LastWave 2.0 Documentation

Emmanuel Bacry
CMAP, Ecole polytechnique, 91128 Palaiseau Cedex, France
email : lastwave@cmap.polytechnique.fr
web : http://www.cmap.polytechnique.fr/~ bacry/LastWave

This documentation includes full description of

- **LastWave Kernel 2.0**, Author: E.Bacry
- **disp 2.0, misc 2.0 and terminal 2.0 packages**, Author: E.Bacry
- **Signal package 2.0**, Authors: E.Bacry, N.Decoster and X.Surraud
- **Image package 2.0**, Authors: E.Bacry and J.Fraleu

The documentation pages of the other packages are not included in this manual, they should be downloaded separately.

Last but not Least... I am a researcher in Applied Mathematics and I was never satisfied by the signal processing softwares that were available. Too expensive, too hard to use, too many bugs or ... whatever ! I have decided in 1997 to write a brand new software for signal processing. A software that is free (GNU License), runs on different computers and in which it is very easy to include one's own code.
I did it. It is called LastWave and there have been several releases since the orginal 1.0 version. The last one before this one was version 1.7. The latest version is version 2.0 which includes major improvements compared to 1.7. I really hope you will enjoy using it !
You'll see, it is very powerful and very easy to install, to use or to include your own code. Of course I'm interested in any comments (bad or good) that you would like to make about it. Feel free to add some new commands or improve some old ones or even write your own packages (yes, it is very easy !). It will be my pleasure to include them in the next release of LastWave.
I wrote the core of LastWave along with some packages. However, most of the numerical packages were written by some other researchers. The names of the authors of each package along with some references can be found on the next page.

What is LastWave about ? LastWave is a signal processing (wavelet oriented) software. It is written in C and runs on both X11/Unix (no Motif or fancy toolkit are necessary just the basic X11 library !), Macintosh computers with MacOS 9 and Classic or MacOS X with XFree86 (a native MacOS X version will be available soon) and Windows computers with CygWin. It has been designed to be used by anybody who knows about signal processing and wants to play around with wavelets and wavelet-like techniques. However, it can be used for many other purposes (just as a regular scripting language for instance).
It mainly consists in a powerful command line language which includes (matlab-like) numerical facilities with high level structures (such as signals, images, wavelet transforms) and a high level object-oriented graphic language which allows to display both some simple structures (e.g., buttons, strings, text using any font, ...) and some complex structures (signals, images, wavelet transforms, extrema representation, ..). All these graphic objects can be fully controlled (along with the mouse behavior) via the command language. Moreover, LastWave can generate postscript files of anything you draw. One can very easily add some new commands in LastWave using either the command language itself or the C-language. These new commands along with some eventual newly defined graphic objects can be grouped into a LastWave package that can be loaded (at run-time) by the user if needed. The core of LastWave includes all the basic commands of the command line language along with the graphic object-oriented language. Several other packages have already been added to LastWave allowing high level signal processing such as (orthogonal and non-orthogonal) wavelet transforms (1D and 2D), extrema representations of wavelet transforms, fractal analysis, matching pursuit,...

Please, be kind enough to reference LastWave in any article that you publish for which you used it. Just refer to the web page :

http://www.cmap.polytechnique.fr/˜ bacry/LastWave

Emmanuel Bacry

LastWave packages

Here is the list of the different packages in LastWave 2.0 along with their authors and some references about the methods that are implemented in the package. **The manual pages of the numerical packages are not included in this manual, they should be downloaded separately**.

Authors of the packages

The different packages were written in collaboration with

- **B. Audit**, Computational Genomics Group, EMBL-European Bioinformatics Institute, Wellcome Trust Genome Campus, Hinxton, Cambridge CB10 1SD, UK.
  e-mail: audit@ebi.ac.uk
  web: http://www.ebi.ac.uk/~audit/
- **G. Davis**, Math Department, Dartmouth College, Hanover, NH 03755.
  email : gdavis@cs.dartmouth.edu
  web : http://www.cs.dartmouth.edu/~gdavis
- **N. Decoster**, Phd sudent at CRPP Avenue Schweitzer, 33600 Pessac France.
  email : decoster@crpp.u-bordeaux.fr
- **J. Fraleu**, Phd student at CMAP, Ecole Polytechnique, 91128 Palaiseau Cedex France.
  email : fraleu@cmap.polytechnique.fr
- **R. Gribonval**, Researcher at IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
  email : remi.gribonval@inria.fr
  web : http://www.irisa.fr/metiss/gribonval
- **J.Kalifa**, LetItWave, Ecole Polytechnique, 91128 Palaiseau Cedex France.
  email : jerome.kalifa@polytechnique.fr
- **W.L.Hwang**, Researcher at the Institute of Information Science,Academia Sinica,Taipei, Taiwan.
  email : whwang@iis.sinica.edu.tw
- **E. Le Pennec**, Phd student, Ecole Polytechnique, 91128 Palaiseau Cedex France.
  email : lepennec@cmap.polytechnique.fr
- **S. Mallat**, Professor at CMAP, Ecole Polytechnique, 91128 Palaiseau Cedex France.
  email : mallat@cmap.polytechnique.fr
- **J.F. Muzy**, Researcher at CNRS, Université de Corse, Quartier Grossetti, 20250, Corte, France.
  email : muzy@univ-corse.fr
- **X. Surraud**, CMAP, Ecole Polytechnique, 91128 Palaiseau Cedex France.

- **C. Vaillant**, Phd student at CRPP, Avenue Schweitzer, 33600 Pessac, France.
  email : vaillant.u-bordeaux.fr
- **S. Zhong**, Researcher at the Department of Civil Engineering and Operations Research, Princeton University, E220 Engineering Quadrangle Princeton NJ 08544.
  email : szhong@chelsea.princeton.edu

The different packages included in this version of LastWave are

Script defined packages

- The *disp* package which is the core of LastWave high-level displays/interactions (author : E.Bacry).
- The *misc* Package that regroups very useful miscellenaous script commands and graphic classes (author : E.Bacry).
- The *terminal* Package that regroups all the bindings for managing terminal history, help and file completion systems (author : E.Bacry).

1d numerical packages

- The *signal* package (authors : E.Bacry, N.Decoster and X. Surraud).
- The *sound* package (authors : E.Bacry and R.Gribonval(for the package and the adaptation of the *sndfile* library). The original *sndfile* library was written by E. de Castro Lopo).
- The *wtrans1d* package which allows a large variety of 1D wavelet transform (authors : B.Audit, E.Bacry, N. Decoster, S. Mallat). This package includes the fast continuous wavelet transform C-library written by N.Decoster. For information about 1d wavelet transforms you can read :
  - *Ten lectures on wavelets*, I.Daubechies, SIAM (1992).
  - *An introduction to wavelets*, C.K.Chui, Academic Press (1992).
  - *Les ondelettes, algorithmes et applications*, Y.Meyer, Armand Colin (1994).
  - *Fractales, Ondelettes et Turbulence : de l'ADN aux croissances cristallines*, A. Arneodo, F. Argoul, E. Bacry, J. Elezgaray and J.F. Muzy, (Diderot Edition, 1995).
  - *A wavelet tour of signal processing*, S.Mallat, Academic Press. (1998)
- The *extrema1d* package which allows dealing with the local maxima of the 1d wavelet transform (authors : B.Audit, E.Bacry, J.F.Muzy and C.Vaillant). For information about extrema of wavelet transforms you can read :
  - *A wavelet tour of signal processing*, S.Mallat, Academic Press. (1998)
- The *wtmm1d* package which implements the Wavelet Transform Modulus Maxima method for fractal anlysis of 1d signals (author : B.Audit). To learn about this method, you could read :

  - *The multifractal formalism revisited with wavelets*, J.F. Muzy, E. Bacry and A. Arneodo, Int. J. of Bifurcation and Chaos 4, 245 (1994).
  - *The thermodynamics of fractals revisited with wavelets*, A. Arneodo, E. Bacry and J.F. Muzy, Physica A 213, 232 (1994).
  - *Fractales, Ondelettes et Turbulence : de l'ADN aux croissances cristallines*, A. Arneodo, F. Argoul, E. Bacry, J. Elezgaray and J.F. Muzy, (Diderot Edition, 1995).

- The *stft* package which implements short time fourier transforms and other time-frequency representations (author : R. Gribonval)

- The *mp* package which implements Matching Pursuit of 1d signals (authors : R. Gribonval, E. Bacry and J. Abadia). To learn about this method, you should read

  - *Matching pursuit with time-frequency dictionnaries*, S.Mallat and Z.Zhang, IEEE Trans. on Sig. Proc. 41 (12), 3397 (1993).
  - *Sound signals decomposition usinga high resolution matching pursuit*, R.Gribonval, P.Depalle, X.Rodet, E.Bacry, S.Mallat, Proceedings Int. Computer Music Conf. (ICMC '96), 293 (1996)
  - *A wavelet tour of signal processing*, S.Mallat, Academic Press. (1998)
  - Phd Thesis (1999), R.Gribonval.
    Available on the web: http://www.irisa.fr/metiss/gribonval

2d numerical packages

- The *image* package (author : E.Bacry and J.Fraleu). This package allows simple manipulation on images.

- The *owtrans2d* package (authors : G.Davis, E.Bacry and J.Fraleu). This package allows to perform the decomposition/reconstruction of images using orthogonal wavelet transform. The original code was written in C++ by Geoff Davis (gdavis@cs.dartmouth.edu http://www.cs.dartmouth.edu/ ~ gdavis) and was translated into C and adapted to LastWave by E.Bacry and J.Fraleu.

- The *compress2d* package (authors : G.Davis, E.Bacry and J.Fraleu). This package allows to perform image compression using uniform quantization of orthogonal 2d wavelet coefficients along with arithmetic coding. The original code was written in C++ by Geoff Davis (gdavis@cs.dartmouth.edu http://www.cs.dartmouth.edu/~gdavis) and was translated into C and adapted to LastWave by E.Bacry and J.Fraleu.

- The *dwtrans2d* package (authors: E. Bacry, J. Fraleu, W.L. Hwang, J. Kalifa, E. Le Pennec, S. Mallat, S. Zhong). This package implements dyadic 2d wavelet transform and extrema reconstruction along with denoising using extrema chains.

  For the last 3 packages, you should read the book

  - *A wavelet tour of signal processing*, S.Mallat, Academic Press. (1998)

Different distributions

Most of LastWave files are machine independent. However, in order to be able to run on different computers, there are about 4 machine dependent files (one for managing graphics, one for audio, one for the terminal and one for system calls).

- The **Unix/X11** and *Macintosh* adaptation has been written by **Emmanuel Bacry**.
- The **Windows/CygWin** adaptation has been written by **Rémi Gribonval**
  Researcher at IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
  email : remi.gribonval@inria.fr
  web : http://www.irisa.fr/metiss/gribonval

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

Version 2, June 1991

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.
If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.
4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.
If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.
It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.
This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail

to address new problems or concerns.
Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Contents

Part I

Introduction

Chapter 1

How to install LastWave ?

1.1 Installing Lastwave on a Macintosh (MacOs 9 or Classic)

If you are just interested in running LastWave, you should just keep the `LastWave 2.0` file and the `scripts` directory (`PPC` only, `68k` users should read remark below) and throw away everything else (except this documentation of course!). If you want to add new C-code, you will need the Code Warrior compiler from Metrowerks version 3.0 (or later).

For earlier version of CodeWarrior If you have an earlier version, you will have to create a new project including all the source files in all of the `src` directories which are in the directories

- `kernel/` : contains all the files related to the core of LastWave
- `unix/` : contains all the machine dependent unix files
- `win32/` : contains all the machine dependent windows files
- `ansi/` : contains some ansi files
- `user/` : contains files that you must change if you add some new C-packages
- `package_<name>/` : contain files related to the package named `<name>`.

and set the include paths to be the corresponding `include` directories (for each `src` directory there is a `include` directory). You will have to add also the libraries `MSL C.PPC.Lib`, `InterfaceLib`, `MathLib` and `MSL RuntimePPC.Lib`. In the `preferences` pannel you should set 2 things which are very important

1) the `Preferred Heap size` should not be less than 8000 (it depends of course on how big the data you are processing are) and

2) the `stack size` should NOT be smaller than 512K.

68K users : it is very easy to make a 68K project from the PPC project (you just have to change the preferences and the libraries). I did it on an early version of LastWave. It worked fine. If you are familiar with Code Warrior, it should be very easy for you to do.

There is just one little thing you have to do before compiling the project : you have to replace one line in the file `macwindow.c`. Just look for the `68K-Warning` line in this file.

Running LastWave Before running LastWave (i.e., double clicking on the file `LastWave 2.0`, you should know that each time you start up LastWave, it executes the script file named `startup` (in the `scripts` directory). I highly recommend that you use the one that I wrote before writing your own (it adds a lot of basic functionalities that you will need). Understanding how it works is an excellent exercise for practicing programming new scripts. Before you enjoy playing around with LastWave, I would like to make a few comments

1) An history file will be created by Lastwave that will be in the same directory as the application `LastWave 2.0` and whose name is `LastWaveHistory`.

2) Most Macintosh computers do not have a 3 button mouse. If you are as lucky as I am and do have 3 buttons, you should assign the left button to the regular mouse button, the middle button to `Cmd-Left Arrow` and the right button to `Cmd-Right Arrow`. In the startup file, LastWave executes the command `system mouse3` because it assumes the mouse has 3 buttons. If you only have 1 you must uncomment the line `system mouse1` and comment the line `system mouse3`. In the case you have just one button, it will correspond (in LastWave) to the left button. Moreover, you can simulate a middle button click by pushing the mouse button while the `Opt` key is down and a right button key by pushing the mouse button while the `Cmd` key is down. In this case, you will not be able to use the `Opt` key as a modifier key of LastWave as explained later.

3) If your keyboard does not send KeyUp events you should uncomment the line `system noForceKeyUp` in the startup file and comment the `system forceKeyUp` line. If you don't, it is generally not very important. However, if you use a 3 button mouse and if the keyboard does not send KeyUp events, each time you hit the middle or right button of the mouse, since they correspond to key events, LastWave will not receive the corresponding ButtonUp events and will wait for them ad infinitum.... The only way to go out of LastWave will be to force it to quit (using the whatever `escape sequence` that works on your Mac). Thus, in this case it is very important that you change the startup. In the case `system forceKeyUp` is executed, LastWave will send right after each key down (resp. middle/right button down in case `system mouse3` has been executed) a corresponding key up event (resp. a button up event).

1.2 Installing Lastwave on a Unix (including MacOS X/Darwin) using the X11 window manager

LastWave uses some very basic C-Unix functions and the standard X11 library only (no call to X11/Toolkit, no need for Motif or any other fancy stuff !). If you use MacOS X you should install the XFree86 library (a native MacOS X version of LastWave is on its way... but its not ready yet). When you download LastWave, you get a file named `LastWave.2.0.unix.tar.gz`. This is a gzipped file that you must expand using the regular `gzip` command (e.g., `gzip -d LastWave.2.0.unix.tar.gz`). Then you can the archive unix file `LastWave.2.0.unix.tar` that you must expand using the `tar` command (generally

you would just need to do `tar xvf LastWave.2.0.unix.tar`). A new directory called `LastWave_2_0` will be created with the following directories in it:

- `kernel/` : contains all the files related to the core of LastWave
- `unix/` : contains all the machine dependent unix files
- `win32/` : contains all the machine dependent windows files
- `ansi/` : contains some ansi related files
- `user/` : contains files that you must change if you add some new C-packages
- `package_<name>/` : contain files related to the package named <name>.

Each of these directories contains 3 subdirectories :

- `src/` : contains all the sources of the corresponding package
- `include/` : contains all the .h files of the corresponding package
- `obj/` : contains a `FileList` file that defines all the files corresponding to this package and some specific options for compilation of this package. This directory will contain i) the specific Makefile of the package and ii) after compilation, several subdirectories (one for each type of computer, e.g., `sun`, `hp`,...) containing all the corresponding .o files.

On top of the package directories, the `LastWave_2_0` directory also contains the following directories

- `bin/` : contains (after performing the compilation) several subdirectories (for each type of computer) each of them containing the corresponding executable file `lw` that you should call to start LastWave
- `scripts/` : contains all the standard scripts of the command language (and the ones of the different packages too)
- `Makefiles/` : contains the generic makefile, some computer dependent definitions (`MakeDefs.xxx`) and some computer dependent rules (`MakeRules.xxx`).

For compiling lastWave, you must

1- Define in your startup shell file (`.cshrc, .login, .tcshrc,`...) the `ARCH` variable as `xxx` where `xxx` must be (depending on the computer you are running) one of `darwin` (MacOS X), `dec`, `hp`, `linux`, `sg` (Silicon Graphics) or `sun`. This should be done using a command like `setenv ARCH dec`.

2- In the same way, you should define the variable `LWPATH` that should indicate the path of the LastWave directory. It should look like `setenv LWPATH $HOME/LastWave_2_0`.

3- Go in the `LastWave_2_0/Makefiles` directory

4- Type `make dirs` which generates some directories

5- Type `make makes` which generates all the makefiles

6- And finally Type `make` for compiling LastWave. It should call the Makefile of each package and compile each of them and then generate an executable file called `lw` in the `bin/xxx/` directory.

Remark : In order to"reset" the make (i.e., delete all the `.o` files), you can execute `make clean`.

Remark : If you need to compile LastWave on a computer different from the above list. You just need to create a new `MakeDefs.xxx` file and a new a new `MakeRuless.xxx` file in the directory `Makefiles/`.

Before running LastWave, you should define the shell variable `LWSOURCEDIR` in your startup shell file. This variable corresponds to the main script directory in which the `startup` script file will be looked for when starting up LastWave. The file `scripts/startup` is a very good example of such a file. I highly recommend that you use that one before writing your own (it adds a lot of basic functionalities that you will need). Understanding how it works is an excellent exercise for practicing programming new scripts. Thus the line you should add to your startup shell file should look like `setenv LWSOURCEDIR $HOME/LastWave_2_0/scripts`

Then you are ready to run LastWave by simply executing the file named `lw` in the `bin/xxx` directory!

Before you enjoy playing around with LastWave, I would like to make a few comments:

1) Don't forget to specify the `DISPLAY` X11 environment variable if you need to. If LastWave cannot open the display, it will warn you and switch to a non-display mode in which you can do everything but displaying graphics.

2) In the display mode, LastWave runs 2 processes, one which will be waiting for characters in the terminal window and one which will be waiting for X11 events. So don't worry if you see two `lw` when you do a `ps`! However if it happens that the program core dumps (which should never be the case of course!), you should check that both processes are killed (during my tests, it always killed both but... who knows?).

3) An history file will be created by Lastwave that will be in the Home directory and whose name is `.LastWaveHistory`.

4) You can run LastWave as a batch in order to perform heavy computations wich do not need to be interactive. To do so, you can use the option `-b` of `lw` along with input and output indirections. Do not forget the `-b` option otherwise it will not work. Thus you would type something like

```
lw -b <fileIn >fileOut &
```

where `fileIn` is a script you want to run and `fileOut` whatever is printed to the LastWave terminal.

5) You can pass argument to LastWave that you will be able to use in the startup file. Whatever follows the `lw` command to run LastWave (except the `-b` option) will be assigned to the `argv` script variable of LastWave. Thus, for instance, you can easily make LastWave load only the packages you need by passing to `lw` some special arguments.

6) **Known bug**: DO NOT use the window menu to close a LastWave window. It will core dump (this bug will be fixed in a future version). You must send a `delete` message to the window (cf chapter 12) such as `msge <windowName> delete` or hitting the `f1` key while the mous pointer being in the window.

1.3 Installing Lastwave on a Windows computer, using CygWin

So far there does not exist a complete port of LastWave to Windows systems. However, it is possible to run LastWave on most Windows machines using Cygwin and an X11 server such as Cygwin/XFree86.

1.3.1 Installing Cygwin

You can download Cygwin, its User's guide and Documentation from the URL

`http://www.cygwin.com/`

Here is the description taken from this URL

> **What Is Cygwin?**
> *Cygwin is a UNIX environment for Windows. It consists of two parts: A DLL (cygwin1.dll) which acts as a UNIX emulation layer providing substantial UNIX API functionality.*
> *A collection of tools, ported from UNIX, which provide UNIX/Linux look and feel.*
> *The Cygwin DLL works with all versions of Windows since Windows 95, with the exception of Windows CE.*

Let us quote a more complete description taken from the FAQ page

`http://www.cygwin.com/faq/faq_1.html`

> **What is it?**
> *The Cygwin tools are ports of the popular GNU development tools for Microsoft Windows. They run thanks to the Cygwin library which provides the UNIX system calls and environment these programs expect.*
> *With these tools installed, it is possible to write Win32 console or GUI applications that make use of the standard Microsoft Win32 API and/or the Cygwin API. As a result, it is possible to easily port many significant Unix programs without the need for extensive changes to the source code. This includes configuring and building most of the available GNU software (including the packages*

included with the Cygwin development tools themselves). Even if the development tools are of little to no use to you, you may have interest in the many standard Unix utilities provided with the package. They can be used both from the bash shell (provided) or from the standard Windows command shell.

What versions of Windows are supported?
Wait a minute... Cygwin is only supported if you are paying for it, such as through a support contract with Red Hat. For information about getting a Red Hat support contract, see
http://www.redhat.com/products/support/cygwin/.
That said, Cygwin can be expected to run on all modern versions of Windows, except Windows CE. This includes Windows 95/98/ME/NT/2000, and, once it is officially released by Microsoft, Windows XP.
Keep in mind that Cygwin can only do as much as the underlying OS supports. Because of this, Cygwin will behave differently, and exhibit different limitations, on the various versions of Windows.

Where can I get it?
The main WWW page for the Cygwin project is http://cygwin.com/. There you should find everything you need for Cygwin, including links for download and setup, a current list of ftp mirror sites, a User's Guide, an API Reference, mailing lists and archives, and additional ported software.
You can find documentation for the individual GNU tools at
http://www.fsf.org/manual/. (You should read GNU manuals from a local mirror. Check http://www.fsf.org/server/list-mirrors.html for a list of them.)

Is it free software?
Yes. Parts are GNU software (gcc, gas, ld, etc...), parts are covered by the standard X11 license, some of it is public domain, some of it was written by Cygnus and placed under the GPL. None of it is shareware. You don't have to pay anyone to use it but you should be sure to read the copyright section of the FAQ more more information on how the GNU General Public License may affect your use of these tools.
In particular, if you intend to port a proprietary (non-GPL'd) application using Cygwin, you will need the proprietary-use license for the Cygwin library. This is available for purchase; please contact sales@cygnus.com for more information. All other questions should be sent to the project mailing list cygwin@cygwin.com. Note that when we say "free" we mean freedom, not price. The goal of such freedom is that the people who use a given piece of software should be able to change it to fit their needs, learn from it, share it with their friends, etc. The Cygwin license allows you those freedoms, so it is free software.
The Cygwin 1.0 product was a "commercial" distribution of cygwin. As such, it included such non-software things as printed manuals, support, and aggregation of useful utilities. There was nothing (software-wise) in there that you couldn't

get off the net already, if you took the time to find and download everything (and usually, build it yourself), although the versions available for download may have been different than those distributed with the commercial product. We tested it all to make sure it worked together, and packaged it in a convenient form. We considered such testing and packaging to be a valuable service and thus charged a fee for it. Plus, it provided income for the cygwin project so we could continue working on it. However, Red Hat is no longer offering Cygwin 1.0 on CD. There are tentative plans to offer a new Cygwin CD sometime in early 2001. So far, however, there are no definite plans. The latest news about this can be found at http://cygwin.com/cygwin-cd.html.

1.3.2 Installing Cygwin/XFree86

Cygwin/XFree86 and its Documentation can be downloaded from the URL

`http://xfree86.cygwin.com/`

The following short description is quoted from this URL :

Cygwin/XFree86 is a port of XFree86 to the Microsoft Windows family of operating systems. Cygwin/XFree86 runs on all recent consumer versions of Windows and all recent business versions of Windows; as of 2001-04-11, those versions are specifically Windows 95, Windows 98, Windows Me, Windows NT 4.0 and Windows 2000.
Cygwin/XFree86 consists of an X Server, Xlib, and nearly all of the standard X clients, such as xterm, xhost, xdpyinfo, xclock, and xeyes. Cygwin/XFree86, as the name implies, uses the Cygwin project which provides a UNIX-like API to Xlib and X clients, thereby minimizing the amount of porting required.
Cygwin/XFree86 is licensed under an X style license; Cygwin is licensed under a modified GNU General Public License that specifically allows libcygwin1.a to be linked to programs that are licensed under an Open Source compliant license without such linking requiring that those Open Source programs be licensed under the GNU General Public License (see the Cygwin licensing page for more information). Source code and binaries for both projects are freely available.

1.3.3 Elementary configuration of Cygwin/XFree86

If you don't want to spend too much time in the documentation of Cygwin and Cygwin/XFree86, especially if you installed it just to have LastWave run, here are a few tips to configure it quickly.

Essentially you just need to copy the files `cyglocal.bat`, `.bash_profile`, and `cygnet.bat` (if you need it) from the distribution of LastWave (they should be in the directory `INSTALL_CYGWIN`) to the right place!

The `cyglocal.bat` file

```
@ECHO OFF
```

```
c:
chdir \cygwin\bin
SET MAKE_MODE=UNIX
SET DISPLAY=localhost:0

bash --login -i
```

Where to put it

You should simply copy the `cyglocal.bat` file on your Windows desktop. You will start Cygwin by double clicking it.

The `.bash_profile` file

```
# Environment variables for LastWave
export LWPATH=$HOME/softs/LastWave_2_0/
export LWSOURCEDIR=$LWPATH/scripts/
export ARCH=cygwin
# Putting X11R6 and LastWave in the path
export PATH=$LWPATH/bin/cygwin:/usr/X11R6/bin:$PATH

X&
twm&

xterm -sb -tn vt100 -e bash
exit
```

Where (and why) to put these files

When you run Cygwin by double-clicking the `cyglocal.bat` link on your Windows desktop, you get a `bash` window where you can type in commands.

You should put the `.bash_profile` file in your home directory (under Cygwin). This can be done using Windows Explorer, the destination folder should be something like c:\cygwin\home\<your_login>

The `.bash_profile` file is executed by the `bash` shell when you log in. It first defines some environment variables that will be useful when you compile or run LastWave. Then it starts the X server and a simple window manager (twm). Eventually, it starts an xterm.

Remark You need to click somewhere with the left button of your mouse to let the xterm appear on the screen.

Remark You can edit the `cyglocal.bat` file (with Windows Notepad, for example) to modify it at your will

WARNING The syntax in `cygwin.bat` is that of DOS, not of a Unix shell. For example, you would use %PATH instead of $PATH to evaluate the (Windows) environment variable

PATH.

1.3.4 Choosing which X server to use

I run LastWave on a Windows notebook with a commercial X server installed, X-Win32^{TM}, which only works when connected to my office network and only accepts requests from machines with a proper address. When I am on the go, I use Cygwin/XFree86 as a X server by double clicking the `cyglocal.bat`, but when I am connected to the network and X-Win32^{TM}is already running, I prefer to use it.

