the experiment file? If the experiment file is modified to use the parameters from the oblique.nml stimulus file what happens? Which parameter from oblique.nml is invalid given the overall experimental design from the original experiment file (Hint: Look at the timings of things)?

You may want to consider using default parameters in the experiment file and the stimulus file, but have the stimulus file contain all comments regarding the parameters related to a specific stimulus. This will help to reduce the amount of comments within the experiment file.

2. Determining the conditions that are needed.

Now that the stimuli have been defined and default parameters are listed, the conditions that are needed to vary specific stimulus parameters can be established. All conditions will be listed within the <CONDITIONS> tag. There are two different methods⁴, i.e. stand-alone conditions and factorial design, which may be employed to document all applicable conditions for an experiment. Each method will be discussed separately.

Stand-alone Conditions:

Any condition can be listed as its own separate stand-alone condition. Each condition can be listed by using a separate <CONDITION> tag within the overall <CONDITIONS> element, see below example from nsLLDots.

⁴ Neurostim will still work using the historical method of including stand-alone conditions in the stimulus files. In the historical case, the same number of total conditions must be included in each stimulus file. Each condition will contain information concerning parameters relative to the stimulus file which contains the condition. This historical method will not be discussed further since it is recommended that all conditions be placed directly in the experiment file.

```
<CONDITION Name="left">
  <nsLLDots>
    <X>-10</X>
    <coherence>1</coherence>
  </nsLLDots>
</CONDITION>
```

To modify parameters for a specific stimulus, you would type the stimulus name tag, i.e. `<nsLLDots>`, and then list the parameters to be varied. Note how conditions can be named with the Name attribute. This name will be used to reference this condition when setting up blocking for your experiment. Blocking will be discussed in the next step of this section. Again, you need only list parameters, which need to change for each condition. Any parameters common to all conditions can be included in the `<Default>` element.

Each `<CONDITION>` element contains information for one condition. This is unlike the factorial design, where certain elements may contain information for multiple conditions. In this latter case, values of parameters for different conditions are separated by commas. Since the `<CONDITION>` element only provides information for one condition, you cannot list multiple values for `<X>` and `<coherence>` in the above example. If this is needed you will need to use a factorial design or create an additional `<CONDITION>` tag.

 Try it out

- E. In your updated nsLLDots experiment file, add a new condition where the horizontal position is 10 and the coherence is 0 (Use a `<CONDITION>` tag).

Factorial Design:

Many times when developing conditions, we may have scenarios where we are varying specific parameters systematically. For example, let's say I want to develop an experiment for a line bisection task. I may want the stimulus to appear on the left, right and in the center of the screen. For each location, I may want to present a pair of intersecting lines with the point of intersection being varied. If I have 7 different points of intersection, that would result in 21 different conditions. If my task starts to require additional considerations the amount of conditions can begin to get very large. Rather than writing out each condition, the factorial design allows for you to reduce the amount of coding needed in the experiment file to account for all conditions.

Basic Syntax:

Let's see how this will work. Again returning to the nsLLDots example, we can see a simplified use of this design for syntax purposes and then modify it to show the benefits of factorial design. As before, a section of the nsLLDots experiment file has been copied for convenience.

```

<FACTORIAL Name="coherence" Levels="3" >
  <FACTOR1>
    <nsLLDots>
      <X>0,0,0</X>
      <coherence>0,0.5,1</coherence>
    </nsLLDots>
  </FACTOR1>
</FACTORIAL>

```

The <FACTORIAL> tag will contain a compilation of conditions that is constructed through the use of one or more <FACTOR> tags. In the above example, the overall factorial is named coherence and there is one <FACTOR> tag. Each <FACTOR> tag begins with the name FACTOR followed immediately by a number. In this example, since there is only one it is listed as <FACTOR1> and contains 3 conditions. To modify parameters for a specific stimulus, you would type the stimulus name tag, i.e. <nsLLDots>, and then list the parameters to be varied. In this case, there are 3 conditions where the collection of dots are all presented with a center point at horizontal position 0 and varying levels of coherence, i.e. 0, 0.5 or 1.

Using more than 1 factor:

This past example illustrated the syntax of the factorial design, but these conditions could be listed as three stand-alone conditions. While using a factorial design may still be helpful in this latter case, the true benefits of the factorial design are not best represented in this example. Therefore, let's say the experimental design was modified and for 3 different horizontal positions you wanted to display 5 coherence levels resulting in 15 conditions. The above listed code can easily be modified to support this update.

```

<FACTORIAL Name="myFactorialDesign" Levels="3,5" >
  <FACTOR1 Name="horizontalPos">
    <nsLLDots>
      <X>-10,0,10</X>
    </nsLLDots>
  </FACTOR1>
  <FACTOR2 Name="coherenceLvl">
    <nsLLDots>
      <coherence>0,0.25,0.5,0.75,1</coherence>
    </nsLLDots>
  </FACTOR2>
</FACTORIAL>

```

In the above example, the original <FACTORIAL> tag was modified. Notice how the horizontal position and coherence have been separated into different factors within the <FACTORIAL> element. This tells Neurostim that you would like all 5 coherence levels to be presented at a horizontal position of -10, 0 and 10. Therefore, Neurostim is expanding the factorial and setting up all the conditions you need without you having to type out each one. This expansion has been visualized in the below table.

		Factor 1 – Horizontal Position		
		<i>X=-10</i>	<i>X=0</i>	<i>X=10</i>
Factor 2 - Coherence	<i>Coherence=0</i>	X=-10 Coh=0	X=0 Coh=0	X=10 Coh=0
	<i>Coherence=0.25</i>	X=-10 Coh=0.25	X=0 Coh=0.25	X=10 Coh=0.25
	<i>Coherence=0.5</i>	X=-10 Coh=0.5	X=0 Coh=0.5	X=10 Coh=0.5
	<i>Coherence=0.75</i>	X=-10 Coh=0.75	X=0 Coh=0.75	X=10 Coh=0.75
	<i>Coherence=1</i>	X=-10 Coh=1	X=0 Coh=1	X=10 Coh=1

If a <FACTOR3> tag was added with 2 separate values for a parameter then this would create a new dimension of this matrix resulting 30 conditions. This process would continue for each <FACTOR> element that was created.

Each <FACTOR> tag is associated with a particular number of levels indicating the number of manipulations that are necessary for that factor. This also corresponds to how many separate parameter values are listed in each <FACTOR> element. You must tell Neurostim the levels associated with each <FACTOR> element. This is done in the <FACTORIAL> tag. In the nsLLDots example, this lets Neurostim know that <FACTOR1> has 3 levels or parameter values and <FACTOR2> has 5 levels or parameter values resulting in 15 overall conditions.

It is important, though, that all parameters defined within a particular <FACTOR> tag have the appropriate number of values as defined in Levels attribute of the <FACTORIAL> tag. If this is not the case, Neurostim will not run the experiment and you will receive an error message indicating which factorial is not set up correctly.

 Try it out

- F. In your updated nsLLDots experiment file, modify the existing <FACTORIAL> element to have 2 <FACTOR> tags similar to the above example. For this example, though, have the “horizontalPos” factor only use 3 different <coherence> values, i.e. 0, 0.5 and 1. (Hint: Make sure the overall <FACTORIAL> is “aware” of your updates). Try running the experiment.
- G. Now, add a <FACTOR3> tag. In this factor you will be modifying the number of dots used in a condition, i.e. <nrDots>. For your new factorial, set <nrDots> equal to 100 or 500. Try running the experiment. What was the original setting of <nrDots>?

- H. Lastly, let's say you want the center of your area of dots to be at (-10,5), (0,0) and (10,-5) for this factorial. Modify your updated experiment file so that the vertical position can also vary (Hint: You can modify an existing <FACTOR>). Try running the experiment.

Using multiple stimuli within a factor:

It is also possible to list parameters from multiple stimuli within a <FACTOR> tag. Let's say when the dots are presented at different horizontal locations on the screen that you would like for the fixation stimulus to be a different shape. Then the code can be modified as follows:

```
<FACTORIAL Name=" myFactorialDesign " Levels="3,5" >
  <FACTOR1 Name="horizontalPos">
    <nsFixation>
      <shape>LEFT, DOT,RIGHT</shape>
    </nsFixation>
    <nsLLDots>
      <X>-10, 0,10</X>
    </nsLLDots>
  </FACTOR1>
  <FACTOR2 Name="coherenceLvl">
    <nsLLDots>
      <coherence>0,0.25,0.5,0.75,1</coherence>
    </nsLLDots>
  </FACTOR2>
</FACTORIAL>
```

Now, when the horizontal position is on the left of the screen for conditions within this factorial, the fixation stimulus will be an arrow directed left and similarly when the horizontal position is on the right of the screen, the fixation stimulus will be an arrow directed right. When the horizontal position is 0, the fixation stimulus will change to be a small dot.

General Notes:

Using the same stimulus in every condition:

Initially when nsLLDots was setup, the same fixation stimulus was used in every condition. Even though, the default parameters had been set up for nsFixation, nsFixation must be referenced directly in the <CONDITIONS> section of the experiment file. This can be done by including it under every <CONDITION> or <FACTORIAL> tag, but this may be a little tedious. Instead nsFixation can be included by using the <EVERY> tag as shown in the nsLLDots example.

Weighting specific conditions:

Currently, you can add weighting features within a <FACTOR> element so that the conditions within this factor can be presented a particular percentage of time. Let's consider an example adapted from nsLLDots.


```

<FACTORIAL Name=" myFactorialDesign " Levels="2,5" >
  <FACTOR1 Name="horizontalPos">
    <nsLLDots>
      <X>-10, 10</X>
    </nsLLDots>
  </FACTOR1>
  <FACTOR2 Name="coherenceLvl">
    <nsLLDots>
      <coherence>0,0.25,0.5,0.75,1</coherence>
    </nsLLDots>
  </FACTOR2>
</FACTORIAL>

```

For this experiment, we will only be presenting conditions on the left or right side of the screen. Let's say, though, that 80% of the time we wanted to present conditions on the left side of the screen, i.e. $X = -10$. We can easily add this stipulation within the existing code by adding a Factor1Weights attribute to the <FACTORIAL> tag, i.e. <FACTORIAL Name="coherence" Levels="2,5" Factor1Weights="4,1">. If we wanted to weight the coherence values that were being presented we would use Factor2Weights and so on. Please note, the weights must be listed as integers corresponding to the ratio of levels within a factor you wish to present. By adding in a weighting feature, Neurostim will adjust the number of trials for each condition to ensure the ratio is met. For example, with a 4:1 ratio for the horizontal position, all 5 conditions with an X value of -10 would be presented 4 times, while the 5 conditions with an X value of 10 will only be presented once. Therefore, this <FACTORIAL> element will generate 25 trials in Neurostim.

