AMAPmod v1.8. Introduction and reference manual

Christophe Godin, Yann Guédon

AMAPmod

Introduction and Reference Manual

Version 1.8

Christophe Godin

Yann Guédon

With software contribution of:
Samir Bellouti
Pascal Ferraro
Christophe Nouguier
Nicolas Dones
Boris Adam

And collaboration of :
Yves Caraglio
Evelyne Costes
Hervé Sinoquet

Edition : Marie-Hélène Lafond

AMAPmod : Exploring and Modeling Plant Architecture
Linux version

Christophe Godin, Yann Guedon

CIRAD/INRA - UMR Modelisation des Plantes

with contribution of:
Samir Bellouti, Nicolas Dones,
Pascal Ferraro, Boris Adam,
Christophe Nouguier

Open Software used: gnuplot v3.7, readline v2.2 (GNU).

Forum for AMAPmod users : amapmod@cirad.fr
Send reports on bugs or comments: aml@cirad.fr

Forum archive:http://www.cirad.fr/mail-archives/listes/amapmod
Online doc :http://amap.cirad.fr/amapmod/referman18/couverture.html
Other infos: http://www.cirad.fr/presentation/programmes/amap/logiciels/amap_mod

Cover picture credits: 3D Reconstruction image with AMAPmod of apple tree digitized by E. Costes (UFR d'arboriculture fruitière, INRA-ENSAM Montpellier) and H. Sinoquet (PIAF, INRA Clermont Ferrand)

Printed November, 2001

Table of Contents

PART II REFERENCE MANUAL ---------- 6-1

Part I AMAPMOD

1 INTRODUCTION

In order to better control economical outcomes, a new trend in agronomic research consists of determining how production variables (*e.g.* wood biomass and quality, fruit quantity and quality) are distributed within plant architecture. The AMAPmod software defines a set of methods and tools to address these questions, i.e. to measure, analyse and model plant architectures. This software has been available to the scientific community for 6 years now and has been used in the analysis of various types of plant architecture databases, *e.g.* [1; 6; 17; 28; 29; 48]. Several researchers from several institutes have contributed either to the development of concepts or to the software itself (the list of contributors is detailed on the first page of the manual). AMAPmod consists of about 200 000 lines of C++ code and its modular architecture makes it possible to add specialised modules. It is is free of charges. Moreover, it is now available on Linux plateforms and is intended to become a truely “open software”.

The AMAPmod modeling methodology is illustrated in **Figure 3-1**. The architecture of the plant (or the set of plants) is first measured according to a precise application protocol. Plant topological structures are encoded in textual forms. These code files may contain various additional types of information about plant entities (e.g. geometrical, spatial, botanical or micro-climatic information). These files describing formally plant architecture may then be read by the AMAPmod software which creates an internal representation of the measured plants (called Multiscale Tree Graphs, or "MTG"). The user can then explore the resulting MTG using the AML language, looking for regularities in the collected data. This research, driven by the final goals of the application, leads the user to identify or to confirm hypotheses about the plant development. Based on these hypotheses, a family of models can be identified as a good candidate for modeling the extracted data (as for now, the AMAPmod software contains mostly probabilistic models). For all these models, training procedures are available

to automatize the identification of the model, in the selected family, which best synthesizes the extracted data.

The AML language enables the user to access the AMAPmod primitives. It is divided in several modules: the kernel module, the MTG module, the STAT module. The Kernel module contains functions that deal with standard types like integers, reals, strings, arrays, etc. The MTG module contains functions to build formal representation of plants (MTGs) and to extract information from them. The STAT module contains functions that enable the user to analyse various type of data samples, to build probabilistic models associated with these samples.

The statistical inference approach developed in AMAPmod uses models that can account for the structural information contained in the MTG. The analysis of the information contained in an MTG requires the use of different techniques applied at different scales, dates, parts of plants. In view of the structural nature of the MTG, most of these techniques incorporate structural components (*e.g.* the graph of possible transitions for the hidden semi-Markov chain in (**Figure 3-13**).

The need for a common language to describe the architecture of a wide range of plant species and the will to share experience and tools in the exploration of plant architecture databases have been major motivations in the development of AMAPmod. Our long-term objective is to provide the agronomic community with a database management system that could be used as a standard tool. This standardisation process concerns various stages of AMAPmod methodology: plant architecture representation, its coding language, field observation protocols for given agronomic applications, macro-functions for database exploration (3D visualisation, sample extraction, ...), tools for analysing structured data, etc. Such standardisation would facilitate the constitution and diffusion of plant databases and would enable modellers to compare their models on the basis of publicly available databases.

2 INSTALLATION

This chapter describes how to install AMAPmod on UNIX machines. AMAPmod version 1.4 and greater can be run on a Linux system.

2.1 Minimum System Requirements

The AMAPmod software is available on PCs under operating system Linux.

- Linux kernel greater than 2.0.
- 30 MB free hard disk space
- 64 MB of RAM. (128 MB is recommended for large databases).

2.2 Getting the software

To obtain the latest version of AMAPmod (currrently 1.8 in november 2001), download one on the anonymous ftp site :

ftp://ftp.cirad.fr/dist/amap/AMAPmod

You must enter a valid email address as a password.

AMAPmod is now distributed in rpm format. rpm is a powerful package manager, which can be used to build, install, query, verify, update, and uninstall individual software packages. A package consists of an archive of files, and package information, including name, version, and description, some compilation options, dependencies, etc. It allows to detect at install time, missing or perturbing elements. So, installing rpm is easy and secure.

The rpm tool offer a wide variety of option, we will comment the more usefull :

Installation of a package :

```
rpm -i  package.rpm
```

Update of a package :

```
rpm -U package.rpm
```

Remove a package

```
rpm -e package
```

Searching the package containing a given file :

```
rpm -qf /dir/file
```

General information of an installed package

```
rpm -i package
```

General information of a package to install

```
rpm -qi package.rpm
```

Some graphic managers exist, like kpackage, gnorpm, rpmdrake the let install package intuitively. The designation of rpm file follow some precise rules :

projectname-version-release.processor.rpm

The release may contains informations on the distribution on which the package has been made.

For example : AMAPmod-1.8-rh62.i386.rpm contains AMAPmod project version 1.8 released on a red hat 6.2, compiled on a pentium.

An other example : AMAPmod-1.8-1.i586.rpm contains AMAPmod project version 1.8, first release compiled on a pentium II or more.

To reference the different distribution, those different prefix are used on release designation :

rh for Red Hat, mdk for Mandrake and suse for Suse.

Dependencies of dynamic AMAPmod RPM :

RPM to install to use AMAPmod :

- qt 2.*
- qt-GL 2.* (Not existant for every linux distribution)
- Mesa (OpenGL)
- libtermcap
- readline
- gnuplot
- XFree86

We recommend to have a comfortable use of Linux to install :

- Netscape
- KDE
- gftp
- xemacs and all its packages
- Gnumeric

- Xview

Downloading the sources The AMAPmod software is now distributed as open software. The source code is available at

ftp://ftp.cirad.fr/dist/amap/AMAPmod/AMAPmod-1.8-1.tgz

This is a compressed tar file which can expanded on Linux with the command :

```
tar xvzf AMAPmod-1.8-1.tgz
```

Sources can be compiled to build the AMAPmod project using the following commands:

cd AMAPmod/

make alldepends

make all

make install

The linking phase of AMAPmod depends on the availability of a number of external libraries. To get a list of these libraries, the INSTALL file in the directory AMAPmod should be checked.

Almost all the source files of AMAPmod are included in this directory. However, AMAPmod currently uses the library Tools.h++ from RogueWave inc. (for historical reasons). This software is proprietary and thus, is not included in this distribution. However, people who would like to compile AMAPmod can buy the library (see www.roguewave.com). It is our intention that AMAPmod version 2 will entirely be an open software.

If you wish to participate in the development of AMAPmod, please contact us at aml@cirad.fr.

Then, you are done with the installation of AML.

2.3 File access convention

Each time a new application is studied with AMAPmod, you may create a new directory in the user's home directory. Files used by AML can be located anywhere in the Linux hierachical file system, provided the user can access them. All references to files from within a file or from AML must be given explicitly. References to files must always be made relatively to the location where the reference is made.

If you have problems please contact :

aml@cirad.fr

or refer to the amapmod forum :

amapmod@cirad.fr

To subscribe to the list, you have to send an email at the address amapmod-request@cirad.fr and simply say "subscribe" in the mail message body. Upon subscribing, you should receive an introductory message, containing list policies and features. Save this message for future reference; it will also contain exact directions for unsubscribing. Then messages can be sent to the list using amapmod@cirad.fr.

In order to unsubscribe to the list, the introductory message contains the exact command which should be used to remove your address from the list. However, in most cases, you may simply send the command "unsubscribe" followed by the list name:

```
unsubscribe amapmod
```

(This command may fail if your provider has changed the way your address is shown in your mail.)

Please do not put your commands on the subject line; Majordomo does not process commands in the subject line.

3 AMAPMOD: AN OVERVIEW

3.1 Introduction

During the 70's, plant architecture progressively emerged as a new area of interest in different research domains, *e.g.* computer simulation [34], theoretical biology [18], botany [30; 31], agronomy [8], forestry [38], horticulture [36], plant-environment interaction modelling [45]. All these domains considered plants from very different perspectives, but they all had in common particular interest in the organisation of plant vegetative structures, *i.e.* in plant architecture.

In order to model plant architectures, people first attempted to measure plant organ parameters and other morphological characters. This was used to set the values of plant growth model parameters using empirical data, *e.g.* [10; 15; 17; 35; 40], or to better understand the organisation of plant components, *e.g.* [3; 33; 42]. Recently, studies using plant architecture have entered a second phase, where the objective is to study variations of biological phenomena within crowns, *e.g.* [7; 50]. To achieve this goal, several works proposed to record topological information of plant components and to organise other information according to topology [16; 25; 32].

Godin and Caraglio recently introduced a plant representation formalism that is able to integrate several scales of description within a single model [20]. This formal representation is the central data structure of the AMAPmod system, which is a computational platform providing tools to create, explore and analyse plant architecture databases [22]. The representation model accounts for plant architectures measured at different scales (node, annual shoot and axis for example), different dates and may integrate different types of attributes, geometrical or biological [22]. A set of dedicated statistical tools is used to identify in plant architecture remarkable structures or regularities which are not directly apparent in the data [25; 28]. Tools are also provided to compare biological structures, such as axes or branching systems, based on a comparison of their components [13; 28]. In agronomic applications, these tools have been used to characterise and compare genotype behaviour and cultural conditions [5; 6; 17; 28; 46; 50].

This chapter describes the constitution of a plant architecture database and its exploration with the AMAPmod system. It emphasises how the different techniques and tools can be combined and applied using AMAPmod. Section 3-2 gives an overview of AMAPmod and outlines the different data structures and models that can be constructed and used in the system and gives an overall description of the system. The use of AMAPmod is then illustrated in Section 3-3 on an actual-scale plant architecture database. The different steps in a typical exploration are successively illustrated using this database.

3.2 AMAPmod system overview

AMAPmod provides users with a methodology and corresponding tools to measure plants, create plant databases, analyse information extracted from these databases. This methodology can be depicted as follows (**Figure 3-1**) :

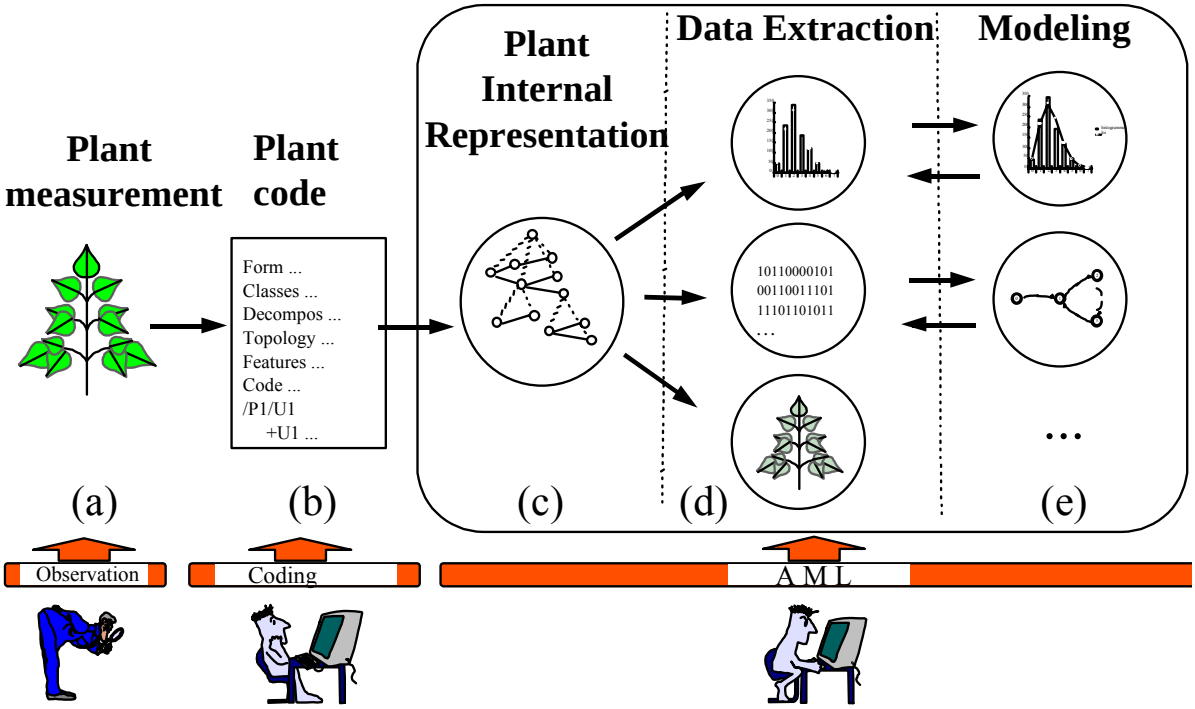


Figure 3-1 Synoptic of the AMAPmod system.

Plant architectures are described from field observations using a dedicated encoding language (**Figure 3-1a** and **1b**). These descriptions can then be decoded by the AMAPmod system which builds a specific internal representation of plant architecture (**Figure 3-1c**). The resulting database can then be analysed with various statistical analysis tools. Plants can be graphically reconstructed and vizualised in 3 dimensions. Various types of data can be extracted and analysed with different viewpoints (**Figure 3-1d**). Different families of probabilistic or stochastic models are provided in the system (**Figure 3-1e**). These models are intended to be used as advanced statistical analysis tools for exploring in greater depth the information contained in the database. All these tools are available through a querying language called AML (AMAP Modelling Language) which enables the user to work on various objects, i.e. formal representation of plants, samples of data or models. AML provides the user with a homogeneous language-based interface to load, display, save, analyse or transform each type of object. Let us briefly review these AMAPmod components.

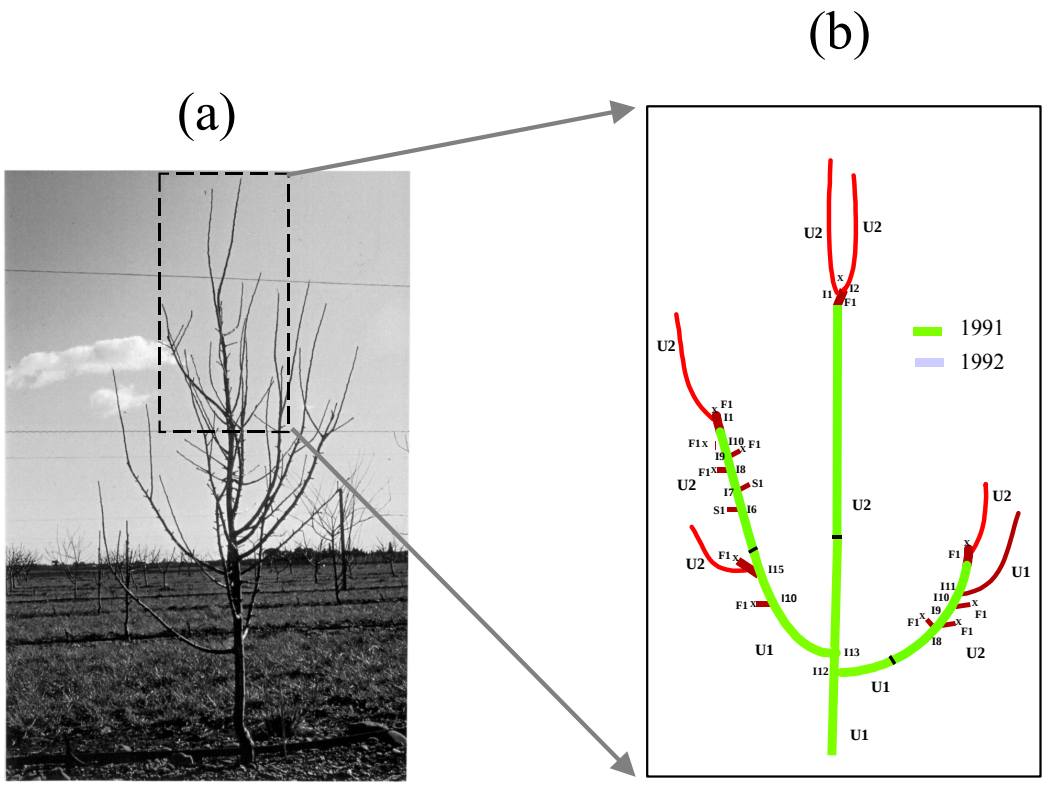
3.2.1 Plant architecture databases

Plants are formally represented in AMAPmod by multiscale tree graphs (MTGs), [20]. A MTG basically consists of a set of layered tree graphs, representing plant topology at different scales (internodes, growth units, axes, etc.). To build up MTGs from plants, plants are first broken down into plant components, organised in different scales (**Figure 3-2a** and **6b**). Components are given labels that specify their type (**Figure 3-2b**, U = growth unit, F = flowering site, S = short shoot, I = internode). These labels are then used to encode the plant architecture into a textual form. The resulting coding file (**Figure 3-2c**) can then be analysed by AMAPmod to build the corresponding MTG. Basically, in an MTG, the organisation of plant components at a given scale of detail is represented by a tree graph, where each component is represented by a vertex in the graph and edges represent the

physical connections between them. At any given scale, the plant components are linked by two types of relation, corresponding to the two basic mechanisms of plant growth, namely the apical growth and the branching processes. Apical growth is responsible for the creation of axes, by producing new components (corresponding to new portions of stem and leaves) on top of previous components. The connection between two components resulting from the apical growth is a "precedes" relation and is denoted by a '<'. On the other hand, the branching process is responsible for the creation of axillary buds (these buds can then create axillary axes with their own apical growth). The connection between two components resulting from the branching process is a "bears" relation and is denoted by a '+'. A MTG integrates - within a unique model - the different tree graph representations that correspond to the different scales at which the plant is described.

Various types of attribute can be attached to the plant components represented in the MTG, at any scale. Attributes may be geometrical (*e.g.* diameter of a stem, surface area of a leaf or 3D positioning of a plant component) or morphological (*e.g.* number of flowers, nature of the associated leaf, type of axillary production - latent bud, short shoot or long shoot -).

MTGs can be constructed from field observations using textual encoding of the plant architecture as described in [22] (**Figure 3-2**). Alternatively, code files representing plant architectures can also be constructed from simulation programs that generate artificial plants, [11]. The code files usually have a spreadsheet format and contain the description of plant topology in the first few columns and the description of attributes attached to plant components on subsequent columns.



(c)

/P1			
^/A91/U1			
	/I12+A91/U1		
	^<U2		
		/I8+A92/F1	
		/I9+A92/F1	
		/I10+A92/F1	
		/I11+A92/U1	
	^<A92/F1		
		/I1+U2	
	/I13+A91/U1		
		/I10+A92/F1	
		/I15+A92/F1	
			/I1+U2
	^<U2		
		/I6+A92/S1	
		/I7+A92/S1	
		/I8+A92/F1	
		/I9+A92/F1	
		/I10+A92/F1	
	^<A92/F1		
		/I1+U2	
^<U2			
^<A92/F1			
	/I1+U2		
	/I2+U2		

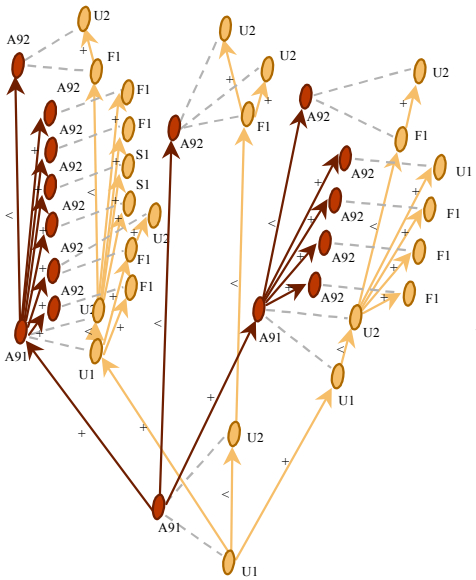


Figure 3-2 Encoding plant architecture as a MTG. **(a)** A part of the plant is considered. **(b)** Plant components are identified within the crown. **(c)** The organisation of these components and their attributes are encoded in a code file. **(d)** A MTG representing the branching system can be built by AMAPmod from this code file. The plant representation at annual shoot scale is in red and at growth unit scale in yellow.

3.2.2 Coding Individuals

Different strategies have been proposed for recording topological structures of real plants, *e.g.* [44; 32] for plant represented at a single scale and [21; 25], for multiscale representations. In AMAPmod, plant topological structures are abstracted as multiscale tree graphs. Describing a plant topology thus consists of describing the multiscale tree graph corresponding to this plant. The description of a given plant can be specified using a "coding language". This language consists of a naming strategy for the vertices and the edges of multiscale graphs. A graph description consists of enumerating the vertices consecutively using their names. The name of a vertex is constructed in such a way that it clearly defines the topological location of a given vertex in the overall multiscale graph. The vertices and their features are described using this formal language in a so called "code file". Let us illustrate the general principle of this coding language by the topological structure of the plant depicted in **Figure 3-3**.

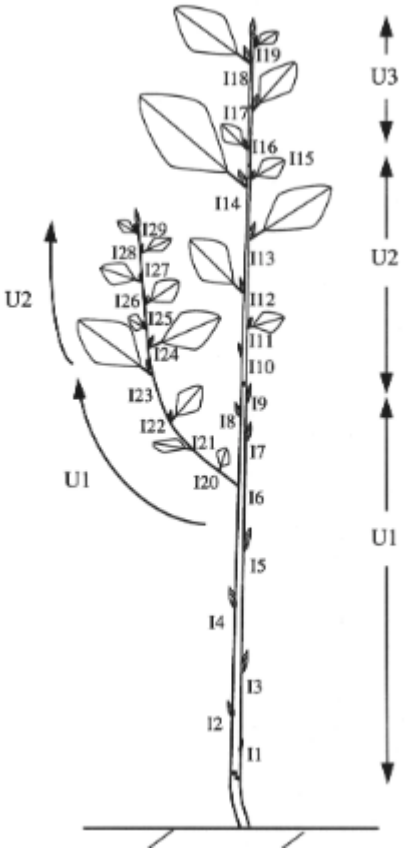


Figure 3-3 Coding the topological structure of a two year old poplar tree.

Each vertex is associated with a *label* consisting of a letter, called its *class*, and an integer, called its *index*. The class of a vertex often refers to the nature of the corresponding botanical entity, *e.g.* `I` for internode, `U` for growth unit, `B` for branching system, etc. The index of a vertex is an integer which enables the user to locally identify a vertex among its immediate neighbors. Apart from this purely structural role, indexes may be used to convey additional meaning : they can be used for instance to encode the year of growth of an entity, its rank in an axis, etc.

At a given scale, plants are inspected by working upwards from the base of the trunk and symbols representing each vertex and its relationship to its father are either written down or keyed directly into a laptop computer.

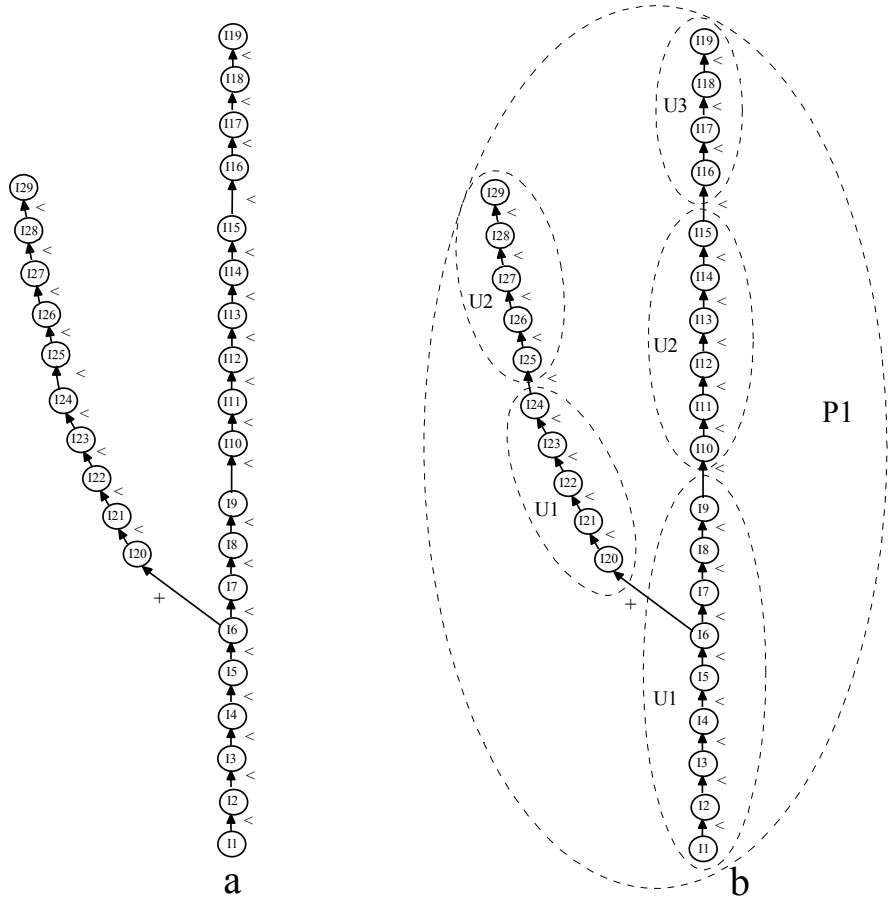


Figure 3-4 (a) tree graph at internode scale **(b)** multiscale tree graph (MTG)

The coded string starts with the single symbol '/'. Coding a single axis (*e.g.* the series of internodes of the trunk depicted in **Figure 3-4a**) would then yield the string :

```
/I1<I2<I3<I4<I5<I6<I7<I8<I9<I10<I11<I12<I13<I14<I15<I16<I17<I18<I19
```

For a branching structure (**Figure 3-4a**), encoding a tree-like structure in a linear sequence of symbols leads us to introduce a special notation, frequently used in computer science to encode tree-like structures as strings (*e.g.* [39]). A square bracket is opened each time a bifurcation point is encountered during the visit (*i.e.* for vertices having more than one son). A square bracket is closed each time a terminal vertex has just been visited (*i.e.* a vertex with no son) and before backtracking to the last bifurcation point. In the above example, entity `I6` is a bifurcation point since the description process can either continue by visiting entity `I7` or `I20`. In this case, the bifurcation point `I6` is first stored in a bifurcation point stack (which is initially empty). Secondly, an opened square bracket is inserted in the output string and thirdly, the visiting process resumes at one of the two possible continuations, for example `I20`, leading to the following code :

```
/I1<I2<I3<I4<I5<I6[+I20
```

The entire branch `I20` to `I28` is then encoded like entities `I1` to `I6`. Entity `I29` has no son, and thus is a terminal entity. This results in inserting a closed square bracket in the string :

```
/I1<I2<I3<I4<I5<I6[+I20<I21<I22<I23<I124<I25<I26<I27<I28<I29]
```

The last bifurcation point can then be popped out of the bifurcation point stack and the visiting process can resume on the next possible continuation of `I6`, *i.e.* `I7`, leading eventually to the final output code string :

```
/I1<I2<I3<I4<I5<I6[+I20<I21<I22<I23<I124<I25<I26<I27<I28<I29]
                   <I7<I8<I9<I10<I11<I12<I13<I14<I15<I16<I17<I18<I19]
```

Let us now extend this coding strategy to multiscale structures. Consider a plant described at three different scales, for example the scale of internodes, the scale of growth units and the scale of plants (**Figure 3-4b**). The depth first procedure explained above is generalized to multiscale structures in the following way. The multiscale coding strategy consists basically of describing the plant structure at the highest scale in a depth first order. However, during this process, each time a boundary of a macroscopic entity is crossed when passing from entity *a* to entity *b*, the corresponding macroentity label, suffixed by a '`/`', must be inserted into the code string just before the label of *b* and after the edge type of (*a*,*b*). If more than one macroscopic boundary is crossed at a time, corresponding labels suffixed by '`/`' must be inserted into the code string at the same location, labels of the most macroscopic entities first. In the multiscale graph of **Figure 3-4b** for example, the depth first visit is carried out at the internode level (highest scale). The visit starts by entering in vertex `I1` at the scale of internodes. However, to reach this entity from the outside, we cross boundaries of `P1` and `U1`, in this order. Then the depth first visit starts by creating the code string :

```
/P1/U1/I1
```

Then, the coding procedes through vertices `I1` to `I6`, with no new macroscopic boundary encountered. `I6` is a bifurcation point and as explained above, this vertex is stored in the bifurcation point stack, a '`[`' is inserted in the code string and the depth first process continues on the son of `I6` whose label is `I20`. Since to reach `I20` from `I6` the macroscopic boundary of the first growth unit of the branch is crossed, on `I20` the generated code string is

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20
```

Similarly on the new branch, coding continues and crosses a growth unit boundary between internodes `I24` and `I25` :

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20<I21<I22<I23<I24<U2/I25<I26
                                                 <I27<I28<I29]
```

Once the end of the branch is reached at entity `I29`, a '`]`' is inserted in the code string and the process backtracks to bifurcation point `I6` in order to resume the visit at the internode scale on the next son of `I6`, *i.e.* `I7`. Then coding goes through to the end of the poplar trunk since there are no more bifurcation points. Between entities `I7` and `I19`, two growth unit boundaries are crossed which generate the final code string :

```
/P1/U1/I1<I2<I3<I4<I5<I6[+U1/I20<I21<I22<I23<I24<U2/I25<I26<I27<I28
       <I29]<I7<I8<I9<U2/I10<I11<I12<I13<I14<I15<U3/I16<I17<I18<I19]
```

It is often the case in practical applications that a number of attributes are measured on certain plant entities. Measured values can be attached to corresponding entities using a bracket notation, '{...}'. For instance, assume that one wants to note the length and the diameter of observed growth units. For each measured growth unit, a pair of ordered values defines respectively its measured length and diameter. Then, the precedent code string would become :

```
/P1/U1{10,5.9}/I1<I2<I3<I4<I5<I6[+U1{7,3.5}/I20<I21<I22<I23<I24
          <U2{4,2.1}/I25<I26<I27<I28<I29]<I7<I8<I9<U2{8,4.3}/I10<I11
                        <I12<I13<I14<I15<U3{7.5,3.9}/I16<I17<I18<I19
```

In this string, we can read that the first growth unit of the trunk, U1, has length 10 cm and diameter 5.9 mm (units are assumed to be known and fixed).

In practical applications, coding plants as raw sequences of symbols becomes quite unreadable. In order to give the user a better feedback of the plant topology in the code itself, we can slightly change the above code format in order to achieve better legibility. Each square bracket is replaced by a new line and an indentation level corresponding to the nested degree of this square bracket. Similarly, a new line is created after each feature set and the feature values are written in specific columns. The following table gives the final code corresponding to the example in **Figure 3-3**.

	Length	Diameter
/P1/U1	10	5.9
/I1<I2<I3<I4<I5<I6		
+U1	7	3.5
/I20<I21<I22<I23<I24<U2	4	2.1
/I25<I26<I27<I28<I29		
<I7<I8<I9<U2	8	4.3
/I10<I11<I12<I13<I14<I15<U3	7.5	3.9
/I16<I17<I18<I19		

3.2.3 Exploration: a simple example

Once a plant database has been created, it can be analyzed using the AMAPmod software. The different objects, methods and models contained in AMAPmod can be accessed through a functional language called AML. This language has been designed to optimize access to plant databases.

- **Creating plant representations**

The formal representation of a plant, and more generally of a set of plants, can be built by AMAPmod from its code file using the AML function MTG() :

```
AML> g = MTG("tree_code_file.txt")
```

The procedure `MTG` attempts to build the plant formal representation, checking for syntactic and semantic correctness of the code file. If the file is not consistent, the procedure outputs a set of errors which have to be corrected before applying a new syntactic analysis. Once the file is syntactically consistent, the MTG is built (*cf.* **Figure 3-4b**) and is available in the variable `g`. However, for efficiency reasons, the latest constructed MTG is said to be "active" : it will be considered as an implicit argument of most of the functions dealing with MTGs. To get the list of all vertices contained in `g`, for instance, we write :

```
AML> vlist = VtxList()
```

instead of

```
AML> vlist = VtxList(g)
```

The function `VtxList()` extracts the set of vertices from the active MTG `g` and returns the result in variable `vlist`.

Once the MTG is loaded, it is frequently useful to make sure that the database actually corresponds to the observed data. Part of this checking process has already been done by the `MTG()` function. But, some high-level checking may still be necessary to ensure that the database is completely consistent. For instance, in our example, we might want to check the number of plants in the database. Since plants are represented by vertices at scale 1, the set of plants is built by :

```
AML> plants = VtxList(Scale -> 1)
```

Like `vlist`, the set `plants` is a set of vertices. The number of plants can be obtained by computing the size of the set `plants`.

```
AML> plant_nb = Size(plants)
```

Each plant constituting the database can be individually and interactively accessed via AML. For instance, assuming the plant corresponding to the example of **Figure 3-4b** is represented by a vertex (at scale 1) with label P1. Plant P1 can be identified in the database by selecting the vertex at scale 1 having index 1 :

```
AML> plant1 = Foreach _p In plants : Select(_p, Index(_p)==1)
```

In this expression, the `Foreach` construct is used to browse the set of plant vertices `plants`. For each plant vertex `_p` in this set, operator `Select` is applied and returns a non void value only for vertices whose index value is 1. `Plant1` thus contains the vertex representing plant P1. Now it is possible to apply new functions to this vertex in order to explore the nature of plant P1. Assume for instance we want to know the number of growth units composing P1 :

```
AML> gu_nb = Size(Components(plant1))
```

`Components()` is a built-in function which applies to a vertex v and returns the vertices composing v at the next superior scale. Since `plant1` is a vertex at scale 1, representing plant P1, components of `plant1` are vertices at scale 2, *i.e.* growth units. It is also possible to

compute the number of internodes composing a plant by simply specifying the optional argument `Scale` in function `Components` :

```
AML> internode_nb = Size(Components(plant1, Scale -> 3))
```

Many such direct queries can be made on the plant database which provide interactive access to it. However, a complementary synthesizing view of the database may be obtained by a graphical reconstruction of plant geometry. Geometrical parameters, like branching and phyllotactic angles, diameters, length, shapes, are read from the database. If they are not available, mean values can be inferred from from samples or can be inferred from additional data describing plant general geometry [19]. A 3D interpretation of the MTG provides the user with natural feedback on the database. Built-in function `PlantFrame()` computes the 3D-geometry of plants. For example,

```
AML> frame1 = PlantFrame(plant1)
```

computes a 3D-geometrical interpretation of P1 topology at scale 2, *i.e.* in terms of growth units (**Figure 3-5a**). Like in the previous example, `PlantFrame` takes `Scale` as an optional argument which enables us to build the 3D-geometrical interpretation of P1 at the level of internodes (**Figure 3-5b**) :

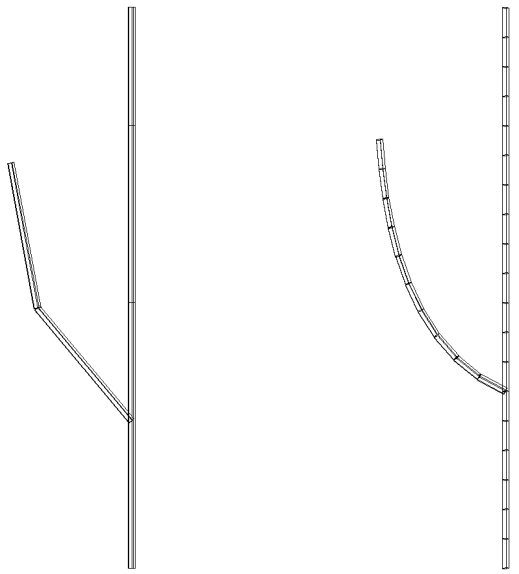


Figure 3-5 3D geometrical reconstruction of the MTG. Reconstruction **(a)** at growth unit scale. **(b)** at internode scale.

Refinements of this 3D geometrical reconstruction may be obtained with the possibility to change the shape of the different plant components, possibly at different scales, to tune geometrical features (length, diameter, insertion angle, phyllotaxy, ...) as functions of the topological position of entities in the plant structure.

- **Extraction of plant entity features**

When attributes of entities are available in MTGs, it is possible to retrieve their values by using the function `Feature()` :

```
AML> first_gu = Trunk(plant1)@1
AML> first_gu_diameter = Feature(first_gu, "Diameter")
```

The first line retrieves the vertex corresponding to the first growth unit of the trunk of P1 (function `Trunk()` returns the ordered set of components of vertex P1, and operator `@` with argument 1 selects the first element of this set). Then, in the second line, the diameter of this growth unit is extracted from the database. Variable `first_gu_diameter` then contains the value 5.9 (see the code file). Similarly the length of the first growth unit can be retrieved :

```
AML> first_gu_length = Feature(first_gu, "Length")
```

Variable `first_gu_length` contains value 10.

The user can simplify this extraction by creating alias names :

```
AML> diameter(_x) = Feature(_x, "Diameter")
AML> length(_x) = Feature(_x, "Length")
```

It is then possible with these functions to build data arrays corresponding to feature values associated with growth units.

```
AML> growth_unit_set = VtxList(Scale -> 2)
AML> Foreach _x In growth_unit_set : length(_x)
```

Moreover, new synthesized attributes can be defined by creating new functions using these basic features. For example, making the simple assumption that the general form of a growth unit is a cylinder, we can compute the volume of a growth unit :

```
AML> volume(_x) = (PI*diameter(_x)^2 / 4)*length(_x)
```

where `PI` denotes the real constant π and `^` denotes the power function. Now, the user can use this new function on any growth unit entity as if it were a feature recorded in the MTG. For instance, the volume of the first growth unit is computed by :

```
AML> first_gu_length = volume(first_gu)
```

The total volume of the trunk :

```
AML> trunk_volume = Sum( Foreach _gu In Trunk(plant1) :
     volume(_gu))
```

The wood volume of the whole plant can be computed by :

```
AML> plant_volume = Sum( Foreach _gu In Components(plant1) :
     volume(_gu))
```

- **Extracting more information from plant databases**

As illustrated in the previous section, plant databases can be investigated by building appropriate AML queries. Built-in words of the AML language may be combined in various ways in order to create new queries. In this way, more and more elaborated types of queries can be constructed by creating user-defined functions which are equivalent to computing

programs. In order to illustrate this procedure, let us assume that we would like to study distributions of numbers of internodes per growth units, such distributions being an important basic prerequisite for botanically-based 3D plant simulations (*e.g.* [2; 9; 37; 3]). At a first stage, we consider all the growth units contained in the plant database together. We first need to define a function which returns the number of internodes of a given growth unit. Since in the database, each growth unit (at scale 2) is composed of internodes (at scale 3) we compute the set of internodes constituting a given growth unit `_x` as follows :

```
AML> internode_set(_x) = Components(_x)
```

The object returned by function `internode_set()` is a set of vertices. The number of internodes of a given growth unit is thus the size of this set :

```
AML> internode_nb(_x) = Size(internode_set(_x))
```

Second, the entities on which the previous function has to be applied, must be located in the database. A set of vertices is created by selecting plant entities having a certain property.

The set of growth units is the set of entities at scale 2 :

```
AML> gu_set = VtxList(Scale -> 2)
```

Third, we have to apply function `internode_nb()` to each element of the selected set of entities :

```
AML> sample1 = Foreach _x In gu_set : internode_nb(_x)
```

We use iterator `Foreach` in order to browse the whole set of growth units of the database, and to apply our `internode_nb()` function to each of them.

Now, we want to get the distribution of the number of internodes on a more restricted set of growth units. More precisely, we would like to study the distribution of internode numbers of different populations corresponding to particular locations in the plant structure. We thus have to define these populations first and then to iterate the function `internode_nb()` on each entity of this new population like in the previous example. Let us consider for example the population made of the growth units composing branches of order 1. Consider again the whole set of growth units `gu_set`. Among them, those which are located on branches (defined as entities of order 1 in AML) are defined by :

```
AML> gu1 = Foreach _x In VtxList(Scale->2) :
     Select(_x, Order(_x) == 1)
```

Here again, we use the iterator `Foreach` in order to browse the whole set of growth units of the database, and to apply the `Select` operator to each of them. `Select` will return only growth unit vertices whose order is 1. AML variable `gu1` thus contains all the growth units in the corpus which are located on branches. Eventually, after the sample of values is built, the above function is applied to the selected entities :

```
AML> sample = Foreach _x In gu1 : internode_number(_x)
```

At this stage, a set of values has been extracted from the plant database corresponding to a topologically selected set of entities. This sample of data can be further investigated with

appropriate AML tools. For example, AML provides the built-in function `Histogram()` which builds the histogram corresponding to a set of values.

```
AML> histo1 = Histogram(sample)
AML> Plot(histo1)
```

This plot gives the graph depicted in **Figure 3-6a**. Similarly, by selecting samples corresponding to different topological situations, we would obtain the series of plots in **Figure 3-6** [4].

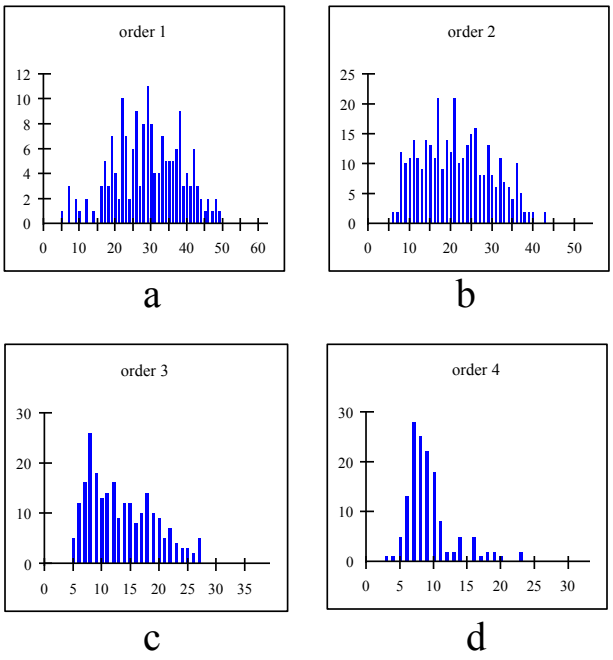


Figure 3-6 Different distributions of the number of internodes per growth unit, in different topological situations.

3.2.4 The AMAPmod querying language: AML

AML is a functional language that provides users with an interactive interface to plant architecture databases. Databases, extracted data, statistical data samples and models are all represented in AML by dedicated data types that can be built and managed using specific functions. These functions are designed in order to provide users with high-level access to these generally complex objects. Let us first sketch the basics of the AML language.

From a practical viewpoint, AML is an interactive command interpreter that processes user's commands one after the other. AML's prompt `AML>` indicates that the system is waiting for user's input. After each input, the system evaluates the user's command and returns a message displaying the type of the object computed from the evaluation and its contents. For example, the command to read an array of integers from an ASCII file provides the following interaction:

```
AML> ARRAY("my_file_of_integers")
     <ARRAY(INT)> [1,9,3,7,12,19]
```

In AML, all operations are expressed as function calls. AML contains a built-in set of functions, called primitives that can be split into several groups.

A first group provides a kernel of standard functions comprised of doing arithmetics, reading/writing data, storing variables, working on built-in data types, building new data types, displaying graphics, etc.

A second group of functions consists of primitives that enable the user to access the plant architecture database. This contains primitives for loading, exploring, extracting information, vizualising and comparing MTGs.

A third group consists of statistical tools that can be used to explore and to analyse data extracted from MTGs. Primitives to estimate model parameters and to check for the validity of these models, to generate data samples from simulation are provided for each model.

Based on AML primitives, the user can build his own functions, which correspond to AML programs. A user-define function is an expression containing one or several free variables that is given a name. For instance, consider the definition of a cone frustum with base diameter *b*, top diameter *t* and height *h*. The volume of a cone frustum can be defined in AML as a function, named `cfvol`, taking three arguments:

```
AML> cfvol(_b,_t,_h) = Pi * _h * (_b^2 + _t^2 + _b*_t ) / 12
```

The underscore sign is used to make a distinction between free variables - used to define user functions - and normal computer variables, used to store values and computed objects. Free variables are not used to store any value, they only appear in expressions to identify a given term in different positions of an expression. Computer variable have a name not preceded by an underscore sign and appear in the left-hand side of an assignment expression:

```
AML> volume1 = cfvol(1,1,4)   # volume1 is a computer variable
     <REAL> 3.14159           # value computed by the call to
                              cfvol and stored in volume1
```

Several types of objects are defined in AML. They are first basic types like `INT`, `REAL`, `CHAR`, `STRING`, `DATE`, `BOOL`, etc. These types can be combined to create new types using type constructors such as `ARRAY`, `SET` or `LIST`:

```
AML> array = [1,9,3]   # creates an array containing 3 INTs
     <ARRAY(INT)> [1,9,3]
```

`ARRAY`s are ordered sets of elements with identical types; `SET`s are sets of elements of identical types with no notion of order and containing no duplication of elements; `LIST`s are ordered sets of elements with possibly different types. Since `ARRAY` and `LIST` objects are ordered sets, their *i*-th element can be defined:

```
AML> array@2   # extracts the second element from array
     <INT> 9
```

All the preceding types correspond to objects that can be built by primitives that belong to the AML kernel. However, other AML primitives allow the user to build more specific objects. These specific objects are usually built by primitive constructors that have the name of the object type: for instance the primitive `MTG` can be applied to a file containing the code of a

plant architecture to check its syntactic and semantic correctness and to build a MTG object containing the plant architecture information:

```
AML>  my_plant = MTG("codefile")
      <MTG> vtxnb = 1546  size = 10 kb
```

Similarly, an object of type HISTOGRAM can be built from an array of integers using the primitive Histogram:

```
AML>  h1 = Histogram([1,2,2,3,3,2,4,2,3,2])
      <HISTOGRAM> sample size: 10  mean: 2.4
      standard-deviation: 0.843
```

or from a file containing the frequencies for each possible value.

Once specific objects such as MTG or HISTOGRAM are built, they can be displayed, plotted, transformed or compared using special primitives that apply to these objects. A sub-sample for instance can be extracted from the object h1 using the primitive ValueSelect by specifying the interval of values that must be kept:

```
AML>  h2 = ValueSelect(h1,2,3)    # selects a sub-sample of h1
                                  # [2,2,3,3,2,2,3,2]
      <HISTOGRAM> sample size: 8  mean: 2.375
      standard-deviation: 0.518
```

Primitive functions have mandatory arguments and optional arguments. Mandatory arguments are identified according to their position in the argument list while optional arguments are given names and are not mandatory in the primitive function argument list. For example, in the primitive MTG primitive, to bound the number of errors output by the parser if any, the user may specify the optional variable MaxErrorNb as follows:

```
AML>  my_plant = MTG("codefile", MaxErrorNb -> 10)
```

In AML, loops can be carried out using iterators. The most common iterator enables the user to browse the elements of a set (either an ARRAY, a SET or a LIST) and to apply to each of them a particular function. For instance, the set of square values corresponding to an array of integers can be computed as follows:

```
AML>  square_vals = Foreach _x In [1,2,3,4] : _x^2
      <ARRAY(INT)> [1,4,9,16]
```

In section 3.3, we will illustrate several aspects of constitution and analysis of plant architecture databases by providing the AML queries that correspond to each step of a modelling session.

3.2.5 Types of extracted data

As mentioned above, various types of data can be extracted from MTGs. For each plant component in the database, attributes can be extracted or synthesised using the AML language. The wood volume of a component, for instance, can be synthesised from the diameter and the length of this component measured in the field. The type of measurement carried out in the context of architectural analysis emphasises the use of discrete variables

which can be either symbolic, *e.g.* the type of axillary production at a given node (latent bud, short shoot or long shoot) or numeric (number of flowers in a branching structure). In general, a plant component can be qualified by a set of attributes, called a multivariate attribute. A plant component, for instance, could be described by a multivariate attribute made up of the volume, the number of leaves, the azimuth and the botanical type of the constituent.

Multivariate attributes correspond to the first category of data that can be extracted from MTGs. A second and more complex category of particular importance in AMAPmod is defined by sequences of - possibly multivariate - attributes. The aim of this category is to represent biological sequences that can be observed in the plant architecture. These sequences may have two origins: they can correspond to changes over time in the attributes attached to a given plant component. In this case, the sequences represent the trajectories of the components with respect to the considered attributes and the index parameter of the sequences is the observation date. Sequences can also correspond to paths in the tree topological structures contained in MTGs. In this case, the index parameter of the sequences is a spatial index that denotes the rank of the successive components in the considered paths. Spatially-indexed sequence is a versatile data type for which the attributes of a component in the path can be either directly extracted or synthesised from the attributes of the borne components. In the later case, all the information contained in the branching system can be efficiently summarised into a sequence of multivariate attributes, corresponding to the main axis of the branching system.

A third category of object can be extracted from MTGs, namely trees of – multivariate - attributes. Like sequences, these objects are intended to preserve part of the plant organisation in the extracted data. Tree structures represent the raw organisation of the components that compose branching structures of the plant at a certain scale of analysis.

Data extracted from MTGs can thus be ordered according to their level of structural complexity: unstructured data, sequences, trees. These levels correspond to different degrees to which the structural information contained in the MTG is summarised and are associated with different statistical analysis techniques.

3.2.6 Statistical exploration and model building

To explore plant architecture, users are frequently led to create data samples according to topological criteria on plant architecture. A wide range of AML primitives that apply to MTGs enable the user to express these topological criteria and select corresponding plant components. Samples of the three main structural data types can be created as described below:

Multivariate samples: Simple data samples can be created by computing the set of - possibly multivariate - attributes associated with a selected set of components, *e.g.* the number of flowers borne by components that appeared in the plant structure during 1995. Since a very large panoply of methods are available in statistical packages for analysing multivariate samples, only a reduced core of tools have been integrated in AMAPmod for exploring these objects. If more specific statistical methods are required, the user can export data to other softwares such as SAS or S-PLUS.

Samples of multivariate sequences: The focus in AMAPmod is on data analysis tools for samples of sequences. In the context of plant architecture analysis, these objects present two advantages. On the one hand, part of the plant organisation is directly preserved in the sample

through the notion of "sequence" discussed above. On the other hand, the structural complexity of samples of sequences still remains tractable and efficient exploratory tools and statistical models can be designed for them [28; 29]. The AMAPmod system includes mainly classes of stochastic processes such as (hidden) Markov chains, (hidden) semi-Markov chains and renewal processes for the analysis of discrete-valued sequences. A set of exploratory tools dedicated to sequences built from numeric variables is also available, including sample (partial) autocorrelation functions and different types of linear filters (for instance symmetric smoothing filters to extract trends or residuals).

Samples of multivariate trees: The analysis of samples of tree structured data is a challenging problem. A sample of trees could represent a set of comparable branching systems considered at different locations in a plant or in several plants. Similarly, the development of a plant can be represented by a set of trees, representing different steps in time of a branching system. Plant organisation for this type of object is relatively well preserved in the raw data. However, this requires a higher degree of conceptual and algorithmic complexity. We are currently investigating methods for computing distances between trees [13] which could be used as a basis for dedicated statistical tools.

AMAPmod contains a large set of tools for analysing these different types of samples, with special emphasis on tools dedicated to the analysis of samples of discrete-valued sequences. These tools fall into one of the three following categories:

- exploratory analysis relying on descriptive methods (graphical display, computation of characteristics such as sample autocorrelation functions, etc.),
- parametric model building,
- comparison techniques (between individual data).

The aim of building a model is to obtain adequate but parsimonious representation of samples of data. A parametric model may then serve as a basis for the interpretation of a biological phenomenon. The elementary loop in the iterative process of model building is usually broken down into three stages:

1. The *specification* stage consists of determining a family of candidate models on the basis of the results given by an exploratory analysis of the data and some biological knowledge.

2. The *estimation* stage, consists of inferring the model parameters on the basis of the data sample. This model is chosen from within the family determined at the specification stage. Automatic methods of model selection are available for classes of models such as (hidden) Markov chains dedicated to the analysis of stationary discrete-valued sequences. In AMAPmod, the estimation is always made by algorithms based on the maximum likelihood criterion. Most of these algorithms are iterative optimisation schemes which can be considered as applications of the Expectation-Maximisation (EM) algorithm to different families of models, [12; 26; 27]. The EM algorithm is a general-purpose algorithm for maximum likelihood estimation in a wide variety of situations best described as incomplete data problems.

3. The *validation* stage, consists of checking the fit between the estimated model and the data to reveal inadequacies and thus modify the a priori specified family of models. In the AMAPmod system, theoretical characteristics can be computed from the estimated model parameters to fit the empirical characteristics extracted from the data and used in the exploratory analysis.

The parametric approach based on the process of model building is complemented by a nonparametric approach based on structured data alignment (either sequences or trees). Distance matrices built from the piece by piece alignments of a sample of structured data can be explored by clustering methods to reveal groups in the sample.

3.3 Illustration: exploring an apple tree orchard

Let us now illustrate the approach implemented in the AMAPmod system in a real application. To do this, we shall consider an apple tree orchard and show how a plant architecture database can be created from observations [24]. Then, we shall use this database to illustrate the use of specific tools employed to explore plant architecture databases.

3.3.1 Biological context and data collection

The application is part of a general selection program, conducted at INRA (Institut National de la Recherche Agronomique), and aims to improve apple tree species as regards morphological characters and more classical criteria such as fruit quality and disease resistance. In this particular example, two apple tree clones were chosen for their contrasting growth and branching habits. The first clone ('Wijcik') exhibits a very particular growth and branching habit, characterised by short internodes, great diameters and the absence of long axillary branches. By contrast, the second clone ('Baujade') exhibits many long and flexible branches. A population of 102 hybrids was obtained by crossing these two clones. The objective of this work was to study how morphological characters, such as the length of the internodes or the number of long lateral branches, are distributed within the progeny.

Creation of the database: The branching systems borne by the three-year-old annual shoot of the trunk is described for each individual. The branching system is first broken down into axes *i.e.* linear portions of stem derived from the same bud. Each axis is then divided into portions created during the same year (called annual shoots). When cessation and resumption of growth occur within a year, the annual shoot can be split into growth units, *i.e.* portions created over the same period (or between two resting periods). Finally, the growth units can be divided into internodes, *i.e.* portions of stem between two leaves. Regarding these successive decompositions, a given branching system is simultaneously considered at four scales. The different plant components and their connections are represented into a code file as explained previously.

In order to give a quantitative idea of the total resources necessary for an application of this size, it should be noted that all the measures were carried out by a team of 6 persons over 5 days. The collected data, initially recorded on paper, were then computer-entered by 1 person over 20 days using a text editor and consists of a file of approximately 16000 lines of code. The corresponding MTG is constructed in 45 seconds on a SGI-INDY workstation. It contains about 65000 components and some 15000 attributes. The overall size of the database is 7 Mb.

3.3.2 3D visualisation of real plants

To build the database associated with the collected data, the AMAPmod system is launched and an MTG is built from the encoded plant file:

```
AML> plant_database = MTG("appletree_code.txt")
```

The primitive `MTG` attempts to build a formal representation of the orchard, checking for syntactic and semantic correctness of the code file. If the file is not consistent, the procedure outputs a set of errors which must be corrected before applying a new syntactic analysis. Once the file is syntactically consistent, the MTG is built and is available in the variable `plant_database`. However, for efficiency reasons, the latest constructed MTG is said to be "active": it will be considered as an implicit argument for most of the primitives dealing with MTGs. For example, to obtain the set of vertices representing the plants contained in the database, *i.e.* vertices at scale 1, the primitive `VtxList` is used and applies by default to the active MTG `plant_database`:

```
AML> plant_list = VtxList(Scale -> 1)
```

It is then possible to obtain an initial feedback on the collected data by displaying a 3D geometrical interpretation of a plant from the MTG. This notably allows the user to rapidly browse the overall database. For instance, a geometric interpretation of the 5th plant in the set of plants described in the MTG can be computed and plotted using the primitive `PlantFrame` as follows, (**Figure 3-7a**):

```
AML> geom_struct = PlantFrame(plant_list@5)
AML> Plot(geom_struct)
```

Such reconstructions can be carried out even if no geometric information is available in the collected data. In this case, algorithms are used to infer the missing data where possible (otherwise, default information is used) [19]. In other cases, plants are precisely digitised and the algorithms can provide accurate 3D geometric reconstructions [7; 22; 47; 49].

Apart from giving a natural view of the plants contained in the database, these 3D reconstructions play another important role: they can be used as a support to graphically visualise how various sorts of information are distributed in the plant architecture. **Figure 3-7b** for example shows the organisation of plant components according to their branching order (trunk components have order 0, branch components have order 1, etc.). In AML, this would be obtained by the following commands:

```
AML> color_order(_x) = Switch Order(_x) Case 0: MediumGrey
     Case 1: DarkGrey  Case 2: LightGrey  Case 3: Black
     Default: White
AML> Plot(geom_struct, Color -> color_order)
```

This representation emphasises different informations related to the branching order: it can be seen in **Figure 3-7b** that the maximum branching order is 4, that this order is reached only once in the tree crown, and that this occurs at a floral site (black component).

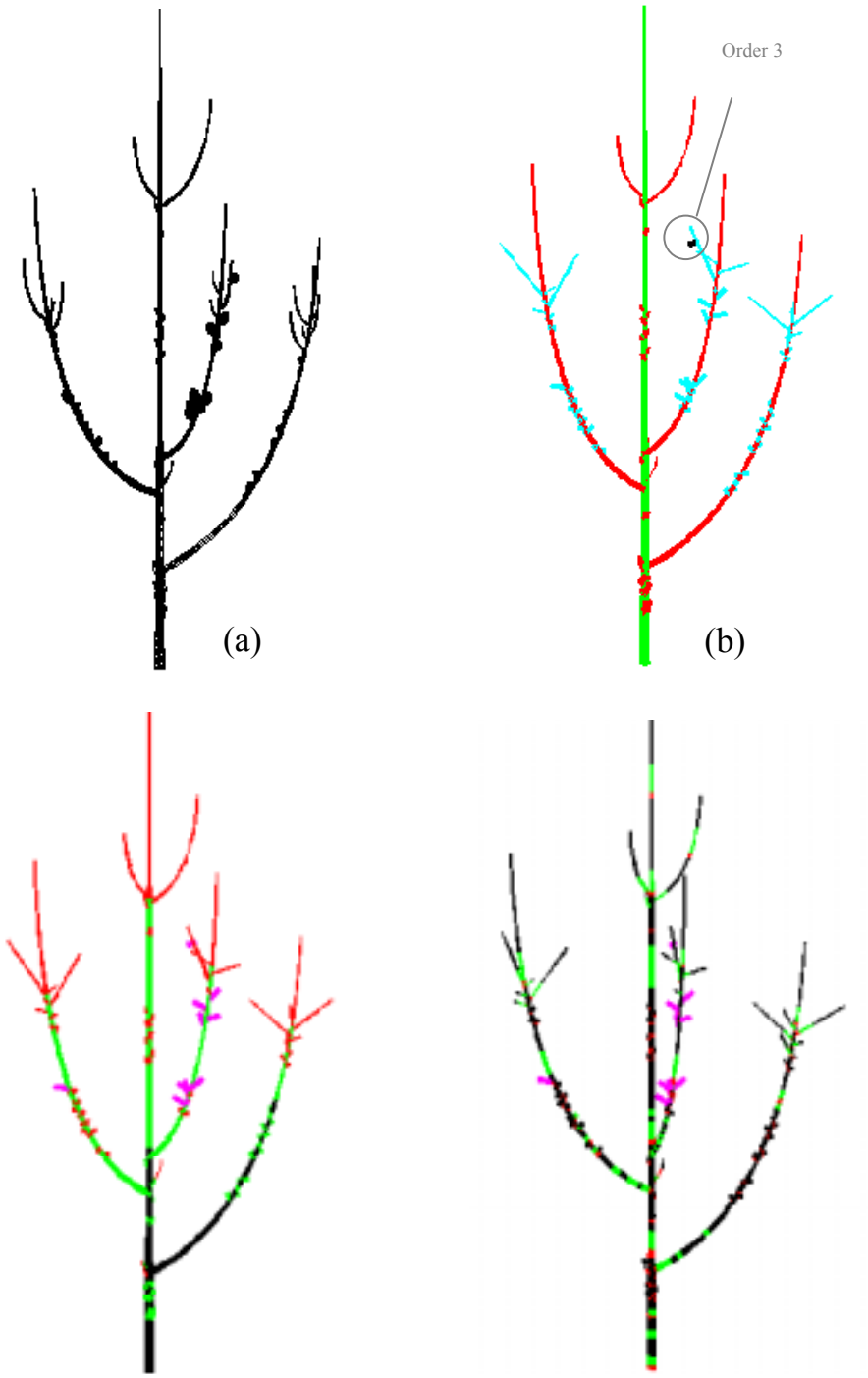


Figure 3-7 (**a**) 3D reconstruction of an apple tree recorded in the database. **(b)** Branching orders (green = 0, red = 1, light blue = 2, black = 3). **(c)** Years of growth (black = 1st year, green = 2nd year, red = 3rd year). **(d)** Growth rhythms (black = long internodes, red = scars, green = short internodes).

The use of the 3D representation of plant structure can also be illustrated in the context of plant growth analysis. The year in which each component grew can be retrieved from a careful analysis of the plant morphological makers. If this information is recorded in the MTG, it is then possible to colour the different components accordingly. **Figure 3-7c** shows, for instance, that a branch appeared on the trunk during the first year of growth. This information can then be linked to other data, *e.g.* the branching order of a component or the

number of fruits borne by a component, and thus provides deeper insight into the plant growth process.

Thanks to the multiscale nature of the plant representation, more or less detailed information can be projected onto the plant structure. Let us consider again the context of plant growth analysis. Plant growth is characterised by rhythms that result in the production of long internodes during periods of high activity and short internodes during rest periods (indicated on the plant by scares close together). These informations, at the level of internodes, can be projected onto the plant 3D structure (**Figure 3-7d**). Like the year of growth, this information enables us to access plant growth dynamics, but now, at an intra-year scale.

Finally, another use for the virtual reconstruction of measured plants is illustrated in **Figure 3-8a** and **3-8b**. These plants have been reconstructed from the MTG at the scale of each leafy internode. This enables us to obtain a natural representation of the plant which can be used for instance in models that are intended to describe the interaction of the plant and its environment (*e.g.* light) at a detailed level, *e.g.* [41]. More generally, the user can plot a set of plants from the database (**Figure 3-9**):

```
AML> orchard = PlantFrame(plant_list)
AML> Plot(orchard)
```

Figure 3-8 Virtual 3D reconstruction of the geometry of a plant with positioning of leaves **(a)** and fruits **(b)**.

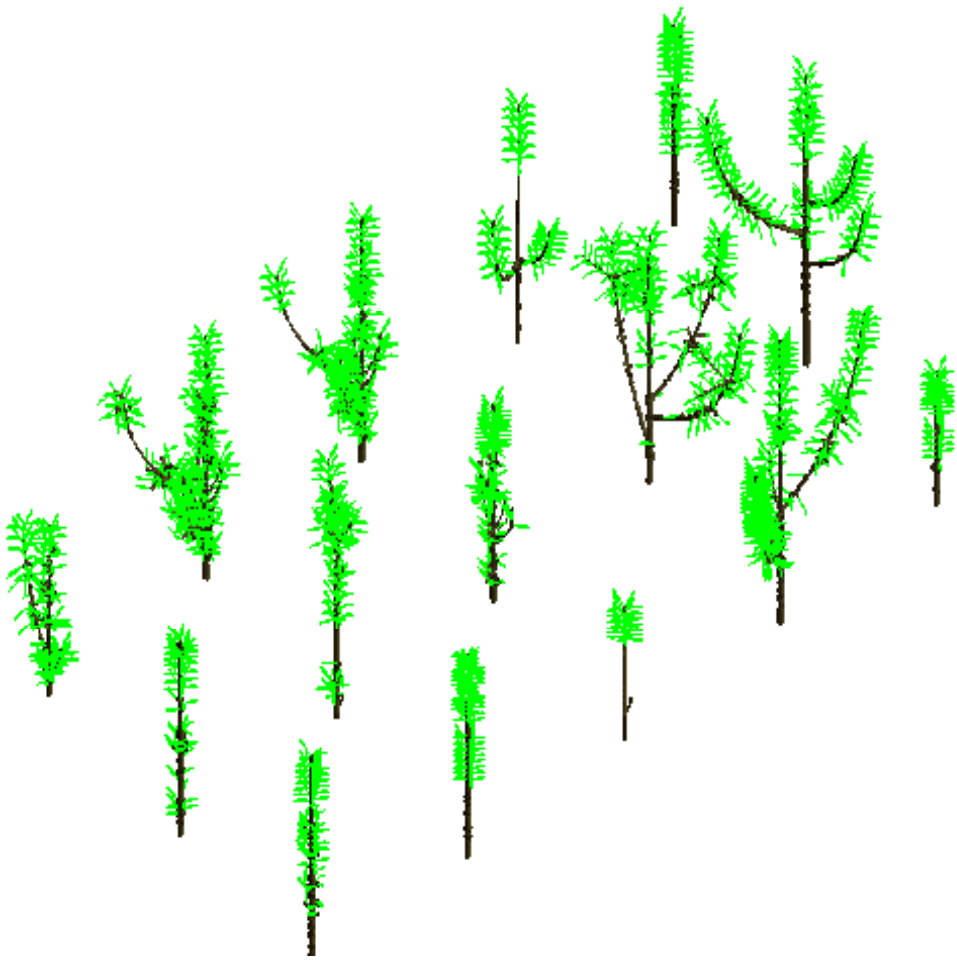


Figure 3-9 3D representation of the information contained in the architectural database (Only 15 plants are represented out of 102).

3.3.3 Extraction of data samples

Visualizing informations projected onto the 3D representation of plants is one way to explore the database. More quantitative explorations can be carried out and the most simple of these consists of studying how specific characters are distributed in the architecture of the plant population. To do this, samples of components are created corresponding to some topological or morphological criteria, and the distributions of one or several characters (target characters) are studied on this sample. This data extraction always follows the three following steps:

- Firstly, a sample of components is created to study the target character.
- Secondly, the character itself is defined. It may be more or less directly derived from the data recorded in the field. For example, it is straightforward to define the diameter of a component if this has been measured in the field. On the other hand, the maximum branching order of the components that are borne by a given component needs some computation.
- Thirdly, the target character is computed for each component of the selected sample of components.

The output of these three operations is a set of values that can be analysed and visualised in various ways. Let us assume for instance that we wish to determine the distribution of the

number of internodes produced during a specific growth period for all the plants in the database. It is first necessary to determine the sample of components on which we wish to study this distribution. In our case, we assume that we are interested in the growth units of the trunk that are produced during the first year of growth. This would be written in AML as:

```
AML> sample = Foreach _component In growth_unit_list :
     Select(_component,Order(_component) == 0 And
     Index(_component) == 90)
```

The variable `sample` thus contains the set of growth units whose order is 0 (*i.e.* which are parts of trunks) and whose growth year is 1990 (assuming 1990 corresponds to the first year of growth). The second step consists of defining the target character. This can be done by defining a corresponding function:

```
AML> nb_of_internodes(_x) = Size(Components(_x))
```

The number of internodes of a component `_x` (assumed to be a growth unit) is defined as the size of the set of components that compose this growth unit `_x` (assuming that growth units are composed of internodes). Finally, this function is applied to each component in the previously selected sample and the corresponding histogram is plotted (**Figure 3-10**):

```
AML> sample_values = Histogram(Foreach _component In sample :
     nb_of_internodes(_component))
AML> Plot(sample_values)
```

This example illustrates the kind of interaction a user may expect from the exploration of tree architecture. In the field, the growth units of the trunks produced during the first year of growth present a variable length, ranging roughly from 10 to 100 internodes. However, the quantitative exploration of the database shows that the histogram exhibits two relatively well-separated sub-populations of components (**Figure 3-10**). The sub-population of short components corresponds to the first annual shoots of the trunk, made up of two successive intra-annual growth units, while the sub-population of long components corresponds to the first annual shoots made up of a single growth unit.

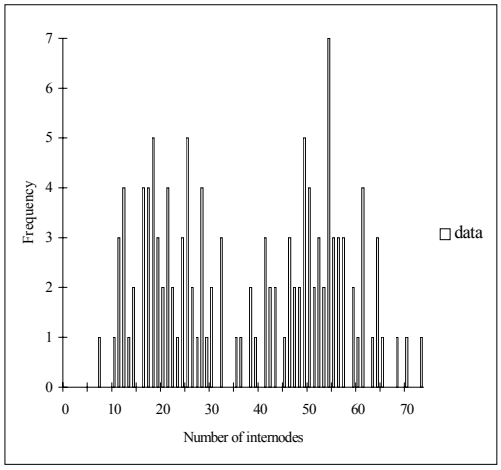


Figure 3-10 Histogram of the number of growth unit internodes for year 90 on the trunk.

In order to separate and characterise these two sub-populations, we can make the assumption that the global distribution is a mixture of two parametric distributions, more precisely, two

negative binomial distributions. The parameters of this model can be estimated from the above histogram as follows:

```
AML> mixture = Estimate(sample_value, "MIXTURE",
     "NEGATIVE_BINOMIAL", "NEGATIVE_BINOMIAL")
AML> Plot(mixture)
```

For all parametric models in the system, the function `Estimate` performs both parameter estimation and computation of various quantities (likelihood of the observed data for the estimated model, theoretical characteristics, etc) involved in the validation stage. As demonstrated by the cumulative distribution functions in **Figure 3-11b**, the data are well fitted by the estimated mixture of two negative binomial distributions. The weights of the two components of the mixture are very close (0.49 / 0.51), the first being centred on 21 internodes and the second on 53 internodes (**Figure 3-11a**). Due to the small overlap of these two mixture components (**Figure 3-11a**), the extracted sample can be optimally split up into two optimal sub-populations with a threshold fixed at 37.

As illustrated in this example, using AMAPmod, the user can query the database, make assumptions and look for data regularities. This interactive exploration process enables the user to build a rich and detailed mental representation of the architectural database, which relies on various complementary viewpoints.

3.3.4 Extraction and analysis of biological sequences

The previous section illustrates the extraction of a simple sample type, made up of numeric values. In this section, we consider a more complex sample type, made up of sequences of values. For example, in the apple tree database, let us consider sequences of lateral productions along trunks. Our aim is to analyse how lateral branches are distributed along the trunks of hybrids.