To deal with it I have replaced the lines

```
X&
twm&
```

in the `.bash_profile` with

```
case "$DISPLAY" in
foo.fsf.org:0)
    ;;
localhost:0)
    X&
    twm&
    ;;
esac
```

and I have written a new file `cygnet.bat`

```
@ECHO OFF

c:
chdir \cygwin\bin
SET MAKE_MODE=UNIX
SET DISPLAY=foo.fsf.org:0

bash --login -i
```

which I can double click to start Cygwin when I am connected to the network.

1.3.5 Installing and Starting LastWave

Once Cygwin and an X11 server are installed on your Windows machine, you can install LastWave by following the instructions for its installation on a Unix machine (see Section 1.2).

Notes on the `MakeDefs.cygwin` file

- `LIBS` variable :

  - `-lwinmm` : tells the linker to use the win32 multimedia DLL, which is normally provided with Cygwin. In the current implementation of LastWave , this DLL is used only for playing and recording audio.
  - `-L /usr/X11R6/lib` : location of X11 libraries, may depend on your distribution of Cygwin/XFree86.

- `X11INCLUDE` : `/usr/X11R6/include`, may depend on your version of Cygwin/XFree86.

1.3.6 Known bugs

There seems to be a bug with the X11 server that sometimes prevent correct refresh of X windows and correct color display. Any help on solving this is welcome.

Chapter 2

For LastWave 1.xx users

This chapter is for former users of LastWave 1.xx.
It gives a *short* description of the main differences with LastWave 2.0.

2.1 Introduction

There are many differences between Lastwave 1.xx and LastWave 2.0. The 2.0 version is a major upgrade. The command language has changed quite a bit. It is much faster and much more powerful than before.. In this chapter we try to give you some hints about the different changes in the command language so that you should be able to update your scripts and write some more.

However, this chapter will only the basics anout the new features of LastWave. There is no way you can learn about all the new aspects of LastWave 2.0 by just reading this chapter. **At some point, you should definitely read the manual pages.**

2.2 Evaluation - Types

In LastWave 2.0, by defaults, all the arguments of a command are *evaluated*. In 1.xxx versions it was the case only of numbers, signals and images. Indeed, when you wanted to pass a number as an argument you were able to write `2` or `2*i` (if `i` was a number variable) or any numerical expression. The expression was evaluated before being sent to the command.
In LastWave 2.0, by default, this is the of all arguments. Thus for instance, if a command `cmd` expects a string `string` as an argument in LastWave 1.xxx you would write `cmd string` whereas, now you have to write `cmd 'string'`. Indeed, if you wrote `cmd string`, `string` would be interpreted as a variable which content should be a string whereas the quotes `'` in `'string'` indicates that this argument should be evaluated as the string `string`. Thus for instance to read a signal `s` from a file `myFile`, in `1.xxx`, you wrote

```
read s myFile
```

now you have to write

```
read s 'myFile'
```

To send the argument `i=3` to the command `cmd`, in 1.xxx, you wrote

```
> cmd $i
```

in 2.0, you write

```
> cmd i
```

Most of the time it is no longer necessary to use the $ syntax.

There is a specific syntax for dynamically building any type of arguments, for instance

- `&string` : `"a string"` or `'a string'` (see Section 4.5)
- `&signal` : a y-signal made of 4 values `<1,2,3,10>` or a xy-signal `XY(<0,1,2,3>,<1,2,3,10>)` (see Sections 4.8 and 9)
- `&image` : a 2x2 image `<1,2;3,4>` (see Section 10)

Let us note that actions of commands are *non*-evaluated arguments, e.g.,

```
stats print s
```

and not

```
stats 'print' s
```

In LastWave 2.0, you can type in any expression in the terminal, and it will be evaluated, e.g.,

```
a> 1+3*2
= 7
```

The command = does not exist anymore

2.3 New types

There are a lot of new types in LastWave 2.0 that make it much more powerful. Here are two very important ones

- `&listv` : list of values (of any type), {`1 2 'hello' <1,2>`} (see Section 4.6)
- `&range` : to store (much more efficiently than in signals) uniformely distributed numbers `1:10`, `1:.5:10` (see Sections 4.9 and 9)

2.4 Linear algebra

Images can be seen as matrices and signals as (*horizontal*) vectors. LastWave 2.0 lets you perform basic linear algebra using images and signals. You should read Section 10.

2.5 Basic assignements

The command `set` should not be used any more. It still exists for compatibility but will be removed in the next version. The `set` command in LastWave 1.xxx let you set a variable with a non-evaluated argument. In LastWave 2.0 the `setv` (the `v` stands for `value`) lets you set a variable with an *evaluated* argument. For the sake of simplicity you can use the traditional `=` sign (it will be replaced by LastWave by the equivalent `setv` command). Thus for instance, in 1.xxx you wrote

```
set s hello
```

in 2.0, you should write

```
setv s 'hello'
```

or equivalently

```
s='hello'
```

2.6 Assignements of signals or images

You have to be aware that if `s` is a signal, then the assignement

```
s1 = s
```

erases the variable `s1` (if existed before) and assign the signal `s` to it. So

- at the end, both `s1` and `s` will point to the *same* signal
- the variable content of `s1` is first erased

If `s1` is a signal, you might want to copy the values of the signal `s` into the signal `s1`. **This is not achieved by the statement** `s1=s`. In order to perform the copy, you need to type

```
s1[]=s
```

(Let us note that both syntaxes work exactly the same way for images.)
Thus, if you want to write a procedure that fills up a signal passed as an argument with a random noise you must write

```
setproc test {{&signal sig}} {sig[] = Grand(100)}
test s
```

Of course, the following will not work

```
setproc test {{&signal sig}} {sig = Grand(100)}
test s
```

using a different syntax, the following will work

```
setproc test {} {return Grand(100)}
s = [test]
```

Let us note that when you type `0a = s`, since the signal `0a` cannot be deleted (it *belongs* to the current object `a`), it will perform the exact same action as if you typed `0a[]=s`.

2.7 The `return` command

In 1.xxx, the `return` command did not evaluate its argument. In 2.0, it does evaluate. Thus in 1.xxx, you wrote

```
return hello
```

in 2.0, you write

```
return 'hello'
```

Moreover you can return any type of arguments (a signal, a listv...)! You can write

```
return {1 2 <1,2> 'hello'}
```

2.8 Fields - Extraction

In 1.xxx versions, all the structures (e.g., *signals*) had a corresponding command (e.g., `setsignal`) for setting the fields of the structure. This is no longer the case, you can directly access the fields (both for getting/setting values) using the simple syntax `struct.field`, e.g., in 1.xxx, you would write

```
> setsignal dx s .5
> setsignal dx s
= .5
```

in 2.0, it becomes

```
> s.dx=5
> s.dx
= .5
```

Moreover you can use the syntax `value[...]` for getting/setting some fields of some values. This is an extremely powerful syntax that lets you for instance address the values of a signal using real abscissa. You should read Section 4.10 for the basics, and Sections 8.2, 9.8, 9.9 and 9.11.2 for advanced examples.
You can use this syntax on a lot of different types (including `&string`, `&listv`, `&signal`, `&range` and `&image`). A very simple example would be

```
> s.Y[2] = 5.1
```

instead of

```
> setsignal Y s 3 5.1
```

WARNING: In 2.0, the indices start at 0 and not at 1

2.9 Graphics

The graphics has not changed much except that, again, all the arguments are, by defaults, evaluated arguments. Thus color names should appear in between quotes `'...'`, font names too,... For instance, in 1.xxx you would write

```
disp s -..1 -fg red -curve .  -title hello -..fv1.title -font 12
```

in 2.0, it becomes

```
disp s -..1 -fg 'red' -curve '.'  -title 'hello' -..fv1.title -font '12'
```

2.10 The help system

The help system has been greatly improved. As before, you can get help/completion using the `esc esc` key sequence on the command line. It works both for help/completion on commands, for help/completion on fields and for help on `*options` for extraction.
You should try the new `apropos` command that allows you to find all the commands related to a subject.
To learn about the help system, you should run `Help` when starting up LastWave as suggested at startup.
You can get different helps on any graphic object by just pointing with the mouse on it and hitting the `h` key several times... This system replaces the old *crazy* system where you had to hit themouse key along with many keys...

2.11 A list of changes

To end this chapter, we just give a list of some important changes/remarks.

- Redirection syntax of i/o has changed (read Section 8.3.3). You must now put the characters `::` in between the command itself and the redirection signs. Thus, for instance, in `1.xxx` you wrote

  ```
  echo hello baby >>file
  ```

  while in `2.0`, you should write

  ```
  echo hello baby ::  >>file
  ```

  (don't forget the space characters before and after the `::`).

- The help of a script procedure is specified using a different syntax. It is an evaluated string (instead of a non evaluated string), i.e., in `1.xxx` you wrote

  ```
  setproc test {i} {{{usage} {help}}} {...}
  ```

  in `2.0`, you must write

  ```
  setproc test {i} '{{{usage} {help}}}' {...}
  ```

- The $ syntax is still valid, however, since arguments are evaluated by default, most of the time , you won't use it anymore. **you should try to remove all of the $, it slows down LastWave**

- Since all arguments are evaluated, there is no need for the commands `s=` and `i=`. they don't exist anymore. Just use the `=` syntax instead.

- The test expressions in all the test commands (e.g., `if, for` or `while`) is a regular evaluated argument that should be a number (see Section 6.2). Thus, it should NOT be surrounded by brackets (which indicates LastWave that it should be evaluated as a `&listv`!). Thus, in 1.xxx you wrote

  ```
  if {$x==1} {return}
  ```

  or

  ```
  if {$y==hello} {return}
  ```

  in 2.0, you write

  ```
  if (x==1) {return}
  ```

  or

  ```
  if (y=='hello') {return}
  ```

- In the last examples, you cannot write

  ```
  if (y=='hello') return
  ```

  you HAVE to write

  ```
  if (y=='hello') {return}
  ```

  Indeed, the second argument of the `if` command is script (i.e., a value of type `&script`). If you do not put the {...} LastWave interprets `return` as a variable which content should be a `&script`. Whenever, LastWave expects scripts, a valid syntax is the following {`the script`} (it is the only case a {...} is not interpreted as a `&listv`). You should read Section 7.3 and more generally Section 7.2.

- for the `printf,errorf` and `sprintf` commands, all the arguments are evaluated too, thus you should write

  ```
  printf "%s" 'coucou'
  ```

  instead of

  ```
  printf "%s" coucou
  ```

- If you want to *group* words without quoting them (which will make them a `&string`), you should use backquotes

  ```
  echo ‘This is the first argument‘
  ```

- The 1.xxx `import` command corresponds now to the `import args` command. Moreover, the list of imported variables are not evaluated. Thus, to import the variable `x` from the calling environment and make it `y` in the new one, in 1.xxx you wrote

  ```
  import "x y"
  ```

  in 2.0, you write

  ```
  import ‘x y‘
  ```

- The `foreach` command lets you loop on `&listv`, `&signal` or `&ranges` (see Section 6.4.1)

- The syntax for specifying the help of a command has changed (see Section 7.1.5)

Part II

The LastWave command Language

Chapter 3

Getting started

3.1 The `startup` file

When you run LastWave, the first thing that LastWave does is to run a script file called the *startup file*. The startup file is, by definition, the file named `startup` in the `script` directory (for unix computers, let us recall that the script directory is indicated by the unix variable LWSOURCEDIR, see Sectionsinstall). The default startup file does the following

- define the prompt procedure that will manage the prompt terminal
- (macintosh only) set up whether you use a 1-button or a 3-button mouse and set up the size and position of the terminal window
- load some packages (you should comment out all the lines corresponding to the packages you do not use. In order to comment out a line just put the # character at the begining of the line)
- initialize some colormaps: one named `color` which is a *rainbow* color map of 30 colors and a greyscale color map (named `grey`) of 256 greys.

3.2 The prompt

After printing the Copyright and some information on how the execution of the startup file went, LastWave displays a prompt which should look like

```
2(&wtrans) a>
```

The `2` is the last command number in the history of all the commands you typed before. We will come back later on (see Section 8.5) the `a` and `(&wtrans)` that appear in the prompt. For now just ignore it. So this number will increase by 1 each time you type a new command that is executed. In order to see the history you can use the command `h` and in order to recall a command you just need to type in its corresponding number.
There are numerous ways of editing the command line and navigating in the history. To learn about them just type `HelpTerm`. You should definitely do it before going on. You can also read Section 6.5

3.3 Some Demos

Okay, you succeeded in running LastWave. As suggested, you should run demos first. LastWave comes with demos about each numerical packages included (there are no demo for LastWave kernel, you should read the next section to learn about the command language).

Just type `Demo` followed by the carriage return key. It will give you a list of the demo files you can run. Just read the instructions. Again, as suggested, you should first run the `DemoSignal` by just typing `source DemoSignal`. It will define new commands. Each of them corresponding to a different demo. It will print out the name of each commands along with a short description. For instance, the `DemoSignal` will define a single command named `DemoSignalDisp` that shows you basics about displaying signals. In order to run it you should just type `DemoSignalDisp` followed by the carriage return key. Again just follow the instructions, then move on to a different demo...

3.4 Hello World

Just type `printf 'Hello World\n'` into the terminal:

```
a> printf 'Hello World\n'
Hello World
```

You just executed the command `printf` with a single argument which is the string `'Hello World\n'`. This command just prints the string (the `\n` corresponds to the *newline* character). You could assign the same string to a variable using the = sign

```
a> s = 'Hello World\n'
= 'Hello World
'
```

(The fact that the second quote `'` is on the next line is because the new line character `\n` is printed at the end of the string). Let us note that the = sign actually corresponds to the `setv` command (`setv` stands for *set value*). The assignation we just made is equivalent to

```
a> setv s 'Hello World\n'
= 'Hello World
'
```

Thus the `setv` command takes two arguments, the first one which is the name of the variable and the second one which is the value to assign to the variable. Let us note that it returns the value which was assigned (the `printf` command does not return any value).

Variable names: The name of a variable should start with a capital or small letter or the _ character and should be followed by any alphanumeric character including _

WARNING : If you try to use variable named `a` or `b` you will get an error because these variables are assigned (at startup) to wavelet transform structures. For the sake of clarity, LastWave does not let you overwrite a variable which corresponds to a high level structure (such as a wavelet transform) with anything else. If you want to force the assignement you first have to delete the variable using the command `var delete`.

We can then type

```
a> printf s
Hello World
```

In order to know the value of a variable or, more generally, of an expression you can just type the expression in the terminal window

```
a> s
= 'Hello World
'
a> s = 'Hi there'
= 'Hi there'
a> s1=s+', how are you?'
= 'Hi there, how are you?'
a> s1
= 'Hi there, how are you?'
```

The `+` operator performs concatenation of strings. In Sections 4.5, 8.2.2 and 8.2.3, you will see that LastWave lets you manipulate strings in a very advanced way.
The `+` operator can of course also be used for adding numbers

```
a> n=37
= 37
a> n+12
= 49
```

Each value has a type in LastWave. the type names always start with a `&` character. Thus `s` is of type `&string` whereas `n` is of type `&num`. In order to know the type of a value you can use the "operator" `type`, e.g.

```
a> type(n)
= '&num'
a> type(s)
= '&string'
a> type(type(n))
= '&string'
```

The `printf` command lets you print values of any type. It actually uses the same syntax as the C-language `printf` instruction. The first argument is a string which corresponds to the *format* that will be used to print the values and the other arguments are the values. For instance

```
a> printf 'one+one = %d\n' n 2
one+one = 2
```

The `%d` token in the format string specifies where the first value, which must be a number (the letter `d` stands for *decimal*) should be printed. The token `%s` corresponds to string values (`s` stands for *string*). Thus,

```
a> printf 'The type of %d is %s\n' type(n)
the type of 37 is &num
```

Or

```
a> printf 'sin(%d)=%d\' 1.5 sin(1.5)
sin(1.5)= 0.997495
```

The operator `sin` computes the sinus of a number. Let us note that *operators* are different from *commands* in LastWave. They are both functions with arguments. However operators generally correspond to much more basic actions than commands and their syntax is completely different. Apart from the operators `+,* ...` the syntax of an operator is `operator(arg1,...,argN)` whereas the syntax of a command is `command arg1 ...argN`. Both `setv` and `printf` are commands. There are other operators than `type` e.g., `sin`). Operators always return values. This is not automatically the case of commands.
It is time to learn about the main types in LastWave and the corresponding operators.

Chapter 4

Values and types

Starting from version LastWave 2.0, by defaults, all the arguments of a command are first evaluated before the command is executed. The result of an evaluation is a *value* of a given *type*.

4.1 LastWave types: an overview

In the next sections, you will learn the basics about several important types in LastWave (type names always start with a `&` character in LastWave):

- `&num`: for managing numbers (both floats and integers), such as 1 or 3.1415
- `&string`: for managing strings, such as `'HellO hOw are you!'`
- `&null`: a type associated to a single value `null` that is used by various functions
- `&array`: for managing arrays of values of arbitrary types, using string indices,
- `&listv`: for managing list of values of arbitrary types,
- `&signal`: for managing 1d (uniformly or non-uniformly sampled) functions. It is also used as an efficient way (more efficient than `&listv`) of storing 1d arrays of numbers (see Section 9 for full description),
- `&range`: for managing 1d uniformly distributed list of numbers such as the list of the integers 1,2,3,4,5. This is done more efficiently than both `&listv` and `&signal`.

You should be aware that LastWave knows about some more types. They will be described later in this manual. They are

- `&image`: for managing 2d images for image processing or matrices for linear algebra (see Section 10),
- `&script`: for managing scripts, i.e., piece of code using LastWave command language (see Section 7.3),
- `&proc`: for managing commands, i.e., LastWave commands defined either in C or using LastWave command language (see Section 7.2).

You should be aware that some packages define new types. For instance the package `wtrans1d` which allows computation of 1d wavelet transform defines the type `&wtrans` to store the result of a wavelet transform.

All these types corresponds to structures which have *fields* (as classes have fields in an object-oriented language). Thus, for instance, the field `length` is a read-only field for the string type, it corresponds to the number of characters in the string. To get the length of a string `s`, one just need to type `s.length`. In order to address the `n`th character of a string `s`, one must use the syntax `s[n]`. This is no longer a field. It is referred to as an *extraction*. Of course, though many types allow extractions, not all of them do (`&num` doesn't for instance).
Let us get some more insights about the most important types.

Remark (**getting help on types**): You can get help (a short description along with the list of fields and the extraction sytax) on any type by just typing `help &type` where `type` is the type you want to get help on (for instance you can type `help &string`).

4.2 The operators `type`, `is` and `isnot`

These three operators can be applied to any value of any type. We have already seen (Section 3.4) the operator `type` which returns a string which indicates the type of the value. The syntax is `type(`<`value`>, e.g.,

```
a> type(27+3)
= '&num'
a> type(type(27+3))
= '&string'
```

The operator `is` lets you test wether 2 values are *physically* the same values (i.e., they point to the same object). It returns 1 if it does and 0 otherwise. The syntax is <`value1`> `is` <`value2`>. Thus for instance

```
a> x=1.5
= 1.5
a> y='hello'
= 'hello'
a> (x is x)
= 1
a> (x is 1.5)
= 0
a> (x is y)
= 0
a> (1 is 1)
= 0
```

The operator `isnot` is the negative of the operator `is`.

Remark: Let us note that if you type `x is x` instead of `(x is x)` in the last example, LastWave will generate an error because it will look for the command `x`, in order to send

to it the two arguments `is` and `x`. Since the command `x` does not exist, it will lead to an error.

4.3 The `null` value

The `null` value is a special value : it is the only value associated to the `&null` type.

```
a> null
= null
a> type(null)
= '&null
a> (1 is 1)
= 0
a> (null is null)
= 1
```

4.4 Working with numbers (`&num` type)

If you just type in the terminal window a mathematical expression followed by the `return` key, LastWave evaluator will evaluate it for you and will return the result as a `&num` value:

```
a> 1+2*3
= 7
a> sqrt(2)/5
= 0.28284273
```

The operators for numbers are

- standard operations : `+, -, *, /` (along with `+=, -=, *=, /=`, see below)
- the absolute power : `^` (`x^f` $= |x|^f$) (along with `^=`, see below)
- the integer power : `*^` (`x*^n` $= x^n$, where n is a positive integer),
- integer division and remainder : `//`, `%`,
- regular tests (returns 0 if false and 1 if true) : `==` (is equal?), `!=` (is not equal?), <=, <, >=, >,
- boolean operators : `&&` (logical AND), `||` (logical OR), `!` (logical NOT),
- standard math function : `ln` (natural logarithm), `log` (base 10 logarithm), `log2` (base 2 logarithm), `exp` (exponential), `sqrt` (square root), `abs` (absolute value), `min` (the minimum of two numbers), `max` (the maximum of two numbers) `int` (integer part), `frac` (fractional part), `round` (closest integer), `ceil` (closest greater integer), `floor` (closest smaller integer),
- the constant (i.e., no argument) `pi` (3.14159...),

- and the random values (no argument) `urand` (a random number following a uniform law between 0 and 1), `grand` (a random number following a gaussian law).

Let us note that the random generator can be initialized using the `randinit` command.

WARNING : A single integer (as `1`) typed in the terminal will try to recall the history line number 1, i.e., it does not call the evaluator and will not return 1.

In order to perform an assignement you just need to use the `=` sign (or equivalently the `setv` command). Thus for instance

```
a> x=2*3
= 6
a> x = 2*x
= 12
```

The last line could also be written (following C-language syntax)

```
a> x=2*3
= 6
a> x *= 2
= 12
a> x
= 12
```

You can use the `*=` syntax with some other operators. LastWave evaluator understands `+=`, `-=`, `*=`, `/=`, `%=`, and `^=`.

4.5 Working with strings (`&string` type)

A string is any text in between double quotes `"` or single quotes `'`.

```
a> "hello"
= 'hello'
a> 'I can use single quote too'
= 'I can use single quote too'
a> 'I can include a " in between single quotes'
= 'I can include a " in between single quotes'
a> type('hello')
= '&string'
a> type(sqrt(2)/5)
= '&num'
```

The operators for strings are

- `==` and `!=`: comparison character by character
- `<=`, `>=`, `<` and `>`: comparison using alphabetical order
- `+` and `+=`: for concatenation of two strings

- * and *=: for repetition (the syntax is <string>*<n>)

```
a> x = "this is"
= 'this is'
a> x+' '+'a concatenation'
= 'this is a concatenation'
a> x*3
= 'this isthis isthis is'
a> (x*2 == x+x)
= 1
```

Fields of a &string : As explained before, some types have fields. The type &num does not have any field associated to it. The type &string has several fields. The most useful are

- length : a read-only field corresponding to the number of chatacters of a string,
- tonum : a read-only field corresponding to the number associated to the string. If it does not correspond to a number, tonum returns the value null.

Thus, for instance :

```
a> x = "this is"
= 'this is'
a> x.length
= 7
a> x.length=8
** Error :  Read only field 'length'
-- > x.length=8
a> x.tonum
= null
a> x='34.5'
= '34.5'
a> x.tonum
= 34.5
```

Remark (Getting some help on fields) : While typing in the terminal, as soon as you typed the . character after a value with the intention of typing in a field name, you can get help (and completion) hitting twice the escape key. You can get some more explicit help by hitting the escape key and then the h key.

Extraction of a &string : To access the nth character of a string s, you can use the extraction syntax : s[n]. The integer n must range in [0,s.length-1]. Thus

```
a> x = "this is"
= 'this is'
a> x[3]
```

```
= 's'
a> x[3] = 'S'
= 'thiS is'
a> x[10]
** Error :  x[10]
**                .^.
** --> Indexes are out of range
--> x[10]
```

One of the power of LastWave evaluator is the very flexible way you can use extractions. You will learn more about extractions in Section 4.10.

4.5.1 String substitutions - The $ syntax

Before what you typed in is evaluated by LastWave, you have the possibility to ask for some string substitutions. String substitutions are very useful. As we will see later, they are extremely useful in (mainly) two cases :

- for indirect access (mainly in arrays, see Section 4.7.1) and
- for dynamically building scripts (see Sections 7.3 and 7.2).

Whenever the syntax `$varName` is used, where `varName` corresponds to a string or a number variable (only!), then, before evaluation is performed, the whole string `$varName` is replaced by its value. It is important to understand that this substitution takes place `before` evaluation is performed.

WARNING : $-substitutions *do not* take place in between {...} (this is to avoid substitution in a script, see Sections 7.3 and 7.2)

Thus for instance

```
a> x = 1
= 1
a> s = 'one is $x'
= 'one is 1'
```

4.6 Working with list of values (`&listv` type)

A `&listv` is a list of values. They do not have to be of the same types. The syntax for specifying a list of values is {<value1> <value2> ...<valueN>}. Each value must be separated by (at least) one space.

```
a> {'a listv' 'made' 'of' 4 'elements'}
= {'a listv' 'made' 'of' 4 'elements'}
a> type({1 're'})
= '&listv'
a> {}
= ''
```

The last example corresponds to an empty listv. Of course, you can embed a listv into another listv :

```
a> {'like' {'this' 'listv'}}
= {'like' {length=2}}
```

As you see in this last example when the evaluator prints a list, the values are printed using an abreviation. The abreviation for a `&listv` is of the form {`size=<s>`} where `<s>` is the number of values.

The operators for listv's are

- `==` and `!=`: comparisons of 2 listv's (by comparing each value)
- `+` and `+=`: for adding an element at the end of a listv (the syntax is `<listv>+<elem>`)
- `*` and `*=`: for repetition (the syntax is `<listv>*<n>`)

```
a> x={'a' 1}
= {'a' 1}
a> x=x+'b'+2
= {'a' 1 'b' 2}
a> x+='c'+3
= {'a' 1 'b' 2 'c' 3}
a> x*=2
= {'a' 1 'b' 2 'c' 3 'a' 1 'b' 2 'c' 3}
a> x={'a' 1}
= {'a' 1}
a> x+x
= {'a' 1 'a' 1}
a> x+{x}
= {'a' 1 {'a' 1}}
```

WARNING : As you see at the end of the last example, adding (using `+`) a listv to another listv appends the two listv's. It DOES NOT make the second listv the last element of the first one. In order to do so you just need to embed the second listv within {} (making thus a listv which has a single element : the second listv).

You can test whether 2 listv's point to the exact same value (not copies) using the `is` (or `isnot`) operator

```
a> x={'a' 1}
= {'a' 1}
a> (x is x)
= 1
a> ({'a' 1} is {'a' 1})
= 0
```

Fields of a `&listv` : The type `&listv` basically has a single field :

- `length` : a read/write field corresponding to the number of values in the listv,

When you write the field `length` with a value (which must be a positive integer), if the requested length is smaller than the previous length, the listv is shortened, if not, allocation is performed and the listv is completed with 0 values. Thus, for instance :

```
a> l = {'a' 1}
= {'a' 1}
a> l.length
= 2
a> l.length=7
= 7
a> l
= {'a' 1 0 0 0 0 0}
```

Extraction of a `&listv` : To access the `n`th value of a listv `l`, you must use the *extraction* syntax : `l[n]` :

```
a> l = {'a' 1}
= {'a' 1}
a> l[0]
= 'a'
a> l[0] = 12
= 12
a> l
= {12 1}
```

WARNING : a listv is not copied when assigned. Indeed, if you write

```
a> l = {1 2 3}
= {1 2 3}
a> l1 = l
= {1 2 3}
```

Both `l1` and `l` refer to the *same* listv. The assignment command `l1=l` did not perform any copy of the listv. Thus if you change `l`, `l1` will change at the same time:

```
a> l[1]='a'
= 'a'
a> l
= {1 'a' 3}
a> l1
= {1 'a' 3}
```

If you want to make a copy of the listv you should use the generic `copy` command (see Section 8.6). Let us note that, on the contrary, a string or a number is copied when assigned.

