LyX Notebook
Version 0.1alpha

Allen Barker*

May 27, 2012

Abstract

Lyx Notebook is a program which allows the Lyx word processor to be used as a code-evaluating notebook, similar to Mathematica notebooks, the Sage notebook, or interactive editors such as Tex/Mathematica in Emacs. Code is written in cells, which the user can automatically evaluate in an interpreter. Several different interpreted, interactive languages are supported. The printed output is formatted with syntax highlighting, using the Listings package.

1 Introduction and Overview

Lyx Notebook[1] is a program which allows the Lyx[2] word processor to be used as a code-evaluating notebook, similar to Mathematica notebooks, the Sage notebook, or interactive editors such as Tex/Mathematica in Emacs. The basic Lyx Notebook system can be used with a variety of different interpreted languages. The currently-supported languages are Python 2.x, Python 3.x, Sage, Scala, R, and Bash, but it should not be difficult to add the specifications for additional interactive interpreters.

The Lyx Notebook system starts up and manages interpreter processes in the background. It uses the native interpreter for any particular interpreted language, run from a pseudo-tty, rather than defining its own prompt-read-eval-print loop. Users who are already familiar with a particular system can therefore use it in the ways in which they are accustomed to using it. Note that multiple interpreted languages can be used simultaneously: both in the same document and in the same editing session.

The basic code input to the Lyx Notebook system is via custom insets called **code cells**. Code cells are similar to Lyx's built-in listings insets (they are also

*Allen.L.Barker@gmail.com

[1] In most contexts in this document the typesetting Lyx, Latex, and Tex is used rather than LyX, LaTeX, and TeX.

[2] Lyx Notebook works with Lyx, and is licensed under the same GPL, but is not officially affiliated with the Lyx project.

wrappers for the Listings package). Unlike the listings insets, code cell insets always have a label specifying an interpreted language, and they must contain executable code in that language. The code from any code cell can be sent to its corresponding interpreter to be evaluated when the user presses a Lyx Notebook command key. The results of the evaluation are written into another custom inset, called an **output cell**.

There are two different types of code cells: **standard cells** and **init cells**. The primary difference between init cells and standard cells is that any multi-cell Lyx Notebook commands which evaluate general code cells always evaluate the init cells before the standard cells. Otherwise, execution is strictly sequential according to placement in the Lyx file. Thus, for example, blocks of uninteresting initialization code can be placed in an appendix and yet still be executed before the standard code cells of the main text. The insets for init cells and standard cells have slightly different labels in order to clearly differentiate between the two types of code cells inside Lyx.

When a file is converted to a printable format (via Latex) the listings program is used to automatically format the code in the code cells. The output cells are also printed, though without any special formatting except for line-breaking. In printed format the default convention is to print init cells with a doubled frame borders on the top and sides, while the standard cells have single-line frame borders.

A simple example of an init cell is shown below. It was initially entered in Lyx (after the setup procedure described in Section 2) by selecting the menu item **Insert** ▷ **Custom Insets** ▷ **LyxNotebookCell:Init:PythonTwo**. Notice the small "Python" label at the right of the cell, which indicates that the language associated with the cell is Python (i.e., the cell contains Python code and the Python interpreter is used to evaluate it).[3] The doubled top and side frames indicate that this is an init cell.

```
# A Python 2.x init cell.
import math
print "end of init cell"
```

Python

```
end of init cell
```

The cell above has been evaluated, and the output cell follows directly after the code cell. When Lyx Notebook is running a single cell at the cursor point can be evaluated simply by pressing the key bound to the command "evaluate current cell." All the cells in the current buffer can be evaluated by pressing the key bound to the command "evaluate all code cells."

Output cells do not need to be directly inserted by the user. When a code cell is

[3]Both Python 2.x and Python 3.x cells are currently simply labeled "Python" in the printed output. This can be changed if, for example, you are writing a paper comparing the two versions of Python. See Section 10 on customizing. By default Python 3.x *insets* in Lyx are labeled as "Python," but Python 2.x *insets* are labeled as "PythonTwo". (That particular string becomes part of several Latex command and environment names, and Latex does not allow digits like "2" in those names.)

evaluated the Lyx Notebook program will automatically create an output cell if one is not already present. The rule is that there can be no space between a code cell inset and its corresponding output cell inset.

Here is a standard cell, also evaluated:

Python

```
# An example calculating the areas of ellipses.
print "The area of certain ellipses:"
print "%10s%10s%10s" % ("Axis 1", "Axis 2", "Area")
for x in [1,2]: # note loops are written like in a file
   for y in [3,4]:
      print "%10.2f%10.2f%10.2f" % (x, y, math.pi * x * y)
```

```
The area of certain ellipses:
    Axis 1    Axis 2      Area
      1.00      3.00      9.42
      1.00      4.00     12.57
      2.00      3.00     18.85
      2.00      4.00     25.13
```

Notice that even though the code is evaluated by the interactive Python interpreter it is entered into a cell as if it were in a file. Blank lines are ignored and could be added anywhere in the code. The printed version of this documentation shows how the cells look after formatting. The view from within Lyx is shown in Figure 1.

Moving on to another example, we now insert a standard Sage cell and use it to evaluate an integral:

Sage

```
# An example using Sage to evaluate an integral.
result = integral(x*sin(x^2), x)
print "result is:", result
print "the latex of the result is:", latex(result)
```

```
result is: -1/2*cos(x^2)
the latex of the result is: -\frac{1}{2} \, \cos\left(x^{2}\right)
```

Both the text form and the Latex form of the result have been printed to the output cell. If the Latex code is pasted into a math inset the result is: $-\frac{1}{2}\cos\left(x^2\right)$.

The Lyx Notebook program is intended to be a useful tool for writing research papers (as opposed to just being a demo or a toy). It does, however, have some limitations. The package currently only works with Linux systems, and it has only been tested with Lyx 2.0.3 on Fedora 15 Linux. The code is written in Python and most of the testing has been done under Python 2.7, although most features also work when running with a recent Python 2.x or with Python 3.x.

Due to a current lack of Lyx functions (LFUNs) which directly implement certain required operations, some of the basic functionality had to be implemented via workarounds. The workarounds should be relatively reliable, but, for the time being, they are not particularly elegant or efficient. This is described further in later sections, but from a user's perspective the temporary insertion of a "magic cookie"

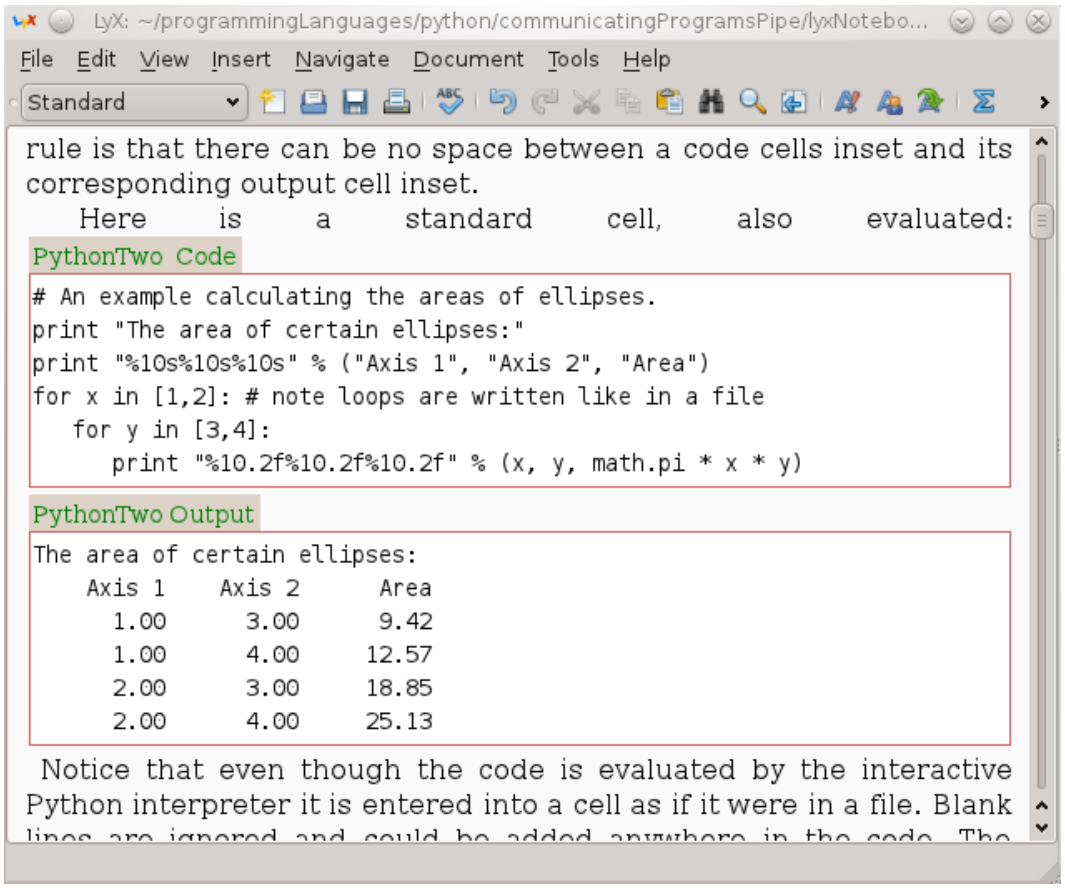


Figure 1: A standard Lyx Notebook code inset for Python 2, along with its output cell, as it appears in Lyx.

character string is easily noticeable. The level of efficiency should be sufficient for most user interactions.