Weighting of conditions can also be done by creating separate <FACTORIAL> or <CONDITION> tags and adjusting how many times a factorial or condition appears in a block. Blocking will be discussed in the next section.

 Try it out

- I. Returning to your experiment file with the updated <FACTORIAL> element, add in a weighting feature such that 50% of the time conditions will be presented at (0,0), 25% of the time conditions will be presented at (-10,5) and the remaining 25% of the time at (10,5).

3. *Determining how conditions will be presented.*

We are now towards the end of the <DESIGN> element and we have defined the stimuli to use in our experiment and listed all the conditions that are needed for this experiment. The last thing to do is determine exactly how we want the conditions to be presented to the subject. This takes us to the <BLOCKS> element.

The <BLOCKS> element will contain all defined blocks within an experiment. Each block is defined by a <BLOCK> tag. Within the <BLOCK> element you can refer to your conditions by the name you defined in the <FACTORIAL> or <CONDITION> tag. Let's use nsLLDots as an example.

```

<BLOCKS Repeats="5">
  <BLOCK Randomize ="WithoutReplacement" Retry ="RANDOMINBLOCK">
    <coherence Repeats="5"/>
  </BLOCK>
  <BLOCK Retry ="IGNORE">
    <left Repeats="2"/>
  </BLOCK>
</BLOCKS>

```

In nsLLDots, there are two different blocks that are used. The first block contains a reference to <coherence>, our factorial. In each tag within this section, i.e. in the <BLOCKS> tag, <BLOCK> tag or within a specific condition/factorial name tag, a Repeats attribute can be used. In nsLLDots, the factorial <coherence> will be repeated 5 times within the first block. Since “coherence” has 3 separate conditions, Block 1 will contain 15 trials that are randomized without replacement, as noted in the Randomize attribute, and will be retried randomly within a block, as noted in the Retry attribute⁵. A trial will be retried if the subject breaks fixation, for example.

Instead of putting the Repeats attribute in the <coherence> tag, it could have been placed in the <BLOCK> tag. Although this would still result in the same total number of trials the presentation would potentially be altered. Instead of randomizing all 15 trials and presenting these 15 in a random order, the 3 conditions within coherence would be randomized and then presented in a block with 3 trials. This method would occur for the amount of repeats listed, i.e. 5 in this example.

The second block references the “left” condition and repeats this condition 2 times. Since the Retry attribute is set to ‘IGNORE’, though, the “left” condition will never be displayed more than the specified number of repeats even if there is a fixation break or other activity that terminates a trial.


Since the <BLOCKS> element contains both the “coherence” and “left” blocks, each block would be presented once and in the order listed. This process would be repeated 5 times.

The methods of randomization that can be used are WithoutReplacement, WithReplacement and Sequential. Randomization can be added in the <BLOCKS> tag to apply a particular type of randomization, except for WithReplacement, to the presentation of the defined blocks. Randomization can also be added in the <BLOCK> tag to apply a particular type of randomization to the presentation of the conditions included within the block. A summary of each type of randomization is listed in the following table.

⁵ The valid values for the Retry attribute are RANDOMINBLOCK, IGNORE and IMMEDIATE.

Type	Valid Tags	General Information
Sequential	<BLOCKS> <BLOCK>	<p>Blocks or conditions will be presented in the order they are listed.</p> <p>This is the default value in the event no randomization attribute has been listed.</p>
WithoutReplacement	<BLOCKS> <BLOCK>	<p>All blocks or conditions will be presented the applicable amount of times. The order they will be presented, though, is random.</p> <p>For example, assume there are 10 unique conditions that are being repeated 5 times. This results in a total of 50 conditions that will be presented during a block. The presentation order for these 50 conditions would be random, but all 50 conditions would be displayed at least once (depending on the setting for the Retry attribute).</p>
WithReplacement	<BLOCK>	<p>A sample of conditions out of the total conditions available will be presented based on the number of repeats. Conditions will be chosen such that the same condition can be chosen multiple times and other conditions may not be shown at all.</p> <p>For example, assume there are 10 unique conditions that are being repeated 5 times. This results in a total of 50 conditions that could be presented during a block. During the experiment, 50 trials would be constructed out of the available 50 conditions. At an extreme, the same condition could be presented 50 times.</p>

Given that randomization is available to be applied across all blocks or within a block, there is a lot of flexibility for how you want to present your conditions.

 Try it out

- J. Update your experiment file so that there is only one <BLOCK> element that includes “coherence” and “left.” Have “coherence” and “left” only repeated once, but set it up so the overall block repeats only 10 times. You can vary other parameters.

CONTROLLERS

Neurostim was developed to allow interaction with a number of other programs and devices. For instance, you may want to test your stimulus on your own desktop, but later, when actually running an experiment, Neurostim should communicate with an Eyelink eyetracker, or with an MRI scanner. To achieve this flexibility, different controllers can be specified in the Experiment file. In the example above, we used the Keyboard controller – it allows you to select a particular condition with a key press and is intended only for debugging at your own workstation. The current controllers and their quirks are listed here. Note that every Controller ends a trial when a trial has reached the duration specified as <design><trialduration> in the Experiment file.

Name	Goal	Special Controls
Keyboard	Graphical inspection and debugging of your stimulus before using it in a real experiment with one of the other controllers.	Press one of the number keys to select conditions 1-9. Press 'n' to show the next condition. (What is "next" depends on the setting of the randomization mode in the experiment file).
Eyelink	Allows Neurostim to retrieve Eye Position information from the EyeLink II camera, to check fixation, pursuit, or other behavioral requirements .	Use Ctrl-s to setup the Eye tracker camera. Once Ctrl-s has been pressed, you have complete control over the Eyelink computer from the Neurostim keyboard.
Stimulus	Run a behavioral experiment, without external devices.	After a done() command has been issued, the controller selects a new condition to run in the next

	<p>This controller leaves the decision about when to start a new trial up to the stimulus itself. The (derived) stimulus class does this by checking some state (which could for instance be a user response) and then issues the <code>done()</code> command.</p>	<p>trial. The selection mechanism is specified in the experiment file</p>
Cortex	<p>The Cortex controller is an alternative Receive program for Dual Computer Cortex (DOS: v 5.9.5 and all Windows versions). Communication with Cortex is via the usual Serial port, or via Ethernet (TCP/IP). The Cortex Timing File programming language has been extended to select specific frames in the animation and to obtain parameter values that are set in Neurostim.</p>	<p>After starting the Neurostim program, the Cortex Send program takes over the control and starts the appropriate stimuli.</p> <p>The Cortex Send program selects the condition to run.</p> <p>A number of timing file commands have been defined to allow two-way communication between the Cortex Send program and the Neurostim Receive program.</p>
Nabeda	<p>This controller interacts via the serial (COM) and parallel ports (LPT) with the Nabeda program for electrophysiological recording.</p>	<p>Stimulation starts when the LPT-pin is activated after Nabeda has sent at least one valid "stimulus:configuration" string and the first condition number via COM.</p> <p>Strings via COM are interpreted as "stimulus:configuration", numbers >0 are taken as the next condition nr, number 0 resets Neurostim and starts a new logfile.</p> <p>Conditions are determined by Nabeda, and then sent over to the Neurostim program.</p>

ET50Control	Enables Neurostim to be used as slave for the calibration routine of the Thomas Recording Eyetracker (ET49/50). A specially programmed stimulus is needed to make this work.	<p>Stimulus-start is determined by the commands of the ET-system that are received via COM.</p> <p>Only one condition is used. Target position is controlled via <code>remoteCommand</code>.</p> <p>Communication via serial line follows the protocol described in the ET49 manual. The stimulus displaying the calibration target has to understand a <code>remoteCommand</code> w/ <code>msgId 0</code> and two float parameters (x- and y-coordinate).</p> <p>In the "Controller"-section of your experiment file you additionally have to set the following five variables: <code>REGIONX1</code>, <code>REGIONX2</code>, <code>REGIONY1</code>, <code>REGIONY2</code> and <code>TARGETSTIMULUS</code>. The <code>REGION--</code> parameters define the desired calibration rectangle, <code>TARGETSTIMULUS</code> has to be set to the name of the stimulus that should receive the <code>remoteCommand</code>.</p>
--------------------	--	---

CONTROLLING THE CONTROLLER

Although some controllers have specific keypresses they respond to, most share at least the following functionality:

Ctrl-q

Quits the application. All data are written to file first.

Ctrl-d

Toggle Debug mode. This writes additional information to the command line and to the screen.

Ctrl-s

Toggle Step/Run mode. In step mode, frames can be advanced one at a time by pressing the space bar.

Ctrl-c

Toggle Cursor visibility and show the current coordinates of the cursor on the screen. In order for this to be set <ALLOWRFCHANGE> must be set to 1.

Ctrl-f

Toggle Full screen/Windowed mode. More than one press may be required to get real full screen.

Ctrl-e

Toggle Erase/Keep mode. In keep mode, the successive frames are not erased. This can be useful to trace out all the positions on the screen that your stimulus reaches.

Ctrl-l

Display a list of current stimuli and their parameter values on the command prompt.

Spacebar

Toggle freeze/run. This only works in Step mode.

FINAL EXERCISES

This concludes the tour of a Neurostim experiment. To experiment with the various settings, copy the directory nsLLDots to a temporary directory or your home directory so that you can change the experiment and stimulus files. Here are a few things to try:



1. Modify your experiment file to include all of the following specifications:
 - a. A controller other than “Keyboard” (e.g. Stimulus if you have no external devices to connect to).
 - b. Set up a block where you have the “oblique”⁶ and “spiral”⁷ stimuli presented at 4 different locations on the screen, i.e. upper left, lower left, upper right and lower right. At each location present the stimuli at 5 different coherence values, i.e. 0, 0.25, 0.5, 0.75 and 1. You will also want the “oblique” stimulus to appear twice as often as the “spiral” stimulus.
 - c. Set up a separate block that only refers to the “oblique” stimulus. Include all conditions from part b, but also add a condition where the stimulus is presented in the center. This time though, only have 0 and 1 coherence values presented at this location. The conditions in the center should be presented 4 times in the block.
 - d. Repeat all blocks 3 times and randomize using WithoutReplacement.
2. Design an experiment in which the subject is to make a rapid eye movement from one dot to another, while the nsLLDots “oblique” stimulus is visible all the time. Tip: you’ll need two nsFixation stimuli and use <on> and <off> judiciously.