4.6.1 Multiple assignations - The .l syntax

You can assign each value of a listv to a different variable using the syntax

{*var1 ... varN*} = <*listv*>

Thus, for instance

```
a> l = {'a' 1 {1 2}}
= {'a' 1 {1 2}}
a> {e f g}=l
= {'a' 1 {1 2}}
a> e
= 'a'
a> f
= 1
a> g
= {1 2}
```

The number of variable names must match exactly the length of the listv except if the last variable name starts with a dot `.`. This last variable is then assigned to the remainder of the listv

```
a> {e .f}=l
= {'a' 1 {1 2}}
a> e
= 'a'
a> f
= {1 {1 2}}
```

4.7 Working with arrays (&array type)

LastWave lets you manipulate arrays of values indexed by strings. They are implemented as hash tables whose indexes could be any valid variable name (extended to names starting with a number). Each index points to a variable that can correspond to a value or to an array again. That lets you simulate multidimensionnal arrays. The content corresponding to an <index> of an array named <array> is accessed using the syntax <array>.<index>. Thus, let us create an array named `tab`, with indexes 1, 2 and 3 respectively assigned to the strings `'one'` `'two'` and `'three'` :

```
a> tab.1='one'
= 'one'
a> tab.2='two'
= 'two'
a> tab.3='three'
= 'three'
```

You can access the values using the syntax :

```
a> tab.1
'one'
```

You can add another index named `total` whose value is the number 6

```
a> tab.total=6
= 6
```

As said above an index of an array could correspond to another array, e.g.,

```
a> tab.anotherarray.4='four'
= 'four'
a> tab.anotherarray.5='five'
= 'five'
a> tab.anotherarray.total=9
= 9
```

Remark: You can obtain the listv of all the indexes associated to an array using the `array` command. Moreover the `nice` command (defined in the script file `scripts/basic/misc`) allows you to make a "nice" display of the content of an array.

4.7.1 Indirect access

Using the $ syntax along with arrays allow *indirect* access in a very natural way :

```
a> index=4
= 4
a> tab.$index
'four'
a> tab.$index=4
=4
a> tab.$index
4
```

or even

```
a> index='anotherarray'
= 'anotherarray'
a> tab.${index}.5
'five'
```

Let us note that the braces $\{\dots\}$ are very important. If you do not type them, LastWave will interpret the line the following way `tab.${index.5}` and since `index.5` is not a variable it will generate an error.

4.8 A short introduction to signals (`&signal` type) :

WARNING : In this section, we introduce *signals* which is a major structure in LastWave. We will teach you really elementary things about signals : basically, only what you will need to know in order to use them in extractions. The signal structure is much richer than that. It

lets you define 1d uniformly or non uniformly sampled functions. The *extraction* mechanism (i.e., the `[...]` syntax) is very powerful. The whole section 9 is devoted to signals.

In this section we will deal only with uniformly sampled signals. It corresponds basically to a list of numbers. Let us note that it is a much more efficient structure (both in terms of memory and in terms of access) than would be a listv made of numbers. One way of building a signal is to use the syntax <<num1>,<num2>,...,<numN>>.

```
a> x = <1,-10,20.1>
= <1,-10,20.1>
```

Most of the operators you can use on numbers can be used on signals. If the operator takes a single argument LastWave evaluator simply applies it to each value of the signal :

```
a> abs(x)
= <1,10,20.1>
```

If the operator takes 2 arguments, in they are both signals, they must be the same size and the result is a signal of the same size which nth value is the result of the operator when applied to both the nth value of the first signal and the nth value of the second one :

```
a> y = x+x
= <2,-20,40.2>
a> y += y
= <4,-40,80.4>
```

If one of the argument is a number, the result is the same as if this number was replaced by a constant signal of the same size as the input signal :

```
a> y = 2*x
= <2,-20,40.2>
a> y *= 2
= <4,-40,80.4>
a> y == int(y)
= <1,1,0>
```

The operator `any(<signal>)` returns 1 if any value of `<signal>` is non zero and 0 otherwise. The operator `all(<signal>)` returns 1 if all the values are non zero and 0 otherwise.

```
a> any(y ==int(y))
= 1
a> all(y ==int(y))
= 0
a> all(y>-80 && y<81)
= 0
```

You can generate random signals using the `Urand(<size>)` (uniform law between 0 and 1) and the `Grand(<size>)` (gaussian law) operators. In some cases, if the `<size>` can be deduced by LastWave you can ommit it.

```
a> z = Urand(3)
= <0.116015,0.129491,0.0770746>
```

Concatenation of `&signal`'s : You can concatenate signals with other signals or with floats using the same syntax

```
a> u=<y,0,1,y>
= <size=8;4,-40,80.4,0,1,4,...>
```

Let us note that the full signal does not print because it is too long. Only the first values are printed. If you want a full print of the signal values you can type `print u`. The syntax `<<listv>>` where `<listv>` is a listv of numbers converts it to a signal. Thus If you want to concatenate the signal `y` `10` times you just need to do

```
a> u=<{y}*10>
= <size=30;4,-40,80.4,4,-40,80.4,...>
```

Fields of a `&signal` : The only field you need to know about for now is :

- `size` : a read/write field corresponding to the number of values in the signal.

You should be aware that signals have much more fields than that. You will learn about them in section 9.

Extraction of a `&signal` : To access the `n`th value of a signal `y`, you must use the *extraction* syntax : `y[n]` :

```
a> y[2]
= 80.4
a> y[2]=12
= 12
a> y
= <4,-40,12>
```

As we will see in the next section ranges allow to manipulate equally distributed series of numbers.

WARNING : a signal is not copied when assigned. Indeed, if you write

```
a> s = <1 2 3>
= <1 2 3>
a> s1 = s
= <1 2 3>
```

Both `s1` and `s` refer to the *same* signal. The assignment command `s1=s` did not perform any copy of the signal (as when you assign a listv). Thus if you change `s`, `s1` will change at the same time:

```
a> s[1]=10
= 10
a> s
= <1 10 3>
a> s1
= <1 10 3>
```

If you want to make a copy of the signal you should use the generic `copy` command (see Section 8.6). Moreover, If `s1` is a signal that already exists, then to copy all the values of `s` in `s` you can do `s1[]=s`.

4.9 Working with ranges (&range type) :

If you want to specify regularly spaced indices you can use range values (of type `&range`). The syntax to build a range is either <first>:<last> or <first>:<step>:<last>. You can remove the last point using the syntax <first>:<step>:!<last> (or simply <first>:!<last>) or the first point using <first>!:<step>:<last> (or simply <first>!:<last>). Let us note that you will see some other syntaxes in Section 9). In the first case <step> is default to 1. In both cases it corresponds (without building it effectively) to the list of numbers <first>, <first>+1, <first>+2,..., <first>+n where n is such that <first>+n $\leq$<last><<first>+n+1. Ranges have a few fields including `first`, `last`, `size` and `step` to access/set respectively the first element of the list, the last one the number of elements in the list and the step used. You can use a range for a signal concatenation and you can use extraction in the same way as for signals. Let us play around.

```
a> i=1:2:8
= 1:2:7
a> i.first
= 1
a> i.last
= 7
a> i.size
= 4
a> i.step
= 2
a> <i,0,2>
= <1,3,5,7,0,2>
a> i[1]
= 3
```

Ranges are very useful for extraction mechanism.

Remark: In numerical expressions, ranges behave most of teh times as signals, thus in order to learn more about ranges you should read the Section 9.

4.10 Extraction mechanism

4.10.1 Basic extraction

We have seen simple extraction mechanism for `&string`, `&listv`, `&signal` and `&range`. Basically if `x` is either a `&string`, a `&listv`, a `&signal` or a `&range`, we saw that the syntax `x[n]` allows to get the `n`th character or value of `x`. Actually you can extract several values at once using the syntax `x[n1,n2,...,nN]` the result is `&string`, `&listv` or a `&signal` or a `&range` (when possible) depending on the type of `x` made of the elements `x[n1]`, `x[n2]`, ..., `x[nN]`.

```
a> s='abcdef'
= 'abcdef'
a> s[1,3,5]
= 'bdf'
a> l={'a' 1 'b'}
= {'a' 1 'b'}
a> l[0,2]
= {'a' 'b'}
a> x=<1,2,10,-3>
= <1,2,10,-3>
a> x[0,2]
= <1,10>
```

If you want to specify regularly spaced indices you can use range values

```
a> s='abcdef'
= 'abcdef'
a> s[1:2:5]
= 'bdf'
a> s[5:-1:0]
= 'fedcba'
```

You can ommit <`first`> or <`last`> in the range if default value is clear

```
a> s[1:2:]
= 'bdf'
a> s[:-1:]
= 'fedcba'
```

The notations `@<` refers to the minimum possible index (generally 0) and `@>` to the maximum. Thus if you want to extract the whole string but the last character you can use either

```
a> s[:s.length-2]
= 'abcde'
```

or

```
a> s[:@ >-1]
= 'abcde'
```

You can combine ranges with other ranges or single indexes (and they don't have to be sorted!):

```
a> s[1:3,0,0,1:2:]
= 'bcdaabdf'
```

In order to build complex list of indices you are allowed also to use within the brackets `[]` listv of indices and signals made of indices values! Thus for instance if `x` is a signal

```
a> x=<1,2,10,-3>
= <1,2,10,-3>
```

and if you want to build a signal made of the first 2 values repeated 3 times, you could either do

```
a> <x[:1],x[:1],x[:1]>
= <size=6;1,2,1,2,1,2>
```

or using the listv syntax

```
a> <{x[:1]}*3>
= <size=6;1,2,1,2,1,2>
```

or even working on the indices directly

```
a> i = {0:1}*3
= {0:1 0:1 0:1}
a> x[i]
= <size=6;1,2,1,2,1,2>
```

If s is a string,e.g.,

```
a> s='abcdef'
= 'abcdef'
```

and if you want to randomly choose 10 characters from that string (allowing repetitions) you can do

```
a> s[int(Urand(10)*s.length)]
= 'cefcabfdef'
```

This would work the same way on a `&listv`, a `&signal` or a `&range`.

4.10.2 `*options` in extractions

When using extractions **for getting values only** (i.e., not for setting values, as it will be addressed in the next section) one can specify options. The syntax is

$$s[*option1, *option2, ..., *optionN, indices]$$

or equivalently (with no comma between options)

$$s[*option1 * option2... * optionN, indices].$$

If not ambiguous only the first letters of the option name can be typed in. Let us see some practical example.
Normally if one of the indices used for extract exceeds the allowed range, it generates an error :

```
a> x = 'this is'
= 'this is'
a> x[10,2]
** Error :  x[10,2]
**                .^.
** --> Indexes are out of range
--> x[10,2]
```

You can avoid that by specifying the option `*nolimit`. If it is specified then all the indices out of range are ommited. Since the option `*nolimit` is the only one (among `&string` extraction options) to start with the letters `*no` you just need to specify these letters, e.g.,

```
a> x[*no,10,2]
= 'i'
a> x[*no,3,10,2,-1]
= 'si'
```

The possible options depend of course on the type of value extracted. When compatible, it is possible to combine options using the syntax

*x[*opt1,*opt2,...,*optN,value1,...]*

or equivalently

*x[*opt1*opt2... *optN,value1,...]*

Remark (Getting some help on the extraction options) : There two ways of getting the list of possible options. While you are typing your command line, after typing `[`, if you hit twice the escape key you will get some help on the options. To get some more explicit help you can hit the escape key and then the `h` key. Otherwise, you can just type in the terminal window `help &type` where `type` is the type you are interested in. It will print a full help on the specified type.

We will not give here an exhausted list of all extraction options for all the types. We will talk about the main ones. Actually, a lot of options are shared by `&string`, `&listv`, `&signal` and `&range` types. They allow to manage "border effect", to specify what happens when an index is out of range. We have already talked about the `*nolimit` option (which works for all the types mentionned above). There are some other ones :

- `*nolimit` : indices out of range are not taken into account,
- `*bperiodic` : an index `i` out of range is replaced by the positive remainder `i%length` of the division of `i` with the length of the value (e.g., the size of the `&signal` of the length of the `&listv`), i.e., it amounts in "periodizing" the value,
- `*bmirror1` : an index `i` out of range is replaced by `i%(2*length)` if this new index is in the range `[0,length[` and `j = 2*length-1-i%(2*length)` if not, i.e., it amounts in first taking a "mirror image" of the object (repeating the last point) and then periodizing with a period of `2*length`,

- `*bmirror` : an index `i` out of range is replaced by `i%(2*length-2)` if this new index is in the range `[0,length[` and `j = 2*length-2-i%(2*length-2)` if not, i.e., it is equivalent to `*bmirror1` except that the last and the first element are not repeated when taking the "mirror image",
- `*bconst` : an index `i` out of range is replaced by `length-1` if `i`$\geq$ `length` and 0 if `i`<0.

Strings have one more, the `*list` option which is described in Section 8.2.1. Ranges and signals have much more, they will be described in Section 9. We can play around with the options described above. We will give an example with a `&range` but it works exactly the same way with `&string`, `&listv` and `&signal`.

```
a> s = 1:.5.:2
= 1:0.5:2
a> <s>
= <1,1.5,2>
a> s[*no,-3:2]
= 1:0.5:1.5
a> s[*bper,-3:1]
= <1,1.5,2,1,1.5>
a> s[*bmirror1,-3:1]
= <2,1.5,1,1,1.5>
a> s[*bmirror,-3:1]
= <1.5,2,1.5,1,1.5>
a> s[*bconst,-3:1]
= <1,1,1,1,1.5>
```

4.10.3 Using extractions for setting values

Except for single indexed extraction (e.g., `s[i] = 2`) we have not yet seen any example of an extraction syntax used to set several indices at once. When you perform an assignation (using `=`) and using an extraction syntax sith several indices, by default, LastWave assumes that the number of indices must match the number of elements on the right handside of the `=` sign. Let us give an example :

```
a> s = 'hello'
= 'hello'
a> s[1,2] = 'ELL'
** Error :  String lengths do not match
--> s[1,2] = 'ELL'
a> s[1,2] = 'EL'
= 'hELlo'
```

This would work exactly the same way on `&listv` or `&signal` (not on `&range`, there is no way to use an extraction to set range values). Let us give an example on a signal :

```
a> s = <1,10,12,3,-2>
= <1,10,12,3,-2>
```

```
a> s[:2,4] = <0,0,0,0>
= <0,0,0,3,0>
```

WARNING : You cannot use extraction options (`*options`) for setting values.

In the last example, we would like to set the values of the signal `s` corresponding to index 0,1,2 and 4 to the same value : 0. You would want to be able to write `s[:2,4] = 0`. However, in more complex situations, it can be very misleading. LastWave ask you to be always aware when the number of elements on left handside and right handside of the `=` sign might be different . You must use the operator `:=` instead of `=`

```
a> s = <1,10,12,3,-2>
= <1,10,12,3,-2>
a> s[:2,4] = 0
** Error :  Size of both handsides should match (left size = 4, right size =1)
--> s[:2,4]=0
a> s[:2,4] := 0
= <0,0,0,3,0>
```

On a `&listv` it works the same way

```
a> s={'a' 1 'b' 2 'c' 3}
= {'a' 1 'b' 2 'c' 3}
a> s[1:2:]  := {0}
= {'a' 0 'b' 0 'c' 0}
```

What if we want to replace a whole of successive values by another bunch of values (not necessarily of the same size)? Or equivalently, what if we want to replace a whole bunch of characters in a string by another bunch (not necessarily of the same size)?

```
a> s = 'hello'
= 'hello'
```

Let us say that we want to replace the substring `lo` by the letter `p`, i.e., to change `hello` into `help`. It is clear that `s[3:4] = 'p'` will not work because the sizes of the strings on the left and right handside of the `=` sign don't match. `s[3:4] := 'p'` will not lead to an error but will not perform what we want, i.e., it will change both the 3rd and 4th indexed character by the same letter `p` and then would lead to the string `helpp` and not `help`. We actually want to change the characters associated to the *group of indices* `3,4` by the group formed by the single letter `p`. This is done using a `&listv` on the right handside

```
a> s[3:4] = {'p'}
= 'help'
```

You could do the same things with two groups of indices

```
a> s = 'hello'
= 'hello'
a> s[0,3:4] = {'la b' 'la donna'}
= 'la bella donna'
```

To erase the first word, you could simply do

```
a> s[0:2] = {''}
= 'bella donna'
```

Let us note that you do not have to use the `:=` sign because the number of groups on each side is the same. However, if you want to replace several groups of letters by the same group of letter, you can do

```
a> s = 'la bella donna'
= 'la bella donna'
a> s[:2,8:]  := {''}
= 'bella'
```

All these examples will work exactly the same way on `&signal`. As you will see in a later sections, this will allow very concise high-level manipulations on strings or signals or listv's.

Chapter 5

The command line

5.1 The command line syntax

LastWave knows also about *commands*. We have already seen the `printf` and the `setv` commands (see Section 3.4). As we have seen, the assignation `=` actually corresponds to the `setv` command, e.g.

```
a> l='hello baby'
= 'hello baby'
a> setv l 'hello baby'
= 'hello baby'
```

are totally equivalent statements (the `=` sign is replaced at compilation time by the `setv` command).
Syntaxically a command line is just a list of *words*, the first word being the command name and the other words the arguments. Each word is separated using either the space character, the new line character `\n` or the carriage return character `\r`. However no word separation occurs within braces `{...}`, parenthesis `(...)`, single quotes `'...'`, double quotes `"..."`, backquotes `` `...` `` and brackets `[...]`. The meaning of backquotes will be explained in Section 5.6 and the meaning of brackets will be explained in Section 5.5. So whenever you type in some characters and then push the carriage return key, the interpreter

- tries to separate the line into words and check that the first word corresponds to a valid command name (i.e., starting with a letter or the `_` character and followed by any alphanumeric character including `_`). If it does it just calls the corresponding command with the corresponding arguments (by default each argument is evaluated) and prints the returned value if any. If the command does not exist it generates an error.

- if the first word does not correspond to any valid command name, it tries to evaluate the expression. If the evaluator returns successfully, it prints the result, otherwise it returns an error.

- Let us note that there is one exception to that rule : for assignement the `=` sign can be used and corresponds to the command `setv`. However the `=` sign is not the first word of the command line and you do not need to surround the `=` sign with spaces.

Thus

```
a> setv l {'hello ' 'baby'}
= {'hello ' 'baby'}
```

is considered by the interpreter as a list of 3 words: `setv`, `l` and `{'hello ' 'baby'}`. The first word `setv` is interpreted as the command name and the 2 other ones as arguments to this command. Let us note that the command `setv` does *not* evaluate its first argument (which is `l`) however it does evaluate its second argument.

Remark : If you want to put several commands on the same line you can use the command separator `;;` (two semi colons), e.g.

```
a> l=3;;j=2
= 2
a> l
= 3
a> j
= 2
```

5.2 The escape \ character

If you want any "special" character (e.g., {, }, ', ", [, *space, carriage return*, ...) to be considered as a "regular" character, you just need to escape it using the backslash character \. Thus

```
a> setv l 'I don\'t care'
= 'I don't care'
```

Thus, if you want to write a command on several lines, you can escape the new line character

```
a> setv l \
--> 'I don\'t care'
= 'I don't care'
```

Moreover some escaped characters have a special meaning, thus, for instance, a new line character can be obtained using the character sequence `\n`. This sequence is not used as a word separator. The escaped characters are

- an escaped new line , `\n` (new line), `\r` (carriage return), `\t` (tabulation), `\ ` (a space character) : None of these characters are word separators
- `\{, \}, \[, \], \', \", \‘, \\` : these escaped characters correspond to the character that follows the \ sign.

5.3 Commands on several lines - Aborting commands

You could type in any command on several lines. For instance

```
a> l = {
-- > 1 2 'er'
-- > }
-- > = {1 2 'er'}
a> l = \
-- > {1 2 'er'}
-- > = {1 2 'er'}
```

Indeed, after typing in the first line, LastWave waits for the matching closing brace } before sending the line to the interpreter. As soon as it is typed in, the command is executed. If we did not, the first closing brace would have ended the command. **A command can take several lines as long as you put an \ before each carriage return or if at the end of a line there is at least one of `[, {", (, ', ", ‘` that has not been closed.**

Aborting a command: When LastWave is waiting for the end of the command line to be typed in the terminal, you can abort the command by just typing in a dot `.` right after the `-->` prompt.

5.4 Actions associated to commands

Some commands take as the first argument an *action* name. For instance the `listv` command can be associated to 3 actions : `map` for mapping a command to each element of a listv, `niceprint` for printing nicely a listv in columns and `sort` for sorting the elements of a listv. Thus for instance if a listv is only made of numbers, the action `sort` sort the the numbers:

```
a> l = {1.5 0 -3 12}
= {1.5 0 -3 12}
a> listv sort l
= {-3 0 1.5 12}
```

(let us note that you will learn how to specify your own sorting command later in this manual.)

5.5 Embedded call using brackets [...]

It is important to understand that in this last example thevariable `l` did not change. The command returned a new listv which corresponds to the listv `l` which has been sorted. In order to store the result of this command into another variable you need to use the `[...]` syntax. You would type

```
a> l1=[listv sort l]
= {-3 0 1.5 12}
```

```
a> l1
= {-3 0 1.5 12}
a> l
= {1.5 0 -3 12}
```

Thus, if the argument a command must be evaluated and if this argument is surrounded by brackets `[...]`, whatever is between brackets is considered as a command line which is evaluated. The resut of this evaluation is the value of the argument.

5.6 The ‘ . . . ‘ syntax

Backquotes are simply used to forbid word separation during the command line syntax analysis. Thus, for instance, if you type

```
a> cmd ‘this is arg1‘ arg2
```

it calls the command named `cmd` with the first argument being `this is arg1` (NOT the string `'this is arg1'`) and the second argument being `arg2`. This syntax is mainly used to build dinamically some scripts as you will see in Sections 7.3 and 7.2.

Chapter 6

Basic LastWave commands

6.1 The `echo`, `print` and `info` commands

In Section 3.4 we have already seen the command `printf` and the command `setv`. The command `printf` allowed to print formatted strings. There are two other commands for printing, the first one is the `echo` command which simply prints its arguments without evaluating them. Thus, for instance,

```
a> echo hello baby
hello baby
```

The other printing command you should know about is the `print` command which allows to print (in an extensive way) its (evaluated) arguments.

```
a> l='hello baby'
= 'hello baby'
a> print l l*2
l =
'hello baby'
l*2 =
'hello babyhello baby'
a> print {l}*10
{l}*10 =
&listv {
  0 :   'hello baby'
  1 :   'hello baby'
  2 :   'hello baby'
  3 :   'hello baby'
  4 :   'hello baby'
  5 :   'hello baby'
  6 :   'hello baby'
  7 :   'hello baby'
  8 :   'hello baby'
  9 :   'hello baby'
}
```

```
a> print <{<1,2>}*4>
<{<1,2>}*4>=
  0 1
  1 2
  3 1
  3 2
  4 1
  5 2
  6 1
  7 2
```

The last example prints a signal made up of the numbers 1 and 2 repeated 4 times (the first column is the index number and the second number the corresponding value of the signal).

You should know also about the command `info` which gives you information about a variable

```
a> l='hello'
= 'hello'
a> info l
Type '&string'   [nref=2] [0x209c93e8]
    length = 5
```

It gives you the type of the variable and in the case of a string, its length. Let us note that it gives you also 2 other informations mainly for debugging purposes : the number of references of the value and the address of the value. In the last case, the number of references was 2 : 1 reference by the variable `l` and one by the command `info` during its execution.

6.2 The `if` command

The `if` command has the following syntax :

```
if <testExpr1> <bodyScript1> elseif <testExpr2> <bodyScript2> ...
```

where <`bodyScriptN`> are scripts, i.e., list of commands (generally) surrounded by {...} (in Section 7.3, we will explain exactly what the syntax for a script is) and <`testExprN`> are expressions which should evaluate to a number. The test fails if and only if the number is 0. The `elseif` and `else` clauses are optional. Let us look at a simple example:

```
a> r=1
= '1'
a> if r==1 {echo YES} else {echo NO}
YES
```

Warning: There should not be any space in the test expression `r==1`, otherwise the command line will be split into words in the wrong way. In order to avoid that you could use `(...)`, i.e.,

```
a> if (r == 1) {echo YES} else {echo NO}
YES
```

You could type in the `if` command on several lines, i.e.,

```
a> if (r == 1) {
-- > echo YES
-- > } else {
-- > echo NO
-- > }
YES
```

Indeed, after typing in the first line, LastWave waits for the matching closing brace } before sending the line to the interpreter. As soon as it is typed in, the command is executed. This is the reason why we had to write the first closing brace on the same line as the `else`. If we did not, the first closing brace would have ended the command.

Remark: Let us recall that when LastWave is waiting for the end of the command line to be typed in, you can abort the command by just typing in a dot `.` right after the `-->` prompt.

Test with signals There are a lot of very convenient operators on signals (resp. images) that let you test the values of a signal (resp. images). They will be fully described in Sections 9 and 10, however we will give a couple of examples here. The operators `>`, `<`, `>=`, `<=`, `==` and `!=` when applied on signals compare each value of a signal and returns a signal made of 1's and 0's depending on the results of the test. Thus for instance

```
a> s = <1,2,-1,0>
= <1,2,-1,0>
a> s>=0
= <1,1,0,1>
a> s>=<3,1,1,1>
= <0,1,0,0>
```

There are two very important operators: `all` and `any`. The first one returns 1 if all the values of a signal are non 0 and the second one returns 1 if any of them is non zero. In other cases both return 0. Thus to test if all the values of a signal are strictly positive, you should just type

```
a> s = <1,2,-1,0>
= <1,2,-1,0>
a> if (all(s>0)) {echo YES} else {echo NO}
NO
a> if (any(s>0)) {echo YES} else {echo NO}
YES
```

Moreover the `find` operator lets returns a listv of all the indices corresponding to non zero values. Thus in order to build a signal only made of the positive values of the original signal, one could type

```
a> s1 = s[find(s>0)]
= <1,2>
```

6.3 The `source` and `setsourcedirs` command

This command allows to execute all the commands contained in a file (referred to as a *script file*). The syntax is source <`filename`> The interpreter will look for the file named <`filename`> in the current directory and in all the directories corresponding to the list that can be set or get using the `setsourcedirs` command. The command `setsourcedirs` with no argument sends back the current listv of script directories (i.e., a listv made of strings). The command `setsourcedirs` <`listv of directories`> sets the list of directories to <`listv of directories`>. Thus, to add a directory <`dirName`> to the source directory list, one just needs to type in

```
a> setsourcedirs [setsourcedirs]+<dirName>
```

Remark : In a script file, the character `#` used at the begining of a line comments out the whole line.

