Bayes Forecast

Dynamic Modelling of Operations and Markets

TOL

TOL MANUAL

Contents

1 Introduction

This manual aims to provide an introduction to analysts in the use of *TOL* language. *TOL* is a language that is fundamentally aimed at analysing time-indexed information. It is especially useful for statistical analysis and for modelling dynamic processes. To do this, it uses time algebra and time-series as instruments that enable good information analysis and management.

Syntactic and semantic elements of the language will be introduced throughout the manual, as well as some functions and operators to handle different types of data.

This manual is also essential reading for those starting out in MMS-based modelling who have no prior *TOL* programming knowledge.

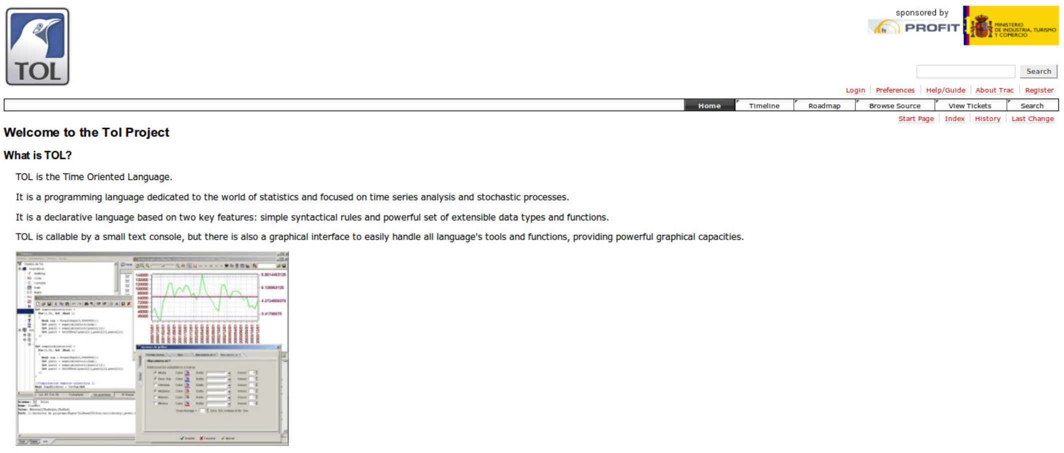
The key characteristics of *TOL* are:

- It is oriented to time-series management. TOL uses an object to represent time-series. This forms the basis for the representation of time-series. Please see section 2.4.
- Interpreted: this characteristic gives the analyst a certain amount of flexibility when it comes to building prototypes, as its trial and error cycle is very agile in comparison with a compiled language.
- Strong typing: All *TOL* expressions have a specific type of data which make it possible for the interpreter to apply optimisations to code evaluation. Please seesection 2.1.1.
- Self-evaluation: This allows for the evaluation of *TOL* code within *TOL* code. This creates the possibility of customised *TOL* programs. Although this is a very flexible construction, it shouldn't be abused as the expressions evaluated in this way need to be analysed by the *parser* every time.
- Oriented to objects: Although TOL isn't a language purely oriented towards objects, it does implement class concepts with multiple inheritence; therefore facilitating the implementation of solutions through an object-oriented design. This characteristic is implemented from version 2.0.1 onwards. Please see section 2.5.2.

The *TOL* working environment consists of a website which acts as the point of assistance to *TOL* users and also as the place where y*TOL* evaluation tools can be downloaded. In this chapter, we'll describe the TOL work environment in more detail.

1.1 *TOL-Project* trac. Wiki

The development of the *TOL-Project*, is located at the website http://www.tol-project.org. This acts as the meeting point for *TOL* user and language developers


sponsored by PROFIT
Search
Login | Preferences | Help/Guide | About Trac | Register
Home | Timeline | Roadmap | Browse Source | View Tickets | Search
Start Page | Index | History | Last Change

Welcome to the Tol Project

What is TOL?

TOL is the Time Oriented Language.

It is a programming language dedicated to the world of statistics and focused on time series analysis and stochastic processes.

It is a declarative language based on two key features: simple syntactical rules and powerful set of extensible data types and functions.

TOL is callable by a small text console, but there is also a graphical interface to easily handle all language's tools and functions, providing powerful graphical capacities.

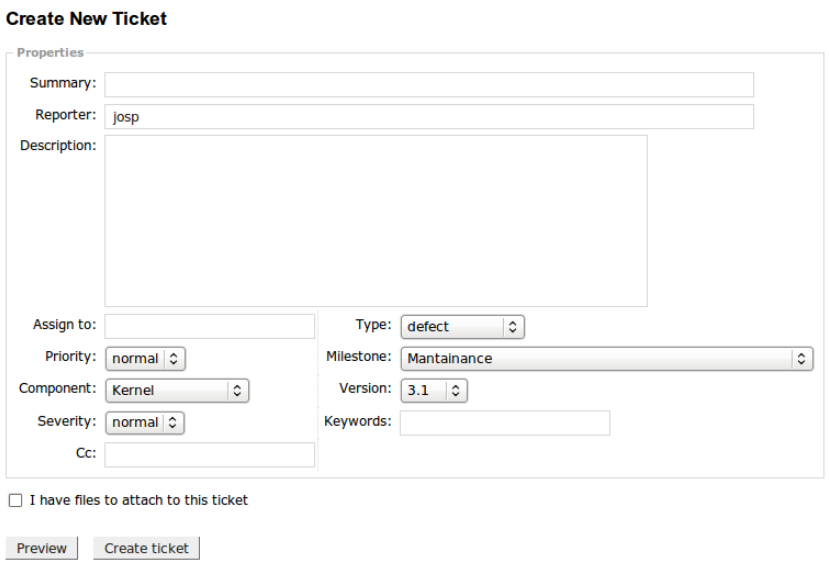
Figure 1.1.1: *TOL-Project* website.

TOL-Project is based on the *trac* system (http://trac.edgewall.org) which offers a collaborative framework (*wiki*) to *TOL* developers to create informative content about the language. It also acts as a system for reporting and following up incidents (tickets), and their use by the user community.

The first recommended step for any new user of *TOL* is to register with the *TOL-Project trac* . Once registration is complete, the user will be able to interact with the user/developer community via the ticket reporting and follow-up system. All that's then required is for the selection of a unique username, a password and plenty of enthusiam to collaborate with *TOL*!

1.2 Ticket system

There are many different ways to collaborate in *TOL*'s development. The simplest one is to start using it and to give feedback on doubts experienced, errors encountered, possible improvements and user tips and shortcuts. In certain cases users find solutions to existing errors and offer them for inclusion in future versions of *TOL*.



Create New Ticket

Properties

Summary:			
Reporter:	josp		
Description:			
Assign to:		Type:	defect
Priority:	normal	Milestone:	Mantainance
Component:	Kernel	Version:	3.1
Severity:	normal	Keywords:	
Cc:			

☐ I have files to attach to this ticket

Figure 1.2.1: New ticket form.

When creating a new ticket, it's necessary to add your (field Summary) and description (field Description) to the form. The user can also indicate a priority level (field Priority) to the ticket, indicating the importance being given to the request, as well as choosing a severity level for the issue in question (field Severity).

Other important fields when creating a new ticket include:

- Component : The *TOL* module or component related with the ticket. In cases of uncertainty, the value for this field can be omitted. Each component is associated with a developer. If the field Assign to is left blank, a developer will be assigned automatically..
- Cc: usernames/user email addresses of those who need to be copied in on notifications regarding changes to the request.
- Milestone: is the stage or category in which the ticket is to be resolved. The *milestone* is generally used for version planning. However, in *TOL-Project* it is more commonly used to conceptually classify requests.
- Version: The version number of *TOL* where the error was detected. This is very important as the first step to resolving the issue to reproduce the original situation as exactly as possible. In order to do this, the same version of *TOL* is required.
- Keywords: This is a list of keywords associated with the request, which can be very useful when searching for specific terms.

It's also possible to attach files that may help explain the issue in question. For example, *TOL* source code or screen-grabs of the moment the error occured .

Note that the text included in the ticket description can make use of the *trac wiki*'s capacity for formatted text content . The *wiki* supports a hypertext format with very symple syntax. A description of this syntax can be seen at https://www.tol-project.org/wiki/WikiFormatting.

Please take into account how important it is to include all possible relevant information in the ticket. This is especially true in cases of error-reporting. Including a segment of code in the description, or in an attached file, can be of immense value to the developer trying to reproduce the error in his working environment. However, there may be instances where this isn't possible or time doesn't allow for the error to be isolated. No matter, the most important thing is to include the description, the error trace or *TOL* log. This is always preferable to not providing any information or leaving the error hidden.

1.3 Source-code

As of 2002, *TOL* adopted an open-source development model, under an open source-code licence, namely the *GNU GPL* licence. This development model has allowed us to use software packages developed under open source-licences. This has acted as a huge catalyst for *TOL*'s development.

A large part of *TOL*'s algorithms are supported by the use of other packages which are available in open source form. From those, it is worth higlighting:

- *ALGLIB*: A cross-platform numerical analysis and data processing library. http://www.alglib.net
- *BLAS*: A basic linear algebra library. http://www.netlib.org/blas/index.html

- *Boost Spirit*: A *Boost* component for the implementation of object-oriented recursive descent parsers http://spirit.sourceforge.net, http://www.boost.org
- *BZip2*: A high-quality, open-source data compressor http://www.bzip.org
- *CLUSTERLIB*: A library used for segmentation. Written in *C*. http://bonsai.ims.u-tokyo.ac.jp/~mdehoon/software/cluster/software.htm#source
- *DCDFLIB*: >C/C++ *library used for distribution function evaluation.* http://people.scs.fsu.edu/~burkardt/cpp_src/dcdflib/dcdflib.html
- *Google Sparse Hash*: Extremely efficient for *hash_map* data structure implementation. http://code.google.com/p/google-sparsehash
- *GSL*: *GNU*'s scientific data library. http://www.gnu.org/software/gsl
- *iKMLOCAL*: Implementation in *C++*, effective for segmentation based in*K-Means*. http://www.cs.umd.edu/users/mount/Projects/KMeans
- *LAPACK*: A lineal algebra package, goes hand-in-hand with *BLAS*. http://www.netlib.org/lapack/index.html
- *Optimal_bw*: Effective use of kernal density estimation with optimal bandwith selection. http://www.umiacs.umd.edu/~vikas/Software/optimal_bw/optimal_bw_code.htm
- *SuiteSparse CHOLMOD*: A library for *Cholesky* factoring in disperse storage. http://www.cise.ufl.edu/research/sparse/cholmod
- *ZipArchive*: A *C++* library for the compression of files and data in*ZIP*format. http://www.artpol-software.com.

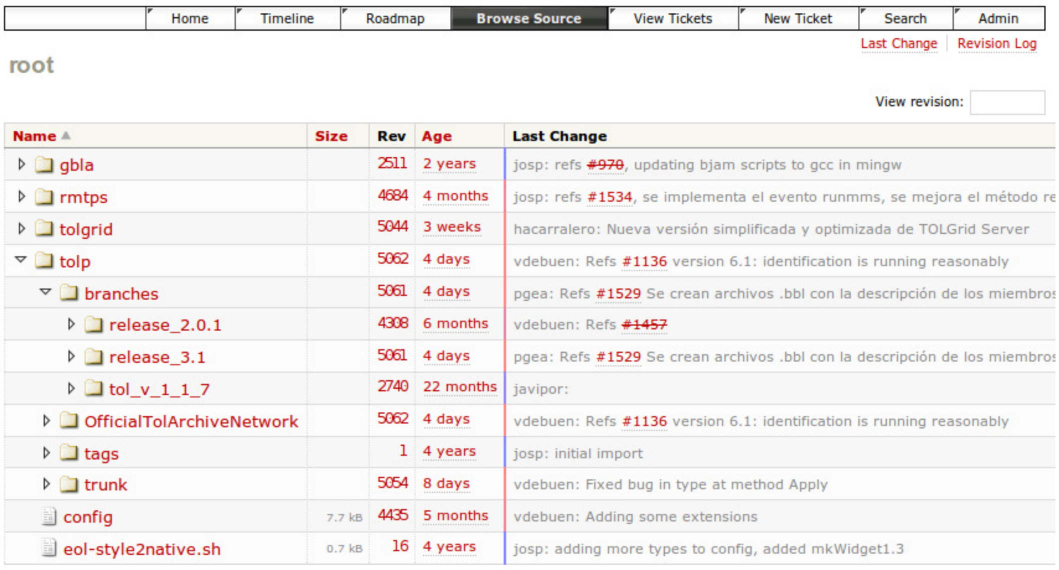
TOL is implemented in *C++* and runs on the *Linux* and *Windows* operating systems. There are different interfaces to interact with the language which include a command-line interface, remote command server, graphic interface and web interface. It's also possible to use the language via a dynamic link with the *TOL* library. This is currently possible from *C++*, *Java*, *Tcl* and *Visual Basic*.

Although this manual isn't aimed at developers, we will explain how to access and explore *TOL* source code. This can end up being useful when revising part of the standard library implemented in *TOL*, when its source code is stored in the same repository as that of the *TOL* kernel written in *C++*.

Nowadays, software is always developed with the support of a revision control system. Various programs exist for this purpose. Some of them are proprietary, whilst some are well-known free software such as *CVS*, *GIT*, *FOSSIL*, *SVN*, amongst others. *TOL* source code is stored in the *SVN* revision control system (http://subversion.apache.org). With*SVN* we can manage various versions of the code, look through the entire history of changes implemented and find out all the differences between source code from one version to the next. We can also identify the developer responsible for a particular change, as well as other functions.

We can explore *TOL* source code from the appropriate *TRAC* or through the support of an*SVN* program client. The majority of distributions of *Linux* offer the option of installing *SVN* from their software package repository. In the case of *Windows* we can download a client program from http://subversion.apache.org/packages.html.

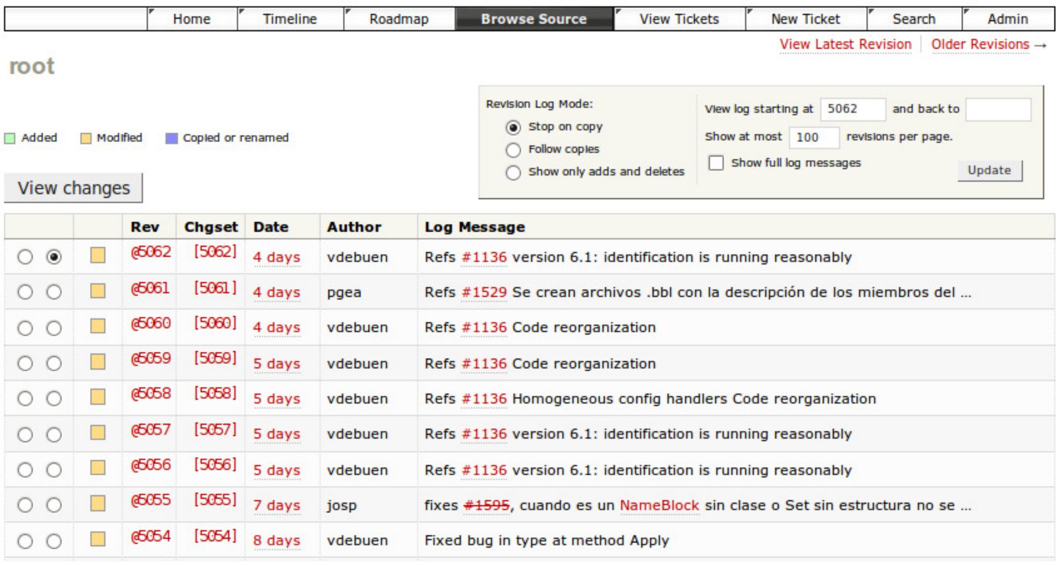
We can explore the repository's content and access the files contained in registered *SVN* directories via the web interface. *TOL* source-code is stored in the *tolp* directory; inside which we can find the directories *trunk* and *branches*, amongst others. The *trunk* directory contains the base code, from which the development version of *TOL* is built. This version generally contains quite unstable and untested functionality. Newly released versions can be located in the *branches* directory, each one with its own change history. These versions are usually more stable than the *trunk* version, although with less functionality.



Name ▲	Size	Rev	Age	Last Change
▷ gbla		2511	2 years	josp: refs ~~#970~~, updating bjam scripts to gcc in mingw
▷ rmtps		4684	4 months	josp: refs #1534, se implementa el evento runmms, se mejora el método re
▷ tolgrid		5044	3 weeks	hacarralero: Nueva versión simplificada y optimizada de TOLGrid Server
▽ tolp		5062	4 days	vdebuen: Refs #1136 version 6.1: identification is running reasonably
▽ branches		5061	4 days	pgea: Refs #1529 Se crean archivos .bbl con la descripción de los miembros
▷ release_2.0.1		4308	6 months	vdebuen: Refs ~~#1457~~
▷ release_3.1		5061	4 days	pgea: Refs #1529 Se crean archivos .bbl con la descripción de los miembros
▷ tol_v_1_1_7		2740	22 months	javipor:
▷ OfficialTolArchiveNetwork		5062	4 days	vdebuen: Refs #1136 version 6.1: identification is running reasonably
▷ tags		1	4 years	josp: initial import
▷ trunk		5054	8 days	vdebuen: Fixed bug in type at method Apply
config	7.7 kB	4435	5 months	vdebuen: Adding some extensions
eol-style2native.sh	0.7 kB	16	4 years	josp: adding more types to config, added mkWidget1.3

Figure 1.3.1: *TOL* source code web browser in the *TOL-Project trac*.

With *SVN*, each change made by a developer results in an increase in an internal number that acts as a "screengrab" of the whole repository. However, the change is stored internally as a change to the previous state. This number is known as *revision*.



Revision Log Mode: [✓ Stop on copy] Follow copies Show only adds and deletes

View log starting at 5062 and back to

Show at most 100 revisions per page.

☐ Show full log messages

			Rev	Chgset	Date	Author	Log Message
○	●	🟨	@5062	[5062]	4 days	vdebuen	Refs #1136 version 6.1: identification is running reasonably
○	○	🟨	@5061	[5061]	4 days	pgea	Refs #1529 Se crean archivos .bbl con la descripción de los miembros del ...
○	○	🟨	@5060	[5060]	4 days	vdebuen	Refs #1136 Code reorganization
○	○	🟨	@5059	[5059]	5 days	vdebuen	Refs #1136 Code reorganization
○	○	🟨	@5058	[5058]	5 days	vdebuen	Refs #1136 Homogeneous config handlers Code reorganization
○	○	🟨	@5057	[5057]	5 days	vdebuen	Refs #1136 version 6.1: identification is running reasonably
○	○	🟨	@5056	[5056]	5 days	vdebuen	Refs #1136 version 6.1: identification is running reasonably
○	○	🟨	@5055	[5055]	7 days	josp	fixes ~~#1595~~, cuando es un NameBlock sin clase o Set sin estructura no se ...
○	○	🟨	@5054	[5054]	8 days	vdebuen	Fixed bug in type at method Apply

Figure 1.3.2: *TOL* code revision explorer in the *TOL-Project trac*.

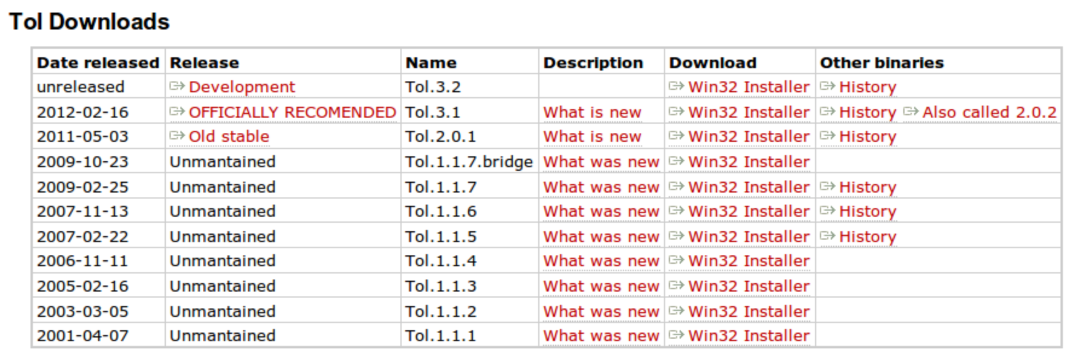
One of the functions accessible via the web interface is the change history. We can see the changes introduced to the code during its most recent revision, just as we can see the changes between any other previous versions of the revision and compare them with a more recent one. To carry out this function, we need to select the revisions that we want to compare and then click on View Changes.

1.4 Download and installation

As previously stated *TOL* can be used in *Windows* and *Linux*.l At the time of writing this manual, all the binary programs that we generate for both operating systems consist of *32 bits*. Therefore, even if a *64 bit* computer is being used, we have to use *TOL* compiled for *32 bits*.

1.4.1 Installation in *Windows*

To install the interpreters for *TOL* in *Windows*we have to download the program installer from https://www.tol-project.org/wiki/DownloadTol. This page includes a table which shows which versions are available to install *TOL*.



Tol Downloads

Date released	Release	Name	Description	Download	Other binaries
unreleased	Development	Tol.3.2		Win32 Installer	History
2012-02-16	OFFICIALLY RECOMENDED	Tol.3.1	What is new	Win32 Installer	History Also called 2.0.2
2011-05-03	Old stable	Tol.2.0.1	What is new	Win32 Installer	History
2009-10-23	Unmantained	Tol.1.1.7.bridge	What was new	Win32 Installer	
2009-02-25	Unmantained	Tol.1.1.7	What was new	Win32 Installer	History
2007-11-13	Unmantained	Tol.1.1.6	What was new	Win32 Installer	History
2007-02-22	Unmantained	Tol.1.1.5	What was new	Win32 Installer	History
2006-11-11	Unmantained	Tol.1.1.4	What was new	Win32 Installer	
2005-02-16	Unmantained	Tol.1.1.3	What was new	Win32 Installer	
2003-03-05	Unmantained	Tol.1.1.2	What was new	Win32 Installer	
2001-04-07	Unmantained	Tol.1.1.1	What was new	Win32 Installer	

Figura 1.4.1: Tables of versions available to download in*Windows*.

The first row of the table shows the version currently in development. Previously released versions are listed in the subsequent rows of the table, in descending order of newest to oldest. The link required for the download appears in the column marked Download<t1/>

Running the installer, for example the one downloaded from http://packages.tol-project.org/win32/tolbase-v3.2-setup.exewill result in the installation of binary files included by the *TOLBase* graphic interface, the command-line interface control panel (to program *tol* and *tolsh*) and the *vbtol* library to enable use of of*TOL* from *Visual Basic*.

There are some user functions installed in *TOL* that use *R*, invoking it externally via the command-line interface. These functions therefore require that both *R* and the packages *quadprog*, *coda*, *Rglpk* and *slam* be installed. In order to do this, *R* should be downloaded and installed from http://www.r-project.org before going on to run the following instructions on an R console:

```
install.packages("quadprog")
install.packages("coda")
install.packages("Rglpk")
install.packages("slam")
```

1.4.2 Installation in *Linux*

There isn't a similar installer available for*Linux* to the one for *Windows*. Therefore, the most common course of action is to compile the source-code. This requires certain abilities relating to the compilation process in *Windows*, as well as aptitude in using certain *Linux* commands, to install the compilation requirements. This manual doesn't include a description of the steps to compile the source code of*TOL* interpreters.