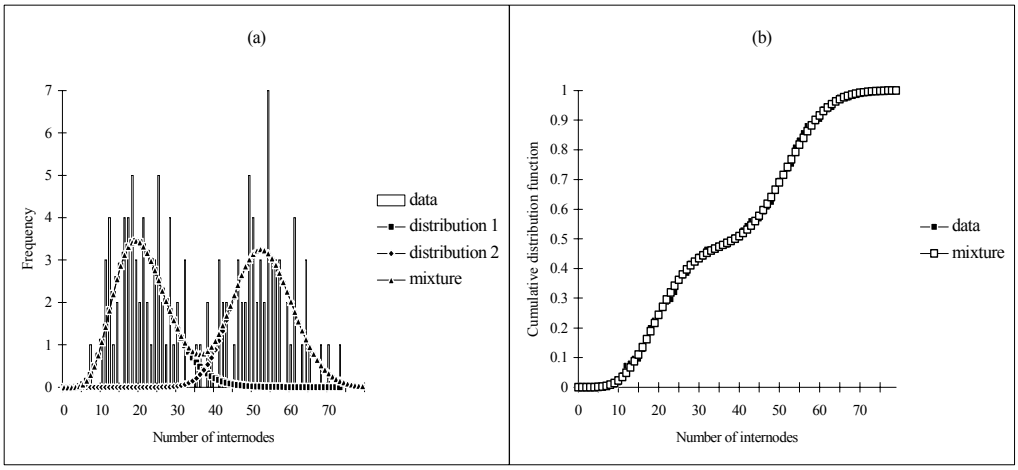


Figure 3-11 Modelling the number of growth unit internodes for year 90 on the trunk by a mixture of two parametric distributions.

The sequences are coded as follows: for each plant, the 90 annual shoot of the trunk is described node by node from the base to the top. Each node is qualified by the type of lateral production (latent bud: 0, one-year-delayed short shoot: 1, one-year-delayed long shoot: 2 and immediate shoot: 3). This sample of sequences is built as follows in AML:

```
AML> seq = Foreach _component In growth_unit_sample :
     Foreach _node In Axis(_component, Scale -> 4) :
     Switch lateral_type(_node)
     Case BUD: 0 Case SHORT: 1 Case LONG: 2
     Case IMMEDIATE: 3 Default: Undef
```

The AML variable `growth_unit_sample` contains the set of growth units of interest (assumed to be selected before). For each component in this set, the array of nodes that compose its main axis is browsed by the second `Foreach` construct. Finally, for each node, a function `lateral_type()` (defined elsewhere) is used to encode the nature of the lateral production at that node.

Figure 3-12 illustrates the diversity of annual shoot branching structures encountered in the studied hybrid family, which results from the different branching habits of the two parents. In our context, we wish to characterise and classify the hybrids according to their branching habits. The difficulty arises from the fact that the branching pattern is made of a succession of branching zones which are not characterised by a single type of lateral production but by a combination of types (*e.g.* short shoots interspersed with latent buds). We shall use this example to illustrate how parametric models may be used in AMAPmod to identify and characterize successive branching zones along these annual shoots.

We assume that sequences have a two-level structure, where annual shoots are made up of a succession of zones, each zone being characterised by a particular combination of lateral production types. To model this two-level structure, we use a hierarchical model with two levels of representation. At the first level, a semi-Markov chain (Markov chain with null self-transitions and explicit state occupancy distributions) represents the succession of zones along the annual shoots and the lengths of each zone [6; 28; 29]. Each zone is represented by a state of the Markov chain and the succession of zones are represented by transitions between states. The second level consists of attaching to each state of the semi-Markov chain a discrete distribution which represents the lateral productions types observed in the corresponding zone. The whole model is called a hidden semi-Markov chain [26; 27].

The model parameters are estimated from the extracted sample of sequences by the function `Estimate`:

```
AML> hsmc = Estimate(seq, “HIDDEN_SEMI-MARKOV”, initial_hsmc,
     Segmentation -> True)
```

The first argument `seq` represents the extracted sequences, `“HIDDEN_SEMI-MARKOV”` specifies the family of models and `initial_hsmc` is an initial hidden semi-Markov chain which summarises the hypotheses made in the specification stage. An optimal segmentation of the sequences is required by the optional argument `Segmentation` set at `True`.

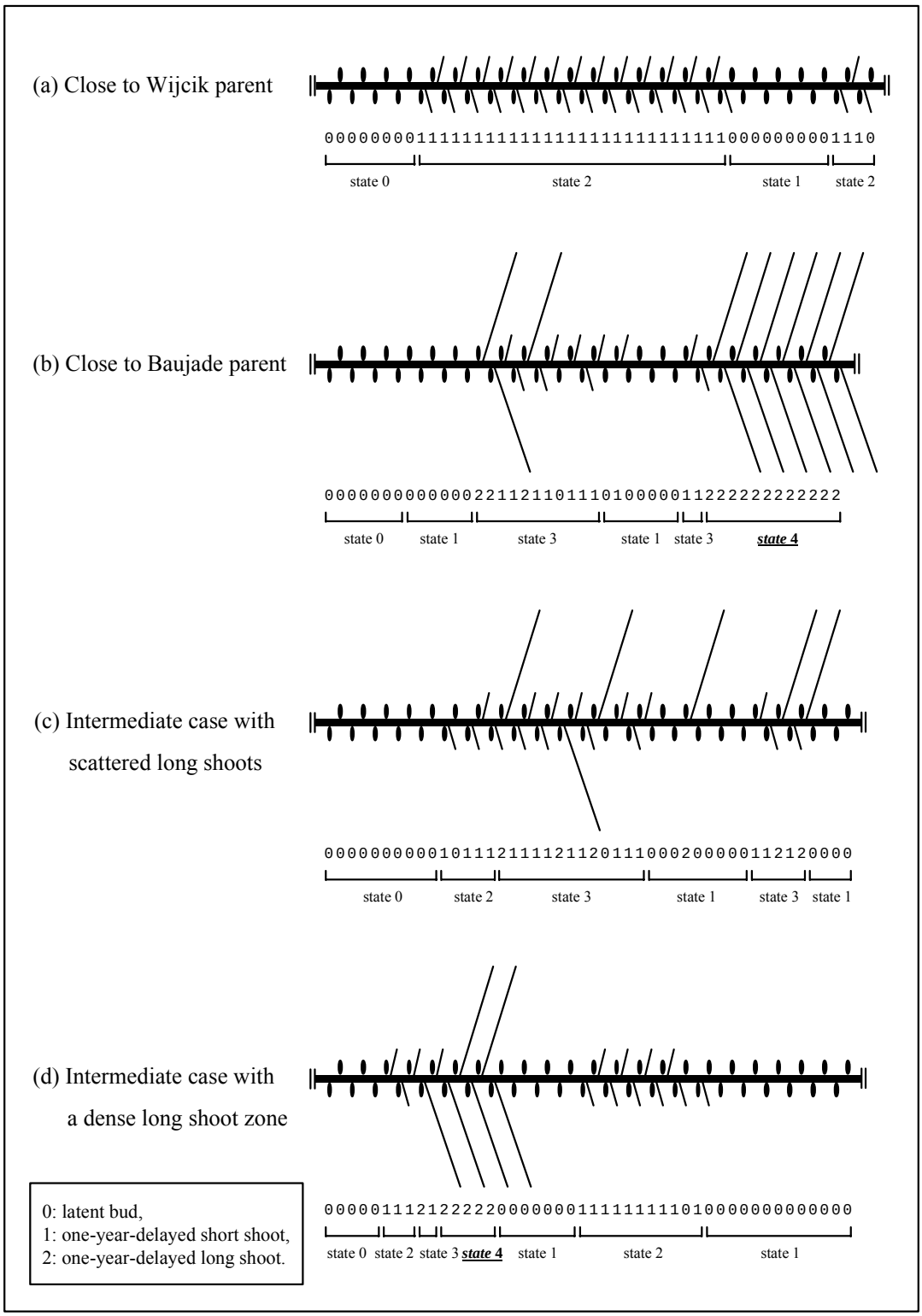


Figure 3-12 Example of sequences showing different branching habits in the hybrid family.

The hidden semi-Markov chain built from the 90 annual shoots of the 102 hybrids is depicted in **Figure 3-13** with the following convention: each state is represented by a box numbered in the lower right corner. The possible transitions between states are represented by directed edges with the attached probabilities noted nearby. Transient states are surrounded by a single line while recurrent states are surrounded by a double line. State *i* is said to be recurrent if starting from state *i*, the first return to state *i* always occurs after a finite number of transitions.

A nonrecurrent state is said to be transient. The state occupancy distributions which represent the length of the zones in terms of number of nodes are shown above the corresponding boxes. The possible lateral productions observed in each zone are indicated inside the boxes, the font sizes being roughly proportional to the observation probabilities(for state 3, these probabilities are 0.1, 0.62 and 0.28 while for state 4, these probabilities are 0.01, 0.07 and 0.92 for latent bud, one-year-delayed short shoot and one-year-delayed long shoot respectively). State 0 which is the only transient state is also the only initial state as indicated by the edge entering in state 0. State 0 represents the basal non-branched zone of the annual shoots. The remaining five states constitute a recurrent class which corresponds to the stationary phase of the sequences.

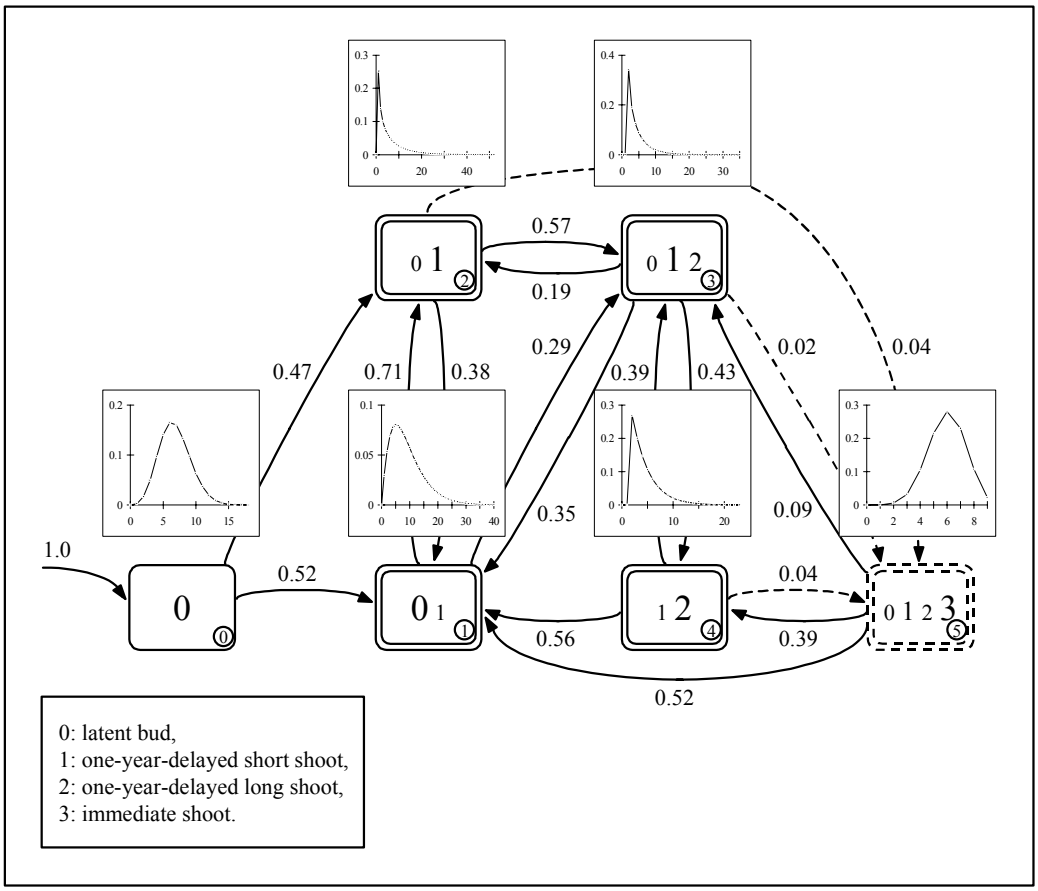


Figure 3-13 Hidden semi-Markov chain built from the 90 annual shoots of the 102 hybrids. Only transitions whose probability ≥ 0.02 are represented. The less probable transitions (respectively states) are represented by dotted edges (respectively dotted boxes).

Building a parametric model gives us a global insight into the structure of the 90 annual shoot of the trunk for the 102 hybrids. The adequacy of the estimated model to the data is checked by examining the fitting of theoretical characteristic distributions computed from the model parameters to the corresponding observed characteristic distributions extracted from the data. Counting characteristic distributions for example focus on the number of occurrences of a given feature per sequence. The two features of interest are the number of series (or clumps) and the number of occurrences of a given lateral production type per sequence. The fits of counting distributions (**Figure 3-14**) can be plotted by the following function:

```
AML> Plot(hsmc, "Counting")
```

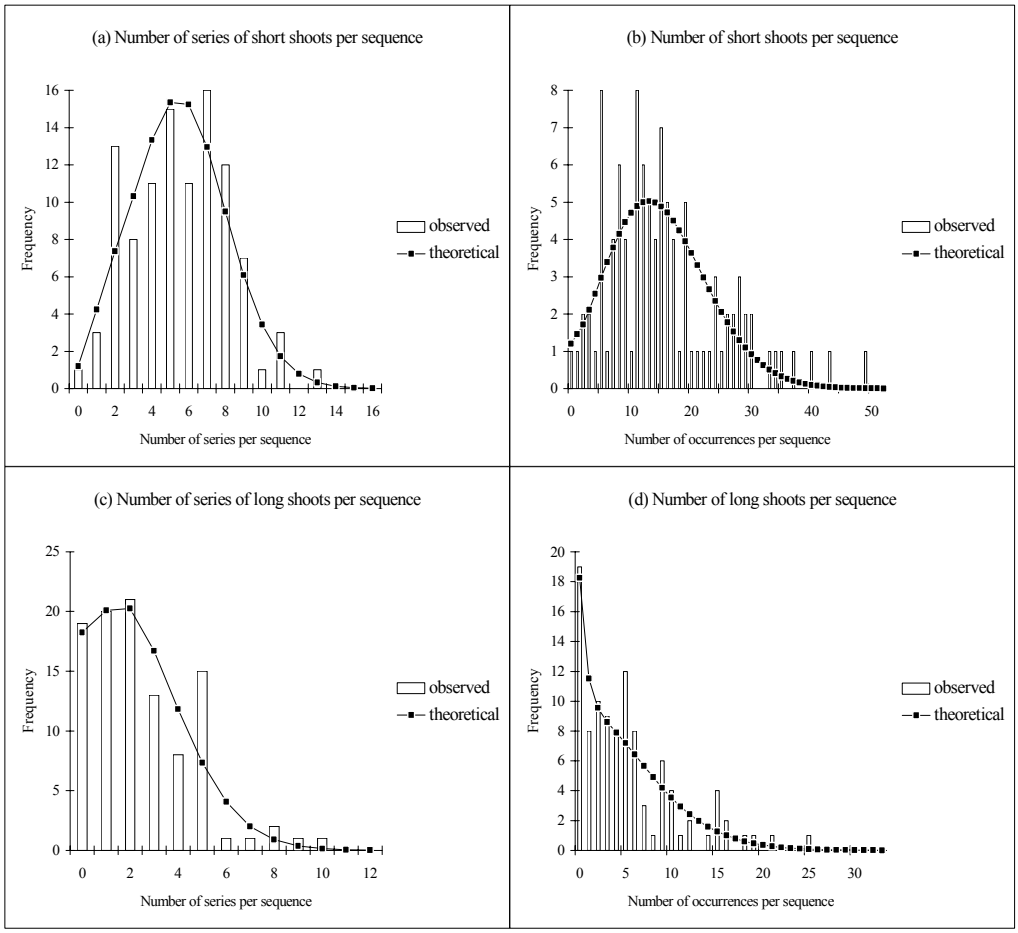


Figure 3-14 Fit of theoretical characteristic distributions computed from the model parameters to the corresponding observed distributions extracted from the data.

In addition, the optimal segmentation of the observed sequences in successive zones (**Figure 3-12**) can be extracted from the model as a by-product of estimation of model parameters by the following function:

```
AML> segmented_seq = ExtractData(hsmc)
```

`segmented_seq` represents the observed sequences augmented by a variable which contains the corresponding optimal state sequences (**Figure 3-12**). A careful examination of this optimal segmentation help us highlight a discriminating property: it suggests using the absence of state 4 in this optimal segmentation as a discrimination rule between hybrids closer to the Wijcik parent than to the Baujade parent (and conversely). State 4 corresponds to a dense long branching zone characteristic of the Baujade parent. Two sub-populations close to each of the parents are extracted by the function `ValueSelect` relying on the absence/presence of state 4 on the 1st variable:

```
AML> wijcik_seq = ValueSelect(segmented_seq, 1, 4,
     Mode -> Reject)
AML> baujade_seq = ValueSelect(segmented_seq, 1, 4,
     Mode -> Keep)
```

Simply counting the number of axillary long shoots per sequence would not have been sufficient, since for a given number of long shoots, these can be either scattered (**Figure 3-12c**) or aggregated in a dense zone (**Figure 3-12d**). This is confirmed by comparing the empirical distributions of the number of series with the number of occurrences of axillary long shoots per sequence extracted from the two hybrid sub-populations. The empirical distributions of the number of series/number of occurrences of axillary long shoots (coded by 2) per sequence for the sub-population close to the Wijcik parent can be simultaneously plotted by the following function (**Figure 3-15a**):

```
AML> Plot(ExtractHistogram(wijcik_seq, "NbSeries", 2, 2),
     ExtractHistogram(wijcik_seq, "NbOccurrences", 2, 2))
```

These empirical distributions are very similar for the sub-population close to the Wijcik parent, (**Figure 3-15a**). Most of the series are thus composed of a single long shoot. These empirical distributions are very different for the sub-population close to the Baujade parent, (**Figure 3-15b**). In this case, the series are frequently composed of several successive long shoots.

The studied sample of sequences encompasses a broad spectrum of branching habits ranging from the Wijcik to the Baujade parent one. Hence, the building of a parametric model is mainly used for identifying a discrimination rule to separate the initial sample of branching sequences into two sub-samples.

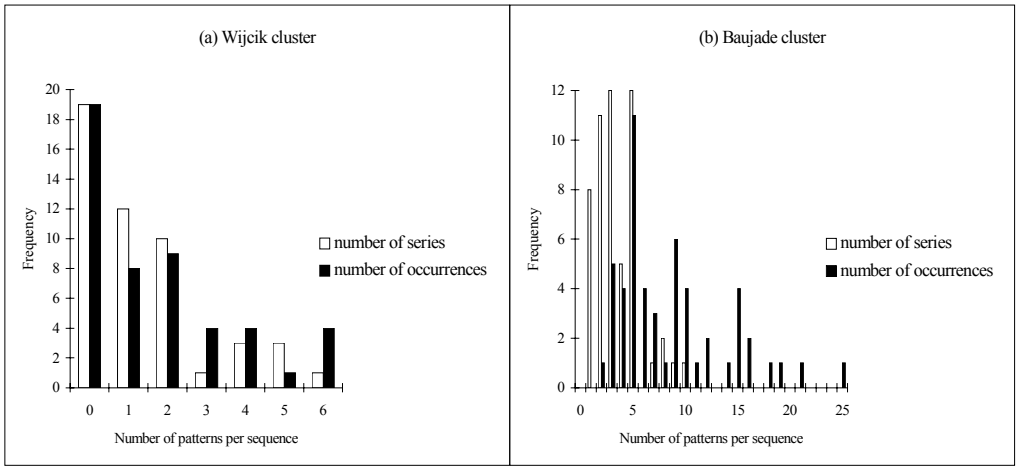


Figure 3-15 Number of series/occurrences of long shoots per sequence for the two sub-populations close to each parent.

4 THE AML LANGUAGE

AML is a functional language which enables the user to build more or less complex data-structures and to explore them by applying various high-level primitives. Some of them are classical data structures like arrays, lists, or histograms for example, while others are specifically dedicated to plant analysis, like plant formal representations, sequentially organized data or various types of stochastic models available in AMAPmod.

The set of system functions that are available in AMAPmod is divided into modules (Kernel, MTG, STAT,...). The kernel module contains functions working on standard types, *i.e.* arithmetic and mathematical functions, functions to deal with strings, arrays, sets, lists, functions to convert types to other types, and general purpose functions.

4.1 Starting an AMAPmod session

An AMAPmod session is launched by the command

```
aml
```

which calls the AMAPmod shell language (or command interpreter): `aml`. This is indicated by the following prompt:

```
AML>
```

which means that the interperter is ready for analyzing user commands. As you type in commands, the AML interprets what is type after each newline. The result of the computation is displayed on the next line, before resuming with a new prompt. AML uses the readline function to buffer the command line, which means that you can edit your line using standard line editor keys (arrows, backspace, character insertion). You can obtain a detailed description of the command line functions and short-hands by invoking the readline man page if available on your system.

```
> man readline
```

If the command is launched without argument, the program looks for a file called "`.aml`" in the current directory. If the file is not found here, aml looks for it in the user's home directory. If this file is finally found, it is interpreted before the interpreter prompt is displayed. Otherwise, the interpreter is launched and a message quoting the fact that the file "`.aml`" could not be found is displayed. The option `-i` deactivate the search for the file "`.aml`".

```
> aml -i
```

A certain number of other options can be passed to the aml command.

- Option `-h` displays the list of all the options available with the aml command.
- Option `-i` "filename" loads file "filename" before displaying the user's prompt.
- Option `-b` "filename1" "filename2" ... loads a series of files in batch mode, in the order where they appear in the argument list.

The user closes an AML session by typing the command :

```
AML> bye
```

During an AML session, all what is typed by the user is recorded after each newline in a file named .amlog in the current directory. You can use this file to restore what was typed during the last session. WARNING, this file is cleared each time you launch aml. If you wish to save the information it contains, you must copy this file into a file with a different name:

```
> cp .amlog my_session.save
```

4.2 AML

A command line in AML enables the user to execute an AMAPmod command. These built-in functions are called primitives and can be used as atomic entities to build new high-level functions.

Objects can be either temporarily built or stored in AML variables.

4.3 Data structures

The AML language contains several types of objects corresponding to different data structures.

4.3.1 Simple types

Simple types are integers (`INT`), reals (`REAL`), booleans (`BOOL`), characters (`CHAR`), strings of characters (`STRING`), and vertices (`VTX`). Types are syntactically detected by AML: when you type `1`, it is automatically recognized as an `INT` whereas when you type `1.0` it is recognized as a `REAL`.

Elementary types can be combined using type constructors in order to define new types. Type constructors are `ARRAY`, `SET` and `LIST`.

4.3.2 Arrays

An array is an object made of an ordered collection of objects of the same type. Let us consider how arrays can be built in AMAPmod on a few examples:

```
AML> # Explicit construction
AML> a1 = [33,2,8,4]
     <ARRAY(INT)> [33,2,8,4]
AML> # Automatic construction
AML> a1 = [1:10]
     <ARRAY(INT)> [1,2,3,4,5,6,7,8,9,10]
AML> # Construction of an array as the value returned by a
     function
AML> vlist = VtxList()
     <ARRAY(VTX)> [1,2,3,4,5,6,7,8,9,10,11,12,13,14]
AML> vlist = Sons(2)
     <ARRAY(VTX)> [3,4,5,6]
AML> …
```

`VtxList()` and `Sons()` are example of AML built-in functions, called *primitives*. Primitives may have mandatory arguments and optional arguments. Function `Sons()` for instance has a mandatory argument of type `VTX`, while function `VtxList()` has no mandatory argument.

Function `Sons()` has optional arguments which can be specified by their name: for example the set of sons connected to their father by a '+' edgetype can be obtained by specifying the optional argument `EdgeType` as follows:

```
AML> Vlist = Sons(2,EdgeType->'+')
     <ARRAY(VTX)> [3,4,5]
```

Similarly, the set of vertices of a given MTG at a given scale may be obtained by specifying the optional argument `Scale`:

```
AML> Vlist = VtxList(Scale->1)
     <ARRAY(VTX)> [1,9]
```

4.3.3 Sets

A set is an unordered collection of objects. A set contains objects with same type and may not contain several objects with same value. For example:

```
AML> s=Set(2,3,10,3,2,2,2,10,11,2)
     <ARRAY(INT)> [2,3,10,11]
AML> s=Set(2,3,Undef,3,Undef)
     <ARRAY(INT)>[2,3,Undef]
AML> s=Set([1,2,3],[9],[],[1,2,3],[9])
     <ARRAY(ARRAY((INT))>[[],[9],[1,2,3]]
```

In a set the order of the element (when displayed) is irrelevant. The operator @ thus cannot be applied to sets. An `ARRAY` can be transformed into a `SET` using primitive `ToSet`:

```
AML> a=Array(2,3,10,3,2,2,2,10,11,2)
     <ARRAY(INT)> [2,3,10,3,2,2,2,10,11,2]
AML> s=ToSet(a)
     <ARRAY(INT)> [2,3,10,11]
```

4.3.4 List

A list is an ordered collection of objects, possibly of different types. In a list, the order of the object is significant (the operator @ can be used):

```
AML> l = List("A",7)
     <LIST(STRING,INT)> [A,7]
AML> l = List("A",Undef,7)
     <LIST(STRING,UNDEF,int)> [A,Undef,7]
AML> l@3
     <INT> 7
AML> l = List("A",7,List([1,91],True))
     <LIST(STRING,INT,LIST(ARRAY(INT),BOOL))>
     [A,7,[[1,91],TRUE]]
AML> l@3@1
     <ARRAY(INT)> [1,91]
```

4.4 Iterators

AML provides a conventional set of statements for expressing selection and looping. Here we will give examples of foreach statements. Consider copyning 10 elements from one array into another :

```
AML> a1 = [1,2,3]
AML> a2 = Foreach _i In a1 : _i + 2
<ARRAY(VTX)> [3,4,5]
```

An iterator is used to go through collection of objects of type ARRAY or SET or LIST (here a1) and to apply a given function (here _i+2) to each object (here represented by variable _i). This is a very generic way to explore data bases and extract samples of different types.

It is often useful in AML to filter arrays (or sets, or lists) according to given boolean criteria. This can be performed with the AML primitive Select. Assume, for example, that we want to extract the values of an array less than a given threshold, we would write:

```
AML> Foreach_i In array1:Select(_i,_i<7)
     (ARRAY(REAL)) [2.2,3.1,0.9,2.1,2.9]
```

4.5 Functions

A user can define AML his own functions on the basis of AML primitives. This AML program is actually defined as a function. The notion of function is very similar to the notion of mathematical function. For instance the mathematical function

$$f(x) = 3x + 2$$

would be defined in AML by :

```
AML> f(_x) = 3*_x+2
```

In this example, f is a user-defined function. AML functions can be considered as a programs which computes output values for given input values. A function f is given an input value in the following way:

```
AML> f(5)
     <INT> 17
```

If one wants to apply a function to a series of values, a first solution would be to apply consecutively the function to each value of the series. However, more generally, a function can be applied in one step to a series of values using an iterator. Assume a series of values is defined in the variable a1 for example:

```
AML> a1 = [1,2,3,4,5,6]
     <ARRAY(INT)> [1,2,3,4,5,6]
AML> a2 = Foreach _x In a1:f(_x)
     <ARRAY(INT)> [5,8,11,14,17,20]
```

The Foreach iterator scans the array a1=[1,2,3,4,5,6] and successively applies the function f to each element. The iterator returns the array made of the values returned by function f on each element of a1 : [f(1),f(2),f(3),f(4),f(5),f(6)]. AML contains conditional functions such as

`If Then Else` whose result depend on a boolean condition. For instance a function for encoding a series of `REAL` values in a binary form depending on whether they are above or under a given threshold would be defined as:

```
AML> encode(_r) = If _r>7 Then 1 Else 0
```

The function can then be applied to encode a series of values in one step:

```
AML> array1=[2.2,3.1,0.9,10.1,9.1,7.0,2.1,2.9]
     <ARRAY(REAL)> [2.2,3.1,0.9,10.1,9.1,7.0,2.1,2.9]
AML> Foreach_i In array1:encode(_i)
     <ARRAY(INT)> [0,0,0,1,1,0,0,0]
```

4.6 Comments and indentation

Files containing AML commands can contain comments. Judicious use of comments and consistent use of indentation can make the task of reading and understanding a program much more efficient. AML uses the # symbol to indicate a comment:

```
# This is a comment of the end of the line
```

When AML encounters the # symbol, the interpretor ignores all text remaining on that line. Comments can be grouped on several lines within forward parenthesis and # as shown here :

```
(# This is a comment
which continues in a second line
#)
```

In this case parenthesis and # open and close the comment. To open the comment (# must be placed on the begining of the line.

If you must write macros complicated, you can define your function on several lines separating each line by a slash:

```
function (_x) = Foreach _i In [1,2,3,4,5,6] :\
                    Foreach _j In [1,2,3,4] : i*j
```

This function is written on two lines and computes the producted of two arrays. Similarly nested `If Then Else` statements can be idented for better legibility:

```
color(_x,_y) = \
     If Size(Sons(_x))==0 And Year(_y)==99 \
     Then return Blue \
          If Class(Father(_x)) == 'S' \
          Then Red \
          Else Yellow \
     Else Black
```

4.7 Access to shell commands

At any time you can launch a shell command by typing command after a `!`:

```
AML> ! ls
my_file1.txt    my_file2.txt    my_file3.txt
AML> ! pwd
```

```
/home/jens/amapmod/oaktree
```

4.8 Input and Output

AML provides different ways of displaying, saving, reading or printing objects. These input and output operators are generic for all objects.

A short description of the value of an object can be displayed in textual form on the standard output by typing its name:

```
AML> array1 = [1,2,3]
AML> …
AML> array1
```

This command displays on the standard output the contents of variable `array1` by giving its type followed by its value :

```
<ARRAY(INT)> [1,2,3]
```

The function `Display` allows the user to display an ASCII representation of an object on the standard output more or less exhaustively. This description depends on the particular object used. The option “Level” can be used to get more or less information about the object.

Several objects can be saved to a file in differents formats. Using the function Save, an objetc is saved by default in an ASCII form. Such a saved object can be loaded using its explicit constructor:

```
AML> array = [1,2,3]
AML> Save(array1,”filename.dat”)
AML> array2 = Array(”filename.dat”)
```

Some objects can be also saved with the `SpreadSheet` format which enables, for example, the user to save a possibly multidimentional array in a format that can be loaded later on with a spreadsheet program like Excel. On the other hand a binary file can be generated using the option `Format->Binary`. The object can be reloaded later with the command `Load`.

```
AML> array=[[0,1,2,3],[1,1,2,4]]
AML> seq = Sequences(array)
AML> Save(seq,”filename.bin”,Format->Binary)
AML> seq2 = Load(”filename.bin”)
```

In fact, in the version 1.2 of AMAPmod, not all objects have a binary format option, such an object need to be recomputed each time AML is launched. Version 2.0 of AMAPmod will have a binary format option for all objects.

For certain types of object (`Array`, `Sequences`, ...), a graphical display can be obtained with the command `Plot`. Except `PlantFrame` objects, the `Plot` function creates gnuplot windows :

```
AML> array1 =[1,2,3]
AML> Plot(array1)
```

On Irix only

A `PlantFrame` objects creates a `glance` window. `Glance` files can be saved by specifying a file name :(by default the file name is “line”)

```
AML> pf = PlantFrame([1]) # PlantFrame rooted in 1
AML> Plot(pf, File->”filename”)
```

AML creates three files respectively called “filename.dta”, “filename.inf”, “filename.lig”. The object can then be displayed using glance when AML is closed. File names which differs only by a terminating number (*i.e.* filename1 and filename2) generate a conflict for `glance` and should be avoided.

5 THE MTG MODULE

A plant architecture described in a coding file can be loaded in AML using primitive `MTG` :

```
g1 = MTG("filename")
```

The MTG primitive attempts to read a valid MTG description and parses the coding file. If errors are detected during the parsing, they are displayed on the screen and the parsing fails. In this case, no MTG is built and the user should make corrections to the coding file. If the parsing succeeds, This function creates an internal representation of the plant (or a set of plants) encoded as a `MTG`. In this example, the `MTG` object is stored in variable `g1` for further use. Note that a MTG should always be stored in a variable otherwise it is destroyed immediately after its building. The last built MTG is considered as the "active" `MTG`. It is used as an implicit argument by all the functions of the MTG module.

It is possible to change the active MTG using primitive `Activate` :

```
AML> g1 = MTG("filename1") # g1 is the current MTG
AML> g2 = MTG("filename2") # g2 becomes the current MTG
AML> Activate(g1)          # g1 is now again the current MTG
```

5.1 AML primitives related to MTGs

In addition to standard primitives for managing `ARRAY`s, `LIST`s, `SET`s, etc., AML provides a set of primitives for accessing more specific objects. There exists for example a comprehensive set of primitives related to MTGs. These primitives may be directly used on the active MTG or they may be combined with each other in order to define new functions on MTGs. Let us give a few examples of these specific primitives.

- MTG constructor: `MTG(STRING)`.

  A MTG can be built from its code file by using the primitive `MTG()` which takes one mandatory argument, *i.e.* the name of the MTG code file.

- Extraction of vertex sets: *e.g.* `VtxList()`.

  Different types of lists of vertices can be extracted from a MTG through the primitive `VtxList()` (see part II, 2-82). Notably, the set of primitives at a given scale is obtained with the optional argument `Scale`.

- Primitives returning vertex attributes: *e.g.* `Class(vtx)`, `Index(vtx)`, `Feature(vtx, feature_name)`.

  The different attributes attached to a given vertex can be retrieved by these functions. The class and the index of a vertex are respectively returned by primitives `Class()` and `Index()`.

  The value of any other attribute may be obtained by specifying its name:

  ```
  AML> Feature(v1,"Length")
  ```

  Returns the `length` (if any) of vertex `v1`. These primitives return *scalar* (`INTEGER`, `STRING`, `REAL`), *i.e.* elementary types different from `VTX`.

- Primitives for moving in MTGs: *e.g.* `Father(vtx); Complex(vtx), Successor(vtx), Predecessor(vtx)`.

Some primitives take a VTX as an argument and return a VTX. These primitives allow topological moves in the MTG, *i.e.* they allow to select new vertices with topological reference to given vertices.

- Primitives for creating collections of vertices: *e.g.* `Sons(vtx)`, `Components(vtx)`, `Axis(vtx)`.

  These primitives return sets of vertices associated with a certain vertex. `Components()` returns all the vertices that compose at the scale immediately superior a given vertex. `Axis()` returns the ordered set of vertices which compose the axis which the argument belongs to.

- Primitives for creating graphical representations of MTGs: `PlantFrame(vtx)`, `Plot(PlantFrame)`, `DressingData(filename)`, `VirtualPattern()`.

  `PlantFrame` enables the user to compute 3D-geometrical representations of MTGs (see part II, 2-49).

The above primitives can be combined together using the AML language to extract from plant databases various types of information.

6 THE STAT MODULE

Le module STAT d'AMAPmod propose un ensemble de méthodes d'analyse de données à base de probabilités discrètes et de processus stochastiques à temps discret et à espace d'états discret. Ces méthodes font appel soit à des techniques non-paramétriques (par exemple calcul d'une distance entre deux séquences) soit à des techniques paramétriques (par exemple estimation des paramètre d'un mélange fini de lois discrètes à partir d'un échantillon de valeurs discrètes). Le module STAT intègre ainsi un ensemble de méthodes exploratoires pour les échantillons de valeurs discrètes et les échantillons de séquences basé essentiellement sur des techniques non-paramétriques. L'approche paramétrique repose d'une part sur des algorithmes d'estimation efficaces et d'autre part sur des méthodes d'évaluation de l'adéquation du modèle estimé à l'échantillon de données utilisé pour l'estimation. Le coeur du module STAT réside dans l'inférence de processus stochastiques à temps discret et à espace d'états discret à partir d'échantillons de séquences discrètes éventuellement multivariées.

Le module STAT a été conçu pour répondre à un certain nombre de problématiques d'analyse de données dans le cadre de l'étude de la croissance de la plante. Ceci explique d'une part que l'on se soit concentré sur le discret étant donné la part prépondérante de ce type de variable aussi bien qualitative (devenir d'un bourgeon axillaire choisi parmi bourgeon latent, rameau court, rameau long et rameau à développement immédiat) que quantitative (nombre d'entrenoeuds d'un unité de croissance, nombre de cycles d'une pousse annuelle, ordre maximum porté…) dans la description de la structure de la plante. D'autre part, la description sous forme de séquences discrètes qui permet de préserver tout ou partie de l'information structurelle est à la base de l'essentiel des méthodes proposées.

Le module STAT peut aussi être utilisé indépendamment du contexte de l'étude de la croissance des plantes grâce à la possibilité de construire des échantillons de données directement à partir de fichiers ASCII.

6.1 L'organisation du module STAT

Les types manipulés par le module STAT d'AMAPmod appartiennent à deux catégories :

- les échantillons de données,
- les modèles.

Les types « données » et les types « modèle » se regroupent en applications figurées par les cadres en pointillés dans la **Figure 6-1** :

- lois et combinaisons de lois (A),
- processus de renouvellement (B),
- modèles Markoviens (C),
- analyse des cimes (D).

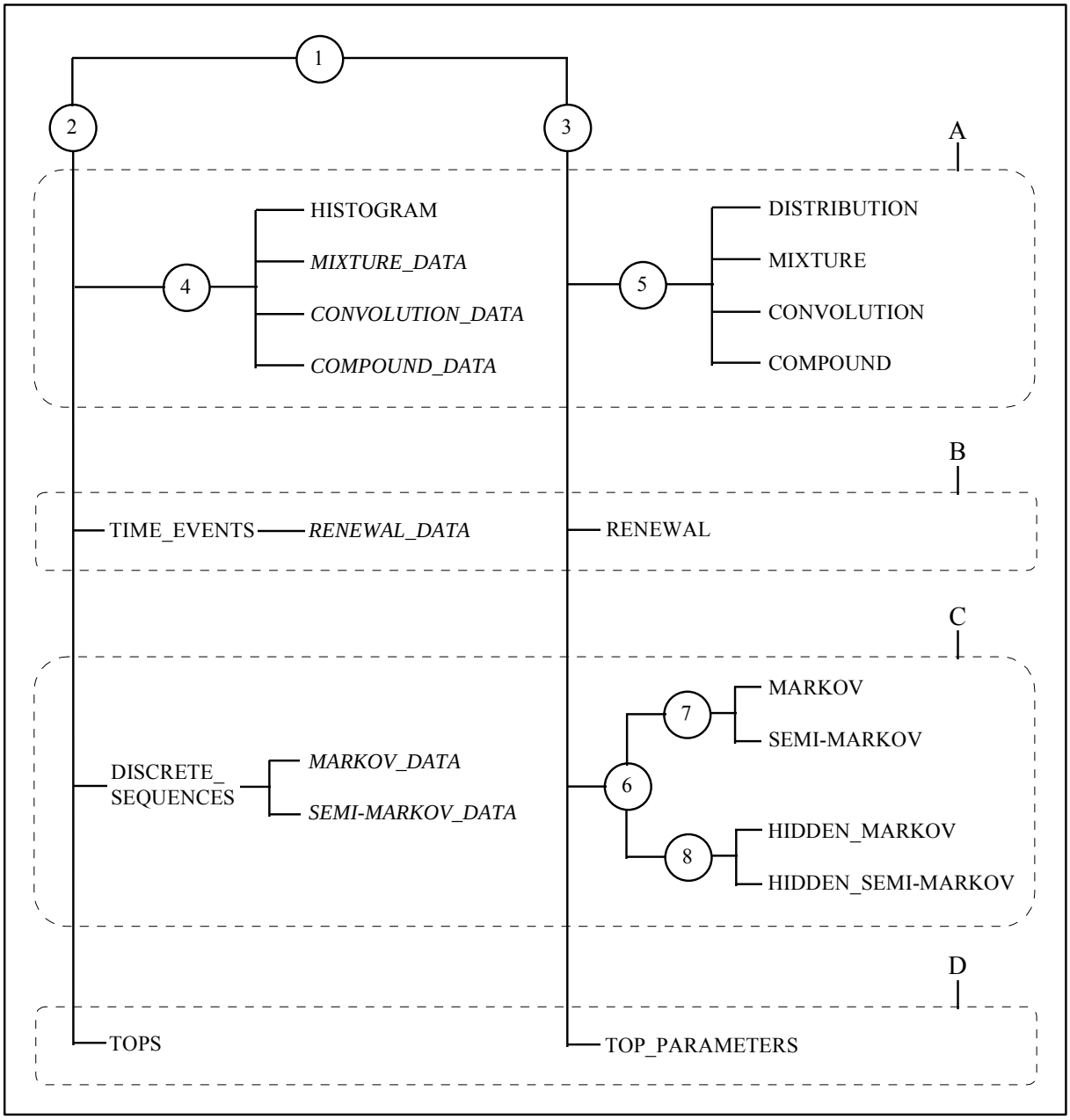


Figure 6-1 Organisation des types du module STAT.

Ces deux niveaux d'organisation sont traduits dans la fig 1. Les différents types sont structurés en une arborescence qui représente la notion d'héritage. Ainsi, les types « données » (type 2) sont des types particuliers (type 1) et les types `HISTOGRAM`, `MIXTURE_DATA`, *`CONVOLUTION_DATA`* et *`COMPOUND_DATA`* sont des types « histogramme » particuliers (type 4). Les sommets numérotés représentent les types dont l'utilisateur ne peut pas créer d'instances (d'objets réels). A chacun de ces types correspond un ensemble de fonctions partagées par tous les types hérités du type en question. Ainsi, tous les types (type 1) partagent un certain nombre de fonctions d'entrée (`Load`) et de sortie (`Display`, `Plot`, `Print`, `Save`). Tous les types « données » (type 2) peuvent être utilisés comme argument de la fonction `Estimate` alors que tous les types « modèle » (type 3) peuvent être utilisés comme argument de la fonction `Simulate`. Les sommets associés à un nom représentent les types dont l'utilisateur peut créer des instances. Ces instances peuvent être obtenues soit par un algorithme à partir d'un objet du module STAT, soit par lecture d'un fichier ASCII ou d'un

fichier binaire, soit par extraction à partir d'une représentation de plantes appelée MTG (*cf.* 5 The MTG module). Les types dont des instance peuvent être crées à partir d'un fichier ASCII ou par extraction sont figurés en fonte standard alors que les types dont les instances sont obligatoirement le résultat d'algorithmes à partir d'un objet du module STAT sont figurés en italique.

6.1.1 Application lois et combinaisons de lois

Le type 5 traduit la notion de loi discrète. Les types hérités du type 5 effectivement utilisables sont les suivants :

- `DISTRIBUTION` : loi discrète,
- `MIXTURE` : mélange fini de lois discrètes,
- `CONVOLUTION` : produit de convolution de lois discrètes,
- `COMPOUND` : loi composée construite à partir de lois discrètes.

Le type `DISTRIBUTION` couvre les lois paramétriques discrètes usuelles (binomiale, binomiale négative, Poisson) munies d'un paramètre de translation. Notons que le loi binomiale négative est définie avec un paramètre réel et une probabilité. Les trois autres types de lois discrètes correspondent à des combinaisons de lois discrètes.

Le type 4 traduit la notion d'ensemble de réalisations d'une variable aléatoire discrète. Les types hérités du type 4 effectivement utilisables sont les suivants :

- `HISTOGRAM` : histogramme,
- *`MIXTURE_DATA`* : données générées par un mélange fini de lois discrètes,
- *`CONVOLUTION_DATA`* : données générées par un produit de convolution de lois discrètes,
- *`COMPOUND_DATA`* : données générées par une loi composée.

6.1.2 Application processus de renouvellement

Le type `RENEWAL` correspond aux processus de renouvellement. Les processus de renouvellement sont construits à partir de lois discrètes, telles que définies dans le type `DISTRIBUTION`, représentant l'intervalle de temps entre 2 événements et appelée loi inter-événement. Le type `TIME_EVENTS` correspond à un ensemble de couples de réalisations de deux variables aléatoires, la première traduisant l'intervalle de temps entre deux dates observation et la seconde, le nombre d'événements survenus entre ces deux dates. Très souvent, l'intervalle de temps entre les deux dates observation est le même pour toutes les mesures de nombre d'événements et ce type peut alors être vu comme un histogramme de nombre d'événements survenus pendant un intervalle de temps fixé donné. Le type *`RENEWAL_DATA`* hérité du type `TIME_EVENTS` correspond à des données générées par un processus de renouvellement.

6.1.3 Application modèles Markoviens

Le type 6 se décomposent en deux types, les types 7 et 8 qui traduisent respectivement la notion de modèle Markovien et de modèle Markovien caché.

Les types hérités du type 7 effectivement utilisables sont les suivants :

- `MARKOV` : chaîne de Markov,

- SEMI-MARKOV : semi-chaîne de Markov.

Les types hérités du type 8 effectivement utilisables sont les suivants :

- HIDDEN_MARKOV : chaîne de Markov cachée,
- HIDDEN_SEMI-MARKOV : semi-chaîne de Markov cachée.

Les chaînes de Markov, de même que les chaînes de Markov cachées sont d'ordre quelconque (dans la pratique limité à 4). Il est possible de s'intéresser à des chaînes de Markov non-homogènes, c'est à dire telles que les probabilités de transition dépendent de l'index. Les lois d'occupation des états des semi-chaînes de Markov et des semi-chaînes de Markov cachées sont des lois discrètes paramétriques telles que définies dans le type DISTRIBUTION avec la restriction que le paramètre de translation est supérieur ou égal à 1 ce qui traduit le fait que l'on reste au moins un instant dans un état. Enfin, ces types de modèle s'appliquent de manière intéressante si le nombre de réalisations possibles de chacune des variables aléatoires indexées est limité (à 10 par exemple). Par contre, il n'y a pas de contraintes sur les natures des états de ces modèles (combinaison quelconque d'états récurrents, transitoires ou absorbants).

Le type DISCRETE_SEQUENCES traduit la notion d'ensemble de séquences discrètes. On entend par séquence discrète une suite de vecteurs aléatoires discrets indexés par un paramètre. Le type *MARKOV_DATA*, hérité du type DISCRETE_SEQUENCES, correspond à des données générées par des chaînes de Markov ou des chaînes de Markov cachées alors que le type *SEMI-MARKOV_DATA*, aussi hérité du type DISCRETE_SEQUENCES, correspond à des données générées par des semi-chaînes de Markov ou des semi-chaînes de Markov cachées.

Deux types annexes non-représentés sur la **Figure 6-1** font partie de l'application modèles Markoviens :

- SEQUENCES : séquences assujetties à des contraintes plus faibles que les séquences représentées dans le type DISCRETE_SEQUENCES et ne pouvant donc servir d'entrée à l'estimation des paramètres d'un modèle Markovien,
- CORRELATION : coefficients de corrélation calculés à partir d'un ensemble de séquences.

6.1.4 Application analyse des cimes

Le type TOP_PARAMETERS correspond aux paramètres d'une cime (probabilité de croissance axe porteur, probabilité de croissance axe porté et rapport de rythme d'élongation axes portés/axe porteur). Le type TOPS correspond à un ensemble de cimes, c'est à dire à un ensemble de systèmes ramifiés avec un seul ordre de ramification.

Enfin, nous avons les cinq types annexes suivants :

- VECTORS : ensemble de vecteurs,
- VECTOR_DISTANCE : paramètres de définition d'une distance entre vecteurs,
- DISTANCE_MATRIX : matrice des distances/dissimilarités entre formes,
- CLUSTERS : résultat d'une partition en *k* groupes d'un ensemble de formes à partir de la matrice des distances entre formes,
- REGRESSION : résultats d'une régression simple.

6.2 Les fonctions AML du module STAT

Nous distinguons trois catégories de fonctions :

- les fonctions d'entrées/sorties,
- les fonctions de manipulation des données,
- les fonctions algorithmiques permettant notamment de créer un objet de type « modèle » à partir d'un objet de type « données » par estimation ou de créer un objet de type « données » à partir d'un objet de type « modèle » par simulation.

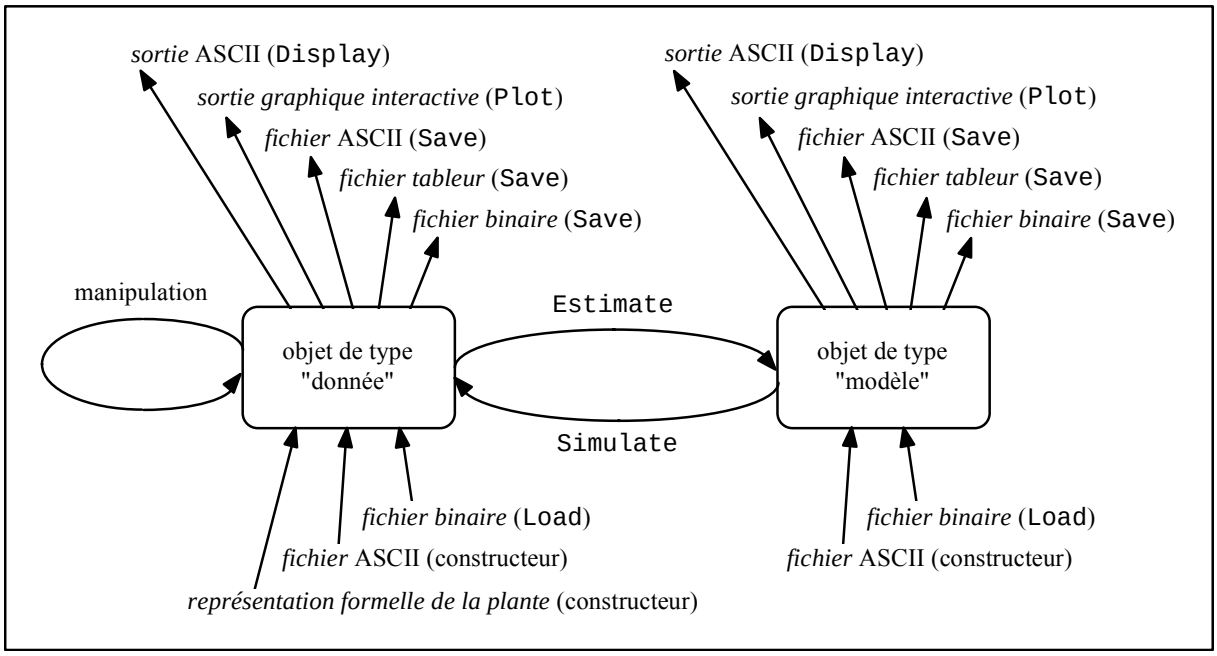


Figure 6-2 Schéma de principe d'application des fonctions aux objets.

6.2.1 Les fonctions d'entrées/sorties

A chaque type figuré en fonte standard sur la **Figure 6-2** correspond une forme syntaxique qui permet de définir une instance de ce type dans un fichier ASCII. La forme syntaxique des types « données » se rapproche de tableaux de nombres alors que la forme syntaxique des types « modèle » est construite à partir de mots clés qui traduisent la structure du modèle. Par convention, le séparateur est une suite quelconque d'espaces et de tabulations. Il est possible d'insérer des commentaires (ligne commençant par un # ou fin de ligne après le #) dans ces fichiers ASCII. Les fonctions d'entrée ou constructeur ont pour nom le type de l'objet créé. Par exemple, la fonction `Histogram` construit l'objet `histo` de type `HISTOGRAM` à partir du fichier `"exemple.his"`.

```
histo = Histogram("exemple.his")
```

Les objets de type `DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`, `RENEWAL` peuvent être construits à partir de lois discrètes ou de familles de lois discrètes, c'est à dire d'objets de type `DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`. Les objet de type « données » peuvent être construits soit à partir de fichiers ASCII, soit à partir de structures de données extraites d'un MTG.

Il est possible de visualiser tout objet à l'écran au format ASCII grâce à la fonction `Display`.

En plus de la forme syntaxique définissant l'objet, différentes informations supplémentaires sont affichées, ce qui permet d'avoir un compte rendu du traitement ayant généré l'objet. Le niveau de détail de ces informations supplémentaires est géré par l'argument optionnel `Detail`. La forme ASCII d'un objet peut être imprimée par le fonction `Print`.

Un objet peut être sauvegardé dans un fichier grâce à la fonction `Save`. Trois formats de fichier sont possibles :

- format ASCII (`Format->ASCII`),
- format binaire (`Format->Binary`),
- format Tableur (`Format->SpreadSheet`).

Les fichiers au format ASCII sont identiques à ce que sort à l'écran la fonction `Display` pour un niveau de détail donné. Tout objet du module STAT peut être sauvegardé au format binaire et rechargé grâce à la fonction `Load`. Les fichiers au format Tableur sont destinées à la mise en page de graphiques en vue de la production de documents.

Un objet peut être visualisé graphiquement grâce à la fonction `Plot`. Les visualisations graphiques sont faîtes par le logiciel GNUPLOT.

6.2.2 Les fonctions de manipulation des données

Différentes manipulations sont possibles sur les données. Il est ainsi toujours possible de concaténer des ensembles de données du même type (fonction `Merge`). De nombreuses manipulations spécifiques sont aussi possibles.

6.2.3 Les fonctions algorithmiques

Les trois principales fonctions sont la fonction `Estimate` qui crée un objet « modèle » à partir d'un objet « données » par estimation, la fonction `Simulate` qui crée un objet de type « données » à parti d'un objet de type « modèle » par simulation et la fonction `Compare`. La fonction `Compare` calcule des mesures de dissimilarités entre histogrammes, ou des distances entre vecteurs ou entre séquences, ou les vraisemblances de séquences discrètes pour une famille de modèles Markoviens (chaîne de Markov, semi-chaîne de Markov, chaîne de Markov cachée ou semi-chaîne de Markov cachée) ou encore des divergences entre modèles Markoviens.

La fonction `Clustering` réalise la partition en *k* groupes d'un ensemble de formes à partir de la matrice des distances entre formes. La fonction `ComparisonTest` compare deux histogrammes au moyen de tests d'hypothèses. La fonction `ContingencyTable` calcule un tableau de contingence à partir d'un ensemble de vecteurs. La fonction `ModelSelectionTest` teste l'ordre ou l'agrégation des états d'une chaîne de Markov à partir d'un ensemble de séquences discrètes. La fonction `Regression` réalise une régression linéaire ou non-paramétrique simple (une seule variable explicative). La fonction `Segment` permet de segmenter des séquences discrètes en utilisant une chaîne de Markov cachée ou une semi-chaîne de Markov cachée. Cette fonction crée donc un objet de type « données » à partir

d'un objet de type « données » initial et d'un objet de type « modèle ». La fonction `VarianceAnalysis` réalise une analyse de variance à un facteur.

Part II REFERENCE MANUAL

1 THE KERNEL MODULE OF AML

This part describes the functions of the kernel module of the AML language.

1.1 Liste alphabétique des fonctions AML

1.2 Liste par type des fonctions AML

Arithmetic operators

```
e1 + e2
e1 - e2
e1 * e2
e1 / e2
- e1
e1 Mod e2
Abs(e1)
Floor(e1)
Ceil(e1)
Rint(e1)
Truncate(e1)
ToInt(e1)
ToReal(e1)
```

Numeric functions

```
Sqrt(e1)
Log(e1)
Log10(e1)
```

```
Exp(e1)
e1 ^ e2
Cos(e1)
Sin(e1)
Tan(e1)
Acos(e1)
Asin(e1)
Atan(e1)
```

Constants

```
Undef UNDEF
Black INT
White INT
Green INT
Red INT
Blue INT
Yellow INT
Violet INT
LightBlue INT
True BOOL
False BOOL
CurrentPlottedObj ANY
CurrentWindow WINDOW
DefaultWindow WINDOW
Pi REAL
```

Boolean operators

```
e1 And e2
e1 Or e2
Not e1
```

Logical operators

```
e1 == e2
e1 != e2
```

Comparison operators

```
e1 < e2
e1 <= e2
e1 > e2
e1 >= e2
```

Operators on Dates

```
Second INT
Minute INT
Hour INT
Day INT
DateUnit INT
DateFormat STRING
e1 + e2
```

```
e1 - e2
Date(e1)
ToTimeUnit(e1)
```

Control expressions

```
Select(e1,pred)
If e1 Then e2 Else e3
Switch e1 Case e2 : e3 Case e4 : e5 ... Default : e6
```

Set type operators

```
[e1,e2,...,eN]
[e1:e2]
[e1:e2:e3]
Array(e1,e2,...,eN)
Set(e1,e2,...,eN)
List(e1,e2,...,eN)
e1 @e2
Head(e1)
Tail(e1)
Pos(e1,e2)
e1+e2
e1-e2
Append(e1,e2)
Remove(e1)
InsertAt(e1,e2)
RemoveAt(e1,e2)
Invert(e1,e2)
ToArray(e1)
ToSet(e1)
ToList(e1)
Sum(e1)
Prod(e1)
Max(e1)
Min(e1)
Flatten(e1)
Size(e1)
Sort(e1)
e1 | e2
e1 & e2
SetMinus(e1, e2)
Filter(e1, e2)
EDist(e1,e2)
SProd(e1,e2)
VProd(e1,e2)
Angle(e1,e2)
Norm(e1)
```

Iterator

```
Foreach _c In e1 : e2
```

General functions

```
EchoOn()
EchoOff()
Delete(e1,e2,...,eN)
Display()
Display(e1)
DisplayAll()
Save(e1)
Plot(e1)
NewPlot(e1)
```

1.3 Detailed description

Abs

Absolute value.

USAGE

```
Abs(x)
```

ARGUMENTS

`x` (`INT` or `REAL`) : a numerical value

RETURNED OBJECT

The value returned by `Abs` has the same type as `x`. If `x` is `Undef`, returns `Undef`.

DESCRIPTION

Returns the absolute value of `x`.

NOTE

This function is similar to the corresponding double function of the host system (Unix, ...).

SEE ALSO

`Ceil`, `Floor`, `Truncate`, `Rint`, `ToInt`, `ToReal`.

EXAMPLES

```
AML>  Abs(3)
      <INT>3
AML>  Abs(-3)
      <INT>3
AML>  Abs(-3.1)
      <REAL>3.1
```

AllPos

The set of all positions of an element in an array or a list.

USAGE

```
AllPos(array, element)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(T)`). `T` is any type. If the argument is `Undef`, returns `Undef`
`element` (`T`). element that is searched for in the `array`.

RETURNED OBJECT

The function returns an array of integers `ARRAY(INT)`. If the argument is `Undef`, returns `Undef.`

DESCRIPTION

`AllPos` returns the set of positions of all the occurences of `element` in `array`.

SEE ALSO

`ARRAY` constructor, `Tail`, `At`, `Head`.

EXAMPLES

```
AML> AllPos([10,11,10,13,10,15],10)
     <ARRAY(INT)>[1,3,5]
AML> AllPos([1,2,3,2,2,3,4,Undef,3],Undef)
     <ARRAY(INT)>[8]
AML> AllPos([1,2,3,2,2,3,4,Undef,3],7)
     <ARRAY(ANY)>[]
```

Angle

Angle made by the vectors corresponding to two arrays of reals.

USAGE

```
Angle(a1, a2)
```

ARGUMENTS

`a1, a2` (`ARRAY(REAL)`). These two arrays must have identical size.

RETURNED OBJECT

The result is an angle expressed in radians (real between 0 and Pi). If one of the arguments is `Undef`, returns `Undef`. If the dimension is 0 the result is `Undef`.

DESCRIPTION

Angle made by the vectors corresponding to two arrays of reals.

SEE ALSO

`EDist`, `SProd`, `Norm`, `VProd`, `Array`.

EXAMPLES

```
AML> Angle([1.,0.,0.],[0.,1.,0.])
     <REAL> 1.5708
AML> Angle([1.,0.,0.],[1.,1.,1])
     <REAL> 0.955317
```

Append

Appends a value to ARRAYs, SETs, STRINGs.

USAGE

```
Append(set, val)
Append(stg1, stg2)
```

ARGUMENTS

set (ARRAY(T)) or set (SET(T)) or set (LIST(...)),
val (T), value that must be appended at the end of set.
stg1, stg2 (STRING): contatenation of a character to a STRING.

RETURNED OBJECT

- When an element appended to a set object set, the initial object is physically affected by the operation. The value returned is the (modified) argument itself (and not a modification of a copy of the argument). If set is Undef, returns Undef. If set is not Undef and val is Undef, then Undef is appended to the argument.
- For a STRING, on the contrary, the concatenation of the two strings is a new STRING, different from both stg1 and stg2.

DESCRIPTION

If set is an ARRAY, Append(set,val) makes the element val beappended at the end of set.

SIDE EFFECT

Important: contrary to most functions in AML which leave their argument unchanged, this fonction modifies its set argument. The value returned is the modified argument.

SEE ALSO

Union, Inter, SetMinus, Date operators, ToSet, ToArray.

EXAMPLES

```
AML> a=[1,2,4,3,4,5,6,4]
     <ARRAY(INT)> [1,2,4,3,4,5,6,4]
AML> Append(a,2)
     <ARRAY(INT)> [1,2,4,3,4,5,6,4,2]
AML> a
     <ARRAY(INT)> [1,2,4,3,4,5,6,4,2]
AML> s=Set(1,2,4,3,4,5,6,4)
     <SET(INT)> [1,2,3,4,5,6]
AML> Append(s,2)
     <SET(INT)> [1,2,3,4,5,6]
AML> Append(s,7)
     <SET(INT)> [1,2,3,4,5,6,7]
AML> s
     <SET(INT)> [1,2,3,4,5,6,7]
```

```
AML> # Concatenation of strings
AML> stg1=”AMAP”;stg2=”mod”
     <STRING> mod
AML> Append(stg1,stg2)
     <STRING> AMAPmod
AML> stg1
     <STRING> AMAP
```

Arithmetic operators

+,-,*,/ Classical arithmetic operators. Addition of a value to ARRAYs, SETs, LISTs, STRINGs and DATEs.

USAGE

`a + b`, `a - b`, `a * b`, `a / b`, `- a` : arithmetic operators.
`set + val`, `set – val` : physical addition of an element to a set object.
`stg1 + stg2` : contatenation of a character to a STRING.
`d1 + i`, `d1 – i` : addition of i time units to a date.

ARGUMENTS

`a`, `b` (INT or REAL) : numerical values. For arithmetic operators, any combination of types INT and REAL is possible.
`set` (ARRAY(T)) or `set` (SET(T)) or `set` (LIST(...)), `val` (T)
`stg1` (STRING), `stg2` (STRING)
`d1` (DATE), `i` (INT).

RETURNED OBJECT

- For arithmetic operators, the result is INT only in the case where both a and b are INTs. Otherwise the result is REAL. If either `a` or `b` is `Undef`, returns `Undef`. A division by 0 returns an `Undef` value.
- When an element is added to or removed from a set object `set`, the initial object is physically affected by the operation. The value returned by either these operators is the (modified) argument itself (and not a modification of a copy of the argument). If `set` is `Undef`, returns `Undef`. If `set` is not `Undef` and `val` is `Undef`, then the result
- For a STRING, on the contrary, the concatenation of the two strings is a new STRING, different from both `stg1` and `stg2`.
- For date incrementation, *cf.* `Date` operators.

DESCRIPTION

- If `a` and `b` are integer values, `a / b` performs an integral division. The arguments have to be casted to REALs if one wants to apply a division between arguments considered as REAL numbers.
- '+' and '-' can be used to physically add or remove elements from a set object.
- If `set` is an ARRAY, `set + val` makes the element `val` be appended at the end of `set`. `set - val` removes the first occurence of element `val` in the array `set1`.
- '+' and '-' can be used to concatenate two strings.
- '+' and '-' can be used for adding `i` time units to a date, *cf.* `Date` operators.

SIDE EFFECT

Important: contrary to most functions in AML which leave their argument unchanged, this fonction modifies its `set` argument. The value returned is the modified argument.

SEE ALSO

`Union`, `Inter`, `SetMinus`, `Date` operators, `ToSet`, `ToArray`.

EXAMPLES

```
AML> # Arithmetic operators
AML> 9 / 2
     <INT>4
AML> 5 + 3
     <INT>8
AML> 5.0 + 3
     <REAL>8
AML> 9 / 0
     Undef
AML> # Addition / Deletion of elements of set objects
AML> a=[1,2,4,3,4,5,6,4]
     <ARRAY(INT)> [1,2,4,3,4,5,6,4]
AML> a + 2
     <ARRAY(INT)> [1,2,4,3,4,5,6,4,2]
AML> a
     <ARRAY(INT)> [1,2,4,3,4,5,6,4,2]
AML> a - 4
     <ARRAY(INT)> [1,2,3,4,5,6,4,2]
AML> a - 4
     <ARRAY(INT)> [1,2,3,5,6,4,2]
AML> s=Set(1,2,4,3,4,5,6,4)
     <SET(INT)> [1,2,3,4,5,6]
AML> s + 2
     <SET(INT)> [1,2,3,4,5,6]
AML> s + 7
     <SET(INT)> [1,2,3,4,5,6,7]
AML> s
     <SET(INT)> [1,2,3,4,5,6,7]
AML> s - 2
     <SET(INT)> [1,3,4,5,6,7]
AML> # Concatenation of strings
AML> stg1 = ”AMAP” ; stg2 = ”mod”
     <STRING> mod
AML> stg1 + stg2
     <STRING> AMAPmod
AML> stg1
     <STRING> AMAP
```

Array

ARRAY constructor

USAGE

```
[x1,x2,...,xn]
[i:j]
[i:j:step]
```

ARGUMENTS

x1, x2,..., xn (T) : values of identical type. Type T can be any type. Any xk can be Undef.
i, j, step (INT or REAL): for incremental constructor.

RETURNED OBJECT

In the constructor form x1, x2,..., xn, assuming the type of xk is T, the value returned an array constructor is an ARRAY(T) if at least one is not Undef. If all are Undef the constructor returns Undef.
In the incremental form of the constructor, if all the arguments i, j, step are integers, the array returned has type ARRAY(INT). If any of the argument is REAL, the array returned has type ARRAY(REAL). If any of the arguments is Undef, the result is Undef.

DESCRIPTION

An ARRAY is a set-type, *i.e.* the type of a collection of objects. Like for SETs, all the elements of an ARRAY must have the same type. However, an ARRAY can contain Undef elements. Contrary to SETs, elements of an ARRAY are ordered by a total order relation. It is thus meaningful to speak of the ith element of an ARRAY, provided i is greater than 0 and less than the size of the array.
An incremental constructor, [i:j], enables the user to build incremental series of values starting at a lower-bound i and finishing at a higher-bound j. The default increment value is 1 if i≥j and -1 if i>j. If i≤j for example, the resulting ARRAY is [i, i+1, i+2,..., i+*m*, j] where *m* is such that i+*m*<j≤i+*m*+1. To change this increment, one must specify the step of the increment by using [i:j:step].

DETAILS

- In version 1.x of AMAPmod there is no way of saving or loading binary representations of arrays

- An array can be saved in ASCII format, using primitive Save:

```
AML> Save(array1,"filename")
```

For exporting data to other programs, an option Format is provided that save an array in the selected format. Current values of Format option can be ASCII (default option) or SpreadSheet. The SpreadSheet value enables the user to save a possibly

multidimentional array in a format that can be loaded later on with a spreadsheet program:

```
AML> array1=[[1,2,3],[4,5,6],[7,8,9]]
AML> Save(array1,"filename",Format→SpreadSheet)
```

This command saves `array1` in file `"filename"` as follows:

```
#2
1 2 3
4 5 6
7 8 9
```

- An array that has been saved with the previous command in ASCII format can be loaded in ASCII format using its explicit constructor `Array`:

```
AML> Array("filename")
```

A file which consists of objects of type `INT`, `REAL`, `STRING`, having the same type, separated by white space characters can be loaded as a multidimentional array as follows:

```
AML> Array("filename",[n1,n2,...,n_{m-1}])
```

where filename is the name of a file containing homogeneous ASCII data (of only one type). The data in the file can be any of the simple types: `INT`, `CHAR`, `REAL`, `BOOL`, `DATE`, `STRING`. If a multidimentional array with *m* dimensions is expected, then the number of elements of each dimension, except the last one *nm*, must be indicated as an argument, in increasing order of dimensions. The number of element of the last dimension derives from the numbers other dimensions and from the size of the file. For example, assume we have an ASCII file of 21 integers and we want to load this file into a 2-dimensional array whose first dimension has 7 elements and whose second dimension has 3 elements. Then, we must load the file as follows:

```
AML> Array("filename",[7])
     <ARRAY(ARRAY(int))> [[46,58,4,10,11,-23,0],[5,16,
     -7,15,17,-29,1],[36,26,5,12,11,-13,1]]
```

The same file considered as a monodimensional array would be loaded as follows:

```
AML> Array("filename",[])
     <ARRAY(INT)> [46,58,4,10,11,-23,0,5,16,-7,15,17,
     -29,1,36,26,5,12,11,-13,1]
```

- For certain types of arrays, a graphical display of an array can be obtained with the command `Plot`. These arrays are either 1- 2- or 3- dimensional arrays of `INT`s, `REAL`s or `DATE`s. These arrays can be Plotted as follows:

`Plot` of 1-dimensional arrays:

```
AML> y1 = [2,4,22,40,49,53,55,45,33,27,19,13,10,12,7]
AML> Plot(y1)#fig
```

This simple command assumes that the elements of array `y1` are *y*-coordinates of points whose *x*-coordinate can be implicitly defined by the rank of the element in the array `y1`. The output of the `Plot()` command thus looks as follows:
In this graphic points are linked to each other by a line. Other styles of plot can be usedby specifying the optional parameter `Style`. `Style` can be one of the following: `lines`, `points`, `linespoints`, `dots`, `impulses`, `steps`, `boxes` (`linespoints` is the default).

```
AML> Plot(y1,Style->"boxes")
```

The default values of the implicit *x*-coordinates can be changed with optional parameters `XOutSet`, `XStep`, `XMax`, `XMin`. Similarly, the bounds of the *y* axis that are graphically displayed is estimated from the values of array `y1`. These bounds can be changed with options `YMin` and `YMax`.

```
AML> Plot(y1,XOutset->-10,XStep->2,XMax->50,YMax->100) # fig
```

Titles can be given to coordinate axes and to the plot with options `XTitle`, `YTitle`, `Title` which must take `STRING` values.