The Lyx Notebook system is only intended to be used with interpreted languages, not with compiled languages. This system is different from some others in that the code is executed by the user at same time as the document is being written. The output then becomes part of the current document, as opposed to systems which batch-process all the output when Latex is run and then substitute-in evaluations at that point. Lyx Notebook also maintains active interpreters with their full internal state, as opposed to systems which send commands one-by-one to be interpreted but which lose the results of previous computations.

2 Installing and Running

This section describes the installation and setup of Lyx Notebook. Most of the setup is automated, so most users will only need to carry out steps 1–5 below. Steps 1–3 are basic setup. Steps 4 and 5 cover how to use the system in a particular document (and should be repeated for each document which uses the system). The setup program assumes that the user's Lyx home directory is **`~/.lyx`**, and that the names of the Lyx server pipes are set to their default values. These settings can both be changed in the file **`lyxNotebookUserSettings.py`**.

The setup program creates two key-binding files in the user's home Lyx directory: **`lyxNotebookKeyBindings.bind`**. and **`userCustomizableKeyBindings.bind`**. The program always asks before overwriting either of these files. The first file contains the key bindings for the Lyx Notebook commands. By default the function keys are redefined to become Lyx Notebook commands. To bind arbitrary keys to arbitrary Lyx Notebook functions, see Section 10 on customization.

Those Lyx Notebook command-key bindings (to function keys) are *included* from the second file, **`userCustomizableKeyBindings.bind`**, which users will set as the active Lyx key-binding file. The relevant include line (**`\bind_file`** line) can be commented-out at times when the user does not want the function key bindings to be redefined. Arbitrary bindings can be added to this new active binding file, and any additional bindings files which the user wishes to use can also be included from that file. The cua bindings are used by default, but that can easily be changed in this file. Advanced users can, of course, arrange the binding files however they prefer.

To install Lyx Notebook, carry out the following steps:

1. **Unpack the program**. Unpack the file containing the source directory of the Lyx Notebook program. From a terminal window, the command for unpacking a **`.tar.gz`** file is:

   ```
   tar xvf lyxNotebook.tar.gz
   ```

   where the actual name may include version information. Note the path to the newly-created source directory.

2. **Run the `setup.py` program**. In a terminal window, change directories to the new Lyx Notebook source directory and type:

   ```
   python setup.py &
   ```

   You need to have Python 2.x or Python 3.x installed and in your **`PATH`** for this to work. Also, the EasyGUI package must be installed if it is not already (in Fedora: **`sudo yum install python-easygui`**). The setup command can be re-run arbitrarily many times.[4]

3. **Reconfigure Lyx and restart**. Open Lyx, click on the menu item **Tools** ▷ **Reconfigure**, close Lyx, and then restart. These steps are also described on the text window that pops up after the setup program has finished.

4. **Start a document**. Start a Lyx document in the usual way. In addition to the usual settings, you need to include any Lyx Notebook modules for the cell types which you wish to use. Go to the modules menu at **Document** ▷ **Settings** ▷ **Modules** and add the **LyxNotebook** module for *each* language which you wish to use in the document.

5. **Use the system**. The system should now be ready to use. To insert a Lyx Notebook cell, go to the menu **Insert** ▷ **Custom Insets** and choose the type you want to insert. In order to evaluate cells the program **`lyxNotebook`** in the Lyx Notebook directory must be running:

   (a) If step 1 worked correctly you should be able to run Lyx Notebook just by pressing F12 (assuming the default keymap). This is by far the easiest way to use the program; Shift+F12 then halts the program's execution.

   (b) You can also start the program by entering the command

   ```
   <pathToLyxNotebookSourceDir>/lyxNotebook
   ```

   in a terminal window, replacing **`<pathToLyxNotebookSourceDir>`** with enough path so the file will be found.

   Allow the Lyx Notebook program to remain running as long as you wish to use its active features (such as cell evaluations and goto commands). This also maintains the internal state of the interpreters.

After the steps above have been completed, all the commands and examples described in this document should be functional (assuming the interpreted languages are installed). For documentation purposes, or if problems arise, the full process carried out by **`setup.py`** in steps 1 and 2 is described in alternate steps 1 and 2 below. Users who successfully completed the steps above can skip the remainder of this section.

[4] Re-running **`setup.py`** is required if the full pathname to the user's home directory changes or if the pathname to the Lyx Notebook source directory changes. It may be required if the Lyx Notebook program is updated, and is recommended in that case just in case. It is also required when certain of the settable parameters are changed (but most such parameters do not require it).

1. **Set up the key binding files**.

   (a) Edit the file **`userCustomizableKeyBindings.template`** in the subdirectory **`filesForDotLyxDir`** of the Lyx Notebook source directory and change **`<<userHomeLyxDirectory>>`** to the actual path (usually **`~/.lyx`**). Write the results to a file **`userCustomizableKeyBindings.bind`**.

   (b) Edit the file **`lyxNotebookKeyBindings.template`** in the subdirectory **`filesForDotLyxDir`** of the Lyx Notebook source directory and change **`<<absPathToLyxNotebookSourceDir>>`** to the absolute pathname to the Lyx Notebook source directory. Note that this *must* be an absolute path. Write the results to **`lyxNotebookKeyBindings.bind`**.

   (c) Copy the two **`.bind`** files created above to the the local user Lyx directory (usually **`~/.lyx`**).

   (d) Open Lyx, go to the menu **Tools** ▷ **Preferences** ▷ **Editing** ▷ **Shortcuts**, and enter (or browse to) the pathname of the **`userCustomizableKeyBindings`** file in the **Bind file** box on the top right (enter it without the **`.bind`** suffix).

2. **Set up the modules for Lyx Notebook cells**. Copy all the files in the subdirectory **`filesForDotLyxLayoutsDir`** with the suffix **`.module`** to your local user Lyx layouts directory (usually **`~/.lyx/layouts`**). If you do not need all the interpreted languages available you can only copy the **`.module`** files which include the languages you need in the name. These files are generated by the program **`generateModuleFilesFromTemplate.py`**, which must be re-run if certain customizations are done (see Section 10). The setup program runs that program automatically (and does the file copying).

3 Using Lyx Notebook

After the system is installed, and the modules for a given interpreter are included in the document settings, Lyx Notebook code cells for those interpreters can be freely inserted into the document. The different cell types can be found on the menu **Insert** ▷ **Custom Insets**.

Whenever the Lyx Notebook program is running any of the Lyx Notebook commands can be entered. In the default setup the program can be started simply by pressing the F12 key. After that the system should be fairly intuitive to use. To get started you can just remember: F12 to start, F1 for a menu, and F4 to evaluate a cell at the cursor position.

The full list of Lyx Notebook commands with key bindings is shown in Table 1. The commands can also be seen by starting Lyx Notebook (F12) and then pressing F1 to get a menu. There are additional commands on the menu, not bound to any

Default Key	Command
F1	pop up submenu
S+F1	toggle prompt echo
F2	goto next init cell
S+F2	goto prev init cell
F3	goto next standard cell
S+F3	goto prev standard cell
F4	evaluate current cell
S+F4	evaluate newlines as current cell
F5	evaluate all code cells
S+F5	evaluate all code cells after reinit
F6	evaluate all init cells
S+F6	evaluate all init cells after reinit
F7	evaluate all standard cells
S+F7	evaluate all standard cells after reinit
F8	reinitialize current interpreter
S+F8	reinitialize all interpreters for buffer
S+F9	insert most recent graphic file
F11	open all cells
S+F11	close all cells
F12	start lyx notebook
S+F12	kill lyx notebook process

Table 1: The Lyx Notebook commands with default keyboard bindings.

key by default, which are discussed in Section 4. Before going into those details we first cover some general rules.