⁶ For simplicity, the stimuli are referred to by the name of the stimulus file. You can modify the stimulus to be whatever name you prefer.

⁷ If you have not already added a new stimulus referring to the spiral.nml stimulus file, please see Try it Out exercise C.

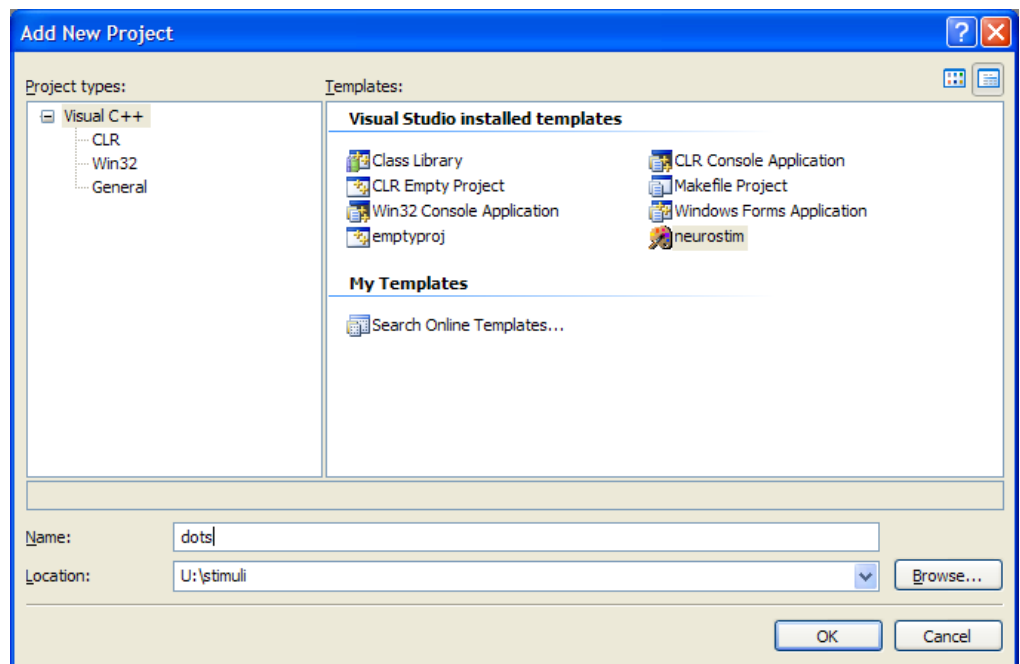
3 Using the Visual Studio Stimulus Wizard

SHOULD I READ THIS?

-  You use Visual Studio C++
-  You want to make a new stimulus instead of using an existing stimulus.

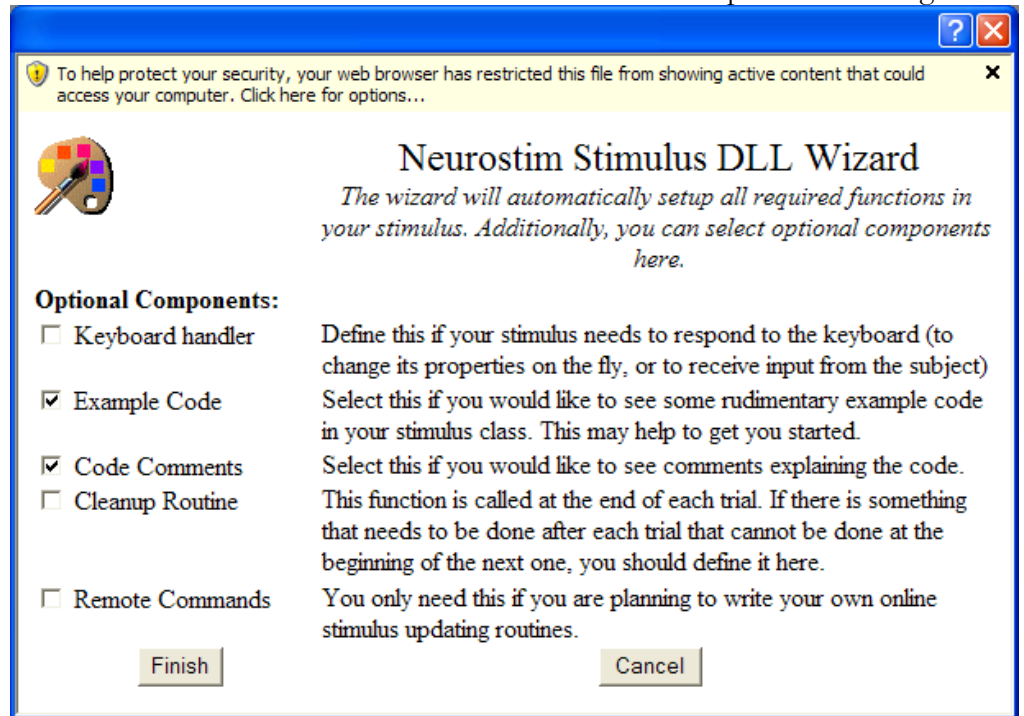
The previous chapter showed you how to use Neurostim when someone gives you a stimulus (DLL) and tells you which parameters can be configured in that stimulus (in the stimulus .nml file). This chapter tells you how to use Microsoft Visual C++ Express to create your own stimuli in C++.

1. Create a folder where your stimuli will be developed. I will assume this is `u:\stimuli`
2. Copy the file `C:\Program Files\Neurostim\make\Stimulus Wizard\stimuli.sln` to this directory.
3. Double click the `stimuli.sln` file. This should open the Visual Studio development environment. On the left, you see Solution 'stimuli' (0 projects). This means that this solution file does not have any stimulus projects yet. Let's add one.
4. Go to `File | Add | New Project`. This will open the following dialog box:

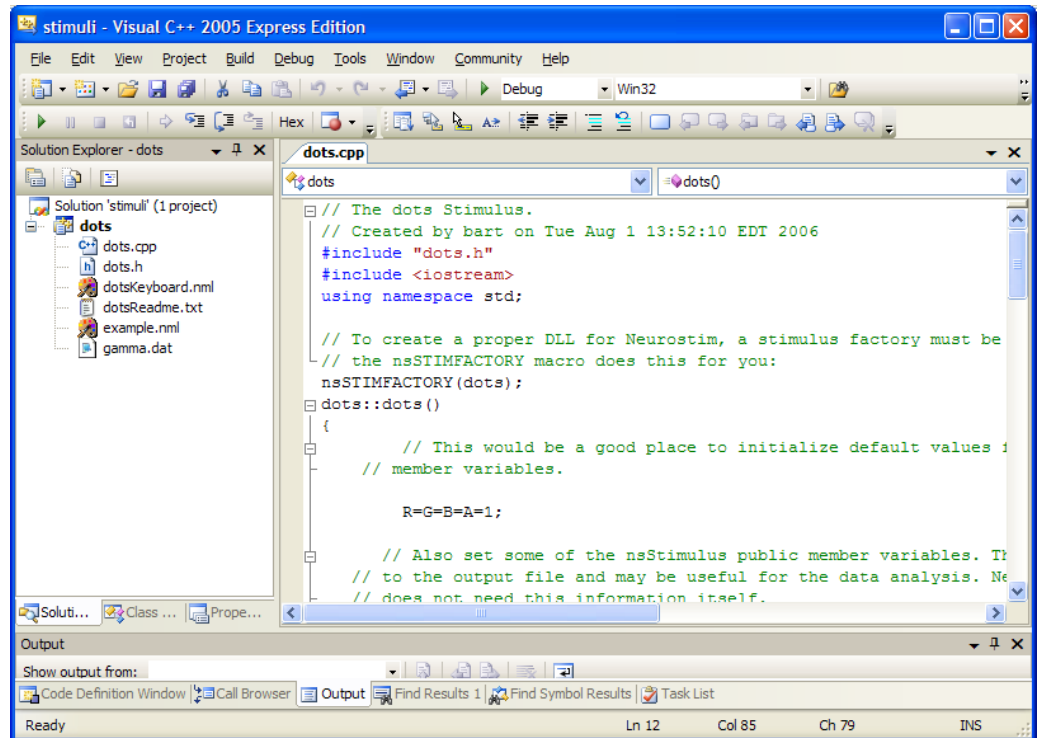


5. Make sure to select Visual C++ in the left window, and Neurostim under Visual Studio Installed Templates. In the Name box, type the name of your stimulus. Don't use spaces or other non-standard characters in this name. In this example I chose to name the stimulus 'dots'. This will be the name of the project, but also of the DLL file that is created eventually.
6. Make sure that the Location refers to the directory in which you copied your Stimuli.sln file. (`U:\stimuli`). Then press OK.

7. A new window appears. Depending on your security settings, the yellow warning bar at the top may be present. Right click it and select 'Allow Blocked Content' to allow the active content to run. Accept the warning.





8. Select the options that you want to keep. For beginners, the default settings will be fine. More advanced users may want to uncheck the Example Code and Code Comments. Press Finish.
9. The Wizard creates a .cpp and a .h file with the C++ code necessary for your own stimulus. Additionally, it will create an example experiment file and a stimulus file and finally a readme.txt with some information on these files. All these files are referenced in the Solution Explorer and you can click the file to open it in the editor.



10. To create the stimulus DLL file, select Build|Build Solution. This will create the DLL file and put it in the ./dlls directory. If the Build is successful, you can do a test run.
11. Make sure that your new project is the Startup Project by right clicking the project in the Solution Explorer and selecting 'Set as Startup Project'. To run an experiment with your new stimulus, press F5 or select Debug|Start Debugging. In this example this will use the dotsKeyboard.nml experiment file.
12. Because the dotsKeyboard.nml file specifies a Keyboard controller, you start the experiment by pressing Enter in the Neurostim window and can select the conditions with the number keys on the keyboard.
13. Now that you have the basic infrastructure of the Neurostim stimulus, you can start adding your own C++ and OpenGL code.
14. Once you are ready to run a real experiment, copy the DLL file, your experiment files and the stimulus files to a new directory. That is all that is needed to run the experiment.
15. To add a new stimulus, open the stimulus.sln solution file, select File|Add|New Project and start again from step 4.