```
AML> year=1997
AML> Plot(y1,Title->"Observationsin"+ToString(year),\
AML> Xtitle->"Branchorde",YTitle->"amount of Apples") # fig
```

If both *x* and *y* coordinates of every points are explicitly defined, it is possible to plot the points in two different ways. First, we may give an array of points defined by their (*x*, *y*) coordinates:

```
AML> p1=[[10,2],[15,4],[22,22],[24,40],[30,49],[45,53],\
AML> [58,45],[61,33],[67,27],[80,19],[87,13],[90,10],\
AML> [95,12],[100,7]]
```

We thus have a set of points whose *x*- and *y*-coordinates are respectively in the arrays `x1` and `y1`. To plot the corresponding points, we have to specify with option `XAxis` which of them corresponds to the *x* coordinates

```
AML> Plot(p1,XAxis->1) # fig
```

In this expression `XAxis` has value 1, which means that the first element of each array defining the coordinates of points of `p1` must be interpreted as the *x*-coordinate of the point.
This method can be extended for plotting several curves on the same graphic. If we assume that for each *x*-coordinate, two *y*-coordinates are defined, cooresponding respectively to two different curves. We can make up 3uples (x, y_1, y_2) containing one *x*-coordinate and the two corresponding *y*-coordinates for the two curves, *i.e.* respectively `y1` and `y2`. Thus defining the two curves comes down to giving a set of 3-uples like:

```
AML> p2=[[10,2,-1],[15,4,-6],[22,22,-12],[24,40,-5],\
AML> [30,49,-1],[45,53,-6],[55,55,-10],[58,45,-6],\
AML> [61,33,-2],[67,27,-5],[80,19,-13],[87,13,-7],\
```

```
AML> [90,10,-1],[95,12,-5],[100,7,-15]]
AML> Plot(p2,XAxis->1)#fig
```

This method can be extended to any number of curves *n*. In this case an array pn of *n*-uples has to be built. The command `Plot(pn)` plots *n* curves with implicit *x*-coordinates. If the *i*th value of the *n*-uples designates the *x*-coordinates, this can be indicated by option `XAxis`:

```
AML> Plot(pn,XAxis->i)
```

which plots *n*-1curves.
There exists another way to plot several curves in the same window. One has to create an array representing each series of *y* values (one series for each curve and represented as an array like `y1`). For instance, to plot the two curves corresponding to array `y1` and to array `x1` (defined below), we just have to specify that the arrays `y1` and `x1` corresponds to groups of data associated with our curves. This is done with option `Groups` which specifies how should be interpreted the elements (which are arrays of integers `y1` and `x1`) of the array passed as an argument to plot:

```
AML> x1=[10,15,22,24,30,45,55,58,61,67,80,87,90,95,100]
AML> Plot([y1,x1],Groups->"curves").
```

In this case, the *x*-coordinates of curve points are defined implicitly: elements of `y1` corresponds to a first group of *y*-coordinates (defining the first curve) and elements of `x1` corresponds to a second group of *y*-coordinates (defining a second curve). The *x*-coordinates can be explicitly defined by specifying a group of values associated with them. In our example, we can for instance specify with option `XAxis` that the second array, `x1`, actually corresponds to the *x*-coordinates of the curve points whose *y*-coordinates are given in the first array `y1`:

```
AML> Plot([y1,x1],Groups->"curves",XAxis->2).
```

NOTE

Two arrays are equal whenever their *i*th element are equal, for all possible `i`. This notably entails that two identical arrays must have the same size.

SEE ALSO

`Size`, `At`, `Sum`, `Series`, `Pos`, `AllPos`, `Invert`, `Set`, `List`, `ToArray`, `ToSet`, `ToList`.

EXAMPLES

```
AML> a=[".aml",".mtg",".amlog",".hmm"]
     <ARRAY(STRING)> [.aml,.mtg,.amlog,.hmm]
AML> a=[[1,2],[3,4,5],Undef,[7],[8,9,10]]
     <ARRAY(ARRAY(INT))>[[1,2],[3,4,5],Undef,[7],[8,9,10]]
AML> a=[1:9]
     <ARRAY(INT)>[1,2,3,4,5,6,7,8,9]
AML> a=[1:9:2]
     <ARRAY(INT)>[1,3,5,7,9]
```

```
AML>  a=[9:1:-2]
      <ARRAY(INT)>[9,7,5,3,1]
AML>  a=[9:1]
      <ARRAY(INT)>[9,8,7,6,5,4,3,2,1]
AML>  a=[1.1:9:2.7]
      <ARRAY(REAL)>[1.1,3.8,6.5]
AML>  a=[Date("01/01/97"),Date("11/01/97"),Date("24/01/97"),\
AML>  Date("03/02/97")]
      <ARRAY(DATE)> [01/01/97, 11/01/97, 24/01/97, 03/02/97]
```

At

*i*th element of an array, @.

USAGE

`array@i`

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type
`i` (`INT`). It can have either a positive or a negative value. It cannot be 0.

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `T`. If `array` has type `LIST(T1,T2,...,TN)` the function returns an element of type T_i.

DESCRIPTION

Returns the *i*th element of an array or a list. If `i` has a negative value, `@` returns the *i*th element with respect to the end of `array`, *i.e.* if *n* is the size of `array`, it corresponds to the *n*-`i`+1 the element of `array`.

SEE ALSO

ARRAY constructor, `Pos`.

EXAMPLES

```
AML>  i=[10,11,12,13,14,15]@5
      <INT> 14
AML>  i=[10,11,12,13,14,15]@1
      <INT> 10
AML>  i=[10,11,12,13,14,15]@-1
      <INT> 15
AML>  i=[10,11,12,13,14,15]@-5
      <INT> 11
AML>  l=List("A",3,[1,2,3],True,67,9.8)@3
      <ARRAY(INT)> [1,2,3]
```

Boolean operators

True, False, And, Or, Not

USAGE

```
x And y
Not(x != True)
x Or y
```

ARGUMENTS

`True` and `False` (`BOOL`) are boolean constants.

`x`, `y` (`BOOL` or `INT`). If an `INT` is used, a non-zero value is consired as a `True` value and 0 is equivalent to `False`.

RETURNED OBJECT

The values by these functions are boolean (type `BOOL`). Any of the arguments may have value `Undef`. In this case, the result is always `Undef`.

DESCRIPTION

These are classical boolean operators. They are commonly used to make up predicates, *i.e.* functions that return a boolean value.

SEE ALSO

`Comparison` operators, `Select`, `If-Then-Else`, `Switch`.

EXAMPLES

```
AML> 1 And True
     <BOOL> True
AML> Not(Not(0 Or False))
     <BOOL> False
AML> Undef And True
     Undef
AML> # Definition of apredicate
AML> pred(_x) = If_x > 3 And _x < 10 Then True Else False
     <FUNC> Function
AML> pred(4)
     <BOOL> True
AML> pred(11)
     <BOOL> False
```

Ceil

Integer no less than.

USAGE

```
Ceil(x)
```

ARGUMENTS

x (INT or REAL) : a numerical value

RETURNED OBJECT

The value returned by Ceil has the same type as x. If x is Undef, returns Undef.

DESCRIPTION

Returns the integer no less than x.

NOTE

This function is similar to the corresponding double function of the host system (Unix, ...).

SEE ALSO

Abs, Floor, Truncate, Rint, ToInt, ToReal.

EXAMPLES

```
AML> Ceil(3)
     <INT>3
AML> Ceil(3.1)
     <REAL>4
AML> Ceil(-3)
     <INT>-3
AML> Ceil(-2.9)
     <REAL>-2
```

Comparison operators

==, !=, <, <=, >, >=

USAGE

```
x == y
x != y
x <= y
```

ARGUMENTS

x, y (ANY) for logical equality/difference operators == and !=. The arguments must have identical types.
x, y (INT, REAL, DATE) for order comparison operators <, <=, >, >=

RETURNED OBJECT

The returned values are boolean values (type BOOL). Any of the argument may have value Undef.

DESCRIPTION

The logical truth value of expressions x == y and x != y is defined by the type of object that are being compared (for the precise definition associated with a particular type of object, look at the constructor functions of that object).

SEE ALSO

AML Objects constructors, Not, And, Or.

EXAMPLES

```
AML> 1 == (9 Mod 4)
     <BOOL> True
AML> Not(1 == (9 Mod 4))
     <BOOL> False
AML> Undef == 1
     <BOOL> False
AML> Undef == Undef
     <BOOL> True
AML> Date("04/01/97") < Date("05/01/97")
     <BOOL> True
```

Constants

`Undef`, `True`, `False`, `Pi`, `Black`, `White`, `Green`, `Red`, `Blue`, `Yellow`, `Violet`, `LightBlue`, `CurrentPlottedObj`, `CurrentWindow`, `DefaultWindow`, `Second`, `Minute`, `Hour`, `Day`, `DateUnit`, `DateFormat`.

Cov

Computes the covariance of elements of two arrays.

USAGE

```
Cov(a1,a2)
```

ARGUMENTS

`a1,a2` (`ARRAY(T)` or `SET(T)`). `T` is either `INT` or `REAL`. `Arguments must have the same type`

RETURNED OBJECT

If `a1 and a2` have type `ARRAY(T)` or `SET(T)`, the function returns an element of type `REAL`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the mean value of the elements of `array`. If `a1` = $\langle x_1, x_2, .., x_i, .. \rangle$ and `a2` = $\langle y_1, y_2, .., y_i, .. \rangle$ then `Cov(a1,a2)` is the real :

$$1/n \sum_{i=1,n} x_i y_i - \overline{x}\,\overline{y}$$

If `a1 or a2` contain `Undef` elements, these elements are considered as `Null` values.

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `EDist`, `Angle`, `Norm`, `VProd`, `Plus`, `Times`, `Var`, `Mean`.

EXAMPLES

```
AML> Cov([6.5,4.5,4.9,7.2,0.6,4.5],[7.1,3.9,9.5,18.4,6.7,6.7])
     <INT> 5.18167
```

Date operators

+ , -, Second, Minute, Hour, Day, DateUnit, DateFormat, Date, ToTimeUnit.

USAGE

```
d1 + nb
d1 - nb
d1 - d2
Date(stg)
ToTimeUnit(i)
```

ARGUMENTS

d1, d2 (DATE)
nb, i (INT)
stg
Second, Minute, Hour, Day (INT): definition of classical date units in terms of integers.
DateUnit (INT): value of the current date unit.
DateFormat (STRING): STRING indicating the format of dates currently used in the system.

RETURNED OBJECT

The addition/substraction of a date and an integer returns a DATE. The new date is the old date incremented/decremented by nb times DateUnit.
The difference between two dates returns an INT corresponding to the number of DateUnit between these two dates. This difference can be a negative integer.
Second = 1, Minute = 60, Hour = 3600, Day = 86400 are global constant values.
Global variables and their default values: DateFormat = "DDMMYY", DateUnit = 86400. These values are used as parameters by other date functions.
The value returned by Date() is an INT corresponding to the integer encoding of a date defined as a STRING. If stg is Undef, returns Undef.
The value returned by ToTimeUnit is the conversion of number i into a corresponding number of DateUnits. If i is Undef, returns Undef.

DESCRIPTION

DateUnit is given the default value Day. It may be changed to other values, for example DateUnit = Hour. Once a DateUnit is fixed, other date functions can be used. The DateUnit is implicitly used as a global variable by the other functions when necessary (e.g. in incrementation of dates, or integer conversions to dates).
Function Date() converts a STRING into a date. It uses the current value of DateFormat to encode strings as DATE objects. Valid values of DateFormat are strings: "DDMM", "DDMMYY", "MMYY", "DDMMTIME", "DDMMYYTIME". A date STRING with format DDMMYY is a STRING like "09/07/97", where '/' is a mandatory separator between date fields. A time STRING is a series of 1 to 3 integers separated by a ':'. Valid time strings are "12:35:01", "12:35:", "12:35" or "12". Whenever time information is used, e.g. DateFormat = DDMMTIME, valid time

strings have to be concatenated to date strings, separated by a white space: e.g. "09/07 12:35:01".

NOTE

Dates have to be greater than 01/01/1901.

SEE ALSO

MTG module: `DateSample`, `FirstDefinedFeature`, `LastDefinedFeature`, `NextDate`, `PreviousDate`

EXAMPLES

```
AML> DateUnit
     <INT>86400
AML> DateFormat
     <STRING>DDMMTIME
AML> d1 = Date(”10/0912:33”)
     <DATE> 10/09 12:33
AML> DateFormat
     <STRING>DDMMYY
AML> d1 = Date(”10/09/97”)
     <DATE> 10/09/97
AML> # Adding 10 days to d1
AML> d2 = d1 + 10
     <DATE> 20/09/97
AML> d2 - d1
     <INT>10
AML> DateUnit = Minute
     <INT>60
AML> ToTimeUnit(420)
     <INT>7
```

Delete

Delete an object from the current set of AML objects.

USAGE

```
Delete(name1, name2, ...)
```

ARGUMENTS

name1, name2,... (STRING) : names of the object to be deleted.

RETURNED OBJECT

No value is returned.

DESCRIPTION

The objects passed as arguments are deleted from the set of AML objects. This means that the name of the corresponding variables is no longer used and that the corresponding memory zone is freed.

NOTE

Since the a AML variable is designated by its name, this name is always evaluated to the corresponding AML object and not to a STRING. Therefore, to apply Delete correctly to a variable whose name is var1 for instance, one has to explicitly write Delete("var1") or more simply Delete("var1") (the quote character before a STRING name suppress the evaluation of the STRING).

SEE ALSO

EXAMPLES

```
AML> I = 9; j="Hello"; var1=[1,2,3]
     <ARRAY(INT)> [1,2,3]
AML> ?
      I      INT          :9
      j      STRING       :Hello
      var1   ARRAY(INT)   [1,2,3]
AML> Delete('i,'var1)
AML> AML>?
      j      STRING       Hello
```

Display

Display an ASCII representation of an object.

USAGE

```
Display(obj)
Display(obj, Detail→ 2)
```

ARGUMENTS

`obj` (`T`) : the object can have any type.

OPTIONAL ARGUMENTS

`ByteSize` (`INT`). This option can be used for any object to change the size of other optional arguments can exist depending on the object.
e.g. `Detail` (`INT`) : level of detail at which the argument is being displayed.

RETURNED OBJECT

No value is returned.

DESCRIPTION

The type of ASCII description depends on the particular AML object used. Some objects provides ASCII representations at several levels of detail. Refer to object constructors for precise display of a particular display.

SEE ALSO

?, ??

EXAMPLES

```
AML> # example of a Sequences object
AML> seq1 = Sequences("datafile.seq")  # this command builds a
                  Sequences object from a list of ascii sequences
AML> Display(seq1, ViewPoint->Data)    # displays the raw data
                                    corresponding to the sequence
```

DisplayAllNames, DisplayAllUserNames

Display user or system variables and functions, ?, ??

USAGE

```
?
? varname
??
```

ARGUMENTS

varname (STRING) : name of a user variable

RETURNED OBJECT

This function returns no value.

DESCRIPTION

Command ? prints the content of an object identified by its name to the screen. If no name is given, the command prints the list of user variables. To print all the functions and variables defined in the system, use the command ??.

NOTE

When using command ??, variables, constants of functions defined in the system are displayed with ':' as a prefix character. The names are sorted in alphabetical order.

SEE ALSO

Display.

EXAMPLES

```
AML> ?
     No user object defined.
AML> i=1
     <INT>1
AML> ?
     i <INT> : 1
AML> f(_x)=_x+1
     f <FUNC> : Function
AML> ?
     f <FUNC> : Function
     i <INT> : 1
AML> AML>?i
     i <INT> : 1
```

Echo

Displays messages on the screen

USAGE

```
Echo(arg1, arg2, …, argi,…, argn)
```

ARGUMENTS

`argi` (`ANY`) : the *i*th arg (*i*=1…*n*) can be of any type. The number of argument *n* is arbitrary, provided it is greater than or equal to 1.

OPTIONAL ARGUMENTS

`ByteSize` (`INT`) : This optional argument must be used when the size of the string representing an object in the list of arguments exceeds 10000 bytes (you are warned by a message whenever this occurs).

RETURNED OBJECT

`STRING`

DESCRIPTION

This function can be used to build messages during some AML function computation. This is helpful while debugging AML functions to store and display intermediate results.

SEE ALSO

`Display, EchoOn, EchoOff.`

EXAMPLES

```
AML> a = [1,2,3]
AML> Echo("array a = ", a, "has a size = ", Size(_a))

AML> f(_x) = Switch _x \
AML> Case 1: Green
AML> Case 2: Red
AML> Case 3: Yellow
AML> Default: (m=Echo("Color not found for value ", _x);Black)

AML> f(2)
<INT> 3    # code for Red
AML> m
<STRING> m

AML> f(7)
<INT> 0    # code for Black
AML> m
<STRING> "Color not found for value 7"
```

EchoOn / EchoOff

Put echo on during batch reading of a file

USAGE

```
EchoOn()
EchoOff()
```

ARGUMENTS

No argument.

RETURNED OBJECT

No returned value.

DESCRIPTION

When a file is run in batch, using: `aml -i filename`

DESCRIPTION

No output is made of the computations made within the file. It is possible to change this by writing in the file "filename" the command `EchoOn()`. As soon as it is read, aml outputs the results of subsequent computation on the standard output. The effect of this command can be cancelled by the command `EchoOff()`.

NOTE

These functions can be used for debugging a batch file as follows: in order to trace the different steps of a program, the user can call `EchoOn()` and insert debugging strings in the file. Since `STRING`s objects are evaluated as strings of characters, the resulting `STRING` will be output to the standard output as soon as a `STRING` object is encountered in the batch file. When a particular debugging `STRING` is output to the screen, the user thus knows that a particular step of computation has been reached.

SEE ALSO

`Display`

EXAMPLES

```
# example of a batch file
EchoOn()
# string object which will be echoed to the screen,
# indicating to the user which step of computation is being
computed
"step1"
1+4
"step2" # another debugging string
+2
EchoOff()
Sqrt(10)
```

EDist

Euclidean distance between two points represented by two arrays.

USAGE

```
EDist(array)
```

ARGUMENTS

`array` (`ARRAY(REAL)`). These two arrays must have identical size.

RETURNED OBJECT

The distance between two array is a `REAL`. If one of the arguments is `Undef`, returns `Undef`

DESCRIPTION

Denote $a_1 = [x_1, x_2, \ldots x_n]$, $a_2 = [y_1, y_2, \ldots y_n]$. The norm of arrays `array` is defined by:

$$s = \sqrt{\sum_{k=1}^{n} (y_k - x_k)^2}$$

SEE ALSO

`Norm`, `Angle`, `SProd`, `VProd`, `Array`.

EXAMPLES

```
AML> EDist([1.,0.,0.],[1.,0.,10.])
     <REAL> 10
AML> EDist([1.,0.],[0.,1.])
     <REAL> 1.41421
```

Filter

Applies a linear filter to an array, which outputs an array made of locally weighted mean values of the initial array.

USAGE

```
Filter(array, filt)
```

ARGUMENTS

`array` (`ARRAY(T)`). `T` is either `INT` or `REAL`. `array` is the array to be filtered.
`filt` (`ARRAY(T')`). `T'` is either `INT` or `REAL`. `filt` is the array used to filter `array`. `filt` must have an odd number of elements.

RETURNED OBJECT

The function returns an element of type `ARRAY(REAL)`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

This function is used to apply a linear filter to an array of numeric values. This can be used for instance to smooth a series of values by replacing each value of the initial array `array` by a mean of its neibourgh values. Let us denote `array`=$[x_1, x_2, \ldots x_n]$ and `filt`=$[a_1, a_2, \ldots a_n]$. Since k is an odd number, let us denote l [illegible]2. Then the function `Filter` returns an array whose ith element is given by:

$$\frac{\sum_{j=-l}^{l} a_{l+j+1} . x_{i+j} . \delta_{(i+j)\in[1,n]}}{\sum_{j=-l}^{l} a_{l+j+1} . \delta_{(i+j)\in[1,n]}}$$

with the convention that then $\delta_{i+j\in[1,n]} = 0$ if i [illegible] $[1,n]$ and 1 otherwise. In this case the sum at the denominator is limited to j such that i [illegible] $[1,n]$. If for some i, the value of this sum happens to be null, the division is not made and the result is simply $\sum_{j=-l}^{l} a_{l+j+1} . x_{i+j} . \delta_{(i+j)\in[1,n]}$.

SEE ALSO

`Array, List, Pos, At, Head, MovingAverage.`

EXAMPLES

```
AML> Filter([1,5,2,1,-7,1,1,1,1,9,1,1,1,1],[1,1,1,1,1]) \
AML> # smoothes a series of values
     <ARRAY(INT)> [2.67,2.25,0.4,0.4,-0.4,-0.6,
     -.6,2.6,2.6,2.6,2.6,2.6,1,1]
AML> Filter([1,5,2,1,-7,1,1,1,1,9,1,1,1,1],[1,-2,1]) \
AML> # emphasizes the deviations
     <SET(INT)> [-3,-7,2,-7,16,-8,0,0,8,-16,8,0,0,1]
```

Flatten

Flattens the structure of nested arrays or sets

USAGE

```
Flatten(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `SET(T)`). `T` is any type.

RETURNED OBJECT

If `array` has type `ARRAY(ARRAY(T))`, the function returns an element of type `ARRAY(T)`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

This function is used to flatten the structure of nested arrays or sets. Let us assmue that `array` is a nested array: `array` has the form $[x_1, x_2, \ldots x_n]$, where x_i is an `ARRAY`: $x_i = [y_1^i, y_2^i, \ldots, y_{kn}^i]$.

The array `array` can thus be written $[[y_1^1, y_2^1, \ldots, y_{k1}^1], [y_1^2, y_2^2, \ldots, y_{k2}^2], \ldots, [y_1^n, y_2^n, \ldots, y_{kn}^n]]$. Then, the flattened array obtained from `array` is the `ARRAY`:
$[y_1^1, y_2^1, \ldots, y_{k1}^1, y_1^2, y_2^2, \ldots, y_{k2}^2, \ldots y_1^n, y_2^n, \ldots, y_{kn}^n]$

NOTE

This function can be applied to any nesting combination of `ARRAY`s and `SET`s (*cf.* example)

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Head`.

EXAMPLES

```
AML>  Flatten([[10,11],[12],[],[13,14,15]])
      <ARRAY(INT)> [10,11,12,13,14,15]
AML>  Flatten(Set([10,11],[12],[],[11,11,12])
      <SET(INT)> [10,11,12]
```

Floor

Integer no greater than.

USAGE

```
Floor(x)
```

ARGUMENTS

x (INT or REAL) : a numerical value

RETURNED OBJECT

The value returned by Floor has the same type as x. If x is Undef, returns Undef.

DESCRIPTION

Returns the integer no greater than x.

NOTE

This function is similar to the corresponding double function of the host system (Unix, ...).

SEE ALSO

Abs, Ceil, Truncate, Rint, ToInt, ToReal.

EXAMPLES

```
AML> Floor(3)
     <INT>3
AML> Floor(3.1)
     <REAL>3
AML> Floor(-3)
     <INT>-3
AML> Floor(-2.9)
     <REAL>-3
```

Foreach

Iterator.

USAGE

```
Foreach _x In set : apply(_x)
```

ARGUMENTS

`_x` (tied variable). This variable denotes one element of the set object `set`.
`set` (`ARRAY(T)` or `SET(T)`). `T` can be any type.
`apply()` (`FUNC`). This is a function which is used by the iterator.

RETURNED OBJECT

If `set` has type `ARRAY(T)`, the iterator returns an `ARRAY`. If `set` has type `SET(T)`, the iterator returns a `SET`.

DESCRIPTION

The iterator allows us to browse a set object and to apply to each object `_x` of the set a given function, `apply`. This is a very generic way of exploring data bases and computing samples of any things.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Append`.

EXAMPLES

```
AML> Foreach _x In [1:5] : _x+1
     <ARRAY(INT)> [2,3,4,5,6]
AML> Foreach _x In [1:10] : Select(_x, _x/2==0)
     <ARRAY(INT)> [2,4,6,8,10]
AML> Foreach _x In [1:10] : Select(_x+1,_x/2==0)
     <ARRAY(INT)> [3,5,7,9,11]
```

Head

First element of an array or a list.

USAGE

```
Head(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `T`. If array has type $\texttt{LIST(T}_1\texttt{,...)}$, the function returns an element of type $\texttt{T}_1$. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the first element of an array or a list. Ihis is equivalent to the expression `array@1`.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Tail`.

EXAMPLES

```
AML> Head([10,11,12,13,14,15])
     <INT> 10
AML> Head([[1,2],[3,4,5],Undef,[7],[8,9,10]])
     <ARRAY(INT)>[1,2]
AML> Head(List([3,4,5],"alpha",5.9))
     <ARRAY(INT)>[3,4,5]
```

Identity

Function identity.

USAGE

```
Identity(x)
```

ARGUMENTS

x (ANY) : the argument can have any type

RETURNED OBJECT

Same type as the argument x.

DESCRIPTION

The identity function simply returns its argument, unchanged.

SEE ALSO

Definition of AML functions.

If-Then-Else

Conditional expression.

USAGE

```
If Then else
```

ARGUMENTS

pred (BOOL). Boolean expression which expresses the condition
e1 (T_1). T_1 can be any type.
e2 (T_2). T_2 can be any type.

RETURNED OBJECT

Depending on the pred value, the object returned has the same type as either e1 or e2.

DESCRIPTION

This expression returns e1 if pred is True. It returns e2 otherwise.

SIDE EFFECT

Important: in the first version of AMAPmod, the evaluation procedure of expressions evaluates all subexpressions before evaluating a given expression. Thus, in every cases, both e1 and e2 are evaluated first. Then depending on the value of pred the evaluation of the If Then Else expression leads to select either e1 or e2. This means that some time may be lost in computing useless expressions. This behavior will be corrected in version 2.

SEE ALSO

Array, List, Pos, At, Append.

EXAMPLES

```
AML> A = 3
     <INT>
AML> If a == 3 Then "Yes" Else "No"
     <STRING>Yes
AML> f(_x) = If _x == 3 Then "Yes" Else _x-1
     <FUNC> Function
AML> f(2)
     <INT> 1
AML> f(3)
     <STRING> Yes
```

InsertAt

Inserts an element at a given position in an array or a list.

USAGE

```
InsertAt(array, elem, i)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type
`elem` (`T`). Element to be inserted. If `array` is of type `ARRAY(T)`, `elem` must be of type `T`.
`i` (`INT`). Position at which the element must be inserted. It can have either a positive or a negative value, such that $0 < $ `Abs(i)` $\leq$ `Size(array)`.

RETURNED OBJECT

If `array` is of type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`. If `array` has type $\texttt{LIST}(T_1, T_2, \ldots, T_N)$, the function returns an element of type $\texttt{LIST}(T_1, T_2, \ldots, T_{I-1}, T, T_I \ldots, T_N)$.

DESCRIPTION

Inserts an element at the *i*th position in an array or a list, *i.e.* after the insertion, element `elem` is at position `i` in `array`. If `i` has a negative value, the function inserts the element at position `i` with respect to the end of `array`, *i.e.* if *n* is the size of `array`, it corresponds to position *n*– `i` + 1 of `array`.

SIDE EFFECT

Important: contrary to most functions in AML which leave their argument unchanged, this fonction modifies its `array` argument. The value returned is the modified argument.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Append`.

EXAMPLES

```
AML> a = [10,11,12,13,14,15]
     <ARRAY(INT)>[10,11,12,13,14,15]
AML> InsertAt(a,9,1)
     <ARRAY(INT)>[9,10,11,12,13,14,15]
AML> a
     <ARRAY(INT)>[9,10,11,12,13,14,15]
AML> InsertAt([10,11,12,13,14,15],9,-1)
     <ARRAY(INT)>[10,11,12,13,14,15,9]
AML> InsertAt([10,11,12,13,14,15],9,5)
     <ARRAY(INT)>[10,11,12,13,9,14,15]
AML> InsertAt([10,11,12,13,14,15],9,-5)
     <ARRAY(INT)>[10,11,9,12,13,14,15]
```

```
AML>  InsertAt(List(”alpha”,5,True),[1,2,3],3)
      <LIST(STRING,INT,ARRAY(INT),BOOL> [alpha,5,[1,2,3],True]
```

Inter

Intersection of to set objects, &

USAGE

```
Inter(set1, set2)
set1 & set2
```

ARGUMENTS

set1, set2 (ARRAY or SET) : these two arguments must have the same type

RETURNED OBJECT

Inter returns a set object of the same type as its two arguments, *i.e.* either ARRAY or SET. If either set1 or set2 is Undef, returns Undef.

DESCRIPTION

Inter(set1, set2) returns the set object made up by the objects that both belong to set1 and to set2. If set1 and set2 are ARRAYs, Inter() preserves the order of object from set2.

NOTE

Inter can also be invoked with the minus sign &.

SEE ALSO

Union, ToSet, ToArray, Sort

EXAMPLES

```
AML> Inter( [1,2,3,4,5,6,7],[0,2,3,4,10])
     <ARRAY(ANY)>[2,3,4]
AML> Inter_set = [1,2,3,4,5,6,7] & [0,3,2,4,10]
     <ARRAY(ANY)>[3,2,4]
AML> Set (1,2,3,4,5,6,7) & Set(0,3,2,4,10)
     <set(ANY)>[2,3,4]
```

Invert

Inverts the order of the elements of an array or a list.

USAGE

```
Invert(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`. If `array` has type $\texttt{LIST}(T_1, T_2, \ldots, T_N)$, the function returns an element of type $\texttt{LIST}(T_N, \ldots, T_2, T_1)$. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the an array or a list made of all the element of `array` in reverse order.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Head`, `Reverse`.

EXAMPLES

```
AML> Invert([10,11,12,13,14,15])
     <ARRAY(INT)> [15,14,13,12,11,10]
AML> Invert([[1,2],[3,4,5],Undef,[7],[8,9,10]])
     <ARRAY(ARRAY(INT))>[[8,9,10],[7],Undef,[3,4,5],[1,2]]
```

List

List constructor

USAGE

```
List(x1,x2,...,xn)
```

ARGUMENTS

x1,x2,...,xn (T_1,T_2,...,T_N) : Type T_K, *i* ❍*k* ❍*n*, can be any type. Any xk can be Undef.

RETURNED OBJECT

If the type of the arguments xk is T_K, the value returned by a set constructor is an LIST(T_1,T_2,...,T_N).

DESCRIPTION

A LIST is a set-type, *i.e.* the type of a collection of objects. Contrary to ARRAYs and SETs, LIST collections are heterogenous collection of objects. Elements of a LIST can have the different types. The order of the elements of a LIST is relevant.

DETAILS

- In version 1.x of AMAPmod there is no way of saving or loading binary or ASCII representations of lists.
- There is no possibility to Plot LISTs.
- Two LISTs are equal if they have the same elements, in the same order.

SEE ALSO

Size, Sum, Array, Set, ToArray, ToSet, ToList, At, Pos, AllPos, Head, Tail, InsertAt, RemoveAt, Invert.

EXAMPLES

```
AML>  l = List("A",7)
      <LIST(STRING,INT)> [A,7]
AML>  l = List("A",Undef,7)
      <LIST(STRING,UNDEF,int)> [A,Undef,7]
AML>  l@3
      <INT> 7
AML>  l = List(Undef,Undef,Undef)
      <LIST(UNDEF,UNDEF,UNDEF,UNDEF)> [Undef,Undef,Undef]
AML>  l = List("A",7,List([1,2],True))
      <LIST(STRING,INT,LIST(ARRAY(INT),BOOL))>
      [A,7,[[1,2],TRUE]]
```

Mathematical functions

Sqrt, Power (^), Log, Log10, Exp

USAGE

```
Sqrt(x)
Exp(x)
a ^ y
```

ARGUMENTS

`x, a, y` (`INT` or `REAL`)
`Sqrt` (square root) takes a non-negative argument.
`a ^ y` is not defined for `a` < 0 and `y` not an integer.
Arguments of Log and Log10 must be positive.

RETURNED OBJECT

The returned values are `REAL` values. If the above conditions on the argument are not met, `Undef` is returned. If either argument is `Undef`, returns `Undef`.

DESCRIPTION

These functions are similar to the corresponding double function of the host system (Unix, ...). Type "man math" on the host system for more details.

SEE ALSO

`Trigonometric functions`.

EXAMPLES

```
AML> Sqrt(81)
     <REAL> 9
AML> (-34)^5
     <REAL> -4.54354e+07
AML> (-34)^-5
     <REAL> -2.20093e-08
```

Max

Computes the maximum value of the elements of an array or a set.

USAGE

```
Max(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `SET(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)` or `SET(T)`, the function returns an element of type `T`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the element whose value is maximum over the set of elements of `array`. If `array` contains `Undef` elements, these elements are discarded.

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `Sum`, `Min`.

EXAMPLES

```
AML> Max([10,11,12,13,14,15])
     <INT> 15
AML> Max([2,2,Undef,7,1])
     <INT> 7
AML> Max(Foreach _x In[0:Pi:Pi/100] : Cos(_x))
     <REAL> 1
```

Mean

Computes the mean value of elements of an array.

USAGE

`Mean(array)`

ARGUMENTS

`array` (`ARRAY(T)` or `SET(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)` or `SET(T)`, the function returns an element of type `REAL`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the mean value of the elements of `array`. If `array` = $\leftarrow x_1, y_1, z_1 \rightarrow$ then `Mean(array)` is:

$$1/n \sum_{i=1,n} x_i$$

If `array` contains `Undef` elements, these elements are considered as `Null` value.

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `EDist`, `Angle`, `Norm`, `VProd`, `Array,Plus,Times`.

EXAMPLES

```
AML>  Mean([10,11,12,13,14,15])
      <INT> 12.5
AML>  Mean([10,11,12,13,14,Undef,15])
      <INT> 10.7143
```

`Min`

Computes the minimum value of the elements of an array or a set.

USAGE

```
Min(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `SET(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)` or `SET(T)`, the function returns an element of type `T`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the element whose value is minimum over the set of elements of `array`. If `array` contains `Undef` elements, these elements are discarded.

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `Sum`, `Max`.

EXAMPLES

```
AML> Min([10,11,12,13,14,15])
     <INT> 10
AML> Min([2,2,Undef,7,1])
     <INT> 1
AML> Min(Foreach _x In [0:Pi:Pi/100] : Cos(_x))
     <REAL> 0
```

Minus

Difference of two arrays.

USAGE

```
Minus(a1, a2)
```

ARGUMENTS

a1, a2 (ARRAY(REAL)). These two arrays must have the same dimension.

RETURNED OBJECT

The result is an ARRAY(REAL) with the same dimension as a1 and a2. If one of the arguments is Undef, returns Undef

DESCRIPTION

The function returns the vectorial difference of two arrays of dimension *n*. If a1= $\leftarrow x_1, y_1, z_1 \rightarrow$ and a2= $\leftarrow x_2, y_2, z_2 \rightarrow$ then a= $\leftarrow x, y, z \rightarrow$ = Plus(a1, a2) = $\leftarrow x_1 \not\leftarrow x_2, y_1 \not\leftarrow y_2, z_1 \not\leftarrow z_2 \rightarrow$

NOTE

There is no limit on the dimensions of the arguments.

SEE ALSO

EDist, Angle, Norm, VProd, Array, Plus, Times.

EXAMPLES

```
AML> Minus([1.2,0.,0.,8.5],[0.2,1.,0.,-9.2])
     <ARRAY(REAL)> [1.0,-1.,0.,17.7]
```

Mod

Modulo function

USAGE

```
x Mod y
```

ARGUMENTS

x, y (INT) : integral value

RETURNED OBJECT

The value returned by Mod is an INT. If either x or y is Undef, returns Undef.

DESCRIPTION

Returns the remainder of the integral division of x by y.

NOTE

This function is similar to the corresponding double function of the host system (Unix, ...).

SEE ALSO

/

EXAMPLES

```
AML>  23Mod3
      <INT>2
AML>  -23Mod3
      <REAL> -2
AML>  23Mod-3
      <REAL> 2
```

Norm

Norm of an arrays.

USAGE

```
Norm(array)
```

ARGUMENTS

`array` (`ARRAY(REAL)`). These two arrays must have identical size.

RETURNED OBJECT

The norm of an array is a `REAL`. If one of the arguments is `Undef`, returns `Undef`

DESCRIPTION

Denote `array`=$[x_1, x_2, \ldots x_n]$. The norm of arrays `array` is defined by:

$$s = \sqrt{\sum_{k=1}^{n} x_k^2}$$

SEE ALSO

`EDist`, `Angle`, `SProd`, `VProd`, `Array`.

EXAMPLES

```
AML>  Norm([1.,0.,0.])
      <REAL> 1
AML>  Norm([1.,3.,0.,.4])
      <REAL> 3.18748
```

Plot, NewPlot

Plot an object to the screen.

USAGE

```
Plot(obj)
Plot(obj, Window-> w)
NewPlot(obj)
```

ARGUMENTS

`obj` (`T`) : the object can have any type.

OPTIONAL ARGUMENTS

Optional arguments can exist depending on the type of the first argument.
`Window` (`WINDOW`) : enables the user to redirect a plot to a given graphic window. See function `NewPlot` for a description of how to create new graphic windows.

RETURNED OBJECT

`Plot` returns no value. `NewPlot` returns a new `WINDOW` object.

DESCRIPTION

Many AML object have a graphical representation. `Plot` displays these graphic representations to the screen. whenever they exist. See the object constructors for more details and a list of optional arguments.

SEE ALSO

`Load`, `Object constructors`, `Display`, `Save`.

EXAMPLES

```
AML> # Plotting an array
AML> a1 = [10,12,3,14,10,6]     # builds a simple array.
AML> Plot(a1)                   # Plot the array
AML> a2=[1,10,-4,-1,0,2]        # builds a second array.
AML> w = NewPlot(a2)            # Plot the array in another
                                # window
AML> a3=[100,109,154,125,156,203] # builds a third array.
AML> Plot(a3, Window->w)        # Plot the array in the second
                                # window and replaces the plot
                                # of a2
AML> # Plotting a MTG
AML> g = MTG("myplant.mtg")     # builds a MTG.
AML> F = PlantFrame(VtxList(Scale->@1) #Builds a geometric
                                # interpretation of the first
                                # branching system of g
AML> Plot(f)                    # gives a default visual
                                # interpretation of the MTG
```

Plus

Addition of two arrays.

USAGE

```
Plus(a1, a2)
```

ARGUMENTS

a1, a2 (ARRAY(REAL)). These two arrays must have the same dimension.

RETURNED OBJECT

The result is an ARRAY(REAL) with the same dimension as a1 and a2. If one of the arguments is Undef, returns Undef

DESCRIPTION

The function returns the vectorial sum of two arrays of dimension *n*. If a1= $\leftarrow x_1, y_1, z_1 \rightarrow$ and a2= $\leftarrow x_2, y_2, z_2 \rightarrow$ then a= $\leftarrow x, y, z \rightarrow$ = Plus(a1, a2) = $\leftarrow x_1 \; x_2, y_1 \; y_2, z_1 \; z_2 \rightarrow$

NOTE

There is no limit on the dimensions of the arguments.

SEE ALSO

EDist, Angle, Norm, VProd, Array.

EXAMPLES

```
AML> Plus([1.2,0.,0.,8.5],[0.2,1.,0.,-9.2])
     <ARRAY(REAL)> [1.4,1.,0.,-0.7]
```

Pos

Position of an element in an array.

USAGE

```
Pos(array, element)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type. If the argument is `Undef`, returns `Undef`
`element` (`T`). Element that is searched for in `array`.

RETURNED OBJECT

The function returns an integer (`INT`) if `element` is in `array`. It returns `Undef` if not. If the argument is `Undef`, returns `Undef`

DESCRIPTION

`Pos` returns the position of the first occurence of `element` in `array`.

SEE ALSO

`Array`, `List`, `Tail`, `At`, `Head`.

EXAMPLES

```
AML> Pos([10,11,12,13,14,15],14)
     <INT>5
AML> Pos([[1,2],[3,4,5],Undef,[7],[8,9,10]],[3,4,5])
     <INT>2
AML> Pos([1,2,Undef,2],Undef)
     <INT>3
AML> Pos([1,2,3],7)
     <UNDEF>Undef
AML> Pos(List("alpha",True,[1,2,3],7)),True)
     <INT>2
```

Prod

Computes the product of the elements of an array or a set.

USAGE

```
Prod(array)
```

ARGUMENTS

`array` (`ARRAY(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `T`, with the same number of elements. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the product of the elements of `array`. Let `array=`$[x_1, x_2, \ldots x_n]$, then the result is $\bigotimes_{k=1}^{n} x_k$.

If `array` contains `Undef` elements, these elements are considered as having value 1 (they do not affect the overall product of other elements).

SEE ALSO

`Array`, `Set`, `Series`, `Size`.

EXAMPLES

```
AML>  Prod([1,2,3,4])
      <INT> 24
AML>  Prod([10,11,12,Undef,14,15])
      <INT> 277200
```

ProdSeries

Computes the series of incremental products associated with a given sequence of numerical values.

USAGE

```
ProdSeries(array)
```

ARGUMENTS

`array` (`ARRAY(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`, with the same number of elements. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the mathematical series associated with the sequence of numerical values defined in `array`. Let `array`=$[x_1, x_2, \ldots x_n]$, then the result is the array whose *i*th element is defined by $\prod_{k=1}^{i} x_k$. If `array` contains `Undef` elements, these elements are considered as having value 1 (they do not affect the overall product of other elements).

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `Prod`, `Sum`.

EXAMPLES

```
AML> ProdSeries([1,2,3,4])
     <ARRAY(INT)> [1,2,6,24]
AML> ProdSeries([10,11,12,Undef,14,15])
     <ARRAY(INT)> [10,110,1320,1320,18480,277200]
```

Rand

Computes a random value.

USAGE

```
Rand()
Rand(i,j)
```

ARGUMENTS

`i,j` (`INT`). `j` must be greater than `i`.

RETURNED OBJECT

If no argument is given, `rand` returns a `REAL` between 0 and 1 (following a uniform distribution). If two arguments are given, `rand` returns an `INT` at random between `i` and `j`

DESCRIPTION

Based on Linux function Random().

SEE ALSO

STAT module.

EXAMPLES

```
AML> Rand()
     <REAL> 0.45675
AML> Rand(22,35)
     <REAL> 28
     # Rand is evaluated only once in the following iteration:
AML> Foreach _i In [1,5]:Rand(22,35)
     <REAL> [34,34,34,34,34]
     # The following trick is used to have Rand evaluated for
     # each _i:
AML> Foreach _i In [1,5]:Rand(22,35+_i-_i)
     <REAL> [29,29,31,22,25]
```

RemoveAt

Removes an element at a given position in an array or a list.

USAGE

```
RemoveAt(array, i)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type
`i` (`INT`). Position at which the element must be removed. It can have either a positive or a negative value. The argument, $0 < \texttt{Abs(i)} \leq \texttt{Size(array)}$.

RETURNED OBJECT

If `array` is of type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`. If `array` has type `LIST`, the function returns an element of type $(T_1, T_2, \ldots, T_{I-1}, T_{I+1}, \ldots, T_N)$.

DESCRIPTION

Removes the element at the *i*th position in an array or a list. If `i` has a negative value, the function removes the element at position `i` with respect to the end of `array`, *i.e.* if *n* is the size of `array`, it corresponds to position $n - \texttt{i} + 1$ of `array`.

SIDE EFFECT

Important: contrary to most functions in AML which leave their argument unchanged, this fonction modifies its `array` argument. The value returned is the modified argument.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Append`, `InsertAt`.

EXAMPLES

```
AML>  a=[10,11,12,13,14,15]
      <ARRAY(INT)>[10,12,13,14,15]
AML>  RemoveAt(a,2)
      <ARRAY(INT)>[10,12,13,14,15]
AML>  a
      <ARRAY(INT)>[10,12,13,14,15]
AML>  RemoveAt([[1,2,3],[10,15]],-1)
      <ARRAY(ARRAY((INT))> [[1,2,3]]
AML>  RemoveAt(List("alpha",[1,2,3],5,True),3)
      <LIST(STRING,ARRAY(INT),BOOL> [alpha,[1,2,3],True]
```

Rint

Round to nearest integer.

USAGE

```
Rint(x)
```

ARGUMENTS

x (INT or REAL) : a numerical value

RETURNED OBJECT

The value returned by Rint has the same type as x. If x is Undef, returns Undef.

DESCRIPTION

Returns the integer nearest to x.

NOTE

This function is similar to the corresponding double function of the host system (Unix, ...).

SEE ALSO

Abs, Ceil, Truncate, Floor, ToInt, ToReal.

EXAMPLES

```
AML> Rint(3)
     <INT>3
AML> Rint(3.4)
     <REAL>3
AML> Rint(3.5)
     <REAL>4
AML> Rint(3.6)
     <REAL>4
AML> Rint(-2.9)
     <REAL>-3
AML> Rint(-2.5)
     <REAL>-2
AML> Rint(-2.4)
     <REAL>-2
```

Save

Save an object to a file in different formats.

USAGE

```
Save(obj, filename)
Save(obj, file, Format-> Binary)
```

ARGUMENTS

`obj` (`T`) : the object can have any type.
`filename` (`STRING`): identifies the file in which the object `obj` is saved.

OPTIONAL ARGUMENTS

Optional arguments can exist depending on the type of the first argument. By default an ASCII form of the object is saved. This default can be changed using the optional argument `Format` (`STRING`) which specifies the format to be used (ASCII, BINARY). In version 1.x of AMAPmod, not all objects have a binary format option. Objects that cannot be saved in BINARY need to be recomputed each time AML is launched. This will be corrected in version 2.

RETURNED OBJECT

No value is returned.

DESCRIPTION

Save an object to a file in different formats.

SEE ALSO

`Load`, `Object` constructors, `Display`.

EXAMPLES

```
AML> #Savinganarray
AML> a1 = [1,2,3,4,5,6]          # builds a simple array.
AML> Save(a1, "file1.dat")       # Save the array to file named
                                 # file1.dat
```

Select

Conditional identity function

USAGE

```
Select(exp, pred)
```

ARGUMENTS

`exp` (`ANY`): argument that can be selected. This argument can have thevalue `Undef`
`pred` (`BOOL`): argument that enables/disable the selection of the value ofthe first argument

RETURNED OBJECT

`Select` returns `exp` if `pred` is `True`, `VOID` if `pred` is `False`. If `exp` is `Undef`, returns `Undef` if `pred` is `True` and returns no value otherwise. If `pred` is `Undef`, returns `Undef`.

DESCRIPTION

`Select` is the identity function if its second argument is `True`, and returns no value (`VOID`) if `False`. If pred has value `Undef`, `Select` returns `Undef` whatever the value of `exp`.

EXAMPLES

```
AML>  Select(3,True)
      <INT>3
AML>  Select(3,False)
      <VOID>
AML>  Select(Undef,True)
      <UNDEF> Undef
AML>  Select(Undef,False)
      <VOID>
AML>  #Filtering an array of values
AML>  Foreach _i In [1,2,3,2,2,4,1,1] : Select(_i,_i!=2)
      <ARRAY(INT)>[1,3,4,1,1]
```

Series

Computes the mathematical series associated with a given sequence of numerical values.

USAGE

```
Series(array)
```

ARGUMENTS

`array` (`ARRAY(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`, with the same number of elements. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the mathematical series associated with the sequence of numerical values defined in `array`. Let `array`=$[x_1, x_2, \ldots x_n]$, then the result is the array whose *i*th element is defined by $\sum_{k=1}^{i} x_k$. If `array` contains `Undef` elements, these elements are considered as having `Null` value.

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `Prod`, `Cumulate`.

EXAMPLES

```
AML>  Series([1,2,3,4])
      <ARRAY(INT)> [1,3,6,10]
AML>  Series([10,11,12,Undef,14,15])
      <ARRAY(INT)> [10,21,33,33,47,62]
```

Set

SET constructor

USAGE

```
Set(x1,x2,...,xn)
```

ARGUMENTS

x1,x2,...,xn (T) : values of identical type. Type T can be any type. Any xk can be Undef.

RETURNED OBJECT

If the type of the arguments xk is T, the value returned by a set constructor is an SET(T) if at least one xk is not Undef. If all xk are Undef the constructor returns Undef.

DESCRIPTION

A SET is a set-type, *i.e.* the type of a collection of objects. Like for ARRAYs, all the elements of a SET must have the same type. However, a SET can contain Undef elements. Contrary to ARRAYs, elements of a SET are not ordered. It is thus meaningless to speak of the *i*th element of a SET. Consequently, each element of a set appears only once in the SET. There are no duplicates of elements.

DETAILS

– In version 1.x of AMAPmod there is no way of saving or loading binary or ASCII representations of sets. For ASCII representations, one may want to save a SET as an ARRAY:

```
AML> Save(ToArray(set1),"filename")
```

Then, it is possible to load an ASCII ARRAY and to cast it to a SET:

```
AML> ToSet(Array("filename"))
```

– There is no possibility to Plot SETS.

– Two sets are equal if they contain the same elements (no matter the order). The semantics of the equality between two elements is either straightforward (*e.g.* equality beween to integers or between an integer and a REAL, etc.) or defined in the constructor man page of the element.

SEE ALSO

Size, Sum, Array, List, ToArray, ToSet, ToList.

EXAMPLES

```
AML> s=Set(2,3,10,3,2,2,2,10,11,2)
     <ARRAY(INT)> [2,3,10,11]
AML> s=Set(2,3,Undef,3,Undef)
     <ARRAY(INT)>[2,3,Undef]
AML> s=Set([1,2,3],[9],[],[1,2,3],[9])
     <ARRAY(ARRAY((INT))>[[],[9],[1,2,3]]
```

SetMinus

Difference between to set objects, -

USAGE

```
SetMinus(set1,set2)
```

ARGUMENTS

`set1,set2` (`ARRAY` or `SET`) : these two arguments must have the same type

RETURNED OBJECT

`SetMinus` returns a set object of the same type as its two arguments, *i.e.* either `ARRAY` or `SET`. If either `set1` or `set2` is `Undef`, returns `Undef`.

DESCRIPTION

`SetMinus(set1,set2)` returns the set object made up by the objects of `set1` that are not in `set2` If `set1` and `set2` are `ARRAY`s, `SetMinus(set1,set2)` preserves in the resulting `ARRAY` the order of elements of `set1`.

NOTE

`SetMinus` can also be invoked with the minus sign `-`.

EXAMPLES

```
AML> SetMinus([1,2,3],[3,1,4])
     <ARRAY(ANY)>[2]
AML> SetMinus(Set(1,2,3,4,5),Set(1,3,5,6,7))
     <SET(ANY)>[2,4]
```

Size

Gives the number of elements of an array, a set or a list.

USAGE

```
Size(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `SET(T)` or `LIST(...)`). `T` is either any type. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)` or `SET(T)`, the function returns an element of type `T`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the number of the elements of `array`.

SEE ALSO

`Array`, `Set`, `Series`, `Size`.

EXAMPLES

```
AML>  Size([10,11,12,13,14,15])
      <INT> 6
AML>  Size([2,2,Undef,7,1])
      <INT> 5
AML>  Size([2:100])
      <INT> 99
```

Sort

Sorts the order of the elements of an array or a set.

USAGE

```
Sort(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `SET(...)`). `T` is any type. If the argument is `Undef`, returns `Undef`. The elements of `array` must have a simple type, *i.e.* `T` = `INT` or `VTX` or `REAL` or `DATE`, or `CHAR` or `STRING`.

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`. If `array` has type `SET(T)`, the function returns an element of type `ARRAY(T)`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns a sorted array made of all the element of `array`.

NOTE

- If `array` is a `SET`, the object returned is not a `SET`. Rather, it is an `ARRAY`.
- The comparison semantics of complex objects is undefined.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Head`, `Invert`.

EXAMPLES

```
AML> Sort([12,14,12,15,11,13])
     <ARRAY(INT)> [11,12,12,13,14,15]
AML> Sort([[1,2],[3,4,5],Undef,[7],[8,9,10]])
     <ARRAY(ARRAY(INT))>[[8,9,10],[7],Undef,[3,4,5],[1,2]]
```

SProd

Scalar product of two arrays.

USAGE

```
SPROD(A1, A2)
```

ARGUMENTS

`a1`, `a2` (`ARRAY(REAL)`). These two arrays must have identical size.

RETURNED OBJECT

The result of a scalar product is a `REAL`. If one of the arguments is `Undef`, returns `Undef`

DESCRIPTION

Denote `a1` $[x_1, x_2, \ldots, x_n]$ and `a2` $[y_1, y_2, \ldots, y_n]$. The scalar product of two arrays with same dimension *n* is defined by:

$$s = \sum_{k=1}^{n} x_k \cdot y_k$$

SEE ALSO

`EDist`, `Angle`, `Norm`, `VProd`, `Array`.

EXAMPLES

```
AML>  SProd([1.,0.,0.],[0.,1.,0.])
      <REAL> 0
AML>  SProd([1.,1.,0.,4.],[1.,1.,0.,2.])
      <REAL> 10
```

SubArray

Subarray of an array or a list.

USAGE

```
SubArray(array, i, j)
SubArray(array, i)
```

ARGUMENTS

`array` (`ARRAY(T)`). `T` is any type. If the argument is `Undef`, returns `Undef`

`i`, `j` (`INT`). Define the bounds (minimum and maximum index in `array`) of the subarray. For syntax `SubArray(array,i,j)`, `i` must be positive, non null and `j` must be greater than `i`. For syntax `SubArray(array,i)`, `i` must be non null, but can have a negative value.

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`. If one of the arguments is `Undef`, returns `Undef`

DESCRIPTION

– `SubArray(array,i,j)` returns the array made of the element of `array` that have a position between `i` and `j`. The elements of the resulting array are in the same order as in `array`. Let us denote `array` $\leftarrow [x_1, x_2, \ldots, x_n]$. Then syntax returns the subarray $[x_i, \ldots, x_j]$. If `j` > `n` then the result is the subarray $[x_i, \ldots, x_n]$.

– Syntax `SubArray(array,i)` returns the first `i` element of `array` if `i`>0 and the last `i` elements if `i`<0. If `Abs(i)`>*n*, then the entire array `array` is returned.

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Head`, `Tail`, `ValueSelect`.

EXAMPLES

```
AML>  SubArray([10,11,12,13,14,15],2,4)
      <ARRAY(INT)> [11,12,13]
AML>  SubArray([10,11,12,13,14,15],2,18)
      <ARRAY(INT)> [11,12,13,14,15]
AML>  SubArray([[1,2],[3,4,5],Undef,[7],[8,9,10]],3)
      <ARRAY(ARRAY(INT))>[[1,2],[3,4,5],Undef]
AML>  SubArray([[1,2],[3,4,5],Undef,[7],[8,9,10]],-2)
      <ARRAY(ARRAY(INT))>[[7],[8,9,10]]
```

Sum

Sum up the elements of an array or a set.

USAGE

```
Sum(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `SET(T)`). `T` is either `INT` or `REAL`. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)` or `SET(T)`, the function returns an element of type `T`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the sum of the elements of `array`. If `array` contains `Undef` elements, these elements are discarded during the `Sum` operation (they are considered as having `Null` value).

SEE ALSO

`Array`, `Set`, `Series`, `Size`.

EXAMPLES

```
AML>  Sum([10,11,12,13,14,15])
      <INT> 75
AML>  Sum([2,2,Undef,7,1])
      <INT> 12
AML>  Sum(Foreach _x In [1,10,100] : _x*2)
      <INT> 222
```

Switch

Multi-conditional expression.

USAGE

```
Switch v Case v1 : e1 Case v2 : e2 Case ... Default ek
```

ARGUMENTS

v (T). is any expression
v1 (T_i). T_i can be any type.
e1 (T_i). T_i can be any type.

RETURNED OBJECT

The object returned can have the same type as either e1, e2,...or ek.

DESCRIPTION

This expression returns e1 if v = v1, e2 if v = v2, etc. It returns ek if none of the Case expressions matches v.

SIDE EFFECT

Important: in the first version of AMAPmod, the evaluation procedure of expressions evaluates all subexpressions before evaluating a given expression. Thus, in every cases, both e1 and e2 are evaluated first. Then depending on the value of v the evaluation of the Switch expression leads to select either e1 or e2 or any other e*i* expression. This means that some time may be lost in computing useless expressions. This behavior will be corrected in version 2.

SEE ALSO

Array, List, Pos, At, Append.

EXAMPLES

```
AML> Switch 1 + 3 Case 1 :"A" Case 2 : "B" Case 3 : "C" \
AML> Case 4 : "D" Default : "Z"
     <STRING>D
AML> # A function for coloring a PlantFrame (the argument
      enotes a plant entity)
AML> color _order(_x) = Switch Order(_x) Case 1 : Black \
AML> Case 2: Blue Case 3 : Green Case 4 : Red Default : Yellow
     <FUNC>Func.
AML> f = PlantFrame(1,Scale-> 3)
     <PlantFrame> Standard.
AML> Plot(f,Color->color_order)
```

Tail

Tail of an array.

USAGE

```
Tail(array)
```

ARGUMENTS

`array` (`ARRAY(T)` or `LIST(...)`). `T` is any type. If the argument is `Undef`, returns `Undef`

RETURNED OBJECT

If `array` has type `ARRAY(T)`, the function returns an element of type `ARRAY(T)`. If `array` has type $\texttt{LIST}(T_1, T_2, \ldots, T_N)$, the function returns an element of type $\texttt{LIST}(T_2, \ldots, T_N)$. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the an array or a list made of all the element of `array` except the first element. The elements of the resulting array are in the same order as in `array`. This is not equivalent to the expression `SubArray(array,2,Size(array))` since `Tail` can take an argument that contains only one or zero elements (which is not the case of the `SubArray` expression).

SEE ALSO

`Array`, `List`, `Pos`, `At`, `Head`.

EXAMPLES

```
AML> Tail([10,11,12,13,14,15])
     <ARRAY(INT)> [11,12,13,14,15]
AML> Tail([[1,2],[3,4,5],Undef,[7],[8,9,10]])
     <ARRAY(ARRAY(INT))>[[3,4,5],Undef,[7],[8,9,10]]
AML> Tail([3])
     <ARRAY(ANY))>[]
AML> Tail([])
     <ARRAY(ANY))>[]
```

ToArray

Cast type to ARRAY.

USAGE

```
ToArray(x)
```

ARGUMENTS

x (ARRAY or SET or LIST or SEQUENCES or DISCRETE_SEQUENCES or MARKOV_DATA or SEMI-MARKOV_DATA)

RETURNED OBJECT

The value returned by ToArray has type ARRAY. If x is Undef, returns Undef.

DESCRIPTION

Transforms any set type (ARRAY, SET or LIST or SEQUENCES) to ARRAY whenever possible. It is always possible to cast a SET to an ARRAY, but a LIST can only be cast to a set if all its element have identical type.

NOTE

This function may be useful to remove duplicated elements from an array (*cf.* examples)

SEE ALSO

ToSet, ToList, ToString, Sequences.

EXAMPLES

```
AML>  a = [1,2,2,2,3,2,4,4,4,2,3,2]
      <ARRAY(INT)> [1,2,2,2,3,2,4,4,4,2,3,2]
AML>  b = ToSet(a)
      <SET(INT)>[1,2,3,4]
AML>  # c corresponds to arraya where duplicated element shave
      been removed
AML>  c = ToArray(b)
      <ARRAY(INT)>[1,2,3,4]
AML>  s = Sequences([[1,2,3,4],[1,2,3]])
      <SEQUENCES> 1 VARIABLE  2 sequences  cumulative length: 7
AML>  c = ToArray(s)
      <ARRAY(ARRAY(INT))>[[1,2,3,4],[1,2,3]]
```

ToInt

Cast type to INT.

USAGE

```
ToInt(x)
```

ARGUMENTS

x (INT or REAL or VTX or DATE) : a numerical value

RETURNED OBJECT

The value returned by ToInt has type INT. If x is Undef, returns Undef.

DESCRIPTION

Cast a numerical value value to an integer. If x has type REAL or INT, ToInt is equivalent to Truncate. If x has type VTX or DATE, ToInt returns a value corresponding to a unique internal integer code representing the object x. No additional meaning is associated with this internal code.

SEE ALSO

Abs, Ceil, Rint, Floor, Truncate, ToInt, ToReal.

EXAMPLES

```
AML> ToInt(3)
     <INT>3
AML> ToInt(3.4)
     <REAL>3
AML> ToInt(3.6)
     <REAL>3
AML> ToInt(-2.9)
     <REAL>-2
AML> ToInt(-2.4)
     <REAL>-2
```

ToList

Cast type to LIST.

USAGE

```
ToList(x)
```

ARGUMENTS

x (ARRAY or SET or LIST)

RETURNED OBJECT

The value returned by ToList has type LIST. If x is Undef, returns Undef.

DESCRIPTION

Transforms any set type (ARRAY, SET or LIST) to a LIST whenever possible. It is always possible to cast an ARRAY or a SET to a LIST.

SEE ALSO

ToArray, ToSet, ToString.

EXAMPLES

```
AML>  a = [1,2,3,4]
      <ARRAY(INT)> [1,2,3,4]
AML>  b = ToList(a)
      <LIST(INT,INT,INT,INT)>[1,2,3,4]
```

ToReal

Cast type to REAL.

USAGE

```
ToReal(x)
```

ARGUMENTS

x (INT or REAL) : a numerical value

RETURNED OBJECT

The value returned by ToReal has type REAL. If x is Undef, returns Undef.

DESCRIPTION

Cast a numerical value value to a REAL.

SEE ALSO

ToInt, ToReal.

EXAMPLES

```
AML>  I = 3
      <INT>3
AML>  ToReal(i)
      <REAL>3
AML>  ToReal(3.4)
      <REAL>3.4
```

ToSet

Cast type to SET.

USAGE

```
ToSet(x)
```

ARGUMENTS

x (ARRAY or SET or LIST)

RETURNED OBJECT

The value returned by ToSet has type SET. If x is Undef, returns Undef.

DESCRIPTION

Transforms any set type (ARRAY, SET or LIST) to a SET whenever possible. It is always possible to cast an ARRAY to a set, but a LIST can only be cast to a set if all its element have identical type.

NOTE

This function may be useful to remove duplicated elements from an array (*cf.* examples)

SEE ALSO

ToArray, ToList, ToString.

EXAMPLES

```
AML> a = [1,2,2,2,3,2,4,4,4,2,3,2]
     <ARRAY(INT)> [1,2,2,2,3,2,4,4,4,2,3,2]
AML> b = ToSet(a)
     <SET(INT)>[1,2,3,4]
AML> # c corresponds to arraya where duplicated element shave
     been removed
AML> c = ToArray(b)
     <ARRAY(INT)>[1,2,3,4]
```

ToString

Cast type to STRING.

USAGE

```
ToString(x)
```

ARGUMENTS

x (INT, REAL, VTX, DATE, CHAR)

RETURNED OBJECT

The value returned by ToString has type STRING. If x is Undef, returns Undef.

DESCRIPTION

Transforms simple types to ASCII strings.

SEE ALSO

ToInt, ToReal.

EXAMPLES

```
AML> ToString(3)
     <STRING>3
AML> "AMAPmodv" + ToString(1)
     <STRING> AMAPmod v1
```

Trigonometric functions

Sin, Cos, Tan, Acos, Asin, Atan.

USAGE

```
Cos(x)
Tan(x)
Asin(x)
```

ARGUMENTS

x (INT or REAL)
Tan argument must be different from $\pi/2 \pm n\pi$ for any positive or negative *n*.
Arguments of Acos and Asin must be in the REAL interval [0,1].

RETURNED OBJECT

The returned values are REAL values. If the above conditions are not met, Undef is returned. If the argument is Undef, returns Undef.

DESCRIPTION

These functions are similar to the corresponding double function of the host system (Unix, ...). Type "man math" on the host system for more details.

SEE ALSO

Mathematical functions.

EXAMPLES

```
AML>  Cos(Pi/2)
      <REAL>1
AML>  Atan(1)
      <REAL> 0.785398
```

Truncate

Truncate to integer.

USAGE

```
Truncate(x)
```

ARGUMENTS

x (INT or REAL) : a numerical value

RETURNED OBJECT

The value returned by Truncate has the same type as x. If x is Undef, returns Undef.

DESCRIPTION

Truncate x to integer.

NOTE

This function is similar to the corresponding double function "trunc" of the host system (Unix, ...).

SEE ALSO

Abs, Ceil, Rint, Floor, ToInt, ToReal.

EXAMPLES

```
AML>  Truncate(3)
      <INT>3
AML>  Truncate(3.4)
      <REAL>3
AML>  Truncate(3.6)
      <REAL>3
AML>  Truncate(-2.9)
      <REAL>-2
AML>  Truncate(-2.4)
      <REAL>-2
```

Union

Union of to set objects, |

USAGE

```
Union(set1,set2)
set1 | set2
```

ARGUMENTS

`set1, set2` (ARRAY or SET) : these two arguments must have the same type

RETURNED OBJECT

Union returns a set object of the same type as its two arguments, *i.e.* either ARRAY or SET. If either `set1` or `set2` is `Undef`, returns `Undef`.

DESCRIPTION

`Union(set1,set2)` returns the set object made up by the objects of `set1` and of `set2`. If `set1` and `set2` are ARRAYs, `Union(set1,set2)` concatenates the two arrays.

NOTE

`Union` can also be invoked with the minus sign `|`.

SEE ALSO

`Inter`, `ToSet`, `ToArray`, `Sort`, `Merge`

EXAMPLES

```
AML> Union([1,2],[0,2,3,4])
     <ARRAY(ANY)>[1,2,0,2,3,4]
AML> Union(Set(1,2),Set(0,2,3,4))
     <SET(ANY)>[0,1,2,3,4]
```

Var

Computes the covariance of elements of two arrays.

USAGE

```
Var(a1)
```

ARGUMENTS

`a1` (`ARRAY(T)` or `SET(T)`). `T` is either `INT` or `REAL`.

RETURNED OBJECT

If `a1` has type `ARRAY(T)` or `SET(T)`, the function returns an element of type `REAL`. If the argument is `Undef`, returns `Undef`

DESCRIPTION

Returns the mean value of the elements of `array`. If `a1` = $\leftarrow x_1, x_2, .., x_i, .. \rightarrow$ then `Var(a1)` is the real :

$$1/n \sum_{i=1,n} x_i^2 - \overline{x}^2$$

If `a1` contains `Undef` elements, these elements are considered as `Null` values.

SEE ALSO

`Array`, `Set`, `Series`, `Size`, `EDist`, `Angle`, `Norm`, `VProd`, `Plus`, `Times`, `Cov`, `Mean`.

EXAMPLES

```
AML> Var([6.5,4.5,4.9,7.2,0.6,4.5])
     <INT> 4.40333
```

`VProd`

Vectorial product of two arrays.

USAGE

```
VProd(a1, a2)
```

ARGUMENTS

`a1`, `a2` (`ARRAY(REAL)`). These two arrays must have dimension 3.

RETURNED OBJECT

The result of a vectorial product is an `ARRAY(REAL)` with the same dimension as `a1` and `a2`. If one of the arguments is `Undef`, returns `Undef`

DESCRIPTION

The function returns the vectorial product of two arrays in dimension 3. If `a1` $=[x_1,y_1,z_1]$ and `a2` $=[x_2,y_2,z_2]$, then $a=[x,y,z]=$ `VProd(a1, a2)` with

$$x = y_1 z_2 - z_1 y_2$$
$$y = z_1 x_2 - x_1 z_2$$
$$z = x_1 y_2 - y_1 x_2$$

NOTE

Contrary to `SProd`, the dimension is limited here to 3.

SEE ALSO

`EDist`, `Angle`, `Norm`, `VProd`, `Array`.

EXAMPLES

```
AML>  VProd([1.,0.,0.],[0.,1.,0.])
      <ARRAY(REAL)> [0.,0.,1.]
AML>  SProd([0.,1.,0.],[1.,0.,0.])
      <ARRAY(REAL)> [0.,0.,-1.]
```

2 THE MTG MODULE

This part describes the functions of the module MTG of AMAPmod.

2.1 List of AML functions from module MTG (alphabetic order)

2.2 List of AML functions from module MTG (by type)

Construction

```
MTG (e1)
Activate(e1)
Active()
Entry points in the MTG
MTGRoot()
VtxList() ARRAY(VTX)
```

Feature functions

```
Class(e1)
Index(e1)
Scale(e1)
Feature(e1, e2)
Feature(e1, e2, e3)
ClassScale(e1)
EdgeType(e1, e2)
Defined(e1)
Order(e1)
Rank(e1)
Height(e1)
Order(e1,e2)
Rank(e1,e2)
AlgOrder(e1,e2)
AlgRank(e1,e2)
AlgHeight(e1,e2)
```

Date functions

```
DateSample(e1)
FirstDefinedFeature(e1,e2)
LastDefinedFeature(e1,e2)
DateSample(e1)
NextDate(e1)
PreviousDate(e1)
```

Functions for moving in MTGs

```
Father(e1)
Successor(e1)
Predecessor(e1)
Root(e1)
Complex(e1)
Location(e1)
Sons(e1)
Ancestors(e1)
Descendants(e1)
Extremities(e1)
```

```
Components(e1)
ComponentRoots(e1)
Path(e1,e2)
Axis(e1)
Trunk(e1)
```

Geometric interpretation

```
PlantFrame(e1)
TopCoord(e1, e2)
RelTopCoord(e1, e2)
BottomCoord(e1, e2)
RelBottomCoord(e1, e2)
Coord(e1, e2)
DressingData(e1)
Plot(e1)
BottomDiameter(e1, e2)
TopDiameter(e1, e2)
Alpha(e1, e2)
Beta(e1, e2)
Length(e1, e2)
VirtualPattern(e1)
PDir(e1, e2)
SDir(e1, e2)
```

Comparison functions

```
TreeMatching(e1)
MatchingExtract(e1)
```

Options (*cf.* `Father()`)

```
EdgeType        (+, <, *)
RestrictedTo    (NoRestriction, SameComplex, SameAxis)
ContainedIn     (i)
Scale           (i)
Format          (ASCII, GNUPLOT)
ErrorNb         (i)
```

2.3 Detailed description

Activate

Activate a MTG already loaded into memory

USAGE

```
Activate(mtg)
```

ARGUMENTS

`mtg` (`MTG`) : MTG to be activated

RETURNED OBJECT

`VOID`

DESCRIPTION

All the functions of the MTG module use an implicit MTG argument which is defined as the "active MTG". This function activates a MTG already loaded into memory which thus becomes the implicit argument of all functions of module MTG.

DETAILS

When several MTGs are loaded into memory, only one is active at a time. By default, the active MTG is the last MTG loaded using function `MTG()`. However, it is possible to activate an MTG already loaded using function `Activate()`. The current active MTG can be identified using function `Active()`.

BACKGROUND

MTGs, AML language

SEE ALSO

`MTG`, `Active()`.

Active

Returns the active MTG.

USAGE

```
Active()
```

ARGUMENTS

None.

RETURNED OBJECT

`MTG`

DESCRIPTION

This function returns the active MTG. If no MTG is loaded into memory, `Undef` is returned.

DETAILS

When several MTGs are loaded into memory, only one is active at a time. By default, the active MTG is the last MTG loaded using function `MTG()`. However, it is possible to activate an MTG already loaded using function `Activate()`. The current active MTG can be identified using function `Active()`.

BACKGROUND

MTGs, AML language.

SEE ALSO

`MTG`, `Activate()`.

AlgHeight

Algebraic value defining the number of components between two components

USAGE

```
AlgHeight(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

INT. If v1 is not an ancestor of v2 (or vice versa) , or if v1 and v2 are not defined at the same scale, an error value Undef is returned.

DESCRIPTION

This function is similar to function Height(v1, v2): it returns the number of components between two components, at the same scale, but takes into account the order of vertices v1 and v2. The result is positive if v1 is an ancestor of v2, and negative if v2 is an ancestor of v1.

BACKGROUND

MTGs

SEE ALSO

MTG, Rank, Order, Height, EdgeType, AlgOrder, AlgRank.

AlgOrder

Algebraic value defining the relative order of one vertex with respect to another one

USAGE

```
AlgOrder(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

INT. If v1 is not an ancestor of v2 (or vice versa) , or if v1 and v2 are not defined at the same scale, an error value Undef is returned.

DESCRIPTION

This function is similar to function Order(v1, v2): it returns the number of '+' -type edges between two components, at the same scale, but takes into account the order of vertices v1 and v2. The result is positive if v1 is an ancestor of v2, and negative if v2 is an ancestor of v1.

BACKGROUND

MTGs

SEE ALSO

MTG, Rank, Order, Height, EdgeType, AlgHeight, AlgRank.