The following are rules which *must be* followed when using the Lyx Notebook program.

1. There can be no space between a code cell and its corresponding output cell (or else a new cell will be created on evaluation). Multiple output cells can be maintained in this way, but they will probably not look good in printed form unless some extra vertical space is inserted before them.[5]

2. Be careful not to click somewhere while the updates are taking place, since that can sometimes move the cursor position (which the updating algorithm assumes is not externally modified). Use "undo" if this kind of problem occurs.

3. No inset with Plain Format inside it should be placed immediately after a code cell unless the intention is for output from the code cell to be written into that inset. Two code cells should always be separated by at least one character.

4. Documents must not contain the cookie string used by Lyx Notebook. If any instances of the cookie are left in the file after an error, use undo or search-and-delete to remove them. The current default value is the string "zZ4Qq" except with a 3 instead of a 4. The value of the cookie string can be changed in the file **`lyxNotebookUserSettings.py`**.

5. Do not change the definition of interpreter prompts within a cell (the definitions in the file **`interpreterSpec.py`** file must remain valid). Never write code in a cell which prints a newline followed by a prompt-string for the given interpreter. This holds for main prompts as well as continuation prompts.[6]

6. Cells cannot occur in titles, section headings, etc.

7. Code cells cannot be placed inside floats or minipages.[7] Cells in floats seem

[5]Some negative space is inserted at the beginning of an output cell so that the frame around it will be juxtaposed with the frame around its preceding code cell. This default can be modified; see Section 10.

[6]Whenever a code cell is evaluated the Lyx Notebook program waits for a newline followed by a prompt (and possible whitespace) in order to tell when the interpreter has finished evaluating the line of code. Code which prints a newline followed by a prompt (and which then delays for a bit) will cause problems. Later a fancier detection method may be implemented for interpreters which allow for redirecting the prompt output.

[7]This is because, like the Flex insets, those insets use Plain Layout (according to the **`server-get-layout`** LFUN). That distinction is used to tell when the cursor is inside a cell inset versus in surrounding text (which is Standard Layout). So the evaluation and creation of output cells will not work correctly. They will still evaluate correctly if they are enclosed in, say, an enumerate inside the float, as long as the output cell is input directly afterward, by hand, or there is at least one character after the cell and inside the enumerate. Cells in floats and minipages do print correctly, however, even when they do not evaluate correctly.

like a bad idea anyway, since the sequential nature of the cell-evaluations is lost in the final text.

4 Lyx Notebook Commands

Whenever the Lyx Notebook program is running it will respond to any of the defined command-keys in Lyx. By default the key F12 starts the program. Any interpreters which are required in order to evaluate code are started-up by the program (on demand) and remain running in the background.

Note that by default different interpreters are started to evaluate code cells in different buffers. For example, if you have two documents open and they both have PythonTwo cells in them then those cells will be evaluated in *separate* interpreter instances, one for each buffer. This behavior can be changed, via a setting in **`lyxNotebookUserSettings.py`**, to only use a single instance of any interpreter-type. Under the default setting, if the name of a buffer is changed (such as with "Save As") then the the succeeding evaluations in that buffer will have new interpreter instances started for them.[8]

In discussing the general Lyx Notebook commands it is helpful to organize them in groups. The command to open the submenu (bound by default to the F1 key) is a special one, but it should not require any further explanation. The remaining commands can be classified as goto commands, miscellaneous commands, ordinary cell-evaluation commands, batch cell-evaluation commands, and cell-open and -close commands. Each group of commands is discussed in a subsubsection below.

4.1 Goto Cell Commands

These commands are used to go to the next or previous cell of a given type. They work by inserting a cookie in all such cells, using a forward or reverse search for the cookie, and then deleting all the cookies. The user can use these commands to conveniently move forward and backward between cells, but they are mainly used in the implementation of commands which evaluate all the cells of a given type.

4.2 Miscellaneous Commands

There are several miscellaneous commands, each of which is described in a paragraph below.

The "write all code cells to files" command extracts all the code from all the code cells in the document and writes it to executable files (one file for each interpreter

[8] Also, any old interpreter instances which were actually started will still be active, though idle, until Lyx Notebook closes or the "reset all interpreters in all buffers" command is issued. This overhead will usually not be a problem, but it is worth noting. This happens because the Lyx Notebook program has no way to keep track of all the buffers in Lyx in order to determine which ones have closed (short of regularly cycling through them all with **`buffer-next`**).

with a cell in the document). The files have names of the form:

<bufferFileName>.allcells.<insetSpecifier>.<codeSuffix>

For example, the Scala cells in a document **newfile.lyx** would be written to the file

newfile.allcells.Scala.scala

The files are written in the directory corresponding to the current buffer, i.e., the directory where the **.lyx** file in the current buffer is located.

The "toggle prompt echo" command is useful for debugging, or just for getting a feel for how Lyx Notebook is processing the input to and output from an interpreter. When prompt echoing is turned on none of the interpreter output will be suppressed in the output cells. The output will look almost identical to a session in the interpreter (it is quite a bit more verbose). Here is a cell which was evaluated with prompt echoing turned on:

Python

```
# An example with prompt echoing turned on.
for i in [0,1]:
   if i == 1: print
   for j in [2,3]:
      print "(", i, ",", j, ")",
```

```
>>> # An example with prompt echoing turned on.
... for i in [0,1]:
...    if i == 1: print
...    for j in [2,3]:
...       print "(", i, ",", j, ")",
...
( 0 , 2 ) ( 0 , 3 )
( 1 , 2 ) ( 1 , 3 )
>>> pass
>>>
```

If prompt echoing had been turned off then only the two lines with numbers in parentheses would have appeared in the output cell. (See Section 6 for the details of how Python code is processed.)

The "evaluate newlines as current cell" command is useful to "unjam" a cell if, for example, code errors cause it to stick at a continuation prompt. This command simply sends two newlines to the interpreter instead of sending the actual code in the cell. The output is replaced, but the code in the cell remains unchanged. The cell can then be re-evaluated when the problem is fixed.

The "reinitialize all interpreters for buffer" command resets all the interpreters for the current buffer. Completely new processes are started up for each interpreter, so it is a hard reset. The commands to reinitialize only the current interpreter are not currently implemented; all interpreters are restarted. The command "reinitialize all interpreters for all buffers" resets all interpreters for all buffers, cleaning up any "orphan" processes which have been left due to closing or renaming documents.

The "insert most recent graphic file" command searches the directory subtree rooted in the directory associated with the current buffer, looking for any graphics files. The most recent one found is inserted in Lyx as a graphics inset. See Section 8 for examples of how this command can be used.

4.3 Ordinary Cell-Evaluation Commands

The simplest cell-evaluation command is "evaluate current cell," bound by default to the F4 key. It evaluates the cell at the current cursor point and writes the output to the corresponding output cell. Commands such as "evaluate all code cells" and "evaluate all init cells" use the goto commands to sequentially visit each cell of the specified type and evaluate it (using the same method as "evaluate current cell"). The user can watch as each cell is evaluated.

To halt a multi-cell evaluation which is in progress a user can press any Lyx Notebook function key (any key bound to **`server-notify`**). After the current cell's evaluation is completed a pop-up menu will ask whether or not to continue the evaluations.

These commands also have variants which reinitialize the interpreters before any evaluations.

4.4 Batch Cell-Evaluation Commands

The ordinary cell-evaluation commands are useful, and it is helpful at times to see each cell being evaluated. The sequential goto operations tend to be slow, however. The batch cell-evaluation commands also evaluate multiple cells in a document, but they operate on the Lyx file without using the goto operations.

The commands such as "batch evaluate all code cells" create a new **`.lyx`** file with the specified type of cells all evaluated. They then open that file as a new buffer. The file has the same name as the file in the current buffer except that the suffix is **`.newOutput.lyx`** rather than just **`.lyx`**.