6.4 Loops

6.4.1 The `foreach` command

There are several way of looping in LastWave. The most efficient one is the `foreach` one. It allows to loop on a `&listv`, a `&signal`, a `&range` or a `&list`. We have not yet introduced the `&list` type, mainly because it is not often used (see Section 8.2.1). A &list is actually nothing but a &string that can be separated into words (using the same rule the command line is separated into words). The syntax of the `foreach` loop is

*foreach *variableName* <val> <script>*,

where *variableName* is a (non evaluated) variable name (the fact that it is not evaluated is indicated in the syntax by the surrounding `*`), <`val`> is an evaluated argument which can be of type `&listv, &signal, &range` or `&list` (i.e., `&string`) and where <`script`> is a script. The command `foreach` loops on each index of <`val`>, assigns the variable `*variableName*` with the corresponding value and executes the script. It ends when the last index is reached.

Using `foreach` on `&range` For instance, to build a &listv whose elements the integers $1 \ldots 10$, you could type in

```
a> l = {}
= {}
a> foreach i 1:10 {l+=i}
a> l
= {1 2 3 4 5 6 7 8 9 10}
```

Let us note that you could type in the `foreach` command on several lines, i.e.,

```
a> foreach i 1:10 {
-- > l+=i
-- > }
```

Using `foreach` on `&signal` If you want to create a listv of 10 (Gaussian) random numbers you could type in

```
a> l = {}
= {}
a> foreach r Grand(10) {
-- > l+=r
-- > }
a> l
= {0.067469 -0.0630144 0.356936 1.45415 -0.320958 0.804815 -0.739786 -2.00512 -1.39016 -
```

The command `Grand(10)` creates a signal of 10 Gaussian random numbers.

Using `foreach` on `&listv` You can then loop on the listv `l` to compute the sum and the products of the numbers

```
a> p = 1
= 1
a> s=0
= 0
a> foreach e l {
-- > p*=e
-- > s+=e
-- > }
a> p
= 0.000812681
a> s
= -2.52705
```

Let us note that if you want to write the different commands of a script on the same line, you can use the `;;` separator, i.e,

```
a> p = 1;;s=0
= 0
a> foreach e l {p*=e;;s+=e}
```

Let us note that you could use the operator `sum` to compute the sum of the values of a signal (e.g. `sum(Grand(10))`).

Using `foreach` on `&list` As explained above, a &list is actually just a &string that can be splitted into *words*, using the same rules as for splitting command lines into *words*. Thus, for instance, in order to split the words of a sentence and make it into a listv :

```
a> s = 'this is the sentence'
```

```
= 'this is the sentence'
a> l = {}
= {}
a> foreach word s {l+=word}
a> l
= {'this' 'is' 'the' 'sentence'}
```

6.4.2 The `for` loop

Let us recall that the `foreach` loop is much faster than the `for` loop. Thus, you should try to use as many `foreach` as possible and avoid using *for loops* whenever it is possible. The `for` command has the following syntax :

for <*startScript*> <*testExpr*> <*nextScript*> <*bodyScript*>

where the <`startScript`>, <`nextScript`> and the <`bodyScript`> are scripts and the <`testExpr`> is an arithemetic expression. The <`startScript`> is executed once at the begining of the first loop, the <`nextScript`> is executed at the end of each loop and the <`bodyScript`> is the core script that you want to loop on. The loops goes on as long as the <`testExpr`> evaluates to a non zero value. Thus, for creating a listv of 10 positive Gaussian random numbers you could type in

```
a> for {l={};;i=0} i<10 {} {
-- > r=grand
-- > if (r>=0) l+=r;;i+=1
-- > }
a> l
= {0.508718 0.122369 1.03 0.858427 0.105972 1.64096 0.21377 0.625155 0.680699 0.395544}
```

6.4.3 The `while` loop and the `do` loops

The syntaxes are

while <*continueTest*> <*bodyScript*>,

and

do <*bodyScript*> <*continueTest*>.

These are the regular `do` and `while` loops, there is nothing to comment on.

6.4.4 The `break` and `continue` commands

Inside any <`bodyScript`> of any loop (`foreach, for, while` or `do`), one could use the `break` command to go out of the loop immediately or the `continue` command to continue the loop as if the end of the <`bodyScript`> was reached. Thus

```
a> for {l=;;i=0} i<10 {} {
-- > r=grand
-- > if (r>=0) l+=r;;i+=1
-- > }
```

can also be written

```
a> for {l=;;i=0} 1 {} {
-- > if (i==10) {break}
-- > r=grand
-- > if (r<0) {continue}
-- > l+=r
-- > i+=1
-- > }
```

6.5 The help system

6.5.1 Terminal line edition and on-line Help

In order to learn about them you should run the `HelpTerm` command. It will give you a complete description of the command line editing facilities. Here is a short description. First, you should be able to use the arrow keys for moving the cursor around or for using history. You can also use the tabulation key for history completion and the key sequence `esc esc` (escape key twice) for command completion, field completion and for getting the usage of the command (or field, or extraction option) currently displayed on the terminal input line. `esc f` (on unix computers) ot `shift-esc` will perform file completion. If you need a short description then you can use `esc h`. In these helps, the <..> are used for delimiting arguments that will be evaluated, `*...*` for delimiting non evaluated arguments, the brackets `[...]` are used for optional arguments and the parentheses `(.. | .. | ..)` for multiple choice arguments. On unix machine you also have the standard `ctlr a` (begining of line), `ctrl e` (end of line) and `ctrl u` (erase line) keys. On a Macintosh, these keys are respectively `Opt-Left Arrow`, `Opt-Right Arrow` and `Opt-Delete`. The behavior of all these keys were defined in the script file `scripts/terminal/keys`.When you've learned about using LastWave, please feel free to change them.

In the case this editing facility does not work properly, it certainly means that the key codes corresponding to the arrow keys and the `ctrl` keys are not the one expected. You should restart the program without executing the `startup` file. Write down the codes for the arrow keys and the `ctrl` keys (just hit them and their codes will appear such as `escAD` for `escape+A+D`). Then just replace the codes in the `scripts/terminal/keys` file with yours and then execute the startup file or restart the program. Otherwise, the best way might be to just use a "regular" xterm for running LastWave.

6.5.2 The `help, helpp and helpv` commands

The `help` command lets you ask some help about a command or about a `&type` (e.g., `help print` or `help &listv`). The `helpp` command lets you ask some help about a package (packages are LastWave toolboxes). If you type `helpp` with no argument it gives you the list of all available packages, otherwise it gives you help on a specific package and the list of all the corresponding commands. Finally the `helpv` command lets you get some help on a specific variable.

6.5.3 The `apropos` commands

The `apropos` command lets you obtain a list of all the commands related to a topic. More precisely, if you type `apropos *word1* ...*wordN*`, it looks for commands whose help contain all the words `*word1* ...*wordN*`. the list of the so-obtained commands is printed (package by package) and for each a command the help string around the word `*word1*` is printed.

6.5.4 Aborting a command

When you hit the carriage return key after entering a command line, if a } or a " is missing the interpreter will display the prompt `-->` instead of the regular prompt. It indicates that it is waiting for the end of the command to be typed in. If you want to abort the command you can type `.` (dot) character right after the prompt and then hit the carriage return key. That will terminate the command and ask for a new command to be typed in. Otherwise, you can just type the end of your command line. The interpreter implicitely assumes that the end of the command is reached when you hit the carriage return key and all the { and " have been closed. However if you have not finished typing your command and want to continue it on the next line, you can escape the carriage return by typing \ just before it. You can repeat this as many times as you want.

Chapter 7

Defining new commands using the command language

7.1 The `setproc` command

7.1.1 Some simple examples

In order to define a new script command, i.e., a new command associated to a script, one should use the `setproc` command. Its basic syntax is

*setproc *commandName* <variables> <bodyScript>*

Remark : from now on, we suggest that instead of typing directly the `setproc` definitions in the terminal window, you type them in a script file and that you source that file. It saves time in case you make typos. Thus, from now on, we will ommit the prompt for `setproc` definitions.

Thus one can write a simple command that returns a (uniformely distributed on [0,1]) random number :

```
setproc my {} {
  return urand
}
```

In order to use the function you just need to type

```
a> my
= 0.2873768
a> my
= .89182732
```

Command names: The name of a command should start with a capital or small letter or the _ character and should be followed by any alphanumeric character including _. There is a very important convention in LastWave that you should know about. All the names of the commands that should not be called directly by the user start with the underscore character _. Thus, these functions are not described in this manual. Moreover, if you write new packages, please keep that notation in mind...

If you want to write a command with one argument that adds 1 to the first (numerical) argument, you could write

```
setproc my {n} {
  return n+1
}
```

7.1.2 Optional arguments

If you want to be able to specify as an (optional) argument the amount that must be added to the first argument, you would write

```
setproc my {i {n 1}} {
  return i+n
}
```

Then

```
a> my 10
= 11
a> my 10 2
= 12
a> my 'ab' 'cd'
= 'abcd'
```

7.1.3 Checking the type of arguments

By default, all the arguments of a script command are evaluated and no type checking is performed. This explains why the command `my` works both for integers and for numbers. If you want to forbid the `my` command to be called with something else than numbers, you can specify a type for any of the argument. The syntax for each argument is the following

```
{/&type1] arg1 [defValue]}
```

Thus for instance, one could write

```
setproc my {{&num i} {&num n 1}} {
  return i+n
}
```

Then

```
a> my 'ab' 'cd'
** Error :  Bad type &num for argument ab
```

Let us note that you could use *extended* types such as `&int` as argument type. Extended types are not "true" types, they can be used only in the argument definition of a procedure (See Section 7.4). One very important extended type is `&word`. It allows not to evaluate the argument of a procedure. For instance, if you want to define a procedure `my` that has an action name as a first argument, you could type

```
setproc my {{&word action} i} {
  if (action=='str') {return i+'1'}
  if (action=='num') {return i+1}
  errorf 'unknown action %s' action
}
```

Then

```
a> my num 1
= 2
a> my str '1'
= '11'
a> my uy '1'
** Error :  Error :  unknow action uy
```

The `errorf` command works as `printf` except that i) it generates an error and ii) a new line is added at the end of the string. Ideally, you might want to check the type of the argument `i`. Since its type depends on the action, you cannot include the type in the definition of the procedure. You want to do dynamic type checking. This can be done using the `val type` command, i.e.,

```
setproc my {{&word action} i} {
  if (action=='str') {
    if ([val type i]=='&string') {return i+'1'}
    errorf 'Bad type %s for second argument' [val type i]
  }
  if (action=='num') {
    if ([val type i]=='&num') {return i+1}
    errorf 'Bad type %s for second argument' [val type i]
  }
  errorf 'unknown action %s' action
}
```

7.1.4 Command with a variable number of arguments

As we already explained for multiple assignation with a listv (see Section 4.6.1), the last variable of a command can be "dotted", i.e., LastWave will assign to that variable all the (evaluated) arguments which have not been assigned yet. Thus for instance

```
setproc my {arg1 .l} {
  printf 'arg1=%V, l=%V\n' arg1 l
}
a> my 1 2
arg1=1, l={2}
a> my 1 2 'e'
arg1=1, l={2 'e'}
a> my 1
arg1=1, l={}
```

Let us note that the `%V` (resp. `%v`) format in `printf` lets you print any value using a long (resp. short) string representation.

Remark: the dotted variable by default is of type `&listv`. However, if you do not want all the remainder arguments to be evaluated, you can specify the type `&wordlist` (see Section 7.4) for the dotted variable. The only valid types for a dotted variable is either `&listv` or `&wordlist`.

7.1.5 Specifying the usage and the help of a command

When defining a script command, you can specify the usage and the help of a command. Right after the argument definitions and before the script of the command you should insert a string of the form

*'{{{*usageAction1*} {*helpAction1*}} ...{{*usageActionN*} {*helpActionN*}}}'*

where `usageActionK` is the usage corresponding to the `K`th action (if any) of the command and `helpActionK` the corresponding help. If the command has no action, you should just put the usage in place of `usageAction1` and the help in place of `helpAction1`. Thus, for instance,

```
setproc my {n} '{{{<n>} {Adding 10 to <n>}}}' {
  return n+10
}
```

Then

```
a> help my
[Terminal]
--- my <n>
  Adding 10 to <n>
```

The `[Terminal]` string lets you know that the script command was defined directly in the terminal and not from a file.

7.1.6 Overloading a C-command

You can overload a LastWave command defined in C by defining a new script command which has the same name as the C-command. If you want to access the original C-command you can either delete the script command or escape its name when you use it. Thus, if the command name is `cname`, you access the original C command by the name `\cname`. However since the \ character is a special character (for escaping other characters such as a { sign), if you just type `\cname`, it will try to escape the `c` character. Thus you need to escape the \ character. So, to call the C command associated to the name `cname` you must type `\\cname`.

7.1.7 Are arguments of a command copied?

As we have already explained, when assigned, a listv or a signal is not copied. This is also the case when passed by arguments. Actually if the type of an argument is either `&listv`,

`&signal`, `&image`, `&script` or `&proc`, the argument is *not* copied. In all the other cases (i.e., `&num`, `&range` or `&string`) the argument is copied before being passed.
Let us note that all the other types defined in packages are always passed without being copied.

7.2 The `&proc` type - Anonymous command

If you type in the terminal

```
a> setproc my return 1
= %my
```

you see that the setproc command returns the value `%my` which is the syntax used to refer to the command named `my`. The type of this value is

```
a> type(%my)
= '&proc'
```

the string '`&proc`'.

Fields of a `&proc` The fields are

- `help`: (read only) a listv describing the usage and help
- `shelp`: a string describing the usage and help. The string must be formatted as the help string argument of the `setproc` command (see Section 7.1.5)
- `file`: (read only) the filename the command was defined in (it is `Terminal` if defined in the terminal)
- `package`: (read only) the package name the filename the command is attached to
- `script`: (read only) the script associated to the command (value of type `&script`, see Section 7.3)
- `name`: (read only) the name of the command

In order to delete a command you must use the `proc` command associated to the `delete` action.

You can define anonymous command using the syntax

%{argument definition}‘script text‘

where the *argument definition* ahs the same syntax as for the `setproc` command. Thus for instance

```
a> c = %{n m}‘return n+m‘
= %<0x23740c40>
```

Since it is an anonymous procedure, it does not have any name, thus the address (i.e., `0x23740c40`) is used as the name for printing. You can then use the `apply` command to apply this procedure to a list of arguments. You have two options, either you want to simulae the call to the procedure as a regular call, e.g.

```
a> apply args c 1 10
= 11
```

or you want to send the arguments as a listv of arguments

```
a> apply listv c {1 10}
= 11
```

Let us note that this second way of calling the command keeps you from sending non-evaluated arguments.

We can use anonymous command to sort listv's. For instance, if we want to sort a listv of integers into two classes, even numbers and odd numbers, we would do

```
a> listv sort {1 2 7 26 3 5 42 7 1} %{e1 e2}‘return e1%2-e2%2‘
= {1 7 1 3 5 7 42 26 2}
```

The second argument of `listv sort` must be a command with two arguments, which returns -1, 0 or 1 depending on whether te first one is smaller, equal or greater to the second one. The command `listv` lets you also `map` each element using a command and form another listv of all the resulted values. For instance

```
a> listv map 1 2 "se" 4
= {3 6 'sesese' 12}
```

Building commands dynamically Let us define a command with one (numerical) argument `n` that returns a command which multiplies its only argument by `n`:

```
setproc generate {{&int n}} {
  return %{x}‘return x*$n‘
}
a> c = [generate 3]
= %<0x2380d0a8>
a> apply args c 9
= 27
a> apply args c 'e'
= 'eee'
```

If you want to build more complex scripts, you can use a string variable to elaborate your script and then just define the procedure at the end, e.g.,

```
setproc generate {{&int n}} {
  str = 'return x*'+'$n'
  return %{x}$str
}
```

7.3 The &script type

Up to now whenever LastWave was expecting a script as an argument you used the syntax

{*script text*}

This was the case for instance for the `setproc`, the `foreach` or the `if` commands. This is one way of specifying a script. Another way (that lets you define scripts dynamically easily, in the same way as explained in the previous Section) is the syntax

%%‘script text‘

Thus

```
setproc my {i} {return i+10}
```

is equivalent to

```
setproc my {i} %%‘return i+10‘
```

or

```
str = ’return i+10’
setproc my {i} %%$str
```

or even to

```
s = %%‘return i+10‘
setproc my {i} s
```

When defining a command you can get its script using the `script` field, e.g.

```
a> setproc my {i} {return i+10}
= %my
a> s = %my.script
= (&script)
a> print s
s =
(&script) {
  return i+10
}
```

You can evaluate a script (in the current environment) :

```
a> eval s
** Error :  i+10
** --> Variable i does not exist
```

You get an error because the variable `i` does not exist in the current environment. So the evaluation of `i+10` led to an error. You could do

```
a> i=30
= 30
a> eval s
= 40
```

7.4 Extended types

As already explained, the extended types are not, properly speaking, types. They have the same syntax as regular types (they start with &) however they are used only for type checking, and mainly in the `setproc` definition. The extended types are

- `&val` : this is the default type of any argument of a procedure. It just means that the argument must be evaluated. The effective type of the so-obtained value can be anything.
- `&valobj` : this is the same type as `&val` except that the value must not be a number (this is useful for management of the current object, see Section 8.5
- `&list` : this is the same type as `&string` except that the string can be splitted into *words* using the same rule as for splitting a command line into *words* (see Section 8.2.1)
- `&word` : it means that the argument must not be evaluated. It is for instance the type you should use for an argument which corresponds to an *action* of a command. The so-obtained argument is a `&string` which value is the original expression
- `&wordlist` : same as `&word` except that the expression must correspond to a `&list`
- `&int` : a number which must be an integer
- `&var` : the argument should correspond to a variable. This variable is sent to the command by reference and not by value (see next Section for examples)

7.5 Environments and levels - Importing variables

While executing the script of a script command, the only accessible variables are the variables in the current "environment" meaning the variables that were declared in the command definition using `setproc`. Let us imagine that you want to define a script command `mul2` that changes the value of a variable by multiplying it by 2. Thus if the variable `i` is set to 3, the call `mul2 i` should set the variable `i` to 6. Thus, in order to access (by reference and not by value) the variable `i` in the script of the command `mul2`, we need to `import` it from the calling environment to the current one. This is how it is done :

```
setproc mul2 {{&var v}} {v*=2}
```

So now we can type

```
a> i=3
= 3
a> mul2 i
= 6
a> i
6
```

Let us note that if the variable `i` did not exist then `mul2 i` would create the variable `i` and initialize with `null`.

Before moving on, you should know about another way of importing variables from a different environment to the current one. As we have just said, it could be done in specifying the type `&var` in the variable list of `setproc`. Actually, importing a variable can also be done via the `import` command. It is a much more powerful way to import variables. Its most common syntax is

*import args '*varName* [*newVarName*]'*

which imports the variable named `*varName*` in the calling environment and make it correspond to the variable `*newVarName*` in the local environment. If `*newVarName*` is not specified, the same variable name is used in both environments. If the variable `*varName*` does not exist in the calling environment, it creates it and set it to `null`. Thus the `mul2` command can be rewritten using `import`

```
setproc mul2 {&word vname} {
  import '$vname v'
  if (v==null) {v = 0} else {v*=2}
}
```

In this version of `mul2`, if the variable does not exist, it creates it and sets it to 0, otherwise it multiplies the variable by 2 (in order not to create the variable if it does not exist, you ccould use the `var exist` to test whether it exists or not before calling `import`.

Actually `import` has an optional argument which corresponds to the environment you want to import the variable from. Each environement is referred to as a number called the `level`. You can address an environment relatively to the current environment (level 0 corresponds to the current environment, level -1 corresponds to the calling environment, level -2 to the one which called the calling environment and so on) or in an absolute way (level 1 corresponds to the global environment, level 2 to the first environment called by the global one and so on). Thus, by default, import assumes you want to import a variable from the calling environment, i.e., the level -1. However, if you need to, you can specify a different level using the syntax

*import args [<level>=1] '*varName* [*newVarName*]'*

(Actually, let us note that import lets you import several variables at the same time, you check the full syntax). A particular case is when you want to import a global variable. You could either do

*import args 1 '*globalVarName* [*newVarName*]'*

or use the `global` command

*global '*globalVarName* [*newVarName*]'*

Let us mention that `import` ahs other actions. One of them is `list`, it lets you simulate what happens when a script command is called, i.e., you can specify optional values for variables and types. Finally, let us mention that the `eval` command lets you specify the level you want to evaluate the script at.

Chapter 8

Learn more about LastWave

8.1 Some predefined variables

The following variables are predefined LastWave variables (they are defined in the C-source code before the startup file is executed) :

- `Home` (`&string`): The home directory (empty string for macintosh computers)
- `System` (`&string`): Either `'unix'` or `'mac'`
- `Computer` (`&string`): Either `'sgi', 'dec', 'linux', 'hp', 'sun'` or `'mac'`
- `Term` (`&string`): corresponds to the unix TERM variable (for unix computers only)
- `BinaryMode` (`&string`): It is either `'little'` (for little endian, i.e., Low Byte First) or `'big'` (for big endian, i.e., High Byte First)
- `Display` (`&array`): It is an array whose indexes are :
  - `flag` (`&num`): This is either 1 (if graphics are allowed) or 0 (if not)
  - `BWScreen` (`&num`): This is either 1 (if screen is Black and White) or 0 (if not)
  - `ScreenDepth` (`&num`): This could be 8,16,24 or 32.
  - `ScreenRect` (`&listv`): The listv {`0 0 <width> <height>`} that defines the rectangle screen.
  - `ScreenType` (`&string`): This is either `'GrayScale', 'StaticGray', 'PseudoColor', 'DirectColor', 'StaticColor'` or `'TrueColor'`
- `argv` (`&string`): On Unix computers, it contains the arguments the `lw` command was executed with (except for the eventual `-b` flag that you should use when running `lw` in batch mode)

8.2 Advanced manipulations on strings (&string)

8.2.1 Working with &list - The *list extraction option

A &list is actually an **extended type** (see Section 7.4). nothing but a &string that can be separated into **words** using the same rule the command line is separated into words. As explained previously, you can manipulate lists using the **foreach** as command, e.g.,

```
a> l = '1 2 {my baby} yes'
= '1 2 {my baby} yes'
a > foreach i l {printf '
1
2
{my baby}
yes
```

Actually, you can manipulate list (not for assignement) using the extraction mechanism with the *list option. When this option is set, the indices refer to the different words and no longer to the different characters, e.g.

```
a> l[2]
= '2'
a> l[*list,2]
= '{my baby}'
a> l[*list,2:]
= '{my baby} yes'
```

Let us note that this option can be used along with the other extracting options (e.g., *bperiodic, *nolimit,...).

8.2.2 The str substr command

The **str** command has several actions. The two most useful actions are the **substr** action to find a substring within a string and the **match** action to find occurences of regular expressions within a string. Let us first look ar the first action.

```
a> l='hello hello baby'
= 'hello hello baby'
a> str substr l 'hello'
= {0:4 6:10}
```

Thus **str substr** returns a listv of ranges corresponding to indices where the substring **'hello'** was found. If you just want to look for the first occurence you use the optional argument (the last one)

```
a> str substr l 'hello' 1
= {0:4}
```

If LastWave does not find any such substring an empty listv is returned

```
a> str substr l 'r'
= {}
```

Since, we have seen that one can make multiple assignement of substrings using the syntax `str[range1,...,rangeN] = {substr1,...,substrN}` or `str[range1,...,rangeN] := {substr1}`, in order to replace all the occurences of the word `hello` by the word `bonjour`, you can type

```
a> l[[str substr l 'hello']] := {'bonjour'}
= 'bonjour bonjour baby'
a> l
= 'bonjour bonjour baby'
a> l[[str substr l 'bonjour'][@>]] = {'mon'}
= 'bonjour my baby'
```

Remark: when `str substr` look for substrings it starts at the begining of the string as soon as the first substring is found it goes on *starting again from the end* of the first substring. Thus, the obtained ranges will *never* overlap. To get the last occurence you can use the syntax `@>` to get the index of the last object of a listv.

Remark: Let us note that since `l[{}] = ...` does not perform anything, if the substring `'hello'` was not found the variable `l` would not have change. You should be aware that, on the contrary, `s[] = ...` does perform assignation.

8.2.3 The `str match` command - Wild cards

The `match` action performs exactly the same thing as the `substr` action except that you are allowed to use regular expressions in the substring. For instance if you want to get the positions of all the `b`'s or the `e`'s in a string, you would do

```
a> l = 'hello hello baby'
= 'hello hello baby'
a> str match l '[be]'
= {1 7 12 14}
```

i.e., they are in position 1, 7 12 and 14. Thus the syntax [...] means any of the characters within the brackets. You want to change each of these letters by the letter `X` ?

```
a> l[[str match l '[be]']] := 'X'
= 'hXllo hXllo XaXy'
```

The syntax `[+...]` (resp. `[*...]`) means at least one (as many) of the characters that follows thus

```
a> l = 'hello hello baby'
= 'hello hello baby'
a> l[[str match l '[+el]']] := 'i'
= 'hio hio baby'
```

If you want to strip spaces at the begining of a string you can use the ^ character that should match the begining of the string:

```
a> l = '   hello hello baby'
= '   hello hello baby'
a> l[[str match l '^[* ]']] := {''}
= 'hello hello baby'
```

The `|...|` syntax allows to specify what the returned range should match. For instance you might want to look for all the characters `'b'` which occur right after an `'a'` and change them into `'B'`. For that purpose you need to match the string `'ab'` but you do not want the ranges to include the leading `'a'`. For that purpose, you just need to do

```
a> l = 'hello hello baby'
= 'hello hello baby'
a> l[[str match l 'a|b|']] := 'B'
= 'hello hello baBy'
```

Combining the different *wild cards*, i.e., the special characters for regular expressions, along with the `match` action is extremely powerful.

Remark: the `match` action uses the type of algorithm as the `substr` action. When it looks for substrings it starts at the begining of the string as soon as the first substring is found it goes on *starting again from the end* of the first substring. Thus, the obtained ranges will *never* overlap. Moreover, it always try to match the longest substrings

The available wild cards are

- `^` : matches the begining of the string
- `$` : matches the end of the string
- `*` : stands for any sequence (even empty) of characters
- `+` : stands for any sequence (at least 1) of characters
- `?` : stands for any character
- `[...]` (resp. `[^...]`): stands for a character which belongs (resp. which does not belong) to the list of characters in between brackets
- `[*...]` (resp. `[*^...]`): stands for as many characters (even none) which belongs (resp. which does not belong) to the list of characters in
- `[+...]` (resp. `[+^...]`): stands for at least one character which belongs (resp. which does not belong) to the list of characters in between
- `[#...]` (resp. `[#^...]`): stands for one character or none which belongs (resp. which does not belong) to the list of characters in between
- any other character: tries to match it item `!...!` : excatly as if the `!` where not there except that the sequence in between the `!` is "escaped", i.e., if a wild card appears in the it is considered as a regular character.

8.3 IO - Redirections and Streams

8.3.1 Output : the `printf, sprintf, errorf` commands

There are several commands for printing formatted strings. You should know about the main ones. We have already seen the `printf` command. It uses a `format` which indicates how the string should be formatted using `%...` syntax. This `format` string has the same syntax as for the `printf` command of the C-language. Most of the `%`-forms are supported. LastWave knows about

- `%d %i %o %x %X %u` for integers
- `%f %e %E %g %G` for floats
- `%s` for strings
- `%c` for characters
- `%%` for the % character

which are standards C `%`-forms (see a C-manual for full description) and

- `%v` for values, short form (e.g., for a listv, just the length is printed)
- `%V` for values, long form (e.g., for a listv, all the values are printed)

Moreover for the standard C `%`-forms, LastWave supports the most common modifiers which are placed in between the % and the character. The `sprintf` command is equivalent to the `printf` command except that the formatted string is put into a variable instead of being printed. The `errorf` command is equivalent to the `printf` command except that a new line is printed at the end and an error is generated.