AlgRank

Algebraic value defining the relative rank of one vertex with respect to another one

USAGE

```
AlgRank(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

INT. If v1 is not an ancestor of v2 (or vice versa) within the same botanical axis, or if v1 and v2 are not defined at the same scale, an error value Undef is returned.

DESCRIPTION

This function is similar to function Rank(v1, v2): it returns the number of consecutive '<'-type edges between two components, at the same scale, but takes into account the order of vertices v1 and v2. The result is positive if v1 is an ancestor of v2, and negative if v2 is an ancestor of v1.

BACKGROUND

MTGs

SEE ALSO

MTG, Rank, Order, Height, EdgeType, AlgHeight, AlgOrder.

Alpha

Angle defining the angle between the principal direction of the geometric model of vertex and the z axis of the global coordinate system.

USAGE

```
Alpha(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

REAL

DESCRIPTION

This function returns an angle between the principal direction of the geometric model of a vertex and the z axis of the global reference system. Note that this angle might note be defined in the MTG coding file since it may result from an inference process in the `PlantFrame` function.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `TopDiameter`, `Length`, `BottomDiameter`, `Beta`.

Ancestors

Array of all vertices which are ancestors of a given vertex

USAGE

```
Ancestors(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

RestrictedTo (STRING): *cf.* Father
ContainedIn (VTX): *cf.* Father
EdgeType (CHAR): *cf.* Father

RETURNED OBJECT

ARRAY(VTX).

DESCRIPTION

This function returns the array of vertices which are located before the vertex passed as an argument. These vertices are defined at the same scale as v. The array starts by v, then contains the vertices on the path from v back to the root (in this order) and finishes by the tree root.

NOTE

The anscestor array always contains at least the argument vertex v.

BACKGROUND

MTGs

SEE ALSO

MTG, Descendants.

EXAMPLES

```
AML> v # prints vertex v
     <VTX>78
AML> Ancestors(v) # set of ancestors of v at the same scale
     <ARRAY(VTX)>[78,45,32,10,4]
AML> Invert(Ancestors(v)) # To get the vertices in the order
from the root to the vertex v
     <ARRAY(VTX)>[4,10,32,45,78]
```

Axis

Array of vertices constituting a botanical axis

USAGE

```
Axis(v)
Axis(v, Scale-> s)
```

ARGUMENTS

v (VTX) : Vertex of the active MTG

OPTIONAL ARGUMENTS

Scale (STRING): scale at which the axis components are required.

RETURNED OBJECT

ARRAY(VTX)

DESCRIPTION

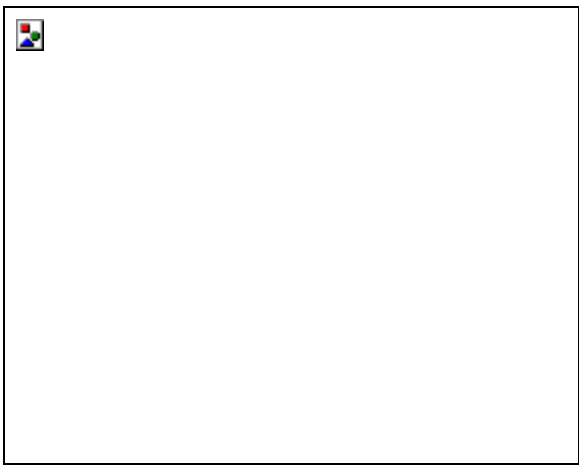
An axis is a maximal sequence of vertices connected by '<'-type edges. Axis return the array of vertices representing the botanical axis which the argument v belongs to. The optional argument enables the user to choose the scale at which the axis decomposition is required.

BACKGROUND

MTGs

SEE ALSO

MTG, Path, Ancestors.



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

Beta

Angle defining the angle between the principal direction of the geometric model of vertex and the x axis of the global coordinate system.

USAGE

```
Beta(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

REAL

DESCRIPTION

This function returns the angle between the principal direction of the geometric model of a vertex and the x axis of the global reference system. Note that this angle might note be defined in the MTG coding file since it may result from an inference process in the PlantFrame function.

BACKGROUND

MTGs

SEE ALSO

MTG, TopDiameter, Length, BottomDiameter, Alpha.

BottomCoord

Coordinates of the bottom of the geometric model of a component

USAGE

```
BottomCoord(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the top of the box containing the geometric model of a plant component. The result is an array of 3 reals.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, TopCoord.

BottomDiameter

Bottom diameter of the geometric model of a vertex

USAGE

```
BottomDiameter(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

REAL

DESCRIPTION

This function returns the bottom diameter of the geometric model of a vertex. Note that this diameter might note be defined in the MTG coding file since it may result from an inference process in the PlantFrame function.

BACKGROUND

MTGs

SEE ALSO

MTG, TopDiameter, Length, Alpha, Beta.

Class

Class of a vertex

USAGE

```
Class(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

RETURNED OBJECT

CHAR

DESCRIPTION

The class of a vertex is a feature always defined and independent of time (like the index). It is represented by an alphabetic character in upper or lower case (lower cases characters are considered different from upper cases). The label of a vertex is the string defined by the concatenation of its class and its index. The label thus provides general information about a vertex and enables us to encode the plant components.

BACKGROUND

MTGs

SEE ALSO

MTG, Index.

ClassScale

Scale at which appears a given class of vertex

USAGE

```
ClassScale(c)
```

ARGUMENTS

`c` (`CHAR`) : symbol of the considered class

RETURNED OBJECT

`INT`

DESCRIPTION

Every vertex is associated with a unique class. Vertices from a given class only appear at a given scale which can be retrieved using this function.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `Class`, `Scale`, `Index`.

Complex

Complex of a vertex

USAGE

```
Complex(v)
Complex(v, Scale-> 2)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG

OPTIONAL ARGUMENTS

`Scale` (`INT`) : scale of the complex.

RETURNED OBJECT

`VTX`

DESCRIPTION

Returns the complex of the argument. The complex of a vertex `v` is has a scale lower than `v:` `Scale(v)-1`. In a MTG, every vertex except for the MTG root (*cf.* `MTGRoot()`), has a uniq complex. `Undef` is returned if the argument is the MTG root or if the vertex is undefined.

DETAILS

When a scale different from `Scale(v)-1` is specified using the optional argument `Scale`, this scale must be lower than that of the vertex argument.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `Components`.

ComponentRoots

Set of roots of the tree graphs that compose a vertex

USAGE

```
ComponentRoots(v)
ComponentRoots(v, Scale-> s)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

Scale (STRING): scale of the component roots.

RETURNED OBJECT

ARRAY(VTX)

DESCRIPTION

In a MTG, a vertex may have be decomposed into components. Some of these components are connected to each other, while other are not. In the most general case, the components of a vertex are organized into several tree-graphs. This is for example the case of a MTG containing the description of several plants: the MTG root vertex can be decomposed into tree graphs (not connected) that represent the different plants. This function returns the set of roots of these tree graphs at scale Scale(v)+1. The order of these roots is not significant.

DETAILS

When a scale different from Scale(v)+1 is specified using the optional argument Scale, this scale must be greater than that of the vertex argument.

BACKGROUND

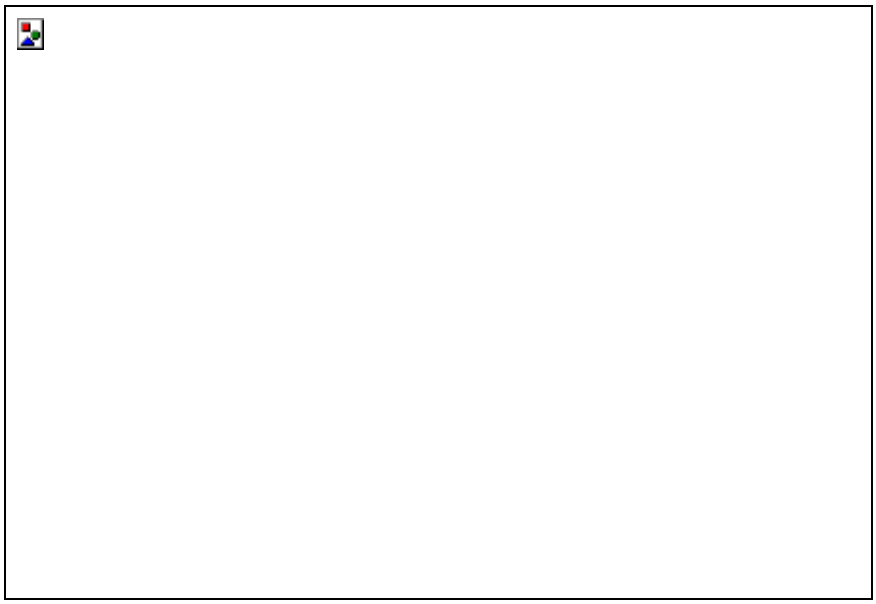
MTGs

SEE ALSO

MTG,Components,Trunk.

EXAMPLES

```
AML> v=MTGRoot() # racine globale du MTG
     <ARRAY(VTX)>0
AML> ComponentRoots(v) # ensemble des premiers vertex des
plantes à l'échelle 1
     <ARRAY(VTX)>[1,34,76,100,199,255]
AML> ComponentRoots(v, Scale-> 2) # ensemble des premiers
vertex des plantes à l'échelle 2
     <ARRAY(VTX)>[2,35,77,101,200,256]
```



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

Components

Set of components of a vertex

USAGE

```
Components(v)
Components(v, Scale-> 2)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

Scale (INT) : scale of the components.

RETURNED OBJECT

ARRAY(VTX)

DESCRIPTION

The set of components of a vertex is returned as an array of vertices. If *s* defines the scale of v, components are defined at scale *s* + 1. The array is empty if the vertex has no components. The order of the components in the array is not significant.

DETAILS

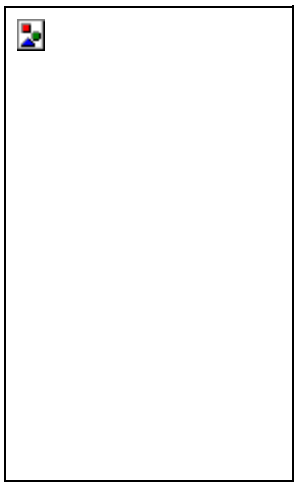
When a scale is specified using optional argument Scale, it must be necessarily greater than the scale of the argument.

BACKGROUND

MTGs

SEE ALSO

MTG, Complex.



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

Coord

Top coordinates of the geometric model of a component

USAGE

```
Coord(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the top of the box containing the geometric model of a plant component. The result is an array of 3 reals.

NOTE

This function is similar to TopCoord.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, TopCoord, BottomCoord.

DateSample

Array of observation dates of a vertex.

USAGE

```
DateSample(v)
DateSample(v, MinDate-> d1, MaxDate-> d2)
```

ARGUMENTS

v (VTX) : vertex of the active MTG.

OPTIONAL ARGUMENTS

MinDate (DATE) : defines a minimum date of interest.
MaxDate (DATE) : defines a maximum date of interest.

RETURNED OBJECT

ARRAY(DATE)

DESCRIPTION

Returns the set of dates at which a given vertex (passed as an argument) has been observed as an array of ordered dates. Options can be specified to define a temporal window and the total list of observation dates will be truncated according to the corresponding temporal window.

BACKGROUND

Dynamic MTGs.

SEE ALSO

MTG, FirstDefinedFeature, LastDefinedFeature, PreviousDate, NextDate.

Defined

Test whether a given vertex belongs to the active MTG.

USAGE

```
Defined(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

RETURNED OBJECT

BOOL

DESCRIPTION

Test whether a given vertex belongs to the active MTG. Note that if the argument is an INT, it is automatically converted to a VTX before the function is applied.

BACKGROUND

MTGs

SEE ALSO

MTG.

Descendants

Set of vertices in the branching system borne by a vertex.

USAGE

```
Descendants(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

RestrictedTo (STRING): *cf.* Father
ContainedIn (VTX): *cf.* Father
EdgeType (CHAR): *cf.* Father

RETURNED OBJECT

ARRAY(VTX)

DESCRIPTION

This function returns the set of descendants of its argument as an array of vertices. The array thus consists of all the vertices, at the same scale as v, that belong to the branching system starting at v. The order of the vertices in the array is not significant.

NOTE

The argument always belongs to the set of its descendants.

BACKGROUND

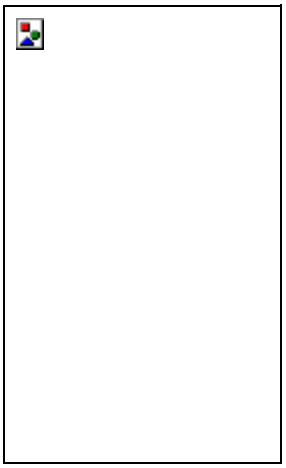
MTGs

SEE ALSO

MTG, Ancestors.

EXAMPLES

```
AML> v #
     <VTX>78
AML> Sons(v) # set of sons of v
     <ARRAY(VTX)>[78,99,101]
AML> Descendants(v) # set of descendants of v
     <ARRAY(VTX)>[78,99,101,121,133,135,156,171,190]
```



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

DressingData

Data and default geometric parameters used to compute the geometric interpretation of a MTG (*i.e.* a PLANTFRAME)

USAGE

```
DressingData(dressing_file)
```

ARGUMENTS

`dressing_file` : Name of the file containing the description of the dressing data.

RETURNED OBJECT

DRESSING_DATA

DESCRIPTION

The dressing data contains the default data that are used to define the geometry of an MTG vertices (*i.e.* of a plant entities) and to compute their geometric parameters when inference algorithms cannot be applied. These data are basically constant values and may be redefined in the dressing file. If no dressing file is defined, default (hard-coded) values are used (*see* **0**). The dressing file .drf , if it exists in the current directory, is always used as a default dressing file.

Objects of type DRESSING_DATA is used by primitive Plantframe. It may also be used by primitive Plot when VIRTUAL_PATTERNs are plotted.

DETAILS

cf. example of a dressing file given in the annex section.
It may be noted that a given DRESSING_DATA object can be used for different PLANTFRAMEs.

BACKGROUND

cf. example of a dressing file given in the annex section.

SEE ALSO

PlantFrame, VirtualPattern, Plot.

EXAMPLES

```
AML> g=MTG("an_mtg")
AML> d=DressingData("file")
AML> fr3=PlantFrame( 1,Scale-> 3,DressingData-> d)
AML> fr4=PlantFrame( 1,Scale-> 4,DressingData-> d)
AML> Plot(fr4)
```

EdgeType

Type of connection between two vertices.

USAGE

```
EdgeType(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

CHAR

DESCRIPTION

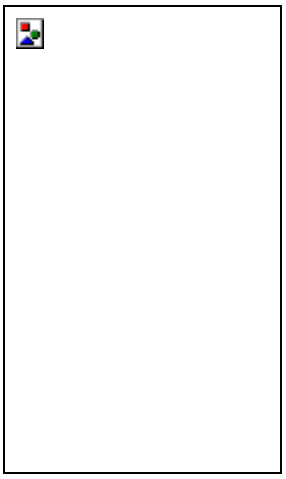
Returns the symbol of the type of connection between two vertices (either '<' or '+'). If the vertices are not connected, `Undef` is returned.

BACKGROUND

MTGs

SEE ALSO

`MTG, Sons, Father.`



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

EulerAngles

Computes the Euler angles corresponding to the local coordinate system of a vertex with respect to the global coordinate system.

USAGE

```
EulerAngles(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

Computes the Euler angles corresponding to the local coordinate system defined by PDir and SDir of a vertex with respect to the global coordinate system. The result is an array of 3 reals. First real is the azimuth (rotation about z-axis), the second real is the elevation (rotation about the y axis). The third real is the twist (rotation about the x axis).

These angles are exactly those used by the Polhemus 3D digitizer.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, RelTopCoord, RelBottomCoord, TopBottomCoord, BottomCoord, Pdir, SDir.

Extremities

Set of vertices that are the extremities of the branching system borne by a given vertex.

USAGE

```
Extremities(v)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG

OPTIONAL ARGUMENTS

`RestrictedTo` (`STRING`): *cf.* Father
`ContainedIn` (`VTX`): *cf.* Father

RETURNED OBJECT

`ARRAY(VTX)`

DESCRIPTION

This function returns the extremities of the branching system defined by the argument as an array of vertices. These vertices have the same scale as `v` and their order in the array is not significant. The result is always a non empty array.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `Descendants`, `Root`, `MTGRoot`.

EXAMPLES

```
AML> Descendants(v)
     <ARRAY(VTX)>[3,45,47,78,102]
AML> Extremities(v)
     <ARRAY(VXT)>[47,102]
```

Father

Topological father of a given vertex.

USAGE

```
Father(v)
Father(v, EdgeType-> `<`)
Father(v, RestrictedTo-> SameComplex)
Father(v, ContainedIn-> cv)
Father(v, Scale-> s)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG

OPTIONAL ARGUMENTS

If no optional argument is specified, the function returns the topological father of the argument (vertex that bears or precedes to the vertex passed as an argument).

It may be useful in some cases to consider that the function only applies to a subpart of the MTG (*e.g.* an axis). The following options enables us to specify such restrictions:

`EdgeType` (`CHAR`) : filter on the type of edge that connect the vertex to its father (values can be '`<`', '`+`', et '`*`'. Value '`*`' means both '`<`' and '`+`'). Only the vertex connected with the specified type of edge will be considered.

`RestrictedTo` (`STRING`): filter defining a subpart of the MTG where the father must be considered. If the father is actually outside this subpart, the result is `Undef`. Possible subparts are defined using keywords: `SameComplex`, `SameAxis`, `NoRestriction`. For example, if `RestrictedTo` is given the value `SameComplex`, `Father(v)` returns a defined vertex only if the father *f* of `v` exists in the MTG and if `v` and *f* have the same complex.

`ContainedIn` (`VTX`): filter defining a subpart of the MTG where the father must be considered. If the father is actually outside this subpart, the result is `Undef`. In this case, the subpart of the MTG is made of the vertices that composed `cv` (at any scale).

The scale of the considered father can also be specified:

`Scale` (`INT`): returns the vertex from scale `s` which either bears or precedes the argument. The scale can be lower than the argument's (*e.g.* corresponding to a question such as "which axis bears the internode ?") or greater (*e.g.* "which internode bears this annual shoot ?")

RETURNED OBJECT

`VTX`

DESCRIPTION

Returns the topological father of a given vertex. And `Undef` if the father does not exist. If the argument is not a valid vertex, `Undef` is returned and a warning message is displayed without producing any error.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `Defined, Sons, EdgeType, Complex, Components`.

Feature

Extracts the attributes of a vertex

USAGE

```
Feature(v, fname)
Feature(v, fname, date)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG.
`Fname(STRING)` : string describing the name of the attribute (as specified in the coding file).
`date(DATE)` : (for a dynamic MTG) date at which the attribute of the vertex is considered.

RETURNED OBJECT

`INT, STRING, DATE,` or `REAL`

DESCRIPTION

Returns the value of the attribute `fname` of a vertex in a MTG. If the value of an attribute is not defined in the coding file, the value `Undef` is returned.

DETAILS

If for a given attribute, several values are available (corresponding to different dates), the date of interest must be specified as a third attribute. This date must be a valid date appearing in the coding file for the considered vertex. Otherwise `Undef` is returned.

BACKGROUND

MTGs and Dynamic MTGs.

SEE ALSO

`MTG`, `Class`, `Index`, `Scale`.

FirstDefinedFeature

Date of first observation of a vertex.

USAGE

```
FirstDefinedFeature(v, fname)
FirstDefinedFeature(v, fname, MinDate-> d1, MaxDate-> d2)
```

ARGUMENTS

v (VTX) : vertex of the active MTG.
fname (STRING) : name of the considered attribute.

OPTIONAL ARGUMENTS

MinDate (DATE) : minimum date of interest.
MaxDate (DATE) : maximum date of interest.

RETURNED OBJECT

DATE

DESCRIPTION

Returns the date *d* for which the attribute fname is defined for the first time on the vertex v passed as an argument. This date must be greater than the option MinDate and/or less than the maximum MaxDate when specified. Otherwise the returned date is Undef.

BACKGROUND

Dynamic MTGs.

SEE ALSO

MTG, DateSample, LastDefinedFeature, PreviousDate, NextDate.

Height

Number of components existing between two components in a tree graph

USAGE

```
Height(v1)
Height(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

INT

DESCRIPTION

The height of a vertex (v2) with respect to another vertex (v1) is the number of edges (of either type '+' or '<') that must be crossed when going from v1 to v2 in the graph. This is thus a non negative integer. When the function only has one argument v1, the height of v1 correspond to the height of v1 with respect to the root of the branching system containing v1.

NOTE

When the function takes two arguments, the order of the arguments is not important provided that one is an ancestor of the other. When the order is relevant, use function AlgHeight().

BACKGROUND

MTGs

SEE ALSO

MTG, Rank, EdgeType, AlgOrder, AlgRank, AlgHeight.

Index

Index of a vertex

USAGE

```
Index(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

RETURNED OBJECT

INT

DESCRIPTION

The index of a vertex is a feature always defined and independent of time (like the class). It is represented by a non negative integer. The label of a vertex is the string defined by the concatenation of its class and its index. The label thus provides general information about a vertex and enables us to encode the plant components.

BACKGROUND

MTGs

SEE ALSO

MTG, Class.

LastDefinedFeature

Date of last observation of a given attribute of a vertex.

USAGE

```
LastDefinedFeature(v, fname)
LastDefinedFeature(v, fname, MinDate-> d1, MaxDate-> d2)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG.
`fname` (`STRING`) : name of the considered attribute.

OPTIONAL ARGUMENTS

`MinDate` (`DATE`) : minimum date of interest.
`MaxDate` (`DATE`) : maximum date of interest.

RETURNED OBJECT

`DATE`

DESCRIPTION

Returns the date *d* for which the attribute `fname` is defined for the last time on the vertex `v` passed as an argument. This date must be greater than the option `MinDate` and/or less than the maximum `MaxDate` when specified. Otherwise the returned date is `Undef`.

BACKGROUND

Dynamic MTGs.

BACKGROUND

MTGs dynamiques.

SEE ALSO

`MTG`, `DateSample`, `FirstDefinedFeature`, `PreviousDate`, `NextDate`.

Length

Length of the geometric model of a vertex

USAGE

```
Length(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

REAL

DESCRIPTION

This function returns the length of the geometric model of a vertex. Note that this length might note be defined in the MTG coding file since it may result from an inference process in the PlantFrame function.

BACKGROUND

MTGs

SEE ALSO

MTG, TopDiameter, BottomDiameter, Alpha, Beta.

Location

Vertex defining the father of a vertex with maximum scale.

USAGE

```
Location(v)
Location(v, Scale-> s)
Location(v, ContainedIn-> cv)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

Scale(INT): scale at which the location is required
ContainedIn(VTX): *cf.* Father

RETURNED OBJECT

VTX

DESCRIPTION

If no options are supplied, this function returns the vertex defining the father of a vertex with maximum scale (*cf.* Father). If it does not exit, Undef is returned. If a scale is specified, the function is equivalent to Father(v, Scale-> s).

BACKGROUND

MTGs

SEE ALSO

MTG, Father.

EXAMPLES

```
AML> Father(v, EdgeType-> `+`)
     <VTX> 7
AML> Complex(v)
     <VTX> 4
AML> Components(7)
     <ARRAY(VTX)> [9,19,23,34,77,89]
AML> Location(v)
     <VTX> 23
AML> Location(v, Scale-> Scale(v)+1)
     <VTX> 23
AML> Location(v, Scale-> Scale(v))
     <VTX> 7
AML> Location(v, Scale-> Scale(v)-1)
     <VTX> 4
```

MatchingExtract

Extract the results of a matching between two branching systems

USAGE

```
MatchingExtract (tm)
MatchingExtract (tm, ViewPoint-> Text)
MatchingExtract (tm, ViewPoint-> DistanceMatrix)
MatchingExtract (tm, ViewPoint-> List,
                 InputTree-> i,ReferenceTree->r)
```

ARGUMENTS

`tm` (`TREEMATCHING`) : Tree Matching obtained after comparison.

OPTIONAL ARGUMENTS

`ViewPoint` (`STRING`) : type of view point which can be extract from the tree matching.

- `Text`: A matrix of distances between tree graphs is returned giving also some statistics results.
- `DistanceMatrix`: A normalised distance matrix is returned. The distance obtained after comparison is normalised by the size of tree graphs. The returned object can be analysed as an object of type `DISTANCE_MATRIX`.
- `List` : This option provides a detailed analysis of the comparison between two tree graphs. A vertex array of matching air is thus returned.

`InputTree` (`INT`) : Index of the test tree graph

`ReferenceTree`(`INT`) : Index of the reference tree graph.

RETURNED OBJECT

`DISTANCE_MATRIX, ARRAY(ARRAY(VTX))`

DESCRIPTION

On sets of plants, the comparison algorithm produces a distance matrix that can be analysed with classical clustering algorithm. For pairs of plants, the algorithm output is a list of matched entities which makes it possible to carry out a detailed analysis of the matched subparts of plants.

BACKGROUND

Method for comparing unordered rooted tree graphs [51]

SEE ALSO

`MTG`, `Plot`, `TreeMatching`, `Distance_matrix`.

EXAMPLES

```
AML>  g = MTG("un_mtg")
AML> plants= VtxList(Scale->1)
AML> roots= Foreach p In plants: Components(p,Scale->2)@1
```

```
AML> match = TreeMatching(roots)
<AML> # Distance between plants
<AML> MatchingExtract(match,ViewPoint->"Text")
<AML> # Extraction of a distance matrix between plants
AML> matrix = MatchingExtract(match,ViewPoint->"Distance")
<DISTANCE_MATRIX>

<AML> # Detailled view point of matching between plant 2 and 4
<AML> list= MatchingExtract(match,ViewPoint->"List",\
<AML> InputTree->2,ReferenceTree->4)
<ARRAY(ARRAY)> : [[206,207,210,…],[827,830,831,…]]
<AML> input_vtx=list@1
<AML> ref_vtx=list@2

<AML> # Functions which return for any vertex it matched vertex
<AML> M(_v) = If (Pos(input_vtx,_v) != Undef) Then \
<AML> ref_vtx@(Pos(input_vtx,_v)) Else \
<AML> input_vtx@(Pos(ref_vtx,_v))
<FUNCTION>
<AML> I(_v) = If (Pos(input_vtx,_v) != Undef) Then \
<AML> _v Else input_vtx@(Pos(ref_vtx,_v))
<FUNCTION>
<AML> # Color functions
<AML> position_color(_v) = Switch Rank(_v) Case 0 : Blue \
<AML>                                      Case 1 : Red \
<AML>                                      Case 2 : Green\
<AML>                                      Case 3: Yellow \
<AML>                                      Case 4 : Violet \
<AML>                                      Default : LightBlue
<FUNCTION>
<AML> slice_color(_v)=If(Order(_v)==0) Then position_color(_v)\
<AML> Else slice_color(Father(_v))
<FUNCTION>
<AML> axis_color(_v)=If(I((Complex(_v)))!=Undef) Then \
<AML> slice_color(I(Complex(_v))) Else White
<FUNCTION>
<AML> # Plant Frame computation
<AML> pf=PlantFrame([plants@2,plants@4],Scale->2)
<AML> # Display the detailed view point: Each vertex of the
<AML> # input plant which appear in a matching pair is colored
<AML> # depending on it rank and it matching pair in reference
<AML> # plant is colored with same color. Other vertices are
<AML> # colored in White.
<AML> Plot(pf,LineFile->"wijick-order",Color->axis_color)
```

MTG

MTG constructor

USAGE

```
MTG(filename, ErrorNb-> 10)
MTG(filename, VtxNumber-> 10000)
```

ARGUMENTS

`filename` (`STRING`) : name of the coding file describing the MTG

OPTIONAL ARGUMENTS

`ErrorNb` (`INT`): Defines the maximum number of errors before exiting the MTG parsing.

`VtxNumber` (`INT`) : A priori estimated number of vertices described in the MTG coding file. This number is necessary to guess a memory space for the MTG data in order to optimize the parsing process. This number enables us to override the default estimation in places where it is not accurate.

RETURNED OBJECT

If the parsing process succeeds, the constructor return an object of type `MTG`. Otherwise, an error is generated and the formerly active MTG remains active.

DESCRIPTION

Builds a MTG from a coding file (text file) containing the description of one or several plants.

SIDEEFFECT

If the MTG is built, the new MTG becomes the active MTG (*i.e.* the MTG implicitly used by other functions such as `Father()`, `Sons()`, `VtxList()`, etc.).

DETAILS

The parsing process is approximately proportional to the number of components defined in the coding file.

NOTE

It can be the case that the estimated size of the MTG is too small. In such a case, the parsing process is dramatically slowed down because the machine incrementally allocates new memory blocks as necessary. To avoid this, it is possible to modify the estimation of the MTG size by giving an overestimate of the total number of vertices contained in the MTG. This is done by using option `VtxNumber`.

BACKGROUND

MTG is an acronyme for Multiscale Tree Graph.

SEE ALSO

`Sons`, `Father`, ...

MTGRoot

Root vertex of the MTG

USAGE

```
MTGRoot()
```

ARGUMENTS

None

RETURNED OBJECT

VTX

DESCRIPTION

Returns the root vertex of the MTG. It is the only vertex at scale 0 (the coarsest scale).

DETAILS

This vertex is the complex of all vertices from scale 1. It is a mean to refer to the entire database.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `Complex`, `Components`, `Scale`.

NextDate

Next date at which a vertex has been observed after a specified date

USAGE

```
NextDate(v, d)
```

ARGUMENTS

v (VTX) : vertex of the active MTG.
d (DATE) : departure date.

RETURNED OBJECT

DATE

DESCRIPTION

Returns the first observation date at which the vertex has been observed starting at date d and proceeding forward in time. Undef is returned if it does not exists.

BACKGROUND

Dynamic MTGs.

SEE ALSO

MTG, DateSample, FirstDefinedFeature, LastDefinedFeature, PreviousDate.

Order

Order of a vertex in a graph

USAGE

```
Order(v1)
Order(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

INT

DESCRIPTION

The order of a vertex (v2) with respect to another vertex (v1) is the number of edges of either type '+' that must be crossed when going from v1 to v2 in the graph. This is thus a non negative integer which corresponds to the "botanical order". When the function only has one argument v1, the order of v1 correspond to the order of v1 with respect to the root of the branching system containing v1.

NOTE

When the function takes two arguments, the order of the arguments is not important provided that one is an ancestor of the other. When the order is relevant, use function AlgOrder().

Warning: the value returned by function Order is 0 for trunks, 1 for branches etc. This might be different with some botanical conventions where 1 is the order of the trunk, 2 the order of branches, etc.

BACKGROUND

MTGs

SEE ALSO

MTG, Rank, Height, EdgeType, AlgOrder, AlgRank, AlgHeight.

Path

Array of vertices defining the path between two vertices

USAGE

```
Path(v1, v2)
```

ARGUMENTS

v1 (VTX) : vertex of the active MTG
v2 (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(VTX)

DESCRIPTION

This function returns the array of vertices defining the path between two vertices that are in an ancestor relationship. The vertex v1 must be an ancestor of vertex v2. Otherwise, if both vertices are valid, then the empty array is returned and if at least one vertex is undefined, Undef is returned.

NOTE

v1 can be equal to v2.

BACKGROUND

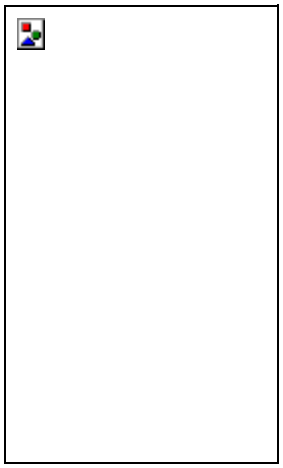
MTGs

SEE ALSO

MTG, Axis, Ancestors.

EXAMPLES

```
AML> v # print the value of v
     <VTX>78
AML> Ancestors(v)
     <ARRAY(VTX)>[78,45,32,10,4]
AML> Path(10,v)
     <ARRAY(VTX)>[10,32,45,78]
AML> Path(9,v) # 9 is not an ancestor of 78
     <ARRAY(ANY)>[]
```



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

PDir

Principal (or primary) direction of the geometric model of a vertex

USAGE

```
PDir(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the axial direction (principal or primary direction) of the box containing the geometric model of a plant component in the global coordinate system. The result is an array of 3 reals.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, RelTopCoord, RelBottomCoord, TopBottomCoord, BottomCoord.

PlantFrame

Constructs a geometric interpretation of a MTG.

USAGE

```
PlantFrame(v, Scale→ s, Translate→ [-10,-10,0])
PlantFrame(varray, VoxelDist→ 100)
PlantFrame(varray, TrunkDist→ 1000)
PlantFrame(v, Scale→ s, DressingData→ dr1)
PlantFrame(v, Scale→ s, Mode→ "Sympodial", Origin→ [12,13.4,
           -28])
PlantFrame(v, Scale→ s, XX→xcoord, YY→xcoord, ZZ→xcoord)
PlantFrame(v, XX→xcoord, YY→xcoord,
ZZ→xcoord,TopDiameter→diam,
           BottomDiameter→diam)
PlantFrame(v, XX→xcoord, YY→xcoord, ZZ→xcoord,
           EulerAngles→euler_flag,
           AA→euler1,BB→euler2,CC→euler3,)
PlantFrame(v, Category→ cat, Length→len, Alpha→alpha,
           Azimuth→azim)
PlantFrame(v, Scale→ s, LengthAlgo→ UseAxisIndexes)
```

ARGUMENTS

`v` (`VTX`): vertex defining a branching system. If no optional argument `Scale` is given, a compound geometric model of `v` is computed based on the geometric of the components `v` (i.e. at scale `Scale`(`v`)+1). Finer components can be considered by defining an optional argument `Scale` with value greater than `Scale`(`v`), provided that components are defined in the MTG at this scale. If optional argument `Scale` has value `Scale`(`v`), the branching system is defined as the set of all descendants of `v` (at the same scale as `v`), including `v` itself. This is a way to compute the geometric model of a part of a branching system only.

`varray` (`ARRAY(VTX)`): vertex array defining a set of branching systems (in the same way as above)

OPTIONAL ARGUMENTS

`Scale` (`INT`): scale at which the plant geometric interpretation is computed. The geometric model is a compound geometric model made of the combination of geometric models of components from scale `s`. By default, the scale at which the compound geometric model is computed is `Scale`(`v`)+1.

`Translate` (`ARRAY(REAL)`): defines a vector for translating a plant in the global coordinate system. Available only for a vertex argument (not for an array of vertices).

`Origin` (`ARRAY(REAL)`): (re)defines the origin (bottom coordinates) of the geometric model of the first component of a plant (at the basis of the trunk).

`VoxelDist` (`INT` or `REAL`): distance between plants (defined as the distance between the voxels containing the plants) in case more than one plant is considered.

`TrunkDist` (`INT` or `REAL`): distance between plants (defined as the distance between the trunks of the plants) in case more than one plant is considered. Lines of plants are aligned along the y axis. The maximum number of plants per line is controlled by variable NbPlantsPerLine which must be set in the DressingFile and which defaults to 10.

`DressingData` (`DRESSING_DATA`): uses the information contained in an object of type `DressingData` which consists of default data that can be used to compute the geometry of a plant.

`Mode` (`STRING`) : `Sympodial`, `Standard` : defines the way to build up axes for the geometric reconstruction. This is useful for sympodial plants where axes are actually apparent axes (as opposed to botanical axes) which correspond to stacks of botanical axes (or stacks of modules). In this case, the complexes of the components at scale `s` are used to group components into apparent axes. The geometric computation is then carried out on these apparent axes. Note that the complexes must of the components at scale `s` must have a `LINEAR` value in the field `DECOMPOSITION` of the class description of an MTG (*see* **Part II4**).

`TopDiameter` (`FUNC`): AML function defining the top diameter of a component.

`BottomDiameter` (`FUNC`): AML function defining the bottom diameter of a component.

`Length` (`FUNC`): AML function defining the length of a component.

`Category` (`FUNC`): AML function defining the category of a component (see below).

`Alpha` (`FUNC`): AML function defining the insertion angle of a component (in degrees) with respect to its father (by default) or to a vertical axis.

`XX` (`FUNC`): AML function defining the x-coordinate of (the top of) a component.

`YY` (`FUNC`): AML function defining the y-coordinate of (the top of) a component

`ZZ` (`FUNC`): AML function defining the z-coordinate of (the top of) a component

`AA` (`FUNC`): AML function defining the first Euler angle of a component.

`BB` (`FUNC`): AML function defining the second Euler angle of a component.

`CC` (`FUNC`): AML function defining the third Euler angle of a component.

`EulerAngles` (`FUNC`): AML function defining by a boolean value the components whose general orientation must be defined by the Euler angles (and not by inference algorithms).

`LengthAlgo` (`STRING`): `UseIndexes`, `UseAxisIndexes` choice of inference algorithms for the computation of component length (see below).

Returned Object

If the geometric reconstruction is successfully carried out, a `PLANTFRAME` object is returned. Otherwise an error is returned.

Description

This function computes a geometric interpretation of a MTG or a part of a MTG. This computation may rely on available geometric data, stored as features of the MTG components. However, algorithms have been designed in order to compute and to tune a geometric interpretation for any MTG, containing any amount of geometric information (including none at all). If geometric information is missing, heuristic algorithms are used to infer the missing data (ex: the length of a branch will be set proportional to its number of components). In the case no heuristic can be applied, default data for different geometric characteristics are retrieved from dressing data

(that are defined either as hard-coded default data or specified in a dressing file). The geometry of a branching system is computed axis per axis, starting from the trunk. Each axis geometry is entirely computed before the geometry of the borne branching systems is recursively computed. This ensures that all the geometric models of a branch are computed before starting the computation of the geometric models of the components of branches borne by this branch.

DETAILS

Definition of the branching system(s) whose geometry is to be computed. This branching system is defined either by a vertex or an array of vertices. A vertex implicitly defines the branching system as explained above. An array of vertices (the vertices must have the same scale) defines a set of branching systems and a geometric interpretation is computed for the branching system defined by each vertex of the array. Each geometric interpretation is then arranged on a lattice that is controlled by optional arguments `VoxelDist` or `TrunkDist.` The space between these plants can also be set in the dressing file (see `DressingData`).

Scale of the branching system. The geometric model of a branching system is a compound geometric model made of the combination of geometric models of components from scale `s`. The scale at which the geometric computation is carried out is by default the scale immediately greater than the scale of the vertex argument. For example, if the argument `v` is defined at scale 2, the geometric interpretation is computed at scale 3 (*i.e.* with finer components). This default behaviour can be changed by using optional argument `Scale` which must specify a scale greater than the scale of `v`.

Translating the origin of a plant. A translation of the origin of a plant can be made by using optional argument `Translate`. The value of this argument must be an array of 3 reals defining the coordinates of the translation. This can be used for instance to move the trunk of a plant at the origin of the coordinate system: if [x,y,z] defines the coordinates of the origin of the plant, the opional argument `Translate->`[-x,-y,-z] moves the origin of the plant at point [0,0,0]. This option usually applies in `PlantFrame`s computed for single vertices. For `PlantFrame`s computed for array of vertices corresponding to digitized plants, plants are by default located at the nodes of a regular grid (see options `TrunkDist` and `VoxelDist` and primitive `DressingData`). To change this default behaviour, you can use `Translate->`[0,0,0] which will make plant be located at their digitized position instead of "on a grid".

Defining the origin of a plant. If coordinates are used in a MTG, they always correspond to the coordinates of the top of a component geometric model (or more precisely, of center of the top face of the box containing the geometric model). We assume that the origin of all components is determined as the top of the father component. This enables us to compute the bottom coordinates of all components in the plant except for those components that have no father. For such components, the coordinates of the origin must be set in feature column `XX`, `YY` and `ZZ` (or `L1`, `L2`, `L3`, for triangular systems) of a complex of this component. If coordinates exist in several complexes of this component, the origin is determined by the coordinates of the complex with the closest scale to the component. Otherwise default value [0,0,0] is taken for the origin (with emission of a warning message).

Basic geometric models. Each MTG component at the specified scale is associated with a basic geometric model (*i.e.* defining its shape). In the current version of AMAPmod, the model is assumed to be controlled using 3 vectors and 3 scalar parameters: an origin in space (which consists of 3 coordinates), principal and secondary directions (defining the general orientation of the model in space), a length, a bottom diameter and a top diameter. The basic geometric model is assumed to be obtained from the geometric dilatation of a normalized model contained in a 1x1x1 box. To define the basic geometric model of a component, this normalized model is linearly scaled up or down using the three parameters: length, bottom diameter and top diameter, corresponding to changing the dimensions of the box.

Due to the topological constraints, not all these parameters need to be specified in order to compute a basic geometric model. For instance, if the top coordinates of two consecutive segments of an axis are know, then the length and the primary direction of the second segment are known (assuming the basis of the second is connected to the top of the first). The function `PlantFrame` makes use of such information.

Using geometric features from the MTG coding file. The function automatically detects certain geometric features if they appear in particular feature columns of the coding file. These feature columns have reserved names: `Length`, `TopDiameter`, `BottomDiameter`, `Category`, `Alpha`, `L1`, `L2`, `L3`, `AA`, `BB`, `CC`, `XX`, `YY` and `ZZ`.

- Columns `Length`, `BottomDiameter`, `TopDiameter` enables us to define the geometric parameters of the basic geometric model associated with a component.
- Columns `XX`, `YY`, `ZZ` (or `L1`, `L2`, `L3` in a triangular reference system) define the top coordinates of the component.
- Columns `AA`, `BB`, `CC` define the orientation of the geometric model in space (both the primary direction and the second direction).
- Column `Alpha` defines the insertion angle of a component (angle between its principal direction and the horizontal plane)
- Column `Category` enables defines types of axes which enable us to tune the overall geometric form of the components of an axes using general shapes (Bezier curves), see `DressingData`.

Coordinate systems. Two types of coordinates systems are available. The type of coordinate system is automatically detected.

Cartesian coordinate system. A cartesian coordinate system is used if feature columns `XX`, `YY`, `ZZ` appear in the MTG coding file. The coordinates of each MTG component must be stored in the feature columns `XX`, `YY`, `ZZ`.

Coordinate system with triangular coordinates. A triangular system is made up of three points (A,B,C) which are defined by giving the three distances between each other. The distances are defined on one of the complexes of the vertex passed as an argument in the feature columns `DAB`, `DAC` et `DBC` of the coding file (respectively for the distances (A,B), (A,C) and (B,C)). Then, the coordinates (*i.e.* the distances to points A, B and C) of each MTG component must be stored respectively in the feature columns `L1`, `L2`, `L3`.

Using AML functions to define or to override values of the geometric parameters. When the user wants to define new values for the geometric parameters of the components of a MTG or alternatively wants to override them, (s)he can define AML functions that take a vertex of the considered scale as an argument and pass these functions as optional argument to function `PlantFrame`. In this case, each time a feature is required by function `PlantFrame` for a vertex, it is computed by applying the corresponding user-defined function to the vertex at hand. If the user-defined function returns `Undef`, then the value is looked after in the corresponding feature column of the coding file. If it is `Undef` again, then inference algorithms (see below) are applied when possible and default values (from the dressing file) are used otherwise.

Controlling the distance between plants. For a list of plants, the distance between individuals can be controlled using two optional arguments. `VoxelDist` enables the user to specify the distance between "virtual boxes" that contain the plants (one box is adjusted to one plant). This makes sure that no overlapping exists between the plant components in the 3D-space. `TrunkDist` can be used to regularly dispatch the plant trunks (like in a real field). Note that the value of `TrunkDist` cannot be 0 but can be as small as necessary (to supperpose exactly two trunks for instance).

Controlling the orientation of the components in space. If no explicit control of the orientation of the plant components is defined, the function `PlantFrame` makes use of the topological connections between components and of the definition or their top coordinates to infer both the location of a component (*i.e.* its origin in 3D space) and its primary direction (axial orientation). The second direction is arbitrary.
It is possible to override this default behaviour by explicitly specifying the 3D orientation of a component in space. This is done by defining the Euler angles of the component local coordinate system (the coordinate system of the box containing the basic geometric model of the component) with respect to the global reference system. The Euler angles can be specified in the coding file using the column features AA, BB and CC or by defining functions that can be passed as optional arguments (with options `AA`, `BB` and `CC`) to `PlantFrame`. In addition, if the user wants to actually use the Euler angles, (s)he must specify the MTG components for which the Euler angles must be used. This is done by defining a function that returns True for those components (and False for the others) and passing this function as an argument of PlantFrame using option `EulerAngles`.

Inference algorithms. Inferences algorithms are used to find missing geometric data if any. For all types of geometric attributes (`Length`, `TopDiameter`, `BottomDiameter`, etc.), the first step consists of verifying whether the feature corresponding to a geometric attribute is explicitly defined (in a feature column of the coding file or by a corresponding AML function). If it is found and when it is a feature of type `REAL`, it is *divided* by a scale factor (which is one by default) and which may be modified in the dressing file. This factor may be used to automatically adapt the units of two different attributes, *e.g.* for diameters in mm and length in cm, length can be automatically converted to mm by setting the length scaling factor to 10. If it is not found, a default inference is applied for certain features (explained below). If this inference does not produce a defined value, then a default value is used (see `DressingData`).

Inference algorithms are used for:

- `Category:` the category of a branch is defined by the category of its first vertex. If this value is not explicitly defined, the branching order is used (0 for the trunk, 1 for the branches, etc.). For a sympodial system (option `MODE->Sympodial`) the order is the apparent order of the axis.
- `Length`
  In some circumstances it is possible to infer the length of a component. Two types of MTGs are considered:

  - *MTGs without coordinates*. There are no columns `XX`, `YY`, `ZZ` (or `L1`, `L2`, `L3`). For these MTGs, the length of a component is estimated using the number of its sub-components. If the length of the sub-components is explicitly defined (either by a valid `Length` value in the corresponding feature column or by an optional function) then these values are summed up to compute the component's length. If not, the length of the sub-components of each sub-component is recursively computed in the same way. If, at some level of decomposition, a component cannot be further decomposed, a default length for that type of component is used (see DressingData).
  - *MTGs with coordinates*.
    1. If all the components of an axis have well defined top coordinates, the length of a component is computed as the distance between its top and the top of the preceding component. For the first component of an axis, the rule is slightly changed since its father component belongs to another axis. The inference algorithm looks for the smallest component (on the other axis but not necessarily at the same scale as the borne component) that bears this first component. If this component has well defined top coordinates, the length is defined as the distance between these topcoordinates and the first component top coordinates. Otherwise, the complex of this smallest bearing component is considered and its top coordinates are checked for definition. This process is recursively applied until a complex of the smallest bearing component is found with valid coordinates or until the scale of this complex corresponds to the scale of the borne component. In this case, the top coordinates of the bearing component are necessarily defined (since the bearing axis geometry has been computed at this scale before).
    2. There may be components of an axis that don't have valid coordinates. In this case, if no option is used, the length of a component is assumed to be homogeneous between two components with valid top coordinates.
       This default behaviour can be changed by using option `LengthAlgo`. `LengthAlgo->UseAxisIndexes` may be used to compute the length of a component in a way relative to the increase of index with respect to its father. For example if the MTG contains a sequence: U3<U8<U14<U15 and U3 and U15 only have valid top coordinates (the distance between these coordinates is denoted by *d*). If option `UseAxisIndexes` is not used the length (len) of the different components is:
       len(U8) = (8-3)*d*/(15-3) = 5*d*/12
       len(U14) = (14-8)*d*/(15-3) = 6*d*/12
       len(U15) = (15-14)*d*/(15-3) = *d*/12

Only the increment of index from one component to the next one is considered. If there is an index decrease in the sequence, like in U3<U8<U6<U14<U15, the decrease with respect with the previous component is not consider and is replaced by an increase with respect to index 0 (at the decreasing point). This would correspond to applying the previous strategy on the modified sequence U3<U8(<U0)<U6<U14<U15:

len(U8) = (8-3)*d*/(5+6+8+1-3) = 5*d*/14
len(U6) = (6-0)*d*/(5+6+8+1-3) = 6*d*/14
len(U14) = (8-6)*d*/(5+6+8+1-3) = 2*d*/14
len(U15) = (15-14)*d*/(5+6+8+1-3) = *d*/14

With this option value (`UseAxisIndexes`), the starting index of an axis is considered to be 0. For example, for the sequence U25+U27<U30, the length of the born components are (assuming U30 has well defined coordinates):

len(U27) = (27-0)*d*/(30-0) = 27*d*/30
len(U30) = (30-27)*d*/(30-0) = 3*d*/30

If one wants instead to use the index of the bearing element (at the same scale) as the starting index, one has to use the slightly different value `LengthAlgo->UseIndexes`. For the same sequence, the new computed length are:

len(U27) = (27-25)*d*/(30-25) = 2*d*/5
len(U30) = (30-27)*d*/(30-25) = 3*d*/5

`TopDiameter` and `BottomDiameter`: The diameter of components with undefined top or bottom diameter is linearly interpolated if the component is located between two components with defined diameters. At the beginning of a branch, a default diameter is assumed to be the one of the bearing component. At the end of a branch, if no component has a defined diameter, a default value is used (see `DressingData`).

BACKGROUND

[23]

SEE ALSO

`Plot`, `DressingData`, `VirtualPatterns`.

EXAMPLES

```
AML> g=MTG("an_mtg")
AML> fr=PlantFrame( 1,Scale-> 3)
AML> Plot(fr)
AML> cl(_x)=If Feature(_x,"DiamTop")!=Undef Then 2 Else 4
AML> Plot(fr, Color-> cl)
AML>
```

Plot

Plots a PLANTFRAME.

USAGE

```
Plot(p, Color-> colorfunc, Show-> showfunc)
Plot(p, Interpol-> interpolfunc, MaxThreshold -> 46)
Plot(p, Interpol-> interpolfunc, )
```

ARGUMENTS

p (PLANTFRAME) : plantframe

OPTIONAL ARGUMENTS

Color (FUNC) : Function defining a color code (Green, Red, …) for a vertex.
Show (FUNC) : Boolean function used to select the components that are visible in the plantframe.
Interpol (FUNC) : Function defining a value for a vertex whose relative position in the overall set of values is represented by a varying coloration.
MaxThreshold (REAL) : Maximum value for an interpolated value.
MinThreshold (REAL) : Minimum value for an interpolated value.
MediumThreshold (REAL) : Intermediate threshold for an interpolated value.
Class (FUNC) : Function used to (re)define the class of a vertex (which can be used to (re)defined the geometric model associated with a plant component). The association between a class and a geometric model is defined in the dressing file (see DRESSINGDATA).
DressingData (DRESSING_DATA) : data used for virtual elements and redefinition of classes.
VirtualLeaves (VIRTUAL_PATTERN) : define a set of virtual leaves and their geometric characteristics.
VirtualFruits (VIRTUAL_PATTERN) : define a set of virtual fruits and their geometric characteristics
VirtualFlowers (VIRTUAL_PATTERN) : define a set of virtual flowers and their geometric characteristics.

On IRIX only:
LineFile (STRING) : save the graphic representation into the file filename.lig (AMAP format compatible with "glance"). Two other files are generated by Plot, namely filename.dta and filename.inf. If option LineFile is not defined, temporary files line.lig, line.dta, line.inf are generated by Plot. Note that from glance, images plants can be saved using format SGI that can be read by standard image editors (e.g. *xv*, etc.).

RETURNED OBJECT

On a PLANTFRAME Plot returns a LINETREE object.

DESCRIPTION

The function `Plot()` displays a 3D graphic representation of the `PLANTFRAME` object (passed as an argument) on the screen. This representation is always possible for a valid plantframe.

Coordinate axes. When the plant appears on the screen, it is centered within the glance window. The *x* axis is directed to the right, the *y* axis is directed normally to the screen and backward, the *z* axis is directed bottom up.

Coloring. If no coloring option is used, the plantframe elements are displayed with a wood-like color. However, two options can be used to color the components of a plantframe.

- The option `Color` can be used to pass to the `Plot` command a function that defines for each component (or vertex) a color code (one of `Green, Red, Blue, LightBlue, Yellow, Violet, Black, White`).
- The option `Interpol` can be used to associate automatically a color with a component reflecting the intensity of a particular numeric attribute of this component. This attribute must be defined as an AML function that returns a numeric value for each component. The application of this function to the set of components in the plantframe defines a minimum and a maximum value. The interval beween these two values define the attribute domaine. Continuously varying colors, ranging from yellow to red are associated to increasing values of the attribute domain. The attribute domaine can be artificially reduced by using options `MinThreshold` and `MaxThreshold` (next versions of the software will correct the name typo). Every value of the attribute lower than `MinThreshold` will be green, while every value of the attribute greater than `MaxThreshold` will be violet. An intermediate value can also be defined using attribute `MediumThreshold`. If this is the case, the domain value is split into two parts: every value above the `MediumThreshold` is interpolated as explained above, i.e. with colors ranging from yellow to red. Every value below the `MediumThreshold` is interpolated as within a blue scheme, i.e. with colors ranging from dark blue to light blue. Values exactly at `MediumThreshold` are interpolated in brown.

Changing the displayed shape of a component. The basic geometric models used to represent plant component shapes correspond to symbols (see `DressingData`) that, by default, are associated with the class of the components. For example, an internode is represented in the MTG by a class, say `I`. In the dressing file, geometric models can be associated with classes (represented by letters. By default, the geometric model of a component will be the geometric model associated with the symbol corresponding to its class. And again by default, unless specified differently in the dressing file, all symbols are associated with a cone frustum of radius 1 and of height 1 (On IRIX this corresponds to the geometric model defined in the SMB file `nentn105.smb`, i.e., a cylinder represented as a polygonal volume with 5 vertical facets, with height 1 and width 1). Option `Class` can be used to associated a symbol (and thus a basic geometric model) with a component, independently of its class in the MTG. For each component, this function must return a character (`CHAR`) which defines the new symbol associated with this component. The connection between a class (represented as a character) and the geometric model is defined in the dressing file (see

DressingData). This is the reason why the dressing data that should be used to make this connection has to be passed on to Plot simultaneously using option DressingData.

Hiding components. It is sometimes useful not to display all the components of a plantframe. In order to filter the set of components that should be displayed, the user may defined a boolean function (*i.e.* a predicate) that returns True for components that should be displayed and False for others. This function can be passed on to Plot using option Show.

Adding virtual elements. Virtual elements are components that are not defined in the MTG which is used to build up a particular plantframe. The user for example may want to add leaves to the plant at some specific locations, and with specific geometry. It is possible to add virtual elements to the plantframe argument. The user first has to build an object of type VIRTUAL_PATTERN, and then pass on this object to the Plot command with either options VirtualLeaves, VirtualFlowers, VirtualFruits (see VirtualPattern)

On IRIX only:
Generated files. The function generates a binary file containing this 3D representation and then calls the glance program (included in the AMAPmod distribution) that displays the data with 3D capabilities. By default the generated file is created in the current directory and is called line.lig. This default location and name of the generated file(s) can be changed by using option LineFile and giving a different path name. Note that the name of the file does not include the .lig extension. It is automatically added by AMAPmod (this may change in future versions).

Two additional files (required by glance) are generated with line.lig: line.dta and line.inf. line.dta is a text file that contains the description of the symbols (basic geometric models) used in the plantframe and their associated colors. Colors are defined in RGB format. If necessary, line.dta could be edited with a simple text editor in order to change the values of the colors of the different symbols.

DETAILS

On IRIX only:
Filenames and glance. For historical reasons, AMAPmod uses a different convention from managing file names that the one used by glance. In AMAPmod, file names are always explicitly given by their UNIX absolute or relative path. This defines their location in the directory tree. In glance, the location of files is defined independently of their names: a file .cfg should be either in the current directory or in your home directory and defines all the locations where glance expects to find different types of file. When used with AMAPmod, the .cfg file should be set so that the following locations are correct:

```
SYMBOLES = /…/my_symbol_directory
POLYGONE = /…/my_polygon_directory
LIGNE = .
```

The first two directories are distributed with the AMAPmod distribution and should be set accordingly. The last line defines the current directory as being the directory for looking for files *.lig. Note that if you generate with Plot() files that are not in

your current directory, you should set accordingly the LIGNE parameter of the .cfg file so that glance can find this file.

BACKGROUND

MTGs

SEE ALSO

MTG, ARRAY, Scale, Class, Index.

EXAMPLES

```
AML> cl(_x)=If Feature(_x,"DiamTop")!=Undef Then 2 Else 4
AML> Plot(fr, Color-> cl)

AML> Plot(PlantFrame, Interpol-> f, MaxTreshold-> x1, \
AML> MinTreshold-> x2,Medium-> x3,LineFile-> "name")
```

Plot (LINETREE)

Plots a LINETREE.

USAGE

```
Plot(lt, Geometry-> geomfunc)
Plot(lt, Appearance-> appfunc)
Plot(lt, Quotient-> quotientfunc, )
```

ARGUMENTS

p (LINETREE) : plantframe

OPTIONAL ARGUMENTS

Geometry (FUNC) : Function defining a geometry for a vertex as a GEOM object. The function returns a string referring to an object previously loaded using the dressing file keyword "Geometry" in the dressong file.

Appearance (FUNC) : Function defining an appearance for a vertex as an APPEARANCE object. The function returns a string referring to an object previously loaded using the dressing file keyword "Appearance" in the dressong file.

Quotient (FUNC) : Predicate on vertex that defines a set of vertices which should be considered as root vertices of new macro-constituents (not necessarily defined in the original MTG). Then the Geometric models are associated with these new macroscopic vertex by using between-scale constraints. More information on this possibility will be available in a future release.

RETURNED OBJECT

On a PLANTFRAME Plot returns a LINETREE object.

DESCRIPTION

BACKGROUND

MTGs

SEE ALSO

MTG, ARRAY, Scale, Class, Index.

EXAMPLES

```
AML> cl(_x)=If Feature(_x,"DiamTop")!=Undef Then RED2 Else
GREEN3
AML> lt = Plot(fr)
AML> Plot(lt, Appearance-> cl)
```

Predecessor

Father of a vertex connected to it by a '<' edge

USAGE

```
Predecessor(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

RestrictedTo (STRING): *cf.* Father
ContainedIn (VTX): *cf.* Father

RETURNED OBJECT

VTX

DESCRIPTION

This function is equivalent to `Father(v, EdgeType-> `<`)`. It thus returns the father (at the same scale) of the argument if it is located in the same botanical. If it does not exist, Undef is returned.

BACKGROUND

MTGs

SEE ALSO

MTG, Father, Successor.

EXAMPLES

```
AML> Predecessor(v)
     <VTX> 7
AML> Father(v, EdgeType-> `+`)
     Undef
AML> Father(v, EdgeType-> `<`)
     <VTX> 7
```

PreviousDate

Previous date at which a vertex has been observed before a specified date

USAGE

```
PreviousDate(v, d)
```

ARGUMENTS

v (VTX) : vertex of the active MTG.
d (DATE) : departure date.

RETURNED OBJECT

DATE

DESCRIPTION

Returns the first observation date at which the vertex has been observed starting at date d and proceeding backward in time. Undef is returned if it does not exists.

BACKGROUND

Dynamic MTGs.

SEE ALSO

MTG, DateSample, FirstDefinedFeature, LastDefinedFeature, NextDate.

Rank

Rank of one vertex with respect to another one

USAGE

```
Rank(v1)
Rank(v1, v2)
```

ARGUMENTS

`v1` (`VTX`) : vertex of the active MTG
`v2` (`VTX`) : vertex of the active MTG

RETURNED OBJECT

`INT`. If `v1` is not an ancestor of `v2` (or vice versa) within the same botanical axis, or if `v1` and `v2` are not defined at the same scale, an error value `Undef` is returned.

DESCRIPTION

This function returns the number of consecutive '<'-type edges between two components, at the same scale, and does not take into account the order of vertices `v1` and `v2`. The result is a non negative integer.

BACKGROUND

MTGs

SEE ALSO

`MTG`, `Order`, `Height`, `EdgeType`, `AlgRank`, `AlgHeight`, `AlgOrder`.

RelBottomCoord

Bottom coordinates of the geometric model of a component expressed in the global reference system translated at the origin of the plant (basis of the trunk).

USAGE

```
RelBottomCoord(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the bottom of the box containing the geometric model of a plant component in the translated coordinate system. The result is an array of 3 reals.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, RelTopCoord, TopBottomCoord, BottomCoord, PDir.

RelTopCoord

Top coordinates of the geometric model of a component expressed in the global reference system translated at the origin of the plant (basis of the trunk).

USAGE

```
RelTopCoord(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the top of the box containing the geometric model of a plant component in the translated coordinate system. The result is an array of 3 reals.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, RelBottomCoord, TopBottomCoord, BottomCoord, PDir.

Root

Root of the branching system containing a vertex

USAGE

```
Root(v)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG

OPTIONAL ARGUMENTS

`RestrictedTo` (`STRING`): *cf.* `Father`
`ContainedIn` (`VTX`): *cf.* `Father`

RETURNED OBJECT

`VTX`

DESCRIPTION

This function is equivalent to `Ancestors(v, EdgeType-> `<`)@-1`. It thus returns the root of the branching system containing the argument. This function never returns `Undef`.

BACKGROUND

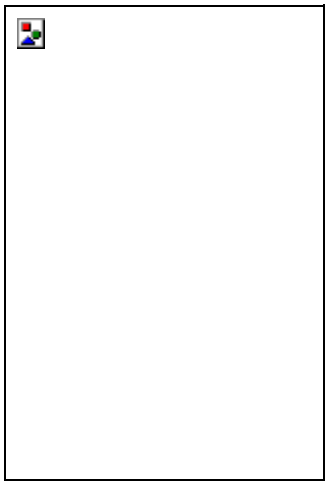
MTGs

SEE ALSO

`MTG`, `Extremities`.

EXAMPLES

```
AML> Ancestors(v) # set of ancestors of v
     <ARRAY(VTX)>[102,78,35,33,24,12]
AML> Root(v) # root of the branching system containing v
     <VTX>12
```



◎ white vertices with double circle are argument(s) of the function
● Black vertices are vertices returned by the function

Save(LINETREE)

Save a LineTree object in different formats.

USAGE

```
Save(obj, filename)
Save(obj, file, Format-> Binary)
```

ARGUMENTS

`obj` (`LINETREE`) : the object can have any type.
`filename` (`STRING`): identifies the file in which the object `obj` is saved.

OPTIONAL ARGUMENTS

Optional arguments can exist depending on the type of the first argument. Option Format can be used to select an output format. Available formats are: GEOM (ascii format for the GEOM library), AMAP (for ascendant compatibility with AMAP software and PovRay, (VGStar and VRML formats will be available in the next release).

RETURNED OBJECT

No value is returned.

DESCRIPTION

Save an object to a file in different formats.

SEE ALSO

`Load`, `Object` constructors, `Display`.

EXAMPLES

```
AML> #Saving a LineTree
AML> pf = PlantFrame(1,Scale->3)     # builds a PlantFrame.
AML> lt = Plot(pf)           # builds a PlantFrame.
AML> Save(lt, "file1.lnt", Format->GEOM)
AML> # Saves the LineTree to file named file1.lnt in ascii
```

Scale

Scale of a vertex

USAGE

```
Scale(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG
v (PLANTFRAME) : PlantFrame computed on the active MTG
v (LINETREE) : LineTree computed on a PlantFrame respresenting the active MTG

RETURNED OBJECT

INT

DESCRIPTION

Returns the scale at which is defined the argument.

BACKGROUND

MTGs

SEE ALSO

MTG, ClassScale, Class, Index.

SDir

Secundary direction of the geometric model of a vertex

USAGE

```
SDir(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the secundary direction (a direction lying in the plane perpendicular to the principal (or primary) direction) of the box containing the geometric model of a plant component in the global coordinate system. The result is an array of 3 reals.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, RelTopCoord, RelBottomCoord, TopCoord, BottomCoord, PDir.

Sons

Set of vertices borne or preceded by a vertex

USAGE

```
Sons(v)
Sons(v, EdgeType-> `+`)
Sons(v, Scale-> 3)
```

ARGUMENTS

`v` (`VTX`) : vertex of the active MTG

OPTIONAL ARGUMENTS

`RestrictedTo` (`STRING`): *cf.* `Father`
`ContainedIn` (`VTX`): *cf.* `Father`
`EdgeType` (`CHAR`) : filter on the type of sons.
`Scale` (`INT`) : set the scale at which sons are considered.

RETURNED OBJECT

`ARRAY(VTX)`

DESCRIPTION

The set of sons of a given vertex is returned as an array of vertices. The order of the vertices in the array is not significant. The array can be empty if there are no son vertices.

DETAILS

When the option `EdgeType` is applied, the function returns the set of sons that are connected to the argument with the specified type of relation. Note that `Sons(v, EdgeType-> `<`)` is not equivalent to `Successor(v)`. The first function returns an array of vertices while the second function returns a vertex.

The returned vertices have the same scale as the argument. However, coarser or finer vertices can be obtained by specifying the the optional argument `Scale` at which the sons are considered.

BACKGROUND

MTGs

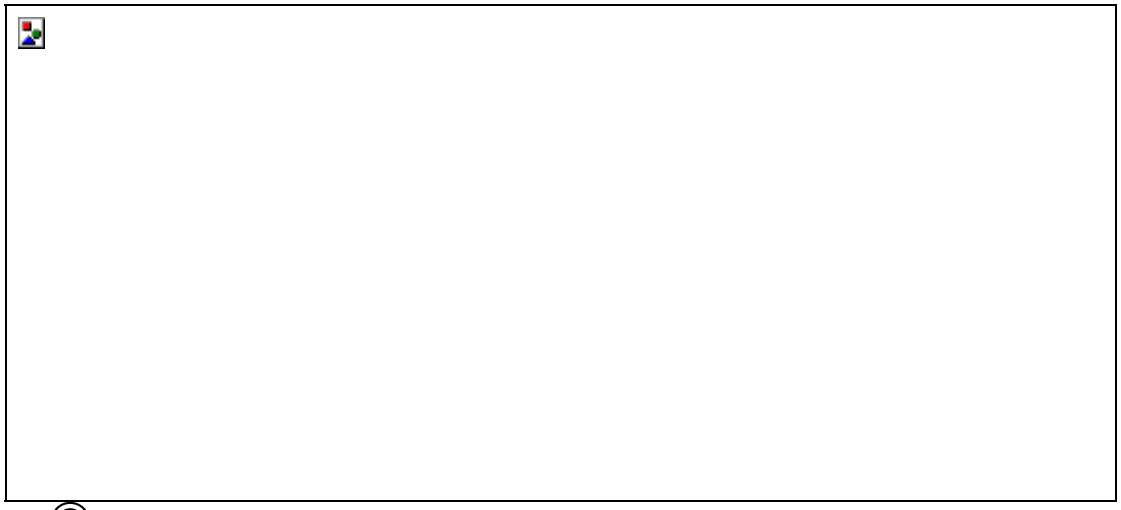
SEE ALSO

`MTG`, `Father`, `Successor`, `Descendant`.

EXAMPLES

```
AML> Sons(v) #
```

```
     <ARRAY(VTX)>[3,45,47,78,102]
AML> Sons(v, EdgeType-> `+`) # set of vertices borne by v
     <ARRAY(VTX)>[3,45,47,102]
AML> Sons(v, EdgeType-> `<`) # set of successors of v on the
same axis
     <ARRAY(VTX)>[78]
```



◎ white vertices with double circle are argument(s) of the function

● Black vertices are vertices returned by the function

Successor

Vertex that is connected to a given vertex by a '<' edge type (i.e. in the same botanical axis).

USAGE

```
Successor(v)
```

ARGUMENTS

v (VTX) : vertex of the active MTG

OPTIONAL ARGUMENTS

RestrictedTo (STRING): *cf.* Father
ContainedIn (VTX): *cf.* Father

RETURNED OBJECT

VTX

DESCRIPTION

This function is equivalent to `Sons(v, EdgeType-> `<`)@1`. It returns the vertex that is connected to a given vertex by a '<' edge type (i.e. in the same botanical axis). If many such vertices exist, an arbitrary one is returned by the function. If no such vertex exists, Undef is returned.

BACKGROUND

MTGs

SEE ALSO

MTG, Sons, Predecessor.

EXAMPLES

```
AML> Sons(v) #
     <ARRAY(VTX)>[3,45,47,78,102]
AML> Sons(v, EdgeType-> `+`) # set of vertices borne by v
     <ARRAY(VTX)>[3,45,47,102]
AML> Sons(v, EdgeType-> `<`) # set of successors of v
     <ARRAY(VTX)>[78]
AML> Successor(v)
     <VTX>78
```

TopCoord

Top coordinates of the geometric model of a component

USAGE

```
TopCoord(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

ARRAY(REAL)

DESCRIPTION

This function returns the 3D-coordinates of the top of the box containing the geometric model of a plant component. The result is an array of 3 reals.

BACKGROUND

MTGs

SEE ALSO

MTG, PlantFrame, BottomCoord.

TopDiameter

Top diameter of the geometric model of a vertex

USAGE

```
TopDiameter(p, v)
```

ARGUMENTS

p (PLANTFRAME) : plantframe containing the geometric representation of v.
v (VTX) : vertex of the active MTG

RETURNED OBJECT

REAL

DESCRIPTION

This function returns the top diameter of the geometric model of a vertex. Note that this diameter might note be defined in the MTG coding file since it may result from an inference process in the PlantFrame function.

BACKGROUND

MTGs

SEE ALSO

MTG, BottomDiameter, Length, Alpha, Beta.

TreeMatching

Comparison of rooted tree graphs (representing branching systems)

USAGE

```
TreeMatching(va)
TreeMatching(va, LocalDist-> by_topology)
TreeMatching(va, LocalDist-> by_components)
TreeMatching(va, LocalDist-> by_weight,
                 FuncList-> fl,VectorDistance->vec)
TreeMatching(va, FuncList-> fl,VectorDistance->vec)
```

ARGUMENTS

va (ARRAY(VTX)) : Array of vertices at a same scale defining a set of branching system.

OPTIONAL ARGUMENTS

LocalDist (STRING) : type of local distance used during the matching algorithm.
- by_weights : weighted local distance
- by_topology : topological local distance
- by_components : algorithm preserving a minimum number of connected components during the matching algorithm.

FuncList (ARRAY(FUNC)) : function array defining vertex arguments used by the weighted local distance

VectorDistance(VECTOR_DISTANCE) : Argument used in order to define the weights, the argument type (numeric or symbolic) and the type of norm (absolute value or quadratic), used by the weighted local distance.

RETURNED OBJECT

TREEMATCHING

DESCRIPTION

Returns a distance between two rooted tree graphs and a optimal valid matching.
The user has first to define a local distance between elementary entities. This distance is defined using either binary symbolic or real attributes of entities. Then the comparison algorithm can be used in two different contexts: either to assess the architecture variability of a set of plants or to carry out a piece by piece comparison between two plants.

DETAILS

The computational time of a comparison between two tree graphs is approximately proportional to the square of the total number of vertices in the compared branching systems.

BACKGROUND

Method for comparing unordered rooted tree graphs [51; 14].

SEE ALSO

`MTG`, `Plot`, `MatchingExtract`.