The operations which write to a new buffer are reasonably safe to use, but may not be convenient. If you are feeling brave, there is also a mode of batch evaluation which replaces the current buffer file and reloads it in the current Lyx buffer. The command "toggle buffer replace on batch eval" will switch to this mode. Have backups when first trying these routines. If this works well it can be made the default initial mode via a setting in the file **`lyxNotebookUserSettings.py`** (the mode can still be toggled on and off interactively).

For safety in this latter mode, Lyx Notebook keeps a number of copies of the replaced buffers. These are written to files with names prefixed by **`.LyxNotebookSave0_`**, **`.LyxNotebookSave1_`**, etc. These files are "hidden" to the usual **`ls`** directory listing operation. The zero file is always the most recent buffer file which was replaced. The number of save files defaults to 5, but can be set in **`lyxNotebookUserSettings.py`**. The command "revert to most recent batch eval backup" will undo by reverting

the current buffer to the most recent backup file (the other backup files are shifted down).

4.5 Commands to Open and Close Cells

These commands just open or close all the cells of a given type. All the cells are currently opened when any of the goto operations are used.

5 General Notes and Suggestions

This list contains some general notes and suggestions which apply to any interpreter. Some Python- and Sage-specific notes and suggestions are given in Section 6.

1. Undo works reasonably well with Lyx Notebook. If you have problems, try un-doing.

2. Remember that the F1 key brings up the Lyx Notebook menu. All of the currently-defined Lyx Notebook commands are listed there, along with the key bindings for those which are have bindings.

3. For the time being the "goto" functions for cells of a given type must open the insets of all of the cells of that basic type (and currently all of the cells are opened, of all types).

4. Due to a bug (or feature), blank lines at the end of Lyx Notebook cells are not printed (and are not preserved when Latex export is used internally, but that is not the default). This holds for both code cells and output cells (such as when a blank line is printed as the last output). This seems to be due to those lines not being preserved on export to Latex.

5. It doesn't seem to make much difference whether Lyx Notebook cells are part of the surrounding text paragraph or are placed in their own paragraph.[9] If a code cell is in its own paragraph then it will be indented in Lyx (unless that particular paragraph property is changed), but the output cell following it will not be. Using a separate paragraph with the "indent paragraph" property turned off can increase the spacing in the formatted output, so that should probably be avoided. The downside of keeping cells as part of the surrounding text paragraph is that the last line can be stretched-out and hard to read.

6. Blank lines are ignored by default. They are not even sent to the interpreter. (They are, however, assumed to end any explicit line-continuation from the previous line.) This setting can be modified; see Section 10.

[9] The built-in Lyx listings insets have the sort of behavior that would be preferable. They remain part of the surrounding paragraph, but they are also placed on a separate line (without the text above being stretched-out). This does not currently seem to be an option for custom Flex insets.

7. If you use **\flushbottom** as well as labels on the cells (specifying their code languages) the labels may become slightly misaligned when the text is shifted. If this becomes a problem you can turn off labels or try **\raggedbottom**, either for the full document or for parts of it. If labels become separated from a cell across a page break, try adding this line to the document's preamble after increasing the default value of 4 to 5 or more:

   \def\lyxNotebookNeedspaceLabelValue{4\baselineskip}

8. Long lines in cells are set by default to auto-wrap (and auto-indent), with "\" as the printed continuation-character in the output cells. Breaking will only be done at spaces. Here is a simple example:

```
print "This is an extremely long line which will be automatically \
  line-wrapped by the Listings program."
```

Python

```
This is an extremely long line which will be automatically line-wrapped \
  by the Listings program.
```

9. As the previous item shows, code cells can be placed inside enumerate environments, list environments, etc. They will be indented like the surrounding text, and the frame will be shortened correspondingly.

10. All output cells are, by default, truncated at 1000 lines. This is to avoid "accidents" when the code in a cell has a bug in it. The number of lines can be changed in the file **lyxNotebookUserSettings.py**.

11. Occasionally output cells are separated from code cells across page breaks, despite attempts to prevent it via a **\nopagebreak** command which is automatically placed at the end of code cells when output cells are set to be printed.

6 Notes and Suggestions for Python and Sage

Unlike most computer languages, in Python and Sage the indentation of code affects how it is evaluated. Lyx Notebook has some special features for dealing with this. The general idea is to be able to write code in cells in the same way that it would be written in a file (i.e., ignoring newlines). This allows sections of indented code to be separated with space, for clarity.

1. In Python and Sage, whenever the indentation level goes down to zero from a higher level an empty line is automatically inserted and evaluated. If a colon is found on a line — and it is neither inside a comment or string nor between

parens, brackets, or curly-braces — then the indentation level is artificially increased. This latter rule is to allow, for example, one-line **if** statements.[10]

2. A zero-indentation-level no-op (**pass**) is automatically evaluated at the end of each code cell, so cells always start and end at the zero indentation level. (This option can be changed, perhaps to allow cells at different indent levels, but the semantics and operational details become more problematic.)

3. In evaluating Python and Sage cells, implicit line-continuations are calculated.[11] Therefore, lines which are carried-over to the next line do not necessarily need to be explicitly continued with the "\" continuation character. The usual Python rules for implicit line-continuations apply.

7 Running in Silent Mode

Interactive interpreters tend to echo the types and/or values of variables which are entered on a line, and the signatures of functions, etc. For example:

Python

```
# assign a variable
x = 5
x

# define a list
lst = ["a", "b"]
lst

# define a function
def sqr(x): return x*x
sqr(2)
sqr
```

```
5
['a', 'b']
4
<function sqr at 0xb77181b4>
```

This behavior is similar to the print function, though not identical.

Lyx Notebook does not try to modify this type of program behavior or the appearance of the output. This behavior is internal to the interpreter itself. Most interpreters which do this kind of thing will have an internal command to modify the behavior.

Python and Sage can be set to silent mode by executing the following code:

[10] The actual indentation-level calculation is no longer correct, but the transitions down to zero are not affected by that.

[11] This actually needs to be calculated in order to correctly keep track of indentation. This is not required for languages where the semantics are independent of indentation levels.

```python
import sys
def silentDisplay(x): pass
sys.displayhook = silentDisplay
```

Python

Notice the way the output cell appears when the cell was evaluated but no output was produced.

Now we re-run the commands above which produced output, after evaluating the previous cell to switch to silent mode.

```python
# repeat the earlier code after silencing the displayhook
x
lst
sqr
print "An explicit 'print' is now required:", x, lst, sqr
```

Python

```
An explicit 'print' is now required: 5 ['a', 'b'] <function sqr at 0xa7437d4>
```

Notice that output now occurs only on an explicit **print** command. Silent mode in Python is especially useful when generating plots with Matplotlib, since otherwise a great deal of extraneous information is written to the output cell.

The Scala command to silence the output is:

```
:silent
```

Scala

```
Switched off result printing.
```

If silent mode is the preferred default then these commands can be placed in Lyx Notebook init cells and always evaluated first. Alternately, many interpreted languages allow for the default execution of user-specified initialization code. For Python the environment variable **PYTHONSTARTUP** can be set to the name of a file containing Python code which will be automatically run each time the interpreter is started in interactive mode. Such a file is typically named something like **~/.pythonrc.py** or **~/.python-startup.py**, but it could be named anything.

8 Graphs and Plots

One of the most useful applications of Lyx Notebook is in creating and perfecting graphs to insert into Lyx documents. Each interpreted language obviously has its own methods for generating graphs. The focus here is on Python with Matplotlib, but the same general principles should hold across the different systems.

The main feature which Lyx Notebook provides for inserting computed figures and graphs is the "insert most recent graphic file" command. This command searches the full directory subtree corresponding to the Lyx file being edited in the current buffer, looking for graphics files. The most recent one found is inserted into Lyx as a graphics inset. This works well when a Lyx Notebook cell-evaluation just wrote the

graphics file. Graphics files can be organized into subdirectories, if desired, since the full subtree is searched.

In this section we will generate two plots using the Python Matplotlib program[12] and inserted into the document. First, however, some initialization commands are defined in an init cell and evaluated. These settings (other than the imports) can alternately be defined in a **matplotlibrc** file.[13]

The goal is to generate publication-quality plots. The Latex fonts are used in the plots since it generally looks better when the text in a plot has the same font as the surrounding text (the plots are, however, slower to generate). As a general note, the default Computer Modern fonts in Lyx do not look good in PDF and should be avoided. It is better to use the Latin Modern font or some other Type 1 or True Type font.