8.3.2 Input : the `scanf, sscanf, getchar, getline` commands

For reading inputs from the terminal, the `scanf` command works the same way as the C-language `scanf` command. You should read the C-manual pages of this command. LastWave supports

- `%[...], %s` for strings
- `%*` for skipping assignation
- `%d %i %o %x` for integers
- `%e %f %g` for floats
- `%c` for characters
- `%%` for the % character
- `%n` for getting the number of characters read

In the same way as `sprintf`, the command `sscanf` lets you read input from a string instead of the terminal.

You should know about the command `getchar` for reading a single character (without waiting for the carriage return) and the command `getline` for reading a whole line (waiting for teh carriage return).

8.3.3 Redirection : the basics

By default, all the IO commands operate on the terminal input/output. However LastWave lets you redirect the terminal input/output while executing a command. For instance, if you want to redirect the output of a command to a file named `fileOut`, you would type

<command> :: >fileOut

This would erase the content of the file `fileOut` and replace it by the output of the command. Instead, if you want to append the output of the command to the `fileOut`, you would have to type

<command> :: >>fileOut

Remark : Be careful : you have to put spaces before and after the `::` and no space in between the > and the `fileOut`.

Similarly, if you want to redirect the terminal input, you would have to type

<command> :: <fileIn>

You can of course combine these effects. Thus

```
a> printf '%s\n' [getline] ::  >>fileOut <fileIn
```

would take the first line of the file `fileIn` and appends it into the file `fileOut`. This is basic example of a manipulation of what is called a `stream`. Let us understand exactly what they are.

8.3.4 Streams

LastWave lets you manipulate streams associated to files, strings or to the terminal. A stream is referred to as an integer. There are two kinds of streams : the input streams (you can read from) and the output streams (you can write to). The stream 0 always corresponds to the keyboard input, the stream 1 always corresponds to the terminal standard output and the stream 2 to the terminal error output. When you type in the command

```
a> echo coucou ::  >>fileOut
```

LastWave actually opens the file `fileOut` and associates to it an output stream which is a number and that will replace the terminal standard output stream (namely 1) by this stream. Actually this could be done by opening manually the file `fileOut` and redirecting the output :

```
stream=[file open 'fileOut' 'a']
file set stdout stream
echo coucou
file set stdout 1
file close stream
```

WARNING : In order not to forget to close streams, LastWave lets not you manipulate streams outside of script commands. It automatically closes all of them as the terminal gets the control back. Thus, the examples in this section that manipulate explicitely stream numbers should be typed inside a script command.

Let us explain this latter example. In the first line we execute the command `file open 'fileOut' 'a'` that lets you open an output stream named `fileOut` and returns the stream number which is stored in the variable `stream`. Let us note that there are actually three modes for opening file streams : `r` (for input file streams), `a` (for output file streams that should append the output to the current content of the file) and the `w` mode (for output file streams erasing any former content of the file). The second line redirects the standard output to be that stream and the third line just executes the command. The 4th line set back the standard output to the terminal standard output (namely the stream number 1) and then we close the stream we opened ! In the same way, we can redirect "manually" the input. Thus instead of

```
a> getline ::  <fileIn
```

We could do

```
stream=[file open 'fileIn' 'r']
file set stdin stream
getline
file set stdin 0
file close stream
```

Let us note that output streams are associated either to the terminal (stream number 1 and 2) or to a file (`'a'` or `'w'` mode) whereas input streams can be associated not only to the terminal (stream number 0) or to a file (`'r'` mode) but also to strings. This is done in the following way

```
a> str ='Just put\n a multiple line\n string here'
a> getline ::  <<$str
```

or "manually"

```
set stream [file openstr str]
file set stdin stream
getline
file set stdin 0
file close stream
```

Let us write a simple command that would copy all the lines of the current input stream into the current output stream whatever these streams are (files, terminal or strings). This could be done in the following way :

```
setproc Copy {} {
  while (![file eof]) {
    getline line
    printf '%s\n' line
  }
}
```

Let us note that the `file` command along with the `eof` action allows to test whether a stream has reached an *end of file* character. Its syntax is

file eof [<stream>=stdin]

where <`stream`> is a stream number which by default is the constant `stdin`, i.e., the current input stream. Let us note that each time the argument of a command is expected to be a stream, you can either type in an explicit stream number of the strings `stdin`, `stdout` or `stderr` which refer respectively to the current standard input stream, the current standard output stream and the current error output stream. Thus, the command `file eof` is equivalent to the command `file eof stdin`. It returns 1 if the current input has reached an *end of file* and 0 otherwise. Let us note that, on the terminal, an *end of file* character is generated by the `ctrl-d` sequence and for a string stream it is automatically generated at the end of the string. Thus while it has not reached the end of file of the current standard input, the `Copy` command loops and read a line and just prints it. To copy the content of a file in another one, you just need to do

```
a> Copy ::  <fileIn >fileOut
```

We could do the same thing but character by character

```
setproc CopyChar {} {
  while (1) {
    getchar c
    if ([file eof]) break
    printf '%c' c
  }
}
```

Remark : On Unix computers, the command `file` along with the actions `cd`, `info`, `list` and `listp` allows to operate some basic operations such as find all the files in a directory or get some information about a file.

8.3.5 More about redirections

Up to now, we have seen 4 symbols for redirecting the standard input and ouptut streams : `>`, `>>`, `<` and `<<`. By specifying a filename right after them, this file is opened and associated to a stream that replaces one of the standard streams for the time the command

is executed. As soon as this command returns, the original streams that were used before this command was executed are on again. Actually, all of the redirection symbols can be used along with a stream that is already opened instead of a filename. Thus instead of doing

```
stream=[file open 'fileOut' 'a']
file set stdout stream
echo coucou
file set stdout 1
file close stream
```

you could simply do

```
stream=[file open fileOut a]
echo coucou >>$stream
file close stream
```

This allows you to work with a file that is kept opened and which is not closed and opened each time it is accessed (this is particularly useful when you want to read the content of a file using several commands). The symbols that you can use for redirecting standard streams are the following

- `>` : for redirecting the standard output
- `>!` : for redirecting the error output
- `>*` : for redirecting the standard output and the error output
- `>>` : for redirecting the standard output using the `append` mode
- `>>!` : for redirecting the error output using the `append` mode
- `>>*` : for redirecting the standard output and the error output using the `append` mode
- `<` : for redirecting the standard input to a stream or a file
- `<<` : for redirecting the standard input to a string

Let us note that the stream number 3 corresponds to the `nul` output stream, i.e., an output stream that does not print anything. You can redirect standard or error output towards this stream by either specifying the stream number 3 after the redirection symbol or by simply typing the redirection symbol with nothing afterwards.

8.4 Packages

A package in LastWave is a set of commands and of variable types that you can load when you need them. Let us note that graphic objects (e.g., objects to display signals,...) can be associated to a package, but we will talk about them later. To deal with packages you must use the command `package`. In order to load a package you must use this command along with the `load` action : `package load <packageFilterName>`.

WARNING : At startup, the `scripts/startup` file is sourced. Right now, this file loads all the available packages. However, it would be very rare that you want to use during the same session all the packages (mp, image, wtmm1d...). Thus you should comment out the corresponding lines in the startup file in order to prevent the packages you know you do not want to use to be loaded (LastWave will be faster). Be careful, some packages needs some other packages to be able to run. You will get an error if for instance you try to use the wavelet transform package `wtrans1d` without the `signal` package. You should always be sure the `signal` package is loaded. It is used by most of the other packages.

When you load a package, several things happen apart from the fact that the package is loaded. In order to allow you to associate some script files that will be automatically sourced when a package is loaded, LastWave does two things :

1) it adds to the source directory path the directory <`packageDir`>/<`packageName`>where <`packageDir`> is the initial script directory (i.e., the application directory referred to as `.` on macintosh computers and the shell variable `LWSOURCEDIR` on unix computers) and <`packageName`> the name of the package which is loaded,

2) if it exists in one of the source directories the file named <`packageName`>`.pkg` is sourced.

Thus, basically, when you load the package `signal` it will add the directory `scripts/signal` in the list of source directories and the file `signal.pkg` in it will be sourced. Let us note that if a package has been already loaded and is asked to be loaded again the corresponding file <`packageName`>`.pkg` will be sourced again.

8.5 The current object `objCur` - The prompt `&wtrans>a`

In many numerical packages, some new types are introduced. These types often correspond to complex structures which include many signals or images. This is the case for instance of the package `wtrans1d` which defines the type `&wtrans` corresponding to structure able to store 1d (continuous or (bi)orthogonal) wavelet transforms. Such a wavelet transform includes many signals (for both approximation signals and details signals). A lot of LastWave commands defined in these packages deal directly with these structures. In order to avoid having to specify each time the structure you are working on, LastWave lets you define a *current* structure named `objCur` which is the default structure many commands will work with. The terminal prompt tells you what the current object is at a given time. Actually, it gives you both its type, and the name of the variable associated to that structure. This is the reason why when you first started LastWave, it displayed the prompt `(&wtrans) a>`. Indeed, at startup the current object is the `&wtrans` variable named `a`. This type and this variable are defined in the numerical package `wtrans1`. It defines also the variable `b` associated to another such structure. The current object is stored in the variable `objCur`. Thus, for instance, if we assign the `objCur` variable to be the content of the variable `b` the prompt will automatically change :

```
(&wtrans) a> objCur=b
(&wtrans) b>
```

Let us note that this behavior of the prompt, is defined in script in the `startup` file using the *terminal prompt* command that lets you associate any script command to compute the prompt. Actually, the package `wtrans1d` defines the two commands `a` and `b` for setting the current object to one of them.

```
(&wtrans) a> b
(&wtrans) b> a
(&wtrans) a> m
(&book) m>
```

Let us note that `m` is a `&book` structure defined in the `mpp` package (this package defines the corresponding command `m` for setting the current object to `m`).

Remark : If you want to use the `objCur` variable (implicitely or explicitely) in a script command, don't forget to import it from the calling level (if it exists there) or from the global level. A very good example is given by the loop in the `scripts/wtrans1d/wtrans1d.pkg` file that allows to create the `a` and `b` wavelet transforms along with the `a` and `b` command.

Remark : As we will see in the next sections, another very important use of the current object system is that it lets you use the signals (or images) of the current structure in a very easy way. You should read Section 9.

8.6 The `copy`, `new` and `delete` commands

The `copy` command lets you copy a *value* which corresponds to a "large structure" (it does not work with `&num`, `&string`, `&image` and `&signal`). The syntax is either

copy <value>

which returns a copy of the <`value`> or

copy <value> <valueOut>

which copies the <`value`> into the content of <`valueOut`>. It is useful mainly for copying large structures. Thus for instance, to get a copy of the wavelet transform `a` you would type

```
a> e = [copy a]
= <&wtrans;0x211eb000>
```

If you want to copy the wavelet transform `a` into the wavelet transform `b` you would type

```
a> copy a b
= <&wtrans;b>
```

The `new` command lets you create a value of any type. The syntax is

*new *type**

and it returns the newly created value.

```
a> e = [new &wtrans]
= <&wtrans;0x211ec208>
```

Similarly, `delete` lets you delete a value, e.g.,

```
a> delete e
```

Chapter 9

Signals and ranges

9.1 Introduction

As we have already seen in Section (4.9), ranges are used to store uniform series of numbers. They can be increasing, e.g., `1.5, 4, 6.5, 9`, or decreasing, e.g., `3.1,1.1,-1.1,-3.1`. Generally a range will be defined by its first value, the step and its last value or eventually its size (i.e., the length of the series).
Signals are used to store a sampled function $y_n = f(x_n)$. The sampling could be uniform (in x) or not. In the first case, we will have to specify an origin x_0 for the abscissa and a sampling rate dx whereas in the second case we will have to specify an array $\{x_n\}$ of sampled abscissa. In both cases, we will have to specify the array $\{y_n\}$ of the sampled ordinates and the *size* of the signal (i.e., the number of points). The uniform sampling case will be referred to as being of type `ysig` (or `y-signal`) and the non uniform one to as being of type `xysig` (or `xy-signal`).
It is important to note that ranges can be seen as particular cases of y-signals (with $dx = 1$, $x_0 = 0$ and y_n being the nth value of the range). Whenever LastWave expects a signal, it can be replaced by a range. There is an implicit conversion from ranges to signals. However ranges are stored in a much more efficient way than signals. This is the reason why, LastWave tries to use ranges as much as possible and performs the conversion as late as possible.

Remark : Most of the commands involving signals, especially the commands that perform some signal processing numerical algorithms, will work only on `y-signals`. The `xy-signals` are supported mainly for display purposes.

Remark : If a command expects an argument to be a signal, this argument will be noted syntaxically <`signal`>. If the signal is supposed to be an "input signal", i.e., a signal that is not of size 0 and that will be used by the command as input, it will be denoted by <`signalIn`>.

9.2 The fields of `&range`

The fields of a `&range` are

- `first` : the first number of the series

- `last` : the last number of the series
- `step` : the increment, one has to add to go from a number to the next
- `size` : the number of points of the series

9.3 Constructors for `&range`

There are several ways to build ranges. Here is an exhaustive list

- <first>[!]:[<step>]:[!]<last> : builds the range with the first value being <first>, the last one being <last> and the step being <step> (or 1 if not specified). Let us note that if $< last >$ $<$ $< first >$ then <step> must be negative. If ! is placed right after <first> then the first value of the range is removed. If ! is placed right before <last> then the last value of the range is removed.
- <first>[!]:#<size>:[!]<last> : same as above except that instead of specifying the step, one specifies the size, i.e., the number of values in the range.
- I(<size>) : this is equivalent to 0:<size>

Let us note that, as we have already seen in Section 4.10, when ranges are used for extraction, in the first two cases, you can ommit $< first >$ if the it is equal to the first available index, $< last >$ if it is equal to the last available index. Moreover, the following notations are available

- @< : the first available index (generally 0)
- @> : the last available index
- @+ : the default step (generally 1, however in the next sections you will see cases where it is not equal to 1)

Finally, as we will see later, in a numerical expression involving signals and/or ranges, if the size of the resulting signal is clear, you can use the syntax I instead of I(<size>). Let's play around!

```
a> r = I(10)
= 0:9
a> r.first
= 0
a> r.size
= 10
a> r = 0:#100:!1
= 0:0.01:0.99
a> r*3+1
= 1:0.03:3.97
a> r.size = 10
= 10
a> r
```

```
= 0:0.01:0.09
a> s = 'abcdefghijk'
= 'abcdefghijk'
a> s[:]
= 'abcdefghijk'
a> s[!:]
= 'bcdefghijk'
a> s[!::!]
= 'bcdefghij'
a> s[5:@>-2]
= 'fghi'
```

9.4 The fields of &signal - The &signali type

The main fields of a `&signal` are

- `xy` : if 1 then the signal is an `xy-signal`, if 0 the signal is an `y-signal`
- `x0` (`y-signal` only): the first abscissa
- `dx` (`y-signal` only): the sampling step of the abscissa
- `size` : the number of points
- `name` : a (string) name for the signal
- `Y` : for managing the y_n values
- `X` : for managing the x_n values

Signals have other fields, if you want to learn what they are just type `help &signal`. You should know about the `firstp` field and the `lastp` field which keep tracks of eventual "border effects" during convolutions. they correspond to the first (resp. last) index which has not been affected by border effects.
The type of a signal is either `&signal` if it is an empty signal or `&signali` if not. However if you use the type `&signal` for an argument in a command definition, the argument will accept both empty and non empty signals whereas if you use `&signali` it will accept only non empty signals.

9.5 Constructors for &signal

For building `y-signals`, you can use the following syntax

- <<value1>,...,<valueN>> : this is the most common syntax. Each <value> can be either (i) a number, (ii) a signal, (iii) a range (since there is an implicit conversion from ranges toward signals), (iv) a listv made only of numbers, signals or ranges. All the numbers/signals/ranges/listvs are concatenated from left to right to form a `y-signal`. By default `x0 = 0` and `dx = 1`

- `Zero[(<size>)]` for building a signal made of 0's only (you do not need to specify the $< size >$, in some cases, if it is "clear" what the size is)
- `One[(<size>)]` for building a signal made of 1's only
- `Grand[(<size>)]` for building a signal made of independent Gaussian random numbers (mean 0 and variance 1)
- `Urand[(<size>)]` for building a signal made of independent uniform (between 0 and 1) random numbers
- `XY(`$< x - range >, < y - signal >$`)` : where $< x - range >$ is a range that will be used to determine `dx`, `x0` and `size` and $< y - signal >$ a signal (of the same size as $< x - range >$) made of the `y` values. Let us note that in the $< y - signal >$ expression, you can use the notation `X` to refer to the $< x - range >$.

For building `xy-signals`, you can use the following syntax

- `XY(`$< x - signal >, < y - signal >$`)` : where $< x - signal >$ and $< y - signal >$ must be signals of the same size. If you use this syntax, in the $< y - signal >$ you can use the notation `X` to refer to the $< x - signal >$.

Let's play around!

```
a> s= <>
= <size=0>
a> type(s)
= '&signal'
a> s = <2,0:-1:-3,1.5>
= <size=6;2,0,-1,-2,-3,1.5>
a> type(s)
= '&signali'
a> s+I(6)
= <size=6;2,1,1,1,1,6.5>
a> s+I
= <size=6;2,1,1,1,1,6.5>
```

In this last example we did not need to specify the size of the range `I` because the parser of LastWave goes from left to right, so when it reaches `I` it already knows that it will have to be added to `s` which is of size 6.

```
a> s.size = 3
= 3
a> s
= <size=3;2,0,-1>
```

When going from a size to a smaller size, the values are kept moreover no desallocation is performed. Thus, if going back to a larger size which is smaller or equal to the (ever) largest size, it will give you back the original values

```
a> s.size = 5
= 5
a> s
= <2,0,-1,-2,-3>
```

Let us note that you can force allocation/desallocation using the fields `sizeAllocY` and `sizeAllocY` (read help).

```
a> s.dx = 0.5
= <x0=0,dx=0.5;2,0,-1,-2,-3>
a> s.X
= 0:0.5:2
a> s.X = 5:#5:25
= <x0=5,dx=5;2,0,-1,-2,-3>
a> print s
s =
  5 2
  10 0
  15 -1
  20 -2
  25 -3
```

The `print` command prints the x-values (first column) and the corresponding y-values (second column). Let us note that the signal `s` is still a `y-signal` since the x-values are uniformly sampled. Let us switch to an `xy-signal`.

```
a> s.xy
= 0
a> s.X = <1,5,6,7,9>
= <size=5;(1/2),(5/0),(6/-1),...>
a> print s
s =
  1 2
  5 0
  6 -1
  7 -2
  9 -3
a> s.xy
= 1
a> info s
Type '&signali' [nref=2] [0x237f0830]
  XY Signal
  size :  5 [5x 6y]
  firstp :  0 (1)
  lastp :  4 (9)
```

Here the `info` command lets you know that you are dealing with an `xy-signal` of size 5 (the allocation of the x-array is (whereas the allocation of the y-array is 6). The first (resp.

last) index not affected by border effects is the first (resp. last) index which is the index 0 (resp. 4). It corresponds to a y-value of 1 (resp. 9). Let's create another `xy-signal` using the `XY` syntax

```
a> s = XY(<1,3,4>,8:10)
XY(<1,3,4>,8:10)
= <size=3;(1/8),(3/9),(4/10)>
a> s = XY(<1,3,4>,X*2)
= <size=3;(1/2),(3/6),(4/8)>
```

Let us give a last example before moving on. Let us build a y-signal which corresponds to a sine wave on $[0, 2\pi[$ (2π excluded) using 200 samples:

```
a> s = XY(0:#200:!2*pi,sin(X))
= <size=200,x0=0,dx=0.0314159;0,0.0314108,0.0627905,0.0941083,0.125333,0.156434,......>
```

9.6 Displaying signals

To display a signal < *signal* > you must use the `disp` command :

disp <signal>

The `disp` command is an extremely powerful command (defined in the `disp` package) for displaying any kind of objects (LastWave recognizes automatically the type of object to be displayed) and has many optional arguments, we refer the reader to the `disp` manual pages and to the section devoted to this command. It is very important that you read it. For now let us just note that the `disp` command does not copy the signals that are displayed. Thus if you change their values, their representation will change. However if the windows are not redrawn automatically. You should do it manually. There are several ways to do so. Either you send a `draw` message to the window (see Section 11.1), or you just type `disp <windowName>` (with no other arguments), it will redraw the window.

Unix users should be aware that there is a bug that do not let you close a window in the usual way using the mouse. To close a window, you should go with the mouse within this window and push the `f1` key

9.7 Operators on `&range` or `&signal`

All the operators that operate on numbers (see Section 4.4) do operate on signals and ranges. On ranges they are just applied to each number of the range. If the result is still a range, it returns a range, otherwise it returns a `y-signal`. On signals, these operators are applied to each y-value. The result is a signal of the same type as the original signal (y or xy-signal). Thus, for instance, on ranges

```
a> r = 1:10
= 1:10
a> r=r+3
= 4:13
```

```
a> r=r/2
= 2:.5:6.5
a> int(r)
= <size=10;2,2,3,3,4,4,...>
```

and on signals

```
a> s = exp(0:#5:1)
= <1,1.28403,1.64872,2.117,2.71828>
a> s = ln(s)
= <0,0.25,0.5,0.75,1>
a> s *= 2
= <0,0.5,1,1.5,2>
a> s = XY(<1,2,4>,<-10,8,1>)
= <size=3;(1/-10),(2/8),(4/1)>
a> s>=0
= <size=3;(1/0),(2/1),(4/1)>
```

If s is y or an xy-signal (with non zero x-values), and if you want to display the same signal in a (natural) logarithm scale in x, you could do

```
a> s = XY(0!:#200:2*pi,sin(X))
= <size=200,x0=0.0314159,dx=0.0314159;0.0314108,0.0627905,0.0941083,0.125333,0.156434,0
a> disp s
a> disp XY(ln(s.X),s.Y)
```

Let us note that `disp XY(ln(s.X),s)` would work the same way.

Warning: If you apply to the x-values a non increasing map, the values of the resulting xy-signal will be permuted so that the x-values are sorted. **The x-values of an xy-signal are always sorted**.

There are a few operators that operates only on signals (or ranges) and not on numbers. They are

- `min, max` : as operators on a single signal it returns the minimum and maximum values (operating on two signals it returns the signal made of the returned values when the oerator is applied to each successive couple of values)
- `der, prim` : builds the derivative and the primitive signals
- `sum` : returns the sum of all the y-values
- `mean` : returns the mean of all the y-values
- `any` : returns 1 if any of the y-values is non zero otherwise returns 0
- `all` : returns 1 if all the y-values are non zero otherwise returns 1
- `find` : returns a listv of all the indices which corresponds to non 0 y-values
- `~s` : transposition of the *horizontal* signal `s` into a *vertical* matrix, i.e.image (see Section 10)

For some examples of `any, all` and `find` you can read the end of Section 6.2.

9.8 Extraction mechanism

Before you start reading this section, you should be sure to have read Section 4.10.
Let us recall "standard use" of extraction as explained in Section 4.10.

```
a> s = <1,3,4,7,10,20>
s[1]=<12,13>
** Error :  Size of both handsides should match (left size = 1, right size = 2)
- - - > s[1]=<12,13>
a> s[1,3]=<12,13>
= <size=6;1,12,4,13,10,20>
a> s[1:2,5]={<0,1> <2,2>}
= <size=7;1,0,1,13,10,2,...>
a> print s
s =
  0 1
  1 0
  2 1
  3 13
  4 10
  5 2
  6 2
a> s[1:2,3:4]:={<1>}
= <1,1,1,2,2>
a> s[1:]+=I(4)
= <1,1,2,4,5>
a> s[2:3]:={<>}
= <1,1,5>
a> s[0,2]:=<10,20>
= <10,20,1,10,20>
```

Let us recall that, when assigned, a signal is *not* copied. Indeed

```
a> s1 = s
= <10,20,1,10,20>
a> s[0] = 0
= <0,20,1,10,20>
a> s1
= <0,20,1,10,20>
```

In order to perform a copy, you can use the `copy` command (see Section 8.6)

```
a> s1 = [copy s]
= <0,20,1,10,20>
```

If $s1$ is a signal that already exists, you can perform a copy of `s` into `s1` in the following way (see Section 9)

```
a> s1[] = s
= <0,20,1,10,20>
```

To create a signal `s1` which is made of the signal `s` repeated 3 times you can do

```
a> s = <1,2>
= <1,2>
a> s1 = <s,s,s>
= <size=6;1,2,1,2,1,2>
```

or, using listv's

```
a> s1 = <{s}*3>
= <size=6;1,2,1,2,1,2>
```

or, using periodic border effects along with extraction

```
a> s1=s[*bper,0:5]
= <size=6;1,2,1,2,1,2>
```

Let us note that if `s` was an x-signal, the x-values would be lost after such an operation.

In all the above examples, the extraction forms specify the indices they should work on. There is a way in LastWave to perform extractions specifying real x-values and not indices. This is done through *options. there are 2 such options

- `*x` : the ranges in between `[...]` refer to x-values and not indices. If interpolation is needed, the signal is interpolated using piece-wise constant functions. The piece-wise constant function is constant in between each sample points and is right continuous.
- `*xlin` : same as `*x` except that the signal is interpolated using piece-wise linear functions.

(Let us recall that `*options` can be used only for getting values and not for assignement.) Let us see how it works:

```
a> s = XY(0:#5:1,2*X)
= <x0=0,dx=0.25;0,0.5,1,1.5,2>
a> s[1]
= 0.5
a> s[*x,1]
= 2
a> s[*x,0.7]
= 1
a> s[*xlin,0.7]
= 1.4
```

You can use ranges too. There are 2 cases: (i) in the case you do not specify the `step` in the range then all the points corresponding to an x-value in between the first and last points of the range are kept and the other ones are removed (ii) in the case the `step` is specified, the result is a $y - signal$ and interpolation is made for each range value. Thus

```
a> s = XY(<0.5,2,2.5,10>,2*X)
= <size=4;(0.5/1),(2/4),(2.5/5),...>
a> s[*x,1:10]
= <size=3;(2/4),(2.5/5),(10/20)>
a> s[*xlin,1:1:10]
= <size=10,x0=1,dx=1;2,4,6,8,10,12,...>
```

The next section gives some advanced examples using all you have learned in this section.