That said, a*TOL* software bundle does exist that allows the installation of *TOL* in the *Linux* distribution referred to as *CentOS*. The aforementioned software bundle had been been tested in version*5.4* of *CentOS*.

This distribution to install *TOL* in *CentOS* can be downloaded at: http://packages.tol-project.org/linux/binaries.

In this case, the installation process can be carried out by taking the following steps:

- Installation of prerequisites

```
sudo rpm -Uvh sysreq/epel/5/i386epel-release-5-4.noarch.rpm
sudo yum install atlas-sse2.i386
sudo ln -s /usr/lib/atlas/liblapack.so.3 /opt/tolapp-3.1/lib/liblapack.so
sudo ln -s /usr/lib/atlas/libf77blas.so.3 /opt/tolapp-3.1/lib/libblas.so
sudo yum install glibc-devel.i386 gsl.i386 R-core.i386 R-devel.i386
echo 'options(repos="http://cran.r-project.org")' > /tmp/Rinstall.R
echo 'install.packages("coda")' >> /tmp/Rinstall.R
echo 'install.packages("quadprog")' >> /tmp/Rinstall.R
echo 'install.packages("Rglpk")' >> /tmp/Rinstall.R
sudo R BATCH -f /tmp/Rinstall.R
rm /tmp/Rinstall.R
```

- Download the distribution

```
cd /tmp
wget http://packages.tol-project.org/linux/binaries/TOLDIST_3.1_p012.tar.bz2
tar zxf TOLDIST_3.1_p012.tar.bz2
cd TOLDIST_3.1_p012
```

- Installation

```
sudo ./install --prefix=/opt/tolapp
```

TOL 's libraries and programs will be installed under the directory indicated by the parameter`--prefix`. Once completed we can interact with *TOL*'s text mode using the following command:

```
/opt/tolapp/bin/tolsh -d
```

```
TOL interactive shell activated...
15:29:30 TOL>
```

We can now run *TOL* sentences, such as:

```
WriteLn(Version);
```

```
v3.1 p012 2012-06-14 19:33:46 +0200 CEST i686-linux-gnu
```

1.5 *TOL* programs

Upon installing *TOL*,*Windows* or *Linux* software,we will have a set of libraries and programs at our disposal which will allow us to develop solutions written in *TOL* language. Out of these programs, the most frequently used are the command console (*tol* o *tolsh*) and the graphic interface *TOLBase*.

1.5.1 The command console

The programs *tol* and *tolsh* are interpreters of *TOL*language, essentially used for batch processing of programs written in*TOL*. We can also run it in interactive mode and compile *TOL* expressions written in the *DOS* console (for *Windows* users) or *SH* (for *Linux*users).

When run in interactive mode, each evaluated expression generates a result that is stored in an object stack. The result can later be reused in the same session.

The programs *tolsh* and *tol* operate in a similar way, except that *tolsh* implements a server mode that allows it to remain running, listening on a *TCP/IP* port for remote evaluation orders. Remote evaluation orders usually arrive from another *TOL* programme client. A *TOL* client program can be the same *tolsh*, *TOLBase* or another one dynamically linked to the *TOL* library.

Upon running the *TOL* interpreter from the comman line interface we can specify various .tol files that will be interpreted in the desired order. We can also make use of the following options:

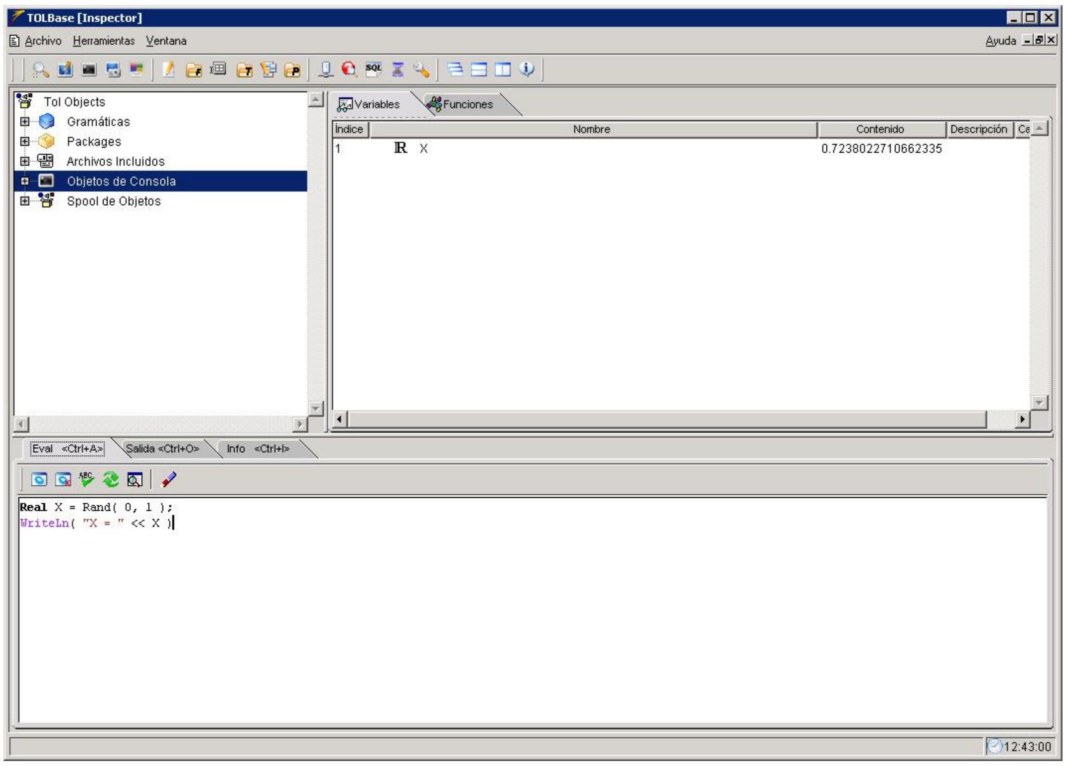
- `-i`: Don't include the standard *TOL* library.
- `-c"..."`: evaluate the *TOL* expression specified in inverted commas.
- `-d`: Start *TOL* in interactive mode. After evaluating the files and expressions specified in the command line, it shows the user where the entry line is to type *TOL* expressions. *TOL* expressions are evaluated upon pressing the "Return" key. The evaluation result is displayed on the screen and a new expression is then asked for.
- `-vE`: activates the u of error messages. Error messages are emitted by the `Error`function or `WriteLn` with message parameter type `"E"`.
- `-mE`: disables the release of error messages.
- `-vW`: activates the release of (*warning*) messages. Warning messages are emitted by the `Warning` function or `WriteLn` with message parameter type `"W"`.
- `-mW`: disables the release of (*warning*) messages.
- `-vS`: activates the release of system messages. System messages are notifications emitted by *TOL*'s functions and internal algorithms, e.g. `Estimate`.
- `-mS`: disables the release of system messages.
- `-vU`: activates the release of user messages.. User messages are emitted by `WriteLn` with parameter message type equal to `"U"`.
- `-mU`:disables the release of user messages>.
- `-vT`: activates the release of trace messages. Trace messages are messages emitted by `WriteLn` with message parameter type equal to `"T"` and those emitted by the internal function `Trace`.
- `-mT`: disables the release of trace messages.
- `-v?A?`: activates all types of releases.
- `-m?A?`: disables all types of releases.

1.5.2 *TOLBase*

TOLBase is a *TOL* client program that offers a graphic interface that facilitates interaction with the *TOL* interpreter and the objects created as a result of of the evaluation of *TOL* code.

In figure 1.5.1 we can see the default components we find upon opening *TOLBase*. The image shows the main window known as *object inspector*. This windows consists of three panels:

- An object tree.
- A panel with the object elements selected in the tree.
- A panel with the evaluation tab(Eval) known as the *console*, the output evaluation tab (Salida) and an information tab.(Info)



Indice		Nombre	Contenido	Descripción	Ca
1	ℝ	X	0.7238022710662335		

```
Real X = Rand( 0, 1 );
WriteLn( "X = " << X )
```

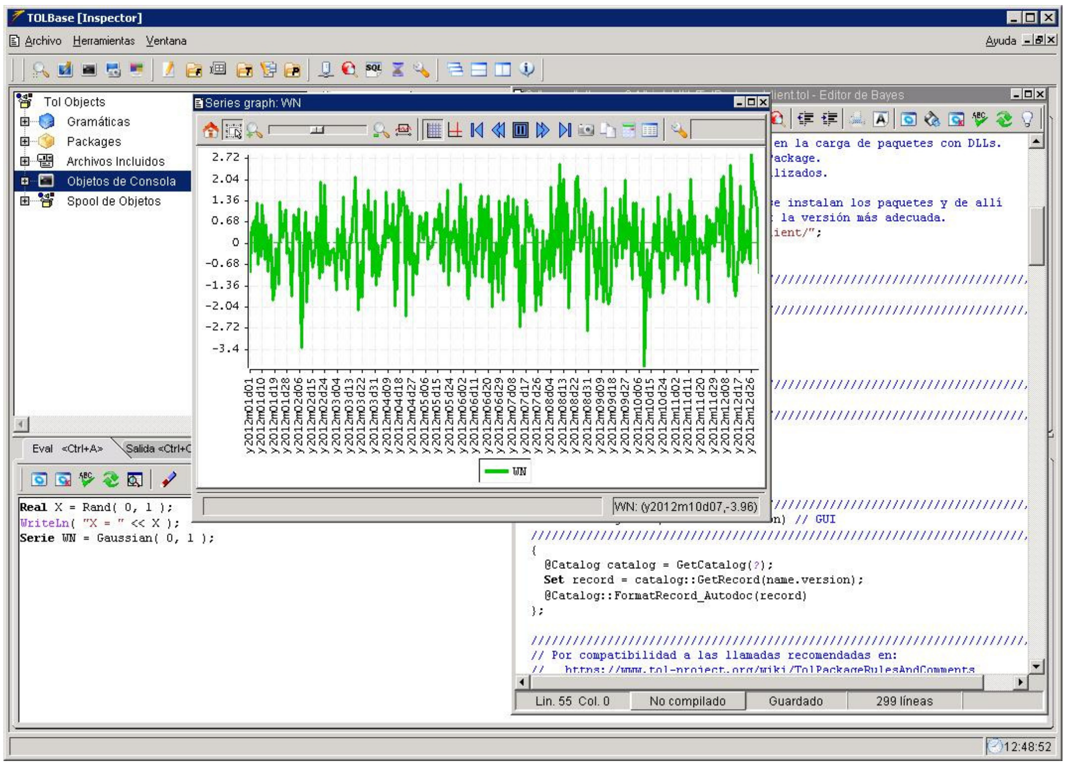
Figure 1.5.1: Main *TOLBase* window: object inspector.

The object tree has five main nodes. These nodes list *TOL* objects that simultaneously act as containers of other *TOL* objects. These main nodes are:

- Grammars: displays the set of data types implemented in *TOL*. There is a sub-tree for each data-type, which in turn contains the global variables created in the evaluation session.
- Packages: displays the *TOL* packages loaded in the *TOL* session. Below each package there is a sub-tree which displays package members. See section 3.4.
- Included Files: displays the .tol file-set included in the evaluation session. From each file, a sub-tree can be displayed which shows the objects created in the evaluation session of said file.
- Console Objects: contains the list of objects contained in the *console*, which are listed in the order they were created.
- Object Spool: Used to apply contextual menu options to a selection of objects. For example, to chart a group of time series, the objects have to belong to a common container. The*spool* is a virtual objects container whose purpose is to bring together the contents of distinct containers into a single one. Objects are inserted into the *spool* through a contextual menu option run on an already existing object.

TOLBase offers many other facilities for editing .tol programs and for visualising the different object types that can be created. These facilities will be explored further in following chapters, as we go through all the different types of data available.

Using just a sample in figure 1.5.2 we can see a *TOL* session in which 3 expressions have been evaluated in the*console*, where a time-series has been charted and a*.tol* file has been opened for editing.



```
Real X = Rand( 0, 1 );
WriteLn( "X = " << X );
Serie WN = Gaussian( 0, 1 );
```

```
en la carga de paquetes con DLLs.
ackage.
lizados.

e instalan los paquetes y de allí
 la versión más adecuada.
ient/";
```

```
n) // GUI
///////////////////////////////////////////////////////////////////////
{
  @Catalog catalog = GetCatalog(?);
  Set record = catalog::GetRecord(name.version);
  @Catalog::FormatRecord_Autodoc(record)
};

///////////////////////////////////////////////////////////////////////
// Por compatibilidad a las llamadas recomendadas en:
//   https://www.tol-project.org/wiki/TolPackageRulesAndComments
```

Figure 1.5.2: *TOLBase* showing some of its functionailty: the *TOL* code evaluation console, time-series charts and file-editing.

To run the sentence-set of sentences written in the *console* we can use theCompile button. The action associated with this button is the evaluation of all the expressions selected in the console, or all of the console code if there isn't an active selection. This action is also associated with the F9 key and is available in the contextual menu displayed on the *console*.

TOL prevents the creation of an object with a pre-existing name on the grounds that we wish to promote the continual evaluation of the piece of code we're trying to create. However, functionality does exist to destroy created objects, The Decompile button , available in the *console*, destroys objects created in previous evaluations which are accessible under the Console Objects node in the object inspector. This action is linked to the F8 key.

We may also find the syntax-check function very useful. It allows us to verify if the written *TOL* code follows the language's syntactical rules, without having to evaluate the code and the resulting creation of *TOL* objects. This option is available via the Syntax button and is associated with the F7 key. Sometimes, the error message that *TOL* emits is a syntactic error and it can be difficult to find out exactly where it is located. In such cases, one technique that may be of use is to apply the syntax-check to selected pieces of code until isolating a piece small enough for diagnosis

TOLBase keeps a history of evaluated expressions in an internal file in the *console*. This archive can be recovered using the action associated with the button Show history file , which opens a file-editing window which shows the historial of commands compiled on the *console* in reverse chronological order.

The editing window also offers previous functionalities, as well as the classic text-editor functions. In cases of edited files, evaluation results appear in the object inspector as a node beneath a node.Archivos Incluidos. This file node contains the sub-tree whose nodes are *TOL* objects resulting from the evaluation. For this reason, destroying the content of an evaluated file causes the deletion of memory belonging to this file and its descendents.

2 The *TOL* language

2.1 Basic notions

2.1.1 Syntax

As we can see in the Introduction, *TOL* is an interpreted programming language, and its code sentences can be run without any previous processing.

If using *TOLBase*, we have some added utilities at our disposal. These permit us to compile and decompile code sentences or .tol files, organise code or inspect variables. Nevertheless, in this section about syntax we'll stick to the use of *TOL* as a language regardless of the application.

In *TOL*, the term *compile* refers to the action of interpreting a particular code, carrying out the actions indicated and creating the stated variables.

Throughout this chapter, we will incorporate lines of code for illustrative purposes and indicate the output that the *TOL* interpreter produces upon their compilation.

As a rule, *TOL* doesn't return any output upon compiling code, leaving the output for notification messages, such as warnings and errors. To check the result of our actions we have to request a print-out or inspect them using the *TOL* graphic interface.

Grammars

TOL has a variety of data-types at its disposal. These are known as *grammars*, and they are used to construct code sentences. Practically every sentence in *TOL* returns one of these variables.

Grammars recognised by *TOL* are: `Anything`, `Real`, `Complex`, `Text`, `Set`, `Date`, `TimeSet`, `Serie`, `Polyn`, `Ratio`, `Matrix`, `VMatrix` and `NameBlock`.

Please bear in mind that the grammar `Anything` isn't strictly a type of variable; rather it is used to express any grammar in a general way.

Syntax

The basic syntax of *TOL* sentences is a grammar, the sentence code, as well as a full-stop and comma to finish.

```
<Grammar> <code...>;
```

In the examples of *TOL* code in this document we use the symbols <> (angle brackets/chevrons) to indicate that this part has been substituted by a valid expression. Bear in mind that these do not form part of the syntax.

Other characteristics of the language to be aware of are :

- It is case-sensitive.
- It allows lines to be split to make them easier to read. *TOL* sentences don't end with the line feed but with a full-stop and comma.(`;`).

Sentences automatically pass a syntax check prior to compilation. If the test is failed, compilation isn't carried out and an error message is shown to enable detection.

Code comments (// y /* */)

A good habit when writing code is to accompany it with notes to allow for easier understanding. They also help us to understand why a particular decision was taken.

TOL<t1/>, as other languages, has two ways of commenting on lines of code:

- A double forward-slash (//) to comment on a whole line or from where it is placed to the end of the line.
- The forward-slash/asterisk and asterisk/forward-slash combinations (/* y */). These allow us to close off the part of the code that isn't to be compiled, acting as if they were brackets. Commenting is possible in this way and can even be done several lines at a time.

<u>For example</u>

```
// The variable "sum" is created with the result 1+1
Real sum = 1+1;
```

2.1.2 Variables

As has already been hinted at, *TOL* programming strongly supports the creation of variables. To define a variable, we must allocate a value to it. *TOL* doesn't allow for variables to be created without this happening.

Allocation (=)

To create a variable we need to use the operator = (signo igual). The syntax is:

```
<Grammar> <name> = <value>;
```

<u>For example</u>

```
Real a = 1.5;
Text b = "Hello";
```

Although in *TOL* sentences always return a variable, we don't have to assign a name to it if we're not interested in using it.

<u>For example</u>

```
Real 1.5;
Text "Hello";
```

Variable names

Variable names in *TOL* con be built with any combination of alpha-numeric characters as long the first character isn't numeric.

Although *TOL* allows the use of the extended ASCII character set of our system (generally LATIN1), to choose names we recomend that you only use standard ASCII characters.

We now move on to take a look at the various different character sets, according to their possible uses:

Capital letters	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Lower case letters	abcdefghijklmnopqrstuvwxyz
Other characters considered alphabetic	_
Numbers	0123456789
Other characters considered numeric	#'.
Characters considered to be operators	!\"$%&()*+,-/:;<=>?@[\\]^`{|}~

Table 2.1.1: Classification of ASCII characters in *TOL*.

Capital letters	ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞŸŠŒŽ
Lower-case letters	àáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿšœž
Other characters considered alphabetic	ƒªµºß
Characters considered to be operators (currently ignored)	€‚„…†‡ˆ‰‹‘’“”•–—™› ¡¢£¤¥¦§¨©«¬–®¯°±²³´¶·¸¹»¼½¾¿×÷

Tabla 2.1.2: Classification of extra LATIN1 characters (ISO 8859-1) in *TOL*.

Although the criteria for choosing variable names can be quite varied, here we include a few suggestions as guidance:

- Begin names in lower-case, reserving names with capital letters for global variables;
- Use the style*CamelCase* to join words together (each new word begins with a capital letter). Alternatively, use a valid separator i.e. a full stop (.) or an underscore(_) ;
- Avoid using abbreviations whenever possible;
- Only use one language when making up names, preferably English or the local language. Also, take care with spelling and grammar.
- Once chosen, use selected criteria consistently.

Reallocation (:=)

The majority of variables in *TOL* allow for reallocation; that's to say that a change in their value. To do so, we must use the operator := (colon-followed by equals sign).

For example

```
Real a = 1;
Real a := 2; //  'a' is reallocated
```

If we use the allocation operator(=) it will result in an error:

```
Real a = 1;
Real a = 2;
```

```
ERROR: [1] Variable 'a' already defined as "a "

It has not been possible to create the variable "Real a".

ERROR: [2] Conflict between variables.
 Attempt was made to modify "a" using variable "a"
```

Allocation by value

Note that the allocation of the majority of grammars in *TOL* is by value, so that each allocation builds a new variable.

For example

```
Real a = 1;  // 'a'  is created with the value 1
Real b = a;  // 'b'  is created with the value of 'a' that is 1
Real a := 2; // The value of 'a' is changed to 2
Real b;       // but 'b' continues to value 1
```

2.1.3 Numbers (`Real`)

The grammar `Real` is a data-type that TOL has at its disposal for the management of all types of real numbers. It doesn't, therefore, distinguish between integers and floating point numbers.

For example

```
Real a = 1;
Real b = 0.25;
Real c = Sqrt(3);
Real d = -1/2;
```

Unknown value (?)

TOL has a special real value at its disposal. This is the unknown value denoted by use of a question mark (?).

```
Real n = ?;
```

It is also the value offered by some mathematical functions when the returned value doesn't exist or isn't a real number:

```
Real Sqrt(-1);
Real Log(0);
Real ASin(3);
Real 0/0;
```

Other real numbers

Some other special real values have been installed in TOL<t1/>, such as:

- The number is: `Real E (2.718281828459045)`
- Pi (π): `Real Pi (3.141592653589793)`
- Infinity: `Real Inf (1/0)`
- The value real true: Real True (1)
- The value real false: Real False (0)
- Elapsed real time: Real Time (variable value)

Operators and mathematical functions

As well the more familiar logical and artihmetic functions, TOL also has an assorted range of mathematical and statistical functions at its disposal. We will now take a look at some of these functions through the use of examples. For more detailed information, please consult the relevant documentation for these functions..

Examples:

```
// We generate a random number between 0 and 1000
Real x = Rand(0, 1000);
// We find its integer part and its decimal part:
Real n = Floor(x);
Real d = x-n;

// The following approximation is commonly used:
//   Log(1+x) ~ x
// when 'x' is small.
// We calculate the error of the approximation:
```

```
Real log_1_plus_d = Log(1+d);
Real relative_error = Abs(log_1_plus_d - d)/log_1_plus_d * 100; // en %
```

```
// Find my own version of pi
// taking advantage of pi/4 being 1
Real my_pi = 4 * ATan(1);
```

Complex numbers (`Complex`)

TOL also includes a data-type capable of managing complex numbers, namely the grammar Complex.

The declaration of a complex number is made in its binomic form as the sum of a real and an imaginary part. This is the product of a real number and an imaginary unit: `Complex i`.

For example

```
// A complex number can be created by indicating its real and imaginary parts:
Complex z = 3 - 2*i;
```

The following functions are available to us to recover the real and imaginary parts of a complex number. They can also be used to obtain the module and argument of its polar notation:

- The function `CReal` for the real part.
- The function `CImag` for the imaginary part.
- The function `CAbs` for the absolute value or complex number module.
- The function `CArg` for the argument or complex number phase.