Python

```python
import matplotlib.pyplot as plt # import pyplot with alias plt
import numpy as np

# set the backend
plt.rcParams["backend"] = "PDF" # for nice PDF output with Latex
# plt.rcParams["backend"] = "PS" # for nice Postscript output with Latex
# plt.rcParams["backend"] = "Agg" # for nice interactive output with Latex

# use the same Latin Modern font as the surrounding document
plt.rcParams["text.usetex"] = True # use Latex for all text handling
plt.rcParams["font.family"] = "serif"
plt.rcParams["font.serif"] = ["Latin Modern Roman"]
plt.rcParams["font.size"] = 15 # slightly larger fonts (12 is default)
plt.rcParams["ps.usedistiller"] = "xpdf" # fix some ugly Postscript output
plt.rcParams["legend.fontsize"] = "small" # default "large", "medium" OK

# some image resolution settings
plt.rcParams["savefig.dpi"] = 200 # default dots per inch is 100
plt.rcParams["ps.distiller.res"] = 6000 # distiller's dpi, default 6000
# the larger the figure size, the smaller the fonts relative to the figure
plt.rcParams["figure.figsize"] = 8, 6 # size in inches, num pixels with dpi
plt.rcParams["figure.dpi"] = 90 # default 80, larger increases preview size
```

Now that the basic setup is out of the way, we create a simple plot. This plot is

[12] The Matplotlib package allows for several different styles of usage. In the examples here we use the style of calling commands in the outer, Pyplot namespace (with only a **plt** prefix). Such commands are always applied to the *current* figure and/or axes. Another style is to assign figures and axes to variables, and then to use those variables as the prefixes in setting the properties. The styles can be mixed, though it may not be a good practice. The current figure can always be obtained by the **plt.gcf()** command, and the current axes can be obtained by the **plt.gca()** command. Note that Pylab is just a convenience import which loads both Pyplot and Numpy into a common namespace.

[13] See http://matplotlib.sourceforge.net/users/customizing.html.

just two sine waves with different phases which are labeled as current versus time. The result is saved to a file, which will be inserted into the document below. Note that Matplotlib plots which are saved in the EPS or PDF file format tend to look best when inserted in a Latex document. The EPS format is preferred for Postscript output and the PDF format is preferred for PDF output, but Lyx will automatically do the required conversions.

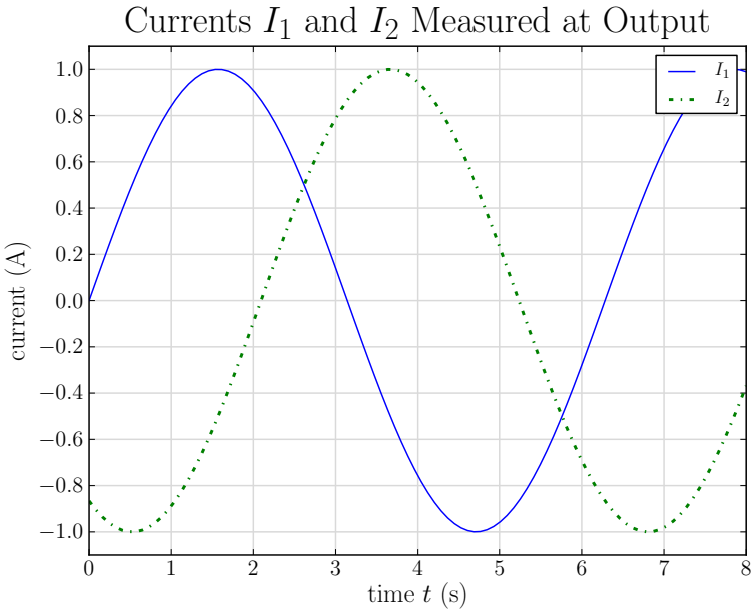
In addition to being saved to a file, the generated plot is also previewed using the Matplotlib **show()** function (this also works in Sage if the appropriate backend is defined). The usual preview window pops up when the function is called. Be sure to save to a file before using show, since the show command destroys the current figure and creates a new one.

Python

```
# Define some parameters to allow for easy changes.
xNumPoints = 100 # how many points to plot for the x dimension
xLow = 0; xHigh = 8; xMinTick = xLow; xMaxTick = xHigh; xTickStep = 1
yLow = -1.1; yHigh = 1.1; yMinTick = -1.0; yMaxTick = 1.0; yTickStep = 0.2

# Set up the grid, labels, ticks, etc. for the figure.
plt.clf() # clear the current figure
plt.grid(True, color="0.85", linestyle="-", linewidth=1) # light gray grid
plt.gca().set_axisbelow(True) # otherwise grid is in front of plotted lines
plt.xlabel(r"time $t$ ($\mathrm s$)", fontsize="large")
plt.xlim(xLow, xHigh)
plt.xticks(np.arange(xMinTick, xMaxTick+xTickStep, xTickStep))
plt.ylabel(r"current ($\mathrm A$)", fontsize="large")
plt.ylim(yLow, yHigh)
plt.yticks(np.arange(yMinTick, yMaxTick+yTickStep, yTickStep))
plt.title(r"\Huge Currents $I_1$ and $I_2$ Measured at Output")

# Generate the x points and two sets of y points and plot them.
xVals = np.linspace(xLow, xHigh, xNumPoints) # equally-spaced in [xLow,xHigh]
y1Vals = [ np.sin(x) for x in xVals ]
y2Vals = [ np.sin(x + np.pi/0.3) for x in xVals ]
plt.plot(xVals, y1Vals, label=r"$I_1$", color="b") # plot the first sine wave
plt.plot(xVals, y2Vals, label=r"$I_2$", color="g",
         linestyle="-.", linewidth=2)             # plot the second sine wave
plt.legend() # create the legend

# Save the figure. Use eps or pdf for best Latex appearance.
# The bbox_inches setting avoids large blank areas around the plot;
# pad_inches controls the amount of padding when "tight" is used.
plt.savefig("phaseWaves.pdf",format="pdf",bbox_inches="tight",pad_inches=.1)
plt.show() # caution: show() creates a new figure, so call after savefig
```

After the cell above is evaluated the plot has been saved in the file **phaseWaves.pdf**. This file is then inserted into the document below. In Lyx this was done simply

by starting a new paragraph, running the Lyx Notebook command "insert most recent graphic file," and then setting the paragraph properties to be centered without indentation. The initial width in the inserted graphics inset is set to 5 in, but that can, of course, be adjusted in Lyx by clicking on the graphics inset. A small protected space inset with a custom value has also been inserted after the plot (in the same paragraph) in order to make the centering look better (correcting for the asymmetry of axis labels appearing on the left but not the right).



A nice feature is that the graphic preview which is displayed in Lyx is usually automatically updated each time the code cell which writes the file is re-evaluated. This makes it convenient to fine-tune the appearance of a graph by tweaking the code and then reevaluating the cell. Sometimes, though, an explicit reload is required (from the right-click menu).

Next, we create a more complex plot with two subplots. The generated output will be placed into a figure float. This plot makes use of the SciPy statistical routines to plot the PDF and CDF of a normal probability distribution. Equations are included in the legends to illustrate the feature, but they would probably at least be simplified in a plot which was more than just an example.