9.9 A few more examples

First example Let us first we build a sine wave on $]0, 2\pi]$ sampled with 200 points

```
a> s = XY(0!:#200:2*pi,sin(X))
= <size=200,x0=0.0314159,dx=0.0314159;0.0314108,0.0627905,0.0941083,0.125333,0.156434,0
disp s
```

Knowing the signal is periodic, extracting the part in the interval $[4\pi, 6\pi]$

```
a> s1=s[*x*bper,4*pi:6*pi]
= <size=200,x0=12.5664,dx=0.0314159;1.74846e-07,0.0314108,0.0627905,0.0941083,0.125333,0
disp s
```

or extracting the part corresponding to the initial interval$+\pi$

```
a> s1=s[*x*bper,@<+pi:@>+pi]
= <size=200,x0=3.17301,dx=0.0314159;-0.0314109,-0.0627906,-0.0941084,-0.125333,-0.15643
disp s
```

Second example Let us build a piece-wise constant uniformely sampled function. For that purpose we first need to specify the *break points*

```
a> s = XY(<0,2,2,3,5>,<0,0,1,-1,2>)
= <size=5;(0/0),(2/0),(2/1),...>
a> disp s
```

Let us note that `s` has a discontinuity at $x = 2$. Now we want to uniformely sample it using a total of 200 points

```
a> s1=s[*x,:#200:]
= <size=200,x0=0,dx=0.0251256;0,0,0,0,0,0,0,...>
a> disp s1
```

As you see, to each break point corresponds a step except for the last point. The last point of the signal is the only one corresponding to a y-value of 2. Indeed, as we have already explained, each "step" starts from a break point to the next. If you want this last point to be removed you can just type

```
a> s1=s[*x,:#200:!]
= <size=200,x0=0,dx=0.025;0,0,0,0,0,0,0,...>
a> disp s1
```

Or if you want a whole step (of size 1) associated to this point, you could do

```
a> s1=s[*x*bcon,:#200:!6]
= <size=200,x0=0,dx=0.03;0,0,0,0,0,0,0,...>
a> disp s1
```

Third example Let us build a piece-wise linear uniformely sampled function using the same break points as the example just above.

```
a> s = XY(<0,2,2,3,5>,<0,0,1,-1,2>)
= <size=5;(0/0),(2/0),(2/1),...>
a> s1=s[*xlin,:#200:]
= <size=200,x0=0,dx=0.0251256;0,0,0,0,0,0,...>
a> disp s
```

Fourth example Let us build a piece-wise constant uniformely sampled function from a y-signal. Let $s(t)$ be the periodic function (of period 4) which is piece-wise constant on each interval $[0,1[$, $[1,2[$, $[2,3[$, $[3,4[$ and for which $s(0) = 1$, $s(1) = 5$, $s(2) = -1$, and $s(3) = 0$. We want to generate a sample of this function (on 200 points) on the interval $[10,22[$!

```
a> s = <1,5,-1,0>
= <1,5,-1,0>
a> s1 = s[*x*bper,10:#200:!22]
= <size=200,x0=10,dx=0.06;-1,-1,-1,-1,-1,-1,...>
disp s1
```

Other examples You should read Section 9.11.2 which gives you some good examples of signal manipulations.

9.10 Using the current object objCur for storing signals

At startup, the current object (see Section 8.5) is the wavelet transform structure `a` (of type `&wtrans`). As you will learn in the `wtrans1d` package manual, a `&wtrans` structure includes many signals. They are grouped in two arrays `D` (the detail signals) and `A` (the approximation signals). Thus `a.A[n]` corresponds to the `n`th approximation signal and `a.D[n]` to the `n`th detail signal. Actually the approximation (resp. detail) signals can be addressed using the shorter syntax `n`a (resp. `.n`a). Thus `a.A[10]` (resp. `a.D[10]` corresponds to the same signal as `10a` (resp. `.10a`). Moreover, if `a` is the current object you don't even need the `a` after the number! Thus `10` (resp. `.10`) refers to the signal `a.A[10]` (resp. `a.D[10]`. Thus for instance

```
a> 0=Grand(256)
= <size=256;2.27747,0.450497,1.21345,1.07444,1.07706,-0.767054,...>
a> 0.size
= 256
a> 0[1]
= 0.450497
a> b
b> 0=Grand(200)
= <size=200;1.35275,-2.26267,-2.09973,-0.171687,1.19217,-0.300141,...>
b> disp 0
b> 0b
= <size=200;1.35275,-2.26267,-2.09973,-0.171687,1.19217,-0.300141,...>
```

```
b> 0a
= <size=256;2.27747,0.450497,1.21345,1.07444,1.07706,-0.767054,...>
b> disp 0 0a
```

Thus when you are working with signals, instead of creating new signals, you can use these one of `a` and `b`. It is very useful to have signals so easily accessed.

Warning If you type just `0` on the terminal line, LastWave understands the number `0` and not the signal `0b`. This is why we had to type in `0b` in the last example (though `b` was the current object)

Warning In a numerical expression you have to use the `0b` notation too (even if `b` is the current object) since a `0` will again be interpreted as the number `0`. This is not the case if you type `disp 0`. Indeed, LastWave knows that the `disp` command must have `&valobj` arguments (see Section 7.4), i.e., anything but not a number. So it understands that `0` means the signal `0` of the current object.

Remark Let us notet that you could also use the `&book` structures `m` and `n` defined in the `mpp` package.

9.11 Some useful commands

9.11.1 The `read`/`write` commands

These commands let you read/write signals on the disc. You can write/read both ascii or binary files. Let us note that the `read` command lets you read multiple columns formatted ascii files.

9.11.2 The `fft` command

The `fft` command lets you compute the Fourier transform of a signal. If the signal you want to compute the Fourier transform of is real, the syntax is the following

fft <signalIn> <signalOutReal> <signalOutImag>

Where <`signalIn`> has a size which must be a power of 2. Thus for instance if s is the sum of gaussian noise (of variance .1) and of two harmonics at frequency 200Hz and 120Hz,

```
a> s = XY(0:#4096:1,sin(X*200*2*pi)+sin(X*120*2*pi)+Grand*sqrt(.1))
= <size=4096,x0=0,dx=0.0002442;0.279043,0.468406,0.869217,1.04903,1.83248,1.68536,...>
a> fft s 1 2
= 0
```

The `0` returned by the command `fft` is the approximate time it took (in seconds)... Let us note that here the result of the Fourier transform has been put in the signals `1a` and `2a` of the current object `a` (which is a `&wtrans` object). If you want to display the amplitude of the Fourier transform between frequencies 0 and 300Hz, just type

```
a> disp sqrt(1a^2+2a^2) -x 0 300
```

you do see clearly the two peaks around the right frequencies. Let us check it directly,

```
a> u = 1a^2+2a^2
= <size=2049,x0=0,dx=0.999756;760.484,648.305,271.585,1290.87,379.507,337.678,...>
a> u.X[find(u>max(u)/2)]
= <119.971,199.951>
```

This is a nice example, where you see the power of LastWave for working with real coordinates.

WARNING : Though the `dx` field of the input-signal is taken into account in the computation of the fft, the `x0` field is *not* taken into account.

Let us note that, if the input signal is real, the Fourier transform is computed only on positive frequencies (and has one more point that the initial signal). The `-i` option lets you the inverse operation. The syntax is (for a real result signal)

fft <fftReal> <fftImag> <signalResult>

Let us take out the 120Hz frequency peak and get the new signal

```
a> l = find(u>max(u)/2) = <120,200>
a> 1[120] = 0
= <size=2049,x0=0,dx=0.999756;27.5769,3.56154,-10.672,-32.8814,6.5794,-18.2641,...>
a> 2[120] = 0 = <size=2049,x0=0,dx=0.999756;0,-25.2115,12.5576,14.4805,-18.3363,-2.0250
a> fft 1 2 3 -i
= 0
```

the result is in the signal `3a`.

If the original signal was a complex signal, the syntax would have been

fft <signalInReal> <signalInImag> <signalOutReal> <signalOutImag>

The abscissa of the output signals would include negative frequency (the option `-s` unshift these negative frequencies).

9.11.3 The `convol` command - The `firstp, lastp` fields

The `convol` command is a high level command that lets you compute the convolution of a signal with a compact support filter. You can specify any border effect for the input signal (one of `'bperiodic', 'b0', 'bconst', 'bmirror', 'bmirror1'`) and you can specify on which abscissa interval you want the result to be computed (both the `dx`, and `x0` fields of both the filter AND the signal are taken into account). Moreover you can choose in between two methods : a direct method and a method using the fft algorithm.

Let us note that the `firstp, lastp` fields of the resulting signal are used to store the first and the last indices which are not affected by border effects.

Chapter 10

Images and basic linear algebra

You should read Section 9 before reading this section.

10.1 Introduction

LastWave is able to deal with images. Images are pretty simple structures (not as complex as signals) made basically of one 2d array of floats of diimension `nrow` × `ncol`. The exact same structure is also used to represent matrices. For linear algebra, the y-values of a signal are considered as a 1d *horizontal* vector, i.e., of dimension `1` × `size`, where *size* is the size of the signal. The transposition of a signal of size *size* thus gives a *vertical* vector of dimension `size` × `1`. This vector will be stored by LastWave as a matrix, i.e., as an image.

10.2 Fields of an image `&image` - The `&imagei` type

The fields of an image (or of a matrix) are

- `ncol` : the number of columns of the image
- `nrow` : the number of rows of the image
- `name` : a (string) name associated to the image

As for signals (i.e, `&signal` or `&signali`), the type of an image is either `&image` if it is an empty image or `&imagei` if not. However if you use the type `&image` for an argument in a command definition, the argument will accept both empty and non empty images whereas if you use `&imagei` it will accept only non empty signals. Let us note that you can use the "pseudo" (read-only) field `tosignal` to convert an image to a signal:

- `tosignal` (read-only): returns the image converted into a signal by concatenating the rows

Thus for instance, if you want to make histograms of the pixel values of an image, you can use the `histo` command for signals and apply it to the converted image.

10.3 Constructors for &image

For building images/matrices, you can use the following constructors

- <<value11>,...,<value1M>;<*value*21>,...,<*value*2*M*>;...;<valueN1>,...,<valueNM>> : this is the most common syntax. Each <value> can be either (i) a number, (ii) a signal, (iii) a range (since there is an implicit conversion from ranges toward signals), (iv) an image or (v) a listv made only of numbers, signals, ranges or images. All the numbers/signals/ranges/images/listvs are concatenated from left to right and from top to bottom, to form an image.
- Zero[(<ncol>,<nrow>)] for building an image made of 0's only (you do not need to specify the dimension, in some cases, if it is "clear" what the diemnsion is)
- I[(<ncol>,<nrow>)] for building an image of coefficient $a_{j,i} = i$
- J[(<ncol>,<nrow>)] for building an image of coefficient $a_{j,i} = j$
- One[(<ncol>,<nrow>)] for building an image made of 1's only
- Id[(<ncol>,<nrow>)] for building the identity matrix
- Grand[(<ncol>,<nrow>)] for building a image made of independent Gaussian random numbers (mean 0 and variance 1)
- Urand[(<ncol>,<nrow>)] for building a image made of independent uniform (between 0 and 1) random numbers

Let's play around!

```
a> j = <;>
= <nrow=0;ncol=0>
a> type(j)
= '&image'
a> j=<7,8;9,10>
= <nrow=2;ncol=2>
7 8
9 10
a> type(j)
= '&imagei'
a> i = <1,2;3,4;5:6;j>
= <nrow=5;ncol=2>
1 2
3 4
5 6
7 8
9 10
a> i+One
= <nrow=5;ncol=2>
2 3
```

```
4 5
6 7
8 9
10 11
```

In this last example we did not need to specify the dimension of the matrix `One` because the parser of LastWave goes from left to right, so when it reaches `One` it already knows that it will have to be added to `i` whose dimension is of `6 × 2`.

```
a> i=J(3,3)
= <nrow=3;ncol=3>
0 0 0
1 1 1
2 2 2
a> i+Id
= <nrow=3;ncol=3>
1 0 0
1 2 1
2 2 3
a> i.nrow
= 3
a> i.ncol
= 3
a> <i,i;i,i>
= <nrow=6;ncol=6>
0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
```

10.4 Displaying images

The generic `disp` procedure that we have seen for displaying signals (Section 9.6) works for any kind of objects including images. Thus, to display an image $< image >$ you must use the `disp` command :

disp $< image >$

Let us note that in order to display the right colors, you must define a colormap and either set it to the current colormap before displaying the image or set the colormap of the displayed image afterwards. In any case, if you want to display a 256 grey level image, the colormap is already build and is called `'grey'`. To make it the current colormap before displaying the image you just need to type

```
colormap current 'grey'
```

Indeed, the default current colormap is a "rainbow" colormap named `'color'`

Unix users should be aware that there is a bug that do not let you close a window in the usual way using the mouse. To close a window, you should go with the mouse within this window and push the `f1` key

10.5 Operators on `&image`

All the operators that operate on numbers (see Section 4.4) do operate on images/matrices. As for signals and ranges (see Section 9.7), they are just applied to each number of the image.

There are a few operators that operates only on images not on numbers. They are

- `min, max` : as operators on a single image it returns the minimum and maximum values (operating on two images it returns the image made of the returned values when the oerator is applied to each successive couple of values)
- `M**N` : matrix `M` multiplied by matrix `N`
- `~M` : transposition of matrix `M`
- `M^^n` : computes M^n where `M` is a square matrix and `n` a positive or negative integer
- `any` : returns 1 if any of values is non zero otherwise returns 0
- `all` : returns 1 if all the values are non zero otherwise returns 1
- `find` : returns a `2×N` image made of index couples corresponding to non 0 values

Thus

```
a> i=J(2,2)
= <nrow=2;ncol=2>
0 0
1 1
a> j=(i+Id)m1
= <nrow=2;ncol=2>
1 0
-0.5 0.5
a> j**(i+Id)
= <nrow=2;ncol=2>
1 0
0 1
a> i^^(-1)
** Error :  ludcmp():  Singular matrix
---> i^^(-1)
a> s = <2,3>
= <2,3>
a> ~s
```

```
= <nrow=2;ncol=1>
2
3
a> i**~s
= <nrow=2;ncol=1>
0
5
```

The operators `all` and `any` work the same way as when operating on signals. We refer the reader to the end of Section 6.2 for examples. Examples of the use of the operator `find` will be given in the next section.

10.6 Extraction mechanism

Extraction works in two different ways on images/matrices depending on what you want to do.

10.6.1 Extraction of isolated values

In order to extract `n` values given their coordinates, the syntax is the following

image[<image1>]

where <`image1`> is an image of dimension $2 \times n$. Each column of the <`image1`> gives the coordinate `row` $\times$ `col` of a value that will be extracted. The result of such an extraction is a signal with all the values extracted. Let us note that you can use the following `*options`

- `*nolimit` : indices out of range are not taken into account,
- `*bperiodic` : periodic border effects (both horizontal and vertical directions)
- `*bmirror1` : mirror (with repetition of the last points) border effects
- `*bmirror` : mirror (without repetition of the last points) border effects
- `*bconst` : constant border effects
- `*b0` : 0 border effects

For assignement the syntax is

image[<image1>] [:]= < value >

where <`value`> is either a signal/range of size n or a float (in the latter case, you need to use the operator `:=` instead of `=`). Thus, for instance

```
a> j = J(4,4)+I = <nrow=4;ncol=4> 0 1 2 3 1 2 3 4 2 3 4 5 3 4 5 6
a> j[<0,1,2,3;0,1,2,3>]
= <0,2,4,6>
a> u = <0,1,2,3;0,1,2,3>
= <nrow=2;ncol=4>
```

```
0 1 2 3
0 1 2 3
a> j[u]
= <0,2,4,6>
a> j[u]+=1
= <nrow=4;ncol=4>
1 1 2 3
1 3 3 4
2 3 5 5
3 4 5 7
a> j[u]=10:10:40
= <nrow=4;ncol=4>
10 1 2 3
1 20 3 4
2 3 30 5
3 4 5 40
```

The `find` operator lets you access some values that are the result of a test, e.g.

```
a> find(j==3)
= <nrow=2;ncol=4>
0 1 2 3
3 2 1 0
a> j[find(j==3)] := 0
= <nrow=4;ncol=4>
10 1 2 0
1 20 0 4
2 0 30 5
0 4 5 40
```

10.6.2 Extraction of sub-images

In order to extract an $n \times m$ sub-matrix, the syntax is the following

image[<value11>,...,<value1n>;<value21>,...,<value2m>]

where all the values are either signals, ranges, integers or listv of integers that describe indices. The first list of values <`value1k`> indicates which rows must be kept and the second list of values <`value2k`> indicates which columns must be kept. In ranges, you can ommit the first or the last point if you want to go down the first index or the last one. Thus in order to extract the third row of a matrix, you should do

```
a> j = J(4,3)+I
= <nrow=4;ncol=3>
0 1 2
1 2 3
2 3 4
3 4 5
a> j[2;:]
= <2,3,4>
```

Here are some more manipulations

```
a> j[2:;1:]
= <nrow=2;ncol=2>
3 4
4 5
a> j[*bper,2:7;1:]
= <nrow=6;ncol=2>
3 4
4 5
1 2
2 3
3 4
4 5
```

As for the other type of extraction, you can use the `*options`: `*nolimit,*bperiodic,*bmirror1,*bmirror *b0, *bconst`.

For assignement, the syntax is

image[<value11>,...,<value1n>;<value21>,...,<value2m>] [:]= <rvalue>

where <`rvalue`> is either an image of size $n \times m$ (or a range/signal if n is equal to 1) or a float (in which case you must use the syntax `:=` instead of `=`). Thus

```
a> j = J(4,3)+I
= <nrow=4;ncol=3>
0 1 2
1 2 3
2 3 4
3 4 5
a> j[2;:]:=0
= <nrow=4;ncol=3>
0 1 2
1 2 3
0 0 0
3 4 5
a> j[2;:1]=<10,10>
= <nrow=4;ncol=3>
0 1 2
1 2 3
10 10 0
3 4 5
```

Remark Let us note that, as for signals or listv, when assigned, an image is *not* copied. In order to perform a copy, you can use the `copy` command (see Section 8.6) Moreover if `i` and `i1` are two different images you can perform a copy of `i` into `i1` using the syntax `i1[]=i`

10.7 Using the current object `objCur` for storing images

We have seen (Section 9.10) that one could store signals in `&wtrans` structures (defined in the `wtrans1d` package). it allowed to refer to them in a very easy way. You could do the same thing for images using `&owtrans2` structures (defined in the `owtrans2d` package). Two such structures are defined at startup, one is `a2` and the other one is `b2`. You can then use the images n`a2` or n`b2`. Thus for instance

```
a> a2
a2> 0=J(2,2)
= <nrow=2;ncol=2>
0 0
1 1
a2> 0[1;1]
= 1 0.ncol
a2>
= 2
```

Let us note that you could also use the `&dwtrans2` structures `a2d` and `b2d` defined in the `dwtrans2d` package.

Part III

Managing Graphics

Chapter 11

Introducing Graphics

LastWave includes a very powerful object-oriented graphic language. It is time to learn about it.

11.1 Hello World

The `window` command lets you deal with windows. Thus to create a window named `Hello`, you should just type

```
a> window new 'Hello'
```

The window does not appear yet on the screen. This is because when a window is created it is hidden. In order to show it you must send it the `show` message. To send a message to a graphic object (often referred to as a `gobject`) one must use the following "object-oriented syntax" :

```
msge *gobjectName* *message* [<arg1>...<argN>]
```

where *gobjectName* is the name of the graphic object you send the message *message* to along with (if required by the message) the arguments <`arg1`>...<`argN`>. Thus to show the window, we must send the `show` message :

```
a> msge Hello show
= 1
```

and the window appears... The answer 1 means that the window understood the message and executed it. No answer would mean that the window dos not know about this message. Let us note that no error would be generated (this can be very useful when you want to send a message that only a few gobects know about, you can just send it to all the gobjects).

WARNING : Right now, there is a bug in the Unix/X11 : Do NOT close the windows using the menu on the window, it will core dump ! You must either send the `delete` message to the window you want to delete or just hit the `f1` key in that window (this behavior is a `binding` defined in the script file `scripts/mist/gobject`).

Let us note that the "current" window (i.e, the last window used in a command or the last window visited by the mouse) can be called directly using the shortcut `.` instead of its name. Thus to hide the window `Hello` again and show it again you could type

```
a> msge .  hide
= 1
a> msge .  show
= 1
```

There are two ways of addressing graphic objects. The first one is to send messages to the object using the syntax we have just seen and the second one is to change its fields. Each object is defined by a list of fields whose values you can change. For instance any graphic object has a field called `-bg` which corresponds to its background color (all the fields start with a `-` character). To get or set the value of a field of a graphic object, one must use the `setg` command. Thus to get the background color of the window, one must type

```
a> setg .  -bg
= 'white'
```

The background color of the window is white (haven't you noticed?). Now if you want to change it to red, you must type

```
a> setg .  -bg 'red'
```

What happened? Nothing! How come the window has not changed color? Well this is because changing fields of a graphic object is not as sending a message to this gobject. Sending a field just corresponds to changing a value characterizing the way the gobject will appear on the screen. However you did not ask the object to redraw itself. For that purpose, you could send a `draw` message

```
a> msge .  draw
= 1
```

It worked! The window appears now all red. You might think its a little long to just change the background color. You are right. That's why there is a command `setgu` which is just like `setg` but after changing the fields it updates the drawing of the gobject. Thus, just try

```
a> setgu .  -bg 'blue'
```

and the window should be all blue. Both `setg` and `setgu` let you set as many fields as you want at the same time. For instance, if you want to change the title of the window as well as its size and its backgound color, you can type

```
a> setgu .  -bg 'green' -size 300 300 -title 'New Title'
```

and the window should change color, change size and have a new title.

Remark : Let us note that the name of a window does not automatically correspond to its title. If no title is specified then the title is the name of the window. But if the title is explicitly changed, the name remains the same. Thus the name of the window curently displayed on the screen is still `Hello`. This is the name you will always refer the window with (unless it is the current window in which case you can use `.`).

Now let us suppose we want to display in this window the string `'Hello World'`. A first way to do it is to just draw the string at a given location using the `draw` command along with the `string` action :

```
a> draw string .  50 50 "Hello World"
```

The `50 50` corresponds to the x,y pixel coordinates the string should be drawn at. The origin of a window is the top-left corner.

Remark : Let us note that the `draw` command is a very useful command that allows to draw strings as well as rectangles, ellipses,... and many other shapes. You will learn more about it later.

Now, if we hide the window and show it again the string will not be drawn again. This is because, we have "just" drawn the string, we have not told the window to remember that there is a string drawn at the location `50 50`. To do so, one must add a graphic object in the window which purpose is to display a string. Such a graphic object has been defined in the script file `scripts/misc/box`. This file is sourced whenever the `misc` package is loaded. If you haven't changed the orginal `startup` file, it should be loaded automatically at startup.

You should not try to understand the `scripts/misc/box` file right now, you do not know enough about graphics to understand it. For now, you just need to know that it creates a new class of graphic objects named `Box` which allow to display a rectangle box with text in it. Let us add a `Box` object at position 50 50 (of width 80 and of height 20) of the window `Hello` and display in it the string `'Hello World'`. We will name this `Box` `'box'` :

```
a> msge Hello add 'box' Box -pos 50 50 -size 80 20 -string "Hello World"
= 1
```

and the string will appear. Now you can hide and show again the window, the string will be automatically redrawn. The window `Hello` contains a gobject named `box` which is of class `Box`. Thus, to add a graphic object to a window we used the message `add`. The first argument is the name of the newly created object, the second one is its class (e.g., `Box`) and then there is a list of fields along with fields values the object will be set with (e.g., the field `-string` allows to control the string displayed in a `Box`).

Remark : Let us note that we did not need to send a `show` mesage to `box` for it to appear. Only windows are created hidden. All the other gobjects when created are automatically shown unless the field `-hide` is set to 1 (i.e, adding `-hide 1` to the initialization string).

Now if we want to adress the `box`, we need to specify that it is the graphic object named `box` in the window `Hello`. To do so we address the box in the following way : `Hello.box`. Be careful it has nothing to do with arrays. It just uses the same notation. It means the graphic object `box` in the window `Hello`. Thus, to change the string displayed in the box one can type

```
a> setgu Hello.box -string "New string"
```

Since `Hello` is the current window we could just adress the box by `.box`. It will work the same way. Let us note that with a single command line you can change fields of the window and of any object in it :

```
a> setgu Hello -bg 'red' -.box -string 'Again' -bg 'blue'
```

Since `Hello` is used as first argument, the first specified fields are the fields of the window `Hello`. Thus `-bg 'red'` indicates that we want the background of the window to be red. Now, `-.box` means that all the following fields no longer concern the gobject specified in the `setgu` command (i.e., the `Hello` window) but the graphic object that is contained in `Hello` and that is named `box`. Thus `-string 'Again' -bg 'blue'` means that the string displayed in the `box` will the the string `'Again'` and that the background color of the `box` will be blue.

Let us add another box in the same window

```
a> msge Hello add 'box1' Box -pos 50 100 -size 80 20 -string "Hello World 1"
= 1
```

Thus there are now 2 graphic objects in the window `Hello`. The box `box` and the box `box1`. They are both graphic objects which can be seen as instances of the same `Box` graphic class. In a single line you can change the fields of all of them

```
a> setgu Hello -bg 'white' -.box -string 'hello' -bg 'green' -.box1 -bg 'green'
```

Thus we can combine on the same line as many `-.<gobject>` as we want: all the fields up to the next `-.<gobject1>` will be considered as fields of the gobject `<gobject>`. Moreover, whenever you send messages using the `msge` command or you set fields using the `setg` or `setgu` commands, you can use wild cards characters in the graphic object names. Thus if you want to change the background of all the graphic objects included in the window `Hello` to blue, you would type indifferently

```
a> setgu Hello.* -bg 'blue'
```

or

```
a> setgu .* -bg 'blue'
```

or

```
a> setgu Hello -.* -bg 'blue'
```

11.2 Playing around with the mouse

11.2.1 Moving graphic objects

In order to move a given graphic object, you can of course send to it the `move` message, using the *msge* command as explained above. However, there is a much shorter way to do it interactively with the mouse. This is very useful, for instance, for moving the labels or the title of a figure before printing it. For that purpose, you just need to use the middle button along with the modifier keys `shift` and `ctrl` and just drag and drop the graphic object wherever you want!

Let us note that this mouse behavior is defined using the script in the file `scripts/misc/gobject` (which is part of the `misc` package which is automatically loaded by the original startup file) . In order to understand that script file , you first need to read about managing the graphic events.

11.2.2 Getting help about a graphic object

In order to get the value of a field of a given graphic object, you can of course use the `setg` command along with the field name as explained above. However if you want to get the whole list of all the field values, you can do it very easily using the mouse. For that purpose, you just need to point to the object with the mouse and then hit the `h` key. A new window will appear with all the field values classified according to the class hierarchy. You will learn very soon about the graphic class hierarchy. For now you just need to know that the basic graphic class is called `GObject`. All the other classes derive from that class. When the fields are displayed, first the `GObject` fields will be listed then the `Box` fields (if the graphic object you clicked in is a `Box`) will be displayed.

If you hit a second time on the key `h` you will get a list of available messages you can send to the object.

If you hit a third time on the key `h` you will get all the available bindings (i.e., mouse or keyboard interactions) that can be performed on the corresponding graphic object. They are classified again using the graphic class hierarchy (starts from `GObject` bindings...). For instance you will always see appearing at the top of this popup box, the mouse interactions (that work on any `GObject`) we have described up to now (moving objects + getting some help). Let us note that you could go directly to that help hitting the `b` key instead of the `b` key.

Hitting a fourth time the `h` key will make the help window disappear.