EXAMPLES

```
AML> g = MTG(”un_mtg”)
AML> plants = VtxList(Scale->1)
AML> # root extraction of plants at scale 2
AML> AML> roots = Foreach _p In plants:Components(_p,Scale->2)
AML> # Topological matching between plants
AML> topo_match = TreeMatching(roots)
AML> # Array of functions
AML> funs = [Order,Class]
<ARRAY(FUNCTION)>
AML> # Vector Distance which defines 2 attributes (numeric and
AML> # symbolic) with same weights .5
AML> v = VectorDistance(.5,“NUMERIC”,.5,”SYMBOLIC”)
AML> # Weighted matching between plant 1 and 2
AML> weighted_match = TreeMatching([roots@1,roots@2],\
AML> FuncList->funs,VectorDistance->v)
```

Trunk

Array of vertices constituting the bearing botanical axis of a branching system

USAGE

```
Trunk(v)
Trunk(v, Scale-> s)
```

ARGUMENTS

v (VTX) : Vertex of the active MTG

OPTIONAL ARGUMENTS

Scale (STRING): scale at which the axis components are required.

RETURNED OBJECT

ARRAY(VTX)

DESCRIPTION

Trunk returns the array of vertices representing the botanical axis defined as the bearing axis of the whole branching system defined by v. The optional argument enables the user to choose the scale at which the trunk should be detailed.

BACKGROUND

MTGs

SEE ALSO

MTG, Path, Ancestors, Axis.

VirtualPattern

Object containing the specification to add (virtual) components to an existing PLANTFRAME.

USAGE

```
VirtualPattern(“Leaf”, Symbol->symb_func, WhorlSize ->ws_func)
VirtualPattern(“DigitizedLeaf”,
                Color->color_func,
                XX->xx,YY->yy,ZZ->zz,AA->aa,BB->bb,CC->cc)
VirtualPattern(“Fruit”, TopDiameter->dia_func, BottomDiameter ->dia_func)
VirtualPattern(“Flower”,
               Length->len_func,
              Alpha->alpha_func,Beta->beta_func)
```

ARGUMENTS

s (STRING): String defining the type of virtual pattern used (can be one of “Leaf”, “DigitizedLeaf”, “Fruit”, “Flower”). The difference between these options is related to the way virtual elements are oriented by default in space. Leaves have a default insertion angle with respect to their bearer of 30 degrees

OPTIONAL ARGUMENTS

PatternNumber (FUNC): AML function defining the number of virtual elements that should be added to a specific component (passed as an argument).

WhorlSize (FUNC): AML function defining the number of virtual elements that should be located at each virtual node of a specific component (passed as an argument).

Class (FUNC): AML function defining the symbols of the virtual patterns associated with each plant component. This symbol can be used to attach a particular geometric model to the virtual elements borne by a specific component (passed as an argument).

Color (FUNC): AML function defining the color of the virtual elements borne by a specific component (passed as an argument).

TopDiameter (FUNC): AML function defining the top diameter of the virtual elements borne by a specific component (passed as an argument).

BottomDiameter (FUNC) : AML function defining the bottom diameter of the virtual elements borne by a specific component (passed as an argument).

Length (FUNC): AML function defining the length of the virtual elements borne by a specific component (passed as an argument).

Alpha (FUNC): AML function defining the insertion angle of the virtual elements borne by a specific component (passed as an argument). This angle is assumed to be defined in degrees. It is relative to the principal direction of the bearing component.

Beta (FUNC): AML function defining the azimuth of the virtual elements borne by a specific component (passed as an argument). This angle is assumed to be defined in degrees. It is relative to the secundary direction of the bearing component.

XX (FUNC): AML function defining the x-coordinate of the virtual element borne by a specific component (should be used only when only one virtual element at most is borne by each component).

YY (FUNC): AML function defining the y-coordinate of the virtual element borne by a specific component (should be used only when only one virtual element at most is borne by each component).

ZZ (FUNC): AML function defining the z-coordinate of the virtual element borne by a specific component (should be used only when only one virtual element at most is borne by each component).

AA (FUNC): AML function defining the first euler angle (azimuth) of a virtual element borne by a specific component (should be used only when only one virtual element at most is borne by each component). These angles are assumed to be defined in radians. They are absolute (expressed in a global coordinate system).

BB (FUNC): AML function defining the second euler angle (elevation) of a virtual element borne by a specific component (should be used only when only one virtual element at most is borne by each component). These angles are assumed to be defined in radians. They are absolute (expressed in a global coordinate system). Warning, to go upwards, use negative angles.

CC (FUNC): AML function defining the third euler angle (twist) of a virtual element borne by a specific component (should be used only when only one virtual element at most is borne by each component). These angles are assumed to be defined in radians. They are absolute (expressed in a global coordinate system).

RETURNED OBJECT

If the geometric reconstruction is successfully carried out, a VIRTUAL_PATTERN object is returned. Otherwise an error is returned.

DESCRIPTION

Virtual elements are components that are not defined in the MTG which is used to build up a particular plantframe. The user for example may want to add leaves to the plant at some specific locations, and with specific geometry whereas actually no leaves were measured in the field. It is possible to add virtual elements to the plantframe argument. The user first has to build an object of type VIRTUAL_PATTERN, and then pass on this object to the Plot command with either options VirtualLeaves (if the virtual pattern was built with argument "Leaf" of "DigitizedLeaf"), VirtualFlowers (for argument "Flower"), VirtualFruits (for argument "Fruit"). The function VirtualPattern does not create directly virtual elements. It creates an object of type VirtualPattern that just contains information defining how to compute virtual elements for a new plantframe. Virtual elements are actually computed by the Plot command if an option VirtualLeaves, VirtualFlowers or VirtualFruits is defined.

Controlling the number of virtual elements per component. The total number of virtual elements that must be associated with a component (vertex) of a plantframe is defined by a function that returns an integer for any valid vertex. This function is defined using option PatternNumber. Note that this function can define 0 virtual element for a vertex, which can be used to filter virtual element based on different criteria.

Controlling the number of virtual elements per virtual node of a component. If more than one virtual element is defined (see option `PatternNumber`) virtual elements are assumed to be arranged along their bearer by whorls at regularly distant virtual nodes between the top and the bottom of the component. A total number of elements per virtual nodes can also be defined with option `WhorlSize`. Therefore, the number of virtual nodes for a component is the number of virtual element divided by the number of virtual element per virtual node. For example, a component that is associated with 12 virtual leaves which are distributed per whorls of size 3, has 4 virtual nodes.

Controlling the position of virtual elements.

1. For `Leaf`, `Fruit` and `Flower`, the origin of the virtual element is determined by its position on the bearing component. As explained above, this depends on the number of virtual elements on a particular component and on the whorl size. In addition, the exact origin is translated to the periphery of the bearing component.
2. For `DigitizedLeaf`, the position of virtual elements is defined by the coordinates of *their basis*. This position is defined by functions defined using options `XX`, `YY` and `ZZ` (WARNING: this convention is different from that used for normal MTG components where `XX`, `YY` and `ZZ` designate the coordinates *of the top* of the component). These options should be used when there is at most one virtual element per component (see option `PatternNumber`), otherwise all the virtual elements will share the same position.

Controlling the orientation of virtual elements.

1. For `Leaf`, `Fruit` and `Flower`, the orientation of a virtual element can be only partially controlled. Two angles can be used to define the principal direction of the virtual element associated with a component in spherical coordinates, using options `Alpha` and `Beta`. `Alpha` defines the angle of the principal direction of the virtual element with the plane normal to the principal direction of the component. For `Leaf` and `Flowers,` it corresponds to the "insertion angle" of the virtual element on its bearing component (or more precisely $\pi/2 - \iota$, where ι the insertion angle). For `flowers`, the principal direction is constant, vertical and and oriented towards the soil. `Beta` corresponds to the relative azimuth of the principal direction of the virtual element with respect to the local reference system of its bearing component. This local reference system is defined by the secondary, ternary and principal directions of the bearing component, in this order.
2. For `DigitizedLeaf`, the orientation of a virtual element can be defined by Euler angles. Three Euler angles can express the exact orientation of the virtual element with respect to the global reference system. These angles are automatically generated by digitizing devices like Polhemus. They should be stored in feature columns `AA`, `BB`, `CC` of the MTG (for example on the bearing component of each virtual leaf). They can also be defined by AML functions and passed on to the `VirtualPattern` function using option `AA`, `BB`, `CC`. (WARNING: the unit of these angles is the radian. This is contrary to the convention used in other primitives of AMAPmod, that consider that angles are usually defined in degrees. This will be fixed in future releases of AMAPmod).

Controlling the geometry of virtual elements. Each virtual element can be associated with a symbol that represent its geometric model. Symbols are associated with geometric models using the dressing file (see `DressingData` and `Plot`). The general geometric parameter `Length`, `TopDiameter` and `BottomDiameter` are used to control the dimensions of the geometric model which is associated with a given component. These parameters can be defined independently for each component by using corresponding AML functions using options `Length, TopDiameter, BottomDiameter.`

Controlling the color of virtual elements. Virtual elements are displayed by default in `Green`. This default color can be changed by defining a function that returns a color code for a given vertex. All the virtual elements borne by a certain component have the same color, defined by the result of the coloring function applied to this component.

BACKGROUND

MTGs.

SEE ALSO

`Plot`, `DressingData`, `PlantFrame`.

EXAMPLES

```
AML> g=MTG("un_mtg")
AML> dr=DressingData("dressingfile")
AML> fr=PlantFrame( 1,Scale-> 3)
AML> pn(_x)=1; nb(_x)=2
AML> l1=VirtualPattern("Leaf",WhorlSize->nb,PatternNumber->pn)
AML> Plot(fr, VirtualLeaves-> l1,DressingData->dr)
```

VtxList

Array of vertices contained in a MTG

USAGE

```
VtxList()
VtxList(Scale-> 2)
```

ARGUMENTS

None

OPTIONAL ARGUMENTS

`Scale` (`INT`) : used to select components at a particular scale.

RETURNED OBJECT

`ARRAY(VTX)`

DESCRIPTION

The set of all vertices in the MTG is returned as an array. Vertices from all scales are returned if no option is used. The order of the elements in this array is not significant.

BACKGROUND

MTGs

SEE ALSO

`MTG, ARRAY, Scale, Class, Index.`

3 THE STAT MODULE

3.1 List of AML functions of the STAT module

3.2 List by categories of the AML functions of the STAT module

- **Input/output functions:**

`Compound`: `COMPOUND` constructor,
`Convolution`: `CONVOLUTION` constructor,
`Distribution`: `DISTRIBUTION` constructor,
`HiddenMarkov`: `HIDDEN_MARKOV` constructor,
`HiddenSemiMarkov:` `HIDDEN_SEMI-MARKOV` constructor,
`Histogram`: `HISTOGRAM` constructor,
`Markov`: `MARKOV` constructor,
`Mixture`: `MIXTURE` constructor,
`Renewal`: `RENEWAL` constructor,
`SemiMarkov`: `SEMI-MARKOV` constructor,
`Sequences`: `SEQUENCES` constructor,
`TimeEvents`: `TIME_EVENTS` constructor,
`TopParameters`: `TOP_PARAMETERS` constructor,
`Tops`: `TOPS` constructor,
`VectorDistance`: `VECTOR_DISTANCE` constructor,
`Vectors`: `VECTORS` constructor,

`Load`: restoration of an object saved as a binary file,

`Display`: ASCII output,

`Plot:` graphical output,

`Print`: ASCII print,

`Save`: save in a file.

- **Functions of data manipulation:**

`Merge`: merging of objects of the same 'data' type or merging of sample correlation functions,

`Cluster`: clustering of values,

`Shift`: shifting of values,

`Transcode`: transcoding of values,

`SelectIndividual`: selection of individuals,

`ValueSelect`: selection of individuals according to the values taken by a variable.

`MergeVariable`: merging of variables,

`SelectVariable`: selection of variables.

set of count data of type {time interval between two observation dates, number of events occurring between these two observation dates}:

`NbEventSelect`: selection of data item according to a number of events criterion,

`TimeScaling`: change of the time unit,

`TimeSelect`: selection of data item according to a length of the observation period criterion.

set of sequences:

`AddAbsorbingRun`: addition of a run of absorbing vectors at the end of sequences,

`Cumulate`: sum of successive values along sequences,

`Difference`: first-order differencing of sequences,

`IndexExtract`: extraction of sub-sequences corresponding to a range of index parameters,

`LengthSelect`: selection of sequences according to a length criterion,

`MovingAverage`: extraction of trends or residuals using a symmetric smoothing filter,

`RecurrenceTimeSequences`: computation of recurrence time sequences for a given value,

`RemoveRun`: removal of the first or last run of a given value (for a given variable) in a sequence,

`Reverse`: reversing of sequences or 'tops',

`SegmentationExtract`: extraction of sub-sequences by segmentation,

`VariableScaling`: change of the unit of a variable.

set of 'tops':

`RemoveApicalInternodes`: removal of the apical internodes of the parent shoot of a 'top'.

dissimilarity matrix:

`Symmetrize`: symmetrization of a dissimilarity matrix.

- **Statistical functions:**

Clustering: application of clustering methods (either partitioning methods or hierarchical methods) to dissimilarity matrices between patterns,

Compare: comparison of frequency distributions, vectors, sequences or Markovian models,

ComparisonTest: test of comparison of frequency distributions,

ComputeCorrelation: computation of sample autocorrelation or cross-correlation functions,

ComputePartialAutoCorrelation: computation of sample partial autocorrelation functions,

ComputeRankCorrelation: computation of a rank correlation matrix,

ComputeStateSequences: computation of the optimal state sequences corresponding to the observed sequences using a hidden Markov chain or a hidden semi-Markov chain,

ComputeWhiteNoiseCorrelation: computation of the autocorrelation or cross-correlation function induced on a white noise sequence by filtering,

ContingencyTable: computation of a contingency table,

Estimate: estimation of model parameters from data sample,

Fit: fit of a frequency distribution by a theoretical distribution,

ModelSelectionTest: test for selecting the order of a Markov chain or an aggregation of states of a Markov chain,

Regression: simple (either linear or nonparametric) regression,

Simulate: generation of random samples from models,

VarianceAnalysis: one-way variance analysis.

- **Miscellaneous functions:**

ComputeSelfTransition: computation of the self-transition probabilities as a function of the index parameter from discrete sequences,

ExtractData: extraction of the 'data' part of an object of type 'model',

ExtractDistribution: extraction of a distribution from an object of type 'model',

ExtractHistogram: extraction of a frequency distribution from an object of type 'data',

ExtractVectors: extraction of vectors from global characteristics of sequences (length or counting characteristics),

ToDistanceMatrix: cast of an object of type CLUSTERS into an object of type DISTANCE_MATRIX,

ToDistribution: cast of an object of type HISTOGRAM into an object of type DISTRIBUTION,

ToHistogram: cast of an object of type DISTRIBUTION into an object of type HISTOGRAM,

TransformPosition: discretization of inter-position intervals.

3.3 List by type of the AML functions of the STAT module

type CLUSTERS

function returning an object of type CLUSTERS:

Load,
Clustering.

function taking as argument an object of type CLUSTERS:

Display, Plot, Print, Save,
ToDistanceMatrix.

type COMPOUND

function returning an object of type COMPOUND:
Compound, Load,
Estimate (1).

function taking as argument an object of type COMPOUND:
Display, Plot, Print, Save,
Compound, Convolution, Mixture, Renewal,
Fit, Simulate (1),
ExtractData, ExtractDistribution.

type COMPOUND_DATA

function returning an object of type COMPOUND_DATA:
Load,
Simulate (1),
ExtractData.

function taking as argument an object of type COMPOUND_DATA:
TimeEvents, Display, Plot, Print, Save,
Cluster, Merge, Shift, Transcode, ValueSelect,
Compare (1)(5), ComparisonTest, Estimate (1), Fit, Simulate (2)(3),
ExtractHistogram.

type CONVOLUTION

function returning an object of type CONVOLUTION:
Convolution, Load,
Estimate (1).

function taking as argument an object of type CONVOLUTION:
Display, Plot, Print, Save,
Compound, Convolution, Mixture, Renewal,
Fit, Simulate (1),
ExtractData, ExtractDistribution.

type CONVOLUTION_DATA

function returning an object of type CONVOLUTION_DATA:
Load,
Simulate (1),
ExtractData.

function taking as argument an object of type CONVOLUTION_DATA:
TimeEvents, Display, Plot, Print, Save,
Cluster, Merge, Shift, Transcode, ValueSelect,
Compare (1)(5), ComparisonTest, Estimate (1), Fit, Simulate (2)(3),
ExtractHistogram.

type CORRELATION

function returning an object of type CORRELATION:

Load,
Merge,
ComputeCorrelation, ComputePartialAutoCorrelation.

function taking as argument an object of type CORRELATION*:*

Display, Plot, Print, Save,
Merge,
ComputeWhiteNoiseCorrelation.

type DISCRETE_SEQUENCES

function returning an object of type DISCRETE_SEQUENCES:

Sequences, Load,
AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract,
LengthSelect, Merge, MergeVariable, MovingAverage,
RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract,
SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect,
TransformPosition.

function taking as argument an object of type DISCRETE_SEQUENCES:

Display, Plot, Print, Save,
AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract,
LengthSelect, Merge, MergeVariable, MovingAverage,
RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract,
SelectIndividual, SelectVariable, Transcode, ValueSelect,
VariableScaling,
Compare (3)(4), ComputeCorrelation, ComputePartialAutoCorrelation,
ComputeStateSequences, ComputeWhiteNoiseCorrelation, Estimate (3),
ModelSelectionTest, Simulate (3),
ComputeSelfTransition, ExtractHistogram, ExtractVectors.

type DISTANCE_MATRIX

function returning an object of type DISTANCE_MATRIX:

Load,
SelectIndividual, Symmetrize,
Compare (2)(3)(5),
ToDistanceMatrix.

function taking as argument an object of type DISTANCE_MATRIX:

Display, Plot, Print, Save,
SelectIndividual, Symmetrize,
Clustering.

type DISTRIBUTION

function returning an object of type DISTRIBUTION:

Distribution, Load,
Estimate (1),
ExtractDistribution, ToDistribution.

function taking as argument an object of type DISTRIBUTION:

Display, Plot, Print, Save,

Compound, Convolution, Mixture, Renewal,
Estimate (1), Fit, Simulate (1),
ToHistogram.

type HIDDEN_MARKOV

function returning an object of type HIDDEN_MARKOV:
HiddenMarkov, Load,
Estimate (3).

function taking as argument an object of type HIDDEN_MARKOV:
Display, Plot, Print, Save,
Compare (4)(5), ComputeStateSequences, Estimate (3), Simulate (3),
ExtractData, ExtractDistribution.

type HIDDEN_SEMI-MARKOV

function returning an object of type HIDDEN_SEMI-MARKOV:
HiddenSemiMarkov, Load,
Estimate (3).

function taking as argument an object of type HIDDEN_SEMI-MARKOV:
Display, Plot, Print, Save,
Compare (4)(5), ComputeStateSequences, Estimate (3), Simulate (3),
ExtractData, ExtractDistribution.

type HISTOGRAM

function returning an object of type HISTOGRAM:
Histogram, Load,
Cluster, Merge, Shift, ValueSelect, Transcode,
Simulate (1),
ExtractHistogram, ToHistogram.

function taking as argument an object of type HISTOGRAM:
TimeEvents, Display, Plot, Print, Save,
Cluster, Merge, Shift, ValueSelect, Transcode,
Compare (1)(5), ComparisonTest, Estimate (1), Fit, Simulate (2)(3),
ToDistribution.

type MARKOV

function returning an object of type MARKOV:
Markov, Load,
Estimate (3).

function taking as argument an object of type MARKOV:
Display, Plot, Print, Save,
Compare (4)(5), Simulate (3),
ExtractDistribution.

type MARKOV_DATA

function returning an object of type MARKOV_DATA:

Load,
ComputeStateSequences, Simulate (3),
ExtractData.

function taking as argument an object of type MARKOV_DATA:
Display, Plot, Print, Save,
AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract,
LengthSelect, Merge, MergeVariable, MovingAverage,
RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract,
SelectIndividual, SelectVariable, Transcode, ValueSelect,
VariableScaling,
Compare (3)(4), ComputeCorrelation, ComputePartialAutoCorrelation,
ComputeStateSequences, ComputeWhiteNoiseCorrelation, Estimate (3),
ModelSelectionTest, Simulate (3),
ComputeSelfTransition, ExtractHistogram, ExtractVectors.

type MIXTURE

function returning an object of type MIXTURE:
Mixture, Load,
Estimate (1).

function taking as argument an object of type MIXTURE:
Display, Plot, Print, Save,
Compound, Convolution, Mixture, Renewal,
Fit, Simulate (1),
ExtractData, ExtractDistribution.

type MIXTURE_DATA

function returning an object of type MIXTURE_DATA:
Load,
Simulate (1),
ExtractData.

function taking as argument an object of type MIXTURE_DATA:
TimeEvents, Display, Plot, Print, Save,
Cluster, Merge, Shift, Transcode, ValueSelect,
Compare (1)(5), ComparisonTest, Estimate (1), Fit, Simulate (2)(3),
ExtractHistogram.

type REGRESSION

function returning an object of type REGRESSION:
Load,
Regression.

function taking as argument an object of type REGRESSION:
Display, Plot, Print, Save.

type RENEWAL

function returning an object of type RENEWAL:
Renewal, Load,

Estimate (2).

function taking as argument an object of type RENEWAL:

Display, Plot, Print, Save,
Compound, Convolution, Mixture, Renewal,
Simulate (2),
ExtractDistribution.

type RENEWAL_DATA

function returning an object of type RENEWAL_DATA:

Load,
Simulate (2).

function taking as argument an object of type RENEWAL_DATA:

Sequences, Display, Plot, Print, Save,
Merge, NbEventSelect, TimeScaling, TimeSelect,
Estimate (2),
ExtractHistogram.

type SEMI-MARKOV

function returning an object of type SEMI-MARKOV:

SemiMarkov, Load,
Estimate (3).

function taking as argument an object of type SEMI-MARKOV:

Display, Plot, Print, Save,
Compare (4)(5), Simulate (3),
ExtractDistribution.

type SEMI-MARKOV_DATA

function returning an object of type SEMI-MARKOV_DATA:

Load,
ComputeStateSequences, Simulate (3),
ExtractData.

function taking as argument an object of type SEMI-MARKOV_DATA:

Display, Plot, Print, Save,
AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract,
LengthSelect, Merge, MergeVariable, MovingAverage,
RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract,
SelectIndividual, SelectVariable, Transcode, ValueSelect,
VariableScaling,
Compare (3)(4), ComputeCorrelation, ComputePartialAutoCorrelation,
ComputeStateSequences, ComputeWhiteNoiseCorrelation, Estimate (3),
ModelSelectionTest, Simulate (3),
ComputeSelfTransition, ExtractHistogram, ExtractVectors.

type SEQUENCES

function returning an object of type SEQUENCES:

Sequences, Load,

Cluster, Cumulate, Difference, IndexExtract, LengthSelect, Merge, MergeVariable, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect, VariableScaling, TransformPosition.

function taking as argument an object of type SEQUENCES:

TimeEvents, Display, Plot, Print, Save, Cluster, Cumulate, Difference, IndexExtract, LengthSelect, Merge, MergeVariable, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect, VariableScaling, Compare (3), ComputeCorrelation, ComputePartialAutoCorrelation, ComputeWhiteNoiseCorrelation, ExtractHistogram, ExtractVectors, TransformPosition.

type TIME_EVENTS

function returning an object of type TIME_EVENTS:

TimeEvents, Load, Merge, NbEventSelect, TimeScaling, TimeSelect.

function taking as argument an object of type TIME_EVENTS:

Display, Plot, Print, Save, Merge, NbEventSelect, TimeScaling, TimeSelect, Estimate (2), ExtractHistogram.

type TOP_PARAMETERS

function returning an object of type TOP_PARAMETERS:

TopParameters, Load, Estimate (4).

function taking as argument an object of type TOP_PARAMETERS:

Display, Plot, Print, Save, Simulate (4), ExtractDistribution.

type TOPS

function returning an object of type TOPS:

Tops, Load, Merge, RemoveApicalInternodes, Reverse, SelectIndividual, Simulate (4).

function taking as argument an object of type TOPS:

Display, Plot, Print, Save, Merge, RemoveApicalInternodes, Reverse, SelectIndividual, Estimate (4), ExtractHistogram.

type VECTOR_DISTANCE

function returning an object of type VECTOR_DISTANCE:

VectorDistance.

function taking as argument an object of type VECTOR_DISTANCE:

Display, Print,
Compare (2)(3).

type VECTORS

function returning an object of type VECTORS:

Vectors, Load,
Cluster, Merge, MergeVariable, SelectIndividual, SelectVariable,
Shift, Transcode, ValueSelect,
ExtractVectors.

function taking as argument an object of type VECTORS:

Display, Plot, Print, Save,
Cluster, Merge, MergeVariable, SelectIndividual, SelectVariable,
Shift, Transcode, ValueSelect,
Compare (2), ComputeRankCorrelation, ContingencyTable, Regression,
VarianceAnalysis,
ExtractHistogram.

3.4 Detailed description

AddAbsorbingRun

Addition of a run of absorbing vectors at the end of sequences.

USAGE

```
AddAbsorbingRun(seq, Length->50)
```

ARGUMENT

`seq` (`DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA`).

OPTIONAL ARGUMENT

`Length` (`INT`): length of the sequences. A default value is computed from the maximum sequence length.

RETURNED OBJECT

An object of type `DISCRETE_SEQUENCES` is returned.

SEE ALSO

`Cluster`, `Cumulate`, `Difference`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `Reverse`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`.

Cluster

Clustering of values.

USAGE

```
Cluster(histo, "Step", step)
Cluster(histo, "Information", information_ratio)
Cluster(histo, "Limit", limits)

Cluster(vec1, "Step", step)
Cluster(vecn, "Step", variable, step)
Cluster(vec1, "Limit", limits)
Cluster(vecn, "Limit", variable, limits)

Cluster(seq1, "Step", step)
Cluster(seqn, "Step", variable, step)
Cluster(discrete_seq1, "Step", step, AddVariable->True)
Cluster(discrete_seqn, "Step", variable, step,
        AddVariable->True)
Cluster(seq1, "Limit", limits)
Cluster(seqn, "Limit", variable, limits)
Cluster(discrete_seq1, "Limit", limits, AddVariable->True)
Cluster(discrete_seqn, "Limit", variable, limits,
        AddVariable->True)
```

ARGUMENTS

histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),

step (INT): step for the clustering of values,
information_ratio (REAL): proportion of the information measure of the original sample for determining the clustering step,
limits (ARRAY(INT)): first values corresponding to the new classes 1, …, *nb_class* - 1. By convention, the first value corresponding to the first class is 0,

vec1 (VECTORS): values,
vecn (VECTORS): vectors,
variable (INT): variable index,

seq1 (SEQUENCES): univariate sequences,
seqn (SEQUENCES): multivariate sequences,
discrete_seq1 (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): discrete univariate sequences,
discrete_seqn (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): discrete multivariate sequences.

OPTIONAL ARGUMENTS

AddVariable (BOOL): addition (instead of simple replacement) of the variable corresponding to the clustering of values (default value: False). This optional argument can only be used if the first argument is of type DISCRETE_SEQUENCES,

MARKOV_DATA or SEMI-MARKOV_DATA. The addition of the clustered variable is particularly useful if one wants to evaluate a lumpability hypothesis.

RETURNED OBJECT

– if step > 0, or if 0 ≤ information_ratio ≤ 1, or if 0 < limits@1 < limits@2 < … < limits@nb_class - 1 < (maximum possible value of histo), an object of type HISTOGRAM is returned, otherwise no object is returned.
– If variable is a valid index of a variable and if step > 0, or if 0 < limits@1 < limits@2 < … < limits@nb_class - 1 < (maximum possible value taken by the selected variable of vec1 or vecn), an object of type VECTORS is returned, otherwise no object is returned.
– If variable is a valid index of a variable of type STATE and if step > 0, or if 0 < limits@1 < limits@2 < … < limits@nb_class - 1 < (maximum possible value taken by the selected variable of seq1, seqn, discrete_seq1 or discrete_seqn), an object of type SEQUENCES or DISCRETE_SEQUENCES is returned, otherwise no object is returned. In the case of a first argument of type SEQUENCES, an object of type DISCRETE_SEQUENCES is returned if all the variables are of type STATE, if the possible values taken by each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

DESCRIPTION

In the case of the clustering of values of a frequency distribution on the basis of an information measure criterion (argument "Information"), both the information measure ratio and the selected optimal step are given in the shell window.

BACKGROUND

The clustering mode "Step" (and its variant "Information") is naturally adapted to numeric variables while the clustering mode "Limit" applies to both symbolic (nominal) and numeric variables. In the case of a symbolic variable, the function Cluster with the mode "Limit" can be seen as a dedicated interface of the more general function Transcode.

SEE ALSO

Merge, Shift, Transcode, ValueSelect, MergeVariable, SelectIndividual, SelectVariable, AddAbsorbingRun, Cumulate, Difference, IndexExtract, LengthSelect, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, VariableScaling.

Clustering

Application of clustering methods (either partitioning methods or hierarchical methods) to dissimilarity matrices between patterns.

USAGE

```
Clustering(dissimilarity_matrix, "Partition", nb_cluster,
           Prototypes->[1, 3, 12])
Clustering(dissimilarity_matrix, "Partition", clusters)
Clustering(dissimilarity_matrix, "Hierarchy")
```

ARGUMENTS

`dissimilarity_matrix` (`DISTANCE_MATRIX`) dissimilarity matrix between patterns,
`nb_cluster` (`INT`): number of clusters,
`clusters` (`ARRAY(ARRAY(INT))`): cluster composition.

OPTIONAL ARGUMENT

`Prototypes` (`ARRAY(INT)`): cluster prototypes.

RETURNED OBJECT

if the second mandatory argument is `"Partition"` and if $2 \leq$ `nb_cluster` $<$ (number of patterns), an object of type `CLUSTERS` is returned, otherwise no object is returned.

DESCRIPTION

In the case where the composition of clusters is a priori fixed, the function `Clustering` simply performs an evaluation of the a priori fixed partition.

SEE ALSO

`SelectIndividual`, `Symmetrize`, `Compare` (2)(3)(5), `ToDistanceMatrix`.

Compare (1)

Comparison of frequency distributions.

USAGE

```
Compare(histo1, histo2,…, type, FileName->"result",
        Format->"ASCII")
```

ARGUMENTS

`histo1`, `histo2`, … (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`),
`type` (`STRING`): variable type (`"NUMERIC"` (`"N"`), `"ORDINAL"` (`"O"`) or `"SYMBOLIC"` (`"S"`)).

OPTIONAL ARGUMENTS

`FileName` (`STRING`): name of the result file,
`Format` (`STRING`): format of the result file: `"ASCII"` (default format) or `"SpreadSheet"`. This optional argument can only be used in conjunction with the optional argument `FileName`.

RETURNED OBJECT

No object returned.

Compare (2)

Comparison of vectors.

USAGE

```
Compare(vec, vector_distance)
```

ARGUMENTS

`vec` (`VECTORS`),
`vector_distance` (`VECTOR_DISTANCE`).

RETURNED OBJECT

An object of type `DISTANCE_MATRIX` is returned.

BACKGROUND

The type `VECTOR_DISTANCE` implements standardization procedures. The objective of standardization is to avoid the dependence on the variable type (chosen among symbolic, ordinal, numeric and circular) and, for numeric variables, on the choice of the measurement units by converting the original variables to unitless variables.

SEE ALSO

`VectorDistance`, `Clustering`.

Compare (3)

Comparison of sequences.

USAGE

```
Compare(seq, vector_distance, RefSequence->3, TestSequence->8,
        Begin->”Free”, End->”Free”, Transposition->True,
        FileName->"result", Format->"SpreadSheet",
        AlignmentFileName->"alignment",
        AlignmentFormat->"ASCII")
Compare(seq, RefSequence->3, TestSequence->8,
        Begin->”Free”, End->”Free”,
        FileName->"result", Format->"SpreadSheet",
        AlignmentFileName->"alignment",
        AlignmentFormat->"ASCII")
```

ARGUMENTS

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`),
`vector_distance` (`VECTOR_DISTANCE`).

OPTIONAL ARGUMENTS

`RefSequence` (`INT`): identifier of the reference sequence,
`TestSequence` (`INT`): identifier of the test sequence,
`Begin` (`STRING`): `"Free"` or `"Fixed"` (the default). If this optional argument is set at `"Free"`, any space at the beginning of the alignment contribute a weight of 0 (begining-space free alignment).
`End` (`STRING`): `"Free"` or `"Fixed"` (the default). If this optional argument is set at `"Free"`, any space at the end of the alignment contribute a weight of 0 (end-space free alignment).
`Transposition` (`BOOL`): use of the transposition operation (default value: `False`). This optional argument requires the second mandatory argument being of type `VECTOR_DISTANCE`.
`FileName` (`STRING`): name of result file,
`Format` (`STRING`): format of result file: `"ASCII"` (default format) or `"SpreadSheet"`. This optional argument can only be used in conjunction with the optional argument `FileName`.
`AlignmentFileName` (`STRING`): file name of the sequences of edit operations (deletion/insertion/exact matching and eventually substitution and transposition) resulting from sequence alignments,
`AlignmentFormat` (`STRING`): format of the file of sequences of edit operations: `"ASCII"` (default format) or `"Binary"`. This optional argument can only be used in conjunction with the optional argument `AlignmentFileName`.

RETURNED OBJECT

An object of type `DISTANCE_MATRIX` is returned.

DESCRIPTION

If number of alignments ≤ 30, the alignments are displayed in the shell window.

BACKGROUND

The result of the comparison of two sequences takes the form of the sequence of edit operations required for transforming the reference sequence into the test sequence with associated costs:

- deletion ('d'): deletion of an element of the reference sequence,
- insertion ('i'): insertion of an element of the test sequence,
- exact matching ('m'): matching of an element of the test sequence with the same element in the reference sequence: null cost,
- substitution ('s') (mismatching): replacement of an element of the reference sequence by another element of the test sequence: The associated cost is the standardized distance between the two elements.
- transposition (two successive 't') or swap: interchange of adjacent elements in the sequence with the additional constraint that each element can participate in no more than one swap. This edit operation applies only in the case where two successive element in the reference sequence exactly match two successive elements in the test sequence taken in reverse order.

A purely structural alignment consists in allowing only exact matching, insertion and deletion. In this case, the argument `vector_distance` specifying the local distance between elements is not required.

SEE ALSO

`VectorDistance`, `Clustering`.

Compare (4)

Comparison of Markovian models for sequences.

USAGE

```
Compare(discrete_seq, mc1, mc2,…, FileName->"result")
Compare(discrete_seq, smc1, smc2,…, FileName->"result")
Compare(discrete_seq, hmc1, hmc2,…, Algorithm->"Viterbi",
        FileName->"result")
Compare(discrete_seq, hsmc1, hsmc2,…, Algorithm->"Viterbi",
        FileName->"result")
```

ARGUMENTS

`discrete_seq` (`DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`),
`mc1`, `mc2`, … (`MARKOV`),
`smc1`, `smc2`, … (`SEMI-MARKOV`),
`hmc1`, `hmc2`, … (`HIDDEN_MARKOV`),
`hsmc1`, `hsmc2`, … (`HIDDEN_SEMI-MARKOV`).

OPTIONAL ARGUMENTS

`Algorithm` (`STRING`): type of algorithm: `"Forward"` (the default) or `"Viterbi"`. This optional argument applies only with models of type `HIDDEN_MARKOV` or `HIDDEN_SEMI-MARKOV`,
`FileName` (`STRING`): name of result file.

RETURNED OBJECT

No object returned.

DESCRIPTION

The result of comparisons is displayed in the shell window.

BACKGROUND

In the case of Markov chains (type `MARKOV`) or semi-Markov chains (type `SEMI-MARKOV`), the comparison relies on the likelihood of a sequence for the different models being compared. In the case of hidden Markov chains (type `HIDDEN_MARKOV`) or hidden semi-Markov chains (`HIDDEN_SEMI-MARKOV`), the comparison relies either on the likelihood of a sequence (of every possible state sequences that can generate the observed sequence: `Algorithm->"Forward"`), or on the likelihood of the state sequence that best explains the observed sequence (`Algorithm->"Viterbi"`).

SEE ALSO

`ComputeStateSequences`.

Compare (5)

Comparison of Markovian models.

USAGE

```
Compare(mc1, length_histo1, mc2, length_histo2,…,
        FileName->"result")
Compare(mc1, mc2,…, nb_seq, length, FileName->"result")
Compare(mc1, seqm1, mc2, seqm2,…, nb_seq, FileName->"result")
Compare(smc1, length_histo1, smc2, length_histo2,…,
        FileName->"result")
Compare(smc1, smc2,…, nb_seq, length, FileName->"result")
Compare(smc1, seqm1, smc2, seqm2,…, nb_seq, FileName->"result")
Compare(hmc1, length_histo1, hmc2, length_histo2,…,
        FileName->"result")
Compare(hmc1, hmc2,…, nb_seq, length, FileName->"result")
Compare(hmc1, seqm1, hmc2, seqm2,…, nb_seq, FileName->"result")
Compare(hsmc1, length_histo1, hsmc2, length_histo2,…,
        FileName->"result")
Compare(hsmc1, hsmc2,…, nb_seq, length, FileName->"result")
Compare(hsmc1, seqm1, hsmc2, seqm2,…, nb_seq,
        FileName->"result")
```

ARGUMENTS

mc1, mc2, … (MARKOV),
smc1, smc2, … (SEMI-MARKOV),
hmc1, hmc2, … (HIDDEN_MARKOV),
hsmc1, hsmc2, … (HIDDEN_SEMI-MARKOV),
length_histo1, length_histo2, … (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA): frequency distribution of lengths of generated sequences,
nb_seq (INT): number of generated sequences,
length (INT): length of generated sequences,
seqm1, seqm2, … (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA).

OPTIONAL ARGUMENTS

FileName (STRING): name of result file. If this optional argument is used, some complementary results, with respect to the returned object of type DISTANCE_MATRIX, are saved on a file.

RETURNED OBJECT

An object of type DISTANCE_MATRIX is returned.

BACKGROUND

The comparison of Markovian models relies on the Kullback-Leibler directed divergence (or cross-entropy or discriminant information). For each model being compared, a sample of sequences is generated and the log-likelihoods of these sequences for the different models are computed (including the reference model used

for simulation). The dissimilarity measure is the 'divergence' from the reference model to the target model on the basis of log-likelihoods of the sequences normalized by their cumulative length. This procedure is repeated for each model.

SEE ALSO

`Estimate` (3).

ComparisonTest

Test of comparaison of frequency distributions.

USAGE

```
ComparisonTest(type, histo1, histo2)
```

ARGUMENTS

`type` (`STRING`): type of test: `"F"`, `"T"` or `"W"` (Wilcoxon-Mann-Whitney),
`histo1`, `histo2` (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`).

RETURNED OBJECT

No object returned.

DESCRIPTION

The result of the test is displayed in the shell window.

BACKGROUND

The objective is to compare two independent random samples in order to decide if they have been drawn from the same population or not.

In the case of samples from normal populations, the Fisher-Snedecor (`"F"`) test enables to test is the two variances are not significantly different. The normal distribution assumption should be checked for instance by the exam of the shape coefficients (skewness and kurtosis coefficients). The test statistic is:

$$F_{n_1-1,n_2-1} = \frac{\sum_{i}^{n_1} (x_{1i} - m_1)^2}{n_1 - 1} \Big/ \frac{\sum_{i}^{n_2} (x_{2i} - m_2)^2}{n_2 - 1}$$

where m_1 and m_2 are the means of the samples.

The Fisher-Snedecor variable F_{n_1-1,n_2-1} with n_1-1 degrees of freedom and n_2-1 degrees of freedom can be interpreted as the ratio of the variance estimators of the two samples. In practice, the larger estimated variance is put at the denominator. Hence $F_{n_1-1,n_2-1} \geq 1$. The critical region is of the form $F_{n_1-1,n_2-1} > f$ (one-sided test).

In the case of samples from normal populations with equal variances, the Student's (`"T"`) test enables to test if the two means are not significantly different. The test statistic is:

$$T_{n_1+n_2-2} = \frac{m_1 - m_2}{\sqrt{\left[\sum_{i=1}^{n_1}(x_{1i} - m_1)^2 + \sum_{i=1}^{n_2}(x_{2i} - m_2)^2\right]\left(\frac{1}{n_1} + \frac{1}{n_2}\right)}} \sqrt{n_1 + n_2 - 2}$$

The critical region is of the form $\left|T_{n_1+n_2-2}\right| \geq t$ (two-sided test). For sufficiently large sample sizes, this test of sample mean comparison can be used for samples from non-normal populations with unequal variances. This test is said to be robust.

The Wilcoxon-Mann-Whitney (`"W"`) test is a distribution-free test relying on the 'homogeneity' of the ranking of the two sample (ranks of one sample should not cluster at either or both ends of the range). It can be seen as the nonparametric analog of the Student's *t* test and can be applied to compare two sets of observations measured on an interval scale when it is supposed that the data are non-normally distributed, or to compare two sets of observations measured on an ordinal scale.

Compound

Construction of an object of type COMPOUND from a 'sum' distribution and an 'elementary' distribution or from an ASCII file.

USAGE

```
Compound(file_name)

Compound(sum_dist, dist)
```

ARGUMENTS

sum_dist (DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND): sum distribution,
dist (DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND): elementary distribution,

file_name (STRING).

RETURNED OBJECT

If the construction succeeds, an object of type COMPOUND is returned, otherwise no object is returned.

BACKGROUND

A compound (or stopped-sum) distribution is defined as the distribution of the sum of n independent and identically distributed random variables X_i where n is the value taken by the random variable N. The distribution of N is referred to as the sum distribution while the distribution of the X_i is referred to as the elementary distribution.

SEE ALSO

Save, Estimate (1), Simulate (1).

ComputeCorrelation

Computation of sample autocorrelation or cross-correlation functions.

USAGE

```
ComputeCorrelation(seq1, MaxLag->10, Type->"Spearman"
                   Normalization->"Exact")
ComputeCorrelation(seqn, variable, MaxLag->10,
                   Type->"Spearman", Normalization->"Exact")
ComputeCorrelation(seqn, variable1, variable2, MaxLag->10,
                   Type->"Spearman", Normalization->"Exact")
```

ARGUMENTS

`seq1` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): univariate sequences,

`seqn` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): multivariate sequences,

`variable` (`INT`): variable index (computation of a sample autocorrelation function).

`variable1`, `variable2` (`INT`): variable indices (computation of a sample cross-correlation function).

OPTIONAL ARGUMENTS

`Type` (`STRING`): type of correlation coefficient: `"Pearson"` (linear correlation coefficient - default value), `"Spearman"` or `"Kendall"` (rank correlation coefficients).

`MaxLag` (`INT`): maximum lag. A default value is computed from the sequence length distribution,

`Normalization` (`STRING`): normalization of the correlation coefficients: `"Approximated"` (the default – usual convention for time series analysis) or `"Exact"`, (highly recommended for sample of short sequences). This optional argument can only be used if the optional argument `Type` is set at `"Pearson"` or `"Spearman"`.

RETURNED OBJECT

If `variable`, or `variable1` and `variable2` are valid indices of variables (and are different if two indices are given) and if $0 \leq$ `MaxLag` $<$ (maximum length of sequences), an object of type `CORRELATION` is returned, otherwise no object is returned.

BACKGROUND

In the univariate case or if only `variable` is given, a sample autocorrelation function is computed. If `variable1` and `variable2` are given, a sample cross-correlation function is computed.

SEE ALSO

`ComputePartialAutoCorrelation`, `ComputeWhiteNoiseCorrelation`.

ComputePartialAutoCorrelation

Computation of sample partial autocorrelation functions.

USAGE

```
ComputePartialAutoCorrelation(seq1, MaxLag->10,
                              Type->"Kendall")
ComputePartialAutoCorrelation(seq  , variable, MaxLag->10,
                              Type->"Kendall")
```

ARGUMENTS

`seq1` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): univariate sequences,
`seqn` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): multivariate sequences,
`variable` (`INT`): variable index.

OPTIONAL ARGUMENTS

`MaxLag` (`INT`): maximum lag. A default value is computed from the sequence length distribution,
`Type` (`STRING`): type of correlation coefficient: `"Pearson"` (linear correlation coefficient – the default) or `"Kendall"` (rank correlation coefficient).

RETURNED OBJECT

If `variable` is a valid variable index and if $1 \leq$ `MaxLag` $<$ (maximum length of sequences), an object of type `CORRELATION` is returned, otherwise no object is returned.

BACKGROUND

The partial autocorrelation coefficient at lag k measures the correlation between x_t and x_{t+k} not accounted for by $x_{t+1},\ldots,x_{t+k-1}$ (or after adjusting for the effects of $x_{t+1},\ldots,x_{t+k-1}$).

SEE ALSO

`ComputeCorrelation`.

ComputeRankCorrelation

Computation of the rank correlation matrix.

USAGE

```
ComputeRankCorrelation(vec, Type->"Spearman")
```

ARGUMENT

`vec` (`VECTORS`).

OPTIONAL ARGUMENT

`Type` (`STRING`): type of rank correlation coefficient: `"Spearman"` (the default) or `"Kendall"`.

RETURNED OBJECT

No object returned.

ComputeSelfTransition

Computation of the self-transition probabilities as a function of the index parameter from discrete sequences.

USAGE

```
ComputeSelfTransition(seq, Order->2)
```

ARGUMENT

`seq` (`DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`).

OPTIONAL ARGUMENT

`Order` (`INT`): Markov chain order (default value: 1).

RETURNED OBJECT

No object returned.

ComputeStateSequences

Computation of the optimal state sequences corresponding to the observed sequences using a hidden Markov chain or a hidden semi-Markov chain.

USAGE

```
ComputeStateSequences(seq, hmc, Algorithm->"ForwardBackward",
                      Characteristics->True)
ComputeStateSequences(seq, hsmc, Algorithm->"ForwardBackward",
                      Characteristics->True)
```

ARGUMENTS

hmc (HIDDEN_MARKOV),
hsmc (HIDDEN_SEMI-MARKOV),
seq (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA).

OPTIONAL ARGUMENTS

Algorithm (STRING): type of algorithm: "Viterbi" (the default) or "ForwardBackward".

Characteristics (BOOL): computation of the characteristic distributions of the model taking into account the lengths of the segmented sequences (default value: False).

RETURNED OBJECT

If the second mandatory argument is of type HIDDEN_MARKOV, an object of type MARKOV_DATA is returned. If the second mandatory argument is of type HIDDEN_SEMI-MARKOV, an object of type SEMI-MARKOV_DATA is returned. The returned object contains both the sequences (including the optimal state sequences as a supplementary variable) and the model.

BACKGROUND

This restoration of the state sequence is either performed by a dynamic programming algorithm referred to as Viterbi algorithm which maximizes the joint probability of the state and output sequence $P(S_0^{\tau-1} = s_0^\tau, X_0^{\tau-1} = x_0^\tau)$ (global criterion) or by a 'forward-backward' algorithm which chooses state j at time t to maximize $P(S_t = j | X_0^{\tau-1} = x_0^\tau)$ (local criterion).

SEE ALSO

ExtractHistogram.

ComputeWhiteNoiseCorrelation

Computation of the autocorrelation or cross-correlation function induced on a white noise sequence by filtering.

USAGE

```
ComputeWhiteNoiseCorrelation(cf, order)

ComputeWhiteNoiseCorrelation(cf, filter)
ComputeWhiteNoiseCorrelation(cf, frequencies)
ComputeWhiteNoiseCorrelation(cf, dist)
```

ARGUMENTS

`cf` (`CORRELATION`): sample autocorrelation or cross-correlation function (in the Pearson's sense),

`order` (`INT`): order of differencing,

`filter` (`ARRAY(REAL)`): filter values on a half width i.e. from one extremity to the central value (with the constraint that $2\sum_{i=1}^{m-1} \texttt{filter}_i + \texttt{filter}_m = 1$),

`frequencies` (`ARRAY(INT)`): frequencies defining the filter,

`dist` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`): symmetric distribution whose size of the support is even defining the filter (for instance `Distribution("BINOMIAL", 0, 4, 0.5)`),

RETURNED OBJECT

No object is returned.

BACKGROUND

The application of linear filters for trend removal induces an autocorrelation structure. The effect of a given linear filter on the autocorrelation structure of the residual sequence can be roughly described as follows: the number of non-zero induced autocorrelation coefficients increase with the width of the filter while their numerical magnitudes decrease.

SEE ALSO

`ComputeCorrelation`.

ContingencyTable

Computation of a contingency table.

USAGE

```
ContingencyTable(vec, variable1, variable2, FileName->"result",
                 Format->"SpreadSheet")
```

ARGUMENTS

`vec` (`VECTORS`),
`variable1`, `variable2` (`INT`): variable indices.

OPTIONAL ARGUMENTS

`FileName` (`STRING`): name of the result file,
`Format` (`STRING`): format of the result file: `"ASCII"` (default format) or `"SpreadSheet"`. This optional argument can only be used in conjunction with the optional argument `FileName`.

RETURNED OBJECT

No object returned.

DESCRIPTION

The contingency table is displayed in the shell window.

Convolution

Construction of an object of type CONVOLUTION from elementary distributions or from an ASCII file.

USAGE

```
Convolution(dist1, dist2,…)
```

```
Convolution(file_name)
```

ARGUMENTS

dist1, dist2, …(DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND): elementary distributions,

file_name (STRING).

RETURNED OBJECT

If the construction succeeds, the returned object is of type CONVOLUTION, otherwise no object is returned.

BACKGROUND

The distribution of the sum of independent random variables is the convolution of the distributions of these elementary random variables.

SEE ALSO

Save, Estimate (1), Simulate (1).

Cumulate

Sum of successive values along sequences.

USAGE

```
Cumulate(seq, Variable->1)
```

ARGUMENT

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`).

OPTIONAL ARGUMENT

`Variable` (`INT`): variable index.

RETURNED OBJECT

The returned object is of type `SEQUENCES` or `DISCRETE_SEQUENCES`. The returned object is of type `DISCRETE_SEQUENCES` if all the variables are of type `STATE`, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

BACKGROUND

`Cumulate` is the inverse function of `Difference` with the optional argument `FirstValue` set at `True`.

SEE ALSO

`AddAbsorbingRun`, `Cluster`, `Difference`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`.

Difference

First-order differencing of sequences.

USAGE

```
Difference(seq, Variable->1, FirstElement->True)
```

ARGUMENT

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`).

OPTIONAL ARGUMENTS

`Variable` (`INT`): variable index,
`FirstElement` (`BOOL`): first element of sequences kept or not (default value: `False`).

RETURNED OBJECT

The returned object is of type `SEQUENCES` or `DISCRETE_SEQUENCES`. The returned object is of type `DISCRETE_SEQUENCES` if all the variables are of type `STATE`, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

BACKGROUND

If the first element of sequences are kept (`FirstValue->True`), `Difference` is the inverse function of `Cumulate`

SEE ALSO

`AddAbsorbingRun`, `Cluster`, `Cumulate`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`.

Display

ASCII output of an object of the STAT module (Print: ASCII printing of an object of the STAT module. Print takes the same arguments as Display).

USAGE

```
Display(obj, Detail->2)

Display(vec, ViewPoint->"Data", Detail->2)
Display(seq, ViewPoint->"Data", Format->"Line", Detail->2)

Display(dist, ViewPoint->"Survival")
Display(histo, ViewPoint->"Survival")

Display(hmc, identifier, ViewPoint->"StateProfile")
Display(hsmc, identifier, ViewPoint->"StateProfile")
```

ARGUMENTS

obj: STAT module object,

vec (VECTORS),
seq (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA, TOPS),

dist (DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND),
histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),

hmc (HIDDEN_MARKOV),
hsmc (HIDDEN_SEMI-MARKOV),
identifier (INT): identifier of a sequence.

OPTIONAL ARGUMENTS

ViewPoint (STRING): point of view on the object ("Survival" or "Data" or "StateProfile"). This optional argument can be set at "Data" only if the first argument is of type VECTORS, SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA or TOPS, can be set at "Survival" only if the first argument is of type DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND, HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA or COMPOUND_DATA and can be set at "StateProfile" only if the first argument is of type HIDDEN_MARKOV or HIDDEN_SEMI-MARKOV.

Detail (INT): level of detail: 1 (default value) or 2. This optional argument cannot be used if the optional argument ViewPoint is set at "Survival" or "StateProfile".

Format (STRING): format of sequences (only relevant for multivariate sequences): "Column" (default value) or "Line". This optional argument can only be used if the optional argument ViewPoint is set at "Data", and hence, if the first argument is of type SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA or TOPS.

DESCRIPTION

ASCII output of sets of sequences or tops (`ViewPoint->"Data"`): the format `"Column"` corresponds to the ASCII file syntax for objects of type `SEQUENCES` or `TOPS`. For a given value of the index parameter, the different variables are successively displayed. With the format `"Line"`, the univariate sequence for each variable are displayed on consecutive lines. In the case of univariate sequences, the two formats give the same output.

ASCII output of a (frequency) distribution and the associate hazard or survival rates (`ViewPoint->"Survival"`): It is assumed that the (frequency) distribution represents lifetime and the hazard or survival rates are deduced from this lifetime distribution.

ASCII output of the state profile given by the smoothed probabilities $P(S_t = j | X_0^{\tau-1} = x_0^\tau)$ as a function of the index parameter t computed from the parameters of a hidden Markovian model for the sequence x_0^τ (`ViewPoint->"StateProfile"`).

RETURNED OBJECT

No object returned.

SEE ALSO

`Plot`, `Save`.

Distribution

Construction of a parametric discrete distribution (either binomial, Poisson, negative binomial or uniform) from the name and the parameters of the distribution or from an ASCII file.

USAGE

```
Distribution("BINOMIAL", inf_bound, sup_bound, proba)
Distribution("POISSON", inf_bound, param)
Distribution("NEGATIVE_BINOMIAL", inf_bound, param, proba)
Distribution("UNIFORM", inf_bound, sup_bound)

Distribution(file_name)
```

ARGUMENTS

`inf_bound` (`INT`): lower bound to the range of possible values (shift parameter),

`sup_bound` (`INT`): upper bound to the range of possible values (only relevant for binomial or uniform distributions),

`param` (`INT`, `REAL`): parameter of either the Poisson distribution or the negative binomial distribution.

`proba` (`INT`, `REAL`): probability of 'success' (only relevant for binomial or negative binomial distributions),

Remark: the names of the parametric discrete distributions can be summarized by their first letters: `"B"` (`"BINOMIAL"`), `"P"` (`"POISSON"`), `"NB"` (`"NEGATIVE_BINOMIAL"`), `"U"` (`"UNIFORM"`),

`file_name` (`STRING`).

RETURNED OBJECT

If the construction succeeds, an object of type `DISTRIBUTION` is returned, otherwise no object is returned.

DESCRIPTION

A supplementary shift parameter (argument `inf_bound`) is required with respect to the usual definitions of these discrete distributions. Constraints over parameters are given in the file syntax corresponding to the type `DISTRIBUTION` (cf. Part II4).

SEE ALSO

`Save`, `Estimate` (1), `Simulate` (1).

Estimate (1)

Estimation of distributions.

USAGE

```
Estimate(histo, "NON-PARAMETRIC")
Estimate(histo, "NB", MinInfBound->1, InfBoundStatus->"Fixed")

Estimate(histo, "MIXTURE", "B", dist,…,
         MinInfBound->1, InfBoundStatus->"Fixed",
         DistInfBoundStatus->"Fixed")
Estimate(histo, "MIXTURE", "B", "NB",…,
         MinInfBound->1, InfBoundStatus->"Fixed",
         DistInfBoundStatus->"Fixed",
         NbComponent->"Estimated", Penalty->"AIC")

Estimate(histo, "CONVOLUTION", dist,
         MinInfBound->1, Parametric->False)
Estimate(histo, "CONVOLUTION", dist,
         InitialDistribution->initial_dist, Parametric->False)

Estimate(histo, "COMPOUND", dist, unknown,
         Parametric->False, MinInfBound->0)
Estimate(histo, "COMPOUND", dist, unknown,
         InitialDistribution->initial_dist, Parametric->False)
```

ARGUMENTS

`histo` (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`),
`dist` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`),
`unknown` (`STRING`): type of unknown distribution: `"Sum"` or `"Elementary"`.

OPTIONAL ARGUMENTS

`MinInfBound` (`INT`): lower bound to the range of possible values (0 - default value - or 1). This optional argument cannot be used in conjunction with the optional argument `InitialDistribution`.

`InfBoundStatus` (`STRING`): shifting or not of the distribution: `"Free"` (default value) or `"Fixed"`. This optional argument cannot be used if the second mandatory argument giving the model type is `"NON-PARAMETRIC"` (`"NP"`).

`DistInfBoundStatus` (`STRING`): shifting or not of the subsequent components of the mixture: `"Free"` (default value) or `"Fixed"`. This optional argument can only be used if the second mandatory argument giving the distribution type is `"MIXTURE"`.

`NbComponent` (`STRING`): estimation of the number of components of the mixture: `"Fixed"` (default value) or `"Estimated"`. This optional argument can only be used if the second mandatory argument giving the distribution type is `"MIXTURE"`. Le number of estimated components is comprised between 1 and a maximum number which is given by the number of specified parametric distributions in the mandatory arguments (all of these distributions are assumed to be unknown).

`Penalty` (`STRING`): type of penalty function for model selection: `"AIC"` (Akaike Information Criterion), `"AICc"` (corrected Akaike Information Criterion - default value) or `"BIC"` (Bayesian Information Criterion). This optional argument can only be used if the second mandatory argument giving the distribution type is `"MIXTURE"` and if the optional argument `NbComponent` is set at `"Estimated"`.

`Parametric` (`BOOL`): reestimation of a discrete nonparametric or parametric distribution (default value: `True`). This optional argument can only be used if the second mandatory argument giving the distribution type is `"CONVOLUTION"` or `"COMPOUND"`.

`InitialDistribution` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`): initial distribution for the EM deconvolution-type algorithm. This optional argument can only be used if the second mandatory argument giving the distribution type is `"CONVOLUTION"` or `"COMPOUND"`. This optional argument cannot be used in conjunction with the optional argument `MinInfBound`.

Remark: the optional arguments `MinInfBound` and `InitialDistribution` are mutually exclusive.

RETURNED OBJECT

In case of success of the estimation procedure, the type of the returned object (chosen among `DISTRIBUTION`, `MIXTURE`, `CONVOLUTION` or `COMPOUND`) is given by the second mandatory argument. Otherwise no object is returned. The returned object of type `DISTRIBUTION`, `MIXTURE`, `CONVOLUTION` or `COMPOUND` contains both the estimated distribution and the data used in the estimation procedure. In the case of mixtures, convolutions, or compound (or stopped-sum) distributions, the returned object contains pseudo-data computed as a byproduct of the EM algorithm which can be extracted by the function `ExtractData`.

SEE ALSO

`ExtractData`, `ExtractDistribution`.

Estimate (2)

Estimation of a renewal process from count data.

USAGE

```
Estimate(timev, type, NbIteration->10, Parametric->True)
Estimate(timev, type, InitialInterEvent->initial_dist,
         NbIteration->10, Parametric->True)
```

ARGUMENTS

`timev` (`TIME_EVENTS`, `RENEWAL_DATA`),
`type` (`STRING`): type or renewal process: `"Ordinary"` or `"Equilibrium"`.

OPTIONAL ARGUMENTS

`InitialInterEvent` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`): initial inter-event distribution for the EM algorithm.
`NbIteration` (`INT`): number of iterations of the EM algorithm.
`Parametric` (`BOOL`): reestimation of a discrete nonparametric or parametric distribution (default value: `False`).

RETURNED OBJECT

In case of success of the estimation procedure, an object of type `RENEWAL` is returned, otherwise no object is returned. The returned object of type `RENEWAL` contains both the estimated renewal process and the count data used in the estimation procedure.

SEE ALSO

`ExtractDistribution`.

Estimate (3)

Estimation of (hidden) Markovian models.

USAGE

```
Estimate(seq, "MARKOV", Order->2, Counting->False)
Estimate(seq, "MARKOV", MaxOrder->3, Penalty->"AIC",
         Counting->False)
Estimate(seq, "MARKOV", states, Penalty->"AIC", Order->2,
         Counting->False)

Estimate(seq, "NON-HOMOGENEOUS_MARKOV", MONOMOLECULAR, VOID,
         Counting->False)

Estimate(seq, "SEMI-MARKOV", Counting->False)

Estimate(seq, "HIDDEN_MARKOV", nb_state, structure,
         SelfTransition->0.9, NbIteration->10,
         StateSequences->"Viterbi", Counting->False)
Estimate(seq, "HIDDEN_MARKOV", hmc, Algorithm->"Viterbi",
         NbIteration->10, Order->2, Counting->False)
Estimate(seq, "HIDDEN_MARKOV", "NbState", min_nb_state,
         max_nb_state, Penalty->"AIC", Order->2,
         Counting->False)
Estimate(seq, "HIDDEN_MARKOV", "NbState", hmc, state,
         max_nb_state, Penalty->"AIC", SelfTransition->0.9,
         Counting->False)

Estimate(seq, "HIDDEN_SEMI-MARKOV", nb_state, structure,
         OccupancyMean->20, NbIteration->10,
         Estimator->"PartialLikelihood",
         StateSequences->"Viterbi", Counting->False)
Estimate(seq, "HIDDEN_SEMI-MARKOV", hsmc, Algorithm->"Viterbi",
         NbIteration->10, Counting->False)
```

ARGUMENTS

seq (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),
states, … (ARRAY(INT)): new states corresponding to a partition of the original state space,
hmc (HIDDEN_MARKOV),
structure (STRING): structural properties of the underlying Markov chain: "Irreductible" or "LeftRight" (i.e. a succession of transient states and a final absorbing state),
nb_state (INT): number of states with $2 \leq$ nb_state ≤ 15,
min_nb_state (INT): minimum number of states,
max_nb_state (INT): maximum number of states with $2 \leq$ min_nb_state $<$ max_nb_state ≤ 15 or (number of states of the initial hidden Markov chain hmc) $<$ max_nb_state ≤ 15.
state (INT): state to be duplicated,
hsmc (HIDDEN_SEMI-MARKOV).

OPTIONAL ARGUMENTS

`Counting` (`BOOL`): computation of counting distributions (default value: `True`).

`Order` (`INT`): Markov chain order (default value: 1). This optional argument can only be used if the second mandatory argument giving the model type is `"MARKOV"`, `"NON-HOMOGENEOUS_MARKOV"` or `"HIDDEN_MARKOV"`.

`MaxOrder` (`INT`): maximum order of the Markov chain (default value: 4). This optional argument can only be used if the second mandatory argument giving the model type is `"MARKOV"`.

`Penalty` (`STRING`): type of penalty function for model selection: `"AIC"` (Akaike Information Criterion), `"AICc"` (corrected Akaike Information Criterion) or `"BIC"` (Bayesian Information Criterion). This optional argument can only be used if the second mandatory argument giving the model type is `"MARKOV"` (default value: `"BIC"`) and if the optional argument `MaxOrder` is set or else, if a new set of states is given (defining a partition of the original state space) or else, if the second mandatory argument giving the model type is `"HIDDEN_MARKOV"` and the third `"NbState"` (default value: `"AICc"`).

`Algorithm` (`STRING`): type of algorithm: `"ForwardBackward"` (the default) or `"Viterbi"`. This optional argument can only be used if the second mandatory argument giving the model type is `"HIDDEN_MARKOV"` or `"HIDDEN_SEMI-MARKOV"`.

`NbIteration` (`INT`): number of iterations of the estimation algorithm.

`SelfTransition` (`REAL`): self-transition probability. This optional argument can only be used if the second mandatory argument giving the model type is `"HIDDEN_MARKOV"` and if the initial model used in the iterative estimation procedure (EM algorithm) is only specified by its number of states, its structural properties and eventually its order.

`OccupancyMean` (`INT`/`REAL`): average state occupancy. This optional argument can only be used if the second mandatory argument giving the model type is `"HIDDEN_SEMI-MARKOV"` and if the initial model used in the iterative estimation procedure (EM algorithm) is only specified by its number of states and its structural properties.

`Estimator` (`STRING`): type of estimator: `"CompleteLikelihood"` (the default) or `"PartialLikelihood"`. In this latter case, the contribution of the time spent in the last visited state is not taken into account in the estimation of the state occupancy distributions. This optional argument can only be used if the second mandatory argument giving the model type is `"HIDDEN_SEMI-MARKOV"` and the optional argument `Algorithm` is set at `"ForwardBackward"`.

`StateSequences` (`STRING`): Computation of the optimal state sequences: no computation (the default), `"ForwardBackward"` or `"Viterbi"`. This optional argument can only be used if the second mandatory argument giving the model type is `"HIDDEN_MARKOV"` or `"HIDDEN_SEMI-MARKOV"` and if the optional argument `Algorithm` is not set at `"Viterbi"`.

RETURNED OBJECT

In case of success of the estimation procedure, the type of the returned object (chosen among MARKOV, SEMI-MARKOV, HIDDEN_MARKOV, HIDDEN_SEMI-MARKOV) is given by the second mandatory argument. Otherwise no object is returned. If the second mandatory argument is "NON-HOMOGENEOUS_MARKOV", in case of success of the estimation procedure, the returned object is of type MARKOV. If the second mandatory argument is "NON-HOMOGENEOUS_MARKOV", the subsequent arguments chosen among "VOID" (homogeneous state), "MONOMOLECULAR" or "LOGISTIC", specify the evolution of the self-transition probabilities as a function of the index parameter. The returned object of type MARKOV, SEMI-MARKOV, HIDDEN_MARKOV or HIDDEN_SEMI-MARKOV contains both the estimated distribution and the data used in the estimation procedure. In the case of the estimation of a hidden Markov chain or a hidden semi-Markov chain, the returned object contains pseudo-data (optimal state sequences corresponding to the observed sequences used in the estimation procedure) computed as a byproduct of the EM algorithm which can be extracted by the function ExtractData.

DESCRIPTION

In the case of hidden Markovian models (second mandatory argument "HIDDEN_MARKOV" or "HIDDEN_SEMI-MARKOV"), either the forward-backward algorithm or the Viterbi algorithm can be used for estimation. The Viterbi algorithm should only be used for the estimation of hidden Markovian models based on an underlying "left-right" Markov chain (i.e. constituted of a succession of transient states and a final absorbing state). Hence, in this case, the model structure is implicitly "LeftRight" and should not be given as argument (only the number of states should be given as argument). Since the optimal state sequences are computed by the Viterbi algorithm, the optional argument StateSequences cannot be used if the optional argument Algorithm is set at "Viterbi".

SEE ALSO

AddAbsorbingRun, ExtractData, ExtractDistribution, ModelSelectionTest.

Estimate (4)

Estimation of 'top' parameters.

USAGE

```
Estimate(top, MinPosition->1, MaxPosition->5, Neighbourhood->2,
         EqualProba->True)
```

ARGUMENT

`top` (`TOPS`),

OPTIONAL ARGUMENTS

`MinPosition` (`INT`): lower position taken into account for the estimation of 'top' parameters.

`MaxPosition` (`INT`): upper position taken into account for the estimation of 'top' parameters.

`Neighbourhood` (`INT`): neighbourhood taken into account for the estimation of 'top' parameters.

`EqualProba` (`BOOL`): growth probabilities of the parent shoot and of the offspring shoots equal or not (default value: `False`).

RETURNED OBJECT

In case of success of the estimation procedure, an object of type `TOP_PARAMETERS` is returned, otherwise no object is returned. The returned object of type `TOP_PARAMETERS` contains both the estimated model and the data used for the estimation.

BACKGROUND

The aim of the model of 'tops' is to related the growth of offspring shoots to the growth of their parent shoot in the case of immediate branching. A model of 'tops' is defined by three parameters, namely the growth probability of the parent shoot, the growth probability of the offspring shoots (both in the sense of Bernoulli processes) and the growth rhythm ratio offspring shoots / parent shoot.

SEE ALSO

`ExtractDistribution`.

ExtractData

Extraction of the ‘data’ part of an object of type ‘model’.

USAGE

```
ExtractData(mixt)
ExtractData(convol)
ExtractData(compound)

ExtractData(hmc)
ExtractData(hsmc)
```

ARGUMENTS

`mixt` (`MIXTURE`),
`convol` (`CONVOLUTION`),
`compound` (`COMPOUND`),

`hmc` (`HIDDEN_MARKOV`),
`hsmc` (`HIDDEN_SEMI-MARKOV`).

RETURNED OBJECT

- if `mixt` contains a ‘data’ part, an object of type `MIXTURE_DATA` is returned, otherwise no object is returned.
- If `convol` contains a ‘data’ part, an object of type `CONVOLUTION_DATA` is returned, otherwise no object is returned.
- If `compound` contains a ‘data’ part, an object of type `COMPOUND_DATA` is returned, otherwise no object is returned.
- If `hmc` contains a ‘data’ part, an object of type `MARKOV_DATA` is returned, otherwise no object is returned.
- If `hsmc` contains a ‘data’ part, an object of type `SEMI-MARKOV_DATA` is returned, otherwise no object is returned.

DESCRIPTION

This function enables to extract the ‘data’ part of an object of type ‘model’ when the estimation of model parameters from data gives rise to the construction of pseudo-data. This situation is notably exemplified by the computation of optimal state sequences from estimated hidden Markovian processes (optional argument `StateSequences` of the function `Estimate` set at `"ForwardBackward"` or `"Viterbi"`).

SEE ALSO

`Estimate` (1)(3).

ExtractDistribution

Extraction of a distribution from an object of type 'model'.

USAGE

```
ExtractDistribution(mixt, mixt_type)
ExtractDistribution(mixt, "Component", index)
ExtractDistribution(convol, "Elementary", index)
ExtractDistribution(convol, "Convolution")
ExtractDistribution(compound, compound_type)

ExtractDistribution(renew, renew_type)
ExtractDistribution(renew, "NbEvent", time)

ExtractDistribution(markov, markov_type, state)
ExtractDistribution(markov, markov_type, variable, output)

ExtractDistribution(top_param, position)
```

ARGUMENTS

`mixt` (`MIXTURE`),
`mixt_type` (`STRING`): type of distribution: `"Weight"` or `"Mixture"`,
`index` (`INT`): index of the elementary distribution,
`convol` (`CONVOLUTION`),
`compound` (`COMPOUND`),
`compound_type` (`STRING`): type of distribution: `"Sum"`, `"Elementary"` or `"Compound"`,

`renew` (`RENEWAL`),
`renew_type` (`STRING`): type of distribution: `"InterEvent"`, `"Backward"`, `"Forward"`, `"LengthBias"` or `"Mixture"`,
`time` (`INT`): observation period,

`markov` (`MARKOV`, `SEMI-MARKOV`, `HIDDEN_MARKOV`, `HIDDEN_SEMI-MARKOV`),
`markov_type` (`STRING`): type of distribution: `"Observation"`, `"FirstOccurrence"`, `"Recurrence"`, `"Sojourn"`, `"NbRun"` or `"NbOccurrence"`,
`state` (`INT`): state,
`variable` (`INT`): variable index,
`output` (`INT`): output,

`top_param` (`TOP_PARAMETERS`),
`position` (`INT`): position.

RETURNED OBJECT

If the arguments (`mixt_type`, `index`, `compound_type`, `renew_type`, `time`, `markov_type`, `state`, `variable`, `output`, `position`) defined an existing distribution, an object of type `DISTRIBUTION` is returned, otherwise no object is returned.

SEE ALSO

`Plot`, `Fit`, `Simulate` (1)(2)(3)(4).

ExtractHistogram

Extraction of a frequency distribution from an object of type ‘data’.