Python

```
from scipy.stats import norm # import the SciPy normal distribution

# Define some parameters to allow for easy changes.
xNumPoints = 100 # how many points to plot for the x dimension
xLow = -4; xHigh = 4; xMinTick = xLow; xMaxTick = xHigh; xTickStep = 1
xBinStep = .25 # size of the histogram bins for the x values
yLow = 0; yHigh = 0.7; yMinTick = yLow; yMaxTick = 0.6; yTickStep = 0.1
```

```
# Begin the actual plotting commands...
plt.clf() # clear the current figure

# Create the first subplot as the current axes (of the current figure).
plt.subplot(121, aspect="auto") # 1 row, 2 cols, these are axes num 1
plt.xlabel(r"$x$")
plt.xlim(xLow, xHigh)
plt.xticks(np.arange(xMinTick, xMaxTick+xTickStep, xTickStep))
plt.ylabel(r"density value")
plt.ylim(yLow, yHigh)
plt.yticks(np.arange(yMinTick, yMaxTick+yTickStep, yTickStep))
plt.title("PDF and 100-Sample Histogram")
normalizeLatex = r"{\scriptstyle{1\over(2\pi)^{-1/2}}}"
gaussLatex = "$" + normalizeLatex + r"\,{\mathrm e}^{-x^2/2}$"

xVals = np.linspace(xLow, xHigh, xNumPoints) # equally spaced in [xLow,xHigh]
yVals = [ norm.pdf(x,0,1) for x in xVals ]
plt.plot(xVals, yVals, label=gaussLatex)

samples = [ norm.rvs() for i in range(200) ]
bins = np.arange(xLow, xHigh, xBinStep)
plt.hist(samples, bins=bins, normed=True, facecolor="orange",
         label="Histogram")
plt.legend()

# Create the second subplot as the current axes (of the current figure).
yLow = 0; yHigh = 1.5; yMinTick = yLow; yMaxTick = 1.0; yTickStep = 0.2
plt.subplot(122, aspect="auto") # 1 row, 2 cols, these are axes num 2
plt.xlabel(r"$x$")
plt.xlim(xLow, xHigh)
plt.xticks(np.arange(xMinTick, xMaxTick+xTickStep, xTickStep))
plt.ylabel(r"cumulative probability")
plt.ylim(yLow, yHigh)
plt.yticks(np.arange(yMinTick, yMaxTick+yTickStep, yTickStep))
plt.title("CDF and 100-Sample Histogram")
gaussLatex = "$" + normalizeLatex \
             + r"\int_{-\infty}^{x}{\mathrm e}^{-s^2/2}{\mathrm d}s$"

yVals = [ norm.cdf(x,0,1) for x in xVals ]
plt.plot(xVals, yVals, label=gaussLatex)

bins = np.arange(xLow, xHigh, xBinStep)
plt.hist(samples, bins=bins, normed=True, facecolor="orange",
         cumulative=True, label="Histogram")
plt.legend()

# set up the combined plot and write it to a file
```

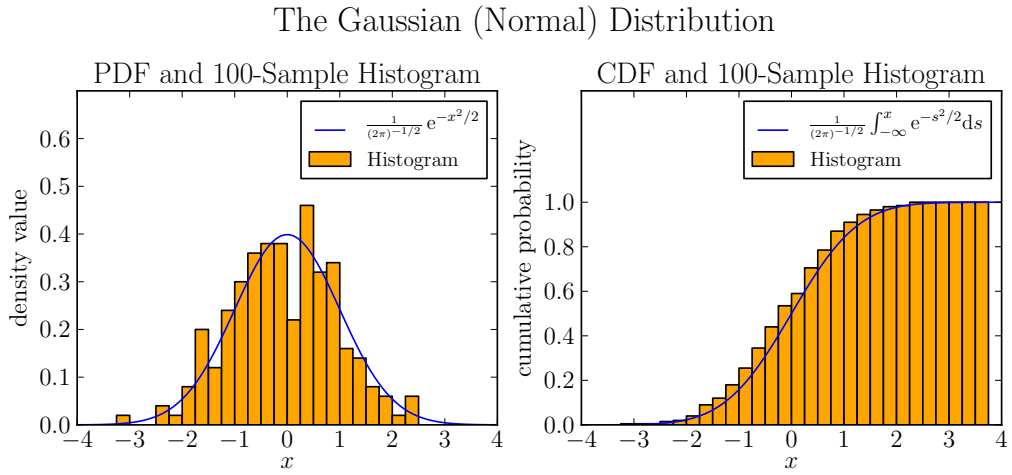


Figure 2: An example of a figure inserted into a figure float.

```
fig = plt.gcf() # get current figure, call it fig
fig.set_size_inches(10,4)
supTitle = fig.suptitle("\huge The Gaussian (Normal) Distribution")
fig.subplots_adjust(top=0.80) # small fraction, bigger gap
fig.savefig("distributions.pdf", format="pdf", bbox_extra_artists=[supTitle],
    bbox_inches="tight", pad_inches=0.0) # extra_artists or suptitle clips
```

The output is shown in Figure 2. A small protected space inset has again been inserted after the plot (inside the float) in order to make the centering look better.

These plots could, of course, have been inserted anywhere in the document. They could even be inserted into a different buffer (i.e., into a different document open in the same Lyx process). The important thing is to use "insert most recent graphic file" just after evaluating the cell, so that the newly-written graphics file is the one which is found.

9 Consistency with Listings Insets

Sometimes a paper will have some cells which are intended to be evaluated, and other cells which are merely explanatory. For consistency, it would be nice for these non-evaluating cells to be formatted in the same style as the code cells. That can be accomplished using the built-in listings insets in Lyx.

The Lyx Notebook program defines a Listings style for each type of code and output cell. The naming convention for Python 2, for example, generates the following six Listings styles:

lyxNotebookNoColorInitPythonTwo,
lyxNotebookNoColorStandardPythonTwo,

lyxNotebookNoColorOutputPythonTwo,
lyxNotebookColorInitPythonTwo,
lyxNotebookColorStandardPythonTwo, and
lyxNotebookColorOutputPythonTwo.

The same convention holds for the other cell languages (with the obvious substitutions).

These strings can be set as the style in a listings inset. From the listings inset's right-click menu, go to the **Settings** ▷ **Advanced** page. Enter **style=<nameOfStyle>**, substituting the name of a style in the format above. Note that many of the special Lyx Notebook Latex commands (such as those involving **true** and **false** values) will not work for the built-in listings insets.

A convention of using a doubled frame on all sides can be used to distinguish a formatted non-evaluating code block from any of the evaluating cells. Two additional styles are defined for each cell language,[14] using this convention:

lyxNotebookNoColorNonEvalPythonTwo, and
lyxNotebookColorNonEvalPythonTwo.

Here is an example:

```
# This is a Lyx Listings inset with "More Parameters" set to:
#     style=lyxNotebookColorNonEvalPythonTwo
# on the advanced settings page.
print "This code will not be evaluated."
```

By default the built-in listings insets are displayed in a normal-sized font inside Lyx, while the custom Lyx Notebook cells use a small font in their Lyx display. A simple module is included in the Lyx Notebook setup which modifies the built-in Listings inset to also use a small font in the Lyx display. All you need to do to make the change is to go to **Document** ▷ **Settings** ▷ **Modules** and insert the module called "Listings with Small Font."

10 Customizing the Formatting

The Lyx Notebook system was designed to provide usable default values for various settings. Nevertheless, there are times when customization is necessary or desirable. This section covers the customization options. Don't forget that you can usually customize the languages themselves, such as defining default start files for things like setting silent mode and setting up graphics parameters. (This was discussed in Sections 7 and 8.)

[14] In Listings in general (such as in an ERT) this can be done simply by using an init or standard style and then redefining the frame. In Lyx 2.0.3 and earlier this cannot be done with the built-in listings insets because the order of additional parameter settings is always alphabetized on the advanced settings menu.

10.1 Customizing the Key Bindings

The key bindings for Lyx Notebook commands can be arbitrarily defined, but they must be consistently defined in two different places. First, they must be defined in the file **keymap.py** in the Lyx Notebook source directory. That is the version which the program itself reads. Second, they must be defined in the file **lyxNotebookKeyBindings.bind** which is generated by the setup program and placed in the user's Lyx home directory (usually **~/.lyx**).[15] Note that there is a slightly different convention for specifying key modifiers (such as the shift key and the alt key) in the two different files. See the comments in the **keymap.py** file.

10.2 Customizing the Print Format

There are several ways to customize the formatting of the cells in the final, printed output. This section documents the options which are available.

1. **Color vs. non-color**. Often a document needs to look good when printed both in color and in grayscale. The Lyx Notebook program is set up to print in both formats, but the default is color. To switch to grayscale, put **\def\lyxNotebookNoColor{true}** in the preamble (or define from an ERT Tex box).

2. **Suppressing cell labels**. By default a tiny, rotated descriptive label is printed to the right of all the code cells. This option can be turned off by putting **\def\lyxNotebookNoCellLabels{true}** in the preamble (or define from an ERT Tex box).

3. **Modifying Listings parameters**. The Listings package for Latex has many, many options. Lyx Notebook allows for arbitrary **lstset** commands to be defined for cells, overriding the default values. To define listings settings for all the cells, put, for example,

   \def\lyxNotebookLstsetAllAll{\lstset{frame=TBLR}}

   in the preamble.[16] This sets all the frames to have doubled borders (and is only an example, since it does not look especially good). Listings settings which are specific to a particular type of cell can also be defined. For example, the definitions

   \lyxNotebookLstsetInitPythonTwo,
   \lyxNotebookLstsetStandardPythonTwo,
   \lyxNotebookLstsetOutputPythonTwo,
   \lyxNotebookLstsetAllPythonTwo,

[15] Strictly speaking, they just need to be bound to the Lyx LFUN **server-notify**.