4 Program your own Stimulus

SHOULD I READ THIS?

-  You know some C++ .
 -  You want to make a new stimulus instead of using an existing stimulus.
-

Every stimulus in Neurostim derives from the `nsStimulus` class. This class has a few virtual functions that a derived class should implement and a number of member variables that will have appropriate values at runtime and which you can use to modify your stimulus on-the-fly. This chapter gives a brief overview of the relevant member functions and variables of `nsStimulus`, followed by a detailed walk-through of how to create a stimulus.

For troubleshooting C++ code it helps to recognize the following prefixes:

ns: all Neurostim library objects and functions start with the 'ns' prefix. For instance, the most important base class (from which all your code will derive) is the `nsStimulus` class. The library exports no objects beyond objects starting with 'ns', and those in the namespace `ns`. Hence as long as you refrain from using `ns` prefixes, there will be no namespace conflict.

gl: OpenGL commands all start with the `gl` prefix ; they are documented in the Visual Studio help files.

Al: OpenAL commands start with 'al'.

THE NSSTIMULUS CLASS

The mother of all stimuli is the `nsStimulus` Class. The important parts of its declaration are shown here (from `nsStimulus.h`):

```
class NS_API nsStimulus:
    public nsKbListener,
    public nsMouseListener
{
public:
    virtual ~nsStimulus();
    nsStimulus();
    virtual void setup();
    virtual void move();
    virtual void draw();
    virtual void cleanUp();

protected:
    virtual void keyboard(const unsigned char key, const int x, const int y);
    virtual void mouseLeftUp(int x, int y);
    int trial;
    int frame;
    double time;
    int condition;

    GLdouble X,Y,Z;
    GLdouble phiX,phiY,phiZ;
    GLdouble scaleX,scaleY,scaleZ;
    std::string name;
    std::string rcsVersion;
};
```

These member functions and member variables form the basic interface that you will use to create your own stimulus. Let's look at them in turn.

Member Functions	
<code>void setup();</code>	
	The program calls this function for each stimulus before the start of a new trial. Usually this function assigns new values to (derived-class specific) variables, initializes graphical elements that remain unchanged during a trial and performs calculations that need to be done only once during a trial.
<code>void move();</code>	
	This function is called every frame. If you specify a framerate of 60hz, for instance, this function is called every 16.7ms. You should use this function to calculate parameters that change in real time. For instance, for a moving stimulus, you would calculate the new position here and put this new value in a class member variable. The internal integer variable 'frame' should be used to determine in which frame the execution currently is. Avoid unnecessary calculations in this function, as it must finish its work within a few milliseconds. If the function takes too long, the program will complain with "Neurostim cannot maintain the xxHz framerate". This is a serious warning as it implies, for instance, that the changes you wanted to take place in 16.7ms actually take at least 33.3ms (thus reducing speed by half).
<code>void draw();</code>	
	Once the <code>move()</code> function has completed (for all active stimuli), the program calls the <code>draw()</code> function for each stimulus. This function contains the OpenGL drawing code and uses member variables that may have been changed in <code>setup()</code> or <code>move()</code> . Only after all the stimuli have been drawn to the back buffer are the buffers swapped. This swap occurs on the vertical refresh of the monitor; hence all the changed or new elements of a frame appear at the same time (as far as this is possible on a CRT).
<code>void cleanUp();</code>	
	This function is optional; if defined, it will be called at the end of each trial.
<code>void keyboard(const unsigned char key, const int x, const int y)</code>	
	This function is optional. If defined, it will be called for every key-press. You can use this to record subjects' responses or to change a stimulus on-the-fly.
<code>void mouseLeftUp(int x, int y)</code>	
	This function is called when the left mouse button is being released. In

	<p>other words, at the end of a click. This is the function used to handle left clicks, it is enabled per default. There are equivalent functions for right, and middle clicks.</p> <p>If a user decides to modify this function, there may be other consequences impacting other Neurostim functionality. For example, this may interfere with using Ctrl-c and being able to move the stimulus to the location of the cursor by left-clicking. The stimulus will do whatever is defined by mouseLeftUp and will not move in response to a left click.</p>
nsStimulus Member Variables	
	<p>Apart from the virtual member functions that need to be redefined by your code, there are some nsStimulus member variables that are useful in your derived class.</p> <p>Configurable parameters are parameters that can be specified in your .nml stimulus configuration file and will automatically be applied to your stimulus.</p> <p>Read only parameters are parameters that the controller assigns; you should not change these values, but they can be used in your code.</p> <p>Bookkeeping parameters are optional; you can set them in the constructor of your stimulus class to write extra information to the output file.</p>
double X,Y,Z	Configurable
	These doubles specify the position of your stimulus on the screen. Whatever their value, Neurostim will put the stimulus there (this could be outside the visible part of your screen). Z is the depth coordinate of three-dimensional stimuli
double phiX,phiY, phiZ	Configurable
	Angles of rotation around the X, Y, Z axes. These default to 0, but can be set to arbitrary values to rotate your object.
double scaleX,scaleY,scaleZ	Configurable
	Scaling factors for the three dimensions. Default to 1, but can be set [0, 1] to scale your object.
double on, off	Configurable
	The milliseconds after the start of a trial when a stimulus is switched on and off. By default, these values are 0, Inf. In other words, the stimulus is visible throughout the trial. Note that the onset time is inevitably imprecise because stimuli can only become visible at the start of a refresh. Only when on and off times are multiples of the framerate will

	you get perfect timing.
double frame	Read Only
	The first frame that becomes visible is frame=1. 'frame' is the number of times that 'move()' has been called for this stimulus.
double time	Read Only
	The time in milliseconds since the trial started.
int trial	Read Only
	The current trial number.
int condition	Read Only
	Tells you which condition this is. You should not edit this variable, the controller does that for you.
string name	Bookkeeping
	A name for this stimulus. Will be written to the data file.
string rcsVersion	Bookkeeping
	Use this string to put automatic version control system tags into the data file.

STEP-BY-STEP

DECLARE YOUR CLASS

The first step in creating your own stimulus will be to derive a class from the nsStimulus class. Let's consider a stimulus that consists of a simple dot as an example. Say we want to specify the position (x,y) , color (R,G,B) and size of the dot. This declaration will go into the simpleDot.h header file. We did not need to define member variables for x,y, because these have already been defined in the nsStimulus class (see above). I only added the size member variable that will be used to specify the size of the dot and the R,G,B variables to specify the color.

```
#include "nsStimulus.h"
class simpleDot : public nsStimulus {
public:
    simpleDot();           // Constructor virtual
    ~simpleDot();           // Destructor
    void draw();           // Overload the draw function
    void setup();          // Overload the setup function
    void move();           // Overload the move function

protected:
    GLfloat size;          // Variable to specify size
    GLfloat R,G,B,A;       // Variables to specify color.
};
```

THE CONSTRUCTOR

In the constructor all variables could be given some initial value, but there are no Neurostim specific requirements.

THE SETUP() FUNCTION

In the setup function, you establish the relationship between the XML tags in the configuration file and the variables in the class.

```
void nsSimpleDot::setup(){
  R = getParmFloat("R"); // <R> defines the variable R.
  G = getParmFloat("G");
  B = getParmFloat("B");
  size = getParmFloat("size");
}
```

As discussed above, this function is called once before every trial. First, note that there is no line specifying that x and y should be read. Because x and y are members of the nsStimulus class, they are always read automatically (if present in your stimulus file, see below). Second, note that this code does not specify which condition is currently read! The condition to read is determined internally, the method to choose the order of conditions can be specified in the experiment file.

THE STIMULUS FILE

Each stimulus will have a number of parameters that define it; the values of these parameters can be read from stimulus configuration files. Stimulus configuration files have the extension .nml, for Neurostim Markup Language and are simple XML text files. The complete format of these files is explained on page 51.

Each stimulus class (including those defined by the user) has its own directory with configuration files. Assume we wish to use a red dot in the one condition, a green in the second condition. The configuration file looks like:

```
<?xml version="1.0" ?>
<Stimulus Name = "simpleDot" Config="RedAndGreen">
  <!-- This is a simple XML format configuration file for Neurostim. -->
  <Condition Name="Red">
    <X>3</X>
    <Y>4</Y>
    <size> 5 </size>
    <R>1</R>
    <G>0</G>
    <B>0</B>
  </Condition>
  <Condition Name ="Green">
    <size> 5 </size>
    <R>0</R>
    <G>1</G>
    <B>0</B>
  </Condition>
</Stimulus>
```

This names the stimulus "simpleDot" and this particular configuration "RedAndGreen". In the first condition, the dot is presented at position (3,4) and will have the color red (R, G, B = 1,0,0). In condition 2, the position is the same but the

color is now green. You can use standard XML comments to add documentation to your file.

THE DRAW() FUNCTION

The actual drawing and hence most of the OpenGL library calls are done in the `draw()` function. This function is called every frame, and it should only be used to draw the stimulus with the properties as specified by its member variables' current values. For our simple dot this means:

```
void nsSimpleDots::draw(){
    glColor4d(R,G,B,1); // Set the colour
    glPointSize(size); // Set the size
    glEnable(GL_POINT_SMOOTH); // Draw smooth points
    glBegin(GL_POINTS);
    glVertex2d(0,0); // Draw at position 0,0.
    glEnd();
}
```

This code snippet only contains OpenGL code (all prefixed with `gl`). Note two things. First there is no reference whatsoever to the current condition or frame that we are in. The `draw()` function should always assume that the member variables (in this case `R,G,B,size`) have the appropriate values for the current frame. This is trivial if these values do not change in a trial (in that case they are set in `setup()` and never changed). If the values change over the course of a trial (to implement motion for instance), this change takes place in the `move()` function (and nowhere else!).

A second thing to note is that I did not use the `x,y` member variables. The reason for this is that this is done implicitly: drawing is always relative to the `x,y` of the current stimulus. Hence the call `glVertex2d(0,0)` draws relative to the `x,y` of the current stimulus. Given that `x,y` are 3,4 in our configuration file, this means that the `simpleDot` will be drawn at position (3,4) relative to the center of the screen.