11.2.3 Deleting windows and other graphic objects

In the same script file (`scripts/misc/gobject`) are defined two bindings which allow to delete a window or a graphic object. You just need to move the mouse inside the graphic object you want to delete, then if you press the `f1` key, the window the graphic object is in will be deleted and if you press the `f2` key only the graphic object you are pointing to will be deleted.

11.3 Using `GList`'s

As `Box`'s, `Window`'s correspond to a graphic class. However, windows have two particularities. The first one is that they can include other objects and the second one is that they cannot be included into any other objects. The `GList` (graphic object lists) class corresponds to graphic objects that can include other graphic objects. Thus, windows are a particular case of `Glist`'s. Using an object-oriented terminology, we will say that the `Window` class inherits from the `Glist` class. It means that all the messages and fields that a `GList` understands are also understood by a `Window`. In other words, a `Window` can be seen as a *GList* whith some particularities (one of them, for instance, being that, contrary to a simple `GList`, it cannot be included in another GList).

Thus `GList`'s can be used to group some graphic objects so that they will share the same origin which is the top left corner of the `GList`. Let's look at an example. Let's create a window with a `Glist` in it and two boxes in it.

```
a> window new 'Hello' -size 300 300 -hide 0
```

This last command allows to create a window of size 400 400 and show it right away (this is because of the field `-hide` which is set to 0). Let's add a `GList`

```
a> msge Hello add 'glist' GList -pos 10 10 -size 200 200 -frame 1
= 1
```

The `-frame` field is a field of the `GObject` class, thus it is understood by any graphic object, it allows to frame the graphic object with a rectangle. Let's now add the two boxes:

```
a> msge Hello.glist add 'box' Box -pos 20 20 -size 80 20 \
--> -string 'Hello World' -bg 'grey'
= 1
a> msge Hello.glist add 'box1' Box -pos 50 100 -size 80 20 \
--> -string 'Hello World 1' -bg 'lightgrey'
= 1
```

Let us first note that the message is sent to the `glist` and not the window since we want to add the boxes to the `glist`. Moreover, the positions of the boxes are specified in local coordinates, i.e., the 0,0 point corresponds to the left-top corner of the `glist`. If you want to get or set the global position of the boxes, you can use the `-apos` (absolute position) field:

```
a> setg Hello.glist.box -apos
= {30 30}
```

which is the sum of the glist origin 10 10 and the box origin 20 20. Let us note that to address the box, you have to specify the full path : `Hello.glist.box` (the `box` object is in `glist` which is the window `Hello`) . Since `Hello` is the current window you could address the box just by the path `.glist.box`. Actually, if you do not want to specify the fact that the `box` is in `glist` you could address the box by `Hello..box`, which means that the `box` is in the window `Hello` but you don't specify the complete path, you just use two dots `..` to ask LastWave to look for it in any `GList` (recursively). Since `Hello` is the current window you could simply use `..box` which is much shorter than the former `Hello.glist.box`. Thus if you want to change the background color of `box`, you would type :

```
a> setgu ..box -bg 'red'
```

If you want to address both boxes, you could use the character `*` :

```
a> setgu ..box* -bg 'blue'
```

Remark : You can combine the usual wild card characters with the two dots `..` notation. Thus the following path is another way to address both boxes : `Hello..g*..b*`.

The two boxes belong to the `glist`. Thus if you move the glist the boxes will move along. Indeed, just try :

```
a> msge ..glist move 60 80
```

This sends the `move` message to the `glist`, asking it to move its origin to position 60 80. The two boxes follow :

```
a> setg Hello.glist.box -apos
= {80 100}
```

Remark : Let us note that if you move the `glist` using the `setg` command along with the `-pos` field, you would need to redraw the whole window to see the effective move. Now, if you use the `setgu` to change the `-pos` field (or any other field), LastWave only redraws the object after changing the corresponding field. Thus the moved object will be drawn at the new position but the old position will not be erased. Thus, if you need to move a graphic object (or resize it) and perform the redrawing right away i its always better to use the `move` (or `resize`) message instead of the `-pos` (or `-size`) field.

11.4 Talking about `View`'s

View objects are `GLists` (i.e., the class View inherits from the class `GList`) whose local coordinates can be redefined. The `-bound` field let's the user specify four values : <`xMin`>, <`xMax`>, <`yMin`> and <`yMax`> that define the abscissa and the ordinates of the boundaries of the left-bottom and right-top corners of the View in the local coordinates. Thus for instance,

```
a> window new 'Hello' -size 300 300 -hide 0
a> msge Hello add 'view' View -pos 10 10 -size 200 200 -frame 1 \
--> -bound 0 1 -10 10
= 1
```

creates a `View` named `'view'` in the Window `Hello` whose top-left corner is at the point 10 10 (window coordinate) and whose size is 200 200. Moreover, the `-bound` field was used to specify that the local coordinates of the `View` will be so that the left-bottom corner will be at local coordinate 0 -10 and the right-top corner at local coordinate 1 10. Let us note that, contrary to any other `GLists`, the local ordinate of a `View` increases when going towards the top of the window (as one usually draws axis). (This is the default behavior of a view. However you can invert any axis using the `-reverse` field). Thus if we want to draw a line from one corner to the other one we would do

```
a> draw line ..view 0 -10 1 10
```

So we can add a `Box` which origin would be at position 0 -10 and which size would be 0.5 5 using the regular `-pos` and `-size` field

```
a> msge ..view add 'box' Box -pos 0 -10 -size 0.5 5 -bg 'lightgrey'
= 1
```

Let us note that, both `-pos` and `-size` fields always use local coordinates. The fields `-apos` and `-asize` fields address the position and the size using global coordinates (window coordinates in pixels) :

```
a> setg ..box -size
= {0.5 0}
a> setg ..box -asize
= {100 50}
```

As in any `GLists`, you can add as many graphic objects to a `View` as you want (the `View` class inherits from the `GList` class):

```
a> msge ..view add 'box1' Box -pos 0.6 -3 -size 0.3 5 -bg 'darkgrey'
= 1
```

If you want to zoom the view in or out, you just need to change the `-bound` field. For zooming in :

```
a> setgu ..view -bound .3 .7 -10 5
```

and for zooming out

```
a> setgu ..view -bound -10 10 -50 50
```

Actually, instead of any value, you can send `'*'` or `'?'` to the `-bound` field. A `'*'` means that the current value should be used, and a `'?'` means that the minimum (for <xMax> or <yMax>) or the maximum (for <xMin> or <yMin>) possible value so that all the graphic objects in the view are visible will be used. Thus if you want to change the <xMax> to 5 and change both <yMin> and <yMax> so that they "stick" to the boxes, you would type

```
a> setgu ..view -bound '*' 5 '?'  '?'
```

The new boundaries will now be

```
a> setg ..view -bound
= {-10 5 -10 2}
```

As you will see later, the `View`'s are used to display signal graphs, images, ... anything that are naturally addressed using real x, y coordinates and which will be constantly zoomed in or out.

11.5 The graphic objects hierarchy

It's time now to tell you about all the graphic object classes that are predefined in LastWave (in the kernel only, not in the other packages). Actually, there are very few. There are only the very basic ones. It is very easy to extend them using scripts as you will see in the following sections (the best examples are the `Box`'s defined in the file `scripts/misc/box` and the `Button`'s in `scripts/misc/buttons`). For more complex graphic classes (such as the one for displaying signals or wavelet transforms), it will be better to directly write C-code. But don't worry! It is as easy as doing it in scripts. And it does not take that many lines! However, you should know about the graphic objects hierarchy defined in the LastWave kernel.

At the top there is the class GObject. All the other classes inherit from that class. The GList class inherits directly from the GObject class and the View class inherits directly from the GList class. However, the Window class does not inherit directly from the GList class, it inherits from the Grid class which itself inherits directly from the GList class. Thus, you have already heard about all the classes except the Grid class.

A word about Grid's

A Grid is a GList (i.e., it inherits from GList) whose local coordinates can be addressed in two ways. The regular way using the -pos and the -size fields and using "grid" coordinates using the -grid field. A Grid is divided into <m> columns and <n> lines using the -mn <m> <n> field. Then a gobject that is placed into a grid can be placed using these grid coordinates using the -grid <x> <y> <w> <h> field where <x> <y> is the position of the origin of the object in the grid coordinates (thus <x> must be greater than 1 and smaller than <m> and <y> must be greater than <1> and smaller than <n>) and <w> (resp. <h>) is the width (resp. height) in grid units of the object. The advantage of doing so is that the position and the size of the object will be automatically recomputed when the grid is moved or resized. Thus for instance if you want to create a window with a box which fills up the bottom half of the window, you would type

```
a> window new 'Hello' -size 300 300 -hide 0 -mn 2 2
a> msge Hello add 'box' Box -grid 1 2 2 1 -string 'Hello' -bg 'grey'
= 1
```

Just try to resize the window and you will see that the box will be automatically resized.

11.6 Let's learn about the disp command

This command is a very powerful command that has been defined using the command language. It uses new graphic classes that are also defined using the command language: the WindowDisp class, the FramedView class and the EView class. Let's see first how one can use this command in order to display signals (the same command will be used for displaying wavelet transforms, extrema, images, matching pursuit books,...).

When you display a single signal s using the command disp, it creates a window of type WindowDisp named Window1 and whose title is Window1 (&signali) indicating the type of objects the window is displaying (in our case &signali, i.e., input signals). Unless a window name is specified as the first argument of disp (in which case the corresponding window is used if it exists, or a new one using that name is created), it uses the last window which was used for displaying the same type of objects (&signali) or the last one which was visited by the mouse.

Each signal is displayed using a graphic object of the class called GraphSignal. These objects will be named 1,2,...N in the same order as they appear in the arguments of disp. Each of these gobjects are included in View's all named view (actually their class is an extended version of the View class called EView and that inherits from it). And each of these views is included into a FramedView which is a Grid which will display axis around the view and a Box (named box) at the bottom for displaying information. The FramedView's are

named `fv1`, `fv2`,..., `fvN` from left to right and top to bottom. Finally all these `FramedView`'s are included in the window (whose class inherits from `Window` and is `WindowDisp`).

Just move around the mouse on a signal after displaying some signals using the `disp` command. In the `Box` of the corresponding `FramedView` will be displayed, along with the mouse local coordinates, the complete path of the corresponding gobject.

As mentionned above, you can also use the left button of the mouse while pressing the `shift` and the `ctrl` key to learn about the different gobjects that are displayed. The `disp` command lets you display several signals (or as you will see later any other objects such as a wavelet transform) controlling the way they are dispatched in the window. To make 4 signals `s1`, `s2`, `s3` and `s4` appear one above the other one you just need to type

```
a> disp s1 s2 s3 s4
```

If you want the first two ones to be at the same level in the window, just type

```
a> disp {s1 s2} s3 s4
```

If you want the first two ones appear on the same graph and the last two ones appear below at the same level, you must type

```
a> disp {{s1 s2}} {s3 s4}
```

Using this syntax you control very easily the way they are dispatched in the window. Before displaying everything, the `disp` command calls the `setg` command on the window the display takes place along with all the optional arguments. Thus if you want to change the field `-curve` of the first signal and the field `-fg` of the third one, you must type

```
a> disp {{s1 s2}} {s3 s4} -..1 -curve '.'  -..3 -fg 'red'
```

as you would do in a `setg` command sent to the window. Actually after displaying some signals, you can change their fields directly using the `disp` command along with the fields argument

```
a> disp -..1 -curve '.'  -..3 -fg 'red'
```

In order to learn about the fields of a GraphSignal you can either read the help corresponding to this class or get the different field values using the mouse as explained in a previous section.

You should know about the fields of the `WindowDisp` itself. First, since it inherits from the `Window` class which itself inherits from the `Grid` graphic class, it has all the `Grid` fields. Let us point out that the `-dxdy Grid` field of a *WindowDisp* is generally set to 1. This is the reason why there are grey lines delimiting each graph. If you want to erase them, just set this field to 0. You can also play with the `-margin Grid` field for increasing or decreasing the margins around the axis. This is particularly useful if the numbers associated to the axis are too long and do not appear completely on the screen.

Apart from the `Grid` fields that are inherited, there are basically 4 fields which are specific to the `WindowDisp`: the `-x` to specify the <`xMin`> and <`xMax`> use in the view, the `-y` to specify the <`yMin`> and <`yMax`> use in the view, the `-S` <`flag`> field to superpose all the signals (if <`flag`>`==1`) and the `-s` field to specify if the different axis of the different graphs are synchronized when zooming in or out. Thus if you want to display the 2 signals `s1` and `s2` one on top of the other from the abscissa 0 to 10 using the same x and y-scales and with 2 different colors, you would type

```
a> disp s1 s2 -x 0 10 -..1 -fg 'blue' -..2 -fg 'red'
```

If you want to add labels to the axis or add a title to a graph, you can use the fields `-xLabel`, `-yLabel` and `-title` of `FramedView`'s. It creates `Text` graphic objects (named respectively `xlabel`, `ylabel` and `title`) that can be moved around using the `-pos` field or, as explained above, the middle mouse button along with the `ctrl` and `shift` keys. The `Text` graphic class will be fully described further. It is a very useful class. For instance, it will allow you to add any legends to a graph. Thus for instance

```
a> disp s1 s2 -..fv1 -xLabel 'x1' -yLabel 'y1' -title "This is s1" \
--> -..1 -fg 'blue' -..fv2 -xLabel 'x2' -yLabel 'y2' -title "This is s2" \
--> -..2 -fg 'red'
```

then you can change position of the title or the labels either using the mouse or directly like:

```
setgu ..fv1.xlabel -pos 10 10
setgu ..fv1.title -font '14' -pos 10 10
```

Remark : There is a pretty sophisticated system that lets you synchronize the x and/or y-scales of `EView`'s. This is managed by the `synchro` command which is automatically called when you set the `-s` flag of a `WindowDisp`. However, you can use the `syncho` command directly.

Remark : Let us note that `EView`'s, `FramedView`'s, `WindowDisp`'s,... use default values for their fields that are defined at the beginning of each corresponding files (e.g., `scripts/disp/framedView`, `scripts/disp/eview`, `scripts/disp/windowDisp`,...). You should play with them. The ones corresponding to `FramedView`'s allow you to change the way `FramedView`'s will look. You can change the margin that will be used to place the `View`, the look of the axis (you can ask for a whole frame and have the ticks inside the frame and change the margin between the view and the axis) etc... As an example, you should try the following values (in the file `scripts/disp/framedView`):

```
gclass.FramedView.default.axisMargin=0
gclass.FramedView.default.ticksIn=1
gclass.FramedView.default.axis=1
gclass.FramedView.default.axisFrame=1
gclass.FramedView.default.bg='darkgrey'
gclass.FramedView.default.viewbg='grey'
gclass.FramedView.default.boxbg='darkgrey'
```

11.7 More on the `disp` command

Later in this manual, you will see that there exist other types than `&signal` of complex structures including wavelet transforms `&wtrans`), images (`&image`), matching pursuit books (`&books`)... The `disp` command can be used to display indefferently any combination of these structures. The syntax is always the same. The general syntax is

`disp [<windowName>] {var11...var1N} ... {varP1...varPN} [display options]`

It will display the structures (e.g. signals, wavelet transforms...) associated to the variables *varij* (for $1 \leq i \leq N$ and $1 \leq j \leq P$) in the window <`windowName`>. The first structure list will be displayed on top of the window at the same level, the second one just below it and so on. Each of the structures will be displayed using a `FramedView` which includes an `EView` which will contain the graphic object that is able to display the corresponding structure.

When parsing these lists of variable names pointing to complex structures, the `disp` parser works the following way on each variable name:

- It looks whether it can be interpreted as a signal (of type `&signali`, thus either a variable name or an signal arithmetic expression). If it cannot, it tries to interpret it as another complex structure, e.g., a wavelet transform (`&wtrans`), an image (`&imagei`, which could be either a variable name or an image arithmetic expression),... As soon as it found a variable type `&<type>` it could be interpreted in it goes to the next step. If no type can be found then it generates an error.

- It creates a graphic object of class `Graph`<`type`> (e.g., `GraphSignal`, `GraphImage`, `GraphWtrans`,...) which, by convention is the name of the graphic class which should be used to display complex structures of type < *type* >.

- All the graphic classes `Graph`< *type* > should have a "set-only" `-graph` field that allows to set the complex structure to be displayed. Thus this field of the newly created graphic object is set with the corresponding structure. The corresponding "get-only" field is generally called `-?`<`type`> (e.g., `-?signal`, `-?wtrans`,...).

- This graphic object is added to the `EView` (named `view`) of either a newly created `FramedView` (named `fv`<`k`>) if the graph is not superposed to another graph or an already existing `FramedView` if not.

If you do not specify the <`windowName`> in the `disp` command, the window it will display all these structures, will be the "current" window (i.e., the last one visited by the mouse or used in a command) of the "right" type. By the "right" type, we mean a window of type `WindowDisp` that has been used fo displaying structures of the same type.

Actually, any `WindowDisp` has a `-type` field that informs you on the type of structures it is currently displaying. Thus, for instance, a `WindowDisp` of type `&signali` is a window that has been used to display signals. By default the `type` of the window is indicated in its title as in `Window1 ($signali)` (`Window1` is the name of the window and it is of type `&signali`). If a window is used to display strcutures of different types, its type is called `&mixed`.

Thus, as we already said if no <`windowName`> is specified, then the "last" window which has the right type is used for displaying the structures. This is very convenient: if you have several `&signali` windows opened, you just need to go with the mouse within the one you want to use and then type your `disp` command for displaying signals. It will use the right window!

If the <`windowName`> is specified, either it exists, in which case it checks whether it has the right type (if not there is an error) or it does not exist, in which case a new window of the right type is created.

11.8 Let's print !

Okay. The good news is that you can generate a (color) postscript file of anything you have displayed in a window. For that purpose, you just need to send the `drawps` message to a window with one argument being the name of the file. Before doing so, you can specify the exact size of the so-obtained print using the `ps` command. Let us note that a bouding box will be generated in the postscript file so that you can encapsulate the figures very easily.

However, you must be aware of something. In this version, LastWave will display exactly what you see on the screen. Thus, for images, it will use the resolution of the screen instead of the printer resolution. If, before printing a window, you just increase its size to get a large window, you will get a more precise print. Don't worry, the precision is really good, and it should not bother you.

Chapter 12

More insights about graphics

12.1 Some useful graphic classes

12.1.1 The `Shape` graphic class

The `Shape` graphic class is a script class (defined in the script file `scripts/misc/shape`) which allows to display ellipses and rectangles adressing them in very different ways. The way to use it is very simple. Let us draw for instance a sinus signal

```
disp sin(2*pi*I(200)/200)
```

This signal has the value 0 at abscissa $x = 100$. Let us imagine we want to circle this value. We would have to add a `Shape` object that we will name `ellipse` that will display an ellipse centered at the position $(100, 0)$. We have to fix the radius of this ellipse using local coordinates. We choose 5 in the x direction and 0.05 in the y direction :

```
msge ..view add 'ellipse' Shape -shape 'ellipse' -pos 100 0 \
--> -radius 5 .05 -hide 0
```

Let us note that if you use the mouse to zoom in or out the signal, the ellipse will be zoomed autmatically. However, in our case, we would like the ellipse not to change size when we zoom in or out. This means that we want to specify the radii as number of pixels. This is possible setting the `-pixel` field to 1 and resetting the radii :

```
setgu ..ellipse -radius 7 7 -pixel 1
```

This will draw a circle of radius 7 pixels. You can zoom in and out, the circle wil appear with always the same size!

In the same way you can draw a rectangle. You can fill these shapes using the `-fill` field, and specify their position using a framing rectangle (instead of specifying the center) using the `-centered` field.

12.1.2 The `Text` graphic class

The `Text` graphic class is a script class (defined in the script file `scripts/misc/text`) which allows to display strings. What is the difference between `Box`'s and `Text`'s? Well basically, whenever you need to frame a text and control precisely the coordinate of the frame itself

(not of the text inside the frame) you should use `Box`'s. Let us note that, the framing rectangle will be zoomed in if you place the `Box` object in a `View` and if you zoom the `View` whereas the string will NOT be zoomed.

Whenever you want to display a string (or a text) with a frame or without one and control precisely the position of the text (using different way to justify it) you should use `Text`'s objects. In the case, you ask for a frame around a `Text`, this frame will NOT be zoomed in if placed in a `View` when zooming the `View` in. Generally, in `View`'s you will use `Text`'s and not `Box`'s.

For instance let us imagine that, as in the previous example, you want to draw a sinus and circle its zero:

```
disp sin(2*pi*I(200)/200)
msge ..view add 'circle' Shape -shape 'ellipse' -pos 100 0 -radius 5 5 \
--> -pixel 1 -hide 0
```

Now, let us imagine we want to write a string `'This is zero'` close to the 0 value. In order not to interfere with the curve itself, we will position the beginning of the string at the abscissa 110 and ordinate 0. This is simply done by adding a `Text` object to the `View` object (named `view`) of the current window

```
msge ..view add 'text' Text -pos 110 0 -string "This is zero"
```

If you want to frame the text you can do

```
setgu ..text -frame 1
```

If you want to set a background color to the text you can do

```
setgu ..text -bg 'grey'
```

Now, try to perform a zoom (using the mouse), you will see that the begining of the text will always stay at the position `110 0` whereas the frame around the text will always be the same number of pixels.

You can pick up several justification modes for the horizontal position

- `left` : This is the default mode, the string is justified to the left,
- `rightN` : Every single line of the string is justified to the right,
- `right1` : The string as a whole text is justified to the right,
- `middleN` : Every single line of the string is justified to the middle,
- `middle1` : The string as a whole text is justified to the middle.

In the same way the vertical justification can be choosen among

- `base` : This is the default mode, the string is justified on the "base" of the first character of the first line. The "base" corresponds to the bottom point of the `a` character.
- `down` : The string is justified on the bottom of the last line. The "bottom" is defined as the bottom point of the lowest character.

- `up` : The string is justified on the top of the first line (it is defined as the top point of the highest character).
- `middle` : The string is justified on the middle of the first line (it is the middle between the lowest and the highest point).
- `middle1` : The string is justified on the middle of the first line (it is the middle between the highest and the base point).

In order to set these modes you need to use the `-posMode` field.

Editing and creating `Text`'s interactively : In the `scripts/misc/text` file are defined some very useful bindings for editing and creating `Text`'s interactively. Let us suppose a `Text` object is displayed on a window. Just go over it with the mouse. Then press the `escape` key to enter the edit mode (LastWave will notify you in the terminal window that you're entering that mode). Be careful : each time you leave the `Text` object with the mouse, you leave the editing mode automatically. Otherwise you have to press `escape` again. In the interactive editing mode you can do several things

- Hitting any key adds (at the end of the string) the corresponding character.
- Hitting the `delete` key deletes the last character.
- The `up/down` keys are used to change the font size.
- The `tab` key is used as a switch between a `plain` font or a `bold` font.

Moreover you can create a `Text` object wherever you want on the screen by simply moving the mouse to the point you want to put it and then press `f5`. It creates it with its name displayed in it and puts you in the editing mode, ready to type whatever you want (don't move the moouse, or you might just go out of the editing mode).

12.2 About Fonts

LastWave allows you to use any font you want when drawing a graphic object. The font names are strings that have the following syntax

```
'<name>-<size>-<style>'
```

where `<name>` is a font name (e.g., Geneva, Symbol,...), `<size>` is the size in points (e.g., 9,10,24,...) and `<style>` is the style of the font which could be any value between `'plain', 'bold', 'italic', 'boldItalic'`.

A font is associated to any graphic object. You can address this field using the field name `-font`.

By default the font of a graphic object is the default font set by the `font default` command. Thus for instance, if you want the default font to be courier 12pts, you must type

```
window new 'w' -size 200 200 -hide 0
font default 'Courier-12-plain'
```

```
msge w add 't' Text -pos 100 100 -string "Hello world\nHow are you"
setg w.t -font
= 'Courier-12-plain'
```

If you want to justify the above text to the right you should just type

```
setg w.t -posMode 'right' 'base'
msge w draw
```

You can get the list of all the fonts available on the computer. This is done using the command `font list` or you can ask whether a font exist using the `font exist` command. For better control of text (such as for the `Text` or `Box` classes), you need to access the bounding rectangle that defines the limit of a given text using a given font. This is done using the `font rect` command. Let us note that if you want to address the default font at a different size, you can ommit use a simple <`size`> syntax as in:

```
font default '18'
= 'Courier-18-plain'
```

12.3 Managing colors

12.3.1 Creating a new `named color`

Up to now we have only used colors associated to symbolic names such as `'red'` or `'blue'`. As we will se later there are other types of colors which we will refer to as `indexed` colors and which will be used for displaying images. In this section we are just going to describe how to define new named colors.

Any named color can be defined in two different ways : either using the RGB convention (each value is an integer between 0 and 65535) or the HSV convention (the Hue is an angle which ranges from 0 to 360 where both 0 and 360 correspond to red and 240 corresponds to blue, the Saturation and the Value range from 0 to 1). Most of the management of the colors is done using the "color" command. Thus, if we want to define a named color corresponding to a light red, we could use either the HSV convention (e.g., H = 0, S = 0.6 and V = 1)

```
a> color nnew 'lightred' hsv 0 .6 1
```

or the RGB convention (e.g., R = 50000, G = 0 and B = 0)

```
a> color nnew 'lightred' rgb 50000 0 0
```

Thus the action `nnew` is used for defining a new named color. Before using it, we must ask LastWave to install all the colors by typing

```
a> color install
```

Then we can use this color for displaying a signal

```
a> 0 = sin(0:#512:2*pi)
a> disp 0 -..1 -fg 'lightred' -pen 3
```

You can redefine this color if you are not happy with it (make it more red for instance) :

```
a> color nnew 'lightred' hsv 0 .9 1
a> color install
```

On a unix computer, you shoud not need to redraw the graphic windows. It should be automatically redrawn. This is never the case on a Macintosh. In any case if you want to be sure that the graphics are updated you must send drawing messages to the different windows.

Let us note that on unix/X11 8-bit station (only), you can change dynamically the definition of color without reinstalling all the colors and thus without anything to be redrawn using the `animate` action of the `color` command.

You can change the background and the foreground colors of the graphic windows as well as the color used when drawing in the inverse mode (the mode which is used generally for interactive display following the mouse) with the `setcolor` function

```
a> setcolor -bg 'red' -fg 'white' -mouse 'black'
```

(You can redefine only one or two colors among -bg, -fg and -mouse by just ommitting the other ones). Try to display a signal. It will appear in white with a red background and the mouse cursor (on a unix/X11 station only) will appear black. On a Macintosh it just "inverts" the actual colors, so you cannot control how it will look like. If you don't want to ruin your eyes, you'd better go back to the original foreground and background colors.

There are 2 predefined (constant) colors whose names are `'bgDefault'` and `'fgDefault'` (corresponding to the original background and foreground colors), so you should type

```
a> setcolor -bg 'bgDefault' -fg 'fgDefault'
```

Let us note that the names `'FG'` and `'BG'` always correspond to the actual colors of the foreground and the background. You cannot change their definition directly. You must use the `setcolor` command.

Remark : Let us note that the `-mouse color` used for drawing using the invert mode will work only on X11/Unix computers with enough colors. On Macintosh computers, you cannot control the color of the invert mode.