USAGE

```
ExtractHistogram(mixt_histo, mixt_type)
ExtractHistogram(mixt_histo, "Component", index)
ExtractHistogram(convol_histo, "Elementary", index)
ExtractHistogram(convol_histo, "Convolution")
ExtractHistogram(compound_histo, compound_type)

ExtractHistogram(vec1)
ExtractHistogram(vecn, variable)

ExtractHistogram(timev, timev_type)
ExtractHistogram(timev, "NbEvent", time)
ExtractHistogram(renewal_data, renew_type)

ExtractHistogram(seq, "Length")
ExtractHistogram(seq1, "Value")
ExtractHistogram(seqn, "Value", variable)
ExtractHistogram(discrete_seq1, seq_type, value)
ExtractHistogram(discrete_seqn, seq_type, variable, value)
ExtractHistogram(simul_seq1, "Observation", value)
ExtractHistogram(simul_seqn, "Observation", variable, value)

ExtractHistogram(top, "Main")
ExtractHistogram(top, "NbAxillary", position)
```

ARGUMENTS

mixt_histo (MIXTURE_DATA),
mixt_type (STRING): type of frequency distribution: "Weight" or "Mixture",
index (INT): index of the elementary frequency distribution,
convol_histo (CONVOLUTION_DATA),
compound_histo (COMPOUND_DATA),
compound_type (STRING): type of frequency distribution: "Sum", "Elementary" or "Compound",

vec1 (VECTORS): values,
vecn (VECTORS): vectors,
variable (INT): variable index,

timev (TIME_EVENTS, RENEWAL_DATA),
timev_type (STRING): type of frequency distribution: "ObservationTime" or "Mixture",
time (INT): observation period,
renewal_data (RENEWAL_DATA),
renew_type (STRING): type of frequency distribution: "InterEvent", "Backward", "Forward", "LengthBias" or "Inside",

seq (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),
seq1 (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): univariate sequences,
seqn (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): multivariate sequences,
discrete_seq1 (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): discrete univariate sequences,
discrete_seqn (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): discrete multivariate sequences,
simul_seq1 (MARKOV_DATA, SEMI-MARKOV_DATA): discrete simulated univariate sequences,
simul_seqn (MARKOV_DATA, SEMI-MARKOV_DATA): discrete simulated multivariate sequences,
seq_type (STRING): type of frequency distribution: "FirstOccurrence", "Recurrence", "Sojourn", "FinalRun", "NbRun" or "NbOccurrence",
value (INT): value,

top (TOPS),
position (INT): position.

Returned Object

If the arguments (mixt_type, index, compound_type, timev_type, time, renew_type, seq_type, variable, value, position) defined an existing frequency distribution, an object of type HISTOGRAM is returned, otherwise no object is returned.

See Also

ExtractVectors, Plot, Compare (1), ComparisonTest, Estimate (1)(2)(3)(4), Fit.

ExtractVectors

Extraction of vectors from global characteristics of sequences (length or counting characteristics).

USAGE

```
ExtractVectors(seq, "Length")
ExtractVectors(seq1, "NbRun", value)
ExtractVectors(seq1, "NbOccurrence", value)
ExtractVectors(seqn, "NbRun", variable, value)
ExtractVectors(seqn, "NbOccurrence", variable, value)
```

ARGUMENTS

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`),
`seq1` (`DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): univariate sequences,
`seqn` (`DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): multivariate sequences,
`value` (`INT`): value,
`variable` (`INT`): variable index.

RETURNED OBJECT

An object of type `VECTORS` is returned.

DESCRIPTION

The type of global characteristic is given by a key word chosen among `"Length"`, `"NbRun"` or `"NbOccurrence"`. In the case of counting characteristics, the key word `"NbRun"` or `"NbOccurrence"` should be followed by a variable index in the case of multivariate sequences, and by the value of interest.

SEE ALSO

`ExtractHistogram`, `Plot`, `MergeVariable`, `Compare` (2), `ContingencyTable`, `Regression`, `VarianceAnalysis`.

Fit

Fit of a frequency distribution by a theoretical distribution.

USAGE

```
Fit(histo, dist)
```

ARGUMENTS

`histo` (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`),
`dist` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`).

RETURNED OBJECT

No object returned.

DESCRIPTION

The result is displayed in the shell window (characteristics of the frequency and theoretical distributions, log-likelihood of the data for the theoretical distribution, information - maximum possible log-likelihood of the data -, χ^2 goodness of fit test).

BACKGROUND

The difference between the information measure and the log-likelihood is the Kullback-Leibler divergence from the observed distribution to the theoretical distribution. It is also one-half the deviance of the theoretical distribution.

Assume that a sample of size n is generated by a given random variable. The statistic D^2 measures the random deviation between the observed frequencies f_i and the theoretical frequencies np_i:

$$D^2 = \sum_{i=1}^{k} \frac{(f_i - np_i)^2}{np_i} \quad \text{with} \quad \sum_{i=1}^{k} f_i = n$$

If each theoretical frequency np_i is greater than a given threshold (between 1 and 5 according to the authors), D^2 has a χ^2 with $k-1$ degrees of freedom.

HiddenMarkov

Construction of an object of type HIDDEN_MARKOV from an ASCII file.

USAGE

```
HiddenMarkov(file_name, Length->40)
```

ARGUMENT

file_name (STRING)

OPTIONAL ARGUMENT

Length (INT): length of sequences for the computation of the intensity and counting characteristic distributions (default value: 20).

RETURNED OBJECT

If the construction succeeds, an object of type HIDDEN_MARKOV is returned, otherwise no object is returned.

BACKGROUND

A hidden Markov chain is constructed from an underlying Markov chain and nonparametric observation (or state-dependent) distributions.

SEE ALSO

Save, Compare (4)(5), Estimate (3), ComputeStateSequences, Simulate (3).

HiddenSemiMarkov

Construction of an object of type HIDDEN_SEMI-MARKOV from an ASCII file.

USAGE

```
HiddenSemiMarkov(file_name, Length->40, Counting->False)
```

ARGUMENT

file_name (STRING)

OPTIONAL ARGUMENTS

Length (INT): length of sequences for the computation of the intensity and counting characteristic distributions (default value: 20),
Counting (BOOL): computation of counting characteristic distributions (default value: True).

RETURNED OBJECT

If the construction succeeds, an object of type HIDDEN_SEMI-MARKOV is returned, otherwise no object is returned.

BACKGROUND

A hidden semi-Markov chain is constructed from an underlying semi-Markov chain (first-order Markov chain representing transition between distinct states and state occupancy distributions associated to the nonabsorbing states) and nonparametric observation (or state-dependent) distributions. The state occupancy distributions are defined as object of type DISTRIBUTION with the additional constraint that the minimum time spent in a given state is 1 (inf_bound ≥ 1).

SEE ALSO

Save, Compare (4)(5), Estimate (3), ComputeStateSequences, Simulate (3).

Histogram

Construction of a frequency distribution from an object of type ARRAY(INT) or from an ASCII file.

USAGE

```
Histogram(array)

Histogram(file_name)
```

ARGUMENTS

array (ARRAY(INT)),

file_name (STRING).

RETURNED OBJECT

If the construction succeeds, an object of type HISTOGRAM is returned, otherwise no object is returned.

DESCRIPTION

In the file syntax, the frequencies f_i for each possible value i are given in two columns. In the case of an argument of type (ARRAY(INT)), it is simply assumed that each array element represents one data item.

SEE ALSO

Save, Cluster, Merge, Shift, Transcode, ValueSelect, Compare (1), Estimate (1).

IndexExtract

Extraction of sub-sequences corresponding to a range of index parameters.

USAGE

```
IndexExtract(seq, min_index, MaxIndex->40)
```

ARGUMENTS

`seq` (`SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA`),
`min_index` (`INT`): minimum index parameter.

OPTIONAL ARGUMENT

`MaxIndex` (`INT`): maximum index parameter (default behaviour: the end of sequences is kept).

RETURNED OBJECT

If $0 \leq$ `min_index` $\leq$ (maximum index parameter if the optional argument `MaxIndex` is set) $<$ (maximum length of sequences), the returned object is of type `SEQUENCES` or `DISCRETE_SEQUENCES`, otherwise no object is returned. The returned object is of type `DISCRETE_SEQUENCES` if all the variables are of type `STATE`, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

`AddAbsorbingRun`, `Cluster`, `Cumulate`, `Difference`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`.

LengthSelect

Selection of sequences according to a length criterion.

USAGE

```
LengthSelect(seq, length, Mode->"Reject")
LengthSelect(seq, min_length, max_length, Mode->"Reject")
```

ARGUMENTS

seq (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),
length (INT): length,
min_length (INT): minimum length,
max_length (INT): maximum length.

OPTIONAL ARGUMENT

Mode (STRING): conservation or rejection of the selected sequences: "Keep" (the default) or "Reject".

RETURNED OBJECT

If length > 0 or if 0 < min_length ≤ max_length and if the range of lengths defined either by length or by min_length and max_length enables to select sequences, an object of type SEQUENCES or DISCRETE_SEQUENCES is returned, otherwise no object is returned. The returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract, Merge, MergeVariable, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect, VariableScaling.

Load

Restoration of an object of the STAT module from a binary file (object persistence).

USAGE

```
Load(file_name)
```

ARGUMENT

`file_name` (`STRING`).

RETURNED OBJECT

If the construction succeeds, the object is returned, otherwise no object is returned.

BACKGROUND

The persistence mechanism is implemented by the `Save` function with the optional argument `Format` set at `"Binary"` for saving and by the function `Load` for restoration.

SEE ALSO

`Save`.

Markov

Construction of a Markov chain from an ASCII file.

USAGE

```
Markov(file_name, Length->40)
```

ARGUMENT

`file_name` (`STRING`).

OPTIONAL ARGUMENT

`Length` (`INT`): length of sequences for the computation of the intensity and counting characteristic distributions (default value: 20).

RETURNED OBJECT

If the construction succeeds, an object of type `MARKOV` is returned, otherwise no object is returned.

BACKGROUND

The type `MARKOV` covers both homogeneous Markov chains and non-homogeneous Markov chains. A Markov chain is said to be non-homogeneous if for at least one state, the self-transition probability is related to the index parameter by a simple parametric function (monomolecular or logistic).

SEE ALSO

`Save`, `Compare` (4)(5), `Simulate` (3).

Merge

Merging of objects of the same 'data' type or merging of sample correlation functions.

USAGE

```
Merge(histo1, histo2,…)

Merge(vec1, vec2,…)

Merge(timev1, timev2,…)

Merge(seq1, seq2,…)
Merge(discrete_seq1, discrete_seq2,…)

Merge(top1, top2,…)

Merge(correl1, correl2,…)
```

ARGUMENTS

histo1, histo2, … (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),

vec1, vec2, … VECTORS,

timev1, timev2, … (TIME_EVENTS, RENEWAL_DATA),

seq1, seq2, … (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),
discrete_seq1, discrete_seq2, … (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),

top1, top2, … TOPS,

correl1, correl2, … (CORRELATION).

RETURNED OBJECT

- If the arguments are of type HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA, an object of type HISTOGRAM is returned.
- If the arguments are of type VECTORS and if the vectors have the same number of variables, an object of type VECTORS is returned, otherwise no object is returned.
- If the arguments are of type TIME_EVENTS, RENEWAL_DATA, an object of type TIME_EVENTS is returned.
- If the arguments are of type SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA and if the sequences have the same number of variables, an object of type SEQUENCES is returned, otherwise no object is returned.
- If the arguments are of type DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA and if the sequences have the same number of variables, an object of type DISCRETE_SEQUENCES is returned, otherwise no object is returned.

- If the arguments are of type TOPS, an object of type TOPS is returned.
- If the arguments are of type CORRELATION, an object of type CORRELATION is returned.

SEE ALSO

Cluster, Shift, Transcode, ValueSelect, MergeVariable, SelectIndividual, SelectVariable, NbEventSelect, TimeScaling, TimeSelect, AddAbsorbingRun, Cumulate, Difference, IndexExtract, LengthSelect, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, VariableScaling, RemoveApicalInternodes.

MergeVariable

Merging of variables.

USAGE

```
MergeVariable (vec1, vec2,…, RefSample->2)

MergeVariable (seq1, seq2,…, RefSample->2)
```

ARGUMENTS

vec1, vec2, … (VECTORS),

seq1, seq2, … (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA).

OPTIONAL ARGUMENT

RefSample (INT): reference sample to define individual identifiers (the default: no reference sample).

RETURNED OBJECT

- If the arguments are of type VECTORS and if the number of vectors is the same for each sample, an object of type VECTORS is returned, otherwise no object is returned.
- If the arguments are of type SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA, if all the variables are of type STATE, and if the number and the lengths of sequences are the same for each sample, an object of type SEQUENCES or DISCRETE_SEQUENCES is returned, otherwise no object is returned. The returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

Merge, Cluster, Shift, Transcode, ValueSelect, SelectIndividual, SelectVariable, AddAbsorbingRun, Cumulate, Difference, IndexExtract, LengthSelect, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, VariableScaling.

Mixture

Construction of a mixture of distributions from elementary distributions and associated weights or from an ASCII file.

USAGE

```
Mixture(weight1, dist1, weight2, dist2,…)

Mixture(file_name)
```

ARGUMENTS

`weight1`, `weight2`, … (`REAL`): weights of each component. These weights should sum to one (they constitute a discrete distribution).
`dist1`, `dist2`, … (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`): elementary distributions (or components),

`file_name` (`STRING`).

RETURNED OBJECT

If the construction succeeds, an object of type `MIXTURE` is returned, otherwise no object is returned.

BACKGROUND

A mixture is a parametric model of classification where each elementary distribution or component represents a class with its associated weight.

SEE ALSO

`Save`, `Estimate` (1), `Simulate` (1).

ModelSelectionTest

Test of the order of a Markov chain or of a possible partition of the original state space (lumpability hypothesis).

USAGE

```
ModelSelectionTest(seq, order0, order1)

ModelSelectionTest(seq, states, Order->2)
```

ARGUMENTS

`seq` (`DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`),

`order0` (`INT`): order corresponding to the null hypothesis,
`order1` (`INT`): order corresponding to the alternative hypothesis,

`states`, … (`ARRAY(INT)`): new states corresponding to a partition of the original state space,

OPTIONAL ARGUMENT

`Order` (`INT`): order of the Markov chain (default value: 1). This optional argument can only be used when testing a lumpability hypothesis.

RETURNED OBJECT

No object returned.

DESCRIPTION

The result of the test is displayed in the shell window.

BACKGROUND

if $0 \leq$ `order0` $<$ `order1`, a likelihood ratio statistic is constructed from the observed frequencies of transitions according to the two hypotheses of order of the Markov chain. This statistic has a χ^2 distribution with k degrees of freedom (k = (number of free parameters of the `order1` Markov chain) - (number of free parameters of the `order0` Markov chain)).

When testing a lumpability hypothesis, a likelihood ratio statistic is constructed from the observed frequencies of transitions (and observations in the case of the lumped process) according to the two hypotheses. This statistic has a χ^2 distributions with k degrees of freedom (k = (number of free parameters of the original Markov chain) - (number of free parameters of the lumped process)).

SEE ALSO

`Estimate` (3).

MovingAverage

Extraction of trends or residuals using a symmetric smoothing filter.

USAGE

```
MovingAverage(seq, filter, Variable->1, BeginEnd->True,
              Output->"Residual",
              FileName->"filtered_sequences")
MovingAverage(seq, frequencies, Variable->1, BeginEnd->True,
              Output->"Residual",
              FileName->"filtered_sequences")
MovingAverage(seq, dist, Variable->1, BeginEnd->True,
              Output->"Residual",
              FileName->"filtered_sequences")
```

ARGUMENTS

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`)
`filter` (`ARRAY(REAL)`): filter values on the half width i.e. from one extremity to the central value (with the constraint $2\sum_{i=1}^{m-1} \texttt{filter@i} + \texttt{filter@m} = 1$),
`frequencies` (`ARRAY(INT)`): frequencies defining the filter,
`dist` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`): symmetric distribution, whose size of the support is even, defining the filter (for instance `Distribution("BINOMIAL", 0, 4, 0.5)`),

OPTIONAL ARGUMENTS

`Variable` (`INT`): variable index,
`BeginEnd` (`BOOL`): begin and end of sequences filtered or suppresses (default value: `False`),
`Output` (`STRING`): `"Trend"` (the default), `"Residual"` or `"DivisionResidual"`,
`FileName` (`STRING`): name of file of non-rounded filtered sequences.

BACKGROUND

Consider a symmetric smoothing filter of half width q applied to a sequence of length τ. Whenever a symmetric smoothing filter is chosen, there is likely to be an end-effect problem for $t < q$ or $t > \tau - q - 1$. We chose to apply the following solution to the first and the last q terms of the sequences: we define $X_t := X_0$ for $t < 0$ and $X_t := X_{\tau-1}$ for $t > \tau - 1$.

RETURNED OBJECT

An object of type `SEQUENCES` or `DISCRETE_SEQUENCES` is returned. An object of type `DISCRETE_SEQUENCES` is returned if all the variables are of type `STATE`, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract, LengthSelect, Merge, MergeVariable, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect, VariableScaling.

NbEventSelect

Selection of data item of type {time interval between two observation dates, number of events occurring between these two observation dates} according to a number of events criterion.

USAGE

```
NbEventSelect(timev, min_nb_event, max_nb_event)
```

ARGUMENTS

`timev` (`TIME_EVENTS`, `RENEWAL_DATA`),
`min_nb_event` (`INT`): minimum number of events,
`max_nb_event` (`INT`): maximum number of events.

RETURNED OBJECT

If $0 \leq$ `min_nb_event` $\leq$ `max_nb_event` and if the range of number of events defined by `min_nb_event` and `max_nb_event` enables to select data items of type {time interval between two observation dates, number of events}, an object of type `TIME_EVENTS` is returned, otherwise no object is returned.

SEE ALSO

`Merge`, `TimeScaling`, `TimeSelect`.

Plot, NewPlot

Graphical output of an object of the STAT module using the GNUPLOT software.

USAGE

```
Plot(obj1, Title->"Distribution")
w1 = NewPlot(obj1, Title->"Distribution")
Plot(Window->w1)
w1 = NewPlot()

Plot(vec1, Title->"Values")
Plot()
Plot(vecn, variable, Title->"Vectors")
Plot(variable)

Plot(obj2, type, Title->"Sequences")
Plot(type)
Plot(obj3, type, variable, Title->"Multivariate sequences")
Plot(type, variable)

Plot(dist1, dist2,…, Title->"Family of distributions")
Plot(histo1, histo2,…,
     Title->"Family of frequency distributions")

Plot(seq, ViewPoint->"Data")

Plot(dist, ViewPoint->"Survival", Title->"Survival rates")
Plot(histo, ViewPoint->"Survival", Title->"Survival rates")

Plot(hmc, identifier, ViewPoint->"StateProfile",
     Title->"Smoothed probabilities")
Plot(hsmc, identifier, ViewPoint->"StateProfile",
     Title->"Smoothed probabilities")
```

ARGUMENTS

`obj1` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`, `HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`, `RENEWAL`, `TIME_EVENTS`, `RENEWAL_DATA`, `SEQUENCES`, `DISTANCE_MATRIX`, `TOP_PARAMETERS`, `TOPS`),

`vec1` (`VECTORS`): values,
`vecn` (`VECTORS`): vectors,
`variable` (`INT`): variable index,

`obj2` (`MARKOV`, `SEMI-MARKOV`, `HIDDEN_MARKOV`, `HIDDEN_SEMI-MARKOV`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): Markovian model for discrete univariate sequences or discrete univariate sequences,
`obj3`: (`MARKOV`, `SEMI-MARKOV`, `HIDDEN_MARKOV`, `HIDDEN_SEMI-MARKOV`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`): Markovian model for discrete multivariate sequences or discrete multivariate sequences,

`type` (`STRING`): type of graphical outputs in the case of Markovian models or sequences: `"SelfTransition"`, `"Observation"`, `"Intensity"`, `"FirstOccurrence"`, `"Recurrence"`, `"Sojourn"` or `"Counting"`,

`dist1`, `dist2`, … (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`),
`histo1`, `histo2`, … (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`),

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`, `TOPS`),

`dist` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`),
`histo` (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`),

`hmc` (`HIDDEN_MARKOV`),
`hsmc` (`HIDDEN_SEMI-MARKOV`),
`identifier` (`INT`): identifier of a sequence.

Optional Arguments

`Window` (`WINDOW`): window (the default: last used window). This optional argument cannot be used with the function `NewPlot`.
`ViewPoint` (`STRING`): point of view on the object (`"Data"` or `"Survival"` or `"StateProfile"`). This optional argument can be set at `"Data"` only if the first mandatory argument is of type `SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA` or `TOPS`, can be set at `"Survival"` only if the first mandatory argument is of type `DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`, `HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA` or `COMPOUND_DATA` and can be set at `"StateProfile"` only if the first mandatory argument is of type `HIDDEN_MARKOV` or `HIDDEN_SEMI-MARKOV`.
`Title` (`STRING`): graphic title (the default: no title).

Returned Object

No object returned by the function `Plot`. An object of type `WINDOW` is returned by the function `NewPlot`.

Description

In the case of Markovian models or sequences, the graphical outputs are grouped as follows:
– `"SelfTransition"`: self-transition probability as a function of the index parameter (non-homogeneous Markov chain),
– `"Observation"`: observation distributions attached to each state of the underlying (semi-)Markov chain (lumped processes or hidden Markovian processes),
– `"Intensity"`: (empirical) probabilities of states/outputs as a function of the index parameter,
– `"FirstOccurrence"`: (frequency) distributions of the time-up to the first occurrence of a state/output (or first-passage time in a state/output distributions),

- "Recurrence" (frequency) distributions of the recurrence time in a state/output,
- "Sojourn": (frequency) distributions of the sojourn time in a state/output (or state/output occupancy distributions). For the frequency distributions extracted from sequences, the sojourn times in the last visited states which are considered as censored are isolated.
- "Counting": counting (frequency) distributions (either distributions of the number of runs (or clumps) of a state/output per sequence or distributions of the number of occurrences of a state/output per sequence).

BACKGROUND

Graphical output of a (frequency) distribution and the associate hazard or survival rates (ViewPoint->"Survival"): It is assumed that the (frequency) distribution represents lifetime and the hazard or survival rates are deduced from this lifetime distribution.

Graphical output of the state profile given by the smoothed probabilities $P(S_t = j | X_0^{\tau-1} = x_0^\tau)$ as a function of the index parameter t computed from the parameters of a hidden Markovian model for the sequence x_0^τ (ViewPoint->"StateProfile").

SEE ALSO

Display, Save.

RecurrenceTimeSequences

Computation of recurrence time sequences for a given value (and for a given variable).

USAGE

```
RecurrenceTimeSequences(seq1, value)
RecurrenceTimeSequences(seqn, variable, value)
```

ARGUMENTS

seq1 (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): univariate sequences,
seqn (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): multivariate sequences,
variable (INT): variable index,
value (INT): value.

RETURNED OBJECT

If the selected variable is of type STATE and if value is a possible value, an object of type SEQUENCES or DISCRETE_SEQUENCES is returned, otherwise no object is returned. The returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract, LengthSelect, Merge, MergeVariable, MovingAverage, RemoveRun, Reverse, SegmentationExtract, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect, VariableScaling.

Regression

Simple regression (with a single explanatory variable).

USAGE

```
Regression(vec, "Linear", explanatory_variable,
           response_variable)

Regression(vec, "MovingAverage", explanatory_variable,
           response_variable, filter,
           Algorithm->"LeastSquares")
Regression(vec, "MovingAverage", explanatory_variable,
           response_variable, frequencies,
           Algorithm->"LeastSquares")
Regression(vec, "MovingAverage", explanatory_variable,
           response_variable, dist, Algorithm->"LeastSquares")

Regression(vec, "NearestNeighbours", explanatory_variable,
           response_variable, span, Weighting->False)
```

ARGUMENTS

vec (VECTORS): vectors,
explanatory_variable (INT): index of the explanatory variable,
response_variable (INT): index of the response variable,

filter (ARRAY(REAL)): filter values on the half width i.e. from one extremity to the central value (with the constraint $2\sum_{i=1}^{m-1} \texttt{filter@}i + \texttt{filter@}m = 1$),

frequencies (ARRAY(INT)): frequencies defining the filter,

dist (DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND): symmetric distribution, whose size of the support is even, defining the filter (for instance Distribution("BINOMIAL", 0, 4, 0.5)),

span (REAL): proportion of individuals in each neighbourhood.

OPTIONAL ARGUMENTS

Algorithm (STRING): type of algorithm: "Averaging" (the default) or "LeastSquares". This optional argument can only be used if the second mandatory argument specifying the regression type is "MovingAverage".

Weighting (BOOL): weighting or not of the neighbours according to their distance to the computed point (default value: True). This optional argument can only be used if the second mandatory argument specifying the regression type is "NearestNeighbours".

RETURNED OBJECT

An object of type REGRESSION is returned.

RemoveApicalInternodes

Removal of the apical internodes of the parent shoot of a ‘top’.

USAGE

```
RemoveApicalInternodes(top, nb_internode)
```

ARGUMENTS

`top` (`TOPS`),
`nb_internode` (`INT`): number of removed internodes.

RETURNED OBJECT

If `nb_internode` > 0 and if the removed internodes do not bear offspring shoots, an object of type `TOPS` is returned, otherwise no object is returned.

SEE ALSO

`Merge`, `Reverse`, `SelectIndividual`.

RemoveRun

Removal of the first or last run of a given value (for a given variable) in a sequence.

USAGE

```
RemoveRun(seq1, value, position, MaxLength->4)
RemoveRun(seqn, variable, value, position, MaxLength->4)
```

ARGUMENTS

`seq1` (`SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA`): univariate sequences,
`seqn` (`SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA`): multivariate sequences,
`variable` (`INT`): variable index,
`value` (`INT`): value,
`position` (`STRING`): position of the removed run: `"Begin"` or `"End"`.

OPTIONAL ARGUMENT

`MaxLength` (`INT`): maximum length of the removed run (default behaviour: the entire run is removed).

RETURNED OBJECT

If `variable` is a valid index of variable, if `value` is a possible value and if `MaxLength` > 0, an object of type `SEQUENCES` or `DISCRETE_SEQUENCES` is returned, otherwise no object is returned. The returned object is of type `DISCRETE_SEQUENCES` if all the variables are of type `STATE`, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

`AddAbsorbingRun`, `Cluster`, `Cumulate`, `Difference`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `Reverse`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`.

Renewal

Construction of a (either ordinary or equilibrium) renewal process from an inter-event distribution or from an ASCII file.

USAGE

```
Renewal("BINOMIAL", inf_bound, sup_bound, proba,
        Type->"Equilibriun", ObservationTime->40)
Renewal("POISSON", inf_bound, param, Type->"Equilibriun",
        ObservationTime->40)
Renewal("NEGATIVE_BINOMIAL", inf_bound, param, proba,
        Type->"Equilibriun", ObservationTime->40)

Renewal(inter_event, Type->"Equilibriun", ObservationTime->40)

Renewal(file_name, Type->"Equilibriun", ObservationTime->40)
```

ARGUMENTS

`inf_bound` (`INT`): lower bound to the range of possible values (shift parameter),

`sup_bound` (`INT`): upper bound to the range of possible values (only relevant for binomial or uniform distributions),

`param` (`INT`, `REAL`): parameter of either the Poisson distribution or the negative binomial distribution.

`proba` (`INT`, `REAL`): probability of 'success' (only relevant for binomial or negative binomial distributions).

Remark: the names of the parametric discrete distributions can be summarized by their first letters: `"B"` (`"BINOMIAL"`), `"P"` (`"POISSON"`), `"NB"` (`"NEGATIVE_BINOMIAL"`),

`inter_event` (`DISTRIBUTION`, `MIXTURE`, `CONVOLUTION`, `COMPOUND`): inter-event distribution,

`file_name` (`STRING`).

OPTIONAL ARGUMENTS

`Type` (`STRING`): type of renewal process: `"Ordinary"` or `"Equilibriun"` (the default).

`ObservationTime` (`INT`): length of the observation period for the computation of the intensity and counting distributions (default value: 20),

RETURNED OBJECT

If the construction succeeds, an object of type `RENEWAL` is returned, otherwise no object is returned.

BACKGROUND

A renewal process is built from a discrete distribution termed the inter-event distribution which represents the time interval between consecutive events. Two types of renewal processes are available:

- ordinary renewal process where the start of the observation period coincides with the occurrence time of an event (synchronism assumption),
- equilibrium or stationary renewal process where the start of the observation period is independent of the process which generates the data (asynchronism assumption).

In the case where the arguments are the name and the parameters of the inter-event distribution, the constraints on parameters described in the definition of the syntactic form of the type `DISTRIBUTION` apply.

SEE ALSO

`Save`, `Simulate` (2).

Reverse

Reversing of sequences or 'tops'.

USAGE

```
Reverse(seq)
Reverse(discrete_seq)

Reverse(top)
```

ARGUMENTS

`seq` (`SEQUENCES`),
`discrete_seq` (`DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA`),

`top` (`TOPS`).

RETURNED OBJECT

If the argument is of type `SEQUENCES`, an object of type `SEQUENCES` is returned. If the argument is of type `DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA`, an object of type `DISCRETE_SEQUENCES` is returned. If the argument is of type `TOPS`, an object of type `TOPS` is returned.

SEE ALSO

`AddAbsorbingRun`, `Cluster`, `Cumulate`, `Difference`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`, `RemoveApicalInternodes`.

Save

Saving of an object of the STAT module in a file.

USAGE

```
Save(obj, file_name, Format->"ASCII", Detail->2)

Save(histo, file_name, ViewPoint->"Data")
Save(vec, file_name, ViewPoint->"Data", Detail->2)
Save(timev, file_name, ViewPoint->"Data")
Save(seq, file_name, ViewPoint->"Data", Format->"Line",
     Detail->2)

Save(dist, file_name, ViewPoint->"Survival",
     Format->"SpreadSheet")
Save(histo, file_name, ViewPoint->"Survival",
     Format->"SpreadSheet")

Save(hmc, ViewPoint->"StateProfile", Sequence->1,
     Format->"SpreadSheet")
Save(hsmc, ViewPoint->"StateProfile", Sequence->1,
     Format->"SpreadSheet")
```

ARGUMENTS

obj: object of the STAT module (except objects of type VECTOR_DISTANCE),
file_name (STRING),

histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),
vec (VECTORS),
timev (TIME_EVENTS, RENEWAL_DATA),
seq (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA,
SEMI-MARKOV_DATA, TOPS).

dist (DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND),

hmc (HIDDEN_MARKOV),
hsmc (HIDDEN_SEMI-MARKOV).

OPTIONAL ARGUMENTS

ViewPoint (STRING): point of view on the object ("Data" or "Survival" or "StateProfile"). This optional argument can be set at "Data" only if the first argument is of type VECTORS, SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA or TOPS, can be set at "Survival" only if the first argument is of type DISTRIBUTION, MIXTURE, CONVOLUTION, COMPOUND, HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA or COMPOUND_DATA and can be set at "StateProfile" only if the first argument is of type HIDDEN_MARKOV or HIDDEN_SEMI-MARKOV.

Detail (INT): level of detail: 1 (default value) or 2. This optional argument can only be used if the optional argument ViewPoint is not set, or if the optional argument

ViewPoint is set at "Data" and if the first mandatory argument is of type VECTORS, SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA or TOPS.

Format (STRING): file format: "ASCII" (default format), "Binary" or "SpreadSheet". These file formats cannot be specified if the optional argument ViewPoint is set at "Data". The optional argument Format can only be set at "Binary" if the optional argument ViewPoint is not set.

Format (STRING): format of sequences (only relevant for multivariate sequences): "Column" (the default) or "Line". This optional argument can only be used if the optional argument ViewPoint is set at "Data", and hence, if the first mandatory argument is of type SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA or TOPS.

Sequence (INT): identifier of a sequence. This optional argument can only be used if the optional argument ViewPoint is set at "StateProfile", and hence, if the first mandatory argument is of type HIDDEN_MARKOV or HIDDEN_SEMI-MARKOV.

DESCRIPTION

Saving of sets of sequences or 'tops' (ViewPoint->"Data"): the format "Column" corresponds to the ASCII file syntax for objects of type SEQUENCES or TOPS. For a given value of the index parameter, the different variables are successively written. With the format "Line", the univariate sequence for each variable are written on consecutive lines. In the case of univariate sequences, the two formats give the same file.

Saving of a (frequency) distribution and the associate hazard or survival rates (ViewPoint->"Survival"): It is assumed that the (frequency) distribution represents lifetime and the hazard or survival rates are deduced from this lifetime distribution.

Saving of the state profile given by the smoothed probabilities $P(S_t = j | X_0^{\tau-1} = x_0^\tau)$ as a function of the index parameter t computed from the parameters of a hidden Markovian model for the sequence x_0^τ (ViewPoint->"StateProfile").

BACKGROUND

The persistence mechanism is implemented by the Save function with the optional argument Format set at "Binary" for saving and by the function Load for restoration.

RETURNED OBJECT

No object returned.

SEE ALSO

Display, Plot, Compound, Convolution, Distribution, HiddenMarkov, HiddenSemiMarkov, Histogram, Markov, Mixture, Renewal, SemiMarkov, Sequences, TimeEvents, Tops, TopParameters, Vectors, Load.

SegmentationExtract

Extraction by segmentation of sub-sequences.

USAGE

```
SegmentationExtract(seqn, variable, value, Mode->"Reject")
SegmentationExtract(seqn, variable, values, Mode->"Reject")
```

ARGUMENTS

seqn (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): multivariate sequences,
variable (INT): variable index,
value (INT): value,
values (ARRAY(INT)): values.

OPTIONAL ARGUMENT

Mode (STRING): conservation or rejection of the selected sub-sequences: "Keep" (the default) or "Reject".

RETURNED OBJECT

If all the variables are of type STATE, if variable is a valid index of variable and if either value or values@1, …, values@n are possible values, an object of type SEQUENCES or DISCRETE_SEQUENCES is returned, otherwise no object is returned. The returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

DESCRIPTION

Sub-sequences corresponding to run of value or values@1, …, values@n are extracted (or to all the possible values except value or values@1, …, values@n) are extracted.

SEE ALSO

AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract, LengthSelect, Merge, MergeVariable, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect, VariableScaling.

SelectIndividual

Selection of vectors, sequences, tops or patterns (in a dissimilarity matrix).

USAGE

```
SelectIndividual(vec, identifiers, Mode->"Reject")

SelectIndividual(seq, identifiers, Mode->"Reject")

SelectIndividual(top, identifiers, Mode->"Reject")

SelectIndividual(dist_matrix, identifiers, Mode->"Reject")
```

ARGUMENTS

vec (VECTORS),

seq (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),

top (TOPS),

dist_matrix (DISTANCE_MATRIX),

identifiers (ARRAY(INT)): identifiers.

OPTIONAL ARGUMENT

Mode (STRING): conservation or rejection of the selected individuals: "Keep" (the default) or "Reject".

RETURNED OBJECT

If identifiers@1, …, identifiers@n are valid identifiers of vectors (respectively sequences, tops or patterns compared in a dissimilarity matrix), an object of type VECTORS (respectively SEQUENCES or DISCRETE_SEQUENCES, TOPS or DISTANCE_MATRIX) is returned, otherwise no object is returned. In the case of a first argument of type SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA, the returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

Cluster, Merge, Shift, Transcode, ValueSelect, MergeVariable, SelectVariable, AddAbsorbingRun, Cumulate, Difference, IndexExtract, LengthSelect, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, VariableScaling, RemoveApicalInternodes, Symmetrize.

SelectVariable

Selection of variables.

USAGE

```
SelectVariable(vec, variable, Mode->"Reject")
SelectVariable(vec, variables, Mode->"Reject")

SelectVariable(seq, variable, Mode->"Reject")
SelectVariable(seq, variables, Mode->"Reject")
```

ARGUMENTS

vec (VECTORS),

seq (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA),

variable (INT): variable index.
variables (ARRAY(INT)): variable indices.

OPTIONAL ARGUMENT

Mode (STRING): conservation or rejection of the selected variables: "Keep" (the default) or "Reject".

RETURNED OBJECT

If either variable or variables@1, …, variables@n are valid indices of variables, an object of type VECTORS (respectively SEQUENCES or DISCRETE_SEQUENCES) is returned, otherwise no object is returned. In the case of a first argument of type SEQUENCES, the returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

Cluster, Merge, Shift, Transcode, SelectIndividual, ValueSelect, MergeVariable, AddAbsorbingRun, Cumulate, Difference, IndexExtract, LengthSelect, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, VariableScaling.

SemiMarkov

Construction of a semi-Markov chain from an ASCII file.

USAGE

```
SemiMarkov(file_name, Length->40, Counting->True)
```

ARGUMENT

`file_name` (`STRING`).

OPTIONAL ARGUMENTS

`Length` (`INT`): length of sequences for the computation of the intensity and counting characteristic distributions (default value: 20),
`Counting` (`BOOL`): computation of counting characteristic distributions (default value: `True`).

RETURNED OBJECT

If the construction succeeds, an object of type `SEMI-MARKOV` is returned, otherwise no object is returned.

BACKGROUND

A semi-Markov chain is constructed from a first-order Markov chain representing transition between distinct states and state occupancy distributions associated to the nonabsorbing states. The state occupancy distributions are defined as object of type `DISTRIBUTION` with the additional constraint that the minimum time spent in a given state is at least 1 (`inf_bound` ≥ 1).

SEE ALSO

`Save`, `Compare` (4)(5), `Simulate` (3).

Sequences

Construction of a set of sequences from multidimensional arrays of integers, from data generated by a renewal process or from an ASCII file.

USAGE

```
Sequences(array1, Identifiers->[1, 8, 12])
Sequences(arrayn, Identifiers->[1, 8, 12],
          IndexParameter->"Position")

Sequences(timev)

Sequences(file_name)
```

ARGUMENTS

`array1` (`ARRAY(ARRAY(INT))`): input data for univariate sequences
`arrayn` (`ARRAY(ARRAY(ARRAY(INT)))`): input data for multivariate sequences,

`timev` (`RENEWAL_DATA`),

`file_name` (`STRING`).

OPTIONAL ARGUMENTS

`Identifiers` (`ARRAY(INT)`): explicit identifiers of sequences. This optional argument can only be used if the first argument is of type `ARRAY(ARRAY(INT/ARRAY(INT)))`.
`IndexParameter` (`STRING`): type of the explicit index parameter: `"Position"` or `"Time"` (the default: implicit discrete index parameter starting at 0). This optional argument can only be used if the first argument is of type `ARRAY(ARRAY(INT/ARRAY(INT)))`.

RETURNED OBJECT

If the construction succeeds, an object of type `SEQUENCES` or `DISCRETE_SEQUENCES` is returned, otherwise no object is returned. The returned object is of type `DISCRETE_SEQUENCES` if all the variables are of type `STATE`, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

DESCRIPTION

The data structure of type `ARRAY(ARRAY(ARRAY(INT)))` should be constituted at the most internal level of arrays of constant size. If the optional argument `IndexParameter` is set at `"Position"` or `"Time"`, the data structure of type `ARRAY(ARRAY(ARRAY(INT)))` is constituted at the most internal level of arrays of size $1+n$ (index parameter, n variables attached to the explicit index parameter). If the optional argument `IndexParameter` is set at `"Position"`, only the index parameter of the last array of size $1+n$ is considered and the first component of successive elementary arrays (representing the index parameter) should be increasing. If the

optional argument `IndexParameter` is set at `"Time"`, the first component of successive elementary arrays should be strictly increasing.

SEE ALSO

`Save`, `ExtractHistogram`, `ExtractVectors`, `AddAbsorbingRun`, `Cluster`, `Cumulate`, `Difference`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`, `ComputeCorrelation`, `ComputePartialAutoCorrelation`, `ComputeSelfTransition`, `Compare` (3)(4)(5), `Estimate` (3), `ComputeStateSequences`, `Simulate` (3).

Shift

Shifting of values.

USAGE

```
Shift(histo, param)

Shift(vec1, param)
Shift(vecn, variable, param)

Shift(seq1, param)
Shift(seqn, variable, param)
```

ARGUMENTS

histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),

param (INT): shifting parameter,

vec1 (VECTORS): values,
vecn (VECTORS): vectors,
variable (INT): variable index,

seq1 (SEQUENCES): univariate sequences,
seqn (SEQUENCES): multivariate sequences.

RETURNED OBJECT

If the shifting makes that the lower bound to the possible values is positive, an object of type HISTOGRAM (respectively VECTORS, SEQUENCES or SEQUENCES) is returned, otherwise no object is returned. In the case of a first argument of type SEQUENCES, the returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

Cluster, Merge, Transcode, ValueSelect, MergeVariable, SelectIndividual, SelectVariable, AddAbsorbingRun, Cumulate, Difference, LengthSelect, MovingAverage, IndexExtract, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, VariableScaling.

Simulate (1)

Generation of a random sample from a distribution.

USAGE

```
Simulate(dist, size)
Simulate(mixt, size)
Simulate(convol, size)
Simulate(compound, size)
```

ARGUMENTS

`dist` (`DISTRIBUTION`),
`mixt` (`MIXTURE`),
`convol` (`CONVOLUTION`),
`compound` (`COMPOUND`),
`size` (`INT`): sample size.

RETURNED OBJECT

- If the first argument is of type `DISTRIBUTION` and if $0 <$ `size` ≤ 1000000, an object of type `HISTOGRAM` is returned, otherwise no object is returned.
- If the first argument is of type `MIXTURE` and if $0 <$ `size` ≤ 1000000, an object of type `MIXTURE_DATA` is returned, otherwise no object is returned.
- If the first argument is of type `CONVOLUTION` and if $0 <$ `size` ≤ 1000000, an object of type `CONVOLUTION_DATA` is returned, otherwise no object is returned.
- If the first argument is of type `COMPOUND` and if $0 <$ `size` ≤ 1000000, an object of type `COMPOUND_DATA` is returned, otherwise no object is returned.

The returned object of type `HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA` or `COMPOUND_DATA` contains both the simulated sample and the model used for simulation.

SEE ALSO

`Distribution`, `Mixture`, `Convolution`, `Compound`, `ExtractHistogram`.

Simulate (2)

Generation of a random sample from a renewal process.

USAGE

```
Simulate(renew, type, time_histo)
Simulate(renew, type, size, time)
Simulate(renew, type, size, timev)
```

ARGUMENTS

`renew` (`RENEWAL`),
`type` (`STRING`): type of renewal process: `"Ordinary"` or `"Equilibriun"`,
`time_histo` (`HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA`): frequency distribution of the length of the observation period,
`size` (`INT`): sample size,
`time` (`INT`): length of the observation period,
`timev` (`TIME_EVENTS`, `RENEWAL_DATA`),

RETURNED OBJECT

If $0 <$ (sample size) ≤ 1000000, if (minimum length of the observation period) ≥ 2 and if (maximum length of the observation period) ≤ 1000, an object of type `RENEWAL_DATA` is returned, otherwise no object is returned. The returned object contains both the simulated sample (not only count data but also time sequences) and the model used for simulation.

DESCRIPTION

If the fourth argument is of type `TIME_EVENTS` or `RENEWAL_DATA`, the simulated sample has the same distribution of length of the observation period than the original sample given by this fourth argument. This simulation mode is particularly useful to study the effects of length biasing.

SEE ALSO

`Renewal`, `ExtractHistogram`.

Simulate (3)

Generation of a sample of sequences from a (hidden) Markovian process.

USAGE

```
Simulate(markov, length_histo)
Simulate(markov, size, length)
Simulate(markov, size, seq)
Simulate(semi_markov, length_histo, Counting->False)
Simulate(semi_markov, size, length, Counting->False)
Simulate(semi_markov, size, seq, Counting->False)
```

ARGUMENTS

`markov` (`MARKOV`, `HIDDEN_MARKOV`),
`semi_markov` (`SEMI-MARKOV`, `HIDDEN_SEMI-MARKOV`),
`length_histo` (`HISTOGRAM`, `MIXTURE_DATA`, `CONVOLUTION_DATA`, `COMPOUND_DATA`): frequency distribution of sequence lengths,
`size` (`INT`): sample size,
`length` (`INT`): sequence length,
`seq` (`DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`),

OPTIONAL ARGUMENTS

`Counting` (`BOOL`): computation of counting distributions (default value: `True`).

RETURNED OBJECT

- If the first argument is of type `MARKOV` or `HIDDEN_MARKOV`, if $0 <$ (sample size) ≤ 100000, if (minimum sequence length) ≥ 2, if (maximum sequence length) ≤ 1000 and if (cumulative sequence length) ≤ 1000000, an object of type `MARKOV_DATA` is returned, otherwise no object is returned.
- If the first argument is of type `SEMI-MARKOV` or `HIDDEN_SEMI-MARKOV`, if $0 <$ (sample size) ≤ 100000, if (minimum sequence length) ≥ 2, if (maximum sequence length) ≤ 1000 and if (cumulative sequence length) ≤ 1000000, an object of type `SEMI-MARKOV_DATA` is returned, otherwise no object is returned.

The returned object contains both the simulated sequences and the model used for simulation.

DESCRIPTION

If the third argument is of type `DISCRETE_SEQUENCES`, `MARKOV_DATA` or `SEMI-MARKOV_DATA`, the simulated sequences has the same length distribution than the original sample given by this third argument. This simulation mode is particularly useful to study the effects of length biasing.

SEE ALSO

`Markov`, `SemiMarkov`, `HiddenMarkov`, `HiddenSemiMarkov`, `ExtractHistogram`.

Simulate (4)

Generation of 'tops' from 'top' parameters.

USAGE

```
Simulate(top_param, size, nb_trial, NbAxillary->2)
```

ARGUMENTS

`top_param` (`TOP_PARAMETERS`),
`size` (`INT`): sample size,
`nb_trial` (`INT`): number of Bernoulli trials for the growth of the parent shoot,

OPTIONAL ARGUMENT

`NbAxillary` (`INT`): number of offspring shoots per node (default value: 1, should be ≤ 4).

RETURNED OBJECT

If $0 <$ `size` ≤ 100000 and if $0 <$ `nb_trial` ≤ 1000, an object of type `TOPS` is returned, otherwise no object is returned. The returned object contains both the simulated 'tops' and the model used for simulation.

SEE ALSO

`TopParameter`, `ExtractHistogram`.

Symmetrize

Symmetrization of a dissimilarity matrix.

USAGE

```
Symmetrize(dist_matrix)
```

ARGUMENT

`dist_matrix` (`DISTANCE_MATRIX`).

RETURNED OBJECT

An object of type `DISTANCE_MATRIX` is returned.

SEE ALSO

`SelectIndividual`.

TimeEvents

Construction of data of type {time interval between two observation dates, number of events occurring between these two observation dates} from time sequences, from an object of type HISTOGRAM or from an ASCII file.

USAGE

```
TimeEvents(seq1, begin_date, end_date, PreviousDate->3,
           NextDate->8)
TimeEvents(seqn, variable, begin_date, end_date,
           PreviousDate->3, NextDate->8)

TimeEvents(histo, time)

TimeEvents(file_name)
```

ARGUMENTS

seq1 (SEQUENCES): univariate time sequences (with an explicit index parameter of type TIME),
seqn (SEQUENCES): multivariate time sequences (with an explicit index parameter of type TIME),
variable (INT): variable index,
begin_date (INT): initial observation date,
end_date (INT): final observation date,

histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA): number of events frequency distribution,
time (INT): time interval between two observation dates (length of the observation period),

file_name (STRING).

OPTIONAL ARGUMENTS

PreviousDate (INT): date preceding the initial observation date to check the increasing character of the number of events. This optional argument can only be used if the first mandatory argument is of type SEQUENCES.
NextDate (INT): date following the final observation date to check the increasing character of the number of events. This optional argument can only be used if the first mandatory argument is of type SEQUENCES.

RETURNED OBJECT

If the construction succeeds, an object of type TIME_EVENTS is returned, otherwise no object is returned.

SEE ALSO

Save, ExtractHistogram, Merge, NbEventSelect, TimeScaling, TimeSelect.

TimeScaling

Change of the time unit of data of type {time interval between two observation dates, number of events occurring between these two observation dates}.

USAGE

```
TimeScaling(timev, scaling_factor)
```

ARGUMENTS

`timev` (`TIME_EVENTS, RENEWAL_DATA`),
`scaling_factor` (`INT`): scaling factor.

RETURNED OBJECT

If `scaling_factor` > 0, an object of type `TIME_EVENTS` is returned, otherwise no object is returned.

SEE ALSO

`Merge`, `NbEventSelect`, `TimeSelect`.

TimeSelect

Selection of data item of type {time interval between two observation dates, number of events occurring between these two observation dates} according to a length of the observation period criterion.

USAGE

```
TimeSelect(timev, time)
TimeSelect(timev, min_time, max_time)
```

ARGUMENTS

`timev` (`TIME_EVENTS, RENEWAL_DATA`),
`time` (`INT`): time interval between two observation dates,
`min_time` (`INT`): minimum time interval between two observation dates,
`max_time` (`INT`): maximum time interval between two observation dates.

RETURNED OBJECT

If either `time` > 0 or if 0 < `min_time` ≤ `max_time` and if the range of lengths of the observation period defined either by `time` or by `min_time` and `max_time` enables to select data items of type {time interval between two observation dates, number of events}, an object of type `TIME_EVENTS` is returned, otherwise no object is returned.

SEE ALSO

`Merge`, `NbEventSelect`, `TimeScaling`.

ToDistanceMatrix

Cast an object of type CLUSTERS into an object of type DISTANCE_MATRIX.

USAGE

```
ToDistanceMatrix(clusters)
```

ARGUMENT

clusters (CLUSTERS).

RETURNED OBJECT

An object of type DISTANCE_MATRIX is returned.

SEE ALSO

Clustering.

ToDistribution

Cast an object of type HISTOGRAM into an object of type DISTRIBUTION.

USAGE

```
ToDistribution(histo)
```

ARGUMENT

histo (HISTOGRAM).

RETURNED OBJECT

If the object histo contains a 'model' part, an object of type DISTRIBUTION is returned, otherwise no object is returned.

SEE ALSO

ToHistogram.

ToHistogram

Cast an object of type DISTRIBUTION into an object of type HISTOGRAM.

USAGE

```
ToHistogram(dist)
```

ARGUMENT

dist (DISTRIBUTION).

RETURNED OBJECT

If the object dist contains a 'data' part, an object of type HISTOGRAM is returned, otherwise no object is returned.

SEE ALSO

ToDistribution.

TopParameters

Construction of 'top' parameters from the three parameters or from an ASCII file.

USAGE

```
TopParameters(proba, axillary_proba, rhythm_ratio,
              MaxPosition->40)

TopParameters(file_name, MaxPosition->40)
```

ARGUMENTS

`proba` (`INT`, `REAL`): growth probability of the parent shoot,
`axillary_proba` (`INT`, `REAL`): growth probability of the offspring shoots,
`rhythm_ratio` (`INT`, `REAL`): growth rhythm ratio offspring shoots / parent shoot,

`file_name` (`STRING`).

OPTIONAL ARGUMENT

`MaxPosition` (`INT`): maximum position for the computation of the distributions of the number of internodes of offspring shoots (default value: 20).

RETURNED OBJECT

If the construction succeeds, an object of type `TOP_PARAMETERS` is returned, otherwise no object is returned.

BACKGROUND

The aim of the model of 'tops' is to related the growth of offspring shoots to the growth of their parent shoot in the case of immediate branching. In the case where the arguments are the three 'top' parameters, the constraints over these parameters are described in the definition of the syntactic form of the type `TOP_PARAMETERS` (*cf.* **Part II4**).

SEE ALSO

`Save`, `Simulate` (4).

Tops

Construction of a set of 'tops' from a multidimensional array of integers or from an ASCII file.

USAGE

```
Tops(array, Identifiers->[1, 8, 12])

Tops(file_name)
```

ARGUMENTS

`array` (`ARRAY(ARRAY(ARRAY(INT)))`),

`file_name` (`STRING`).

OPTIONAL ARGUMENTS

`Identifiers` (`ARRAY(INT)`): identifiers of 'tops'. This optional argument can only be used if the first argument is of type `ARRAY(ARRAY(ARRAY(INT)))`.

RETURNED OBJECT

If the construction succeeds, an object of type `TOPS` is returned, otherwise no object is returned.

DESCRIPTION

The data structure of type `ARRAY(ARRAY(ARRAY(INT)))` should be constituted at the most internal level of arrays of size 2 (position of the offspring shoot on the parent shoot counted in number of nodes from the apex, number of internodes of the offspring shoot). Hence, the first component of successive elementary arrays (representing the index parameter) should be increasing. Note also that only the index parameter of the last array is considered.

SEE ALSO

`Save`, `Merge`, `RemoveApicalInternodes`, `Reverse`, `SelectIndividual`, `Estimate` (4).

Transcode

Transcoding of values.

USAGE

```
Transcode(histo, new_values)

Transcode(vec1, new_values)
Transcode(vecn, variable, new_values)

Transcode(seq1, new_values)
Transcode(seqn, variable, new_values)
Transcode(discrete_seq1, new_values, AddVariable->True)
Transcode(discrete_seqn, variable, new_values,
          AddVariable->True)
```

ARGUMENTS

histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),
new_values (ARRAY(INT)): new values replacing the old ones min, min + 1, …, max.

vec1 (VECTORS): values,
vecn (VECTORS): vectors,
variable (INT): variable index,

seq1 (SEQUENCES): univariate sequences,
seqn (SEQUENCES): multivariate sequences,
discrete_seq1 (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): discrete univariate sequences,
discrete_seqn (DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): discrete multivariate sequences.

OPTIONAL ARGUMENT

AddVariable (BOOL): addition (instead of simple replacement) of the variable to which the transcoding is applied (default value: False). This optional argument can only be used if the first argument is of type DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA.

RETURNED OBJECT

If the new values are in same number as the old values and are consecutive from 0, an object of type HISTOGRAM is returned (respectively VECTORS, SEQUENCES or DISCRETE_SEQUENCES), otherwise no object is returned. In the case of a first argument of type SEQUENCES, the returned object is of type DISCRETE_SEQUENCES if all the variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

BACKGROUND

The function `Cluster` with the mode `"Limit"` can be seen as a dedicated interface of the more general function `Transcode`.

SEE ALSO

`Cluster`, `Merge`, `Shift`, `ValueSelect`, `MergeVariable`, `SelectIndividual`, `SelectVariable`, `AddAbsorbingRun`, `Cumulate`, `Difference`, `IndexExtract`, `LengthSelect`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `VariableScaling`.

TransformPosition

Discretization of inter-position intervals.

USAGE

```
TransformPosition(seq, step)
```

ARGUMENTS

`seq` (`SEQUENCES`),
`step` (`INT`): step of discretization.

RETURNED OBJECT

If the first variable of `seq` is of type `POSITION`, and if $0 < \texttt{step} \leq$ (mean of inter-position intervals), an object of type `SEQUENCES` or `DISCRETE_SEQUENCES` is returned, otherwise no object is returned. The returned object is of type `DISCRETE_SEQUENCES` if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

`AddAbsorbingRun`, `Cluster`, `Cumulate`, `Difference`, `SelectIndividual`, `IndexExtract`, `LengthSelect`, `Merge`, `MergeVariable`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `VariableScaling`.

ValueSelect

Selection of individuals according to the values taken by a variable.

USAGE

```
ValueSelect(histo, value, Mode->"Reject")
ValueSelect(histo, min_value, max_value Mode->"Reject")

ValueSelect(vec1, value, Mode->"Reject")
ValueSelect(vec1, min_value, max_value, Mode->"Reject")
ValueSelect(vecn, variable, value, Mode->"Reject")
ValueSelect(vecn, variable, min_value, max_value,
            Mode->"Reject")

ValueSelect(seq1, value, Mode->"Reject")
ValueSelect(seq1, min_value, max_value, Mode->"Reject")
ValueSelect(seqn, variable, value, Mode->"Reject")
ValueSelect(seqn, variable, min_value, max_value,
            Mode->"Reject")
```

ARGUMENTS

histo (HISTOGRAM, MIXTURE_DATA, CONVOLUTION_DATA, COMPOUND_DATA),

value (INT): value,
min_value (INT): minimum value,
max_value (INT): maximum value,

vec1 (VECTORS): values,
vecn (VECTORS): vectors,
variable (INT): variable index,

seq1 (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): univariate sequences,
seqn (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): multivariate sequences.

OPTIONAL ARGUMENT

Mode (STRING): conservation or rejection of selected individuals: "Keep" (the default) or "Reject".

RETURNED OBJECT

If either value ≥ 0 or if $0 \leq$ min_value $\leq$ max_value and if the range of values defined either by value or by min_value and max_value enables to select individuals, an object of type HISTOGRAM is returned (respectively VECTORS, SEQUENCES or DISCRETE_SEQUENCES), otherwise no object is returned. In the case of a first argument of type SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA or SEMI-MARKOV_DATA, the returned object is of type DISCRETE_SEQUENCES if all the

variables are of type STATE, if the possible values for each variable are consecutive from 0 and if the number of possible values for each variable is ≤ 15.

SEE ALSO

`Cluster`, `Merge`, `Shift`, `Transcode`, `SelectIndividual`, `MergeVariable`, `SelectVariable`, `AddAbsorbingRun`, `Cumulate`, `Difference`, `IndexExtract`, `LengthSelect`, `MovingAverage`, `RecurrenceTimeSequences`, `RemoveRun`, `Reverse`, `SegmentationExtract`, `VariableScaling`.

VariableScaling

Change of the unit of a variable.

USAGE

```
VariableScaling(seq1, scaling_factor)
VariableScaling(seqn, variable, scaling_factor)
```

ARGUMENTS

seq1 (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): univariate sequences,
seqn (SEQUENCES, DISCRETE_SEQUENCES, MARKOV_DATA, SEMI-MARKOV_DATA): multivariate sequences.
variable (INT): variable index,
scaling_factor (INT): scaling factor.

RETURNED OBJECT

If scaling_factor > 0, an object of type SEQUENCES is returned, otherwise no object is returned.

BACKGROUND

The function VariableScaling is mainly useful as a preprocessing when one wants to study the correlation structure of residual sequences. This function enables by an appropriate scaling to control the rounding of the residual sequences and hence to obtain exact sample correlation functions.

SEE ALSO

AddAbsorbingRun, Cluster, Cumulate, Difference, IndexExtract, LengthSelect, Merge, MergeVariable, MovingAverage, RecurrenceTimeSequences, RemoveRun, Reverse, SegmentationExtract, SelectIndividual, SelectVariable, Shift, Transcode, ValueSelect.

VarianceAnalysis

One-way variance analysis.

USAGE

```
VarianceAnalysis(vec, class_variable, response_variable, type,
                 FileName->"result", Format->"SpreadSheet")
```

ARGUMENTS

vec (VECTORS),
class_variable (INT): index of the class or group variable,
response_variable (INT): index of the response variable,
type (STRING): type of the response variable ("NUMERIC" ("N") or "ORDINAL" ("O")).

OPTIONAL ARGUMENTS

FileName (STRING): name of the result file,
Format (STRING): format of the result file: "ASCII" (default format) or "SpreadSheet". This optional argument can only be used in conjunction with the optional argument FileName.

RETURNED OBJECT

No object returned.

DESCRIPTION

The result of the variance analysis is displayed in the shell window.

VectorDistance

Construction of an object of type VECTOR_DISTANCE from types (and eventually weights) of variables or from an ASCII file.

USAGE

```
VectorDistance(type1, type2,…, Distance->"QUADRATIC")
VectorDistance(weight1, type1, weight2, type2,…,
               Distance->"QUADRATIC")

VectorDistance(file_name)
```

ARGUMENTS

type1, type2, … (STRING): variable types ("NUMERIC" ("N"), "ORDINAL" ("O") or "SYMBOLIC" ("S")),
weight1, weight2, … (REAL): weights of variables,

file_name (STRING).

OPTIONAL ARGUMENT

Distance (STRING): distance type: "ABSOLUTE_VALUE" (the default) or "QUADRATIC". This optional argument is only relevant in the multivariate case.

RETURNED OBJECT

If the construction succeeds, an object of type VECTOR_DISTANCE is returned, otherwise no object is returned.

BACKGROUND

The type VECTOR_DISTANCE implements standardization procedures. The objective of standardization is to avoid the dependence on the variable type (chosen among symbolic, ordinal, numeric and circular) and, for numeric variables, on the choice of the measurement units by converting the original variables to unitless variables.

SEE ALSO

Compare (2)(3).

Vectors

Construction of a set of vectors from a multidimensional array, from a set of sequences or from an ASCII file.

USAGE

```
Vectors(array, Identifiers->[1, 8, 12])

Vectors(seq, IndexVariable->True)

Vectors(file_name)
```

ARGUMENTS

`array` (`ARRAY(ARRAY(INT))`),

`seq` (`SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA`, `SEMI-MARKOV_DATA`),

`file_name` (`STRING`).

OPTIONAL ARGUMENTS

`Identifiers` (`ARRAY(INT)`): explicit identifiers of vectors. This optional argument can only be used if the first mandatory argument is of type `ARRAY(ARRAY(INT))`.

`IndexVariable` (`BOOL`): taking into account of the implicit index parameter as a supplementary variable (default value: `False`). This optional argument can only be used if the first mandatory argument is of type `SEQUENCES`, `DISCRETE_SEQUENCES`, `MARKOV_DATA` or `SEMI-MARKOV_DATA`.

RETURNED OBJECT

If the construction succeeds, an object of type `VECTORS` is returned, otherwise no object is returned.

DESCRIPTION

The data structure of type `ARRAY(ARRAY(INT))` should be constituted at the most internal level of arrays of constant size.

SEE ALSO

`Save`, `ExtractHistogram`, `Cluster`, `Merge`, `MergeVariable`, `SelectIndividual`, `SelectVariable`, `Shift`, `Transcode`, `ValueSelect`, `Compare` (2), `ComputeRankCorrelation`, `ContingencyTable`, `Regression`, `VarianceAnalysis`.

4 THE GEOM MODULE

The following chapter lists all GEOM and APP objects in alphabetical order.

Each object description begins with a general explanation. Then each field composing the object are listed within a table, as well as its name, its type, its default values (if any) and a brief comment explaining its role.

AmapSymbol

CLASS

Geometry - Primitive - Surface/Volume - Mesh

DESCRIPTION

The AmapSymbol describes an object of class of *Mesh* stored in the SMB file format of the Amap software. This is provided for ascendant compatibility.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
FileName	STRING	None	specifies the name of the SMB file to bind to the symbol. It must contain the full path and .smb extension if needed. The corresponding file must exist.
Solid	BOOLEAN	False	specifies whether the symbol represents a closed surface.

EXAMPLE

```
# An example of AmapSymbol Object
AmapSymbol enpomf {
   FileName "/usr/local/AMAPmod/databases/SMBFiles/enpomf.smb"
}
```

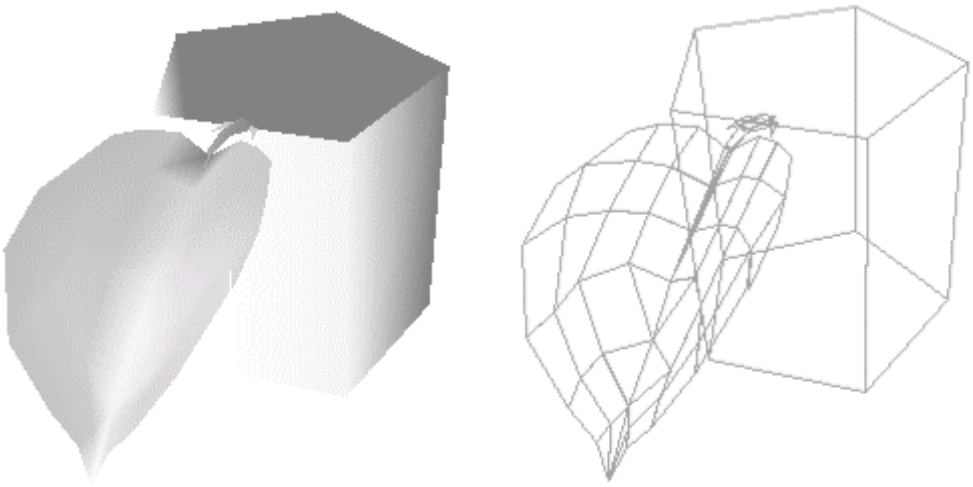


Figure: The AmapSymbol object.

AsymmetricHull

CLASS

Geometry - Primitive - Volume - Hull

DESCRIPTION

An `Asymmetric Hull` describes an object of class of *Mesh* defined by 6 morphological points (see figure **1**). This is an implementation of the asymmetric crowns introduced by Cescatti in [#!ces97!#] and Koop in [#!koo89!#]. The two first morphological points are the bottom and top points of the hull. The four other points are used to defined the peripheral line of the hull (P_1, P_2, P_3, P_4 on figure **2**). The two first points are located along the *x*-axis (P_1, P_2) and the two other along the *y*-axis (P_3, P_4). Finally, the shape coefficients are versatile index which describe the curvature of the hull above and below the peripheral line.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
PosXRadius	REAL	0.5	The Positive X Radius. *y*-coordinate of the first morphological point (P_1).
PosXHeight	REAL	0	The Positive X Height. z-coordinate of the first morphological point P_1).
NegXRadius	REAL	0.5	The Negative X Radius. *y*-coordinate of the second morphological point (P_2).
NegXHeight	REAL	0	The Negative X Height. *z*-coordinate of the second morphological point (P_2).
PosYRadius	REAL	0.5	The Positive Y Radius. *x*-coordinate of the third morphological point (P_3).
PosYHeight	REAL	0	The Positive Y Height. *z*-coordinate of the third morphological point (P_3).
NegYRadius	REAL	0.5	The Negative Y Radius. *x*-coordinate of the fourth morphological point (P_4).
NegYHeight	REAL	0	The Negative Y Height. z-coordinate of the fourth morphological point (P_4).
Bottom	VECTOR3	<0,0,-0.5>	The bottom point of the hull.
Top	VECTOR3	<0,0,0.5>	The top point of the hull.
BottomShape	REAL	2	The bottom shape factor.
TopShape	REAL	2	The top shape factor.
Slices	INTEGER	4	specifies the number of subdivisions around the *z*-axis when discretizing the hull. It must be strictly positive.
Stacks	INTEGER	4	specifies the number of subdivisions along the *z*-axis when discretizing the hull. It must be strictly positive.

EXAMPLE

```
# A cyprus crown represented by an asymmetric hull
AsymmetricHull cyprus {
   PosXRadius 2
   PosYRadius 2
   NegXRadius 2
   NegYRadius 2.5
   PosXHeight 1
   PosYHeight 1
   NegXHeight 1.5
   NegYHeight 1.2
   Top <0,0.5,8>
   Bottom <0.5,0,0>
   TopShape 1.5
   BottomShape 2
   Slices 5
   Stacks 10
}
```

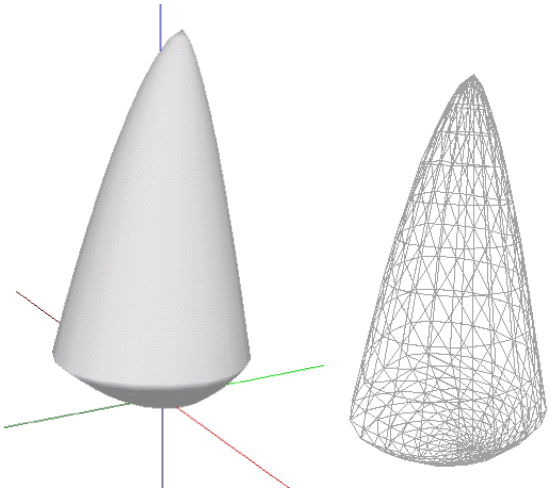


Figure 1: The `Asymmetric hull` object.

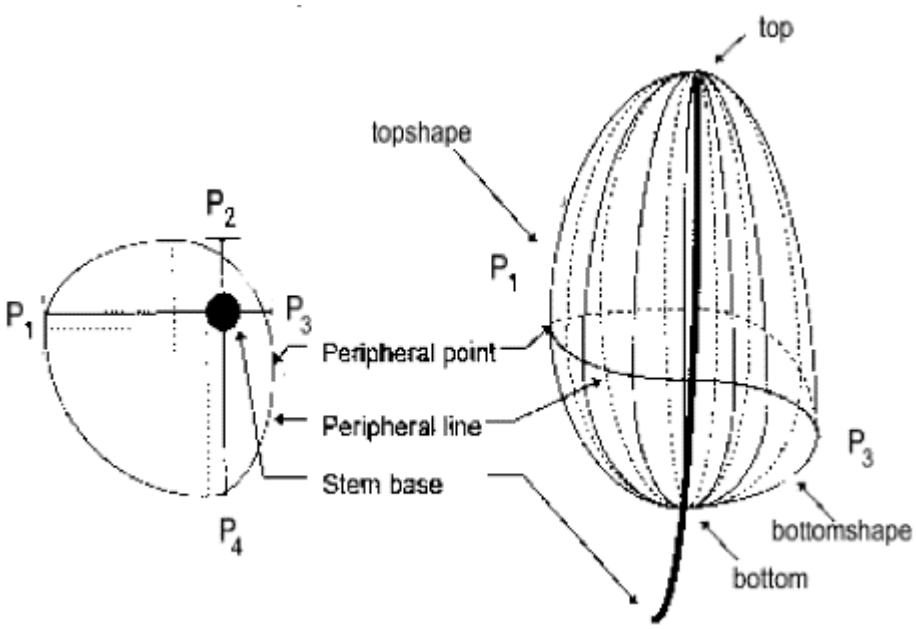


Figure 2: Geometry and Parameters of the `AsymmetricHull`

AxisRotated

CLASS

Geometry - Transformed

DESCRIPTION

The AxisRotated describes an object to which it has been applied a rotation of a specifief angle about a specifed axis. The rotation is given by the matrix:

$$M = \begin{pmatrix} (1-c)x^2 & txy - sz & (1-c)xy + sy \\ (1-c)xy + sz & (1-c)y^2 + c & (1-c)yz + sx \\ (1-c)xz + sy & (1-c)yz = sx & (1-c)z^2 + c \end{pmatrix},$$

where $s = \sin(angle), c = \cos(angle), x$ the x coordinate of *axis*, y the y coordinate and z the z coordinate.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
Axis	VECTOR3	<0,0,1>	specifies the axis of the rotation.
Angle	REAL	0	specifies the angle of the rotation. It must be in degrees.
Geometry	GEOMETRY	None	specifies the GEOM object which is affected by the rotation.

EXAMPLE

```
# A AxisRotated Cylinder of 45 degrees about the x-axis
Cylinder cylinder {
   Height 8
   Radius 1
   Slices 16
}
AxisRotated axisrotated {
   Axis <1,0,0>
   Angle 45
  Geometry cylinder
}
```

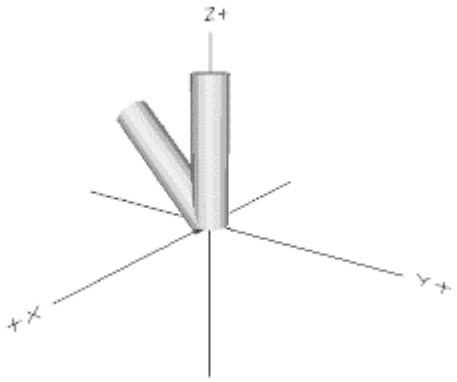


Figure: The axis rotated object.