Note

A note about units: they are completely arbitrary and depend only on what you tell Neurostim about the size of your screen (in the experiment file). If you specify the screen size in meters there, your code can be programmed in meters. If you specify pixels in the configuration file, you can use pixels in the code. For flexible use in different computer setups, I'd recommend programming stimuli in physical units (i.e. meters or centimeters): that way you'll only have to change the experiment file when you move to a new monitor or a different monitor resolution.

Tip

The draw function is time-critical: try to minimize your code and do no lengthy calculations here. If a calculation really needs to be done only once per trial, do it in `setup()`, not in every `move()` or `draw()`. To hide your stimulus for some frames (at the start of the trial, for instance). Simply put: `if (time<someDuration) {return;}` at the start

of your draw function: only the background will be shown.

THE MOVE() FUNCTION

Animation or more generally a change in the stimulus that occurs within a single trial is done in the `move()` function. This function is called before every frame. Let's say we want to move our simple dot horizontally over the screen in a sinusoidal manner. This could be implemented with the following few lines of code:

```
void nsSimpleDot::move(){  
    X = sin(time); // Oscillate X with an amplitude of 1.  
}
```



Note that I used the `nsStimulus` derived member variables `time` to arrive at the current time since the start of the trial in seconds. This is merely a convenience that will allow you to "think in seconds" rather than in frames. Moreover, it would allow you to define a speed in seconds (in the configuration file) and use that parameter here. Of course you can just as well define all speeds and other changes in units of frames. In that case, you use the `frame` member variable to determine where in the trial you are and change the physical characteristics of your stimulus accordingly. The first time the `move()` function is called, the `frame` will be 1. Hence the first view of the stimulus will be the one that corresponds to `frame=1`.

The `move()` function does not change the display: it merely sets the member variables to new values. Behind the scenes, the Neurostim controller will make sure that the `draw()` function is called with these new values, hence at the next monitor refresh; the stimulus will appear with the new parameter values.

5 Interactive Stimuli

The previous chapter discussed how to make a basic stimulus, and how its properties are defined in stimulus files. In most behavioral experiments you will want to collect some information on what the subjects perceive and write this information to the data file. In electrophysiological experiments you may want to change the stimulus on the fly depending on an analysis of the neural response. This chapter discusses the methods that Neurostim offers to allow you to do this.

SHOULD I READ THIS?

-  You know how to make a basic stimulus
 -  You want to collect subject responses, or monitor their eye movements.
-

KEYBOARD

A simple way to collect responses is to ask subjects to press a key on a standard keyboard.

To collect these, you define the `keyboard(const unsigned char key, const int x, const int y)` function.

To continue the `simpleDot` example, let's assume the subjects task is to position the dot in the center of the screen (assuming the `move()` function has been disabled). Define the keyboard function:

```
void simpleDot::keyboard(const unsigned char key, const int x, const int y){
    const double step = 0.1;
    switch (key) {
        case 'l': //Left
            X -= step;
            break;
        case 'r': //Right
            X += step;
            break;
        case 'u': //Up
            Y += step;
            break;
        case 'd': //Down
            Y -= step;
            break;
    }
}
```

With this keyboard function, pressing the 'l','r','u','d' keys simply changes the value for the appropriate spatial member variable (x,y). On the next display (frame), this will result in a new position of the dot. (See page 38 for how to record these key-presses and page 40 how to start a new trial after a key-press).

MOUSE

The mouse can also be used to retrieve subject responses. To use this in your own classes, you first need to derive your class from the `nsMouseListener` class, then define an appropriate handler function (e.g. `mouseLeftUp()`) – to respond to the release of the mouse button) and finally tell the stimulus to pay attention to this particular mouse event (in the constructor, or by using the `listenTo()` function).

For instance, let's create a `simpleDot` stimulus class that positions the stimulus where the left mouse button is clicked.

```

#include "nsStimulus.h"
class simpleDot : public nsStimulus,
public nsMouseListener
{
    public:
        simpleDot();          // Constructor virtual
        ...
        void mouseLeftUp(int mouseX, int mouseY);
protected:
        ...
        ...
};

// The constructor sets up the stimulus to listen to left
// mouse clicks, but ignore all others.
simpleDot::simpleDot :
    nsStimulus(),
    nsMouseListener(true,false,false,false,false,false,false)
{
}

// This function is called when the stimulus is active and the // left mouse button is released.
void simpleDot::mouseLeftUp(int mouseX, int mouseY){
    X = mouseX;
    Y = mouseY;
}

```

This simple example uses the call to the `nsMouseListener` constructor to tell the stimulus to listen to a particular mouse event (in this case releasing the left mouse button: `LeftUp`). Alternatively, this can be done by calling the `listenTo(nsLEFTUP)`, or you can use its functional opposite `ignore(nsLEFTUP)` to ignore a certain mouse event. These functions are useful when you have a stimulus that should sometimes respond to the left and other times to the right mouse button. A complete list of mouse handler functions and their control parameters is given in the table below.

Function Prototype	Activate with <code>listenTo()</code>
<code>void mouseLeftUp(int x, int y);</code>	<code>nsLEFTUP</code>
<code>void mouseLeftDown(int x, int y);</code>	<code>nsLEFTDOWN</code>
<code>void mouseRightUp(int x, int y);</code>	<code>nsRIGHTUP</code>
<code>void mouseRightDown(int x, int y);</code>	<code>nsRIGHTDOWN</code>
<code>void mouseMiddleUp(int x, int y);</code>	<code>nsMIDDLEUP</code>
<code>void mouseMiddleDown(int x, int y);</code>	<code>nsMIDDLEDOWN</code>
<code>void mouseMotion(int x, int y);</code>	<code>nsMOUSEMOTION</code>

Note that the `mouseMotion()` function is called for each mouse movement that is detected by the hardware. This will generate many calls to the `mouseMotion()` function each frame and, if any serious processing is done in the `mouseMotion()` function, this can seriously affect the on-time processing of other events, or the animation of the stimulus.

Also, the `x` and `y` values in the mouse handlers are the position of the cursor when the mouse event occurred. These positions are given in pixels (from the top-left of the screen). Use the `worldCoordinates()` function to convert these values into coordinates on the screen.

EYE MOVEMENTS

When connected to an Eye Tracker (e.g. Eyelink) you can instruct Neurostim to take certain actions on the basis of a subject's eye movements.

For instance, a stimulus can be made the fixation target. You tell Neurostim when the stimulus should be fixated, if the subject is not fixating at that time, the trial is terminated and a warning message can be displayed (or a sound played).

You define required behavior with respect to a stimulus by adding the `<behave></behave>` property in the stimulus file. This parameter works just like any other stimulus parameters; it can be specified once and inherited by other conditions, or it can be (partially) redefined for each condition. For these predefined behaviors no C++ programming is required.

FIXATION

Simply specify the time (in milliseconds) and the window within which this object must be fixated. Note that the fixation check uses the data members X and Y to determine where the object is.

In the example below, the nsFixation stimulus will be visible from trial onset `<on>=0` for 5000 ms (`<off>`). The config file specifies that it must be fixated within 2000 ms of trial onset, and until 3500 ms after trial onset, within a window that is 2 horizontally and 1 vertically. If the subject breaks fixation during the [2000; 3500] interval, a breakFixation event is generated and the trial will be terminated.

```
<Stimulus Name="nsFixation" Config=" ">
  <Condition AllowRfChange="0">
    <Enable>1</Enable>
    <X>0</X>
    <Y>0</Y>
    <R>1</R>
    <G>0</G>
    <B>0</B>
    <size>5</size>
    <on>0</on>
    <off>5000</off>
    <!-- Specify required behavior -->
    <behave>
      <fixate>
        <from>      2000      </from>
        <to>        3500      </to>
        <xwindow>    2        </xwindow>
        <ywindow>    1        </ywindow>
      </fixate>
    </behave>
  </Condition>
</Stimulus>
```

ON/OFF

You can also switch a stimulus ON or OFF contingent on some behavioral event.

For instance, assume you have an experiment in which the nsFixation stimulus is loaded as in the previous example. Neurostim will mark the time that the subject

fixates the nsFixation object, and now you want to switch on a different stimulus 1000 ms after this first fixation and keep it on for a total of 1000 ms. You would use the following Stimulus configuration:

```
<Stimulus Name="nsLLDots" Config="Sample">
  <Condition>
    ....
    other parameters, removed for clarity...
    ....
    <on>
      <fixation>1000</fixation>
    </on>
    <off>
      <fixation>2000</fixation>
    </off>
  </Condition>
</Stimulus>
```

Defining times relative to fixation can also be useful in experiments in which multiple successive eye movements have to be made. For instance, fixate one object first, then show a second object 500ms later and fixate this object 500ms after that.

For the second target object you would use something like:

```
<on>
  <fixation>500</fixation>
</on>
<behave>
  <fixate>
    <from><fixation>1000</fixation></from>
    ...
    ...
  </fixate>
</behave>
```

COMPLEX EYE MOVEMENTS

The predefined behaviors (fixate, on, off) allow you to define a simple behavioral paradigm quite easily (For instance, a passive stimulus paradigm with a fixation stimulus appearing, then a second stimulus appearing once fixation has been achieved, followed by reward for correct fixation, only takes a few lines).

However, if you need more flexibility and want to change your stimuli on the fly on the basis of eye position or fixation data, you do need to program this into your stimulus. The Neurostim library, however, provides some nsStimulus member functions to make this easier.

The function `void getBehavior(string type, vector<double>& values)` retrieves information about the current behavior (such as eye position). The list below shows the possible values for the `type` parameter and what kind of information this will give you.

- **FIXATION**
The function returns true if the subject is fixating the current stimulus (the stimulus in which the `getBehavior` function is called). The time at which the current fixation started and stopped is returned in the values vector.
- **FIRSTFIXATION**
Returns true if currently fixating, and the time at which the first fixation of this trial occurred is returned as the only entry in the values vector.
- **FIXATIONSTIMNR**
Returns true if some stimulus is currently being fixated. The unique ID of the stimulus (an integer) is returned in the vector of values.
- **GAZE**
Returns true if currently fixating any stimulus. The current gaze position (x, y) is returned in the values vector.