For example

```
// Let z be a complex number:
Complex z = 3 + 4*i;
// The real part and imaginary parts can be obtained with:
Real z_re = CReal(z); // -> 3
Real z_im = CImag(z); // -> 4
// The module and argument (of the polar notation) with:
Real z_mod = CAbs(z); // -> 5
Real z_arg = CArg(z); // -> 0.9272952180016122
```

As well as conventional arithmetic operations, we can use the (virgulilla) operator. This allows us to obtain the conjugate of the number on which it is operating. For example:

```
Complex z = 3 + 2*i;
Complex u = z / CAbs(z); // (0.832050294337844)+i*(0.554700196225229)
Complex uC = ~u;         // (0.832050294337844)+i*(-0.554700196225229)
```

Note that some of the functions operating on Real and returning Real could return an unknown value. This could have an implementation whose return value could be a valid complex number. For example:

```
Complex Sqrt(-1);  // (0)+i*(1)
Complex Log(-0.5); // (-0.693147180559945)+i*(3.14159265358979)
```

2.1.4 Text strings (`Text`)

The grammar `Text` allows us to create variables with text strings of undefined length. Text strings must be enclosed between speech marks (`""`).

To place speech marks(`"`), as well as some other special characters, in the appropriate text string, the backslash(`\`) is used as an escape character.

Speech marks	" (ASCII 34)	`\"`	
Backslash	\ (ASCII 92)	`\\`	Where no ambiguity exists, simple is accepted.
Line feed	(ASCII 13)	`\n`	It is also possible to enter it explicitly.
Tabulation	(ASCII 9)	`\t`	It is also possible to enter it explicitly.

Table 2.1.3: escaped characters in *TOL* text strings.

For example

```
Text "The word \"subcontinental\" contains all five vowels";
Text "There are two letter-cases:\n * Capital letters.\n * Lower-case
letters.";
```

Note that the last string could also be written as:

```
Text "There are two letter-cases:
 * Capital letters.
 * Lower-case letters.";
```

Messages (`WriteLn`)

One function (very frequently used for programming) exists that allows messages to be written in the interpreter output. This function is especially peculiar in that it doesn't have a return value. It is available in two formats: `Write` y `WriteLn`. The use of the latter of these two is much more widespread, it adds a line-feed to the end of the message without having to explicitly include it.

```
WriteLn("Hello world");
```

```
Hello world
```

Operations with strings

We often wish to modify or edit a text string. Sometimes we even find strings that host certain information that we wish to obtain. To do this *TOL* has certain functions available to help us.

Here, we demonstrate some of them via means of an example. For more detailed information, please consult the relevant function documentation<t1>.

Example:

```
// We have a string that contains an attribute value
// which we wish to extract.
// The value which interests us can be found after the character ':'
Text string = "language:es";
// We locate the position of the character ':'
Real position = TextFind(string, ":");
// We determine the length of the string
Real length = TextLength(string);
// We obtain the value of the desired string
Text language = Sub(string, position+1, length);
//  Send the value just found to the output
WriteLn("Configuración de idioma: '"+language+"'.");
```

```
Language configuration: 'es'.
```

2.1.5 Functions (`Code`)

Functions in *TOL* are a new type of variables, whose grammar is `Code`.

The syntax to define a function is:

- The output's grammar. In *TOL*, every function should return a variable. If returning a value is either unwanted or unimportant, then returning a real number is recommended..
- The name of the function Éste operates under the same rules as el nombre in general. As with all other variables, the name is optional.
- Arguments. Arguments separated between commas are placed between brackets (`()`). For each argument, we must specify its grammar, along with the chosen function name. Every function has to receive at least one argument. If the function does not need an argument, it's recommended that a single `Real` argument is specified, whose name is not used within the function body.
- The body. The different sentences of function code are indicated between braces (curly brackets) (`{}`). The last line will be the function output. Occasionally it's not necessary to specify a type for the last line executed.(`;`).

```
<Grammar> [<Name>](<Grammar> <name>[, <Grammar> <name>, ...]) {
  ...
};
```

In the above expression, angular brackets (<>) signify that the code has had to be appropriately substituted. The square brackets (`[]`) indicate that the code is optional.

Examples:

```
// We create a function that returns the number of digits.
// of the whole part of a real number
Real IntegerPart_Length(Real number) {
  Real unsignedIntegerPart = Floor(Abs(number));
  // we determine the number of digits of the integer part.
  // making use of the fact that the numbers are in decimal base.
  Floor(Log10(unsignedIntegerPart))+1
};
```

```
// This function displays a greeting every time it's used.
// It doesn't need arguments and its output isn't of importance..
Real PrintHello(Real void) {
  WriteLn("Hello!");
1};
Real PrintHello(?); // //  The unknown value ? is used to emphasise that its
value isn't important DELETE THIS SEGMENT
```

```
Hello!
```

Note that the code indentation isn't part of the language's syntax but is a practical and clear way of writng functions.

Local scope ({})

Bear in mind that all the code contained between braces (curly brackets) (`{}`), such as that in the body of a function, is evaluated in a temporary scope. This is later eliminated (except for its output and final line), once the evaluation block is completed. In the block scope (local scope) access is give to global variables as it is to those of other superior levels. This is done in the same way as when accessing local variables, giving preference to the latter in the event of any uncertainty.

For example

```
Text scope = "Global";
```

```
Real value = {
  Text scope = "Local";
  WriteLn("Scope 1: "<<scope);
  Real phi = Rand(0, Pi);
  Sin(phi)^2 + Cos(phi)^2
}; // -> 1
WriteLn("Scope 2: "<<scope);
```

```
Scope 1: Local
Scope 2: Global
```

2.2 Sets

2.2.1 Sets (`Set`)

The grammar that allows us to create sets is particularly useful. In *TOL* the variable type `Set` represents the ordered set of other variables (including other sets). The set's elements can be from different grammars and can be named or unnamed.

Double square brackets (`[[ y ]]`) are used to create a set, or alternatively, a function that returns one.

For example:

```
Set digits = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]];
Set even_numbers = Range(2, 20, 2);
Set vowels = SetOfText("a", "e", "i", "o", "u");
```

Note that the syntax used for set creation continues being : grammar, name, equals sign and content. The term `Set` makes reference to the object type, the noun *set* and not to the verb *set*, which describes the action of allocation.

Simple square brackets are used to access the set elements (`[` and `]`) or the function `Element` indicating the index (or name if applicable) of the desired element.

For example:

```
Set colours = [["red", "green", "blue"]];
Text first_color = colors[1];
```

Basic elements

The most basic elements about sets are related with the capacity to count, increase and decrease their number of elements. Let's now list the functions that allow us to do this:

- The function `Card` allows us to know the size or number of the set, that is it say its number of elements
- The function `Concat` (as the operator `<<`) allows us to create a new set, connecting (by putting one after another) two or more sets together.
- The function `Append` allows us to add a new set of elements to a given set.
- The function `Remove` allows us to remove an element from the set.

For example:

```
// We create two sets, 's1' and 's2'
Set s1 = [["A", "E", "I", "O", "U"]];
Set s2 = [["Y", "W"]];
// We create a set 's' with the elements of 's1' and 's2'
Set s = s1 << s2; // equi
// We remove the last element of 's'
```

```
Real s_size = Card(s);
Set Remove(s, s_size);
// We add two new elements to 's'
Set Append(s, [["J", "K"]]);
Real s_size := Card(s);
WriteLn("Final size of 's': "<<s_size);
```

```
Final size of 's': 8
```

Empty set (`Empty`)

The syntax of double square brackets doesn't allow us to create an empty set. To do that, it's necessary to make a copy of the special set `Empty`.

```
Set my_elements = Copy(Empty);
```

Allocation by reference

Bear in mind that the allocation of sets is by reference. This is in contrast to real numbers or texts. This is to say that the new set isn't a copy, rather another reference or alias to access the said set.

For example

```
Set a = [[1, 2]];  //  'a' is created with the numbers 1 y 2
Set b = a;         // 'b' is created, which is a reference to said set 'a'
Set a := [[3, 4]]; // the content of 'a' is changed for the numbers 3 y 4
Real b[1];         //  The values of set 'b'  also changed, b[1] es 3
```

Copy (`Copy`)

To force the copy and creation of a new set, we can use the `Copy` function:

```
Set a = [[1, 2]];  //  'a' is created with the numbers 1 y 2
Set b = Copy(a);   // 'b' is created as a copy of set 'a'
Set a := [[3, 4]]; // is the content of 'a' is changed
Real b[1];         // set 'b' is maintained and b[1] is 1
```

Nevertheless, although the <t0/>Copy<t1/> function allows for the creation of a new set, the elements of each set are the same. If the elements of the original set change, the elements of the copied set also change:

```
Set a = [[1, 2]];  //  'a' is created with the numbers 1 y 2
Set b = Copy(a);   // 'b' is created as a copy of set 'a'
Real a[1] := 3;    // if the value of 'a[1] is changed'
Real b[1];         // the value of 'b[1]' also changes and is now 3
```

DeepCopy (`DeepCopy`)

To be able to make a complete copy of a set, namely one of the set and all of its elements, we can use the function `DeepCopy`:

```
Set a = [[1, 2]];     // 'a' is created with the numbers 1 y 2
Set b = DeepCopy(a); // 'b' is created as a complete copy of the set 'a'
Real a[1] := 3;       // if the value of 'a[1]' is changed
Real b[1];            // the value of 'b[1]' is maintained and continues to be
1
```

<u>Summary:</u>

In table 2.2.1 we can see a summary of the internal *TOL* process in cases such as the ones described above:

`Set a = [[1, 2]]`	`Real 1` is created. `Real 2` is created. Set `[[]]` is created and two real numbers are added. Reference '`a`' is created, which accesses the set.
`Set b = a;`	Reference '`b`' is created, which accesses the same set as '`a`'.
`Set c = Copy(a);`	Set `[[]]` is created, to which is added the elements of '`a`'. Reference '`b`' is created, which accesses the new set.
`Set d = DeepCopy(a);`	A new `Real 1<t1/> is created.'` A new`Real 2<t1/> is created'` The new set `[[]]` is created, to which these new real numbers are added. Reference '`b`' is created, which accesses the new set.

Table 2.2.1: Internal *TOL* process in set creation.

Set characteristics

The flexibility of *TOL* when it comes to defining sets allows us to use them for a number of very different purposes. We can identify different types of sets according to some of their characteristics. Taking these into account, the sets could be:

- **Ordered:** The elements of a set being programmed, by their very nature, are always in order. We use this characteristic to highlight the fact that the order of a set is important and affects its functionality.
- **Indexed:** This characteristic is closey related with the order and both of them can be used to locate an element in the set. For a set to be indexed, each of its elements have to have a unique name. The indexing process consists of building an index that associates element numbers with their positions and allows elements to be accessed by name.
- **Simple or without repeated elements:** Although when we talk of sets we think of a collection of distinct elements, it's true to say that one sometimes may contain repeated elements. We refer to a set as simple to highlight its absence of duplicate elements.
- **Homogeneous :** We say that a set is homogeneous when all of its elements are of the same nature or consist of the same grammar.

Algebra of sets

As well as elemental operations with sets, <t0/>TOL<t1/> includes basic algebraic set functions:

- The operator `+` for the union of sets.
- The operator `*` for the intersection of sets.
- The operator `–` for the difference of sets.
- The operator `<:` to check that an element belongs to a particular set.

Note that these operations are naturally defined for simple sets (without repeated elements). To obtain a free set of repeated elements, we can use the function `Unique`.

<u>Example:</u>

```
// Let's consider the following sets of characters
Set c1 = Characters("conjunto"); // (card: 8)
Set c2 = Characters("disjunto"); // (card: 8)
// We delete the repeated characters
Set s1 = Unique(c1); // an 'n' and an 'o' are removed (card: 6)
Set s2 = Unique(c2); // stay the same (card: 8)
// We operate with the two sets
Set union = s1 + s2;        // (card: 9)
Set intersection = s1 * s2; // (card: 5)
Set s1_minus_s2 = s1 - s2;  // (card: 1)
Set s2_minus_s1 = s2 - s1;  // (card: 3)
```

Indexed sets

The `SetIndexByName` function enables us to index sets and speed up access by name. All set elements to be indexed are required to have a name unique to the set.

As well as this function, we have other others available to us such as `HasIndexByName`, which allows us to determine if a set is indexed or not. Another, `FindIndexByName` allows us to find the index or position of an element in a set by name.

Remember that set elements can also be accessed by name using simple square brackets (`[ ]`).

<u>Example:</u>

```
// We create a set with values for each day of the week
// and we index the set:
Set weekdays = [[
  Real monday = 101;
  Real tuesday = 102;
  Real wednesday = 103;
  Real thursday = 104;
  Real friday = 105;
  Real saturday = 106;
  Real sunday = 107
]];
Real SetIndexByName(weekdays);
// We find the value for Friday
Real weekdays["friday"]; // -> 105
// We find the value for Saturday
Real sunday_index = FindIndexByName(weekdays, "sunday"); // -> 7
Real sunday_value = weekdays[sunday_index]; // -> 107
```

2.2.2 Control instructions

TOL contains some control functions and instructions that allow us to modify the flow of programme lines being run. We'll now take a look at some of the most important of these.:

Conditional instruction (`If`)

One of the most fundamental instructions of a language is `If`, which allows us to evaluate any given code according to the value of a certain condition.

In *TOL*, `If` is implemented as a three-argument function (the last of these being optional), with the following syntax.

```
Anything If(Real condition, Anything code_then[, Anything code_else])
```

The first argument is the condition, an expression that must return a true (1) or false (0) value. The two following arguments correspond with the code to be run in each case respectively.

The output of `If`, as when defining functions, corresponds with the last line of code run.

If the value of the condition is a number other than zero, it will be considered true, unless it has an unknown value. In this event, a warning will be displayed and a default value will be returned.

For example

```
Real If(3+4==6, 1, 0);
```

Note that if the third argument isn't indicated and the condition isn't verified, no output will be built. This isn't a problem unless we use the output of `If` e.g. by assigning it to a variable.

```
Real answer = If(3==4, 5);
```

```
ERROR: [1] answer couldn't be created.
```

Logical operations

To construct the condition value, we can make use of the following functions and logical operators:

- The function `And` and the operator `&` that return true only if all arguments are true.
- The function `Or` and the operator `|` that return true, when at least one argument is true.
- The function `Not` and the operator `!` that return the opposite of the value being applied.

And the following functions and comparison operators.

- The function `Eq` and the operator == that return true if arguments are equal.
- The function `NE` and the operator != that return true if arguments are not equal.
- The function `LT` and the operator < that return true when the first argument is smaller than the second.
- The function `GT` and the operator > that return true when the first argument is greater than the second.
- The function `LE` and the operator <= that return true when the first value is equal to or smaller than the second.
- The function `GE` and the operator >= that return true when the first argument is greater than or equal to the second.

When building logical constructions, it's worth remembering that when we negate an expression every boolean operator is substituted for its opposite. Pairs are: `And/Or`, `Eq/NE`, `LT/GE` y `GT/LE`.

For example

```
// Given three numbers:
Real a = Round(Rand(0, 1));
Real b = Rand(0, 6);
Real c = Rand(0, 9);
// and the following condition:
Real condition = ( a==1 & (b>3 | c<=5) );
// The opposite logical value to this:
Real Not(condition);
// can also be rewritten as:
Real a!=1 | (b<=3 & c>5);
```

Multiple conditional instruction (`Case`)

The `Case` function is an extension of the conditional control instruction for more than one case. It allows us to evaluate different codes as they verify one condition or another. Checks are carried out and evaluated sequentially. Once a condition is fulfilled the corresponding code is run and all other checks are interrupted.

The `Case` function always requires a pair of arguments. Arguments in odd positions return the condition's logical value i.e. true or false. In the case of arguments in even positions, code is run when they are fulfilled.

Syntax is:

```
Anything Case(Real condition1, Anything code1
  [, Real condition2, Anything code2 [, ...]]);
```

Conditional loop (`While`)

Another of *TOL*'s most fundamental control instructions is `While`, which allows us to run code cyclically whilst a condition is being verified.

The code's syntax is:

```
Anything While(Real condition, Anything code);
```

<u>Example:</u>

```
// A cycle is created to print out integer numbers.
// until a zero is found
// or until at least 100 numbers are displayed
Real print = 0;
Real rand_number = Floor(Rand(0, 100));
Real While(print<100 & rand_number!=0, {
  WriteLn(FormatReal(rand_number, "%2.0lf"));
  Real print := print + 1; // other has been printed
  "Another integer is generated."
});
```

Conditional arguments

Note that the control functions `If`, `Case` and `While` receive arguments that are lines of code which will only be executed if the control condition is satisfied.

Other useful functions to create code run cycles, are`For` and `EvalSet`. These functions do not, however, use *condition-code*.

Simple loop (`For`)

The function `For` allows us to evaluate an incremental function relating to a set of whole numbers. The function's syntax is:

```
Set For(Real begin, Real end, Code action);
```

The arguments `begin` and `end` are the two whole numbers that indicate the beginning and end of the cycle.

The third argument must be a function of a single `Real` type argument:

```
Anything (Real integer) { ... }
```

The difference between the implementation of this loop in *TOL* and other languages is that in *TOL* a function returns a set with as many elements as cycles; each element being the output of the corresponding call to the function indicated as the third argument..

<u>For example</u>

```
// The following code builds a set with five sets
// each one of these with a whole number and its square.
Set table = For(1, 5, Set (Real n) {
  [[n, n^2]]
});
```

Note the difference with the previous control instructions. The third argument of `For` is a function that could previously have been defined or, as is usual, defined inline. In order to understand this better, take a look at the following alternative to the above code:

```
Set function(Real n) { [[n, n^2]] };
Set table = For(1, 5, function);
```

Also note that the implementation of `For` in *TOL* as a function that returns a `Set` facilitates the construction of a data-set; one that in other languages is usally implemented in a way similar to that demonstrated below:

```
Set table = Copy(Empty);
Set For(1, 5, Real (Real n) {
  Set table := table << [[ [[n, n^2]] ]]
  1
});
```

Loop over the elements of a Set (`EvalSet`)

The `EvalSet` is another control instruction that allows to apply a function to all of a set's elements. As in the case of the `For`, the function returns a set with all of the function's answers when acting upon each one of its elements..

<u>Example:</u>

```
// We obtain a set with indexes in ASCII
// of the character is a phrase:
Set chars = Characters("Hola mundo");
Set asciis = EvalSet(chars, Real (Text t) { ASCII(t) });
```

Note that as `ASCII` is a function that (unambiguously) receives a text as a unique argument; which is the type of the elements of the set that can then be used as the second argument for EvalSet without having to define a new function::

```
Set asciis = EvalSet(Characters("Hola mundo"), ASCII);
```

The complementary function to `ASCII`, that returns the character starting from its ASCII number is `Char`.

<u>An idea:</u>

One way to widen the possibilities offered to us by the function `For` to another number set is to use an `EvalSet` together with a `Range` function:

```
Set countdown = EvalSet(Range(10, 0, -1), Real (Real n) {
  WriteLn(FormatReal(n));
  Sleep(1)
});
```

2.2.3 Set queries

In addition to the operations relating to the aforementioned sets in previous sections, *TOL* also contains functions for the selection, sorting and classification of sets.

Note that these functions allow us to carry out query actions relating to sets, simulating queries from other languages - such as the *SQL* command `SELECT` and the `ORDER BY` and `GROUP BY` clauses..

Selection (`Select`)

The `Select` function allows us to choose the elements of a set that verify a particular condition. The function receives the set and condition function to be applied to each of its elements for its selection and then returns the set with the selected elements. The function syntax is:

```
Set Select(Set set, Code condition_function);
```

Where the condition (`condition_function`) has to be a function with a single argument of the same type as the elements of the set (in general `Anything`), and whose output has to be a real value interpreted as a boolean value (0 means the element is not selected !=0 means the element is selected):

```
Real <True|False> (Anything element) { ... }
```

Example:

```
// We have a set of  normally distributed random numbers.
Set sample = For(1, 1000, Real (Real i) { Gaussian(0, 1) });
// we select those only greater than -1
Set subsample = Select(sample, Real (Real x) { x > (Real -1) });
```

Sorting (`Sort`)

The `Sort` function allows us to sort the elements of a set in a particular given order. The function's syntax is:

```
Set Sort(Set set, Code order_function);
```

When the sorting criteria (`order_function`) must be a two-argument function of the same type as the elements of the set (in general `Anything`), and whose output must be a real value that indicates which of the two has to come first: `-1` if it has to be the first argument, `1` if it has to be the second argument `0` or if the sorting criteria isn't important:

```
Real <-1|1|0> (Anything element1, Anything element2) { ... }
```

TOL `has a function called Compare` that allows for the checking and comparison of different data-types. However, we can define our sorting criteria as normal.

Examples:

```
// Being the set of the letters of the word "set"
Set chars = Characters("conjunto");
// We can place them in alphabetical order:
Set sorted = Sort(chars, Compare); // [["c","j","n","n","o","o","t","u"]]
```

```
// Being the set of name-value pairs:
Set pairs = [[
  [["alpha", 2.3]],
  [["beta", Real -0.5]],
  [["gamma", 0.4]],
  [["delta", 8]]
]];
// We can order them from greatest to smallest value in the following way:
Set Sort(pairs, Real (Set pair1, Set pair2) {
  (-1) * Compare(pair1[2], pair2[2])
});
```

Classification (`Classify`)

The `Classify` function allows us to group set elements together in nine sets (or classes), according to a particular equivalence relationship.

By default, the classify function (`Classify`) expect in the second argument an order relation in the same way as the (`Sort`) function, this will be used as an a equivalence relation, grouping in the same set all the elements for which the relation is evaluated to `0` when compared pairwise.

For example

```
// Being the set of the letters of the word "set"
Set chars = Characters("conjunto");
// If we group them together using the Compare function we will find 6 groups.
// Two of these (the 'o' group and the 'n' group) with two elements.
Set groups = Classify(chars, Compare);
Set EvalSet(groups, Real (Set group) {
  Text firstElement = group[1];
  Real size = Card(group);
  WriteLn("El grupo de las '"<<firstElement<<"' tiene "<<size<<" elementos");
  Real 0
});
```

```
The 'c'  group has 1 element.
The 'j'  group has 1 element.
The 'n"  group has 2 elements.
The 'o' group has 2 elements.
The 't' group has 1 element.
The 'u' group has 1 element.
```

The function gives us the option of indicating the type of relationship used for classification. The complete syntax of the function is:

```
Set Classify(Set set, Code function[, Text relationType="partial order"])
```

For example, if we wish to sort a set in the most natural possible way, using the equivalence relationship that tell us if two elements are equal or not, we indicate `"equivalence"` as the third argument. For example

```
// Let's consider a set of natural random numbers:
Set numbers = For(1, 100, Real (Real i) { Floor(Rand(100, 1000)) });
// We classify it by group according to its last number:
Set groups = Classify(numbers, Real (Real number1, Real number2) {
  Real unit1 = number1 % 10; // last digit of the first number
  Real unit2 = number2 % 10; // last digit of the second number
  unit1==unit2
}, "equivalence");
```

2.2.4 Structures (`Struct`)

Although a set can consist of any type of element at all for without any particular reason for it being there, we can single out 2 types for their content:

- Container type sets: those which contain an undefined number of elements of a homogenous nature. For example, *containers* are a set of study units, the set of open data-base connections. or one of these simple sets:

```
// Odd numbers smaller than 100
Set container1 = Range(1, 99, 2);
// Lower-case ASCII lettes
Set container2 = For(97, 122, Text (Real ascii) { Char(ascii) });
```

- Entity sets: these contain a defined number of elements. These are commonly in order, with a specific meaning but not necessarily homogenous. They represent the unit of a particular concept. Normally, multiple elements can exist or be created with the same characteristics or structure. Examples of *entities* would be the coordinates of a matrix element (a row-column pair), function arguments or some of these simple sets :

```
// A substitution rule: the first text will be substituted for the second.
Set entity1 = [["á", "a"]];
// Function characteristics: name, output, number of arguments
Set entity2 = [["Char", "Text", 1]];
```

Structures

It is highly desirable that entity sets follow a predetermined stucture, in a way that leaves no doubt as to the meaning of each element and facilitates its use.