BezierCurve

CLASS

Geometry - Primitive - Curve

DESCRIPTION

The `BezierCurve` describes rational and non rational Bezier curves defined from the parametric equation

$$C(u) = \sum_{i=0}^{n} B_{i,n}(u) P_i \qquad 0 \le u \le 1$$

where n is called the degree of the curve, the $B_{i,n}(u)$ the classical n-th degree Bernstein polynomials and the geometric coefficients the control points. For more information on this object, you could read [#!pie97!#].

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Degree`	`INTEGER`	`Computed`	specifies the degree of the curve. If not defined, This field is computed from the number of control points - 1. In the other case, a verification of the value's coherence is made.
`CtrlPointList`	`VECTORd[ ]`	`None`	specifies the control points of the curve. It could be points in 3 or 4 dimensions. With 4D control points, the curve is a Rational Bezier Curve.
`Stride`	`INTEGER`	`30`	specifies the number of point to computed when discretizing the curve.

EXAMPLE

```
# A bezier curve of degree 4
BezierCurve a_beziercurve {
   CtrlPointList[<0,0,0>,<4,-4,4>,<8,8,8>,<-12,12,12>,<-16,-
       16,16>]
}
```

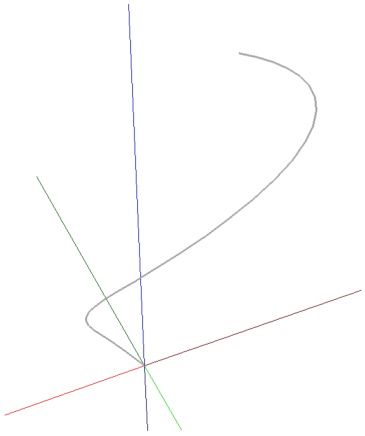


Figure: The `bezier curve` object.

BezierCurve2D

CLASS

Geometry - Primitive - Planar Model - Planar Curve

DESCRIPTION

The `BezierCurve2D` describes 2D rational and non rational Bezier curves.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Degree`	`INTEGER`	`Computed`	specifies the degree of the curve. If not defined, This field is computed from the number of control points - 1. In the other case, a verification of the value's coherence is made.
`CtrlPointList`	`VECTORd[ ]`	`None`	specifies the control points of the curve. It could be points in 2 or 3 dimensions. With 3D control points, the curve is a Rational Bezier Curve 2D.
`Stride`	`INTEGER`	`30`	specifies the number of point to computed when discretizing the curve.

EXAMPLE

```
# A bezier curve of degree 3
BezierCurve2D a_beziercurve {
   CtrlPointList [ <-4,-4>, <-2,4>, <2,-4>, <4,4> ]
}
```

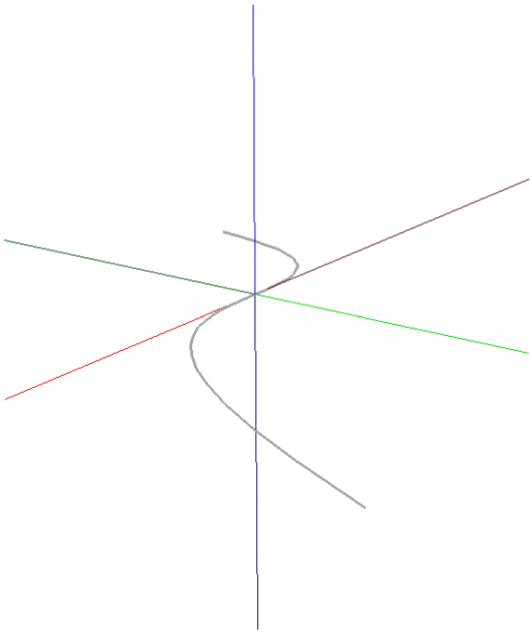


Figure: The `bezier curve2D` object.

BezierPatch

CLASS

Geometry - Primitive - Surface - Patch

DESCRIPTION

The `BezierPatch` describes rational and non rational Bezier surface defined from the parametric equation

$$s(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,p}(u)B_{j,q}(v)P_{i,j}$$

which is a bivariate equation where n and m are called the degrees of the surface, the $B_{i,p}(u)B_{j,q}(v)$ is the product of classical univariate p-th and q-th degrees Bernstein polynomials and the geometric coefficients $P_{i,j}$ a bidirectional net of control points. For more information on this object, you could read [#!pie97!#].

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`UDegree`	`INTEGER`	`Computed`	specifies the degree of the surface. If not defined, This field is computed from the number of control points in the rows of the net -1. In the other case, a verification of the value's coherence is made.
`VDegree`	`INTEGER`	`Computed`	specifies the degree of the surface. If not defined, This field is computed from the number of control points in the columns of the net - 1. In the other case, a verification of the value's coherence is made.
`CtrlPointMatrix`	`VECTORd[][]`	`None`	specifies the control points net of the surface. It could be points in 3 or 4 dimensions. With 4D control points, the surface is a Rational Bezier Surface.
`UStride`	`INTEGER`	`30`	specifies the number of points in the first direction to compute when discretizing the surface.
`VStride`	`INTEGER`	`30`	specifies the number of points in the second direction to compute when discretizing the surface.

EXAMPLE

```
(#
   A bezier patch
```

```
#)

BezierPatch a_bezierpatch {
   CtrlPointMatrix [
      [ <5,0,2>, <5,2,3>, <5,4,0> ],
      [ <1,0,3>, <1,2,3>, <1,4,2> ],
      [ <-1,0,3>, <-1,2,3>, <-1,4,1> ],
      [ <-5,0,3>, <-5,2,4>, <-5,4,2> ]
   ]
}
```

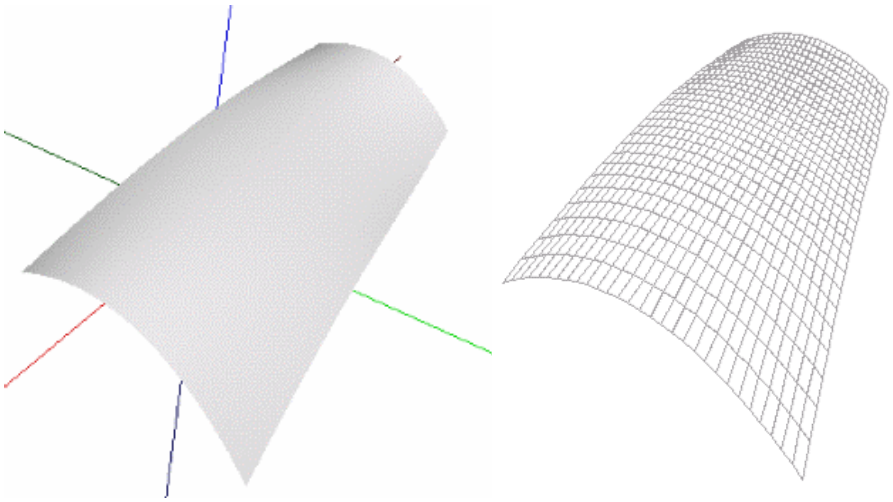


Figure: The `bezier surface` object.

Box

CLASS

Geometry - Primitive - Volume - Mesh

DESCRIPTION

The `Box` describes a rectangular axis-aligned box centered at (0,0,0) and whose extension along the x, y and z-axis is specified with a 3D vector.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Size`	`VECTOR3`	`<.5,.5,.5>`	specifies the x, y and z extents of the box's width, depth and height. Each coordinates must be positive.

EXAMPLE

```
# A Box
Box box {
   Size <2,1,6>
}
```

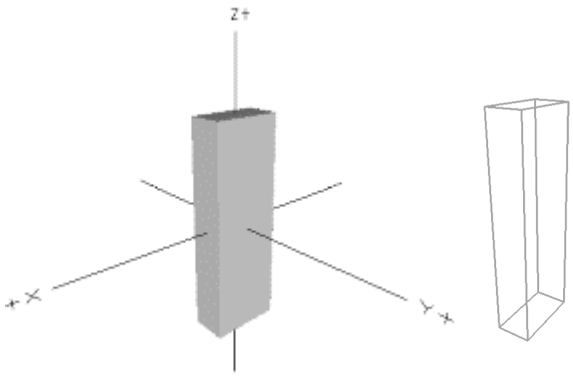


Figure: The box object.

Cone

CLASS

Geometry - Primitive - Surface/Volume - SOR

DESCRIPTION

The **Cone** describes a cone whose base lies into the *x*-*y* plane and central axis is the *z*-axis.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
Radius	REAL	0.5	specifies the radius of the cone. It must be strictly positive.
Height	REAL	1	specifies the height of the cone. It must be strictly positive.
Solid	BOOLEAN	True	specifies whether the bottom side is visible.
Slices	INTEGER	8	specifies the number of subdivisions around the *z*-axis when discretizing the cone.

EXAMPLE

```
# A Cone
Cone cone {
   Radius 8
   Height 6
   Solid True
   Slices 32
}
```

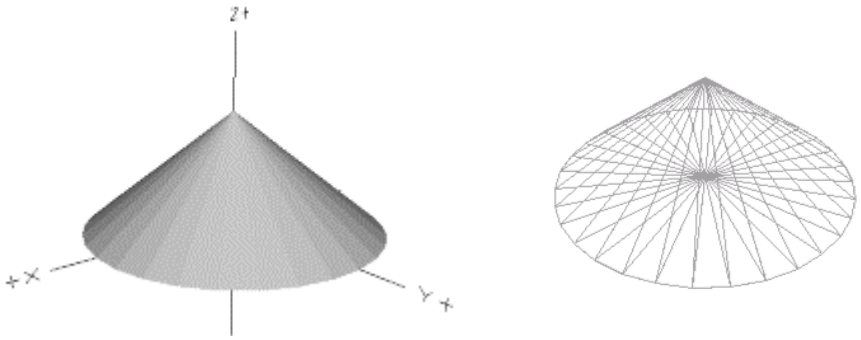


Figure: The cone object.

Cylinder

CLASS

Geometry - Primitive - Surface/Volume - SOR

DESCRIPTION

The `Cylinder` describes a cylinder whose base lies into the *x*-*y* plane and central axis is the *z*-axis.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Radius`	`REAL`	`0.5`	specifies the radius of the cylinder. It must be strictly positive.
`Height`	`REAL`	`1`	specifies the height of the cylinder. It must be strictly positive.
`Solid`	`BOOLEAN`	`True`	specifies whether the bottom and top sides are visible.
`Slices`	`INTEGER`	`8`	specifies the number of subdivisions around the *z*-axis when discretizing the cylinder.

EXAMPLE

```
# A Cylinder
Cylinder cylinder {
   Radius 5
   Height 8
   Solid True
   Slices 64
}
```

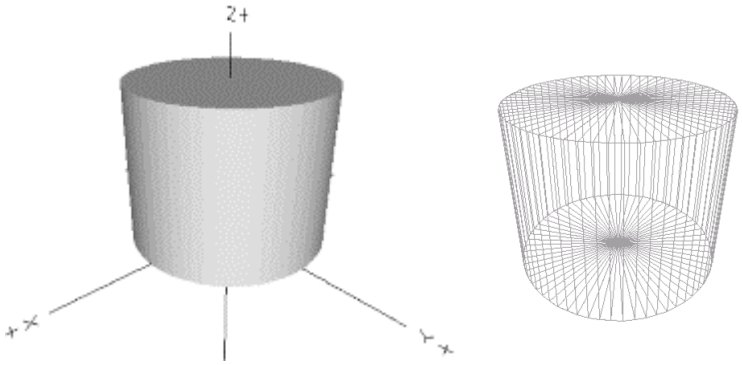


Figure: The `cylinder` object.

Dis**C**

CLASS

Geometry - Primitive - Planar Model - Planar Surface

DESCRIPTION

The `Disc` represents a disc centered at (0,0,0) and lying into the *x*-*y* plane.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Radius`	`REAL`	`0.5`	specifies the radius of the disc. It must be strictly positive.
`Slices`	`INTEGER`	`8`	specifies the number of subdivisions around the *z*-axis when discretizing the disc.

EXAMPLE

```
# A Disc
Disc disc {
      Radius 4
      Slices 16
}
```

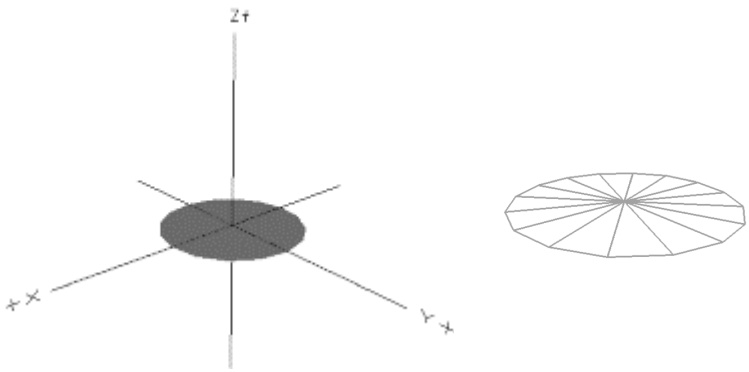


Figure: The `disc` object.

ElevationGrid

CLASS

Geometry - Primitive - Surface - Patch

DESCRIPTION

The `ElevationGrid` describes a regular grid of a specified number of rows and columns and the elevation on each points of that grid. Heights are described row-major order (along the *x*-axis first), left to right and top to bottom.
It is mainly used for terrain modelling.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`HeightMatrix`	`REAL[ ][ ]`	`None`	specify the elevation associated to each points of the grid. Each arrays must contain exactly the same number of values and must contain at leat two values.
`XSpacing`	`REAL`	`1`	specifies the distance between two consecutive points in the *x*-direction. It must be strictly positive.
`YSpacing`	`REAL`	`1`	specifies the distance between two consecutive points in the y-direction. It must be strictly positive.

EXAMPLE

```
# A digital terrain represented with an ElevationGrid.
ElevationGrid elevation_grid {
   HeightMatrix [
      [0.0, 1.5, 2.0, 1.5],
      [0.5, 1.8, 2.2, 1.7],
      [0.8, 2.2, 2.8, 2.2],
      [1.2, 2.6, 3.2, 2.8],
      [1.0, 2.4, 2.5, 2.1],
      [0.8, 1.8, 2.0, 1.6]
   ]
   XSpacing 5
   YSpacing 5
}
```

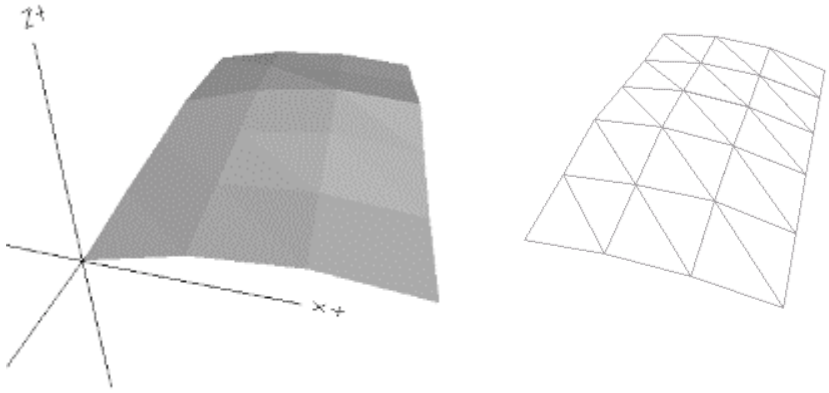


Figure: The `elevation grid` object.

EulerRotated

CLASS

Geometry - Transformed

DESCRIPTION

The `EulerRotated` describes an object to which it has been applied a composition of rotations by *azimuth* about the *z*-axis, by *elevation* about the rotated *y*-axis and by *roll* about the rotated *x*-axis. The equivalent rotation is given by the matrix:

$$M = \begin{pmatrix} ca*ce & ca*ce*sr - sa*cr & ca*se*cr + sa*sr \\ sa*ce & ca*cr + sa*se*sr & sa*se*cr - ca*sr \\ -se & ce*sr & ce*sr \end{pmatrix},$$

where $cr = \cos(roll)$, $sr = \sin(roll)$, $ce = \cos(elevation)$, $se = \sin(elevation)$, $ca = \cos(azimuth)$ and $sa = \sin(azimuth)$.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Azimuth`	`REAL`	`0`	specifies the angle of the rotation about the *z*-axis.
`Elevation`	`REAL`	`0`	specifies the angle of rotation about the rotated y-axis.
`Roll`	`REAL`	`0`	specifies the angle of rotation about the rotated *x*-axis.
`Geometry`	`GEOMETRY`	`None`	specifies the GEOM object which is affected by the rotation.

EXAMPLE

```
# A EulerRotated Box
Box box {
   Size <2,0.2,4>
}
EulerRotated axisrotated {
   Azimuth 45
   Elevation 0
   Roll 60
   Geometry box
}
```

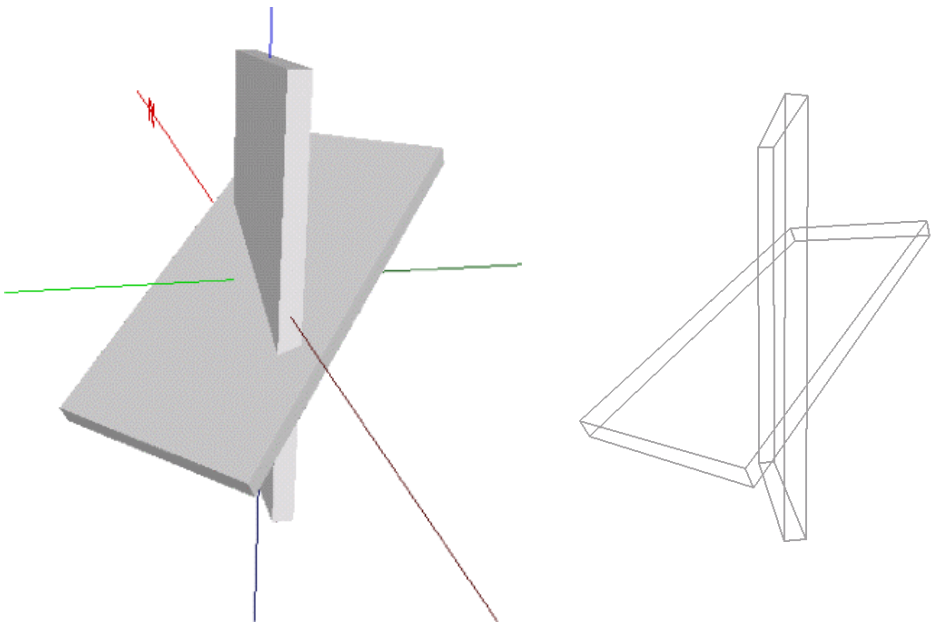


Figure: The `euler rotated` object.

ExtrudedHull

CLASS

Geometry - Primitive - Volume - Hull

DESCRIPTION

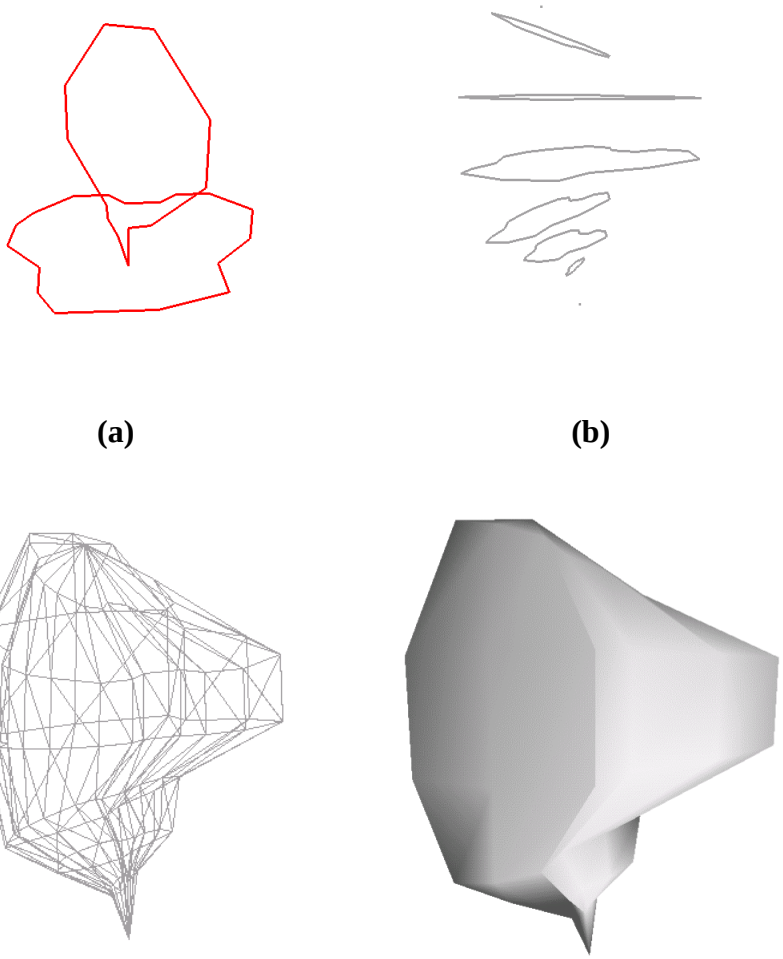
The `Extruded Hull` describes an object of class of *Hull* extruded by a vertical and an horizontal profiles. For more information on this model, see Birnbaum Thesis [#!bir97!#].

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Vertical`	`PLANAR CURVE`	`None`	specifies the vertical profile of the hull. There must be at least one point.
`Horizontal`	`PLANAR CURVE`	`None`	specifies the horizontal profile of the hull. There must be at least one point.
`CCW`	`BOOLEAN`	`True`	Indicates whether each polygon's points are listed in counterclockwise (`True`) or clockwise (`False`) order when viewed from the front.

EXAMPLE

```
ExtrudedHull hull {
   Vertical Polyline2D {
      PointList [ <0,-3.5>, <-1,-3>, <-3,-1>,
         <-3,1>, <-1,3.2>, <1,3.2>,
         <3,1>, <3,-1>, <1,-3>,
         <1,-3.5>, <0.5,-4>, <0,-5> ]
   }
   Horizontal Polyline2D {
      PointList [ <-3.8,0>, <-4,-1>, <-2,-3>,
         <0,-4>, <0,-4>, <1,-3>,
         <3,-3>, <5,-2>, <5,0>,
         <4,1>, <3,1>, <2,2>,
         <2,3>, <1,4>, <-1,4>,
         <-2,3.75>, <-3,2.8>, <-3,1> ]
   }
}
```



(a) **(b)**

(c) **(d)**

Figure: The `extruded hull` object. (a) The two profiles, (b) Construction of the different point set using the two profiles, (c) the resulting mesh and (d) the extruded hull

Extrusion

CLASS

Geometry - Surface/Volume

DESCRIPTION

The `Extrusion` differs from straight Cylinder in that a space curve may act as the central axis and the cross section may vary in size or shape. Because of this variability, the Extrusion can represent a wide variety of forms. This model is one type of *swept surface*. For more information on this model, see Bloomenthal Thesis [#!bloo95!#].

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`CrossSection`	`CURVE2D`	`None`	The Cross Section of the Extrusion.
`Axis`	`CURVE`	`None`	The Axis of the Extrusion.
`Scale`	`VECTOR2[]`	`[<1,1>]`	The scaling factors for 2D scaling transformation. The cross Section is scaled using thoses factors
`Orientation`	`REAL[]`	`[0]`	The orientation angles for 2D orientation transformation.The cross Section is oriented using thoses angles
`KnotList`	`REAL[]`	`[0,1]`	We consider the scaling function to be linear beetween two knots. Must have the same size than Scale
`CCW`	`BOOLEAN`	`True`	Indicates whether cross section's points are listed in counterclockwise (`True`) or clockwise (`False`) order when viewed from the front.
`Solid`	`BOOLEAN`	`False`	specifies whether the bottom and top sides are visible.

EXAMPLE

```
# Definition of the Cross Section.
BezierCurve2D _crossSection {
   CtrlPointList [ <-2,0>, <-2,-2>, <2,-2>,
                   <2,2>, <-2,2>, <-2,0> ]
   Stride 10
}
# Definition of the axis
BezierCurve _axis {
   CtrlPointList [ <0,0,0>, <3,3,3>, <3,3,6>,  <6,6,6> ]
   Stride 10
}
# The Extrusion
Extrusion an_extrusion {
   Axis _axis
   CrossSection _crossSection
   Scale [ <1,1>,<0.8,0.8>,<0.1,0.1> ]
```

```
    KnotList [ 0, 0.2,1 ]
}
```

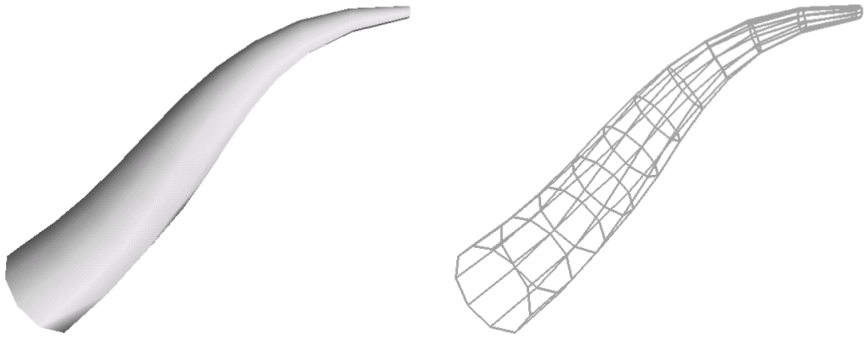


Figure: The `extrusion` object.

FaceSet

CLASS

Geometry - Primitive - Surface/Volume - Mesh

DESCRIPTION

A `FaceSet` describes a surface formed by connected faces. Polygons are specified using indices into a list of vertices located at the specified coordinates.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`PointList`	`VECTOR3[]`	`None`	specifies the coordinates of the constructing points.
`IndexList`	`INTEGER[][]`	`None`	specifies the faces, each specified as a serie of indices into the list of points. Be sure that the indices are not out of range and each serie must contain at least 3 elements.
`CCW`	`BOOLEAN`	`True`	Indicates whether each polygon's points are listed in counterclockwise (`True`) or clockwise (`False`) order when viewed from the front.
`Solid`	`BOOLEAN`	`False`	specifies whether this mesh represent a closed surface.
`Skeleton`	`GEOMETRY`	`Null`	specifies the skeleton of this mesh. It must be a GEOM object of type of Polyline.

EXAMPLE

```
FaceSet my_faceset {
   PointList [ <2.5,0,0>, <2.5,0,4.5>, <1.25,2.16506,0>,
     <1.25,2.16506,4.5>, <-1.25,2.16506,0>,
     <-1.25,2.16506,4.5>,<-2.5,-2.18557e-07,0>,
     <-2.5,-2.18557e-07,4.5>, <-1.25,-2.16506,0>,
     <-1.25,-2.16506,4.5>, <1.25,-2.16506,0>,
     <1.25,-2.16506,4.5>,<0,0,-1.5>, <0,0,6> ]
   IndexList [ [0,2,3,1], [1,3,13], [0,12,2],
     [2,4,5,3], [3,5,13], [2,12,4],
     [4,6,7,5], [5,7,13], [4,12,6],
     [6,8,9,7], [7,9,13], [6,12,8],
     [8,10,11,9], [9,11,13], [8,12,10],
     [10,0,1,11], [11,1,13], [10,12,0]
   ]
   Solid True
   Skeleton Polyline {
   PointList [ <0,0,0>, <0,0,4.5> ]
   }
}
```

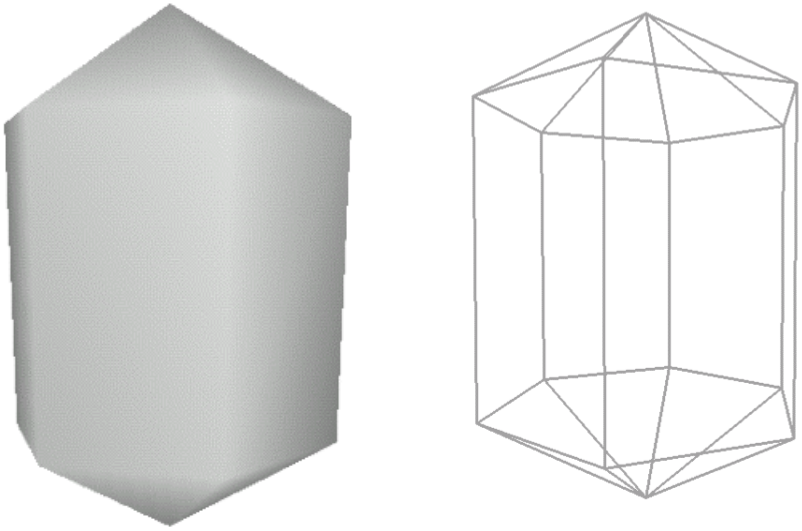


Figure: The `face set` object.

Frustum

CLASS

DESCRIPTION

e *x-y* plane and central axis is the *z*-axis. The ratio between the top radius and the base radius is defined by a

the top radius and the base radius.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Radius`	Real	`0.5`	specifies the radius of the frustum. It must be strictly positive.
`Height`	Real	`1`	specifies the height of the frustum It must be strictly positive.
`Taper`	Real	`0`	specifies the rate of taper. It must be positive.
`Solid`	Boolean	`True`	specifies whether the bottom and top sides are visible.
`Slices`	Integer	`8`	specifies the number of subdivisions around the *z*-axis when discretizing the frustum.

EXAMPLE

CLASS

DESCRIPTION

complex object.

FIELDS DESCRIPTION

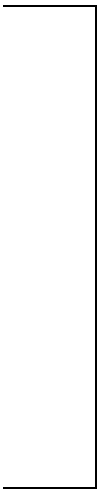
Name	Type	Defaults	Description
`GeometryList`	Geometry[ ]	None	specifies the list of Geoms which are to be grouped.

`Skeleton`	Geometry	`Null`	specifies the skeleton of this group. It must be a GEOM object of type of Polyline.

EXAMPLE

ing a

e.



CLASS

DESCRIPTION

ject determine the way light reflect off an object to create color.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
Ambient	Color3	<160,160,160>	specifies how much ambient light the surface must reflect. Depending on the ambient light, this field define the color of the object.
Diffuse	Real	1.0	specifies the diffuse color, wich reflects all light sources

			depending on the angle of the surface with respect to the light source. The more directly the surface faces the light, the more diffuse light is reflected. The diffuse color is computed as *Ambient * Diffuse* and must be a valid Color*3*.
Specular	Color3	<0,0,0>	specifie the color of an object 's highlights.
Emission	Color3	<0,0,0>	specifies the light produced by a glowing object. Emissive color is usefull for displaying radiosity-based models (where the light energy of the scene is computed explicitly).
Shininess	Real	0	degree of shininess of an object, ranging from 0.0 for a diffuse surface with no shininess to 1.0 for a hightly polished surface.
Transparency	Real	0	degree of transparency of an object, ranging from 0.0 for a completely opaque surface to 1.0 for a completely clear surface.

EXAMPLE

CLASS

DESCRIPTION

FIELDS DESCRIPTION

Name	Type	Defaults	Description
Reflectance	Real	0.8	Must be in [0,1].
Transmittance	Real	0.0	Must be in [0,1].

ance			

EXAMPLE

CLASS

DESCRIPTION

FIELDS DESCRIPTION

Name	Type	Defaults	Description
Reflectance	Real[]	None	Each value must be in [0,1].
Transmittance	Real[]	None	Each value must be in [0,1].
Filter	Index3	[1,1,1]	

EXAMPLE

CLASS

DESCRIPTION

rametric equation

unctions define on a clamped knot vector  and the geometric

oie97!#].

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Degree`	Integer	3 or Computed	specifies the degree of the curve. If not specified and if the KnotList isn't defined too, the degree value is assign to 3. If KnotList specified, the degree value is assign to the number of knots - the number of control points -1.
`CtrlPointList`	Vector *d*[ ]	None	specifies the control points of the curve. It could be points in 3 or 4 dimensions. With 4D control points, the curve is a Rational B-Spline Curve.
`KnotList`	Real[ ]	Computed	specifies the knot value of the curve. If not defined, This field take the value of an adapted clamped uniform vector and the curve is a Uniform B-Spline Curve. The KnotList must be clamped and increasing. If specified and if the degree is specified, the number of knot must be equal to the number of control points + degree +1.
`Stride`	Integer	50	specifies the number of point to compute when discretizing the curve.

EXAMPLE

CLASS

DESCRIPTION

FIELDS DESCRIPTION

Name	Type	Defaults	Description

`Degree`	Integer	3 or Computed	specifies the degree of the curve. If not specified and if the KnotList isn't defined too, the degree value is assign to 3. If KnotList specified, the degree value is assign to the number of knots - the number of control points -1.
`CtrlPoint List`	Vector *d*[ ]	None	specifies the control points of the curve. It could be points in 2 or 3 dimensions. With 3D control points, the curve is a 2D Rational B-Spline Curve.
`KnotList`	Real[ ]	Computed	specifies the knot value of the curve. If not defined, This field take the value of an adapted clamped uniform vector and the curve is a Uniform B-Spline Curve. The KnotList must be clamped and increasing. If specified and if the degree is specified, the number of knot must be equal to the number of control points + degree +1.
`Stride`	Integer	50	specifies the number of point to compute when discretizing the curve.

EXAMPLE

CLASS

DESCRIPTION

: equation

is the product of univariate -th and -th degrees rationnal basis functions, defined on

and the geometric coefficients a bidirectional net of control points. For more information on this object,

FIELDS DESCRIPTION

Name	Type	Defaults	Description
UDegree	Integer	3 or Computed	specifies the degree of the surface in the first direction. If not specified and if the UKnotList isn't defined too, the udegree value is assign to 3. If UKnotList specified, the udegree value is assign to the number of knots of UKnotList - the number of control points in a row - 1.
VDegree	Integer	3 or Computed	specifies the degree of the surface in the second direction. If not specified and if the VKnotList isn't defined too, the vdegree value is assign to 3. If VKnotList specified, the vdegree value is assign to the number of knots of VKnotList - the number of control points in a column - 1.
CtrlPoint Matrix	Vector *d*[ ]	None	specifies the control points net of the curve. It could be points in 3 or 4 dimensions. With 4D control points, the curve is a Rational B-Spline Surface.
UKnotList	Real[ ]	Computed	specifies the knot value of the surface in the first direction. If not defined, This field take the value of an adapted clamped uniform vector. The UKnotList must be clamped and increasing. If specified and if the udegree is specified, the number of knot must be equal to the number of control points in a row + udegree +1.
VKnotList	Real[ ]	Computed	specifies the knot value of the surface. If not defined, This field

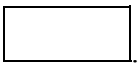
		puted	take the value of an adapted clamped uniform vector. The VKnotList must be clamped and increasing. If specified and if the vdegree is specified, the number of knot must be equal to the number of control points in a column + vdegree +1.
`UStride`	Integer	50	specifies the number of point to compute when discretizing the curve in the first direction.
`VStride`	Integer	50	specifies the number of point to compute when discretizing the curve in the second direction.

EXAMPLE

CLASS

DESCRIPTION

ıonormal basis. The basis is expressed by the matrix:

ternary direction, which is given by:  .

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Primary`	Vector 3	`<1,0,0>`	specifies the primary direction. The vector will be automatically normalized if needed.
`Secondary`	Vector 3	`<0,1,0>`	specifies the secondary direction. It must be orthogonal to the primary direction, that is, the dot product between those 2 vectors must be 0.

			The vector will be automatically normalized if needed.
`Geometry`	Geometry	None	specifies the GEOM object wich is affected by the change of basis.

EXAMPLE

CLASS

DESCRIPTION

where □denotes the height, □the radius and □the shape factor.

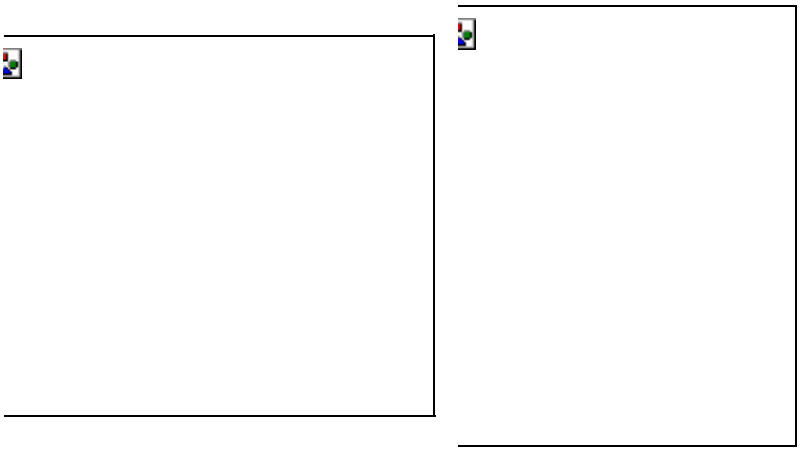
FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Radius`	Real	`0.5`	specifies the radius of the paraboloid. It must be strictly positive.
`Height`	Real	`1`	specifies the height of the paraboloid. It must be strictly positive.
`ShapeFactor`	Real	`0`	specifies the concavity. It must be strictly positive.
`Solid`	Boolean	`True`	specifies whether the bottom and top sides are visible.
`Slices`	Integer	`8`	specifies the number of subdivisions around the *z*-axis when discretizing the paraboloid.
`Stacks`	Integer	`8`	specifies the number of subdivisions along the *z*-axis when discretizing the paraboloid.

EXAMPLE

```
# A neiloid represented by a Paraboloid
Paraboloid paraboloid {
Radius 5
Height 8
ShapeFactor 0.6
Solid True
Slices 25
Stacks 25
}
```

Figure: The `paraboloid` object.



3.26 PointSet

CLASS

Geometry - Primitive

DESCRIPTION

The **PointSet** describes a set of points, each located at the specified coordinates.

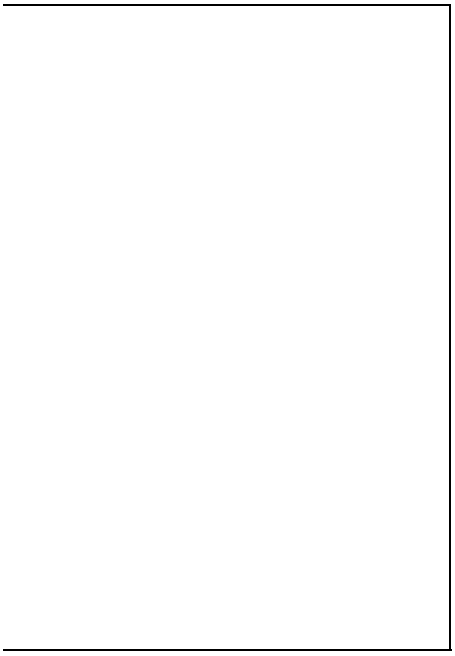
FIELDS DESCRIPTION

Name	Type	Defaults	Description
`PointList`	Vector3[ ]	None	specifies the coordinates of the constructing points of this point set. There must be at least one point.

EXAMPLE

```
# Corners of a cube represented by a PointSet
PointSet point_set {
PointList [ <5,-5,-5>, <-5,-5,-5>, <-5,5,-5>, <5,5,-5>,
<5,-5,5>, <-5,-5,5>, <-5,5,5>, <5,5,5>
]
}
```

Figure: The `point set` object.



3.27 Polyline

CLASS

Geometry - Primitive - Curve

DESCRIPTION

A **Polyline** describes a curve formed by connected segment located at specified coordinates.

FIELDS DESCRIPTION

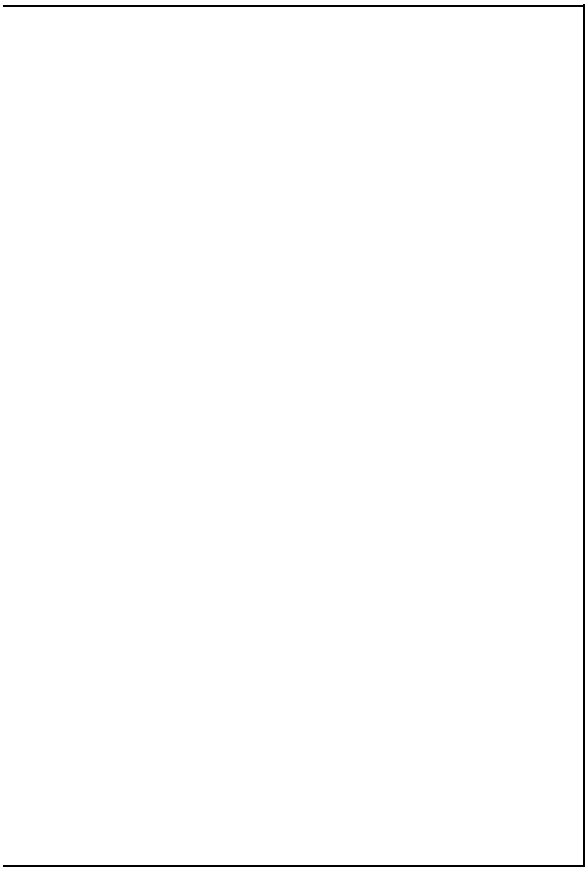
Name	Type	Defaults	Description
`PointList`	Vector3[ ]	None	specifies the coordinates of the constructing points of this polyline. There must be at least two points.

EXAMPLE

```
# A spiral represented by a Polyline
```

```
Polyline polyline {
PointList [ <-4,0,0>, <-3,3,0.5>, <0,4,1>, <3,3,1.5>,
        <4,0,2>, <3,-3,2.5>, <0,-4,3>, <-3,-3,3.5>,
        <-4,0,4>, <-3,3,4.5>, <0,4,5>, <3,3,5.5>,
        <4,0,6>, <3,-3,6.5>, <0,-4,7>, <-3,-3,7.5> ]
}
```

Figure: The `polyline` object.



3.28 Polyline2D

CLASS

Geometry - Primitive - Planar Model - Planar Curve

DESCRIPTION

A **Polyline2D** describes a planar curve formed by connected segment located at specified 2D coordinates.

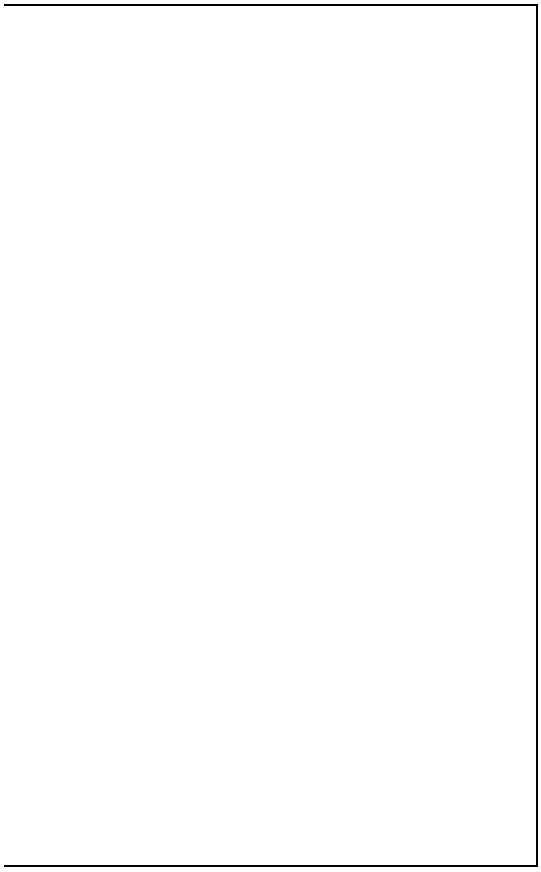
FIELDS DESCRIPTION

Name	Type	Defaults	Description
`PointList`	Vector2[ ]	None	specifies the coordinates of the constructing points of this polyline. There must be at least two points.

EXAMPLE

```
# A spiral represented by a Polyline2D
Polyline2D polyline2D {
PointList [  <0,0>,   <1,0>,  <1,1>, <-2,1>,
        <-2,-2>, <3,-2>, <3,3>, <-4,3>,
        <-4,-4>, <5,-4>, <5,5>, <-6,5>,
        <-6,-6>, <7,-6>, <7,7> ]
}
```

Figure: The `polyline2D` object.



3.29 QuadSet

CLASS

Geometry - Primitive - Surface/Volume - Mesh

DESCRIPTION

A **QuadSet** describes a surface formed by quadrilaterals, four sided polygons. Quads are specified using indices into a list of vertices located at the specified coordinates.

FIELDS DESCRIPTION

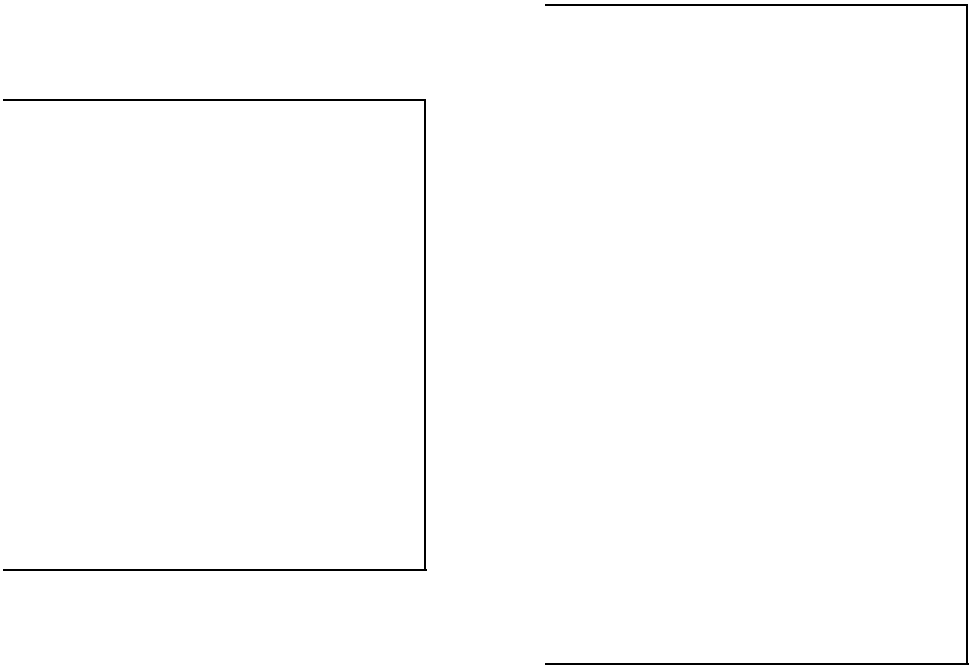
Name	Type	Defaults	Description
`PointList`	Vector3[ ]	None	specifies the coordinates of the constructing points.

`IndexList`	Integer[ ][ ]	None	specifies the faces, each specified as a serie of indices into the list of points. Be sure that the indices are not out of range and each serie must contain at exactly four elements.
`CCW`	Boolean	`True`	Indicates whether each quadrilateral's points are listed in counterclockwise (`True`) or clockwise (`False`) order when viewed from the front.
`Solid`	Boolean	`False`	specifies whether this mesh represent a closed surface.
`Skeleton`	Geometry	`Null`	specifies the skeleton of this mesh. It must be a GEOM object of type of Polyline.

EXAMPLE

```
# A sheared cube represented by a QuadSet
QuadSet quad_set {
PointList [ <2,-5,-5>, <-5,-5,-5>, <-5,5,-5>, <2,5,-5>,
        <5,-5,5>, <-2,-5,5>, <-2,5,5>, <5,5,5>
]
IndexList [ [0,1,2,3], [0,3,7,4], [1,0,4,5],
        [2,1,5,6], [3,2,6,7], [4,7,6,5]
]
CCW True
Solid True
}
```

ure: The `quad set` object.



30 Revolution

CLASS

metry - Primitive - Surface/Volume - SOR

DESCRIPTION

Revolution primitive describes a general surface of revolution generated by the rotation of a planar curve ut the *z*-axis.

number of points within the generatrix curve determines the number of subdivisions along the *z*-axis when retizing the object.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`PointList`	Vector2[ ]	None	specifies the coordinates of the 2D curve to be rotated.

`Slices`	Integer	`8`	specifies the number of subdivisions around the *z*-axis when generating the solid.

EXAMPLE

A glass represented using a Revolution

Revolution revolution {

PointList [

<3,0>, <1,0.2>,

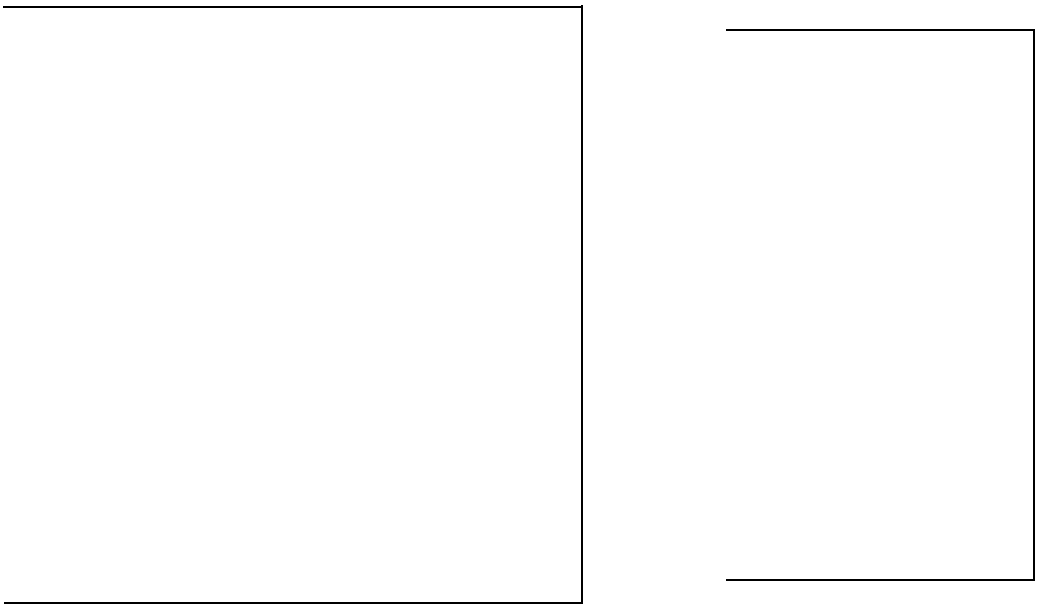
<0.7,0.3>, <0.5,0.6>,

<0.5,4>, <1,5>,

<3,10>, <4,11>

es 64 }

The `revolution` object



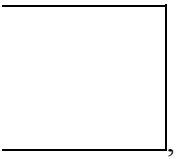
l Scaled

CLASS

- **Transformed**

DESCRIPTION

describes an object to which it has been applied an anisotropic scale. The scaling transformation is given by



,

denotes the scaling factors along the x, y and z-axis.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Scale`	Vector 3	`<1,1,1>`	specifies the scaling factors along the x-axis, the y-axis and the z-axis. Each of the coordinates must be different from 0.
`Geometry`	Geometry	None	specifies the GEOM object wich is affected by the scale.

EXAMPLE

led Cylinder along the x-axis and the z-axis

r cylinder {

5

2

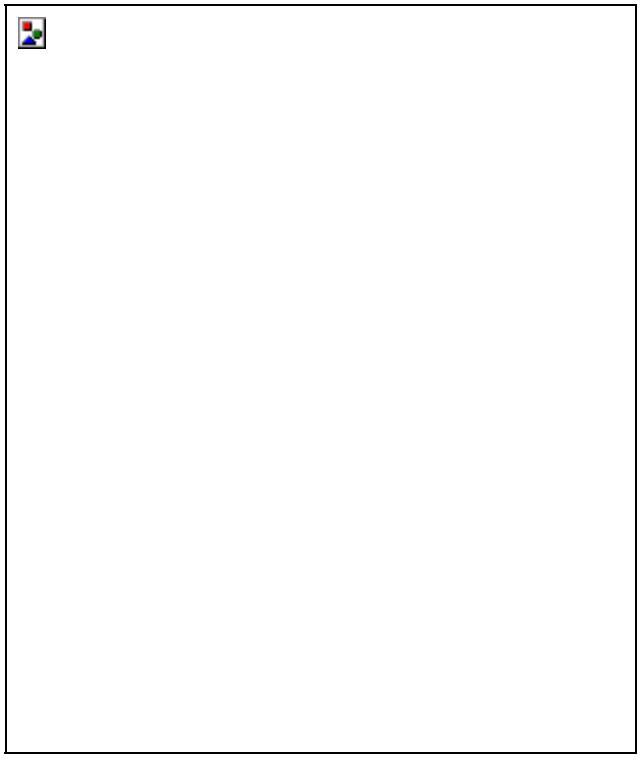
6

ed {

3,1,2>

ry cylinder

e `scaled` object.



ounded by the scaled cylinder.

Shape

CLASS

DESCRIPTION

ect associate a geometric object with an appearance.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Geometry`	Geometry	`None`	specifies the geometry of the shape.
`Appearance`	Appearance	`Default Material`	specifies the appearance of the shape.

EXAMPLE

rown

green

ere

CLASS

tive - Volume - SOR

DESCRIPTION

es a sphere of a specified radius and centered at  .

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Radius`	Real	`0.5`	specifies the radius of the sphere. It must be strictly positive.
`Slices`	Integer	`8`	specifies the number of subdivisions around the z-axis when discretizing

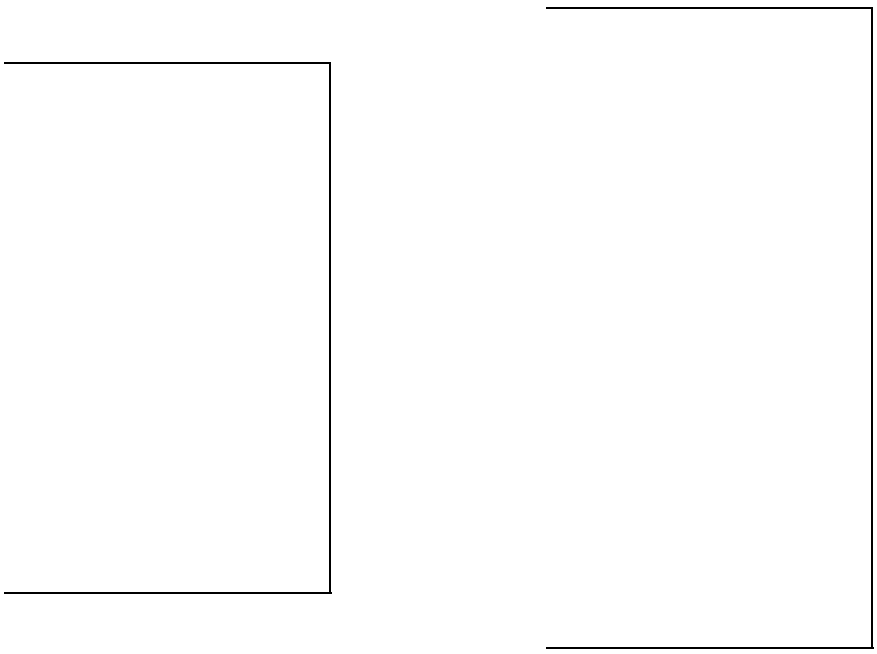
			the sphere. It must be strictly positive.
`Stacks`	Integer	`8`	specifies the number of subdivisions along the z-axis when discretizing the sphere. It must be strictly positive.

EXAMPLE

h a low level of discretization around the z-axis

{

e object.



ered

CLASS

med

DESCRIPTION

s an object to which it has been applied a Taper deformation. A Taper deforms an object in order to be able hin a cone frustum of a specifed base radius and top radius.

sing an object, a Taper scale the polar coordinates according the z-coordinate. The amplitude of the scale is

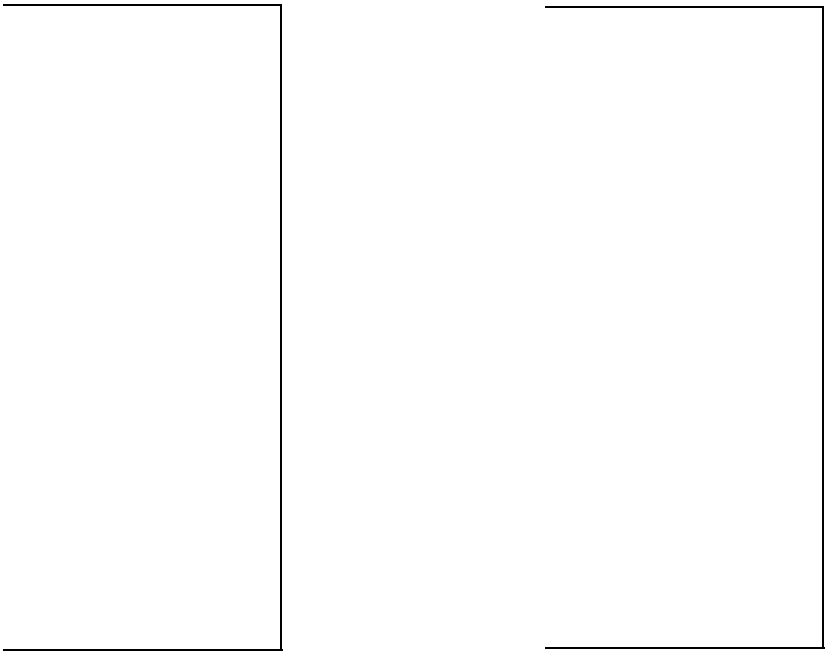
FIELDS DESCRIPTION

Name	Type	Defaults	Description
`BaseRadius`	Real	`1`	specifies the base radius of the cone frustum.
`TopRadius`	Real	`1`	specifies the top radius of the cone frustum.
`Primitive`	Geometry	None	specifies the GEOM object wich is affected by the taper. It must be a GEOM object of type of Primitive.

EXAMPLE

sented by a Tapered Box

ct.



ted

CLASS

DESCRIPTION

bject to which it has been applied a translation of a specified vector. The translation is given by the

ranslation vector.

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`Translation`	Vector3	`<0,0,0>`	specifies translation vector.
`Geometry`	Geometry	None	specifies the GEOM object wich is affected by the translation.

EXAMPLE

ong the y-axis

ct

et

CLASS

Volume - Mesh

DESCRIPTION

formed by triangles, three sided polygons. Triangles are specified using indices into a list of vertices located

FIELDS DESCRIPTION

Name	Type	Defaults	Description
`PointList`	Vector 3[ ]	None	specifies the coordinates of the constructing points.
`IndexList`	Integer[ ][ ]	None	specifies the faces, each specified as a serie of indices into the list of points. Be sure that the indices are not out of range and each serie must contain exactly three elements.
`CCW`	Boolean	`True`	Indicates whether each triangle's points are listed in counterclockwise (`True`) or clockwise (`False`) order

			when viewed from the front.
`Solid`	Boolean	`False`	specifies whether this mesh represent a closed surface.
`Skeleton`	Geometry	`Null`	specifies the skeleton of this mesh. It must be a GEOM object of type of Polyline.

EXAMPLE

by a TriangleSet

5>,<-5,0,-8.5>, <5,0,-8.5>,

<0,-8.5,5>, <0,-8.5,-5>,

<8.5,-5,0>, <-8.5,-5,0>

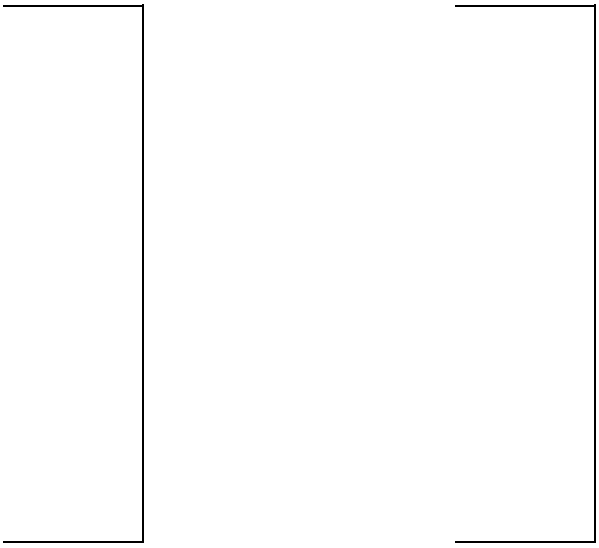
4,5], [4,8,5], [4,1,8],

3], [5,3,2], [2,3,7],

11], [11,6,0], [0,6,1],

11], [9,5,2], [7,11,2]

.



5 FILE SYNTAX

5.1 General conventions

In general, words can be separated by any combination of whitespace characters (SPACE, TAB, EOL). In certain files, TABs or EOL are meaningful (*e.g.* MTG coding files), and therefore are not considered as a whitespace character in these files.

Comments may be introduced anywhere in a file using the sharp sign #, meaning that the rest of the line is a comment. In some files, block comments can be introduced by bracketing the comment text with (# and #).

Files used by AML can be located anywhere in the UNIX hierachical file system, provided the user can access them. All references to files from within a file or from AML must be given explicitly. References to files must always be made relatively to the location where the reference is made.

In various files, user-defined names must be given to objects, attributes, etc. Unless specified otherwise, names always consist of strings of alphanumeric characters (including underscore '_') starting by a non-numeric character. A name may start by an underscore. Some names correspond to reserved keywords. Since reserved keywords always start in AMAPmod with uppercase letter, it is advised, though not mandatory, to define user-defined names starting with lowercase letter to avoid name collision.

5.2 MTGs

♦ Coding strategy

A plant multiscale topology is represented by a string of characters (see sec3.2.2). The string is made up of a series of labels representing plant components (a label is made up of an alphabetic character in A-Z,a-z and a numeric index) and of symbols representing either the physical relationships between the components. Character '/' is used for decomposition relationship (see next paragraph), '+' is used for branching relationship and '<' for successor relationship. For example:

```
/I1<I2<I3<I4+I5<I6
```

is a string representing 6 components with labels I1, I2, I3, I4, I5, I6. I1 to I4 are a sequence of components defining an axis which bears a second axis made up of the sequence of components I5 and I6. In this string every component is connected with at most one subsequent component (either by a '<' or by a '+')

As illustrated by this example, the name of an entity is built by concatenating the consecutive entity labels encountered while moving along the plant structure from the plant basis to the considered entity. For example, consider the decomposition of a plant in terms of axes. Assume this plant is made of 3 axes: axis A1 bears axis A2, which itself bears axis A3. Then, the respective names of the axes are:

```
/A1
/A1+A2
/A1+A2+A3
```

Symbol '+' refers to the type of connection between A1 and A2, A2 and A3 respectively. Now, consider another plant considered at the scale of growth units. A growth unit U90 bears a growth unit U91 which is itself followed on the same axis by U92. The respective names of these growth units are :

```
/U90
/U90+U91
/U90+U91<U92
```

These two examples illustrate how to define the name of plant entities when only one scale of description is considered. When several scales are considered, this strategy can be extended as explained in section 3.2.2.

Assume for instance that axis A1 of the previous example is composed of 3 consecutive growth units and that axis A2 is borne by the second growth units of A1. Then the name of A2 is defined as :

```
/A1/U1<U2+A2
```

♦ Relative names

Every name of an entity is thus the concatenation of a series of pairs (relation symbol,label) :
name = relation label relation label relation label relation...label relation label

Let us consider any prefix *p* of a name *n* of an entity *x* of the plant, made of a series of pairs (*relation label*). According to the recursive construction of entity names, this prefix defines the name of an entity *y* on the path from the plant basis to the entity with name *n*. The name of *x* has thus the form :

n = p m

where *m* is a series *label relation …label relation.* Entity *x* has absolute name *n*. Alternatively we can say that *x* has relative name *m* with respect position *p*, i.e. relatively to entity *y.*

Examples :

`/S1/A1/E1+A3` has relative name `/A1/E1+A3` in position `/S1`

`/S1/A1/E3+A1/E4+S1/U2/E3+U1/E5+U4/E4` has relative name `+U1/E5+U4/E4` in position `/S1/A1/E3+A1/E4+S1/U2/E3`

5.2.1 Coding files

The coding of a plant (or of a set of plants) is carried out in a so called “coding file”. The code consists of a description of the MTG representing plant architectures. A coding file contains two parts:

- a header which contains a description of the coding parameters,
- the code of the plant architecture.

The header contains general informations related to all individuals:

- the set of all entity classes used in the MTG description,
- a detailed description of the topological properties of these classes,
- and the set of all attributes used for any entity in the plant description.

In a MTG coding file, TABs are meaningful. They correspond to column separators. Consequently, a MTG coding file should be edited using a spreadsheet editor. If a sharp ‘#’ is inserted on a line, every character until the next TAB on the same line is considered as a comment and is not interpreted.

- **Header**

General parameter section

For historical reasons, two forms of plant architecture coding have been developed, denoted `FORM-A` et `FORM-B`. `FORM-A` is the most general and should be employed. `FORM-B` is available for ascendant compatibility with former coding forms employed in the AMAP laboratory [43]. Whatever the coding form used the plant built by AMAPmod is the same. The form of the coding language must be specified in the coding file by specifying either `FORM-A` or `FORM-B` following the keyword `CODE`, in the next column, for example :

```
CODE:       FORM-A
```

This definition is mandatory.

Class definition section

Classes must then be declared. This is done in a section beginning with keyword `CLASSES`. Then a line is defined for each class of the MTG. The first column, entitled `SYMBOL`, contains the symbolic character denoting a class used in the MTG. This symbol most be an alphabetic character (either upper or lower-case letter). Two classes either at identical or different scales must have different symbolic characters.

The second column, entitled `SCALE`, represents the scale at which this class appears in the MTG. There are no *a priori* limitation related to the number of classes, however, these must be consecutive integer greater or equal to 0. Scale *i*, *i*>1, can only appear if scale *i*-1 has appeared before.

```
CLASSES
SYMBOL    SCALE   DECOMPOSITION    INDEXATION    DEFINITION
$         0       FREE             FREE          IMPLICIT
P         1       CONNECTED        FREE          IMPLICIT
U         2       <-LINEAR         FREE          EXPLICIT
I         2       <-LINEAR         FREE          EXPLICIT
E         3       NONE             FREE          IMPLICIT
```

Symbol `$` represent the entire database and is defined by definition at scale 0.

Keyword `DECOMPOSITION` defines the types of decomposition that can have a vertex (i.e. a plant constituent) : `CONNECTED`, `LINEAR`, `<-LINEAR`, `+-LINEAR`, `FREE`, `NONE`. Key word `CONNECTED` means that the decomposition graph of a vertex at the next scale is connected. Keyword `LINEAR` means that the decomposition graph of a vertex at the next scale is a linear sequence of vertices. Besides, if this all the constituents of this sequence are connected using a single type of edge (respectively `<` or `+`), then keyword `<-LINEAR` et `+-LINEAR` can respectively be used. Keyword `FREE` allows any type of decomposition structure while keyword `NONE`, specifies that the components of a unit must not be decomposed.

Column `INDEXATION` is not used.

Column `DEFINITION` must be filled with value `EXPLICIT` if any entity of that class has feature values (i.e. attributes). `IMPLICIT` should be used otherwise.

This section is mandatory.

Topological constraints section

Topological constraints are described in the next section, beginning with keyword `DESCRIPTION`. Here, each line defines for a pair of classes at the same scale one allowed type of connection. It contains 4 columns, `LEFT`, `RIGHT`, `RELTYPE`, and `MAX`. For any class in column `LEFT`, the column `RIGHT` defines a list of class (appearing at the same scale) which can be connected to it using a connection of type `RELTYPE`. The maximum number of connections of type `RELTYPE` that can be made on an entity from column is defined in column `MAX`. If column `MAX` contains a question mark '?', the number of connections is not bounded.

If a class does not appear in the column LEFT, then entities of this class cannot be connected to other entities in the MTG.

```
DESCRIPTION:
LEFT    RIGHT    RELTYPE    MAX
U       U,I      +          ?
U       U,I      <          1
I       I        +          ?
E       E        <          1
E       E        +          1
```

Let us resume on the example from the above CLASS section with its DESCRIPTION section. Since class P does not appear in the left column, a P cannot be connected to any other entity at scale 1, e.g. to any other P. Entities of type U can be connected to entities of either type I or U, for any of the connection types < et +. An entity of type U can be connected by relation + to any number of Us or Is. However, they can only be connected by relation < to at most one entity of either type U or I. Entities of type I cannot be connected by relation < to any type of entity, while they can be connected to other I's by relation +. At scale 3, any E can be connected to only one other E by either relation + or <.

This section is mandatory but can contain no topology description.

Attribute section

The third and last part of the header contains a list of names defining the features that can be attached to plant entities and their types. This part begins with keyword FEATURES. Thelist of names appears in column NAME and the corresponding types in column TYPE. The name of an attribute might be either a reserved keyword (see a list below) or a user-defined name. The types of attributes can be INT (integer), REAL (real number), STRING (string of characters from {A…Za…z-+. /} and which are bounded to 14 characters max), DD/MM, DD/MM/YY, MM/YY, DD/MM-TIME, DD/MM/YY-TIME (Dates), GEOMETRY (geometric objects defined in a .geom file), APPEARANCE (appearance objects defined in a .app file), OBJECT (general object defined in generic type of file).

```
FEATURES:
NAME            TYPE

Alias           STRING
Date            DD/MM
NbEl            INT
State           STRING
flowerNb        INT
len             INT
TopDiameter     REAL
geom            GEOMETRY     geom1.geom
appear          APPEARANCE   material.app
```

Certain names of attributes are reserved keywords. They all start by an upper-case letter. If they appear in the feature list, they must be in the same order as in the following description. Alias, of type STRING (formerly ALPHA), must come first if used. It allows the user to define aliases for plant entities to simplify some code strings. Date, is used to define the observation date of an entity. NbEl (NumBer of ELements), defines the number of

components on any entity at the next scale. `Length` is the length of an entity. `BottomDiameter` et `TopDiameter` respectively define the bottom and top stretching values of a tapered transformed that is applied to the geometric symbol representing this entity (for branch segments associated with cylinder as a basic geometrc model, this defines cone frustums). `State` of type `STRING` defines the state of an entity at the time of observation. This state can be `D` (Dead), `A` (Alive), `B` (Broken) , `P` (Pruned), `G` (Growing), `V` (Vegetative), `R` (Resting), `C` (Completed), `M` (Modified). These letters can be combined to form a string of characters, provided they consistent with one another. Such state descriptions are checked during the parsing of the MTG and possible inconsistencies are detected.

This section is mandatory but can contain no features.

- **Coding section**

The section containing the code of a MTG starts by keyword `MTG`.

The next line contains a list of column names. In the first column, the keyword `TOPO` indicates that this column and the next unlabelled column are reserved for the topological code. On the same line, all the names that appear in the `FEATURE` section of the header must appear, in the same order, one column after the other, starting with the first feature name in a column sufficiently far from the `TOPO` column to leave enough space for the topological code (see examples below).

The topological code must necessarily start by a '/' like in

```
/P1/A1 …
```

It can spread on all the columns before the first feature column.

Since entity names have a nested definition, a plant description can be made on a single line. However, if one wants to declare feature values attached to some entity, the plant code must be interrupted after the label of this entity, attributes must be entered on the same line in corresponding columns, and the plant code must continue at the next line.

Note that in the current implementation of the parser, an entity which has no features uses obviously 0 bytes of memory for recording features, however, assuming that the total number of features is *F*, if an entity has at least one feature value defined, it uses a constant space *F**14 bytes to record its feature (whatever the actual number of features defined for this entity).