[16] More generally, any command could be used instead of **\lstset** and it would be executed before each cell.

\lyxNotebookLstsetInitAll,

etc., can be created (and similarly for Python 3, Scala, R, etc.) These can also be defined from ERT Tex boxes. Note that each redefinition overwrites the old one. Setting things like background colors and rounded corners is possible in theory, but in practice such commands do not always behave as expected, especially when frames are used. Sometimes a lot of code-tweaking is necessary.

4. **Basic style settings**. Users will often want to change the basic style of the Listings Latex formatting for cells (the Listings **basicstyle** settings). Those settings include, for example, the basic size of the font and the font family. This can be done with a **lstset** command, as in item 3, but for convenience the default setup uses some modifiable definitions. For example, the font size alone can be changed simply by redefining **lyxNotebookFontSize**. These definitions and their current defaults (for most languages) are given in Table 2.

5. **Bold in typewriter fonts**. The default Type 1 Latex typewriter font (Latin Modern Typewriter) does not have bold, so by default the typewriter font is not used non-color formatting. The typewriter fonts with bold generally look better in non-color Listings code blocks; in color it is a matter of taste, but they also tend to look better. If you have the LuxiMono or the BeraMono font installed you can select and use one of them as the **Document** ▷ **Fonts** ▷ **Typewriter** font. (If installed, one should be listed there as an option.) Set the scaling of the LuxiMono font to around 87% and the BeraMono font to around 84%. This document was formatted with the BeraMono typewriter font, setting the following in the header:

```
\def\lyxNotebookNoColorCodeFontFamily{\ttfamily}
\def\lyxNotebookNoColorOutputFontFamily{\ttfamily}
\def\lyxNotebookFontSize{\small}
```

This uses the typewriter fonts for all cells, and increases the font size to small to compensate for the fact that the BeraMono fonts appear smaller than the default typewriter font.[17]

6. **Non-printing cells**. One way to make Lyx Notebook cells non-printing is to turn off label printing and set the Listings option **print=false**. This may not give the desired results, however, since it can still leave space in the document where the cells used to be. A different method is provided which essentially redefines the Latex for all the cells to be equivalent to a comment-block (from

[17] In the Bera Mono font an underscore, such as in a variable name, can appear faint. Another useful command uses the literate programming feature of Listings to convert all underscores to bold underscores. This mapping can be enabled by placing the following command in the preamble:

```
\def\lyxNotebookLstsetAllAll{\lstset{
    literate={\_}{{{\bfseries\textunderscore}}}1
}}
```

Latex Variable	The Listings **basicstyle** settings it is included in:
\lyxNotebookFontSize	All the cell types. The default is **\footnotesize**, but **\small** is slightly larger and sometimes looks better.
\lyxNotebookColorCodeFontFamily	All code cells when the color option **\lyxNotebookNoColor** is set to **false**. The default is **\ttfamily**.
\lyxNotebookColorOutputFontFamily	All output cells when the color option **\lyxNotebookNoColor** is set to **false**. The default is **\ttfamily**.
\lyxNotebookNoColorCodeFontFamily	All code cells when the color option **\lyxNotebookNoColor** is set to **true**. The default is **\sffamily**.
\lyxNotebookNoColorOutputFontFamily	All output cells when the color option **\lyxNotebookNoColor** is set to **true**. The default is **\sffamily**.
\lyxNotebookLabelFontFormat	This sets the formatting of the small labels to the right of the all cells. The default is the concatenation of **\ttfamily**, **\footnotesize**, and **\bfseries**.

Table 2: The default Listings **basicstyle** setting for each type of cell.

the Verbatim package). In this way, the document can be printed as if the cells were not even there. To use this method, put

\lyxNotebookPrintOff

in either the preamble or in an ERT Tex box. The command

\lyxNotebookPrintOn

turns printing back on, so cell-printing can be turned on and off at various places inside a document. Note that there must be at least one cell in the document or an "undefined" error will occur. These commands can be specialized to apply only to certain types of cells. For example, **\lyxNotebookPrintOffOutputPythonTwo** would turn off printing for all Python 2 output cells following it, at least until the command **\lyxNotebookPrintOnOutputPythonTwo** appeared (or the command to turn on all cell-printing appeared). The general pattern holds in all combinations of the two added specifiers at the end (but "all" is not currently implemented). Again, the document must contain at least one cell of that particular type or an "undefined" error will occur.

7. **Modifying the space before output cells**. The insertion of negative space before the output cells can be suppressed by setting

   \lstset{aboveskip=\medskipamount}

   for output cells, as described in item 3 above. The command above resets the value to the default of the Listings package, but any other length could also be used. This is useful if, for example, you suppress the printing of code cells but still want to display the output cells which the code produced.

10.3 Customizing the Handling of Interpreters

To add a new interpreted language, see Appendix C. There are a few parameters of the interpreters, however, which are easy to change and which users might find useful. Some of these parameters are shown in Table 3. These settings are defined for each interpreter, in a separate section of the file **interpreterSpecs.py** in the Lyx Notebook source directory.

A Defining an LFUN to run Lyx Notebook

The Lyx LFUN named **vc-command** allows arbitrary operating system commands to be executed from within Lyx. This LFUN is bound to the F12 key in the default binding, set to run the Lyx Notebook program. This appendix describes how to create a user-defined LFUN to run Lyx Notebook. For most users this will not be necessary: The long-form **vc-command** LFUN should be sufficient without actually

Parameter	Description
progName	The text of the label which is printed to the right of the cells. The **setup.py** program must be re-run if this is changed, and Lyx should be reconfigured.
mainPrompt	The main prompt of the interpreter; must be reset in the interpreter's startup code if it is changed.
contPrompt	The continuation prompt of the interpreter; must be reset in the interpreter's startup code if it is changed.
runCommand	The operating system command to run the interpreter.
runArguments	A list of arguments to the run command.
startupSleepSecs	The number of seconds to wait for an interpreter to start up before sending anything to it. Should be large enough for Lyx Notebook to work consistently with that interpreter.
promptAtCellEnd	When echo mode is on, whether or not to include the waiting prompt at the end of the cell. Default is **True**.
ignoreEmptyLines	Whether or not to ignore completely blank (all whitespace) lines and not even interpret them. Default is **True**.
runOnlyOnDemand	Whether to start an interpreter only if it is actually needed to evaluate a cell. Default is **True**. If you know you are going to use an interpreter with a slow startup time then setting this to **False** for that interpreter may speed up the first cell evaluation. This only applies to the current buffer when Lyx Notebook is started, not to all buffers.

Table 3: Some user-settable interpreter properties.

creating a user-defined command. Nevertheless, the general principles for running Lyx Notebook from an LFUN apply in all cases.

1. Note that the shell script **lyxNotebookFromLFUN** in the Lyx Notebook directory simply runs the Lyx Notebook program in the background.[18]

2. In your user Lyx directory (usually **~/.lyx**), locate or create the file **default.def** in the subdirectory **commands**. If necessary, create that subdirectory also.

3. Add the following line to that file:

   ```
   \define start-lyx-notebook "vc-command U $$p lyxNotebookFromLFUN"
   ```

   replacing **lyxNotebookFromLFUN** with the full absolute pathname of the script (if it is not in the default execution path).[19]

4. Reconfigure Lyx and restart.

5. Now open the command mini-buffer in Lyx (either from the **View ▷ Toolbars** menu or using the shortcut M-x). In that window you can now type

   ```
   call start-lyx-notebook
   ```

   and the the program should start running in the background.

Once the LFUN is defined then the usual methods can be used to bind it to either a key or to a menu item. Usually this can be done without creating a user-defined command. The default keymap for Lyx Notebook binds the function key F12 to only *the quoted part of the indented command in step 3* above. This gives a much easier setup, since the **default.def** file does not need to be created/modified.

B How It Works

This appendix gives a overview of how the Lyx Notebook program is implemented. The program is written in Python 2.x, but most features (which have been tested) also work when it is run in Python 3.x.