In *TOL*, the concept of set structure exists under a special definition named `Struct`. The structures (namely of type `Struct`) are a type of information that perform the role of a specialised `Set` type grammar. They also allow for the construction of structured sets.

Definition of structures

The definition of a structure contains both the total number and type of elements that each set that we create will have. As well as having the element type in the definition of the structure, we assign it a name with which it can be clearly identified.

To define a new structure, we have to outline the name and grammar of each field using the following syntax:

```
Struct @<StructName> {
  <Grammar1> <Field1>;
  <Grammar2> <Field2>;
  ...
};
```

where the code appears between angle brackets (<>) it's necessary to substitute it for its corresponding value.

Example:

```
// We define @Vector3D to represent tridemensional vectors:
Struct @Vector3D {
  Real X;
  Real Y;
  Real Z
};
```

Note that the names of these structures have to begin with the character `@` (at). In this way they are clearly distinguished from other variables; as on occasions they are syntactically processed in a way that has more in common with a grammar than with a variable.

Structured sets

We refer to the objects created according to the definition of a structure as *structured sets*. Being sets, they are subject to the same processing as all others (`Set` variables).

To create a structured set we use the name of the structure as if it was a function that had as many arguments as the structure has elements:

```
Set  set = @<StructName>(<value1>, <value2>, ...);
```

For example

If to create a set with three real numbers we use;

```
Set set = [[1, 2, 3]];
```

To create a structured set of type `@Vector3D`, we have to do:

```
Set vector3D = @Vector3D(1, 2, 3);
```

We can also add a structure to a pre-existing set that agrees with the definition of the structure (in both the number and type of its elements), using the function `PutStructure`. For example

```
Set set = [[1, 2, 3]];
Set PutStructure("@Vector3D", set);
```

Access to elements (->)

The structuring of entity sets serves a number of purposes. It reinforces the character of this type of set, it ensures a certain amount of coherence between the number and type of its elements, and it facilitates access by field-name.

To access elements by field-name we use the arrow operator (->) with the syntax:

```
<Grammar> element = structuredSet-><FieldName>;
```

or the function `Field` equivalent:

```
<Grammar> element = Field(structuredSet, "<FieldName>");
```

For example

```
Set vector = @Vector3D(4, 3, -2);
Real coordZ = vector->Z; // es equivalente a vector[3]
```

Note that the name of an element doesn't need to correspond to its name in the structure:

```
Real a = 5;
Set vector = @Vector3D(a, 1-a, 0);
WriteLn("El nombre de la coordenada X es: '"<<Name(vector->X)<<"'");
```

```
The name of the coordinate X is: 'a'
```

Information about structures

To find out which are the defined fields in a structure we can use the `StructFields` function, which returns a set with the characteristics of each element. For example

```
Set StructFields("@Vector3D");
```

Note that this function receives a text with the name of the structure as an argument.

To find out a set's name, and whether it is structured or not, we can use the function `StructName`.

2.3 Statistics

TOL is a programming language strongly geared towards statistics. As such, it includes a wide variety of tools for statistical data analysis, both from a descriptive and inferential perspective.

2.3.1 Descriptive statistics

To compute statistics over set of reals numbers, *TOL* has a set of functions that allow an undefined number of real arguments.

```
Real <Function>(Real x1, Real x2, ...);
```

In addition to the arithmetic functions `Sum` and `Prod`, that add and multiply real numbers respectively, it's worth highlighting the following functions that allow us to collect data:

- Averages such as the mean (or arithmetic mean) (`Avr`), the geometric average (`GeometricAvr`) or the harmonic average (`HarmonicAvr`).
- Dispersion methods such as variance (`Var`) or standard deviation (`StDs`).
- Data-distribution methods: the values maximum (`Max`) and minimum (`Min`), median (`Median` or a quantile with any `p` probability:

```
Real Quantile(Real p, Real x1, Real x2, ...);
```

- Other methods related to moment distribution such as asymmetry coefficient(`Asymmetry`) or the Kurtosis coefficient (`Kurtosis`), and in a more general way any order moment `n`, centered (`CenterMoment`) or not (`Moment`):

```
Real Moment(Real n, Real x1, Real x2, ...);
Real CenterMoment(Real n, Real x1, Real x2, ...);
```

Missing data (?)

Bear in mind that real data with an unknown value (`?`) is treated as *missing data* by all these statistical functions. This means that this data isn't taken into account, or is disregarded from the calculattion.

For example

```
Real Sum(2, 3, ?, 4, ?); // -> 9
Real Avr(2, 3, ?, 4, ?); // -> 3
```

Statistics about sets

These same statistics can be obtained about sets of real numbers. To do so, use the corresponding functions for sets called `Set<Statistic>`.

For example

```
Set values = SetOfReal(1.0, -0.3, 0, 2.1,0.9);
Real mu = SetAvr(values);
Real sigma2 = SetVar(values);
```

2.3.2 Probability

In probability theory, a random variable represents a variable whose values correspond with the fulfillment of (non-deterministic) random Stochastic phenomena. These values could be, for example, the values of an experiment which hasn't yet been carried out, or those of a currently existing but unknown value.

Probability distributions

Although the values of a random variable remain undetermined, we can still find out the probability associated with the occurrence of any given value. The relationship that assigns probability to different values is known as *probability distribution*.

Random variables, as well as their probabilty distributions can be discrete or constant. This is dependent on whether their values are restricted to a finite (or infinitely countable) set or not.

Discrete distributions

Each one of the different values that a discrete random variable can adopt has a certain associated probabilty. The function that returns this probability is known as *probability function*.

TOL includes the probability functions of the main discrete distributions, with the following naming convention`Prob<Distribution>`:

- Binomial distribution: `ProbBinomial`.
- Negative binomial distribution: `ProbNegBinomial`.
- Poisson distribution: `ProbPoisson`.
- Geometric distribution: `ProbGeometric`.
- Hypergeometric distribution: `ProbHyperG`.
- Discrete uniform distribution: `ProbDiscreteUniform`.

Similarly, we can obtain the corresponding *distribution function*. This is defined as the probability of the variable taking a value more or less equal than a given, following the nomenclature: `Dist<Distribution>`.

The inverse of the distribution function. It allows us to find the value for which the accumulated probability is a given: It's implemented as: `Dist<Distribution>Inv`.

Constant distributions

The infinite possible values of a constant random variable don't have an associated probability, rather a probability density that allows us to determine the probability of a value found in a particular interval.

The function known as *density function* is implemented in *TOL* as`Dens<Distribution>` for the main distributions of constant probability. The following of these are of particular interest:

- Chi-squared distribution: `DensChi`.
- Exponential distribution: `DensExp`.
- T distribution of Student: `DensT`.
- Normal distribution: `DensNormal`.
- Log-normal distribution: `DensLogNormal`.
- Gamma distribution: `DensGamma`.
- Beta distribution: `DensBeta`.
- F distribution of Snedecor: `DensF`.
- Uniform distribution (constant): `DensUniform`.

As was the case with discrete distributions, *TOL* also offers the corresponding distriubtion functions, and their inverses, with the nomenclature: `Dist<Distribution>` and `Dist<Distribution>Inv` respectively.

Random numbers

To allow the sampling of random variables whose probability distribution is known, *TOL* includes functions generated by random numbers.

For example, the `Rand` function allows us to generate a random number from the uniform distribution for a given interval.

A pseudo-random number generator, which relies on a seed value, is used to generate these sequences of random numbers. The seed value, which runs upon starting a *TOL* session, can be consulted or modified via the functions `GetRandomSeed` and `PutRandomSeed` respectively.

Another very commonly used function to obtain random numbers is the `Gaussian` function, that allows us to obtain realizaciones of a normal o Gaussian distribution.

In addition to the two functions we've already touched on, *TOL* has functions at its disposal to show common probability distributions. These include chi squared (`RandChisq`), exponential distribution, (`RandExp`), gamma distribution (`RandGamma`) and log-normal (`RandLogNormal`).

2.3.3 Matrices (`Matrix`

Although *TOL* generally allows us to work with any real-number sets, it introduces a new type of variable, `Matrix`, which represents the matrices and bidimensional sets of real numbers.

Matrices can be built as a number table following the syntax outlined below:

```
Matrix <m> = ((<m11>, <m12>, ...), (<m21>, <m22>, ...), ...);
```

<u>For example</u>

```
Matrix a = ((1, 2, -3), (4, -5, 6));
```

Bear in mind that all rows have to have the same number of columns.

We can also build row matrices (of a single row) or column matrices (of a single column) using the `Row` and `Col` functions respectively.

<u>For example</u>

```
Matrix row = Row(1, 2, -3);
Matrix column = Col(1, 4);
```

Matrix elements

The number of rows and columns can be obtained via the `Rows` and `Columns` functions respectively.

To access matrix elements we can use the `MatDat` function, which returns the required element value upon us indicating the relevant row and column. To modify the value of an element, we use the `PutMatDat` function, indicating the new value as a third argument.

Matrix composition

TOL includes two operations that allow us to form matrices linking matrices by rows or columns:

- The operation `<<` (and the `ConcatRows` function) to link matrices by rows.
- The operator `|` (and the`ConcaColumns` function) to link matrices by columns.

As is logical, to link by row, matrices have to have the same number of columns. Similarly, to link by columns, matrices must have the same number of rows.

For example

```
Matrix a = ((1, 2), (4, 5));
Matrix b = Col(3, 6);
Matrix c = Row(7, 8, 9);
Matrix (a | b) << c;
```

In the same way that we can combine matrices to compose a larger matrix, we can obtain submatrices from any given matrix. This is outlined in the following functions:

- The`SubRow` function allows us to form a matrix consisting of a selection of rows from another matrix. It receives the original matrix and an index set as an argument.
- The `SubCol` function obtains a matrix from a column selection.
- The `Sub` function allows us to form a submatrix by indicating the row and column of the first element, and the height (number of rows) and width (number of columns) of the submatrix.

For example

```
Matrix a = ((1, 2), (3, 4), (5, 6)),
Matrix SubRow(a, [[1, 3]]); // -> ((1, 2), (5, 6))
```

Operations with matrices

As well as the arithmetic operators, `+` and `−` that allow us n to add and subtract 2 matrices, or one matrix and a real, *TOL* has the operator`*` to carry out matrix multiplication.

Remember that matrix multiplication can take place between two rectangular matrices when the number of columns in the first coincides with the number of rows of the second.

Other characteristic functions of the matrices installed in *TOL* are:

- Matrix transposition: `Tra`.
- Matrix inversion using the Guass inversion method: `GaussInverse`.
- Cholesky factorisation for symettrical and positive matrices. The `Choleski` function gives us the inferior triangular matrix (or Cholesky triangle) of the decomposition.

Element by element operations

To carry out element-by-element matrix multiplication (known as Hadamard's product) or the element-by-element quotient:

- The operator `$*` (or the `WeightProd` function) for the product.
- The operator `$/` (or the function `WeightQuotient`) for the quotient.

Note that the operator ^, works on matrices in a way consistent with Hadamard's product, acting individually on each element. This is true with all sets of mathematical functions to do with real numbers (exponential, trigonometrical, hyperbolic, etc)

It's true that the larger part of mathematical functions to do with real numbers have a matrix-based version, in general we can apply any function to all of a matrix's element using the `EvalMat` instruction, which is similar to (`EvalSet`). This instruction goes through all of the matrix's rows and columns.

For example

```
// We apply the function f(x) = x * Exp(x)
// to a matrix's elements, in two different ways:
Matrix a = ((1, 2, -5), (3, 0, 7));
Matrix b_1 = a $* Exp(a);
Matrix b_2 = EvalMat(a, Real (Real x) { x * Exp(x) });
```

We also have a conditional instruction for matrices, `IfMat` that allows us to build matrices with a conditional structure.

For example

```
// We create a matrix by applying a logarithm to a pre-existing one
// and returning 0 if the element if less than or equal to 1.
Matrix a = ((1, 2, -5), (3, 0, 7));
Matrix b = IfMat(GT(a, 1), Log(a), 0);
```

Statistics about matrices

The statistics introduced in the section 2.3.1 can be obtained for matrices by using the corresponding functions with the name`Mat<Statistic>`.

For example

```
Matrix a = Gaussian(100, 1, 0, 0.5);
Real a_mu = MatAvr(a);
Real a_sigma = MatStDs(a);
```

2.3.4 Linear models

Statistical models allow us to describe a random variable as a function of other variables. This relationship isn't deterministic but rather stochastic, in the sense that the model is nothing more than an approximate description of the mechanism generated by observations.

Linear regression (`LinReg`)

The simplest model that we can use to describe a random variable is that which can be explained as a linear combination of other variables, plus a purely random component or white noise.

$$Y = \sum_i X_i \beta_i + E$$

Where Y is the observed variable that we wish to be described as (referred to as *output* of the model), X_i are the explanatory variables (or *inputs* of the model), β_i are the parameters and E is the unexplained part or model error.

Generally speaking, parameters that describe the relationship between variables are not known and need to be estimated using the observations available from the variables:

In the case of the previous linear model, we only need to solve the corresponding linear regression to obtain a parameter estimate. For this, *TOL* has the `LinReg` function, which solves the expressed model using matrices.

For example

```
// We build a sample of random variable Y
// as the sum of two variables X1 y X2 + noise E
Matrix X1 = Rand(10, 1, 0, 10);
Matrix X2 = Rand(10, 1, 2, 5);
Matrix E = Gaussian(10, 1, 0, 0.1);
Real beta1 = 0.5;
Real beta2 = -0.3;
Matrix Y = X1*beta1 + X2*beta2 + E;
// We estimate the model: Y = X1*beta1 + X2*beta2
Set estimation = LinReg(Y, X1|X2);
Real estimated_beta1 = estimation["ParInf"][1]->Value;
Real estimated_beta2 = estimation["ParInf"][2]->Value;
```

Generalised linear models (`Logit`, `Probit`)

In the analysis of random variables we encounter some of a discrete nature that can't be modelled naturally with a linear model of the type described above.

Nevertheless, when the values of this variable can be made to correspond with those of some particular distributions, they can be rewritten with the help of a link function (*link*). This is what is known as a generalised linear model.

$$\mathrm{E}(Y) = link^{-1}\left(\sum_i X_i\beta_i\right)$$

Where $\mathrm{E}(Y)$ expresses the expectation of the random variable Y.

When the random variable only takes two different values (dichotomous variable or Bernouilli), we have two link functions (with domain in (0,1) and image of the whole real line) for the construction of the generalised model:

- The *logit* function which is the inverse of the logistical distribution function.
- The *probit* function which is the inverse of the normal distribution function.

TOL has two maximum likelihood estimators `Logit` and`Probit` respectively, used for the resolution of generalised linear models *logit* y *probit*. It has a similar syntax to `LinReg`..

```
// We create a dichotomous variable Y from a probability pY.
// built using the normal distribution function
// a linear combination of X1 and X2
Matrix X1 = Rand(500, 1, 0, 5);
Matrix X2 = Round(Rand(500, 1, 0, 1));
Real beta1 = 0.8;
Real beta2 = -0.5;
Matrix pY = EvalMat(X1*beta1 + X2*beta2, Real (Real x) { DistNormal(x) });
Matrix Y = EvalMat(pY, Real (Real x) { Real Rand(0,1)<x });
// We estimate the probit model:  E(Y) = Probit(X1*beta1 + x2*beta2)
Set estimation = Probit(Y, X1|X2);
Real estimated_beta1 = MatDat(estimation[1], 1, 1);
Real estimated_beta2 = MatDat(estimation[1], 2, 1);
```

Configuration

The aforementioned estimators (`Probit` and `Logit`) make use of certain configuration variables, created as global variables.

- The maximum number of iterations for iterative processes: `Real MaxIter`.
- Tolerance of numerical methods: `Real Tolerance`.

2.3.5 Virtual matrices (`VMatrix`)

Virtual matrices, grammar `VMatrix`, are a new type of data used for the declaration of matrices. They encapsulate the processing of special matrices that can't be handled in an efficient way with the `Matrix` type, allowing specialised internal polymorphic formats for various typesof matricial structures.

Virtual matrices encompass various sub-types related to the following concepts:

- The engine(*engine*): Each calculation engine requires its own data-types *ad-hoc* to get the most out of its algorithms. An attempt has been made to include the main systems of matricial algebra to deal with the most commonly occuring problems with regard to dense, sparse (or *sparse*) and structured (Toeplitz, Vandermonde, etc.) matrices. This even extends to the ability to define matrices as generic linear operators. The engines for which the virtual matrix interface currently exist are: *BLAS&LAPACK* y *CHOLMOD*.
- Cell-type(*cell*): Only the cell-type `Real` has initially been implemented with double precision (64 bits) but it can be expanded to simple precision (32 bits) and high precision (80 bits) where packages allow.
- Storage mode (*store*): Each calculation engine offers different ways to store data that define a matrix corresponding to its internal structure and that type of algorithms that will be run on it.

The operations possible with virtual matrices depend on each sub-type, which complicates their use. This is somewhat compensated by the fact that access is available to highly efficent and specialised methods.

Here we take a closer look at a few types of virtual matrices.

Dense matrices (`Blas.R.Dense`)

The matrix type `Blas.R.Dense` that is included in the basic data-type of *BLAS&LAPACK* in the native format of *FORTRAN*. Here, the cells of each column are consecutive, in contrast to the format by row considered as native in *C/C++* and using the type `Matrix`.

Sparse matrices (`Cholmod.R.Sparse`)

Sparse matrices (*sparse*) are used for solving large systems of linear equations, in which the matrices are largely formed of zeros.

The virtual matrix `Cholmod.R.Sparse` is the basic type of sparse matrix *CHOLMOD* and offers a wide range of implemented operations very efficiently.

Other more specific types introduced together with this are:

- The type`Cholmod.R.Factor`: a specialisation used to store the Cholseky factorisation of a sparse matrix in an especially effective way, to solve the linear systems associated with the said decomposition. Take a look at the descriptions of the `CholeskiFactor` and `CholeskiSolve` functions.
- Type`Cholmod.R.Triplet`: which is applied for external storage purposes and interfaces with other systems. They are treated as file-column-value groups that link the value of the none-void cell to the corresponding row and column number, one which combines an important memory saving with sufficiently sparse matrices. Let's take a look at the `Triplet` and`VMat2Triplet` functions.

Operations with virtual matrices

Virtual matrices, `VMatrix`, are implemented with a functionality similar to that of matrices, in that we have similar sets of functions available to us. (`Matrix`):

- Access and editing of cells (with `VMatDat` y `PutVMatDat`).
- Operations with sub-matrices (for example: `SubRow` o `ConcatColumns`).
- Arithmetic operators (such as matrix multiplication `*` or Hadamard's product `$*`).
- Mathematical and logical functions (such as `Log`, `Floor` or `LT`).
- Statistics about the matrix (such as `VMatAvr` or `VMatMax`).

Note that in the functions in which it appears, the term `Mat` (from the functions of the `Matrix` grammar) is substituted by `VMat`.

Creation of virtual matrices

The use of virtual matrices is linked to the use of particular functionalities, which are used as input or output arguments.

To enable the consturction of virtual matrices we have a number of basic toold available to us. These include `Constant` or `Zeros` for matrices that consist of identical cells, `Eye` or `Diag` for diagonal matrices, or `Rand` and `Gaussian` for matrices with random numbers.

We can also use the following mechanisms for conversion between matrices:

- From matrices to virtual matrices: `Mat2VMat`.
- From virtual matrices to matrices: `VMat2Mat`.
- Between differing types of virtual matrices: `Convert`.

2.4 Time variables

2.4.1 Dates (`Date`)

One of the main characteristics that distinguishes *TOL* from other interpreted mathematical languages is its tools for time structures. For this very reason that the name *Time Oriented Language* was chosen in the first place.

The fact that we have left the presentation of *TOL*'s time variables until this far into the manual, doesn't take anything away from their importance. In fact, the opposite is true, as we need to dedicate their own section to them, in order to provide a thorough explanation.

The main grammar for the management of time structures is no other than, `Date`, which is used for representing time intervals and instants.

Dates in *TOL* are expressed in the following way:

```
Date and<year>[m<month:01>[d<day:01>]];
```

Where the code appears between angle brackets (<>), it has to be substituted by the corresponding numeric values, square brackets (`[]`) indicate optional code and a semi-colon (`:`) precedes values by default. The year must consist of four digits, while the month and the day are indicated by two.

Examples:

```
Date y1992m07d25; // -> 25/07/1992
Date y2000;       // -> 01/01/2000
Date y2007m03;    // -> 01/03/2007
```

Although it's more common to use dates to indicate days or larger time intervals (such as weeks, months or years), *TOL* allows us to state small fractions of the day, indicating the hour, minute or second of the instant. The pattern is completed as follows:

```
Date y<year>[m<month:01>[d<day:01>[h<hour:00>[i<minute:00>[s<second:00>]]]]];
```

For example

```
Date y2011m11d11;           // -> 11/11/2011 00:00:00
Date y2000m02d04h06i08s10; // -> 04/02/2000 06:08:10
Date y2012m09d10h13;        // -> 10/09/2012 13:00:00
```

Dates and numbers

Dates can also be built explicitly from the numbers defined via `YMD` function, which can optionally take up to 6 arguments. These being the desired year, month, day, hour, minute and second.

For example

```
Date YMD(2011, 11, 11);           // -> 11/11/2011 00:00:00
Date YMD(2000, 2, 4, 6, 8, 10); // -> 04/02/2000 06:08:10
Date YMD(2012, 9, 10, 13);        // -> 10/09/2012 13:00:00
```

Inversely, we can obtain these numbers from the `Year`, `Month`, `Day`, `Hour`, `Minute` and `Second` functions. These return the year, month, day, hour, minute and second of the selected date respectively.

Another function of this kind, which is especially useful, is `WeekDay` which returns the day of the week with the corresponding number from 1 to 7 (with 1 being Monday).

Special dates

TOL has certain special dates pre-installed:

- Today's date: `Date Today (valor variable).`
- The current instant: `Date Now (variable value).`
- Beginning date: `Date TheBegin`.
- End date: `Date TheEnd`.
- Unknown date: `Date UnknownDate`.

Note that while the `Now` variable also displays the hour, minute and second of the current instant `Today` only returns the year, month and day (time 00:00:00).

Time index

Each date value has an associated number which depends on the number of days elapsed since a given date, with the value 1 being ascribed to the first day of the year 1900.