Once you have assessed the current behavioral state, you may want to change your stimulus. Of course you can just change any of the member variables of your stimulus to change the stimulus. Note, however, that such changes are not logged to the output file and that you do not have such direct access to some internal variables (such as for instance the number specified as `<behave><fixate><from>` in the configuration file). Any variable that is set in the stimulus configuration file can be set with the `void overrule(double value, std::string name)` function. For instance, in a saccade paradigm you may require an initial fixation of 500ms of an object, and then jump the object to a new position and require fixation at the new position. The first fixation requirement can be specified in the stimulus configuration file:

```
<behave>
    <fixate>
        <from>200</from>
        <to>700</to>
    </fixate>
</behave>
```

In your stimulus code you would then need to check whether the current time is 500 ms after first fixation:

```
getBehavior("FIRSTFIXATION", fixTime)
if (time>fixTime[0]+500 && stimulusMoved){
    // OK. Fixated long enough;
    // Require fixation 250ms from "now" (i.e. the subject
    // gets 250ms to move his eyes) overrule(time+250,"behave:fixate:from");
    overrule(time+750,"behave:fixate:to")
    // Move the stimulus
    X = X+10;
    Y = Y+10;
    stimulusMoved =true;
}
```

This code snippet would go in your `move()` function where it gets checked before every frame.

REWARD

Neurostim can “reward” subjects in various ways,. You specify what kind of reward will be provided at the end of a trial. Because reward is not specific to a single stimulus, but rather to the whole set of stimuli, this is specified for the whole experiment in the experiment file. Reward can be given in one or more modalities: **sound** (a tone played on the computer speakers), **liquid** (a drop of juice), and **message** (text on the screen). In the `<reward>` property, you must specify which specific reward is given for each of the behavioral states.

For instance, in the example below, a `<fixationbreak>` state is rewarded with a 200Hz tone, no liquid, and a message that says "Fixation Break".

```
<Experiment>
<Controller Name="Eyelink">
  <Behavior>
    <reward>
      <sound>
        <correct>800</correct>
        <incorrect>100</incorrect>
        <fixation>400</fixation>
        <fixationbreak>200</fixationbreak>
      </sound>
      <liquid>
        <correct>200</correct>
        <incorrect>200</incorrect>
        <fixation>80</fixation>
        <fixationbreak>0</fixationbreak>
      </liquid>
      <message>
        <correct>Correct Answer</correct>
        <incorrect>InCorrect Answer</incorrect>
        <fixation>Fixation OK</fixation>
        <fixationbreak>Fixation Break</fixationbreak>
      </message>
    </reward>
  </Behavior>
  ...
  ...
</Experiment>
```

WRITING RESPONSES TO FILE

Neurostim logs everything that happens during an experiment by generating so-called events. An event in Neurostim is anything that may be relevant for later data analysis. Examples are when the trial started, when a stimulus is turned on or off, when the configuration data is loaded, when a frame is missed, etc. But it also includes the values to which parameters were set. All of those events are logged automatically by Neurostim.

To record subject responses, you generate events using the family of `write()` functions in your code. These user events automatically contain the time at which they occurred,

the trial, the condition, and the stimulus that generated the event. On top of that you can add information that you want to store in the file.

If you want to store 5 or fewer values with a particular event, use the function

```
write(string tag,double a,double b,double c,double d,double e);
```

The tag is a name for the event that you choose and the 5 doubles are the information that you want to store. For instance, if the subject pressed the ‘x’ key in response to some stimulus you could use:

```
write("xKey")
```

just to collect that information. If this paradigm includes some kind of randomization and you need to store which random number was used for this particular trial you would use:

```
write("xKey",randomNumber)
```

If more than five numerical elements need to be stored, you can use a write function that takes a vector of doubles as its argument: all doubles will be written to file.

```
void write(string tag, vector<double> values);
```

All events end up in the Neurostim output data file (.nml) which uses a somewhat readable XML-format. The @Spikes Matlab class can read these files and provides easy access to all events and their data.

A note on tags: you can use any string as a tag, but be aware that stimulus variables are also written to file as events. Therefore if you use “X” as a tag in a write() function, it may be difficult to extract its values from the datafile without getting the data for the position “X”. As a general rule, therefore, use descriptive strings for your write() functions that are different from the stimulus parameters.

6 Experiment Design

The <Design> section of the experiment file specifies how the experiment is run: how long a trial is, how conditions are ordered, how many repeats are run. In this section you also specify a name for the experiment (<Paradigm>), which is used to construct the output file name. The global.nml file in c:\program files\neurostim\settings has some comments on the definition of the <Design> section's parameters.

In many experiments a single trial simply runs until it reaches the <TrialDuration> ; then Neurostim chooses a new condition following the <Randomization> and <Order> instruction and, after the <ITI> starts the next trial. This continues until all conditions have been shown <Repeats> times. Such a simple design works well for a paradigm in which the subject merely stares at the screen and no interaction with the stimuli is expected (e.g. some fMRI, and electrophysiological experiments).

A first level of experiment/subject interaction is achieved by entering eye movement requirements for a stimulus (e.g. fixation). In this case, if the subject does not meet the requirements (e.g. breaks fixation), the trial is terminated and a new trial is started. The <Retry> parameter specifies whether a condition that fails is retried immediately, or randomly at some later point.

COMPLETE CONTROL

A more advanced level of experiment/subject interaction requires some programming in the stimulus code. The function that allows you to terminate a trial early is the done() function.

With this you can, for instance, terminate a trial as soon as a subject presses a certain key (e.g. once the subject has answered the particular question that is asked in a trial, although this can also be achieved with the NAFC specification above.) Typically, the done() function will be called in the keyboard handler, after the subject has pressed a certain key. For instance:

```
void simpleDot::keyboard(...){
    switch (key) {
        case 'l': //The subject reporst 'Left'
            // Write an Answer event with the value -1 (=left)
            write("answer",-1);
            // Terminate the trial
            done();
            break;
        case 'r': //Right
            // Write an Answer event with the value +1 (=right)
            write("answer",+1);
            // Terminate the trial
            done();
            break;
    }
}
```

In general, the `done()` function can be called anywhere in the stimulus code; in response to changes in eye-position, after a sequence of key-presses, etc.

STAIRCASE

For rapid estimation of thresholds, stimulus parameters can be varied in a staircase manner. (For an introduction to such methods, see Treutwein, Vision Research, 1995). In brief, the parameter of interest is changed according to some rules applied to successive subject responses. The easiest example is the 50% threshold staircase; for every correct answer the parameter of interest is decreased (made more difficult), for every incorrect answer the parameter is increased (made easier).

Typically the changes in parameter (the steps) will be large in the beginning and are gradually reduced once more responses have been collected.

To implement this in Neurostim you first need to specify in the stimulus configuration file that a particular parameter will be staircased. This is done by specifying XML attributes for the parameter. The attributes are:

- `Type`
Currently only “Staircase”.
- `LowerLimit`
The lower limit of the value of the parameter.
- `UpperLimit`
The upper limit of the value of the parameter
- `InitialStep`
The initial step of the staircase.
- `StepDecrease`
If step size is adapted, this is the factor by which the step is multiplied every adaptation step.
- `MaxRuns`
Stop the staircase after this many runs. Not currently used.
- `Target`
Which performance level: 50%, 70.7%, 79.4% 29.3%
- `StepMode`
When to adapt the step size.
 - `PERTRIAL` – steps change after each trial
 - `PERRUN` - steps change after a “run”

- FIXED - steps always stay at initialStep

For instance let's define `<X>` to be a staircase parameter. The following snippet should go in the stimulus configuration file. If this is put in the `<default>` section, a single staircase will be used for all conditions. To use separate staircases per condition (the more common way to do staircases), put this definition in each `<Condition>` section of the configuration file.

```
<X      Type="Staircase" // The type of adaptive procedure
      LowerLimit=-0.1 // X will never be less than this
      UpperLimit=0.1  // X will never be more than this
      InitialStep=0.01 // Initially size of staircase step
      StepDecrease=0.9 // Steps are decreased with this factor
      MaxRuns=8        // The procedure stops after 8 runs
      Target="50%",    // A staircase is chosen to find 50%
      StepMode="PerRun"> // Steps are reduced after every run
0.5    // This is the value of X on the first trial
</X>
```

Apart from telling Neurostim that this stimulus parameter will be adapted during the experiment, you also need to implement the adaptation in your stimulus code. As adaptation is typically done in response to subject responses, the logical place to do the adaptation is in the keyboard handler. For instance:

```
void simpleDot::keyboard(unsigned char key, int x, int y){
    // Don't accept keypresses before the end of the stimulus.
    if (time < off){ return;}

    switch (key) {
        case 'l': //Saw this dot on the left
            if (X<0){
                adapt(CORRECT);
            }else{
                adapt(INCORRECT);
            }
            write('c',1,X); // Log answer and current value
                           // of X

            done(); // Terminate trial
        case 'r': //Saw this dot on the right
            if (X>0){
                adapt(CORRECT);
            }else{
                adapt(INCORRECT);
            }
            write('c',-1,X); // Log answer and current value
            done();// Terminate trial
            break;
    }
}
```

This code merely checks whether the current answer was correct or not and then calls the `adapt()` function to perform the appropriate changes (if any) to the parameter. Parameter changes are logged automatically; I put the `X` in the `write()` function just for convenience in the data analysis.

JITTER

Sometimes you may want to introduce some random variability in a parameter. For instance, in a detection task you may want to introduce some variability in the onset time of the target so that the subjects do not know when exactly the stimulus will appear. The Jitter attribute of a parameter in the stimulus configuration file is used for this.

For example, if you place the following tags in the Stimulus file:

```
<on Jitter="500"> 1000 </on>
```

Neurostim will present this stimulus at a random time between 1000-500 and 1000+500. On each trial a new onset time is randomly chosen (from a flat distribution in the interval [value-jitter;value+jitter]).

Note that Neurostim has no way of checking whether the parameters are "reasonable". For instance you could abuse this mechanism to generate a negative <on> time, which would lead to unpredictable consequences...

Also, the Jittered variable is treated internally first as a double variable. If you request a jittered integer, the value and the jitter are cast to doubles, the randomization process runs on the doubles and the end-result is cast back to an integer. This could have some unforeseen consequences, for instance, if you wanted to randomly choose 1 or 2 and thought that <Parm Jitter="0.5"> 1.5</Parm> would work, think again!

Jittering can be used for any variable in the stimulus file. For instance if <G> is the Green Gun value in your stimulus

```
<G Jitter="0.5">0.5</G>  
<R>1</R>
```

would generate stimuli that vary on successive trials from red to yellow.