12.3.2 About `colormaps`

Up to now, we have only used `named` colors, i.e., symbolic names referring to a given RGB or HSV values. However, when displaying 2d real valued functions (e.g., images, wavelet transforms, ...) one needs to define a `colormap`. A colormap is an ordered list of *indexed* colors that are numbered from 0 to $N-1$ where N is the size of the colormap. The color 0 will be used to code the smallest values of a 2d real valued function and the color $N-1$ will be used to code the highest values. You can define several colormaps so that you could display at the same time a grey-scale image and a color image.

Named colors are addressed by their names whereas indexed colors are adressed by their index followed (with no space) by their corresponding colormap name : <`index`><`colormapName`> (where <`index`> varies from 0 to $N-1$). Thus whenever you need to specify a color, you

can use either the name of a named color or the so-described syntax for adressing an indexed colormap. If no <colormapName> is specified then the "current" colormap is used.

At startup, the current colormap is a simple black and white colormap ($N = 2$) called `bw`. However, at the end of the startup file, two colormaps are created. The first one is called `color` and corresponds to 30 colors going smoothly from black to red (going through blue). The second one is called *grey* and corresponds to a 256 grey level colormap going from black to white. Let us note that this second colormap is automatically created only if your screen can display more than 256 colors (allowing the definition of both colormaps).

We will teach in the next section how to create new colormaps. However, you can display them using the `cmdisp` command. This command is a script command defined in the `scripts/misc/color` file. It just display in a new window the current colormap or (if there is an argument) a specified colormap. Try to play with it :

```
cmdisp 'grey'
cmdisp
```

You can set the curent colormap to be the `grey` one using the `colormap current` command :

```
colormap current 'grey'
cmdisp
```

In the same way you can get the name of the current colormap :

```
colormap current
= 'grey'
```

Any defined colormap can be inversed using the syntax _<colormapName>. Thus you could display the inversed `grey` colormap :

```
cmdisp current '_grey'
```

Actually _ alone refers to the inversed current colormap. Thus if you just want to inverse the current colormap you should just type `colormap current _`. For instance

```
colormap current
= '_grey'
colormap current '_'
colormap current
= 'grey'
colormap current '_'
colormap current
= '_grey'
```

Let us note that in order to display a colormap, the `cmdisp` command uses a new graphic class (defined in the `scripts/misc/color` script file) called Colormap. This class allows to display a colormap exactly as it appears using the *cmdisp* command. Thus you can integrate it easily in any window, in order, for instance, to display it close to an image that is displayed using it. This graphic class has a single field which name is `-cm` and which corresponds to the colormap to be displayed.

In the next section, you will learn about computing wavelet transforms. Wavelet transforms are displayed using colormaps. Let's just give an example of display of a continuous wavelet transform of the dirac function. You must first be sure that the `wtrans1d` package has been loaded. If it is not, you should type

```
package load 'wtrans1d'
```

A prompt which looks like

```
a>
```

should appear (`a` represents a wavelet transform you are currently working on). Then you can compute and display the wavelet transform of a Dirac function :

```
a> colormap current 'color'
a> 0= I(512)==256
a> cwtd 1 6 6 2
a> disp a
```

The first line sets the current colormap to be the `standard` one (going from black to red). The second line sets the signal to be analyzed to be a 512-long dirac function. The third one computes the continuous wavelet transform and the fourth one displays the wavelet transform. It displays a color image using the current colormap in which black represents the smallest values of the wavelet transform and red the highest values. This gives an example of how colormaps are used in LastWave. You can set the current colormap to be the `grey` one and then redisplay the wavelet transform

```
a> colormap current 'grey'
a> disp a
```

High values will be coded using white and low values using black. You can specify a different colormap directly using *disp* by setting the `-cm` field of the object used for displaying the wavelet transform

```
a> disp a -..1 -cm '_grey'
```

The colors have been inverted! Since `grey` was the current colormap you could have made it shorter:

```
a> disp a -..1 -cm '_'
```

12.3.3 Creating colormaps

Let's imagine that you want to display a wavelet transform using a black and white colormap (without using the already defined `bw` colormap). Thus, we need to define a new colormap that we will name `cm`. Since it is of size 2, in order to create it, we should type

```
a> colormap new 'cm' 2
```

Then we must define (using the RGB convention for instance) the two indexed colors (0 and 1) of the colormap `cm` using the `color` command along with the `inew` action (exactly the same way as we used the `nnew` action in the previous section)

```
a> color inew 'cm' 0 rgb 0 0 0
a> color nnew 'cm' 1 rgb 65535 65535 65535
```

and then we just need to install the colors and have a look at what we just did:

```
a> color install
a> 0= I(512)==256
a> cwtd 1 5 6 2
a> disp a -..1 -cm 'cm'
```

Remark : Several named color and indexed colors could correspond physically to the same color. For efficiency, in that case, LastWave uses just one colorcell to store all these colors (eventhough it behaves as if you had different ones). However, if you use an 8-bit display, you can use a mamimum of 256 colors at the same time. LastWave will of course not allow you to go beyond that limit. In order to know how many (physically different) colors your display allows you to use at the same time, you should type `color nb`.

In the script file `scripts/misc/color` there is one very useful command for creating complex colormaps. It is called `cminit` and takes three arguments : the first one is the number of colors you want in the colormap, the second one is optional and is 0 if you want the colormap to be grey level and 1 if you want it to be color (which is the default case) and the last one (which is optional) corresponds to the name of the newly defined colormap (default is `color`). In the color case it will make a "temperature" like colormap starting from black and ending by red. In the black and white case it will create a simple grey level colormap. Let us note that if the number of colors you ask for is a negative number, it will generate the inversed colormap.

Remark : You should never create physically both a colormap and its inversed version. This uses more space than just creating one of them and using the _ syntax explained above for the other one.

12.4 The `draw` command

Before moving on, you should know the basics about drawing simple geometrical shapes in a graphic object. Be aware that we are not talking here about creating new graphic objects that represent specific shapes. We are just talking about simple drawing procedures that will be use in scripts for instance, for drawing a cursor that will follow the mouse. Whatever you will draw using the `draw` command will not be remembered by LastWave. It will not be redrawn on a refresh. Indeed in order to be redrawn automatically you would have to add a graphic object (a Shape object for instance). This is not our purpose here.

You can draw geometrical shapes using local coordinates (and clipping rectangle) of any graphic object using the `draw` command. This command is highly used when you want to draw things interactively (e.g., cursors,....) or when you want to create a new graphic class (see the next subsections). As a first example, let us simply draw a circle that fills a window :

```
a> window new 'Hello' -size 200 200 -hide 0
```

```
a> draw ellipse Hello 100 100 100 100 -color 'red' -pen 3 -centered
```

Let's explain what the arguments are. The first argument corresponds to the shape that must be drawn. In that case it is an ellipse. The second argument is the gobject the shape is drawn into (in that case, it is the window `Hello`). In order to specify where the ellipse should be drawn in the window, we must specify a rectangle using local coordinates. The regular way to specify this rectangle is the <x> <y> <w> <h> notation (the center of the ellipse will be at local coordinates <x>+<w>/2 and <y>+<h>/2. Since the `-centered` flag is on, this is not the way that is used. Actually the first two arguments (`100 100`) specify the center of the ellipse and last two arguments the two radii (`100 100`). Since the window is of size `200 200`, the ellipse will fill the whole window. The `-color` option allows to specify the color the ellipse will be drawn with and the `-pen` option specifies the size of the pen. If you wanted to fill the ellipse (and not just draw the contour) you should have specified the `-fill` flag.

In the same way you could draw rectangles (using `rect`), lines (using `line`), points, crosses... To learn more about the `draw` command you should read the help of that command.

In the same way you drew a circle in a window, you can actually draw in any other graphic objects using local coordinates. For instance let's draw a dirac function and let's circle the sample whose value is 1 using the "draw" command and the inverse mode.

```
a> 0 = I(512)==256
a> disp 0a
a> draw ellipse ..1 256 1 3 3 -centered -mode 'inverse' -pixel
```

Thus, as explained in former sections, `..1` refers to the graphic object displaying the dirac signal. The coordinates `1 256` are local coordinates corresponding to the sample whose value is 1. However, we do not want to specify the radii using local coordinates, we want to circle this sample using a certain number of pixels for the radii. This is why the `-pixel` option is on. Without it, the `3 3` would have been taken as distances using local coordinates. Using `-pixel`, the `3 3` means that the radius of the circle should be 3 pixels. Let us note that the `-mode inverse` means that the inverse mode should be used (for unix computers it means that the color specified by the `-mouse` option in the `setcolor` command will be used). The specificity of this mode is that if you redraw exactly the same thing the drawing disappear :

```
a> draw ellipse ..1 256 1 3 3 -centered -mode inverse -pixel
```

Remark : LastWave allows you to draw strings that can be justified in very different manners. Read the `draw string` manual to learn about them or the section about `Text` graphic object. Indeed they use the same options as the `draw string` command.

12.5 Managing events

There are 4 types of events in LastWave :

- The mouse events : motion events, button up and down events (with eventual modifier keys), enter and leave window events

- Keyboard events : key up and key down events
- Objects events : draw event (sent right after a gobject is drawn so that you can draw whatever you want on top of it), delete event (sent right after a gobject has been removed).
- Other events : for now this includes only error events sent right after an error occured.

You can bind to any script any of these events when occurring in a graphic object of a given class or to the terminal window (only key down and error events can be binded to the terminal window). For that purpose you must use the `setbinding` command. Bindings are grouped into binding groups so that they can be (de)activated or deleted all together. Right after a binding has been defined it is always deactivated until you specifically activate it.

A very simple binding, for instance, would be to bind the `esc-1` key sequence in the terminal to a beep. It will be part of the binding group `demo` that will be created. To do so, you just need to type

```
setbinding 'demo' terminal keyDown {esc 1} {terminal beep}
```

Since the binding group `demo` does not exist, this will automatically create it and deactivate it. The `terminal` argument specifies the that we are binding an event that occurs in the terminal. Generally this argument is the name of a graphic class. The `keyDown` argument corresponds to the type of the event you want to bind and `esc 1` the corresponding list of keys. Finally, the script `terminal beep` allows the terminal to beep. To activate this binding you must activate its group

```
binding activate 'demo'
```

Then you can try typing the escape key followed by the `1` key in the terminal window... it will beep!

The graphic class that is used to display signals is the `GraphSignal` class. Now let's say that whenever the right arrow key is pushed in a GraphSignal, we want the point the mouse is pointing to to be circled.

```
setbinding 'demo' GraphSignal keyDown 'right' {
  draw ellipse @object @x @y 3 3 -pixel -centered -color 'red'
}
```

Why dont you try it ? Great ! Let's see how it works.

In the script you must access the name of the graphic object the mouse is in as well as the local coordinates of the mouse. Whenever a binding is executed, LastWave defines a few binding "event variables" that you can access using the character `@`. These variables are not regular LastWave variables since they are of course read only variables. Thus for instance `@object` when used in a binding, always refer to the gobject which is associated to the binding (in our case it is the corresponding GraphSignal). Moreover the two variables `@x` and `@y` are `&num` variables which indicates the coordinates of the mouse using local coordinates of `@gobject`. Now you should fully understand the last example. The exhaustive list of the event variables is

- `@key (&string)` : (For key events only) The key which has been typed in
- `@type (&string)` : The type of the current event
- `@object (&string)` : The graphic object that received the event (Warning : this event variable `is not` a string variable, it directly refers to the gobject.).
- `@objname (&string)`: The name of the `@object`œ.
- `@window (&string)` : The window name the `@object` is in
- `@button (&string)` : The button that was (Un)Press or moved (for mouse events only)
- `@i, @j (&num)` : The current window coordinate of the mouse (in pixels)
- `@i,@j,@m,@n (&num)` : (For draw events only) The rectangle that must be redrawn (window coordinates)
- `@x, @y ( &num)` : The current local coordinates of the mouse
- `@x,@y,@w,@h (&num)` : (For draw events only) The rectangle that must be redrawn (window coordinates)

Remark : In the last example we specified that the binding was on the key `right`. LastWave knows about some special key names (`up, right, down, left, delete, clear, home, tab, esc, end` and all the function keys `f1,f2,f3,...`) and it will recognize any combination of any keys (or keynames) with the `ctrl, opt` (or the `alt` key on Unix computers) or the `shift` key (or several of them at the same time).

WARNING : On Unix/X11 computers, as we have already mentionned at the beginning of this manual, LastWave runs two processes. One of them uses the terminal window for a `getchar` and the other one runs the X Display environement and waits for X11 events. All the graphics windows are thus managed by the second process whereas the terminal window is managed by the first process. I think that the fact that LastWave does not open its own new terminal window feels very comfortable. You keep the type of terminal window you are used to. However, there is a drawback. LastWave does not know the codes that are sent to the terminal when hitting any "special key" (such as `ctrl` characters of function keys). Thus it will print out the corresponding codes. And these codes should be used for binding keyboard events of the terminal window instead of using their symbolic names (such as {`ctrl a`}). Thus, you should not use the names `f1,f2,...,right,left` when defining a terminal binding. Just have a look at the `scripts/keys` file and you will understand. You even might have to add your own codes in this file if they don't match mine's. However, whenever you bind events of graphic objects, it is taken care by the X11 environment, and the key codes are always the same, thus you can use key names such as `f1` and so on...

The most general key sequence for binding events has the following syntax :

```
'{{Modifier1 Key1} {Modifier2 Key2} ...  {ModifierN keyN}}'
```

where modifiers can be one of `ctrl, shift, opt, ctrlOpt, ctrlShift, optShift, ctrlOptShift` (the `opt` name stands for the `option` key on a Macintosh or the `alt` key on a unix computer). Let's delete the binding group `demo`

```
binding delete 'demo'
```

Let's say now that we want to draw a circle that follows interactively the mouse in a `GraphSignal` as soon we press the left button along with the `ctrl+shift` keys. We should use the event `leftButtonDown` along with the modifier keys `ctrl+shift` for drawing the circle the first time, the event `leftButtonMotion` along with `ctrl+shift` for moving the circle and the event `leftButtonUp` for erasing the last drawn circle. In order not to redraw the whole signal each time we move the mouse, we will draw using the invert mode. Let's redirect these two events towards three script commands named `_DemoClick`, `_DemoUnClick` and `_DemoMotion` (in LastWave whenever a variable or a script command is not supposed to be called directly interactively by the user, the corresponding name starts with an `_`)

```
setbinding 'demo' GraphSignal leftButtonDown ctrlShift {_DemoClick}
setbinding 'demo' GraphSignal leftButtonUp ctrlShift {_DemoUnClick}
setbinding 'demo' GraphSignal leftButtonMotion ctrlShift {_DemoMotion}
```

Let us note that you need to write `{_DemoClick}` and not `_DemoClick` since the command `setbinding` expects a `&script` argument. Since in `_DemoMotion` we will have to erase the "old" circle before drawing the new one, we need to remember what are the coordinates of the last drawn circle. The only way to do that is to put these coordinates in global variables that will be imported when we need them. We will call these coordinates `bindings.GraphSignal.demo.x` and `bindings.GraphSignal.demo.y` (The `bindings` global array followed by a graphic class name is used in all the LastWave scripts to store bindings variables related to the graphic class). Then, the command `_DemoClick` would be

```
setproc _DemoClick {} {
  import args 1 &array bindings.GraphSignal.demo
  demo.x=@x
  demo.y=@y
  draw ellipse @object demo.x demo.y 3 3 -centered -pixel -mode 'inverse'
}
```

The command `_DemoUnClick` would be

```
setproc _DemoUnClick {} {
  import args 1 bindings.GraphSignal.demo
  draw ellipse @object demo.x demo.y 3 3 -centered -pixel -mode 'inverse'
  var delete 1 bindings.GraphSignal.demo
}
```

And the command `_DemoMotion`

```
setproc _DemoMotion {} {
  # We erase the last draw circle
  import args 1 bindings.GraphSignal.demo
  draw ellipse @object demo.x demo.y 3 3 -centered -pixel -mode 'inverse'
```

```
  # Then we draw the new one
  demo.x=@x
  demo.y=@y
  draw ellipse @object demo.x demo.y 3 3 -centered -pixel -mode 'inverse'
}
```

That's it ! We just need to activate these bindings

```
binding activate 'demo'
```

Have fun playing with it !

Remark : As you might notice playing with it, even if you go out of the GraphSignal LastWave sends motion events to the same gobject. No `leave` or `enter` events are sent. Actually, whenever you press a button of a mouse, the mouse pointer is "grabbed" by LastWave until you unpress the button.

Remark : If you type all these commands in a file and then source the file, you should put as the first line of the file the following line

```
# This line should be put at the beginning of the script file
binding delete 'demo'
```

so that you can source several times the same file (after modification) with no risk of keeping all the different versions of the same bindings activated at the same time! It would lead to really weird behavior...

In the next subsetcion we present a slightly more complex application that will be the perfect introduction for programming your own binding scripts.

12.6 More about managing events

So now we want to be able to mark some points on the graph of a signal using the mouse. But we want these points to be associated to the `GraphSignal` so that they would appear whenever the graph is redrawn (e.g., while you are zooming or while you resize the view). Thus we need to keep the list of the points in a global variable which depends on the current `GraphSignal`. In order to do so we will use in the variable name an index which corresponds to the "id" of the gobject. The `id` is a string associated to a gobject and that is unique. As done in a lot of LastWave scripts, we will use the `gclass` array and the list variable will be `gclass.GraphSignal.<id>.demo.list`. Thus the click event (to add a point) will look like this

```
binding delete 'demo'

setbinding 'demo' GraphSignal leftButtonDown ctrl {_DemoClick}

setproc _DemoClick {} {
  id=[msge @object id]
  import args 1 gclass.GraphSignal.${id}.demo.list
  if (list is null) {list = {}}
  list+= {{@x @y}}
```

```
  draw ellipse @object @x @y 3 3 -pixel -centered -mode 'inverse'
}
```

Each time the graph is drawn, LastWave sends a `draw` event. Whenever such an event is sent we need to redraw the points if there are any. To test whether some points have been drawn on a graph we just need to test whether the corresponding list is empty or not.

```
setbinding 'demo' GraphSignal draw {_DemoDraw}

setproc _DemoDraw {} {
  id=[msge @object id]
  import args 1 gclass.GraphSignal.${id}.demo.list
  if (list is null) {return}
  if (list.length==0) return
  foreach i list {
    draw ellipse @object i[0] i[1] 3 3 -pixel -centered -mode 'inverse' -clip
  }
}
```

The `-clip` option means that the current clipping rectangle of the gobject should be used (since maybe just a part of the gobject is redrawn).Then, we need to delete the list variable when the graph is deleted, i.e., when a "delete" event is received (a gobject is deleted for instance when you display another signal on the window).

```
setbinding 'demo' GraphSignal delete {_DemoDelete}

setproc _DemoDelete {} {
  id=[msge @object id]
  var delete 1 gclass.GraphSignal.${id}.demo
  _DeleteEmptyGlobalArray gclass.GraphSignal.${id}
  _DeleteEmptyGlobalArray gclass.GraphSignal.${id}
}
```

The last line checks whether the global array `gclass.GraphSignal.${id}` is empty (the same array could be used by another binding). If it is then we should delete this array otherwise after playing around with the demo binding there would a lot of variable created that would not correspond to anything. Finally, Then we just need to activate the binding !

```
binding activate 'demo'
```

That's it ! You can try marking points on a graph then zooming this graph or resizing the window, the marks are still there !

12.7 Creating new graphic classes using the command language

In order to create a new graphic class, you basically need to write 3 script procedures. One for drawing the corresponding gobjects, one for managing their fields and another one for

managing the messages sent to these gobjects.

In this section, we are going to explain how a `Button` class can be implemented using the command language. It basically corresponds to the script file `scripts/misc/buttons`.

Let us first write the 3 procedures we mentionned above. Let's start with the `_GButtonSet` procedure which takes care of the fields of a button. It has 3 arguments, the first one is the name of the gobject, the second one is the name of the field (starting with the `-` character, e.g., `-pos`, `-size`,...) and the last one is either empty, which means that the procedure is supposed to return the value of the field, or it is a list of the arguments that are needed to set the field (generally just one argument). In case the name of the gobject is `?`, we must return a help string that consists of a list of {<fieldName> [<value>]} {one line help} for describing each field. In our case, we will have 5 fields:

- `colorOn` : color used when the button is on,
- `colorOff` : color used when the button is off,
- `title` : the text associated to the button,
- `state` : the state of the button : 0 for off and 1 for on and
- `handle` : the name of the procedure to call whenever the state of the button changes.

Each of these fields must be stored in a global variable whoses name depends on the gobject. Thus, as we already did before, we will use a global array using the id of the gobject `gclass.Button.<id>`. Thus, each field `<field>` will be stored in the global variable `gclass.Button.<id>.<field>`. The `_GButtonSet` procedure is very simple to write :

```
setproc _GButtonSet {obj field .l} {

# The help
if (obj == '?')  {
  return "{{{state [<flagOnOff>]} {Sets/Gets the state flag (1 corresponds to the butto
  {{colorOn [<color>]} {Sets/Gets the color that will be used when button is On.}} \
  {{colorOff [<color>]} {Sets/Gets the color that will be used when button is Off.}} \
  {{title [<title>]} {Sets/Gets the label of the button.}} \
  {{handle [<handleProcedure>]} {Sets/Gets the procedure name that will be called whene
}

# Get the id of the gobject
id=[msge $obj id]

# Import the global array where the fields are stored and call it "struct"
global `gclass.Button.$id struct`

# The "state" field
if (field == 'state') {
  # Case of a get
  if (l.length == 0) {
```

```
    return struct.state
    }
    # Case of a set
    if (type(l[0]) != '&num') {
      errorf "Bad field value for button %V" l[0]
    }
    struct.state= (l[0]!=0)
    return 1
  }

  # We just deal with the other fields altogether
  if ((field == 'colorOn') || (field == 'colorOff') || (field == 'title') ||
    (field == 'draw') || (field == 'handle')} {
    # Case of a get
    if (l.length == 0) {
      return struct.$field
    }
    # Case of a set
    if (type(l[0]) != '&string') {
      errorf "Bad field value for button %V" l[0]
    }
    struct.$field=l[0]
    return 1
    }
  }
```

We will now write the procedure that deals with the messages. On top of the user defined messages, you need to answer 2 standard messages : the `init` message called when the gobject is created (starting from the init message of the `GObject` class going down in the hierarchy to the `init` message of the graphic class of the gobject) and the "delete" message (sent in the reverse order as the `init` message). We are going to define 3 new messages `on`, `off` and `switch` to turn the button on or off or just switch it. As the latter procedure, the `_GButtonMsge` procedure has 3 arguments (the second one corresponding to the message sent instead of the field name) :

```
  setproc _GButtonMsge {obj message .l} {

    # The help
    if ($obj == '?')  {
      return "{{{on} {Turns the button on}} \
      {{off} {Turns the button off}}}"
    }

    # Get the id of the gobject
    id=[msge $obj id]

    # The init message in which we init all the fields
```

```
  if ($message == 'init') {
    gclass.Button.${id}.state=0
    setv gclass.Button.${id}.title "" -l 1
    setv gclass.Button.${id}.colorOn 'lightgrey' -l 1
    setv gclass.Button.${id}.colorOff 'grey' -l 1
    setv gclass.Button.${id}.handle "" -l 1
    return 1
  }

  # The delete message in which we must delete the array variable that stores the fields
  if ($message == delete) {
    var delete 1 gclass.Button.$id
    return 1
  }

  # Import the global array where the fields are stored and call it "struct"
  global `gclass.Button.$id struct`

  # The "on" message
  if (message == 'on') {
    if (struct.state == 1) {return 1}
    struct.state=1
    msge $obj draw
    # We must call the handle procedure
    if (struct.handle=="") return
    $struct.handle obj struct.state
    return 1
  }

  # The "off" message
  if (message == 'off') {
    if (struct.state == 0) {return 1}
    struct.state=0
    # We must call the handle procedure
    if (struct.handle=="") return
    $struct.handle obj struct.state
    msge $obj draw
    return 1
  }

  # The "switch" message
  if (message == 'switch') {
    if (struct.state == 0) {msge $obj on} else {msge $obj off}
    return 1
  }
}
```

Now, we need to write the draw procedure, which is extremely simple: we will just draw a box a a centered text. It takes two argument : the object name and a list of 4 floats which correspond to the rectangle (x, y, w, h) (local coordinates) that needs to be redrawn. Unless it takes a long time to draw the gobject you should always redraw the whole gobject instead of just redrawing the part that needs redrawing (LastWave will clip whatever you redraw to that rectangle so that even if you draw things outside of this rectangle they will not be redrawn).

```
#
# The general Drawing method
#
setproc _GButtonDraw {obj .l} {

  # Get the id of the gobject
  id=[msge $obj id]

  # Get the fields
  global ‘gclass.Button.$id struct‘

  # Get the size of the button
  {w h} = [setg $obj -size]

  # Draw a rectangle (the color depends on the state of the button)
  if (struct.state == 0) {
    draw rect $obj 0 0 w h -fill -color struct.colorOff
  } else {
    draw rect $obj 0 0 w h -fill -color struct.colorOn
  }

  # Then draw a centered string and a frame rectangle
  draw cstring $obj 0 0 w h struct.title
  draw rect $obj 0 0 w h
}
```

We can now define the new graphic class `Button` which inherits from the basic graphic class `GObject` giving the the 3 procedures we just defined :

```
gclass new Button GObject %_GButtonSet %_GButtonMsge %_GButtonDraw "Graphic Class to impl
proc delete %_GButtonSet %_GButtonMsge %_GButtonDraw
```

Since these commands are never called directly by the user, we can remove them from the list of available commands. this is done by the `proc delete` command.

Before using the so-defined Button class, we must define the associated mouse bindings. We will just implement a left button click binding (a more elaborate behavior is implemented in the `scripts/misc/button` file. We just need to switch the button when the user clicks it with the left button :

```
binding delete ’button’
```

```
setbinding 'button' Button leftButtonDown {msge @object switch}
binding activate 'button'
```

Before playing with it, we need to write a simple handle procedure :

```
setproc _SimpleHandle {obj state} {
  echo State of $obj has changed to $state
}
```

Okay, let's just create a simple Hello World button in a window :

```
a> window new 'Hello' -size 200 200 -hide 0
a> msge .  add 'but' Button -pos 50 50 -size 100 50 -title "Hello World" -handle '_Simpl
```

Now just play with it.... clicking in it, or sending messages. You can even, for instance, add a button to a View where a signal is being displayed specifying the size of the button in local coordinates ! When using the zoom, the button will zoom !!

```
a> 0 = Grand(100)
a> disp 0
a> msge ..view add 'but' Button -pos 30 0 -size 50 .5 -title "Hello World" -handle '_Sim
```

As a good exercise, you should read all the script files in the `scripts/misc` directory. They correspond to the `misc` package. The `scripts/misc/box` file gives a simple example of inheritance by creating a `Numbox` class which inherits from the `Box` class and which displays a floating number.

Let us note that in order to perform a class conversion of a graphic object, you need to use the syntax $< class1 >:< gobject >$ which converts the <`gobject`> to <`class1`> (of course it works only if the class of the gobject inherits from <`class1`>). That allows to address messages or fields of ancestor classes when they have been redefined by the current class (an example is given in the `set` message of the `Numbox` class that redefines the `set` message of the `Box` class).