The `DateToIndex` and `IndexToDate` functions allow us to change the date to a number or the number to a date respectively.

For example

```
Real DateToIndex(y2000);    // -> 36525
Real IndexToDate(41161);    // -> y2012m09d10
Date IndexToDate(50000.5); // -> y2036m11d22h12
```

Note that the numeric value that *Microsoft Excel* ascribes to dates is similar apart from the fact that *Excel* erroneously considers 1900 to be a leap year:

```
Real excel_time = DateToIndex(<date>) + 1; // from March 1900
```

Other systems for numbering time such as *Unix_time* (which evaluates all the seconds elapsed since 1970), can easily be obtained from the TOL index via an arithmetic operation.

```
Real unix_time = ( DateToIndex(<date>) - DateToIndex(y1970) ) * 86400;
```

2.4.2 Dated (`TimeSet`)

With dates, we often state not just an instant but all of the time interval until the following instant. That is to say that the day 1st of January 2012 (`y2012`) can represent the first day of that year or the entire year.

To provide this added value to a date we must accompany it with a time-set (daily or annual for the previous example), which allows us to know the following date.

In *TOL* there is a special data-type called `TimeSet` that represents a date/time-set. It acts as support or a time domain, allowing a date to be understood as an interval.

Predefined time-sets.

TOL includes a set of pre-defined time-sets, which are very useful in this time domain function. Out of those, we bring your attention to the following:

- Annual time-set, formed of all the first days of the year: `TimeSet Yearly`.
- Monthly time-set, formed of all the first days of the month: `TimeSet Monthly`.
- Daily time-set, formed of all days `TimeSet Daily`.
- Weekly time-set, formed of all Mondays: `TimeSet Weekly`.
- Quarterly time-set, `TimeSet Quarterly`.
- Half-yearly time-set`TimeSet HalfYearly`.

Operations with dates related to time-sets

There are two very useful functions to be used with dates that allow for the time-set to be used as a time domain.

- The successor function of a date: `Succ`, which allows us to find a date a number of times subsequent to another particular time-set.
- The date difference function: `DateDif`, which allows us to determine the time distance between two dates in a time-set.

Example:

```
// The day before the 1st of March 2012 is:
Date prev_Mar.2012 = Succ(y2012m03, Daily, -1); // -> y2012m02d29
// as the year 2012 is a leap year.
// The number of days in the first quarter of 2012 is:
Real DateDif(Daily, y2012, Succ(y2012, Quarterly, 1)); // -> 91
```

Time-sets

A time-set can also be used to represent an arbitrary set of dates, a selection of all the dates in another time-set.

One example installed in *TOL* is (`Easter`), which is notable for the fact it is a recurring holiday with moving dates. This set represents all of the Sunday's included in Easter (according to western Christianity).

With this criteria in mind, *TOL* includes two special time-sets:

- The set of all days: `TimeSet C` (which coincides with the daily time-set).
- The set without any days: `TimeSet W`.

Creation of time-sets

Whilst it's true that the set of predefined time-sets in *TOL* is sufficient in its time domain function, we still require greater flexibility when it comes to date-sets. For this reason, *TOL* includes a set of functions that allows us to define new schedules:

- The `Y`, `M`, `D`, `H`, `Mi` and `S` functions allow us to obtain time sub-sets. These are formed by the dates of a year, month, hour, minute and second respectively.
- The `WD` function allows us to create a sub-set of dates according to the day of the week.
- The `Day` function allows us to create a time-set with a given date, while the `DatesOfSet` function does the same with all of the dates in a set.

Time algebra

To make it easier to build time-sets, *TOL* also includes algebra set operations:

- The operator `+` and the `Union` function that allow us to join time-sets.
- The operator `*` and the `Intersection` function that allow us to obtain the intersection of time-sets.
- The operator `–` that allows us to subtract all of the dates of one time-set from another.
- The `Belong` function that allows us to check whether a date belongs to a time-set or not.

Examples:

We can replicate the trimestral time-set, in order to illustrate the functions listed above.

```
TimeSet myQuarterly = D(1)*(M(1)+M(4)+M(7)+M(10));
```

To build a time-set with the days Monday to Friday:

```
TimeSet mondayToFriday = Daily-WD(6)-WD(7);
```

Or create a time-set with the first Sundays of each month, which, it goes without saying, are found in the first 7 days of every month.:

```
TimeSet firstSundays = (D(1)+D(2)+D(3)+D(4)+D(5)+D(6)+D(7))*WD(7);
```

The interval function (`In`)

The `In` function allows us to restrict a time-set to a date-set, for example:

```
// All months between the years 2000 and 2012 (inclusive)
TimeSet In(y2000, y2012m12, Monthly);
```

Periodic function (`Periodic`)

The `Periodic` function allows us to obtain a periodic time subset (with a whole period), from one of the dates of the subset.

For example, an alternative way to obtain the set of all Mondays, is to build a time subset of the daily schedule with period 7:

```
// The dare used, 10/09/2012 is Monday
TimeSet mondays_alternative = Periodic(y2012m09d10, 7, Daily);
```

The successor function (`Succ`)

The `Succ` function allows us to transfer particular number of units inside another time-set to a set of dates.

For example, to obtain the set formed of all last days of the month, which can vary from one month to the next, we can seperate the set of first days of the month and move them a day back.

```
TimeSet lastDaysOfMonth = Succ(D(1), -1, Daily);
```

Note that the successor function for time-sets receives the second and third argument in an order that differs from the successor function for dates:

```
Date Succ(Date date, TimeSet dating, Real integer);
TimeSet Succ(TimeSet timeSet, Real integer, TimeSet dating);
```

The range function `Range` is an extension of the successor function (`Succ`). It allows us to create a time-set with an entire interval of movements. For example

```
// Set of the last three days of the month
TimeSet Range(D(1), -3, -1, Daily);
```

2.4.3 Time series (`Serie`)

A time series is a succession of chronologically-ordered data, in which each piece of data corresponds to a partivular moment in a schedule. *TOL* differs from other languages in that it includes a grammar called `Serie`, which is specifically designed for time-series management.

Unlimited time-series

When defining time-series *TOL* allows time-series to be created without restricting them to a particular interval. We refer to these as unlimited time-series.

We can create these unlimited time series using some of the following functions:

Basics:

- The pulse function (`Pulse`), which returns a series with a value of 1 for the indicated date and 0 for the rest. For example

```
Series pulse2012 = Pulse(y2012, Yearly);
```

- The compensation function(`Compens`) which returns a series with a value of 1 for the indicated date, -1 for the next date and 0 for the rest.
- The step function (`Step`), which returns a series with returns a value of 0 until the indicated date, and 1 from this date onwards.
- The trend function (`Trend`), which returns a series with a value 0 until the indicated date, 1 on the indicated date itself and incrementally increases by one for the following dates.
- The straight-line function (`Line`), which returns a series with values in the straight line that passes through the two points indicated (date/value pairs).

Support in a schedule:

- The indictator function of a schedule (`CalInd`), which returns a series with a value of 1 when the date belongs to the schedule, and 0 when it doesn't. We anticpate that the schedule of the series contains the schedule used in the definition. For example

```
// The daily series of the first days of the month:
Series day1 = CalInd(Monthly, Daily);
```

- The cardinal function of a schedule (`CalVar`. This returns a series with a quantity of dates that have the schedule between a given date, and its successor in the series of the schedule. We anticipate that the schedule of the series is contained in the schedule used in the definition. For example

```
// The series of number of days of each month:
Serie num_days = CalVar(Daily, Monthly);
```

Randoms:

- The uniform distribution function (`Rand`), which returns a series of random values between two given numbers.
- The normal distribution function (`Gaussian`), which returns a series of normally distributed random values, with the average and standard deviation indicated.

On occasion, we use the name «inifinite series» to unlimited series, although more strictly speaking, these will be just those which don't have an end but do have a beginning.

Delimited time series

To obtain temporales delimitadas series, we can use the `SubSer` function, which allows us to restrict a particular interval. For example:

```
Serie SubSer(Pulse(y2012, Yearly), y2000, y2020);
```

We can also explicitly build series from the rows of a matrix:

```
Matrix data = (
 (1.2, 2.4, 1.5, 1.0, 3.1, 2.0),
 (0.8, 7.1, 1.1, 4.2, 5.1, 2.2)
);
```

```
Set MatSerSet(data, Yearly, y2000);
```

However, perhaps most interesting of all is obtaining them from a data-source, such as a file or database. In order to see how to build time-series from different data-sources please look at sections secciones 3.1.2 and 3.2.3.

To find out the start and end dates of a series, we can use the functions `First` and `Last` respectively.

Bear in mind that *TOL* considers the previous and following unlimited series data as omitted. This means that it tries to avoid starting and ending series with this value, automatically cutting them as necessary.

For example

```
Matrix data = Row(?, 1.2, 2.4, ?, 1.0, 3.1, 2.0);
Series series = MatSerSet(data, Yearly, y2000)[1];
Date First(series); // -> y2001
```

Series data

To access the elements of a time-series, we can use the `SerDat` function, which returns the series value for a given date. To modify it we use the `PutSerDat` function, indicating the date as the first argument and the new value as the second.

Time-series operators

To work with time-series, we have the most common operators from operations with real numbers at our disposal. The arithmetic operators (`+`, `-`, `*`, `/`) even allow us to operate between a series and a real number, returning the time-series resulting from operating each value in the series with the indicated number.

Bear in mind, that when working with two or more time-series, they must have the same time-set. The operation will only be carried out on the intersection interval; that is to say on all fo the dates that belong to the series.

Linking time-series

To enable us to complete the data in a series, we can use the operators `<<` (with priority on the right) y `>>` (with priority on the left). These allow us to link two time-series one after another, indicating which set values have priority in the event of an overlap.

If we wish to explicitly control this overlap we can use the `Concat` function. This consists of three arguments, which link two time-series together, using a specified date as the reference for when until the data of the first series should be taken (from the left).

For example

```
Serie zeros = SubSer(CalInd(W, Yearly), y2000, y2008);
Serie ones = SubSer(CalInd(C, Yearly), y2005, y2012);
Serie cc1 = zeros >> ones; // SumS -> 4
Serie cc2 = zeros << ones; // SumS -> 8
Serie cc3 = Concat(zeros, ones, y2006); // SumS -> 6
```

Operations with time-series

We don't just have arithmetic time-series at our disposal. We have the entire set of mathematical functions related to real numbers, (exponentials, trigonometric, hyperbolic, etc.). Once these are applied to time-series, they act individually on each of the series's values.

Although most of the mathematical functions relating to real numbers are also available for time series, generally speaking we can apply whichever function to all of the values of a set by using the `EvalSerie` function (similar to `EvalSet`).

We also have a conditional instruction available for time-series, `IfSer`, which allows us to build series with a conditional structure.

For example

```
// We create a series with omitted values as outline in the following example
Series series = MatSerSet(Row(1, 2, ?, 1, 3, ?, 4, 2), Yearly, y2000)[1];
// We substitute these unknown values for zeros.
Series IfSer(IsUnknown(series), 0, series);
```

Time-series statistics (`<Statistic>S`)

The statistical functions outlined in section 2.3.1 are also explicitly installed for time-series with the following nomenclature: `<Statistic>S`.

To these, we can also add some others specifically aimed at time-series, such as:

- The number of pieces of data in a series: `CountS`.
- The first value of series: `FirstS`.
- The last value of a series: `LastS`.

Bear in mind that the syntax of all of these function allows for a pair of dates as the second and third arguments. With these we can delimit the statistical application interval:

```
Real <Statistic>S(Serie series[, Date begin, Date end])
```

For example

```
Series series = SubSer(Gaussian(0, 1, Monthly), y2000, y2012m12);
Real mean = AvrS(series);
Real variance_2011 = VarS(series, y2011, y2011m12);
Real FirstS(series) <= MaxS(series); // -> True
```

Change of time-set(`DatCh`)

The aforementioned statistical functions are especially useful for changing the time-set of a series to a superior one (less detailed and with greater intervals).

The `DatCh` function(*dating change*) allows us to build a series in a new time-set, using the data of another, with the following syntax:

```
Serie DatCh(Serie series, TimeSet dating, Code statistic);
```

receiving the function that will be used as the third argument to obtain each value of the new series from the data of an interval, and which must respond with the extended form of statistical functions relating to time-series:

```
Real (Series series, Date begin, Date end)
```

It's necessary to use different statistics, depending on the nature of the data and change of time-sets. For example, if the data represents the value of a magnitude that depends on the interval size, we should add up the values with `SumS` when making the change of time-set. A typical case would be a series with the sales of a particular product. If conversely, the value of the magnitude doesn't change with the interval size, we can use a statistical average such as `AvrS`, or one that lets us choose an interval value such as: `FirstS`, `LastS`, `MinS` or `MaxS`.

2.4.4 Finite differences. Polynomial delays. (`Polyn`)

For the analysis of time-series, as with sequences, it's very common to make use of *polynomial delays*.

Finite differences perform a role in the study of time-series and difference equations similar to that carried out in the analysis of functions and differential calculus.

Regular difference

Difference (or regular difference) is the name we give to the action and result of a time-series which subtracts the immediately preceding value away from each value in a series.

If X_t is a time-series, its difference is: $\Delta X_t = X_t - X_{t-1}$

To express differences, *TOL* includes a new data-type, the grammar `Polyn`. Using this, polynomial delay operators which act on time-series can be represented.

Delay

The basic delay operator in *TOL* is `Polyn B`. This acts on a time-series by substituting the value in a date for the value immediately preceding it. It can be said that the operator `B` moves each item of data of the time-series one date forward.

To apply `Polyn` variables to time-series, we use `:` operator (semi-colon).

For example

```
Series X_t = SetSer(Range(1, 8, 1), Yearly, y2001);
Series X_t.minus.1 = B : X_t;        // delay X_t
Series Dif.X_t = X_t - X_t.minus.1; // difference of X_t
```

We can see the results of previous operations in table form:

t	2001	2002	2003	2004	2005	2006	2007	2008	2009
X_t	1	2	3	4	5	6	7	8	
X_t.minus.1		1	2	3	4	5	6	7	8
Dif.X_t		1	1	1	1	1	1	1	

Table 2.4.1: Example of the result of applying a delay and a difference to a time-series in table form.

Polynomials

If we apply the delay operator on subsequent occasions we can obtain delays and differences superior to one. We can therefore define polynomial operators in a general way, as a linear combination of powers of the delay operator `B`.

For example:

```
Polyn p = 1 + 0.5*B - 0.8*B^2;
```

Inversely, we can find out the grade and co-efficients of a given polynomial using the `Degree` and `Coef` functions respectively.

Another very useful function is `Monomes`, which breaks a polynomial down into a set of monomials for us.

Note that *TOL* implements the composition of the operator B as the product of variables of type `Polyn`. This is done in a way that allows us to define order differences superior to 1 as powers of the difference operator `Polyn (1-B)`.

```
Polyn dif = 1-B;
Polyn dif_order.2 = (1-B)^2 // second difference
```

Seasonal differences

Seasonal differences are those in which each value of a series is substracted another a certain number of instances previously.

If X_t is a time-series, a seasonal difference p is: $\Delta_p X_t = X_t - X_{t-p}$

These differences are very useful when analysing time-series, especially in those which demonstrate recurrence. For example, series in monthly time-sets which show annual cycles allow seasonal differences of period 12.

```
Polyn dif_period.12 = 1-B^12; // seasonal difference
```

Seasonality shouldn't be confused with a difference in subsequent applications. For example, we can say there is seasonal difference of order 2 and period 12:

```
Polyn dif_period.12_order.2 = (1-B^12)^2;
```

Advance operator

TOL has an advance operator, `Polyn F`, inverse to the delay operator.

```
Polyn B*F; // == Polyn 1
```

It acts on a time-series by substituting the value in a date for the value that comes immediately after it.

2.4.5 Difference equations Polynomial quotients (`Ratio`)

In modelling, it's very common to use *ecuaciones en diferencias* to describe time-series under analysis. De manera general podemos expresar la ecuación en diferencias que describe una serie Y_t como un polinomio de retardos $p(B)$ aplicado a esta serie igualado a una constante u otra serie temporal X_t.

$$p(B)Y_t = X_t$$

TOL includes a new type of data, the grammar `Ratio`, to express the inversion of the polynomial of delays, and with which we express the difference equation:

$$Y_t = \frac{1}{p(B)} X_t$$

Thus, a variable type `Ratio` is defined as the quotient of two variables of type `Polyn`.

Here we see an example:

```
// We divide a series "y" and a polynomial "p"
// and we find the series "x" which verifies:
//   p:y = x
Series y = SubSer(Gaussian(0, 1, Daily), y2000, y2000m12d31);
Polyn p = 1 - 0.5*B;
Serie x = p:y;

// We solve the difference equation and once more find "y" (y_new):
//    y_new = (1/p) : x
Ratio r = 1/p;
Series y0 = SubSer(y, First(y), First(y)); // initial values
Series y_new = DifEq(r, x, y0);
```

Note that to solve the difference equation we have to include the appropriate *valores iniciales* set. We can realise this intuitively once we realise how a differentiated series suffers a loss of information; it contains less data than the undifferentiated set. It's necessary to include this information when carrying out the inverse process, in a way similar to what happens in the integration of the differential calculation.

The second member of a difference equation can also be a differentiated set.:

$$p(B)Y_t = q(B)X_t$$

:

$$Y_t = \frac{q(B)}{p(B)}X_t$$

Given a variable of type `Ratio`, we can obtain the numerator and denominator polynomials, via the `Numerator` and `Denominator` functions respectively.

2.4.6 Modelling with time series

In time-series modelling, it's very common to find a correlation between the different values of the series. This means that we can at least partly explain a particular series value by looking at its previous values.

This is the basis of auto-regressive models (AR), in which the output is described as a linear combination of delays whose:

$$Y_t = \sum_i \varphi_i \, Y_{t-i} + E_t$$

where Y_t is the observed time-series, φ_i are the auto-regressive parameters and E_t is the unexplained part or model error.

ARIMA model

ARIMA (Auto-Regressive Integrated Moving Average) models are an extension of these auto-regressive models. They incorporate differences and mobile averages which can be describe with the following equation:

$$\left(1-\sum_i \varphi_i B^i\right)(1-B)^d Y_t = \left(1-\sum_i \theta_i B^i\right)E_t$$

Where the first polynomial of delays, φ_i, corresponds with the autoregressive part (AR), the second polynomial is the order of differences d and the third, with the parameters θ_i, the mobile average part (MA).

ARIMA blocks(`@ARIMAStruct`)

To define ARIMA models, *TOL* includes the structure `@ARIMAStruct`, which has four fields. In addition to the three polynomials already described, we can use these to indicate a value for the regularity of the model. They can also be used to create seasonal ARIMA blocks (*seasonal* ARIMA or SARIMA).

Generally speaking, we can describe an ARIMA model as a set of ARIMA blocks.

For example

```
// We create a sturcture corresponding with an ARIMA(2,0,0)-SARIMA.12(0,1,1)
Set arima = [[
  @ARIMAStruct(1, 1-0.5*B-0.5*B^2, 1, 1);
  @ARIMAStruct(12, 1, 1-0.5*B^12, 1-B^12)
]];
```

ARIMA etiquettes

To facilitate the specification of ARIMA blocks, *TOL* has an ARIMA model etiquette system with the following form:

```
"P<period>DIF<order>AR<degrees>MA<degrees>"
```

in which the code between angle-brackets (<>) should be substituted for the corresponding numerical values.

For the polynomial of differences, we only need to indicate the number of differences to be applied, as the degree of difference is obtained from the period value. While for the polynomials AR and MA the differente degrees must be explicitly indicated by separating them with full-stops (`.`).

In a case where the polynomial ARIMA consists of more than one block, the etiquette can be composed by use of the underscore(_) to separate the different part fo each block.

For example, the etiqueete of the ARIMA model defined earlier is:

```
"P1_12DIF0_1AR1.2_0MA0_1"
```

as a combination of:

```
"P1DIF0AR1.2MA0" // an AR with two parameters (degrees 1 y 2)
"P12DIF1AR0MA1"  // a difference with seasonality 12
                 //   and an MA with a paramteter (grade 1)
```

The standard *TOL* library, `StdLib` (see section 4.1) has two functions at its disposal to obtain ARIMA blocks from the etiquette and vice versa: `GetArimaFromLabel` and `GetLabelFromArima`.

ARIMA model estimation (`Estimate`)

As well as the ARIMA structure described, we can include explicit terms to model with time-series, in such a way that the equation model would be:

$$Y_t = \sum_i \beta_i X_{i,t} + N_t$$

where the noise term, N_t, is what the previously described ARIMA structure demonstrates.

TOL includes a maximum likelihood estimator, `Estimate`, which allows us to jointly estimate the linear regression parameter(β_i) and the ARIMA part parameters. (φ_i y θ_i).

To specify the distinct characteristics of the model in the `Estimate` function, it's necessary to create a set with the `@ModelDef` structure, indicating the series output, the ARIMA structure polynomials and the series inputs, amongst other arguments. Specifically speaking, the model inputs are indicated through use of polynomial series pairs with an `@InputDef` structure. These allow us, therefore, to incorporate delay polynomials into the explantory terms.

Example:

```
// We create a time series with an MA(2) structure
// to which we add an external component (filter)
// to carry out a controlled estimation as in the example.
Series residuals = SubSer(Gaussian(0, 0.1, Monthly), y1999m11, y2010);
Real phi1 = 0.5;
Real phi2 = -0.3;
Polyn MA = 1 - phi1*B - phi2*B^2;
Series noise = MA:residuals;
Real beta = 1.2;
Series input = SubSer(Rand(1, 2, Monthly), y2000, y2010);
Series output = noise + beta * input;
// We estimate the model: output = beta*input + noise
// con: noise = MA(2):residuals
Set estimation = Estimate(@ModelDef(output, 1, 0, 1, 1, // Output
  Polyn 1, [[Polyn 1]], [[Polyn 1-0.1*B-0.1*B^2]],      // ARIMA
  [[@InputDef(0.1, input)]], Empty)                      // Filtro
);
Real estimated_beta = estimation["ParameterInfo"][1]->Value;
Real estimated_phi1 = estimation["ParameterInfo"][2]->Value;
Real estimated_phi2 = estimation["ParameterInfo"][3]->Value;
```

2.5 Advanced notions

2.5.1 Modular programming (NameBlock)

A new type of variable in *TOL* which similar to sets, (`Set`) has a multiple nature, is the block or *name-block* (`NameBlock`). The similarity to sets comes fromt he fact that it also formed of other variables.

Although their nature and the way we define them are similar to sets, their purpose is rather different. While sets (`Set`) basically exist to contain elements of various types, a block (`NameBlock`) is there to offer a definition space or a local, permanent environment to create variables and functions.

One of the main reasons *nameblocks* have been included in *TOL* is to favour a certain type of organisation and modularity in programming. This allows for the definition of variables and functions with a high degree of local visibility which can go on to form part of an array of variables and global functions.