You can use Global Variables (See page 44) to specify the starting point of the jitter, but not the jitter amount itself. For instance:

```
<G Jitter="0.5"><_GreenJitter/></G>
```

will work - it will use the value specified in the experiment file under <VARIABLES><GreenJitter></GreenJitter></VARIABLES>. But using something like Jitter="<_someVariable>" *will not work*.

Finally, because background stimuli never change and no new parameters are loaded for them on each trial, jittering does not work for background stimuli.

GLOBAL VARIABLES

Sometimes you may wish to have a variable that is used by multiple stimuli, maybe something like a `<Color>` that is changed every recording day. Rather than enter this value in many config files, you can use a global variable to achieve this.

To define a global variable, add it to the `<Variables>` section of the experiment file E.g.

```
<Variables>
  <_TodaysColor> BLUE </_TodaysColor>
</Variables>
```

Note that the first character of a global variable name must be an underscore!

To use the variable in a stimulus configuration file, you simply use the variable name instead of a value:

```
<Stimulus >
  <Color>
    <_TodaysColor></_TodaysColor>
  </Color>
</Stimulus>
```

Neurostim will then read the value of `<TodaysColor>` from the definition in the experiment file and use it as the value for `<Color>`. Using XML shorthand for an element without a value is legal too:

```
<Stimulus>
<X>1</X>
<Color>
  <_TodaysColor/>
</Color>
</Stimulus>
```

Now, when you start a new recording day, you only need to change the value of the global variable in one place (the Experiment file) and not in every stimulus file.

BACKGROUND STIMULI

Background Stimuli are on for the duration of an experiment, even between trials.

This could for instance be a fixation dot during an MRI experiment where you want the subject to fixate throughout the experiment. To make a stimulus a background stimulus, simply set its Background attribute in the experiment file to 1. For instance:

```
<Experiment>
[... ]
<Stimulus Name="nsFixation" Config="mri" background="1"/>
</Experiment>
```

Don't confuse background stimuli with the general background of the screen. This background color can be set directly in the Experiment file. For instance to choose an red background you'd specify:

```
<WINDOW>
  [...]
  <BACKGROUND>
    <R>1</R>
    <G>0</G>
    <B>0</B>
  </BACKGROUND>
</WINDOW>
```

Or if you use a calibrated monitor, you can specify the background color in calibrated units:

```
<WINDOW>
  <BACKGROUND>
    <XCIE>0.33</XCIE>
    <YCIE>0.33</YCIE>
    <LUMINANCE>5</LUMINANCE>
  </BACKGROUND>
</WINDOW>
```

Finally the <BACKGROUND> tag also has the option to show a fixation point on the blank screen before the experiment really starts (this could not be achieved with a user-defined stimulus). Its position, color, and size can be defined directly in the experiment file.


```
<BACKGROUND>
  <FIXATION>
    <ENABLE>0</ENABLE>
    <R>1</R>
    <G>0</G>
    <B>0</B>
    <XCIE>0</XCIE>
    <YCIE>0</YCIE>
    <LUMINANCE>0</LUMINANCE>
    <SIZE>5</SIZE>
    <X>0</X>
    <Y>0</Y>
  </FIXATION>
</BACKGROUND>
```

7 Installation

How to get it and how to get it running

SHOULD I

READ THIS?

 Only if you do not have Neurostim on your computer yet.

SYSTEM REQUIREMENTS

Operating Systems

Current development emphasis is almost entirely on Microsoft Windows. But, because most of the code is ANSI C++, Neurostim programs should run on any machine for which an OpenGL, OpenAL, and a FLTK library are available. At various points in its history, the program has been run on Windows (NT, 95, 2000, XP), IRIX (6.x), and Macintosh (OS 9).

Libraries

OpenGL libraries are built-in in Windows 98 and later, SGI Irix, Mac OS 10. An OpenGL toolkit is available from www.mac.com for OS9. FLTK is also available for many systems. Get the latest versions from www.opengl.org. OpenAL is available (through www.openal.org) from Creative Labs and runs under MS Windows.

Compilers

The currently recommended compiler to make your own Neurostim stimuli is Visual Studio C++ Express 2008. This compiler can be downloaded from Microsoft. The Neurostim file release contains .sln project files to create the Neurostim library as well as a Visual Studio Wizard to create your own stimuli.

The Dev-C++ compiler is an alternative; it is also free and another OpenSource project (www.sourceforge.net/projects/dev-cpp). This compiler is easy to use, but its debugger is not as intuitive as others. It only runs under Windows. The Neurostim file release contains (old) .dev project files to create the Neurostim library as well as example project files to create your own stimuli.

For Mac OS development, I have used Metrowerks Codewarrior and found it to be useful.

USER REQUIREMENTS

This manual assumes you are familiar with programming in C++, as well as OpenGL. There are many good books as well as websites on C++ programming. For OpenGL information, go to www.opengl.org.

DOWNLOAD

All sources, examples, installers, documentation, and a discussion forum can be found on Sourceforge: <http://www.sourceforge.net/projects/neurostim>.



INSTALLING NEUROSTIM

- **Microsoft Windows.** Download the latest release of the install program from Sourceforge. The first time you run this program, Administrator access is needed (to setup file associations and install some libraries in the System32 directory). Subsequent updates can usually be run without Administrator access.

Choose an installation directory. For instance, 'c:\program files\neurostim' will install to 'c:\program files\neurostim'. In what follows this directory will be assumed. The install program will set up a directory with the files that are needed to develop your own Neurostim programs. It will also copy some libraries to your system directory. After installation, go to c:\program files\neurostim\samples\ to look for sample project files. Try double-clicking a .nml file: that should start a Neurostim.

For development ease I typically also associated .nml and .nml.d files with an XML editor, for instance the FOXE editor (See section Windows Registry Script).

Next, try to recompile the sample program. Open the nsLLDots.sln file in Visual Studio and rebuild all. This should compile and link without errors and create a DLL for the example stimulus.

Once you are sure that you can modify and recompile the sample stimuli, you are ready to start creating your own stimulus.

- **Unix, Apple OS-X.** Sorry, no install program is available for these OS. Please follow the instructions or Developers to download the source from Sourceforge.

DEVELOPERS

Developers should obtain the source code from the SVN server (Use yourr favorite Subversion client to run the following command:

```
svn co https://neurostim.svn.sourceforge.net/svnroot/neurostim/trunk neurostim
```

Then go to the neurostim/make directory to find a "make" file for their favorite compiler. If the make/project/solution file exists, build the libraries. If they don't, you will have to set them up yourself.

After recompiling the Neurostim Debug and Neurostim Release libraries, go to the neurostim/samples directory and try to compile an example stimulus program.

8 History

SHOULD I

READ THIS?



If you want to know who made this, when, and why...


When	What	Who
1997	Neurostim is a graphics library for use on SGI systems and uses the GLUT libraries for interaction with the operating system.	Bart Krekelberg, Ruhr University Bochum, Germany
2001	Ported the library to Windows 2000, Mac OS9. Neurostim can run as a standalone program.	Bart Krekelberg, The Salk Institute, La Jolla, California.
2001-2003	Neurostim runs standalone or as a graphics slave to Cortex	Bart Krekelberg, Micah Richert, Jessica Olsen. The Salk Institute, La Jolla, California
2005	Added behavioral control; interaction with the Eyelink tracker, interaction with Measurement Computing data acquisition boards.	Bart Krekelberg, Rutgers University.
2006	Interaction with Nabeda, Thomas Recording eye tracker.	Sebastian Thias. Marburg University
2007	Documentation	Jessica Wright, Bart Krekelberg. Rutgers University
2008	Redesign to use the FLTK toolkit for interaction with the OS, added a GUI for program control, developed a more flexible manner to specify stimuli in XML, developed a plugin interface, improved plugins for Eyelink, MCC, fMRI etc.	Jonas Knoell, Marburg University. Bart Krekelberg, Rutgers University.

9 Answers to Selected Exercises

This section will provide additional details for selected exercises.

SHOULD I

READ THIS?

 You need help with
any of the exercises.

CHAPTER 2 – GETTING STARTED

Try it Out Exercises:

B. You should be varying the following parameters:

```
<XPIXELS>1024</XPIXELS>
<YPIXELS>768</YPIXELS>
<WIDTH>30</WIDTH>
<HEIGHT>20 </HEIGHT>
```

C.

```
<STIMULUSName="nsLLDots" Dll="nsLLDots" Config="oblique" />
<STIMULUS Name="nsLLDots2" Dll="nsLLDots" Config="spiral" />
```

D. The values for <G> and <off> differ. The fixation stimulus will always be red. <off> is set to 3000 ms, but the entire trial duration is only 2000 ms. The experiment file will still work even though the stimulus is set to turn off after the trial has completed, but the best solution is to just use reasonable times.

E.

```
<CONDITION Name="right">
  <nsLLDots>
    <X>10</X>
    <coherence>0</coherence>
  </nsLLDots>
</CONDITION>
```

F, G & H.

```
<FACTORIAL Name="coherence" Levels="3,3,2" >
  <FACTOR1>
    <nsLLDots>
      <X>-10,0,10</X>
      <Y>5,0,-5</Y>
    </nsLLDots>
  </FACTOR1>
  <FACTOR2>
    <nsLLDots>
      <coherence>0,0.5,1</coherence>
    </nsLLDots>
  </FACTOR2>
  <FACTOR3>
    <nsLLDots>
      <nrDots>100,500</nrDots>
    </nsLLDots>
  </FACTOR3>
</FACTORIAL>
```

I. <FACTORIAL Name="coherence" Levels="3,3,2" **Factor1Weights="1,2,1"**>
The rest of the Factorial code is the same as in the answer for F, G & H.

J. <BLOCKS Repeats="10">
 <BLOCK Randomize ="WithoutReplacement" Retry ="RANDOMINBLOCK">
 <coherence/>
 <left/>
 </BLOCK>
</BLOCKS>




OR

```
<BLOCKS>
  <BLOCK Randomize="WithoutReplacement" Repeats="10"
    Retry="RANDOMINBLOCK">
    <coherence/>
    <left/>
  </BLOCK>
</BLOCKS>
```

Either method can be used in this case to achieve the same results. To check, you can use Ctrl-L to see a list of the conditions as they will be presented in the experiment. When you first open up your experiment just press Ctrl-L and view the Command Prompt.

10 Appendices

SHOULD I READ THIS?

-  You need help with installation.
-  You were referred here for more details.
-  You are a developer.