The Lyx Notebook program has three main parts: 1) a module which handles interactions with Lyx via the Lyx server, 2) a module which spawns an interpreter process and manages interactions with it, and 3) an overall controller module which

[18] There is an added complication in that the script cannot just run **python lyxNotebook &**, since that crashes on cell updates (due to it having no tty when run from inside Lyx). So the script uses **ps** to look up the tty of the running Lyx process and then runs the command with both stdout and stderr redirected to that tty. If there is no such process (such as when Lyx was run from the start menu) it tries to open a terminal window and run the program in that.

[19] Note that the **U** is a dummy argument to **vc-command**, the second argument is the directory to change to (**$$p** evaluates to the current document's directory), and the third argument is the path of the command to run.

uses the previous two modules to implement the basic functionality of the system. In the following paragraphs each part is briefly summarized.

B.1 External Interpreter

The module for interacting with an interpreter is in the file **`externalInterpreter.py`**. The main class is called **`ExternalInterpreter`**. This class takes a specification for an interpreter as an argument and starts up an instance of the interpreter. The interpreter is executed in a pseudo-tty via **`pty.fork()`**. This allows the interpreter to run as if it were in an ordinary terminal. When a line of code is interpreted in an external interpreter it is first written to the running process. Then the output is read until an input prompt is detected (so it knows the evaluation has completed). There can be multiple instances of **`ExternalInterpreter`**, each of which runs a separate interpreter process. (The controller module creates a list of such objects, one for each language.)

B.2 Interacting with Lyx

The module for interacting with Lyx is in the file **`interactWithLyxCells.py`**. The main class is called **`InteractWithLyxCells`**. On initialization it checks that the Lyx server pipes are present. If they are, it opens them (and uses them to interact with Lyx when the class' methods are invoked). There is a method to evaluate an arbitrary LFUN, as well as specialized methods for commonly-used or more complex operations. There are methods to goto the next and previous cells of a given basic type, as well as to read the contents of a cell and write it back.

Due to limitations in currently-available Lyx LFUNs, some workarounds are used to implement some of this functionality. To move between cells a magic cookie (currently with default value is the string "zZ4Qq" except with 3 instead of 4) is inserted into all the cells, via the **`inset-forall`** LFUN. A forward or backward search is then done for the cookie, which moves the cursor-point to the correct inset. Another **`inset-forall`** then deletes all the cookies.

To read the contents of a cell a cookie is first inserted into the current cell only. The buffer is then saved or exported to Latex. That saved or exported file is read by Lyx Notebook, looking for the cell with the cookie. When it is found the text associated with the cell is returned. In order to insert text into a cell it is first written to a temporary file, which is then read into Lyx using a **`file-insert-plaintext`** LFUN.

In order to determine whether or not the cursor is in a cell the **`server-get-layout`** function is called. The custom Flex insets have Plain Layout, as opposed to the Standard Layout of their surrounding text. This will fail in some instances, however. Ordinary listings insets are also Plain Layout, so avoid running cell-evaluate commands when the cursor is in such an inset (the action is not meaningful, but

the program cannot detect that and ignore it, like it does for any layout other than Plain Layout).

The **waitForServerNotify** method is essentially a busy-wait loop that waits for a **NOTIFY** event from the Lyx server. The Lyx Notebook function keys are all bound to **server-notify**, so this method just waits until a Lyx Notebook keyboard command is pressed.

B.3 Controller Module

The controller module is in the file **controllerLyxWithInterpreter.py**. Its main class is called **ControllerLyxWithInterpreter**. On startup it creates an instance of **InteractWithLyxCells** as well as a list of **ExternalInterpreter** classes (one for each interpreter type). This high-level module then waits for a Lyx command and implements each one it receives, making use of the lower-level modules. It is essentially a command loop, getting commands Lyx Notebook commands and then executing them.

The file **lyxNotebook**, which users execute, is just a simple script which imports this module and runs it (after making sure that no other instance of Lyx Notebook is running). The script **lyxNotebookFromLFUN**, which is called by Lyx LFUNs, is just a wrapper for calling **lyxNotebook** with stdout and stderr redirected to the terminal associated with the running Lyx process. If the Lyx process is running without a terminal, such as when it is started from a menu, the **lyxNotebookFromLFUN** opens a terminal window and runs the program with that terminal as the I/O.

The basename of the command which was used to execute the running Lyx process (as in the output of the **ps -eo command**) is assumed to be **lyx**. This setting can be changed in the file **lyxNotebookUserSettings.py**. Setting this string to a "wrong" value will cause the script to always open a terminal window (except when it is run from the command line, in a terminal).

C Adding a New Interpreter

To add a new interpreter you need to do the following:

1. Create a new section of the file **interpreterSpecs.py** corresponding to the interpreter which you wish to add.

2. Re-run the **setup.py** program.

See the documentation in the **interpreterSpecs.py** file for more information on that step. You'll need to define some properties of the interpreter, as well as define the Listings format which will be used to format the cells. It is probably easiest to copy an existing one (maybe the whole R section, for simplicity) and then change the relevant variable names and values. Don't forget to append your new definition to the **allSpecs** list after creating it.

D Possible Future Enhancements

This section briefly discusses possible future features. One limitation on adding features is that there is currently no way to pass parameters to the custom Flex insets (in a way similar to, say, the built-in listings insets).[20] This can result in a proliferation of many different types of insets, as opposed to only having a few types of insets with settable parameters.

1. *Printing vs. non-printing cells.* This can currently be turned on and off with ERTs, as described in Section 10, but it would be nicer as a property of the cells themselves. This could be done now by generating a non-printing version of each type of inset, for each language. There is already a proliferation of cell types, however. If the Flex custom insets in Lyx are enhanced at some point to have settable parameters (like the listings insets do, for example), then this option could be set and changed there (along with various other properties).[21]

2. *Output cells which print in Latex math mode.* This could be done just by allowing code cells to write output to math insets as well as to output cells, i.e., to write output to a math inset if one is placed immediately after a code cell. (This is probably possible after a bug fix to the Lyx server which is scheduled for 2.0.4.) For the time being, Lyx Notebook will happily write into the built-in listings insets if they directly follow a code cell. Therefore one possible workaround is to use a listings inset, setting **`mathescape=true`** on the advanced settings, and selecting "inline" on the main settings if that is desired. There is no preview, but the math is printed in the Latex version. Here is an example:

```python
print "$x^2$"
```

   x^2, where the listings inset is inline. The code cell could have been made non-printing via an ERT, and only the math expression would appear (inline) in the final text.

3. *Inline code cells, like the inline listings of the Listings package.* This could be done now, but each type of code inset would have to have a separate inline

[20] The properties of the built-in listings insets can be changed by using the customization **`InsetLayout Listings...End`** in, say, a module. This could probably be set up to turn built-in listings insets (with their settable parameters) into Lyx Notebook cells when certain parameters are set (such as the style). Unfortunately, there can only be one type of label on all the listings insets (as far as I know), which would not differentiate the different types of cells and the different languages inside Lyx. It would be nice to be able to define your own modified copies of the built-in listings inset, such as with something like **`InsetLayout Listings:MyListing...`**

[21] An alternate way to get this effect is to place cells inside a special branch such as one named "Nonprinting Cells." This branch would be deactiviated most of the time. In preliminary testing the Lyx Notebook cells seem to work when inside the branch insets, but the output cells currently also have to be inside the branch.

version. One option would be to use non-printing cells which write Latex math output to math insets (when that is implemented) or to inline listings insets.

4. *Formatting of the code cells in Lyx, not just in the printed version.* On evaluation the code cells are rewritten anyway, so they could be processed through, say, Pygments to add highlighting. This would make things more WYSIWYG, but the interactions with **`PassThru`** would need more investigation. It may require some kind of "partial **`PassThru`**" or a "**`PassThru`** escape."

5. *Preamble command to specify a startup file.* Most interpreters have a setting to first execute a file and then accept commands interactively. A special command in the preamble could be used to modify the startup and execute a specified file for a given interpreter. A similar effect can already be achieved by using an interpreter's predefined method for a startup file (like the **`PYTHONSTARTUP`** environment variable in Python), or by changing the startup command for the interpreter to first execute some given file in the current directory.

6. *Dynamic key-binding.* This may not be possible with the current Lyx LFUNs, but it would be nice if only the F12 key were permanently bound: to start up the Lyx Notebook program. The program itself could then dynamically modify the bindings and bind, for example, the other function keys to Lyx Notebook functions. The Shift+F12 command would then revert to the previous bindings.