To define a block (`NameBlock`) we will make use of the following syntax:

```
NameBlock <name> = [[
  <Grammar1> <name1> = ...;
  <Grammar2> <name2> = ...;
  ...
]];
```

Example:

```
NameBlock block = [[
  Real value = 3.1;
  Text country = "Spain";
  Date date = Today
]];
```

Note that when constructing the *nameblock*, defined variables only have local visibility. This in contrast with the process of creating sets. Compare the following codes:

```
Set set = [[
  Text text1 = "hola"
]];
WriteLn(text1);
```

```
hello
```

```
NameBlock block = [[
  Text text2 = "hello"
]];
WriteLn(text2);
```

```
ERROR: [] text2 isn't a vaild object for the type Text.
```

Member access

Block elements and members are characterised by name, this being a unique and essential characteristic of each element. The (`Code`) functions defined within a block have access to the rest of the block elements by name, as if they were global.

In general, when we want to access the members of a block we use the `::` operator (two semi-colons). This is done as if we were computing an identifier of said member, putting it ahead of the block in which it is found:

```
<Grammar> <blockName>::<memberName>;
```

For example:

```
NameBlock supercomputer = [[
  Text name = "DeepThought";
  Real theAnswer = 42;
  Real GetTheAnswer(Real void) { theAnswer }
]];
Text supercomputer::name;            // -> "DeepThought"
Real supercomputer::GetTheAnswer(?); // -> 42
```

Note that a function can be defined in the frame of a *nameblock* and in this way can be named in the same sentence used to access it.

Encapsulation

Another of the characteristics included in *nameblocks* is that of encapsulation of variables and functions. This gives visibility only to members that are useful when taken out of the context of the local block.

For this, accessibility to block members can be restricted by selecting their name, which then enables three different options.

- Public members, whose access from outside the block is permitted; both for enquiries and editing. Public members' names always begin with a letter.
- Read-only members, whose access from outside the block is for enquiry purposes only. Read-only members' names always begin with an underscore and a full-stop (`_.`).
- Private members, who are not permitted to have access from outside the block. Private members' names begin with an underscore (_), followed by any other alpha-numeric character except for the full-stop. (`.`)

Example:

As an example of the use of *nameblocks* as modular structures, we create a stopwatch-type model, which measures the time elapsed between calls.

```
NameBlock Clock = [[
  Text _.description = "Clock to be used as a stopwatch";
  Real _initialTime = 0; // private member
  Real _elapsedTime = 0; // private member
  Real Start(Real void) {
    Real _elapsedTime := 0;
    Real _initialTime := Copy(Time);
  1};
  Real Stop(Real void) {
    If(_initialTime!=0, {
      Real _elapsedTime := Time - _initialTime;
      Real _initialTime := 0;
    1}, 0)
  };
  Real Reset(Real void) {
    Real _elapsedTime := 0;
    Real _initialTime := 0;
  1};
  Real GetElapsedTime(Real void) {
    Case(_elapsedTime!=0, {
      Copy(_elapsedTime)
    }, _initialTime!=0, {
      Time - _initialTime
    }, True, {
      WriteLn("The clock is not in progess.", "W");
      ?
    })
  }
]];
Real Clock::Start(?);
Real Sleep(1);
Real Clock::GetElapsedTime(?); // -> ~1
Real Sleep(1);
Real Clock::Stop(?);
Real Sleep(1);
Real Clock::GetElapsedTime(?); // -> ~2
```

2.5.2 Classes (`Class`)

The inclusion of the `NameBlock` grammar in *TOL* opens a wide range of possibilities with regards to the creation of functional blocks. This results in program-development truly oriented towards objects.

A lot is written about the best way to introduce the programming paradigm oriented toward objects. Perhaps for a *TOL* user, someone who understands how to clarify structures (`Struct`) in relation to sets (`Set`), the best way to begin would be to say that classes (`Class`) are to blocks (`NameBlock`), what structures are to sets.

In a way similar to the one in which we introduced structures, (see section 2.2.4), we can present classes as a specialisation of the `NameBlock` grammar. This permits us to represent entities and create units of a particular concept. An *objeto*, is defined in this way, as each instance in a class, and which we can describe as a *nameblock estructurado*.

In addition to the block members that host the characteristic information of each unit, classes (conversely to structures) allow us to give a functional structure to all the objects (which is common to all the instances in the same class). This allows us to interact with them. The first of these are named *atributos* and the second *métodos*.

Terminology

To aid user-learning, here we outline some of the terms used for programming oriented towards objects:

- Class: The structure, the design and the pattern with which objects are built.
- Object (variable-type `NameBlock`) created with a class.
- Member: Each one of the elements of an object or instance. This can also be called component.
- Attribute: Any member of an instance's variable type (not function). It's value is particular to each instance. This can also be called property.
- Method: Any member of function type of an instance. It is common to all instances. However, it acts on, and only has (internal) access to, the object on which the call was made.

Class definition

Class definition is performed using the following syntax:

```
Class @<ClassName> {
  // Attributes:
  <Grammar1> <attribute1>;
  <Grammar2> _.<attribute2>; // read-only attribute
  <Grammar3> _<attribute3>;  // private attribute (for internal use)
  <Grammar4> <attribute4> = <defaultValue4>;
  ...
  // Methods:
  <GrammarM1> <Method1>(<arguments...>) {
    <code...>
  };
  <GrammarM2> _<Method2>(<arguments...>) {
    <code...>
  }; //  private method (for internal use)
  ...
};
```

where the code appears between angle brackets (<>) it's necessary to substitute it for its corresponding value.

Bear in mind that the members of a class's instance respond to the same visibility criteria as of any other *nameblock*. This makes it possible to declare members public, private or read-only.

<u>Example:</u>

```
// To highlight the comparison with structures,
// We redifine the @Vector3D as a structure:
Class @Vector3D {
// Attributes:
  Real _.x;
  Real _.y;
  Real _.z;
// Methods:
  Real GetX(Real void){ _.x };
  Real GetY(Real void){ _.y };
  Real GetZ(Real void){ _.z };
  Real SetX(Real x) { Real _.x := x; 1 };
  Real SetY(Real y) { Real _.y := y; 1 };
  Real SetZ(Real z) { Real _.z := z; 1 };
  // spherical and cylindrical additional coordinates:
  Real GetR(Real void) { Sqrt(_.x^2 + _.y^2 + _.z^2) }; // length, radius
  Real GetRho(Real void) { Sqrt(_.x^2 + _.y^2) };       // radial distance
  Real GetPhi(Real void) { _ATan2(_.y, _.x) };          // azimuth angle
  Real GetTheta(Real void) { ACos(_.z/GetR(?)) };       // polar angle
  // auxiliar (private) methods:
  Real _ATan2(Real y, Real x) { ATan(y/x) + If(x<0,Sign(y)*Pi,0) }
};
```

Note that just as with structure names, class names have to begin with the `@` (at) symbol.

Instancing

To create class objects or instances we use the following syntax:

```
@<ClassName> object = [[
  // Value list for attributes
  <Grammar1> <attribute1> = <value1>;
  <Grammar2> _.<attribute2> = <value2>;
  ...
]];
```

Access to attributes from outside the class is made, as with those of any other blocks (`NameBlock`), through use of the `::` operator (two semi-colons):

```
<Grammar1> value1 = object::<attribute1>;
```

For example

To create an instance of the class `@Vector3D`, we write:

```
@Vector3D vector3D = [[
  Real _.x = 1;
  Real _.y = 2;
  Real _.z = 3
]];
```

Note that the main difference that we find with a similar declared block

```
NameBlock block3D = [[
  Real _.x = 1;
  Real _.y = 2;
  Real _.z = 3
]];
```

It's the existence of the methods that `vector3D` has available to it as a result of being of the`@Vector3D` class, that allow us to obtain additional information:

```
Real vector3D_length = vector3D::GetR(?);                  // -> 3.741657
Real vector3D_azimuth = vector3D::GetPhi(?)*180/Pi;        // -> 63.435°
Real vector3D_inclination = vector3D::GetTheta(?)*180/Pi;  // -> 36.70°
```

This said, the class not only take care of building the attributes (and initialize the optional attributes with the default values) but also wrapping the object created with behaviour defined in the class, i.e. the methods

Static members (`Static`)

On occasions, we may wish to include certain information or functionality in a class, thus facilitating access to it with having to create an instance.

TOL allows for member classes to be incorporated as if it were a special type of *nameblock*, simply by putting the the keyword `Static` before the definition:

```
Class <@ClassName> {
  ...
  // Static attributes:
  Static <GrammarC1> <classAttribute1> = <valueC1>;
  ...
  // Static methods:
  Static <GrammarCM1> <ClassMethod>(<arguments...>) {
    <code...>
  };
  ...
};
```

Access to the class members known as *miembros estáticos* can be obtained via the `::` operator (two semi-colons):

```
Anything @<ClassName>::<classMember>;
```

Advance declaration

A class name must be declared in order for us to be able to use it. This is true even if it isn't to be used until a later point in time. If this isn't done *TOL* will return a syntactical error. To help us avoid encountering such issues, TOL allows for the advanced declaration of a class using just its name.

```
Class @<ClassName>;
```

One scenario that clearly demonstrates this is one in which we define two inter-related classes, in such a way that each one makes use of the other in its definition:

```
// We pre-declare @B to be able to use it in the definition of @A
Class @B;

// We pre-declare @A using @B (which is already pre-declared)
Class @A {
  ... @B ...
};

// We declare @B using @A (which is already declared)
Class @B {
  ... @A ...
};
```

2.5.3 Class design

Now let's look at some additional characteristic relating to the creation and use of classes:

Attribute management

Attribute values can be easily modifed, just as any other variable can via:

```
<Grammar1> object::<attribute1> := <newValue1>;
```

However, this isn't very convenient when it comes to the design of many classes, as editing an attribute could lead to adverse knock-on effects in other internal modifications. To avoid this, attributes are declared as read-only (preceded by `_.`) and accompanied by a method for their editing.

A recommended practice in such cases is to create a pair of methods `Get<Attribute>` and `Set<Attribute>` for each attribute (`_.attribute`:

```
<Grammar1> value1 = object::Get<Attribute1>(?);
Real object::Set<Attribute1>(<newValue1>);
```

Alternative constructors

It is often useful to have functions available to us that allow us to build instance from just a handful of arguments. These alternative "constructors" aren't strictly constructors in the truest sense of the word. Rather, they are functions that directly or indirectly make a call to the constructor whose default syntax is:

```
@<ClassName> object = [[ ... ]];
```

These constructor functions are related to class, as it's common (although not essential) to place them there as class statics, in a way that an instance can be created with the syntax:

```
@<ClassName> object = @<ClassName>::<Constructor1>(<arguments...>);
```

The constructor function is no more than a static method that returns an instance. It is defined in the same way as any other method but the declaration is preceded with the word `Static`:

```
Class @<ClassName> {
  ...
  // Constructors:
  Static @<ClassName> <Constructor1>(<arguments...>) {
    <definition...>
  };
  ...
};
```

Inheritance (`:`)

One of the most important characteristics of object-oriented programming is inheritence. This mechanism allows us to derive a general class from a more specific one. This is done in a way in which the the definition of the first is preserved, making it possible to define a new class where only the new members we wish to include are defined.

To create a new class which inherits from another/others, the class-definition syntax is broadened by using the semi-colon operator, (`:`) followed by the name(s) of the classes from which we wish to inherit.

```
Class @<ClassName> : @<ClassName1>[, @<ClassName2>, ...] {
  // Members:
  ...
};
```

where the square brackets (`[]`) indicate optionality.

Redefinition of methods

In the event that the same method from two different classes is inherited, the definition of the latter will always take precedence over the former. Furthermore, methods included in the new class's definition will take precedence over any inherited member.

Bear in mind, however, that the declaration of methods can't be changed; these being the output grammar and the input argument names.

Abstract classes

One possibility associated with the concepts of inheritence and method redefinition is that to create partially-defined classes.. These are what we refer to when talking about virtual methods, or methods that are declared (without being indicated in the body of the function) in the class definition, leaving their definition for derived classes.

Virtual or abstract classes are, therefore, classes which have at least one virtual method. Not being completely defined, it's not possible to directly create instances from them, rather this must be done from a derived class.

Example:

```
// We create an abstract class to represent flat figures.
Class @Shape {
  // We declare two virtual methods which must be derived in derived classes.
  Real GetPerimeter(Real void);
  Real GetArea(Real void);
  // We define other methods that will inherit derived classes.
  Real PrintPerimeter(Real void){ WriteLn("Perimeter: "<<GetPerimeter(?)); 1};
  Real PrintArea(Real void) { WriteLn("Area: "<<GetArea(?)); 1}
};
// We derive two @Shape classes to represent cricles and squares:
Class @Circle : @Shape {
  Real _.radius;
  Real GetPerimeter(Real void) { 2 * Pi * _.radius };
  Real GetArea(Real void) { Pi * _.radius**2 }
};
Class @Square : @Shape {
  Real _.side;
  Real GetPerimeter(Real void) { 4 * _.side };
  Real GetArea(Real void) { _.side**2 }
};
// We create three figure instancing @Circle and @Square
Set shapes = [[
  @Circle c1 = [[ Real _.radius = 1 ]];
  @Circle c2 = [[ Real _.radius = 1.5 ]];
  @Square s1 = [[ Real _.side = 2.25 ]]
]];
// We call the common method WriteArea to print its areas
Set EvalSet(shapes, Real (@Shape shape) { shape::PrintArea(?) });
```

```
Area: 3.141592653589793
Area: 7.068583470577035
Area: 5.0625
```

Internal reference (`_this`)

When designing a class, it is occasionally necessay that an object has a reference to itself at its disposal. All classes have this reference available through a private attribute called `_this`.

Destroy method (`__destroy`)

It is possible to include a destroy method, called upon when an instance is decompiled. However its use isn't particularly common in *TOL* programming, as memory management isn't run under the control of the user.

This method can take charge of tasks such as closing files or open database connections, during the creation or use of an instance.

The declaration reserved for this method is:

```
Real __destroy (Real void)
```

2.5.4 Other language elements

Directives (`#<Directive>`)

This section introduces directives in TOL as a special language element.

`#Embed`: See section 3.1.1.

`#Require`: See section 3.4.2.

2.5.5 Memory use

This section brings together some pointers in order to aid better understanding of the use <t0/>TOL<t1/> makes of memory, and how to improve the efficiency of the algorithms it requires.

Objects in memory (`NObject`)

Access structures by reference (`@<Grammar>`)

3 Use of *TOL*

3.1 System files

3.1.1 Source files .tol

This section includes information relative to the design and creation of *TOL* programs in .tol files. It also mentions other types of files for the construction of code projects such as .prj files.

`#Embed Directive`

3.1.2 File-reading/writing

This section includes information relating to file-reading/writing.

We highlight the: `FOpen`, `ReadFile`, `ReadTable`, `MatReadFile` and `VMatReadFile` functions, and the file-types: .bst, .bdt, .bbm, .bvm, etc.

3.1.3 Serialisation in *TOL*. *OIS*

This section includes informtation relating to serialisation in *TOL*: *OIS* reading/writing of .oza files.

3.1.4 Integration with the operating system.

This section includes information relating to the integration of *TOL* with the file system via the operating system.

We highlight the functions: `GetDir`, `MkDir`, `GetFileName`, `GetFileExtension`, `GetFilePath`, `FileExist` and `FileDelete`, and the function-set `OsDir*` y `OsFil*`.

3.2 Communication

3.2.1 Communication with the operating system

This section includes information relating to the integration of *TOL* with the operating system to enable programs to run via the command line interface, using functions such as: `System` or `WinSystem`.

3.2.2 Obtaining *urls*

TOL has a number of functions for downloading remote files through http or ftp protocols. All of these functions download the content referenced by url given as an argument. The value returned is the downloaded content.

The functions available for this purpose are:

- `Text GetUrlContents.tcl.uri(Text url)` installs the download based in the *uri* package of *Tcl*.
- `Text GetUrlContents.tcl.curl(Text url_)` installs the download based in the *cURL* library.

- `Text GetUrlContents.sys.wget(Text url)` `installs the download based in the` *wget* command line interface tool.
- `Text GetUrlContents.tcom.iexplore(Text url)` installs the download based in *Internet Explorer*'s *API COM*.

A variant of these functions is available which redirects one of them in a pre-configured way.:

- `Text GetUrlContents(Text url)` is a general download function that uses the above functions as a function of the global configuration function`TolConfigManager::Config::ExternalTools::UrlDownloader` (seesection 3.3.1), which can take one of the following values:
    - o `"tcl:uri"` to run `GetUrlContents.tcl.uri.`
    - o `"tcl:curl"` to run `GetUrlContents.tcl.curl`
    - o `"sys:wget"` to run `GetUrlContents.sys.wget` y
    - o `"tcom:iexplore"` to run `GetUrlContents.tcom.iexplore`.

Obtain the *url* via *cURL*

The`GetUrlContents.tcl.curl`function is supported by the `GetUrl` function, which is installed in the `CurlApi` module (seesection 4.1.2). This function is based in the *cURL* tool (http://curl.haxx.se) and allows us to control download parameters. The function has the following prototype:

```
Set CurApi::GetUrl(NameBlock args)
```

Where `NameBlock args` could contain the following members:

- `Text url`: contiene el *url* que referencia al contenido que se solicita descargar.
- `Text file`: contains the path where the content will be downloaded from. This is an optional argument. If no argument is defined, the content is returned as a result of the function.
- `Real timeout`:establishes the maximum time in download seconds. Note that the resolution time for names, and the establishment of a connection normally takes a considerable time. If an argument of less than a few minutes is established, we run the risk of aborting completely normal operations. It is an optional parameter and its value by omission `0.0` implies no time-check.
- `Real connectiontimeout`: establishes a maximum time in seconds dedicated to the establishment of a connection with the server. It only takes into account the connection phase and once the connection is established this option is no longer used. It is an optional parameter and its value by omission `0.0` implies no time-check..
- `Real verbose`: is an indicator for when the true value takes (`!=0`), it causes the release of messages through the standard output of the operations involved in the download. It's very useful for both understanding and filtering the download protocol.

The function returns a set that contains information about the implemented transfer. If the transfer was a success, the set will contain the following fields:

- `Real connectTime`: the time taken in seconds from the beginning to completion of the connection to the remote host.

- `Real fileTime`: the remote date of the downloaded document, counted as the number of seconds since 1st of January 1970, in the *GMT/UTC* time-zone. The value of this field can be -1 for various reasons; such as unknown time, or the server not supporting the command that tells you the date of a document etc.
- `Real totalTime`: The total time of the transaction, in seconds, for the transfer, including name resolution, *TCP* connection, etc.
- `Real sizeDownload`: the total quantity of bytes downloaded.
- `Real speedDownload`: the average download speed,measured in bytes/s for the entire download.
- `Text nameData`: the field-name within the same set that contains the downloaded data. It can take the two `"file"` values in cases where it was requested that the download was written in a file or `"bodyData"`, if the destination file wasn't specified in the call.
- `Text file`: the file where the downloaded content should be stored.
- `Text bodyData`: the downloaded content if no file has been specified as destination.

If the download was unsuccessful, the resulting set will contain the following fields:

- `Real failStatus`: and error code.
- `Text explain`: a chain with the error description.

We'll now move on to take a look at a few examples of the use of the `CurApi::GetUrl` function:

Example 1: Download http content, storing it in a variable.

```
Set result1 = CurlApi::GetUrl([[
  Text url = "http://packages.tol-project.org/OfficialTolArchiveNetwork/"
    "repository.php?tol_package_version=1.1"
    "&tol_version=v2.0.1%20b.4&action=ping&key=658184943";
  Real verbose = 1;
  Real connecttimeout = 3
]]);
If(result1::failStatus, {
  WriteLn("CurlApi::GetUrl error: "<<result1::explain )
}, {
  WriteLn("CurlApi::GetUrl ok, the result is: "<<result1::bodyData)
} );
```

Example 2: Download http content to a file.

```
Set result2 = CurlApi::GetUrl([[
  Text url = "http://packages.tol-project.org/OfficialTolArchiveNetwork/"
    "repository.php?action=download&format=attachment&package=BysMcmc.4.4";
  Text file = "/tmp/BysMcmc.4.4.zip",
  Real verbose = 1;
  Real connecttimeout = 3
]]);
If(result2::failStatus, {
  WriteLn("CurlApi::GetUrl error: "<<result2::explain )
}, {
  WriteLn("CurlApi::GetUrl ok, the result was downloaded to: "
    <<result2::file)
});
```

Example 3: Download ftp content to a file.

```
Set result3 = CurlApi::GetUrl([[
  Text url = "ftp://ftp.rediris.es//pub/OpenBSD/README";
  Text file = "/tmp/README";
  Real verbose = 1;
  Real connecttimeout = 30
]]);
```

3.2.3 Access to a database (`DB<Function>`)

In sections 2.3.3 and 2.4.3, we saw how we can generate time-series or matrices using elemental functions, manually-inserted data or random numbers. Now, we describe some of the functions that *TOL* use to access databases. These allow us to obtain information and build variables.

TOL *includes some basic functions for communication with databases via ODBC*:

- The `DBOpen` function opens a connection with a database.
- The `DBClose` closes a connection with a database.
- The `DBTable` function returns a result in the form of an *SQL* query `SELECT` set (`Set`)
- The`DBSeries`, `DBSeriesColumn` and `DBSeriesTable` functions help us to create time-series *SQL* queries `SELECT`.
- The `DBMatrix` allows the creation of matrices from *SQL* queries `SELECT`.
- The `DBExecQuery` generally allows us to run every type of*SQL* sentence to interact with the database.

Example. Reading files

We now outline an example for using database functions with an *ODBC* to a folder with flat-data files:

<u>Creation data on file</u>

We create a folder which will create the instances of the database. In the example we will use: "C:/Data".

There, all files of type.txt will be placed with the data. To do this we create two files with the following data. We can use the note-pad to create it.

data.txt

```
date,value1,value2
2000/01/01,210,162
2001/01/01,181,142
2002/01/01,192,158
2003/01/01,191,182
2004/01/01,172,144
2005/01/01,207,132
2006/01/01,205,159
2007/01/01,199,162
2008/01/01,205,158
2009/01/01,219,122
2010/01/01,186,125
```

multiple.txt

```
name,date,value1,value2
"A100",2000/01/01,102,312
"A100",2001/01/01,111,282
"A100",2002/01/01,128,315
"A100",2003/01/01,107,308
"A100",2004/01/01,131,322
"A100",2005/01/01,142,295
```

```
"A101",2000/01/01,52,112
"A101",2001/01/01,61,82
"A101",2002/01/01,28,85
"A101",2003/01/01,67,108
"A101",2004/01/01,81,122
"A101",2005/01/01,42,95
```

Note that this data only has an educational purpose.

ODBC

Moving on, we now create an *ODBC* connection using the *Microsoft* text-file driver. We give it the name "TOLTest" (for example).