WINDOWS REGISTRY SCRIPT

For a standard Windows Installation, with the Foxe editor to edit nml and nmld files, the following .reg file provides the correct associations and context menu.

```
Windows Registry Editor Version 5.00
[HKEY_CLASSES_ROOT\Neurostim.nml]
@="Neurostim Experiment File"
[HKEY_CLASSES_ROOT\Neurostim.nml\DefaultIcon]
@="C:\\Program Files\\Neurostim\\bin\\ns.exe,0"
[HKEY_CLASSES_ROOT\Neurostim.nml\shell]
@="Open"
[HKEY_CLASSES_ROOT\Neurostim.nml\shell\Edit]
[HKEY_CLASSES_ROOT\Neurostim.nml\shell\Edit\command]
@="c:\\program files\\foxe\\foxe.exe %1"
[HKEY_CLASSES_ROOT\Neurostim.nml\shell\Open]
[HKEY_CLASSES_ROOT\Neurostim.nml\shell\Open\command]
@="C:\\Program Files\\Neurostim\\bin\\ns.exe %1"
[HKEY_CLASSES_ROOT\Neurostim.nml\shell\OpenDebug]
[HKEY_CLASSES_ROOT\Neurostim.nml\shell\OpenDebug\command]
@="C:\\Program Files\\Neurostim\\bin\\nsDebug.exe %1"
[HKEY_CLASSES_ROOT\Neurostim.nmld]
@="Neurostim Data File"
[HKEY_CLASSES_ROOT\Neurostim.nmld\DefaultIcon]
@="C:\\Program Files\\Neurostim\\bin\\ns.exe,0"
[HKEY_CLASSES_ROOT\Neurostim.nmld\shell]
@="Open"
[HKEY_CLASSES_ROOT\Neurostim.nmld\shell\Open]
[HKEY_CLASSES_ROOT\Neurostim.nmld\shell\Open\command]
@="C:\\Program Files\\Fuxe\\Fuxe.exe %1"
```

FILE FORMATS

Neurostim uses Extensible Markup Language (XML) to specify properties of experiments and stimuli, to write output data, and to set monitor calibration. You can (and should) write comments in these files, using the standard XML format (<!-- comment -->). Be careful not to nest comments!

EXPERIMENT FILES (.NML)

The top level element is <Experiment> which contains exactly one <Controller> element and zero or more <Stimulus> elements. The <Controller> element has one attribute: **Name**, which specifies which Controller will be used (see page 8 for a list of choices).

An experiment file can include another file (which has settings common to all experiments). Use <INCLUDE> fileToBeIncluded.nml</INCLUDE> as an element of the <Controller> element. Properties that are defined in both the experiment file and the included file get the values of the experiment file (i.e. the experiment file overrules).

The default values for each element of the <Controller> element are set in a file called global.nml. This file is stored in neurostim/settings/global.nml. This file also contains some explanation of what all those parameters mean.

The official XML Schema definition of the Experiment files is stored in neurostim/settings/experiment.xsd. Neurostim does not currently check validity against this schema, but using a validating text editor such as the one integrated in Visual Studio will help to reduce errors in experiment files.

STIMULUS FILES (.NML)

Stimulus files specify the properties of a single stimulus. The top-level element is <Stimulus> which contains one <Default> element that specifies the default values of parameters. The <Stimulus> element also contains one or more <Condition> elements which have **Name** and **Active** attributes. The Name attribute gives a name for a condition. The Active attribute can be set to 0 to disable a stimulus in a particular condition.

Apart from the parameters defined by the person who created the stimulus, each <Condition> or <Default> element can also contain elements to specify the properties that all Neurostim stimuli have:

X	the X position of the stimulus
Y	the Y position of the stimulus
Z	the Z position of the stimulus
scaleX	the X scale factor of the stimulus
scaleY	the Y scale factor of the stimulus
scaleZ	the Z scale factor of the stimulus
phiX	the rotation of the stimulus around the X axis
phiY	the rotation of the stimulus around the Y axis
phiZ	the rotation of the stimulus around the Z axis
on	the onset time of the stimulus in milliseconds
off	the offset time of the stimulus in milliseconds.
duration	how long the stimulus stays on (alternative to off)

DATA FILES (.NMLD)

Neurostim produces output files in XML format with the .nml extension. The top-level element is <Data>, which contains one <Experiment> element and one or more <Trial> elements. The <Experiment> element contains the complete settings of the Neurostim program when it produced the data file at hand (in essence a copy of the Experiment file that was used to run the experiment plus all inherited properties from global.nml). <Trial> elements have two attributes which represent the trial number and the condition number. Each <Trial> element contains one or more <SRC> elements. Each <SRC> represents events generated by subparts of the Neurostim program (such as the core program “NS”, the monitor “Monitor”, and the stimuli (e.g. “nsLLDots”). Each <SRC> element contains one or more Events <E> elements. <E> elements have four attributes; their name (N), the time at which the event

occurred (Ti), the trial in which it occurred (T), and the condition that was running in this trial (C).

MONITOR CALIBRATION FILES (.XML)

The files provide Neurostim with information on the visual display that you are using and enable it to present stimuli with known luminance and color.

A simple version exists especially for the Sony Artisan monitor, which can calibrate itself in a linear mode. In this mode the monitor produces only one color but it does so in equal (8-bit) steps from the lowest to the highest luminance. The calibration XML file looks like:

```
<?xml version="1.0" ?>
<Calibration Type="ARTISANLINEAR">
  <name>Artisan Psychophysics Rig</name>
  <luminance>
    <min>0.5</min>
    <max>80</max>
  </luminance>
  <color>
    <xCie>0.33</xCie>
    <yCie>0.33</yCie>
  </color>
</Calibration>
```

A more complete calibration file also contains the <Calibration> top level element but it contains different child elements that specify the details of the luminance dependence for each gun as well as the color primaries:

```

<?xml version="1.0" encoding="utf-8"?>
<Calibration>
  <Info>
    <Photometer>Colorcal</Photometer>
    <Experimenter>bart</Experimenter>
    <Monitor>psych</Monitor>
    <CalibrationFileDuringCalibration/>
    <CalibrationExperiment>calibratePsych</CalibrationExperiment>
    <Date>20-Sep-2007</Date>
  </Info>
  <GammaFunction>
    <!--Information used to generate luminance calibrated stimuli (Lum=A*gun^G+B)-->
    <Alpha>23.270658196143039 51.294657824719884 6.722736928560476 </Alpha>
    <Beta>0.000000000000000 0.000000000000000 0.000000000000000 </Beta>
    <Gamma>2.180132896841701 2.178215215623634 2.170845940768591 </Gamma>
  </GammaFunction>
  <Color><!--Information used to generate calibrated CIE color -->
    <LumToTri>
      <X>1.834700875685935 0.453027554947734 2.101509508797079 </X>
      <Y>1.000000000000000 1.000000000000000 1.000000000000000 </Y>
      <Z>0.083085061359995 0.170296483521275 10.849440243170219 </Z>
    </LumToTri>
    <TriToLum>
      <R>0.716776163992664 -0.305876919170292 -0.110644879204258 </R>
      <G>-0.722629729600546 1.324321510710738 0.017907996452313 </G>
      <B>0.005853565607882 -0.018444591540446 0.092736882751945 </B>
    </TriToLum>
    <Ambient>0.173584094312087 0.197144500000000 0.209757821766350 </Ambient>
  </Color>
</Calibration>

```

LEARNING C++ AND OPENGL

The Stimulus Wizard allows you to concentrate on programming the graphics of your stimulus and leave all the rest to Neurostim and the Wizard. Your task is to define the `setup()`, `move()`, and `draw()` functions. Their specific goals are described in the Neurostim User's Guide.

The information here is not to teach you C++ or OpenGL, but to point you to some useful resources and indicate what you should learn from them. The following is a list of concepts that you should become familiar with:

C++

Variable Types: (declaration, types, int, float, double, char). You should be aware what kind of information can be stored in each of these.

Arrays: (float a[10]) These are used to store multiple elements when you know in advance how many elements you have.

Pointers: (char*, float*). These are used to store an unknown number of elements.

Vectors (vector<int>, vector<double>). This is a better and safer way to store a list/vector of numbers.

Strings (string). Allows you to store/compare/search/manipulate strings. Much preferred over the older char * method to store strings.

Flow control. (if, then, switch, case, while, return). These statements are required to regulate the flow of the program.

Useful Websites:

These sites have C++ tutorials. You don't have to learn everything on these sites to be able to write a Neurostim stimulus.

<http://www.cplusplus.com/doc/tutorial/>

<http://cplus.about.com/od/beginnerctutorial/1/blcplustut.htm>

OPENGL

The OpenGL Red Book, which is an extensive guide to programming OpenGL. The book covers Version 2.0 of OpenGL. Sadly, Microsoft's OpenGL libraries are still at the 1.3 version. Nevertheless, most of the information in the Red Book is correct. When Visual Studio complains about not finding a particular OpenGL command that you copied from the book, the reason is probably that the command had not been defined yet in version 1.3. We'll have to wait until Microsoft (or someone else) releases updated libraries to use those commands. To use Neurostim, the following chapters of the red book provide useful information. Chapters that are not mentioned here can be skipped.

1. Introduction: For a general overview and the concepts of OpenGL Neurostim relies on FLTK to make its windows, but all of that code is hidden from the user's view.

2. State Management and Drawing Geometric Objects

This is where you will learn how to draw a circle, a square, etc.

3. Viewing

This chapter will teach you to think in terms of a model, not in terms of pixels on the screen. All of the calculations referred to in this chapter are done behind the scenes by Neurostim, but it is good to learn about the Camera Analogy in OpenGL.

4. Color

Neurostim uses RGBA color mode.

6. Display Lists

If you are doing complex drawings that take up a lot of time, consider using a display list that packages the whole drawing into one faster bit of code. Otherwise you will not need this.

8. Drawing Pixels, Bitmaps, Fonts, and Images.

If you really need to draw pixels (you seldom do!), read this chapter. Most of the time, you are much better off using geometric objects and texture mapping (see below).

9. Texture Mapping

This is a technique to apply a pattern to a geometric object. For instance, to create a grating you can specify the shape of a polygon and then apply a texture map of a grating to that shape. This will allow you to manipulate the grating by changing the polygon rather than recalculating the whole texture each time the change is made.

Useful Websites:

www.opengl.org

Good tutorials (some of which are used in the Red book) can be found at <http://www.xmission.com/~nate/tutors.html>