This can be done manually using *Windows*'s administrative tools running the following code on the command console:

```
odbcconf configsysdsn "Microsoft Text Driver (*.txt; *.csv)"
  "DSN=TOLTest|defaultdir=C:\Data"
```

Obtaining data

```
Real GlobalizeSeries := 0;
Real DBOpen("TolTest", "", "");
Set DBTable("SELECT * FROM data.txt");
Set DBSeries("SELECT date, value1 FROM data.txt", Yearly, [["A000"]]);
Set DBSeriesColumn("SELECT name, date, value1 FROM multiple.txt", Yearly);
Set DBSeriesTable("SELECT name, date, value1, value2 FROM multiple.txt",
  Yearly, [["_A", "_B"]]);
Real DBClose("TolTest");
```

3.3 *TOL* configuration

3.3.1 Configuration model(`TolConfigManager`)

TOL's configuration is managed from the `TolConfigManager` module. This *nameblock* hosts all of the configuration values, as well as the methods for their use and editing.

The `TolConfigManager` allows the user to modify their *TOL* working preferences, using the same language and storing it on disk in a simple and straightforward way.

The configuration is kept as a *TOL* session in a configuration file in the application's data folder.

```
Text TolConfigManager::_.path
// => Text TolAppDataPath+".tolConfig."+Tokenizer(Version," ")[1]+".tol";
```

and each time *TOL* starts up and the block is built we see: `TolConfigManager::Config`.

The configuration block can be edited within the running session as any other type of `NameBlock` variable, although it's possible that the consequences of applying this configuration won't have any effect until a new session is opened. To save the configuration changes so that they take effect in the next session, we need do no more than call the method `TolConfigManager Save`.

Example:

For example, if we have a machine without an internet connection is recommended that the local *TOL* configuration is changed, to avoid the system wasting time trying to make a connection. This can be done manually using the following code:

```
Real TolConfigManager::Config::Upgrading::TolVersion::CheckAllowed := False;
Real TolConfigManager::Config::Upgrading::TolPackage::LocalOnly := True;
Real TolConfigManager::Save(?);
```

If at a later point in time it does have a connection the previous configuration can be undone via:

```
Real TolConfigManager::Config::Upgrading::TolVersion::CheckAllowed := True;
Real TolConfigManager::Config::Upgrading::TolPackage::LocalOnly := False;
Real TolConfigManager::Save(?);
```

3.3.2 Other configuration mechanisms

Older *TOL* configurations belong in global variables or internal variables that can be edited by other means.

The configuartion related to the*TOLBase* is stored in the `tolcon.ini` file, in the user's*home* directory.

3.4 Package system

From *TOL* version 2 onwards, a mechanism has been implemented to build and automatically distribute reusable software components. In this section we will explain what a *TOL* package is and how to use it. We'll also looks at related information about the management of installed packages and package repositories.

3.4.1 Packages

In section 2.5.1 we presented blocks or *nameblocks* (`NameBlock`) as the grammar that makes it possible for *TOL* to be oriented towards modularity. This allows for the creation of definition spaces, capable of storing sets of variables or related co-prime functions.

The idea of a package comes from the concept of modularity, and the possibility of loading and distributing a whole module as a unit. A package is essentially no more than a *nameblock* distributed in an .oza file. See section 3.1.3 for further details about this type of storage.

Packaging

In the *TOL* programming frame, we identify the package with the (`NameBlock`) that contains the set of class and structure variables, functions and definitions. This is distributed and loads as a unit of the *TOL* package system.

Looking more from a distribution-focused point of view, a package is a compressed file (specifically a .zip file) that contains the *nameblock* serialisation (an .oza file). This is accompanied by other complementary resources that can be: files containing icons or images, documentation, libraries of compiled functions (written in *C++* and distributed in .dll or .so files) and even uncompiled *TOL* code (.tol files) by way of example.

Name and version

A *TOL* package is identifed by its name and version.

The name of the package matches with the name of the *nameblock* that represents it. It generally follows a *CamelCase* structure.

The version consists of integers. The first (the *versión alta* number) shows if there has been an important change in the structure and functionality; one which could break the compatability with the user-code in use. The second however (the *versión baja* number) shows that changes have been made to the package that maintain package-use compatability. These changes could be corrections or the inclusion of new functionalities.

Identifier

The package identifier is created by linking the name and two version numbers `<name>.<version.high>.<version.low> with a full-stop (.)`.

For example, the identifier `GuiTools.3.5` makes reference to version 3.5 of *GuiTools*, which includes utilities for *TOLBase* graphical integration.

3.4.2 Installation and use of packages (#Require)

The simplest way to install and load a package (providing it isn't already installed) is to use the *TOL* directive `#Require`, accompanied by the desired name or package identifier.

For example

```
#Require GuiTools;
```

The use of directives in *TOL* is relatively uncommon and their behaviour is slightly different from other sentences. They differ in that they take effect before both compilation and the interpretation of the rest of the lines of code.

For example:

```
WriteLn("Line 1");
#Require GuiTools;
WriteLn("Line 2");
```

```
The package GuiTools.3.5 has been loaded
Line 1
Line 2
```

Specifically, the directive `#Require` takes care of loading a package (if it isn't already loaded) before compiling the rest of the code. This avoids possible syntactical errors due to the use of declared structures and classes in the package.

Bear in mind that the call to `#Require` tries to load the package by use of all available means, even trying to locate it in a package repository and install it.

Loading versions

The `#Require` directive allows us to specify the package-version that we wish to load. This make it possible to indicate the two version numbers, just the first one or only the package-name, as preferred.

- If we only indicate the package name, the most recently installed version will be loaded. Specifically, the package whose *versión baja*is the largest from those amongst the *versión alta* will be loaded.

```
#Require <PackageName>;
```

- If we indicate the first version number, the latest version of the package whose *versión alta* number matches the one indicated will be loaded.

```
#Require <PackageName>.<VersionHigh>;
```

- If we indicate the complete package identifier (name and two version numbers), only the said version of the package will be loaded.

```
#Require <PackageName>.<VersionHigh>.<VersionLow>;
```

Note that the directive will not try to load the package if it is already loaded, even when the indicated version doesn't match that of the loaded package, as it isn't possible to load two different versions of the same package.

Compatibility with the *TOL* version

On occasions, packages are built which are supported by modifications or updates in the language which mean that don't work in previous versions of *TOL*. For this reason, some packages have a minimal version, from which compatibility is guaranteed.

For example, version 3.5 *GuiTools* requires at least version 3.1 de *TOL*.

Bear in mind that if a version of a package ins't compatible with the version of *TOL* we're using, it will remain invisible for all of the package installation, loading and update mechanisms, making it as if it didn't exist.

Package self-documentation

Amongst its member, a package (the `NameBlock`) includes attributes whose function is self-documentation of the said package. From these we can highlight:

- `Text _.autodoc.name`: the package name.
- `Real _.autodoc.version.high`: the first version number or high version.
- `Real _.autodoc.version.low`: the second version number or low version.
- `Text _.autodoc.brief`: a brief description of the package.
- `Text _.autodoc.description`: a more detailed description of the package.
- `Set _.autodoc.keys`: a set of keywords.
- `Set _.autodoc.authors`: a set of the package's author's.
- `Set _.autodoc.dependencies`: a set of the package's dependencies.
- `Text _.autodoc.minTolVersion`: the minimal compatible *TOL* version.

3.4.3 Package management (`TolPackage`)

The module for *TOL* package management is `TolPackage` It handles a *nameblock* which is structured in such a way that it's sub-models handle different functionalities. The include; installed package management, package repository interaction, package construction, amongst others.

The `TolPackage` module has different version numbers available (as if we were dealing with a package) which aid the management of changes and upgrades.

Moving on, we now look at some basic notions relating to the use of `TolPackage`, valid from version 2 (2.X) onwards, which has been distributed with *TOL* from version 3.1 onwards (more specifically `v3.1 p011`).

Package installation

A package is automatically installed the first time it is required (`#Require`). However, installation can be carried out manually via the `InstallPackage` function, indicating the identifier (name and version) of the package desired for installation:

```
Real TolPackage:InstallPackage(<packageIdentifier>);
```

If we don't know the most recent version we can use the `InstallLastCompatible` function and simply indicate the package name:

```
Real TolPackage:InstallLastCompatible(<packageName>);
```

Update and upgrade (*update & upgrade*)

From time-to-time, new versions of a package are created. We need these to allow us to stay up-to-date.

The packages system identifies two placement mechanisms:

- A package update (*update*), which consists of the subsitution of the package for another more recent one, which has the same version numbers
- A package upgrade (*upgrade*) consists of the installation of a new version of the package. Note that the previous version isn't uninstalled.

A package update usually includes important corrections, as the reconstruction of of a new package with the same version discards the previous one.

To update a package we can use the `Update` function, indicating the package name:

```
Real TolPackage::Update(<packageName>);
```

We can also update all the packages we have installed using the `UpdateAll` function:

```
Real TolPackage::UpdateAll(?);
```

A package upgrade is equally recommendable, as in many cases they include important advances in the implemented procedures and algorithms.

To improve a unique package we can use the `Upgrade` function:

```
Real TolPackage::Upgrade(<packageName>);
```

And in a similar way, the `UpgradeAll` function, to improve all of the installed packages:

```
Real TolPackage::UpgradeAll(?);
```

Low version upgrade (*low-upgrade*)

On the odd occasion, we may wish to upgrade the low version *versión baja* (second version number) of a package maintaining the high-version as it is. For this we have the `LowUpgrade` function, which we can call using the package name and the first version number (*versión alta*):

```
Real TolPackage::LowUpgrade(<packageName.packageVersion>);
```

Bear in mind that if we only indicated the package name, an attempt would be made to upgrade the low version of all of the different high versions of the package found installed.

If we wish to include all of the upgrades in a package, we have the function `FullUpgrade` which joins calls to `Upgrade` and `LowUpgrade`:

```
Real TolPackage::FullUpgrade(<packageName>);
```

We also have version for all installed packages available: `LowUpgradeAll` and `FullUpgradeAll`.

3.4.4 Package repositories

Up until this point we have only made general reference to package repositories without explaining them in any real detail. The package repositories are basically the areas where packages are both stored and distributed for installation and usage.

The internal architecture of repositories isn't of particular interest in terms of use, except in that it contains an indispensable *PHP* interface that facilitates communication and the obtaining of packages.

For example, the corresponding *URL* for the offical *TOL* packages repository is: http://packages.tol-project.org/OfficialTolArchiveNetwork/repository.php (ruta *php*).

Known repositories

The `TolPackage` module makes communication with repositories easier, by managing the installation, updates and upgrades of packages.

The list of known repositories that consult `TolPackage` can be found in the configuration variable:

```
Set TolConfigManager::Config::Upgrading::TolPackage::Repositories;
```

By default, this variable ony contains the URL of the offical repository of packages in *TOL*, although this can be edited in the normal way through the `TolConfigManager` (see section 3.3.1) or using the `AddRepository` function of `TolPackage`:

```
Real TolPackage::AddRepository(<newRepositoryURL>);
```

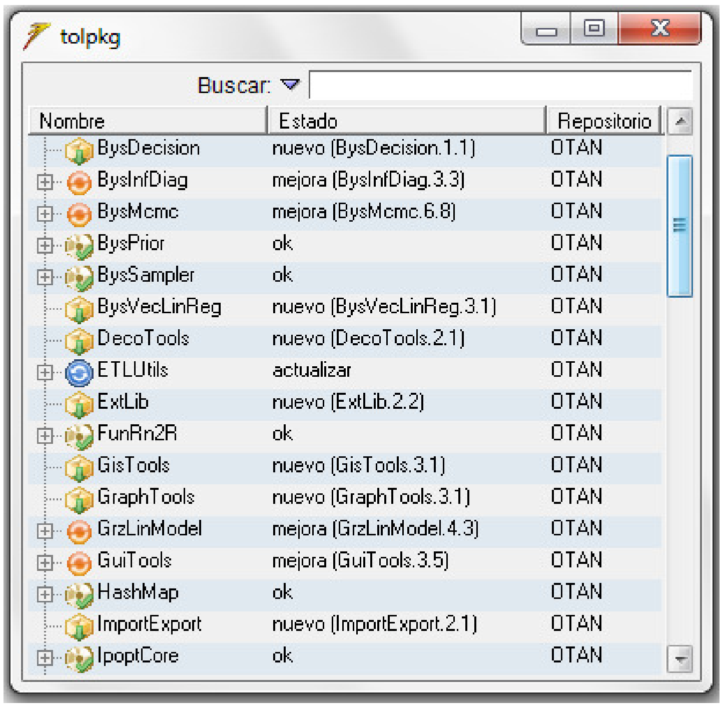
Repository website

Repositories usually have a website which allows users to see the complete list of available packages, inspect all of their characteristics or even download them.

The offical repository of *TOL* packages websites is known as *OTAN* (*Official Tol Archive Network*) and can be found in the following *trac* wiki page: https://www.tol-project.org/wiki/OfficialTolArchiveNetwork (*wiki* path).

3.4.5 Package-management graphic interface

From version 3 onwards, *TOLBase* includes a graphic interface especially designed to make the management of packages easier. It can be accessed from the Tools menu, in the Package management option.



Nombre	Estado	Repositorio
BysDecision	nuevo (BysDecision.1.1)	OTAN
BysInfDiag	mejora (BysInfDiag.3.3)	OTAN
BysMcmc	mejora (BysMcmc.6.8)	OTAN
BysPrior	ok	OTAN
BysSampler	ok	OTAN
BysVecLinReg	nuevo (BysVecLinReg.3.1)	OTAN
DecoTools	nuevo (DecoTools.2.1)	OTAN
ETLUtils	actualizar	OTAN
ExtLib	nuevo (ExtLib.2.2)	OTAN
FunRn2R	ok	OTAN
GisTools	nuevo (GisTools.3.1)	OTAN
GraphTools	nuevo (GraphTools.3.1)	OTAN
GrzLinModel	mejora (GrzLinModel.4.3)	OTAN
GuiTools	mejora (GuiTools.3.5)	OTAN
HashMap	ok	OTAN
ImportExport	nuevo (ImportExport.2.1)	OTAN
IpoptCore	ok	OTAN

Figure 3.4.1: *TOLBase* package-management interface.

The interface offers a list of all available packages from all of the of the repositories to which they have access for installation, updates or upgrades.

Different package states are displayed in the form of various different icons:

- 📦: New, not installed.
- 🔄: Update required; a more recent package of an identical version is available.
- 🟠: Upgrade required, a package with a superior version is available.
- ❗: Upgrade and update required, both of these are required periodically.
- ✓: Package okay, currently installed without requirement for update or upgrade .

Various actions can be carried out on packages. These are offered in the contextual menu.

3.4.6 Package source-code

TOL package source-code is managed via a version-control system called *SVN* (see: http://subversion.apache.org) linked to the project *trac*.

Public code servers

TOL development has two public development *tracs*:

- The main *TOL-Project trac*: https://www.tol-project.org.
- El *MMS* (*Model Management System*) *trac*: https://mms.tol-project.org.

Both of these *tracs* have their corresponding code-server *SVN*, although they share package repositories. See 3.4.4.

For further information about *MMS* please refer to the appropriate manual.

Download code (*svn* path)

Package code can be found organised in folders with each package name in the path: https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork (ruta *svn*).

We need to use an *SVN* program client to download the code.

For example, to download all of the *OTAN* packages to a local directory from the command-line interface, we would write:

```
svn co https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork/
  c:/users/<user>/svn/OTAN
```

In place of using instructions in the command-line interface, we can use a graphic interface application such as *TortoiseSVN* (http://tortoisesvn.net).

Consult code (*browser* path)

If we simply wish to consult a particular detail, or we are interested in seeing various functionality changes to do with tickets,milestones (*milestones*) or *trac* components, we can consult the code from the web explorer available in *trac*: https://www.tol-project.org/browser/tolp/OfficialTolArchiveNetwork (*browser* path).

4 *TOL* packages

In section 3.4 we saw all the information relevant to the *TOL* package system, including its creation and use. In this chapter, we describe functions implemented in some of the packages related to *TOL* development and it's main trac *TOL-Project*.

We won't describe all of the packages available in *OTAN*, rather just those which can be considered to be the most useful. In the final section of the chapter we make reference to a larger set of packages, indicating both its purpose and references to places of interest relating to each package.

Remember that the complete list of packages available in the *TOL* public package repository is called *OTAN* (*Official Tol Archive Network*), and can be found at: https://www.tol-project.org/wiki/OfficialTolArchiveNetwork. See section 3.4.4.

The code for these packages can be downloaded from the *TOL-Project SVN*:
https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork (*svn* path)
or can be consulted from the *trac* website at:
https://www.tol-project.org/browser/tolp/OfficialTolArchiveNetwork (*browser* path).
For further details, see section 3.4.6.

All packages can be found in the code-server . For example, the package `GuiTools` can be found at:
https://www.tol-project.org/svn/tolp/OfficialTolArchiveNetwork/GuiTools (*svn*path) or at:
https://www.tol-project.org/browser/tolp/OfficialTolArchiveNetwork/GuiTools (*browser* path).
We then simply indicate: OTAN/GuiTools.

4.1 `StdLib`

In the early days of *TOL*, the basic encapsualtion unit was the file. Specific functions were drawn from a set of files which all related to specific functionailty.

TOL is distributed with a special pre-installed directory. This contains a set of files relating to the most common/standardised functions, in order to maker user-access more straightforward. This set of function is called *StdLib*, an abbreviation of the term *Standard Library*.

`StdLib` package

This file-based structure is very inflexible. Therefore, with the advent of package-based modularization the `StdLib` is now distributed as a package.

Currently, `StdLib` is a large module that contains a set of functions for a host of different uses. In order to gain more modularity in the future, we are working to divide it into ever smaller and more atomic packages, dedicated to more specfic functionalities.

Bloque `TolCore`

Some functions, data structures and constants can't be extracted from the *TOL* distribution. This is either because they are so closely linked to a number of *TOL*'s internal functions, or as they were essential for the `StdLib` itself. This is the also the case with the `TolPackage` internal module. This set of functions have remained grouped together in an internal *nameblock* called `TolCore`.

All of the member functions of these *nameblocks*, both those of the `StdLib` and those of the `TolCore` are automatically exported to the global domain to aid compatibility with older code.

4.1.2 Bloque `TolCore`

The code corresponding to the `TolCore` block continues to be located in a directory called stdlib, together with the language code. Consult the code/server path:
https://www.tol-project.org/svn/tolp/trunk/tol/stdlib (*svn* path) or
https://www.tol-project.org/browser/tolp/trunk/tol/stdlib (*browser* router),
in front, simply stdlib.

Amongst the functionalities belonging to `TolCore` we find:

- **Package/management functions: implemented in the** `TolPackage` module. See section 3.4.3. The source-code can be found at: stdlib/TolPackage.
- **Internal structures: structures (**`Struct`) referenced in the internal function `Estimate` used in the estimation of `ARIMA` models. See section2.4.6. The source/code can be found at: stdlib/arima.
- ***TOL* configuration functions: implemented in the** `TolConfigManager` module. See section 3.3.1. The source-code can be found at: stdlib/TolConfigManager.
- **Functions for downloading remote files:**`CurlApi`. See section 3.2.2. The source/code can be found at: stdlib/system/CurlApi.
- **Functions for reading/writing files and compressed directories: implemented in the** `PackArchive` module. The source-code can be found at: stdlib/system/PackArchive.

4.1.3 `StdLib` package

However, as has already been mentioned, the `StdLib` package contains functions with very diverse uses. As such, it is currently being revised in the hope of creating greater modularity. In future versions its content will be split up into different packages.

Let's now take a look at some of the currently available functions:

- Access to databases: `Implemented in the DBConnect` module. Unified mechanisms are offered to establish connections to different databases. The source-code can be found at: OTAN/StdLib/data/db.
- **Interpolation functions based in the *API* of *GSL* and *AlgLib*.** The source-code can be explored at: OTAN/StdLib/math/interpolate.
- **lineales precondicionados** solvers: these contain routines for solving symmetric linear systems and badly-conditioned symmetrics. We can revise the source-code at OTAN/StdLib/math/linalg. Also explore these objects:
    - `Code StdLib::SolvePrecondSym`
    - `Code StdLib::SolvePrecondUnsym`

- Funciones de optimización : using *Marquardt*, linear programming (based in the*Rglpk* package of *R*, `StdLib::Rglpk`) and quadratic programming (based in *Rquadprog* in *R*, `StdLib::Rquadprog`). We can find the source-code at: OTAN/StdLib/math/optim.
- **Functions for running *R* from *TOL***: these implement functions that aid the running of written functions in *R*. We can look at two examples of use in the implementation of the `StdLib::Rglp::solveLP` and `StdLib::Rquadprog::solveQP` functions. The source-code can be explored at OTAN/StdLib/math/Rapi.
- **Function for estimating univariate density: based in the** *density* function of *R*. Explore the `StdLib::Rkde` module. The source-code can be found at: OTAN/StdLib/math/stat/kde.

4.2 GuiTools

GuiTools *is a package that allows the use and extension of the graphic interface's funcionalities (GUI*: *Graphical User Interface*) using *TOL* language.

Although conceived independently of the graphic platform, the current implementation only works completely for the *TOLBase* graphic interface, which is implemented in *Tcl-Tk*.

Amongst the most frequently-used functionalities of this package are the module dedicated to:

- Personalisation and management of images: `GuiTools::ImageManager` y
- Personalisation and management of contextual menus: `GuiTools::MenuManager`.

4.2.1 Image management (ImageManager)

Image registry

The image manager carries out two functions inside the `ImageManager` module. These allow images to be registered from a file or a text with an encoded image using *base64*:

- `Real defineImageFromFile(Text imageName, Text imagePath)` registers an image contained in a file. The image can be referenced later on by the identifer given in `imageName`.
- `Real defineImageFromData(Text imageName, Text imageData)` registers an image from a text that contains the coding of the image-date in *base64*. The image can be referenced later on by the identifer given in `imageName`.

Example:

To obtain an image coded in *base64*, *TOL* has a function that allows us to obtain said coding from a file: `EncodeBase64FromFile`. For example

```
Text image_base64 = EncodeBase64FromFile(".../image.gif");
```

The content of the resulting variable has to be passed on as an argument to the `defineImageFromData` function, to register the image for example:

```
Real GuiTools::ImageManager::defineImageFromData("my_image", image_base64);
```

Assigning icons to classes

objetos which are class instances are displayed in the object inspector with the same icon that is common to all objects of the type `NameBlock`. This does sometimes make it difficult to differentiate between objects of different classes. `ImageManager` this contains a function making it possible to associate an icon with instances of a class.

- `Real setIconForClass(Text className, Anything imageName)` defines the image that will be shown in the object inspector for instances of a class with name `className`. The `imageName` argument can be of either `Text` or `Code` type. When it is of `Text` type, it is interpreted as the name of a registered image. When it is of `Code` type, it is interpreted as a function object that returns the name of the registered image. In the case of the latter, the function receives the selected instance as an argument and can decide the appropriate function with respect to the object state.

Example:

Let's now use an example to illustrate how to personalise a class by assigning it an icon.

```
#Require GuiTools;

// A class is created:
Class @TestA {
  Real value
};

// An icon is registered via its base64 coding:
Real GuiTools::ImageManager::defineImageFromData("checkedBox",
  "R0lGODdhCwALAJEAAH9/f////wAAAP///ywAAAAACwALAAACLISPRvEPAE8oAMUXCYAg"
  "JSEiAYRIQkSCAgTJjgiAoEgSEQGEJIRiA9wdwUcrADs=");

// The icon is assigned to the class created.
Real GuiTools::ImageManager::setIconForClass("@TestA", "checkedBox");

// An instance of a class is created
@TestA inst1 = [[ Real value = 1.0 ]];
```

4.2.2 Contextual menu management (`MenuManager`)

If the user right-clicks over the *TOLBase* object inspector variables panel, a contextual menu with options relative to the active selection will open (see figure 4.2.1). The active selection can contain various *TOL* objects and the contents of the menu depend on the composition of this selection.

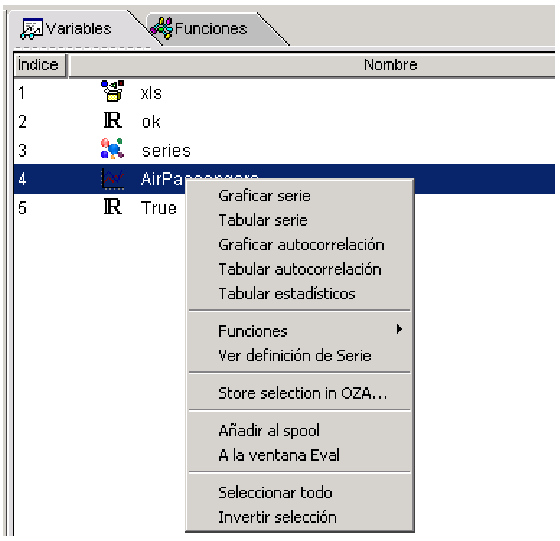


Figure 4.2.1: Contextual menu opened over a `Serie` object type.

When there is a multiple selection, involving different types of objects, the menu is organised into submenus, one for each type of data.

With the `MenuManager` module, it's possible to extend the set of menu options available by default in *TOLBase*.

Arguments to define a contextual menu

When we define a menu option we have to specify the set of arguments used to configure its appearance and functionality. The menu-options builder receives a*NameBlock* `which contains the arguments of the menu-option being defined. The only argument that is required at the moment of defining a menu option is its name (Text name)`. Bear in mind that this must be unique. For example

```
NameBlock menuOption_arguments = [[
  Text name = "Entrada1"
]];
```

The attributes that configure the visual appearance of a menu entry are:

- `Text label`: this contains the etiquette that is displayed in the menu for the defined entry. If it isn't defined, it is assumed to be the same as the name.
- `Text image`: this contains the name of an image registered in `ImageManager` (see section 4.2.1), which takes the vale `""` by default and doesn't display an image in the menu entry.
- `Real rank`: contains a numerical value which is used to order entries within a submenu. By default, this becomes the value `0`. Entries with the same `rank` value are ordered according the definition order.

The attributes that configure the functionality of a menu entry are:

- `Real flagGroup`: A boolean value that specifies if the action is applied to an instance or a set of instances. If `0` (false), the option appears as available in the menu when the active selection only contains an object of the associated type: In the opposite case `!=0` (true), it appears as available in the menu when the active selection contains more than one object of the associated type.
- `Code|Set CmdInvoke`: can be a `Code` object or a `Set` that contains a `Code` object. The `Code` object is a function that must return a `Real` (which is ignored) and is used as an action to be run when the menu-option is selected. The function takes the object contained in the active selection as a first argument (if `flagGroup` is false) or object-set (if `flagGroup` is true). Additionally, it receives in the second argument a set (`Set`) with the own parameter data defined for the option ******** Literal translation. This set of parameters is specified at the time of defining the option.
- `Code|Set CmdCheck`: can be a `Code` object or a`Set` that contains a `Code` object. `Code` is a user function that is used to manage the state in which the menu-option is displayed. The function should return `!=0` (true) to indicate that the option should appear as activated or `0` (false) to indicate that the options should appear as disactivated. It takes the same arguments as the function referenced in the `CmdInvoke` attribute.

- `Text delegateOn`: the value of the `delegateOn` attribute is as an expression to be evaluated on the objective object,returned as a result on which the (`Check` or `Invoke` action) will be run.

We now describe the functions available inside the `MenuManager` module, specifically those used for the definition of contextual menu options associated with data-types.

- `Real defineMenuCommand(Text typeName, NameBlock args)`: this defines an option available for objects of the data-type given in `typeName`. It's not actually necessary for the data-type to exist at the moment of defining the option. The `NameBlock args` argument contains the attributes that specify the visual appearance and action associated with the option.
- `Real replaceMenuCommand(Text typeName, NameBlock args)`:similar to `defineMenuCommand` but if the option is already defined, it replaces its definition with the new attributes given in `args`.
- `Real defineOptionLabel(NameBlock args)` : this defines the visual appearance of a menu label. This applies to labels used for sub-menu entries such as the one needed to group the menu options for each data-type included in the active selection or for sub-menus defined in the name of other menu entries.
- `Real replaceOptionLabel(NameBlock args)` : similar to `defineOptionLabel`but if the oprion is already defined, it replaces its definition with the new attributes given in `args`.

The menu options can be shared by more than one data-type. To do this we must define the menu-option once for the first data-type. For all other data-types we simply indicate the the name of the already created menu-option as an argument.

Contextual sub-menus

Entries in the contextual menu can be organised into sub-menus. To do this, we specify the name of the hierarchical<sub-menu structure using the t0/>/ seperator.

For example the name of the entry `"Padre1/Padre2/Entrada"` defines `"Padre1"` as a sub-menu type menu entry in the superior level. `"Padre1/Padre2"` is defined as a sub-menu type menu entry inside the sub-menu `"Padre1" and "Padre1/Padre2/Entrada"` as a menu entry inside the sub-menu `"Padre1/Padre2"`. The visual appearance of the entries associated with the sub-menus `"Padre1"` and `"Padre1/Padre2"` can be configured with either the `defineOptionLabel` function, or the `replaceOptionLabel` function.

<u>Example:</u>

We finish this section by using a simple example that illustrates the functionailty of `MenuManager`.

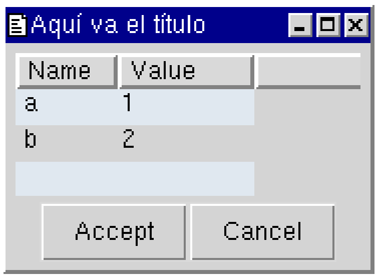
```
#Require GuiTools;

// We want to associate a new menu-option with real numbers

// We create a function that will take the real number
// and an extra set of arguments that we won't use:
Real RealSquare(Real x, Set args) {
  Real sqX = x*x;
  WriteLn( "El cuadrado de " << x << " es " << sqX );
  sqX
};

// We create the menu option for the data-type "Real"
Real GuiTools::MenuManager::defineMenuCommand("Real", [[
  Text name = "Real_SQ";
  Text label = "Square of Real";
  Real flagGroup = 0;
  Set CmdInvoke = [[ RealSquare ]]
]]);
```

4.2.3 Container editing

Another of the functions included in `GuiTools` is that which allows us to interactively edit member-data of a `Set` or `NameBlock`, which here we will call *contenedor*. Upon running this editing function, a dialogue window opens in which we can edit values associated with the container member-data.

Editing can be done in two states: modal or non-modal. In the non-modal state, the editing function opens the window and returns immediately, continuing its evaluation with the instructions below. During this process the editing window remains open. When run in modal state, the function doesn't return until we have closed the editing window.

For example, in figure4.2.2 we can see the window that appears upon commencing the editing of the elements of a `Set` or`NameBlock` with two items of member-data.



Name	Value
a	1
b	2

Figure 4.2:Example of an editing window for a container object `Set` or `NameBlock`.

The functions that permit the editing of container member data are:

- `NameBlock TkNameBlockEditor(NameBlock options, Set args)` this opens a window, which contains a table, where the values of member-data of the t2/>NameBlock options can be edited.
- `Set TkSetEditor(Set options, Set args)` this opens a window, which contains a table, where the values of member-data of the `Set options` can be edited.

The argument `Set args` is a set with pair cardinalidad. Elements in odd positions are interpreted as an option name and the following element as its value, e.g. `[["-modal", "yes", ... ]]` specifies the option with namet <6/>"-modal" and value `"yes"`. In both functions the argument `Set args` defines the state in which the window is open:

- **modal**: when the argument `args` is the empty set, the modal editing is then carried out. This can also be specified with `Set args = [["-modal", "yes"]]`.
- **no modal**: is specified by including the option `"-modal"` with valie `"no"` in the arguments. Furthermore, the following options should be included:

  - "-address": with an equal value to the physical address of the object to be edited. The physical address is obtained via the GetAddressFromObject function.
  - "-tolcommit": with an equal value to the name of the function to be run once the edited data is accepted. This function takes the original object that is being edited as a first argument, as well as a copy of the edited data.

In addition to the previous options, the following ones can also be used:

- "-showbuttons": determines whether the Aceptar/Cancelar buttons are displayed in the window. It can take Boolean values 0 or 1 of Real type, or "yes" or "no" of Text type. By default it takes the value 1, which is to say that it displays the buttons.
- "-title": takes a Text-type value, which is uses the window's title. By default, one of the values "Edit Set" or "Edit NameBlock" is used, depending on if the object is a Set or NameBlock respectively.
- "-checkchanges": determines if a warning message appears or not when the window is closed, when there are pending unsaved changes in the original object. It can take boolean values 0 or 1 of Real type, or "yes" or "no" of Text type. By default, it takes the value 1; that is to say that upon closing the window, if there are pending unsaved changes, the user is warned and given the opportunity to save if necessary

The following examples illustrate the use of the aforementioned editing functions:

Example of the editor in modal form:

```
#Require GuiTools;
Set s1 = { [[ Real a = 1; Real b = 2  ]] };
Set s2 = GuiTools::TkSetEditor(s1, Empty);
```

Example of the editor in non-modal form:

```
#Require GuiTools;
Set s1 = { [[ Real a = 1; Real b = 2  ]] };

Function instigated in the event of accepting the editing window:
Real ApplyChanges(Set to, Set from) {
  to::a := from::a;
  to::b := from::b;
  Real 1
};

Set s2 = GuiTools::TkSetEditor(s1, [[
  "-title", "Título",
  "-modal", "no",
  "-tolcommit", "ApplyChanges",
  "-address", GetAddressFromObject(s1)
]]);

WriteLn("Sigue la ejecuación...");
```

The windows opened in the two previous examples are similar. The only difference is that, in the first, the evaluation is delayed until the editing window closes while in the second the code evaluation continues

4.3 `TolExcel`

The `TolExcel` package implements functions for the input and output from files *Excel 97-2003* (.xls). Let's now look at the basic elements involved in the use of `TolExcel`.

`TolExcel` defines a main class called`@WorkBook`, which allows us to instance objects associated with *Excel* files. The class methods which allow us to create `@WorkBook` instances are `Open` and `New`:

- `@WorkBook xls = @WorkBook::Open( Text path );  creates an instance from the existing file, with the indicated path in the argument path`. The file can't be open at the same time as *Excel* as`TolExcel` requests access to an exclusive file.
- `@WorkBook xls = @WorkBook::New( Text path, Anything wsInfo );` creates an instance of a new file, the new file is created in the path indicated in the argument`path`. The argument `wsInfo` specifies the worksheets that the file will contain. If `wsInfo` is a `Real`number, an object will be created with the corresponding number of worksheets. If it is a (`Set`), as many worksheets as there are set-elements will be created. If the - nth element of the set is a (`Text`), the -nth worksheet will have the name of the value of this element. In the opposite scenario, it will take the default name assigned to it by *Excel*.

The changes made in a `@WorkBook` instance can be stored on disk using the following methods:

- `Real xls::Save(Real void)`: saves the changes made in the `xls` instance in the relevant associated file.
- `Real xls::SaveAs(Text path)`: saves the changes made in the `xls` instance in the file indicated in the `path`. From this point on, the file specified in the argument `path` is associated in such a way that future invocations of the `Save` method will affect this file.

The majority of the methods of a `@WorkBook xls` work on cells or cell-ranges relative to the active worksheet. By default, an instance of `@WorkBook` establishes the first worksheet as active. If we wish to change the worksheet we can run the methods `ActivateWS` and `ActivateNamedWS`:

- `Real xls::ActivateWS(Real workSheetNumber)` : changes the active worksheet. The new active worksheet is the one that appears in the index which is equal to the value of the argument `workSheetNumber`. Valid indexes start from 1 until the number of worksheets of the instance. The method returns `True` if it has been possible to change the active worksheet, or `False` in the case that it hasn't.
- `Real ActivateNamedWS(Text workSheetName)`: establishes the active worksheet as the one whose name coincides with the value of the argument `workSheetName`. The method returns True if it has been possible to change the active worksheet, or False in the case that it hasn't.

After working with the instance, we should close the connection with the file via the `Close` method:

- `Real xls::Close(Real void)` : closes the connection with the physical file, freeing up all associated resources. From the moment `Close` is run onwards, the instance will remain invalid. It's the programmer's responsibility to run one of the methods `Save` or`SaveAs` prior to updating the file content with the changes made safely stored in the memory.

There are some reading function for cells and and ranges that allow stored values to be read, without having to specify a data-type for the result in TOL. In these cases, as good an effort as possible is made to infer what the data-type that corresponds to TOL is. The possible data-types that we can obtain are `Real`, `Text` and `Date`. If the previous data-types aren't recognised, the value `Text ""` will be returned.

The reading and writing functions take a cell-reference or cell-range as arguments following the follow rules:

- **cell**: given by a pair whole indices `(row,col)` where `row` indicates the row and `col` the numeric column number, taking the value 1 as the index of column`A` and increasing accordingly. For example, the cell `B3` has `(3,2)` as its coordinates.
- **range**: given by two pairs, esquina and extension. The corner of the range is `(row_ini,col_ini)` which indicates the cell coordinates and extension given by `(row_num,col_num)` which indicates that the ranges extends from the first cell `row_num` in descending rows and `col_num` from left to right. For example, the range `A1:D1` would be specified as `(row_ini,col_ini)=(1,1)` y `(row_num,col_num)=(1,4).`

Below are some of the functions used to read values from the cells of an active worksheet:

- `Set xls::ReadRange(Real row_ini, Real col_ini, Real row_num, Real col_num, Set colDef)` : reads the cell range specified by the initial cell in coordinates `(row_ini, col_ini)` and extended `row_num` in descending rows `col_num` left to right. The result is a set in which each element is also a set that corresponds with a row from the read field.
- `VMatrix ReadVMatrix(Real row_ini, Real col_ini, Real row_num, Real col_num)` : reads the matrix contained in the cell range specified by the initial cell in coordinates `(row_ini, col_ini)` and extended `row_num` in descending rows `col_num` left to right. The result is a `VMatrix` object. Empty or non-numeric cells are read as the default value.
- `Matrix ReadMatrix(Real row_ini, Real col_ini, Real row_num, Real col_num)` : similar to `ReadVMatrix`, except for the fact that a `Matrix` type object is the result.
- `Set ReadSeriesByCol(Real row_ini, Real col_ini, Real row_num, Real col_num, TimeSet dating, Text dateFormat)`: reads a set of series contained in the range specified by the arguments (`row_ini,col_ini, row_num, col_num)`. The first row of the range is the one which contains the data headings. The first column contains the dates of the series, those which are specified in the time-set in the `dating` argument, if `dating` is `W` therefore the name of the date is read from the cell heading of the first column. When no time-set is specified then `C` is used as default.

The columns, from the second one onwards, contain the data of those series to which the content of the corresponding cell heading is assigned as a name. Dates are interpreted according to the format specified in the `dateFormat` argument. If the value of this argument is "", then they are interpreted according to TOL's default format, by making use of the `TextToDate` function.

- `Set GetFullSeriesByCol(TimeSet dating, Text dateFormat)`: similar to the `ReadSeriesByCol` function. It takes its range from the minimum range that contains data in the active worksheet.
- As well as the aforementioned methods of reading, we have the following one for writing in the active worksheet's cell ranges:
- `Anything WriteCell(Real row, Real col, Anything value)`: writes a value in the cell with coordinates `(row,col)`.
- `Real WriteRange(Real row, Real col, Set cellValues)` : writes a set of sets starting from the cell with coordinates `(row,col)`. The resulting range will have as many rows as the subsets have `cellValues`, and as many columns as the longitude of the largest subset.
- `Real WriteMatrix(Real row, Real col, Matrix values)`: writes the matrix `values` starting from the cell with coordinates `(row,col)`. The resulting range will have the file/column dimensions as the input matrix.
- `Real WriteVMatrix(Real row, Real col, VMatrix values)`: similar to `WriteMatrix`.
- `Real WriteSeries(Real row, Real col, Set series, Text dateFormat)`: writes a series-set starting from the cell specified by the coordinates `(row,col)`. The date column is written according to the format specified in the `dateFormat` argument.

Example:

Lastly, let's illustrate the use of `TolExcel` by way of an example. Let's take a look at the time-series of international air passengers initially referenced in the book *«Time Series: Forecast and Control by Box, Jenkins and Reinsel»* (ISBN: 978-0470272848). The *Excel* data-file can be downloaded from the *url*: http://packages.tol-project.org/data/AirPassangers.xls. This file contains the monthly total of international passengers covering the period between January 1949 to December 1960.

We'll now show the *TOL* code used for loading data, together with an image of the graph of said data (see figure 4.3.1).

```
#Require TolExcel;
// abro el excel
TolExcel::@WorkBook xls =
    TolExcel::@WorkBook::Open("AirPassangers.xls");
Real xls::ActivateWS(1); // activate worksheet 1
// I read the data-series contained in the active worksheet
Set series = xls::GetFullSeriesByCol( TimeSet W, Text "" );
Serie AirPassangers = series[1];
// I close the connection with the Excel file.
Real xls::Close(?);
```

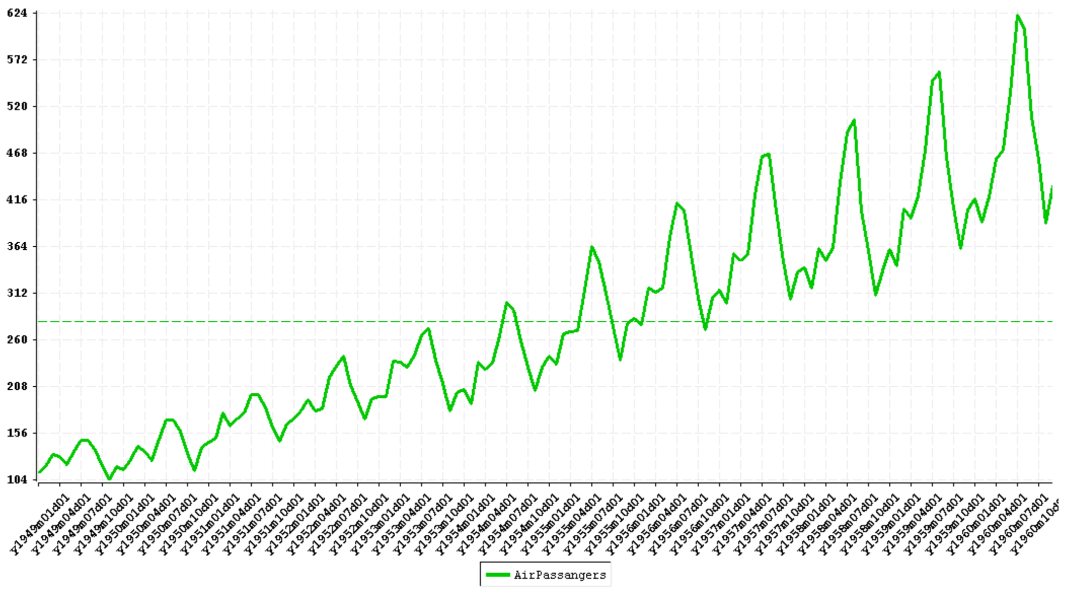


Figure 4.3.1: Graph generated in *TOLBase*, showing the data for the passenger total in the example.

4.4 Other packages

TOL's official package repository (*OTAN*) offers other packages related with concepts such as optimisation, *gis*, modelling, linear algebra and so on. In this section, we'll give a general overview of some of the ones that may be of most interest.

4.4.1 Paquetes de *MMS*

As mentioned in section 3.4.6, some of the public packages available in the *OTAN* repository have been developed in a different *trac* frame, one oriented towards the creation of a statistical-model specification system known as *MMS* (*Model Management System*).

This `MMS` package, along with others such as `RandVar` or `DecoTools`, is managed from this other *trac*. For further information about these packages, consult the project's website https://mms.tol-project.org or the corresponding *MMS* manual.

4.4.2 `TolGlpk`

`k` TolGlpk runs a native interface with the *GLPK* package (*GNU Linear Programming Kit*, http://www.gnu.org/software/glpk). Although similar in terms of functionality to `Rglpk` (contenido en *StdLib*), it's more efficent. This is due to the fact that it establishes a direct link with the functionality run in *C* without having to pass through *R*. For this reason we recommend its use in place of *Rglpk* when it's necessary to resolve linear optimisation problems. Additionally, we can run other options *Rglpk*. The package's source-code can be explored at: OTAN/TolGlpk.

4.4.3 `NonLinGloOpt`

NonLinGloOpt runs a native interface with the*NLopt* package (http://ab-initio.mit.edu/nlopt). This offers functionality for the optimisation of non-linear functions with equality and non-linear inequality functions. There is a documentation page available for this package at: https://www.tol-project.org/wiki/OfficialTolArchiveNetworkNonLinGloOpt.

4.4.4 `TolIpopt`

TolIpopt is another non-linear optimisation package based on the interior point method. Found within the package, run in *C++*, *IPOPT* (*Interior Point OPTimizer*, https://projects.coin-or.org/Ipopt). The package's source-code and some examples can be found via the link: OTAN/TolIpopt.

4.4.5 `MatQuery`

MatQuery `runs consult, select and classify functions for matrices. For` or the `EvalSet`. This package is a good example of the efficient use of matrix operators. The source-code can be explored at: OTAN/MatQuery.

4.4.6 `BysMcmc`

BysMcmc allows *cadenas de markov* (*MCMC*) to be created for the parameters of hierarchical linear models with normal errors. Amongst the various model characteristics, it is worth highlighting the possibility of specificying an *ARIMA* structure for errors, linear parameter priori information, non-linear effects, linear restrictions etc.

For a broader explanation of the theory applied in this package, we recommend reading the document: OTAN/BysMcmc/bsr/doc/BSR_Bayesian_Sparse_Regression.pdf.

The most convenient way of using this package is through the *MMS* package. The package's source-code can be explored from OTAN/BysMcmc.

Indices

Figure index

Table index

TOL manual
1st edition. Fecha: 13/06/2013

Authors: Pedro Gea y Jorge S. Pérez.
Collaboraters: Miguel Á. Fernández and Claudia Escalonilla.
Editor: Pedro Gea.

Correspondence: pgea@bayesinf.com.

Bayes Forecast
www.bayesforecast.com
www.tol-project.org