- **Example**

Here is an example of a coding file corresponding to plant illustrated on **Figure 5-1**:

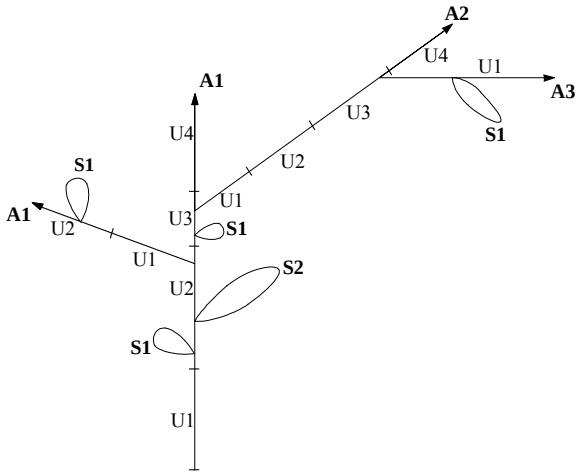


Figure 5-1 Example of a MTG code

```
CODE:    FORM-A

CLASSES:
SYMBOL        SCALE      DECOMPOSITION   INDEXATION  DEFINITION
$             0          FREE            FREE        IMPLICIT
P             1          CONNECTED       FREE        IMPLICIT
A             2          <-LINEAR        FREE        EXPLICIT
S             2          CONNECTED       FREE        EXPLICIT
U             3          NONE            FREE        IMPLICIT

DESCRIPTION:
LEFT    RIGHT      RELTYPE    MAX
A       A,S        +          ?
U       U          <          1
U       U          +          ?


FEATURES:
NAME     TYPE

MTG:
TOPO

/P1/A1
/P1/A1/U1<U2+S1
/P1/A1/U1<U2+S2
/P1/A1/U1<U2+A1
/P1/A1/U1<U2+A1/U1<U2+S1
/P1/A1/U1<U2<U3+S1
/P1/A1/U1<U2<U3+A2
/P1/A1/U1<U2<U3+A2/U1<U2<U3+A3
/P1/A1/U1<U2<U3+A2/U1<U2<U3+A3/U1+S1
/P1/A1/U1<U2<U3+A2/U1<U2<U3<U4
/P1/A1/U1<U2<U3<U4
```

In this example, certain names use frequently the same prefix which can be long (this bit of code contains 225 characters). We are going to introduce successively different strategies in order to simplify this first coding scheme.

The first simplification consists of giving a name (alias) to an entity name which is used frequently in the name of others.

```
# before the header is identical to the previous one

FEATURES:
NAME         TYPE
Alias        ALPHA

MTG:
TOPO                          Alias

/P1/A1                        A1
(A1)/U1<U2+S1                 Branch1
(A1)/U1<U2+S2
(A1)/U1<U2+A1
(A1)/U1<U2+A1/U1<U2+S1
(A1)/U1<U2<U3+S1
(A1)/U1<U2<U3+A2              A2
(A2)/U1<U2<U3+A3
(A2)/U1<U2<U3+A3/U1+S1        Branch2
(A2)/U1<U2<U3<U4
/P1/A1/U1<U2<U3<U4
```

An alias can be associated with a given entity by defining its name in column `Alias`. This name can then be reused in the topological section by enclosing it between parentheses. If an alias is used as a prefix of an entity, the code of this entity must be given relatively to this alias. For entity `A2`, for instance, we can see that its name is `/U1<U2<U3+A2` relatively to position `A1` which is an alias for `/P1/A1`. The absolute name of `A2` is thus, `/P1/A1/U1<U2<U3+A2`. The code part of this file has now a size of 173 characters, *i.e.* 78% of the initial code.

The code of the MTG can be further simplified. We can avoid completely the repetition of bit of codes. Assume that entity *y* has a code of the form `XY` where `X` represents the code of some entity *x*. For example `X` is `/P1/A1` and `Y` is `/U1<U2<U3+A2` in the previous example. If `X` already appears in column of the topological section, then we may consider that if subsequently `Y` appears at a different line, but shifted to the right by one column, then `Y` is actually follows `X` which is thus its prefix. Then `Y` is a relative name with respect to position `X`. In our example, this leads to :

```
/P1/A1                  # code of x
/P1/A1/U1<U2<U3+A2      # code of y
```

which becomes :

```
#column1          #column2
/P1/A1                              # code de x
                  /U1<U2<U3+A2      # code de y
```

The fact that the code of *y* is shifted one column to the right, allows us to interpret `/U1<U2<U3+A2` as the continuation of `/P1/A1` leading to the absolute name `/P1/A1/U1<U2<U3+A2` which is actually the code of *y*.

By applying this new rule on the complete previous example we obtain the following code :

```
MTG:
TOPO
#column1     #column2       #column3       #column4     #column5
/P1/A1
              /U1<U2
                             +S1
                             +S2
                             +A1/U1<U2+S1
                             <U3
                                            +A2/U1<U2<U3
                                                          +A3/U1+S1
                                                          <U4

                                            <U4
```

Now the number of characters used in the code is now 63 and corresponds to 28% of the initial code. However, this compressed code raises two new problems. The first problem is that the number of columns necessary has greatly increased. The second is that it is difficult to recognise the structural organisation of the plant in the way the code displays it.

To address both problem, a new syntactic notation is introduced. Each time a relative code starts with character ^ in a given cell, the current relative code must be interpreted with respect to the position whose code is the latest code defined in the same column just above the current cell. Using the ^ notation:

```
MTG:
TOPO

/P1/A1
^/U1<U2
              +S1
              +S2
              +A1/U1<U2+S1
^<U3
              +A2/U1<U2<U3
                             +A3/U1+S1
              ^<U4

^<U4
```

Here the number of columns used is equal to the number of orders in the plant (i.e. 3), which bounds the total number of columns required and best reflects in the code the botanical structure of the plant. Entities of order *i* are defined in column *i* which greatly improves the code leagibility. Finaly, the number of characters used is 69, i.e. 31% of the initial extended code.

In some cases, a series of consecutive entities must be coded, which produces long lines of code just as this one:

```
A1/U87<U88<U89<U90<U91<U92<U93+A2
```

Such a line can be abbreviated by using the << sign :

```
A1/U87<<U93+A2
```

`U87<<U93` is a syntactic shorthand for `U87<U89<U90<U91<U92<U93`.

Symbol `++` is defined similarly: `U87++U93` is a shorthand for `U87+U89+U90+U91+U92+U93`.

Note that in such cases, the entities implicitly defined cannot have attributes: for instance, the code:

```
TOPO                    diam        flowers
/A1/U87<<U93            10.3        2
```

Means that an axis `A1` is made of a series of 7 growth units, labelled from `U87` to `U93` and that `U93` has a diameter of 10.3 and bears 2 flowers. In some cases, we want to express that the attributes are shared by all entities. This can be expressed as follows:

```
TOPO                    diam        flowers
/A1/U87<.<U93                       1
```

which means that every growth units from `U87` to `U93` has exactly 1 flower. Notation `+.+` is defined similarly.

Here follows the complete code of plant of **Figure 5-1**.

```
CODE:    FORM-A

CLASSES:
SYMBOL    SCALE       DECOMPOSITION    INDEXATION    DEFINITION
$         0           FREE             FREE          IMPLICIT
P         1           CONNECTED        FREE          IMPLICIT
A         2           <-LINEAR         FREE          EXPLICIT
S         2           <-LINEAR         FREE          EXPLICIT
U         3           NONE             FREE          IMPLICIT

DESCRIPTION:
LEFT    RIGHT     RELTYPE    MAX
A       A,S       +          ?
U       U         <          1
U       U         +          ?

FEATURES:
NAME          TYPE

MTG:
TOPO

/P1/A1
^/U1<U2
              +S1
              +S2
              +A1/U1<U2+S1
^<U3
```

```
                +A2/U1<<U3
                                +A3/U1+S1
                <U4
^<U4
```

5.2.2 Examples of coding strategies in different classical situations

- **Non linear growth units**

Until now we have only used linear growth units, i.e. entities whose decomposition in a linear set of entities. It is possible to define branching growth-units, which are not a part of an axis. The plant illustrated in **Figure 5-2**: illustrates such non-linear entities.

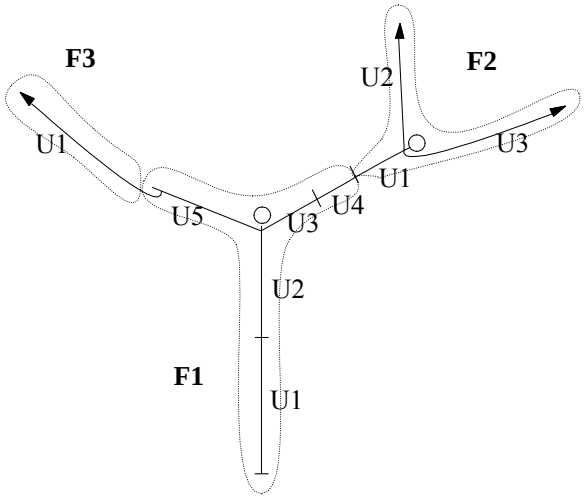


Figure 5-2 MTG containing non-linear growth units

```
CODE:     FORM-A

CLASSES:
SYMBOL        SCALE      DECOMPOSITION   INDEXATION  DEFINITION
$             0          FREE            FREE        IMPLICIT
F             1          CONNECTED       FREE        IMPLICIT
U             2          NONE            FREE        IMPLICIT

DESCRIPTION:
LEFT     RIGHT      RELTYPE     MAX
F        F          +           ?
F        F          <           1
U        U          +           ?
U        U          <           1

FEATURES:
NAME      TYPE

MTG:
TOPO
/F1/U1<U2
                +U3<U4<F2/U1
                                +U2
                                +U3
                +U5+F3/U1
```

- **Sympodial plants**

Sympodial plants often contain apparent axes made up of series of modules (or axes). At a macroscopic scale, the plant is described in terms of apparent axes connected to one another (**Figure 5-3**) depict a typical sympodial plant:

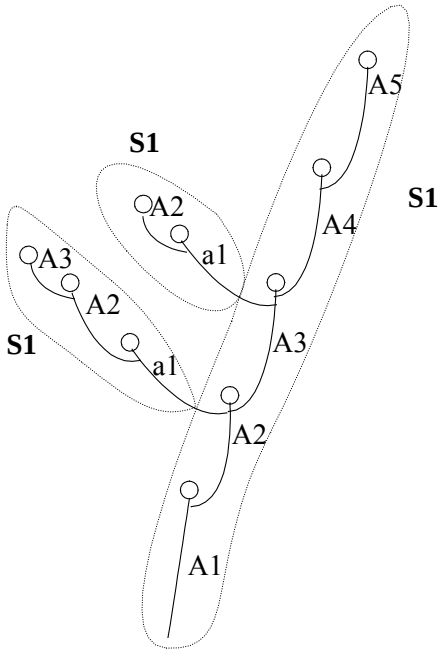


Figure 5-3 MTG containing sympodial axes.

```
CODE:    FORM-A

CLASSES:
SYMBOL        SCALE      DECOMPOSITION  INDEXATION  DEFINITION
$             0          FREE           FREE        IMPLICIT
S             1          +-LINEAR       FREE        IMPLICIT
A             2          <-LINEAR       FREE        IMPLICIT
A             2          <-LINEAR       FREE        IMPLICIT

DESCRIPTION:
LEFT    RIGHT      RELTYPE    MAX
S       S          +          ?
A       A,a        +          1
a       A          +          1

FEATURES:
NAME          TYPE

MTG:
TOPO
/S1
^/A1+A2
               +S1
               ^/a1+A2+A3
^+A3
               +S1
               ^/a1+A2
```

```
^+A4+A5
```

Note in this example the role of ^ which enables us to preserve the structure of the plant into the code itself. Indeed, apparent axes appear in columns corresponding to their apparent order.

- **Dominant axes**

Similarly, dominant axes in a plant can be identified using macroscopic units. **Figure 5-4** illustrates how to code dominant axes:

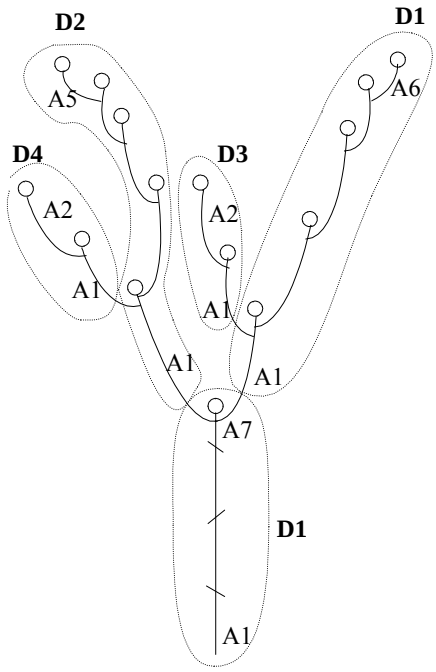


Figure 5-4 MTG dominant axes

```
CODE:    FORM-A

CLASSES:
SYMBOL        SCALE      DECOMPOSITION  INDEXATION  DEFINITION
$             0          FREE           FREE        IMPLICIT
D             1          +-LINEAR       FREE        IMPLICIT
A             2          NONE           FREE        IMPLICIT

DESCRIPTION:
LEFT    RIGHT      RELTYPE    MAX
D       D          +          ?
A       A          +          ?

FEATURES:
NAME          TYPE

MTG:
TOPO
/D1
^/A1++A7
              +D1/A1
```

```
                        +D3/A1+A2
              ^+A2++A6
              +D2/A1
                        +D4/A1+A2
              ^+A2++A5
```

- **Whorls and supra-numerary buds**

Whorls and supra-numerary buds can be encoded in several ways. One possible solution is to use the multiscale property a a MTG as illustrated in the following example.

```
CODE:    FORM-A

CLASSES:
SYMBOL        SCALE      DECOMPOSITION   INDEXATION  DEFINITION
$             0          FREE            FREE        IMPLICIT
U             1          <-LINEAR        FREE        IMPLICIT
E             2          <-LINEAR        FREE        EXPLICIT
V             3          FREE            FREE        EXPLICIT
N             4          NONE            FREE        IMPLICIT

DESCRIPTION:
LEFT          RIGHT         RELTYPE       MAX
U             U             +             ?
U             U             <             1
E             E             <             1
E             E             +             ?

FEATURES:
NAME          TYPE

MTG:
TOPO
/U90
^/E1
              /V1
                        /N1+U91
                        /N2+U91
              /V2
                        /N1+U91
                        /N2+U91
              /V3
                        /N1+U91
^<E2
              /V1
                        /N1+U91
              /V2
                        /N1+U92
                        /N2+U92
              /V3
                        /N1+U92
                        /N2+U92
^<E3 ...
```

Entities E denote internodes. Each internode contains a whorl, whose elements are denoted by class V. Each V can itself be decomposed into several supranumerary positions, denoted by

class N. Then on each position, a growth unit (class U) can be described. Note that within a whorl E, V positions are not connected to one another. They are simply considered as one part of the whorl. This is also true for supra-numerary positions.

- **Plant growth observation**

Plant growth can be observed and described using MTGs. To this end, observation dates are recorded. If some entity is observed at several dates, the new values of its attributes at different dates are recorded on consecutive lines where the topological code of the entity is not repeated but rather replaced by a star symbol '*'.

```
CODE:    FORM-A

CLASSES:
SYMBOL    SCALE       DECOMPOSITION   INDEXATION   DEFINITION
$         0           FREE            FREE         IMPLICIT
P         1           CONNECTED       FREE         IMPLICIT
U         2           NONE            FREE         IMPLICIT

DESCRIPTION:
LEFT    RIGHT    RELTYPE    MAX
U       U        <          1
U       U        +          ?

FEATURES:
NAME          TYPE
Date          DD/MM/YY

MTG:
TOPO                                    Date

/P1
^/U1<U2                                 08/06/00
*                                       19/06/00
*                                       30/06/00
*                                       10/07/00
              +U1<U2                    19/06/00
              *                         30/06/00
              *                         10/07/00
^<U3                                    19/06/00
*                                       30/06/00
              +U1<<U3                   19/06/00
              *                         30/06/00
              *                         10/07/00
              <U4                       30/06/00
              *                         10/07/00
```

- **Branching units located on the bearer according their height from the basis**

In some cases, it is useful to use the index of an entity label to record information. Here, the index of the entity is used to denote the position of an element is used to record the height of this position with respect to the basis of the corresponding axis.

```
CODE:          FORM-A
```

```
CLASSES:
SYMBOL        SCALE      DECOMPOSITION  INDEXATION  DEFINITION
$             0          FREE           FREE        IMPLICIT
X             1          FREE           FREE        IMPLICIT
L             2          NONE           FREE        IMPLICIT
DESCRIPTION:
LEFT          RIGHT          RELTYPE       MAX
X             X              +             ?

FEATURES:
NAME          TYPE

MTG:
TOPO                         Alias

/X90
              /L50+X91
              /L100+X91      A91
              /L123+X92
(A91)                        # Back to axis borne at position L100
              /L10+X92
              /L25+X92
              ...
```

- **Description of a plant from the extremities**

On some plants, it is easier to described branches starting from the bud of the stem on proceeding downward to the stem basis. This is the case for instance, for large trees where biological markers of growth, nodes, growth unit limits, sympodial module, etc., are more leagible near the branch extremities. Here follows a strategy to code the plant in such a case.

```
CODE:    FORM-A

CLASSES:
SYMBOL        SCALE      DECOMPOSITION  INDEXATION  DEFINITION
$             0          FREE           FREE        IMPLICIT
P             1          CONNECTED      FREE        IMPLICIT
U             2          <-LINEAR       FREE        EXPLICIT
E             3          NONE           FREE        EXPLICIT

DESCRIPTION:
LEFT    RIGHT      RELTYPE    MAX
U       U          +          ?
U       U          <          1
E       E          <          1
E       E          +          1

FEATURES:
NAME          TYPE

MTG:
TOPO

/P1
^/U86
```

```
        /E2+U87
^<U87
^<U88
^<U89
        /E10+U89
        /E4+U90
        /E3+U90
        /E1+U90
^<U90
        /E6+U90
        /E3+U90
        /E2+U91
        /E1+U91
^<U91
        /E7+U91    # 7th internode from the apex U91
        /E3+U92    # 3th internode from the apex U91
        /E2+U92    # 2nd internode from the apex U91
```

The entities of the stem must be ordered in the file bottom-up (cf. the firt column where growth units U have increasing indexes). However, the positions within a given growth unit is given from top down to the basis of this growth unit. In addition, if the user wants to enter the stem entities (here growth units) from the top down to the basis of the stem, (s)he can use a laptop computer and insert new growth units (say `U90`) before the ones already observed at the top (say `U91`).

A second solution consists of using a `FORM-B` code. Using this more specific code allows you to enter the entities of the stem from top to basis (see first column).

```
CODE:    FORM-B

CLASSES:
SYMBOL        SCALE      DECOMPOSITION  INDEXATION  DEFINITION
$             0          FREE           FREE        IMPLICIT
P             1          CONNECTED      FREE        IMPLICIT
U             2          <-LINEAR       FREE        EXPLICIT
E             3          NONE           FREE        EXPLICIT

DESCRIPTION:
LEFT    RIGHT      RELTYPE    MAX
U       U          +          ?
U       U          <          1
E       E          <          1
E       E          +          1

FEATURES:
NAME          TYPE

MTG:
TOPO

/P1
^/U91
        /E2+U92    # 2nd internode from the apex U91
        /E3+U92    # 3rd internode from the apex U91
        /E7+U91    # 7th internode from the apex U91
```

```
^/U90
        /E1+U91
        /E2+U91
        /E3+U90
        /E6+U90
^<U89
        /E1+U90
        /E3+U90
        /E4+U90
        /E10+U89
^<U88
^<U87
        /E7+U87
```

5.3 Dressing Files (.drf)

The dressing data are the default data that are used to define the geometric models associated with geometric entities and to compute their geometric parameters when inference algorithms cannot be applied. These data are basically constant values (see the table below) and may be redefined in the dressing file. If no dressing file is defined, default (hard-coded) values are used (see table below). The dressing file .drf , if it exists in the current directory, is always used as a default dressing file.

The dressing data entries can be subdivided into 3 categories (any of these categories can be omitted):

5.3.1 Definition of basic geometric models associated with plant components.

A graphic model can be associated with a component in the following way (all keywords are in boldface characters):

1. First, a set of all the basic geometric models of interest must be defined. This is done by specifying a file containing the geometric description of these models (for a definition of the syntax of geometric models, refer to the annexe section):

   ```
   Geometry   =  file1.geom
   Geometry   =  ../../file2.geom
   ```

   The effect of these lines is to load the geometric models that are defined in files `file1.geom` and in file `../../file2.geom`. Each geometric model defined in these files is associated with a symbolic name. If the same symbolic name is found twice during the loading operation, an error is generated and should be corrected.

   Appearance of these geometric models can be defined also as follows:

   ```
   Appearance   =  ../color1.app
   Appearance   =  ../../color2.app.
   ```

   The effect of these lines is to load the geometric models that are defined in files `../color1.app` and in file `../../color2.app`. Each appearance model defined in these files is associated with a symbolic name. If the same symbolic name is found twice during the loading operation, an error is generated and should be corrected. Appearance can be further used as defined in section Plot(PlantFrame).

2. Any symbolic name (like `internode`) can then be associated with a component using the class of the component as follows:

   ```
   Class   I = internode
   ```

   where `I` corresponds to a class name. This means that all the vertices of class I will have a geometry defined by the geometric model `internode.` Note that class I does not necessarily correspond to a valid class of a MTG (however, it should be a alphabetic letter in `a-z,A-Z`).

Alternatively, to allow for ascendant compatibility with previous versions of AMAPmod, it is possible to directly refer to geometric models defined in .smb files. In this case, the set of geometric models corresponds to the files contained in directory **SMBPath** and a geometric model can be loaded in AMAPmod by identifying a smb file in this directiry. This is done as follows in the dressing file:

```
SMBPath    = ../../databases/SMBFiles
SMBModel   internode2 = nentn105
SMBModel   leaf3 = oakleaf
```

Here, geometric models `internode2` and `leaf3` are respectively associated with polygon files `nentn105.smb` and `oakleaf.smb` which are both located in directory `../../databases/SMBFiles.`

Like exposed above, SMB geometric models can then be associated with vertex classes:

```
Class    J = internode2
Class    F = leaf3
```

Then, global shapes can be defined for branches. This is done using the feature "category" defined for branches. The category of a branch is defined by the category of its first component. Note that the category may depend on the scale at which a branch is considered. For each category, the user can associate a 3 dimensional shape as a 3D bezier curve. The shape of the branch is then fit to the general shape associated with its category.

Assuming a set of Bezier curves are specified in a file beziershapes.crv (for example), we can associate branch categories with the Bezier curves using the following notation:

```
BranchPattern formfile  = ../Curves/beziershapes.crv
Form    1 = curve2 # defines category 1
Form    2 = curve5          # defines category 2
```

Note that the file `beziershapes.`crv is included, using a path relative to the directory where the .drf file itself is located. Alternatively, an absolute filename could be given. The structure of the file `beziershapes.crv` is discribed in

5.3.2 Definition of virtual elements.

Components that don't appear in an MTG description can be added to a MTG (*e.g.* leaves, flowers or fruits). It is possible to define these new symbols as follows:

```
Geometry    =  file1.geom

SMBPath     =  SMBFiles
SMBModel    leaf = feui113

Class    L = leaf
Class    A = apple
Class    B = apricot_flower

LeafClass = L
FlowerClass = B
FruitClass = A
```

A symbol L (a character) is defined and is associated with geometric model leaf. The two last lines associate respectively virtual leaf and fruit components with the geometric model associated with Classes L and A.

5.3.3 Definition of defaults parameters

The value of default parameters used to compute geometric models can be changed in the dressing file. Here follows the complete list of these parameters illustrated on an example:

```
# Default geometric units (these quantities are used
# to divide every value of the corresponding type before use)

LengthUnit = 10
DiameterUnit = 100
AlphaUnit = 1

DefaultAlpha = 30
DefaultTeta = 0
DefaultPhi = 90
DefaultPsi = 180

DefaultCategory = 3
DefaultTrunkCategory = 0

Alpha = Relative
Phyllotaxy = 2/5

DefaultEdge = PLUS    # used for plantframe construction

# Redefinition of default values of the geometric models of
# components (here component S)

MinLength S = 1000
MinTopDiameter S = 20
MinBottomDiameter S = 20

# Redefinition of default values of the geometric models of
# virtual components

LeafLength = 1
LeafTopDiameter = 2
LeafBottomDiameter = 2
LeafAlpha = 0
LeafBeta = 0

FruitLength = 1
FruitTopDiameter = 1
FruitBottomDiameter = 1
FruitAlpha = 0
FruitBeta = 0

FlowerLength = 10
FlowerTopDiameter = 5
FlowerBottomDiameter = 5
FlowerAlpha = 180
```

```
FlowerBeta = 0

DefaultTrunkCategory = 0
DefaultDistance = 1000
NbPlantsPerLine = 6

# Colors for interpolation

MediumThresholdGreen = 1
MediumThresholdRed = 0
MediumThresholdBlue = 0
MinThresholdGreen = 0
MinThresholdRed = 0
MinThresholdBlue = 1
MaxThresholdGreen = 0
MaxThresholdRed = 1
MaxThresholdBlue = 0
```

Any of these keywords can be omitted in the dressing file. If omitted, a parameter takes a default value, hard-coded into AMAPmod. The default values are defined in the following table:

Name of the parameter	Description	Default value	Values
SMBPath	Path where SMB files are recorded	.	STRING
LengthUnit	Unit used to divide all the length data	1	REAL
AlphaUnit	Unit used to divide all the insertion angle	180/PI	REAL
AzimuthUnit	Unit used to divide all the angles	180/PI	REAL
DiameterUnit	Unit used to divide all the diameters	1	REAL
DefaultEdge	Type of edge used to reconstruct a connected MTG	NONE	PLUS or LESS
DefaultAlpha	Default insertion angle (value in degrees with respect to the horizontal plane).	30	REAL
Phyllotaxy	Phyllotaxic angle (given in degrees) or in number of turns over number of leaves for this number of turns.	180	REAL or ratio *e.g.* 2/3
Alpha	Nature of the insertion angle.	Relative	Absolute or Relative
DefaultTeta	Default first Euler angle	0	REAL
DefaultPhi	Default second Euler angle	0	REAL
DefaultPsi	Default third Euler angle	0	REAL
MinLength S	Default length for elements whose class is S.	10	REAL
MinTopDiameter S	Default top diameter for elements whose class is S.	1	REAL
MinBottomDiameter S	Default bottom diameter for elements whose class is S.	1	INT
DefaultTrunkCategory	Default category for elements of the plant trunk. The default category of the other axes	-1	INT

	is their (botanical) order starting at 0 on the trunk.		
DefaultDistance	Distance between the trunk of two plants when several plants are vizualized at a time	100	REAL
NbPlantsPerLine	Number of plants per line when several plants are vizualized at a time	10	INT
MediumThresholdGreen	Green component of the color used for the values equal to the MediumThreshold (see command Plot on a PLANTFRAME) in the case of a color interpolation.	0.05	REAL
MediumThresholdRed	Idem for the red component.	0.07	REAL
MediumThresholdBlue	Idem for the blue component.	0.01	REAL
MinThresholdGreen	Green component of the color used for the values equal to the MinThreshold (see command Plot on a PLANTFRAME) in the case of a color interpolation.	1	REAL
MinThresholdRed	Idem for the red component.	0	REAL
MinThresholdBlue	Idem for the blue component.	0	REAL
MaxThresholdGreen	Green component of the color used for the values equal to the MaxThreshold (see command Plot on a PLANTFRAME) in the case of a color interpolation.	0	REAL
MaxThresholdRed	Idem for the red component	1	REAL
MaxThresholdBlue	Idem for the blue component.	1	REAL
Whorl	Number of virtual symbols per node	2	INT
LeafClass	Class used for a leaf	L	CHAR
LeafLength	Length of the leaf	10	REAL
LeafTopDiameter	Top diameter of the leaf	1	REAL
LeafBottomDiameter	Bottom diameter of the leaf	1	REAL
LeafAlpha	Insertion angle of a leaf	30	REAL
LeafBeta	Azimuthal angle of a leaf (w.r.t its bearer)	180	REAL
FruitClass	Class used for a fruit	F	CHAR
FruitLength	Length of the fruit	10	REAL
FruitTopDiameter	Top diameter of the fruit	1	REAL
FruitBottomDiameter	Bottom diameter of the fruit	1	REAL
FruitAlpha	Insertion angle of a fruit	30	REAL
FruitBeta	Azimuthal angle of a fruit (w.r.t its carrier)	180	REAL
FlowerClass	Class used for a flower	f	CHAR
FlowerLength	Length of the flower	10	REAL
FlowerTopDiameter	Top diameter of the flower	1	REAL
FlowerBottomDiameter	Bottom diameter of the flower	1	REAL
FlowerAlpha	Insertion angle of a flower	30	REAL
FlowerBeta	Azimuthal angle of a flower (w.r.t its carrier)	180	REAL

5.3.4 Example of a dressing file

See **AML File Example**

5.4 Curve Files (.crv)

A curve file contains the specification of Bezier curves. It has the following general structure:

n
curve1
k_1
x_1 y_1 z_1
…
x_{k1} y_{k1} z_{k1}
curve2
k_2
x_1 y_1 z_1
…
x_{k2} y_{k2} z_{k2}
…
curven
k_n
x_1 y_1 z_1
…
x_{kn} y_{kn} z_{kn}

where n, k_1, k_n, are integers and curve1, curve2, …, curven are strings of characters. All coordinates are real numbers.

5.5 Geom Files

5.5.1 Overview

A Geometry file consists of a list of description of 3D geometric objects called *Geom*. A wide range of Geoms is provided covering most classical geometric models and operations in 3D space.

3D geometric models are represented using *Primitives*. Primitives hold their own geometrical parameters and do not contain other Geoms. Operations can be applied to Geoms using *Transformed* and *Group*. They refer to other Geoms in order to form more complex Geoms.

Primitives, Transformed and Group Geoms are now discussed in detail.

- **Primitives**

As mentioned above, Primitives represent most common 3D geometric models. Actually, 11 primitives are provided (*see* **Table 5-1**). Each primitive, and its associated parameters, are described in detail in section section **5.5.3**.

Primitives are described in their own local coordinate system, known as *local reference system*.

Primitives
AmapSymbol
Box
Cone
Cylinder
ElevationGrid
FaceSet
Polyline
QuadMesh
Revolution
Sphere
TriangleStripSet

Table 5-1. Primitive Geoms

3 classes of primitives are distinguished: *curves*, *surfaces* and *volumes*.

Curves

Curves are represented using `polyline`s. A Polyline is a sequence of connected segments, representing a linear approximation of a general parametric curve.

Curves are often used to generate surfaces or to represents the *skeleton* of an object. It generally corresponds to the medial axis of the object.

Surfaces

Surfaces are represented using meshes (*see* **Table 5-2**).

Meshes
AmapSymbol
FaceSet
QuadMesh
TriangleStripSet

Table 5-2. Geoms of type of Mesh

A mesh is a collection of connected polygons, known as *faces*, defining an object in the 3D space. There are several rules to follow when building a polygonal mesh:

- A face must be planar.
- A face must be convex.
- Each face must have the same orientation. The orientation is specified by the order the vertices are described : *clockwise* or *counter-clockwise*.

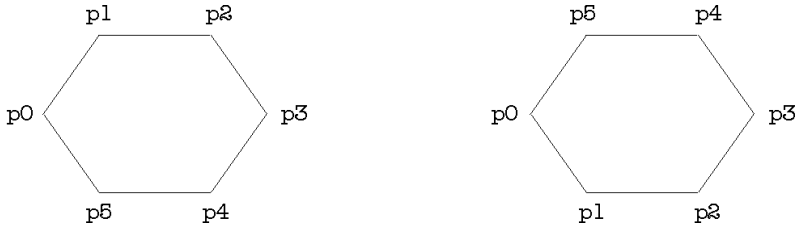


Figure 5-5 Clockwise and counter-clockwise orientation.

Using triangles or quadrilaterals represents an alternative to overcome these requirements and to speed up most of the operations which can be done on these objects.

Polygon meshes are well suited for representing leaves. They can also be used to represent closed shapes, such as fruits or internodes. In that case, an optional flag named *solid* is used to specify whether the mesh represents a closed shape. It may be useful for various operations.

An *elevation grid* is another kind of mesh. It represents a rectangular grid that stores the elevation for each points (*see* • **ElevationGrid**). This type of surface is mainly used for terrain modeling.

Volumes

Volumes describe a set of surfaces, or planes which bound a closed volume. There are generally represented using *solids of revolution*, which are constructed by rotating a 2D planar curve around the *z*-axis (*see* **Table 5-3**).

Solids of revolution
Cone
Cylinder
Revolution
Sphere

Table 5-3. Geoms of type of Solid of revolution

For different purposes such as rendering, volumes are needed to be discretized into meshes. Solids of revolutions are subdivided into *slices* and *stacks*. Slices specify the number of

subdivisions around the *z*-axis whereas stacks specify the number of subdivisions along the *z*-axis (*see* **Figure 5-6**). The simpler the discretization the quicker the object is rendered.

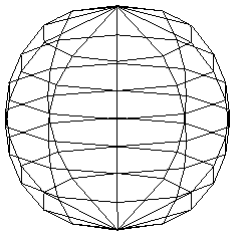


Figure 5-6 A solid of revolution subdivided into slices and stacks.

They are generally used to represent the wooden part or the crown of a tree.

- **Transformed**

The transformed Geoms allows you to specify the position, the orientation and the shape of a Geom in 3D space. It binds a *transformation* to a Geom. A transformation changes an object from its local coordinate system to the global coordinate system known as the *global coordinate space*.

If a Transformed Geom contains another Transformed Geom, they are said to be *nested* and their settings have a cumulative effect. Consequently you need to take care about the order of transformation.

In total, 6 transformed are provided (*see* **Table 5-4**). Each type, and its associated parameters, are described in detail in section **5.5.3**.

Transformed
AxisRotated
EulerRotated
OrthoTransformed
Scaled
Tapered
Translated

Table 5-4. Transformed Geoms

- **Group**

A Group allows you to combine a collection of Geoms in order to build more complex Geoms.

5.5.2 File format syntax

The following section outlines the syntax for the Geometry ASCII file format.

- **Conventions**

In a Geometry file, extra white space created by spaces, tabs and new lines is ignored. Comments serve as an aid to the user. There are two comments delimiters.

– The sharp sign # serves to delimit a single-line comment. It can be placed anywhere on a line and everything to the right of the delimiter on this line is treated as a comment and consequently is ignored.

```
# This is an example of a single line comment
```

– The pair (#,#) serves to delimit a block comment. Every character that falls between the (# and the #) is part of a comment. A comment pair can be placed anywhere. A tab, space, or new-line is permitted and can span multiple lines. For example:

```
(#
   This is an example of a block comment
#)
```

- **Writing Geoms**

Whatever the Geom, the syntax follows the same writing rules. A Geom is written with the following structure:

– The type of the Geom (**Table 5-1** and **Table 5-2**). Each word within the type name begins with an uppercase letter.

– An optional name for the object. As mentioned as above, naming is useful when you wish to refer to the object later in the file. Names can not begin with a digit (`0-9`) and can contain only alphabetical characters (`a-z A-Z`), digits (`0-9`) and the underscore (_). Types of Geom and fields name are reserved keywords and cannot be used for naming. The system assigns a unique name to all the Geoms without name.

– Fields within the Geom if any. A Field appears as a pair consisting of a name and a value. Fields description is enclosed in braces (`{ }`).

This example creates a cone frustum from a Cylinder and a transformation.

```
Tapered a_frustum{
     TopRadius .25
     BaseRadius .1
     Geometry Cylinder { }
}
```

- **Writing Fields**

Fields are predefined in each Geom and require a certain type of value (*see* **Table 5-5**)

Type	Description
BOOLEAN	A boolean value : `True` or `False`
GEOM	A Geom description or a reference to a Geom
INDEX	A list of integer numbers. All numbers must be positives: `<1,2,5,3,6,1>`
INTEGER	An integer number : `5`
REAL	An floating point number : `1.2`
STRING	A sequence of characters. It must appear within double quotes : `'string'`
VECTOR2	A 2D vector. It consists of 2 floating point numbers : `<1.2,0.5>`
VECTOR3	A 3D vector. It consists of 3 floating point numbers : `<1.2,0.5,6.3>`

Table 5-5. Basic field types

There are 2 types of fields : mandatory fields and optional fields. Unspecified optional fields are set to their default values. Furthermore fields can be specified in any order.

A field is written with the following elements:

- The name of the field. Each words within the field name begins with an uppercase letter.
- The value(s) of the field. According to the field, it can be a single value or an array of values. An array of value is expressed as a serie of singles values separated by commas and enclosed in square brackets `[ ]`.

The following example creates a tree as the union of a Cylinder which represents the trunk and a solid of Revolution which represents the crown (*see* **Figure 5-7**)

```
Group a_tree {
   GeometryList [ # A multiple field of type of Geom
      Cylinder the_trunk {
          Height 2 # A single field of type of Real
          Radius .25 # A single field of type of Real
      },
      Translated the_crown {
          Translation <0,0,2> # A single field of type of Vector3
          Geometry Revolution { # A single field of type of Geom
          PointList [<0.25,0>,<1,1>,<1,3>,<0,4>] # A multiple field
                                                  of type of Vector2
      }
   }
}
```

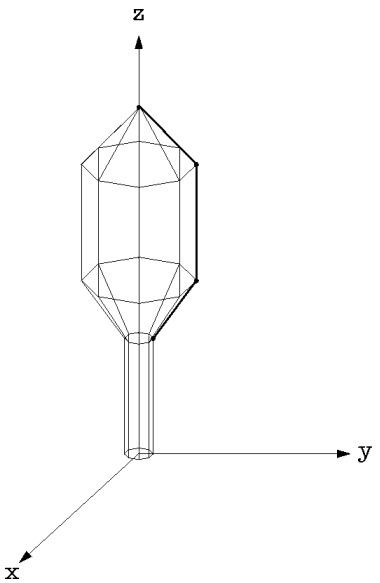


Figure 5-7 The `a_tree` Geom

- **Using multiple instance of a Geom**

Sharing Geoms is often used when you need to reuse a desired Geom. Instead of repeating its description, you can name it the first time you describe it and then refer to it by name each

time you want to reuse it. Shared instancing allows you to save both time and space when writing a Geometry file.

Here's an example of how to share an already described Geom representing a tree in order to build a stand of 3 trees:

```
Group a_stand {
      GeometryList [
            Translated { Translation <-10,-5,0> Geometry a_tree }
            Translated { Translation <3,2,0> Geometry a_tree }
            Translated { Translation <-3,4,0> Geometry a_tree }
      ]
}
```

- **Including other files**

To include a file within another file, you can use the `:include` directive. It is useful when you already have a set of predefined Geoms in a file, it is then possible to refer to them just by including the file.

```
:include "a_stand.geom"
Group a_forest {
      GeometryList [
            Translated { Translation <-15,-3,0> Geometry a_stand }
            Translated { Translation <13,5,0> Geometry a_stand }
      ]
}
```

5.5.3 Geoms format reference

The following section lists all Geom types in alphabetical order. Each object description begins with a brief comment. Then all fields are listed within a table, with the name, the type, the default values (if any) and a comment.

- **AmapSymbol**

Description

The `AmapSymbol` is a primitive Geom, describing an object of class of **Mesh** (*see **Surfaces***). It is stored in the SMB file format of the Amap Software. This is provided for ascendant compatibility.

Fields description

Name	Type	Defaults	Description
`FileName`	`STRING`	NONE	specifies the name of the SMB file to bind to the symbol. It must contain the full path and `.SMB` extension if needed. The corresponding file must exist.
`Solid`	`BOOLEAN`	`False`	specifies whether the symbol represents a closed surface.

Example

```
AmapSymbol a_symbol {
(#
      It creates a polygon mesh from the file nentn105.smb. As it
      represents a trunk of cone it can be marked as a solid (see
      Volumes).
#)
      textbf{FileName} “/home/user/Datas/SMBFiles/nentn105.smb”
      textbf{Solid} True
}
```

- **AxisRotated**

Description

The `AxisRotated` Geom describes a rotated Geom of a specified angle about a specifed axis.

Fields description

Name	Type	Defaults	Description
`Axis`	`VECTOR3`	`<0,0,1>`	specifies axis of the rotation.
`Angle`	`REAL`	`0`	specifies the angle of the rotation. It must be in degrees.
`Geometry`	`GEOM`	NONE	specifies the Geom to be rotated.

Example

```
AxisRotated an_axis_rotated{
(#
      It creates a Geom from the rotation by 45 degrees about the y-
      axis of an arbitrary Geom called a_geom.
#)
      Axis <0,1,0>
      Angle 45
      Geometry a_geom
}
```

- **Box**

Description

The `Box` is a primitive describing a rectangular axis-aligned box.

Fields description

Name	Type	Defaults	Description
`Size`	`VECTOR3`	`<.5,.5,.5>`	specifies the x, y and z extents of the box's width, depth and height. Each coordinates must be positive.

Example

```
Box a_box {
      Size <1,4,2>
}
```

- **Cone**

Description

The `Cone` describes a cone frustum whose base lies into the *x*-*y* plane and central axis is the *z*-axis. The ratio between the top radius and the base radius is defined by a taper rate. Thus the top radius is computed as the product between the radius and the taper rate.

Fields

Name	Type	Defaults	Description
`Radius`	`REAL`	`0.5`	specifies the radius of the cone. It must be strictly positive.
`Height`	`REAL`	`1`	specifies the height of the cone. It must be strictly positive.
`Taper`	`REAL`	`0`	specifies the rate of taper. It must belong to the range [0,1].
`Caps`	`BOOLEAN`	`True`	specifies whether the bottom and top sides are visible.
`Slices`	`INTEGER`	`8`	specifies the number of subdivisions around the *z*-axis when discretizingthe cone.

Example of syntax

```
Cone a_cone{
      Radius 2
      Height 2
      Taper 0.25
      Caps False
      Slices 5
}
```

- **Cylinder**

Description

The `Cylinder` describes a cylinder whose base lies into the *x*-*z* plane and central axis is the *z*-axis.

Fields description

Name	Type	Defaults	Description
`Radius`	`REAL`	`0.5`	specifies the radius of the cylinder. It must be strictly positive.
`Height`	`REAL`	`1`	specifies the height of the cylinder. It must be strictly positive.
`Caps`	`BOOLEAN`	`True`	specifies whether the bottom and top sides of the cylinder are visible.
`Slices`	`INTEGER`	`8`	specifies the number of subdivisions around the *z*-axis when discretizing the cylinder.

Example

```
Cylinder a_cylinder {
      Radius 2
      Height 2
      Caps True
      Slices 5
}
```

- **ElevationGrid**

Description

The ElevationGrid primitive describes a regular grid of a specified number of rows and columns and the elevation on each points of that grid. Heights are described row-major order (along the *x*-axis first), left to right and top to bottom. There are exactly rows*columns values in the height list.

Fields description

Name	Type	Defaults	Description
HeightList	REAL[ ]	NONE	specify the elevation associated to each points of the grid. There must be exactly XDim*Ydim values.
XDim	INTEGER	NONE	specifies the dimension of the grid along the *x*-axis. It must be greater than 1.
YDim	INTEGER	NONE	specifies the dimension of the grid along the *y*-axis. It must be greater than 1.
XSpacing	REAL	1	specifies the distance between two consecutive points in the *x*-direction. It must be strictly positive.
YSpacing	REAL	1	specifies the distance between two consecutive points in the *y*-direction. It must be strictly positive.

Example

```
ElevationGrid an_elevation_grid {
(#
      It creates a 4 x 4 elevation grid with total size equal to 20 x
      20.
#)
      HeightList [ 0,1,1,0, 1,3,2,0, 2,4,3,1, 1,2,1,0 ]
      XDim 4
      YDim 4
      XSpacing 5
      YSpacing 5
}
```

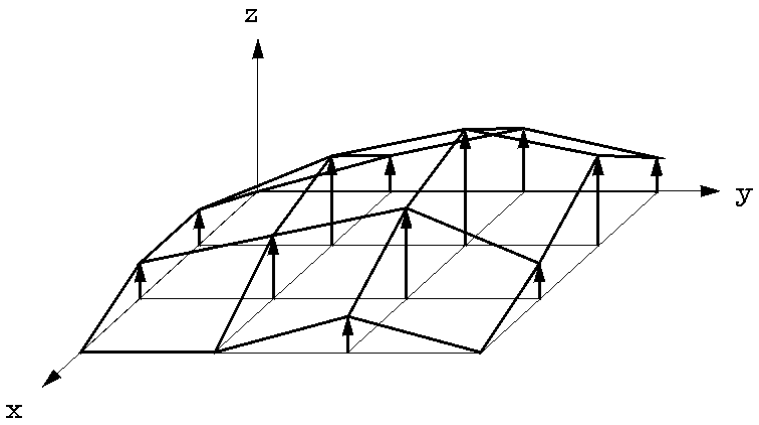


Figure 5-8 The `an_elevation_grid` Geom

- **EulerRotated**

Description

The `EulerRotated` Geom describes a composition of rotations by *azimuth* about the *z*-axis, by *elevation* about the rotated *y*-axis and by twist about the rotated *x*-axis.

Fields description

Name	Type	Defaults	Description
`Azimuth`	`REAL`	`0`	specifies the angle of the rotation about the *z*-axis.
`Elevation`	`REAL`	`0`	specifies the angle of rotation about the rotated *y*-axis.
`Twist`	`REAL`	`0`	specifies the angle of rotation about the rotated *x*-axis.

Example

```
EulerRotated {
      Azimuth 45
      Elevation 30
      Twist 60
      Geometry a_geom
}
```

- **FaceSet**

Description

A `FaceSet` describes a surface formed by connected faces (see ***Surfaces***). Polygons are specified using indices into a list of vertices located at the specified coordinates.

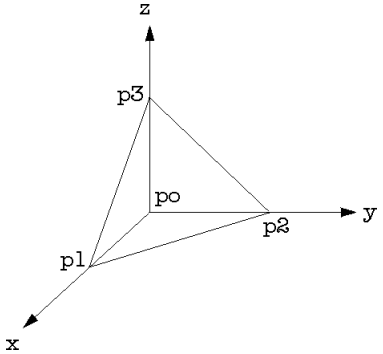


Figure 5-9 The `a_face_set` Geom

Fields description

Name	Type	Defaults	Description
PointList	VECTOR3[ ]	NONE	Specifies the coordinates of the constructing points.
NormalList	VECTOR3[ ]	EMPTY	Specifies the coordinates of the normals attached to the points. If you want the object to appear smoother, you need to explicitly specify normals. In that case, there must be the same number of points and normals. By default, the array of values is empty and normal coordinates are automatically computed.
IndexList	INDEX[ ]	NONE	Specifies the faces, each specified as a serie of indices into the list of points and the list of normals if specified.
CCW	BOOLEAN	True	Indicates whether each polygon's points are listed in counterclockwise (True) or clockwise (False) order when viewed from the front (*see* **Figure 5-5**).
Solid	BOOLEAN	False	Specifies whether this polygon mesh represent a closed surface.
Skeleton	GEOM	Null	Specifies the skeleton of this polygon mesh. It must be a Geom of type of Polyline (*see* • **Polyline**).

Example

```
FaceSet a_face_set {
(#
      It creates a unit tetrahedron from 4 faces which are triangles.
#)
      PointList [ <0,0,0>, <1,0,0>, <0,1,0>, <0,0,1> ]
      IndexList [ <0,2,1>, <0,1,3>, <0,3,2>, <1,2,3> ]
      CCW True
      Solid True
      Skeleton a_polyline
}
```

- **Group**

A Group Geom combines a list of Geoms in order to build a more complex Geom.

Fields description

Name	Type	Defaults	Description
GeometryList	GEOM[ ]	NONE	Specifiesthe list of Geoms which are to be grouped.
Skeleton	GEOM	Null	Specifies the skeleton of this group. It must be a Geom of type of Polyline (*see* • **Polyline**).

Example

```
Group a_Group {
      GeometryList [ a_geom1, a_geom2, a_geom3 ]
      Skeleton a_polyline
}
```

- **OrthoTransformed**

Description

The `OrthoTransformed` Geom applies a change of basis to a Geom. A basis is expressed as a triplet of 3 orthogonal normalized vectors: *primary* direction, *secondary* direction and *ternary* direction. Meanwhile the ternary direction is computed as the cross product between the primary and the secondary directions, thus the basis can be specified by mean of the primary and secondary directions only.

Fields description

Name	Type	Defaults	Description
`Primary`	`VECTOR3`	`<1,0,0>`	specifies the primary direction. The vector will be automatically normalized if needed.
`Secondary`	`VECTOR3`	`<0,1,0>`	specifies the secondary direction. It must be orthogonal to the primary direction, that is, the dot product between those 2 vectors must be 0. The vector will be automatically normalized if needed.
`Geometry`	`GEOM`	NONE	specifies the Geom wich is affected by the scale.

Example

```
OrthoTransformed an_orthotransformed {
(#
      The ternary direction is <1,0,0>
#)
      Primary <0,1,0>
      Secondary <0,0,1>
      Geometry a_geom
}
```

- **Polyline**

Description

A `Polyline` describes a curve formed by connected segment located at specified coordinates.

Fields description

Name	Type	Defaults	Description
`PointList`	`VECTOR3[ ]`	NONE	specifies the coordinates of the constructing points of this polyline. There must be at least 2 points.

Example

```
Polyline {
      PointList [ <0,0,0>, <0,0,1> ]
}
```

- **QuadMesh**

Description

A `QuadMesh` describes a surface formed by quadrilaterals, 4 sided polygons. The mesh formed by the quadrilaterals is a grid with a specified number of rows and columns. Vertices are ordered left to right and top to bottom. There are rows * columns points in the mesh located at the specified coordinates.

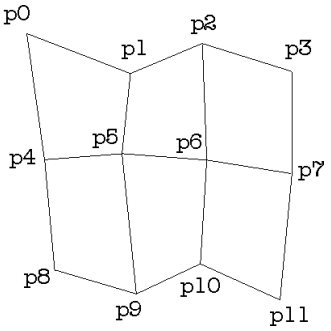


Figure 5-10 A 3 by 4 quadrilateral mesh.

Fields description

Name	Type	Defaults	Description
`PointList`	`VECTOR3[ ]`	NONE	specifies the coordinates of the constructing points.
`NormalList`	`VECTOR3[ ]`	`[ ]`	specifies the coordinates of the normals attached to the points. If you want the object to appear smoother, you need to explicitly specify normals. In that case, there must be the same number of points and normals.
`RowDim`	`INTEGER`	NONE	Specifies the number of rows within the mesh. It must be greater or equal than 2.
`ColDim`	`BOOLEAN`	`True`	Specifies the number of columns within the mesh. It must be greater or equal than 2.
`Solid`	`BOOLEAN`	`False`	specifies whether this quad mesh represent a closed surface.
`Skeleton`	`GEOM`	`Null`	specifies the skeleton of this quad mesh. It must be a Geom of type of Polyline (*see* • **Polyline**).

Example

A 3x4 quadrilateral mesh.

```
QuadMesh a_quad_mesh {
      PointList [ <0,0,0>, <1,0,0>, <2,0,0>, <3,0,0>,
                  <0,1,0>, <1,1,0>, <2,1,0>, <3,1,0>,
                  <0,2,0>, <1,2,0>, <2,2,0>, <3,2,0> ]
      RowDimension 3
      ColDimension 4
      Solid False
      Skeleton a_polyline
}
```

- **Revolution**

Description

The Revolution primitive describes a solid (*see* ***Volumes***) generated by the rotation of a planar curve about the *z*-axis.

The number of points within the generatrix curve determines the number of subdivisions along the *z*-axis when discretizing or rendering the object.

Fields description

Name	Type	Defaults	Description
`PointList`	`VECTOR2[ ]`	NONE	specifies the coordinates of the 2D curve to be rotated.
`Slices`	`INTEGER`	`8`	specifies the number of subdivisions around the *z*-axis when generating the solid.

Example

```
Revolution a_revolution {
      PointList [ <6,0>, <5,1>, <3,3>, <4,5>, <6,8>, <6,10> ]
      Slices 10
}
```

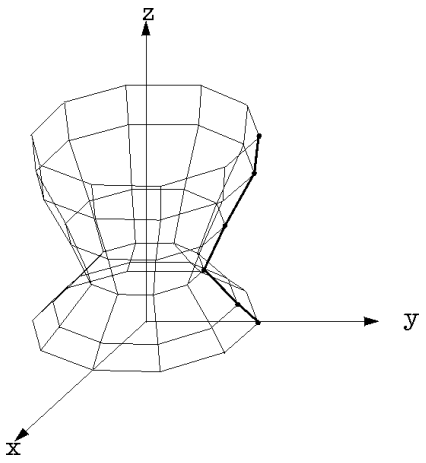


Figure 5-11 The `a_revolution` Geom.

- **Scaled**

Description

The `Scaled` Geom applies a scale to a Geom.

Fields description

Name	Type	Defaults	Description
`Scale`	`VECTOR3`	`<1,1,1>`	specifies the scaling factors along the *x*-axis, the *y*-axis and the *z*-axis. Each of the coordinates must be different from 0.
`Geometry`	`GEOM`	NONE	specifies the Geom wich is affected by the scale.

Example

```
Scaled a_scaled {
      Scale <1,2,-.5>
      Geometry a_geom
}
```

- **Sphere**

Description

The `Sphere` describes a sphere of a specified radius and centered at (0,0,0).

Fields description

Name	Type	Defaults	Description
`Radius`	`REAL`	`0.5`	specifies the radius of the sphere. It must be strictly positive.
`Slices`	`INTEGER`	`8`	specifies the number of subdivisions around the *z*-axis when discretizing the sphere. It must be strictly positive.
`Stacks`	`INTEGER`	`8`	specifies the number of subdivisions along the *z*-axis when discretizing the sphere. It must be strictly positive.

Example

```
Sphere {
      Radius 1.5
      Slices 32
      Stacks 16
}
```

- **Tapered**

Description

The `Tapered` applies a Taper deformation to a Geom. A Taper deforms an object in order to be able to bound the object within a cone frustum of a specifed base radius and top radius.

For each point composing an object, a Taper scale the polar coordinates according the *z*-coordinate. The amplitude of the scale is given by the radii.

Fields description

Name	Type	Defaults	Description
`BaseRadius`	`REAL`	`1`	specifies the base radius of the cone frustum.
`TopRadius`	`REAL`	`1`	specifies the top radius of the cone frustum.
`Geometry`	`GEOM`	NONE	specifies the Geom wich is affected by the taper.

Example

```
Tapered a_tapered {
      BaseRadius 2
      TopRadius 0.5
      Geometry a_geom
}
```

- **Translated**

A `Translated` Geom applies a translation of a specified vector to a Geom.

Fields description

Name	Type	Defaults	Description
`Translation`	`VECTOR3`	`<0,0,0>`	specifies translation vector.
`Geometry`	`GEOM`	NONE	specifies the Geom wich is affected by the translation.

Example

```
Translated a_translated {
      Translation <.5,3,-3>
      Geometry a_geom
}
```

- **TriangleStripSet**

Description

A `TriangleStripSet` describes a surface formed by set of triangle strips. A triangle strip is a serie of triangles described in a specific order as illustred in the figure **Figure 4-12**.

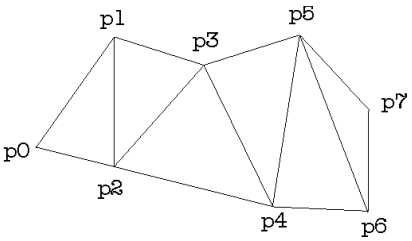


Figure 5-12 A triangle strip.

A triangle strip is specifed using indices to a list of vertices located at the specified coordinates.

Fields description

Name	Type	Defaults	Description
`PointList`	`VECTOR3[ ]`	NONE	specifies the coordinates of the constructing points.
`NormalList`	`VECTOR3[ ]`	`[ ]`	specifies the coordinates of the normals attached to the points. If you want the object to appear smoother, you need to explicitly specify normals. In that case, there must be the same number of points and normals.
`IndexList`	`INDEX[ ]`	NONE	specifies the triangle strips, each specified as a serie of indices into the list of points and the list of normals if specified.
`Solid`	`BOOLEAN`	`False`	specifies whether this object represent a closed surface.
`Skeleton`	`GEOM`	`Null`	specifies the skeleton of this object. It must be a Geom of type of Polyline (*see* • **Polyline**)

Example

```
TriangleStripSet a_triangle_strip_set {
     (# It creates a simple tetrahedron (see Figure 5-9).
     It comes to create only one triangle strip.
#)
     PointList [ <0,0,0>, <1,0,0>, <0,1,0>, <0,0,1> ]
     IndexList [ <0,1,3,2,0,1> ]
     Solid True
     Skeleton a_polyline
}
```

5.6 Glance configuration file (.cgf)

On Irix Only

```
# Pathnames to AMAP directories
ARCHITECTURE =
IMAGES =
LIGNE = Linetree
PARAMETRE =
PERSPECTIVE =
POLYGONE = .
RELIEF =
SCENE =
SYMBOLES = Symbol
TEXTURE =
#
# Name and parameters for AMAP Growth Engine
PROGRAMME =
ECRITURE = 1
HIERARCHIE = 0
FEUILLE =
FLEUR =
NOM_AGE = 0
SEMENCE = 0
SIMPLIFICATION = 1
FICLIB =
#
# Parameters for Images
MESSAGE =
ROUGE = 255
VERT = 255
BLEU = 255
#
# Parameters for AMAP
NOBOX = 1
FOND =
LISSAGE = 0
FORMAT = 2
RESX = 768
RESY = 576
VIDEO_BOARD = 1
VFR_GAMMA = 1.000000
```

5.7 STAT

An ASCII file format is defined for each of the following object type of the STAT module:

COMPOUND
CONVOLUTION
DISTRIBUTION, RENEWAL
HIDDEN_MARKOV
HIDDEN_SEMI-MARKOV
HISTOGRAM
MARKOV
MIXTURE
SEMI-MARKOV
SEQUENCES
TIME_EVENTS
TOPS
TOP_PARAMETERS
VECTOR_DISTANCE
VECTORS

5.7.1 type COMPOUND

A compound (or stopped-sum) distribution is defined as the distribution of the sum of n independent and identically distributed random variables X_i where n is the value taken by the random variable N. The distribution of N is referred to as the sum distribution while the distribution of the X_i is referred to as the elementary distribution. Consider the following example:

```
COMPOUND_DISTRIBUTION

SUM_DISTRIBUTION
NEGATIVE_BINOMIAL  INF_BOUND : 5  PARAMETER : 3.2  PROBABILITY : 0.4

ELEMENTARY_DISTRIBUTION
BINOMIAL INF_BOUND : 0  SUP_BOUND : 5  PROBABILITY : 0.8
```

The first line gives the distribution type. The parametric sum distribution and the parametric elementary distribution are then defined in subsequent lines according to the syntactic form defined for the type `DISTRIBUTION`.

5.7.2 type CONVOLUTION

The distribution of the sum of independent random variables is the convolution of the distributions of these elementary random variables. Consider the following example:

```
CONVOLUTION 2 DISTRIBUTIONS

DISTRIBUTION 1
BINOMIAL  INF_BOUND : 2  SUP_BOUND : 5  PROBABILITY : 0.8

DISTRIBUTION 2
NEGATIVE_BINOMIAL  INF_BOUND : 5  PARAMETER : 3.2  PROBABILITY : 0.4
```

The first line gives the distribution type and the number of elementary distributions (2 or 3). The elementary parametric distributions are then defined in subsequent lines according to the syntactic form defined for the type `DISTRIBUTION`.

5.7.3 type DISTRIBUTION, type RENEWAL

The available parametric discrete distributions are the binomial distribution, the Poisson distribution, the negative binomial distribution and the uniform (rectangular) distribution with an additional shift parameter which defines the lower bound to the range of possible values. The name of the distribution is first given, then the name of each parameter followed by its actual value as shown in the following examples:

```
BINOMIAL  INF_BOUND : 2  SUP_BOUND : 5  PROBABILITY : 0.8

POISSON  INF_BOUND : 0  PARAMETER : 12.2

NEGATIVE_BINOMIAL  INF_BOUND : 5  PARAMETER : 3.2  PROBABILITY : 0.4

UNIFORM  INF_BOUND : 2  SUP_BOUND : 5
```

`INF_BOUND` and `SUP_BOUND` are integer-valued parameters while `PARAMETER` and `PROBABILITY` are real-valued parameters.

For every parametric distributions, the following constraint applies to the shift parameter:

$0 \leq$ `INF_BOUND` ≤ 200

For a `BINOMIAL` or a `UNIFORM` distribution, the following constraint applies to the parameters `INF_BOUND` and `SUP_BOUND` which define the range of possible values:

$0 <$ `SUP_BOUND - INF_BOUND` ≤ 500

For a `BINOMIAL` distribution, the following constraint applies to the probability of 'success':

$0 \leq$ `PROBABILITY` ≤ 1

For a `POISSON` distribution, the following constraint applies to the parameter (which is equal to the mean):

$0 <$ `PARAMETER` ≤ 200

For a `NEGATIVE_BINOMIAL` distribution, the following constraints apply to the parameters:

$0 <$ `PARAMETER`
$0 <$ `PROBABILITY` < 1
`PARAMETER (1 - PROBABILITY) / PROBABILITY` ≤ 200.

A renewal process is built from a discrete parametric distribution (`BINOMIAL`, `POISSON` or `NEGATIVE_BINOMIAL`) termed the inter-event distribution which represents the time interval between consecutive events. Hence, the types `DISTRIBUTION` and `RENEWAL` share the same ASCII file format.

5.7.4 type HIDDEN_MARKOV

A hidden Markov chain is constructed from an underlying Markov chain and nonparametric observation (or state-dependent) distributions. Consider the following example:

```
HIDDEN_MARKOV_CHAIN

2 STATES
ORDER 1

INITIAL_PROBABILITIES
0.8  0.2

TRANSITION_PROBABILITIES
0.6  0.4
0.1  0.9

OBSERVATION_PROBABILITIES

2 VARIABLES

VARIABLE 1

STATE 0
OUTPUT 0 : 1.0

STATE 1
OUTPUT 0 : 0.2
OUTPUT 1 : 0.8

VARIABLE 2

STATE 0
OUTPUT 0 : 0.2
OUTPUT 1 : 0.4
OUTPUT 2 : 0.4

STATE 1
OUTPUT 0 : 0.8
OUTPUT 1 : 0.1
OUTPUT 2 : 0.1
```

The first line gives the object type. The underlying Markov chain is then defined on subsequent lines according to the syntactic form defined for the type MARKOV. The observation (or state-dependent) probabilities relating the output processes to the non-observable state process are then defined. Since the process is 'hidden', at least one possible output should be observable in more than one state.

5.7.5 type HIDDEN_SEMI-MARKOV

A hidden semi-Markov chain is constructed from an underlying semi-Markov chain (first-order Markov chain representing transition between states and state occupancy distributions associated to the semi-Markovian states) and nonparametric observation (or state-dependent) distributions. The state occupancy distributions are defined as objects of type DISTRIBUTION with the additional constraint that the minimum time spent in a given state is 1 (INF_BOUND ≥ 1). Consider the following example:

```
HIDDEN_SEMI-MARKOV_CHAIN

4 STATES

INITIAL_PROBABILITIES
0.8  0.2  0.0  0.0

TRANSITION_PROBABILITIES
0.0  0.6  0.4  0.0
0.0  0.0  0.7  0.3
0.0  0.1  0.8  0.1
0.0  0.0  0.0  1.0

STATE 0 OCCUPANCY_DISTRIBUTION
NEGATIVE_BINOMIAL  INF_BOUND : 2  PARAMETER : 3.2  PROBABILITY : 0.4

STATE 1 OCCUPANCY_DISTRIBUTION
BINOMIAL  INF_BOUND : 1  SUP_BOUND : 12  PROBABILITY : 0.6

OBSERVATION_PROBABILITIES

1 VARIABLE

VARIABLE 1

STATE 0
OUTPUT 0 : 1.0

STATE 1
OUTPUT 0 : 0.3
OUTPUT 1 : 0.6
OUTPUT 2 : 0.1

STATE 2
OUTPUT 0 : 0.2
OUTPUT 1 : 0.4
OUTPUT 2 : 0.4

STATE 3
OUTPUT 2 : 1.0
```

Note that absorbing states such as state 3 ($p_{33} = 1$) are by nature Markovian. It is also possible to define nonabsorbing Markovian states such as state 2 ($0 < p_{22} < 1$). In this case, the resulting model is a hybrid hidden Markov/semi-Markov chain.

The first line gives the object type. The underlying semi-Markov chain (embedded first-order Markov chain and state occupancy distributions associated to the semi-Markovian states) is then defined on subsequent lines according to the syntactic form defined for the type `SEMI-MARKOV`. The observation (or state-dependent) probabilities relating the output processes to the non-observable state process are then defined. Since the process is 'hidden', at least one possible output should be observable in more than one state.

5.7.6 type HISTOGRAM

The syntactic form of the type `HISTOGRAM` consists in giving, in a first column, the values in increasing order and, in a second column, the corresponding frequencies. If a value is not given, the corresponding frequency is assumed to be null. Consider the following example:

```
2   1
3   2
4   4
5  12
6  14
7   6
8   3
9   2
10  1
12  2
14  1
```

5.7.7 type MARKOV

Consider the following example of an homogeneous Markov chain:

```
MARKOV_CHAIN

2 STATES
ORDER 2

INITIAL_PROBABILITIES
0.8  0.2

TRANSITION_PROBABILITIES
0.6  0.4
0.1  0.9
0.3  0.7
0.2  0.8
```

The first line gives the object type. Then, the number of states (between 2 and 15) and the order (between 1 and 4) are defined on the two subsequent lines. On the next lines, the initial probabilities and the transition probabilities are given. Since, the initial probabilities and the transition probabilities for a given memory constitute distributions, the elements of a line should sum to one.

It is also possible to define observation (or state-dependent) probabilities if each possible output can be observed in a single state. With this restriction, the state space corresponds to a partition of the output space and the overall process is a lumped process:

```
OBSERVATION_PROBABILITIES

2 VARIABLES

VARIABLE 1

STATE 0
OUTPUT 0 : 1.0

STATE 1
OUTPUT 1 : 0.2
OUTPUT 2 : 0.8

VARIABLE 2

STATE 0
OUTPUT 0 : 0.7
OUTPUT 1 : 0.3

STATE 1
```

OUTPUT 2 : 0.6
OUTPUT 3 : 0.4

Consider the following example of a non-homogeneous Markov chain:

```
NON-HOMOGENEOUS_MARKOV_CHAIN

3 STATES
ORDER 1

INITIAL_PROBABILITIES
0.5  0.3  0.2

TRANSITION_PROBABILITIES
0.6  0.2  0.2
0.1  0.8  0.1
0.2  0.1  0.7

STATE 0 HOMOGENEOUS

STATE 1 NON-HOMOGENEOUS
MONOMOLECULAR FUNCTION  PARAMETER 1 : 0.99  PARAMETER 2 : -0.34  PARAMETER 3 :
0.3

STATE 2 NON-HOMOGENEOUS
LOGISTIC FUNCTION  PARAMETER 1 : 0. 99  PARAMETER 2 : 2.8  PARAMETER 3 : 0.2
```

The first line gives the object type. Then, the initial probabilities and the transition probabilities are given in the same way as for an homogeneous Markov chain. The non-homogeneous / homogeneous character is then defined state by state. In the case of a non-homogeneous transition distribution, the function $p_{ii}(t)$ represents the self-transition in state i as a function of the index parameter t. The corresponding transition distribution defined in the transition probability matrix gives the relative weights of the probabilities of leaving state i.

For a MONOMOLECULAR function ($p_{ii}(t) = a + b \exp(ct)$), the following constraints apply:

$0 \leq$ PARAMETER 1 ≤ 1
$0 \leq$ PARAMETER 1 + PARAMETER 2 ≤ 1
PARAMETER 3 > 0

For a LOGISTIC function ($p_{ii}(t) = a / (1 + b \exp(ct))$), the following constraints apply:

$0 \leq$ PARAMETER 1 ≤ 1
$0 \leq$ PARAMETER 1 / (1. + PARAMETER 2) ≤ 1
PARAMETER 3 > 0

5.7.8 type MIXTURE

A mixture is a parametric model of classification where each elementary distribution or component represents a class with its associated weight. Consider the following example:

```
MIXTURE 2 DISTRIBUTIONS

DISTRIBUTION 1  WEIGHT : 0.3
BINOMIAL  INF_BOUND : 2  SUP_BOUND : 5  PROBABILITY : 0.8

DISTRIBUTION 2  WEIGHT : 0.7
NEGATIVE_BINOMIAL  INF_BOUND : 5  PARAMETER : 3.2  PROBABILITY : 0.4
```

The first line gives the distribution type and the number of components of the mixture (between 2 and 4). The components are then defined on two lines, the first one giving the associated weight and the second one giving the definition of the elementary parametric distribution according to the syntactic form defined for the type `DISTRIBUTION`. The weights should sum to one.

5.7.9 type SEMI-MARKOV

A semi-Markov chain is constructed from a first-order Markov chain representing transition between states and state occupancy distributions associated to the semi-Markovian states. The state occupancy distributions are defined as objects of type DISTRIBUTION with the additional constraint that the minimum time spent in a given state is at least 1 (INF_BOUND ≥ 1). Consider the following example:

```
SEMI-MARKOV_CHAIN

4 STATES

INITIAL_PROBABILITIES
0.8  0.2  0.0  0.0

TRANSITION_PROBABILITIES
0.0  0.6  0.4  0.0
0.0  0.0  0.7  0.3
0.0  0.1  0.8  0.1
0.0  0.0  0.0  1.0

STATE 0 OCCUPANCY_DISTRIBUTION
NEGATIVE_BINOMIAL  INF_BOUND : 2  PARAMETER : 3.2  PROBABILITY : 0.4

STATE 1 OCCUPANCY_DISTRIBUTION
BINOMIAL  INF_BOUND : 1  SUP_BOUND : 12  PROBABILITY : 0.6
```

The first line gives the object type while the second line gives the number of states (between 2 and 15). The embedded first-order Markov chain is then defined on subsequent lines by its initial probabilities and its transition probabilities (note that, unlike for the type MARKOV, the order should not be specified). Since this embedded Markov chain represents only transitions between distinct states, the self-transitions (i.e. elements of the main diagonal) should be equal to zero except in the case of Markovian states where the self-transitions are strictly positive (e.g. states 2 and 3 in the above example). The state occupancy distributions are then defined for each semi-Markovian state according to the syntactic form defined for the type DISTRIBUTION with the additional constraint that time spent in a given state is at least 1 (INF_BOUND ≥ 1). Like for the type MARKOV, observation (or state-dependent) probabilities can be defined in order to specify a lumped process (with the restriction that each possible output can be observed in a single state).

Note that absorbing states such as state 3 ($p_{33} = 1$) are by nature Markovian. It is also possible to define nonabsorbing Markovian states such as state 2 ($0 < p_{22} < 1$). In this case, the resulting model is a hybrid Markov/semi-Markov chain.

5.7.10 type SEQUENCES

The syntactic form of the type SEQUENCES is constituted of a header giving the number and the type of variables and of the sequence. Consider the following example of univariate sequences:

```
1 VARIABLE

VARIABLE 1 : STATE

1 0 0 0 1 1 2 0 2 2 2 1 1 0 1 0 1 1 1 1 0 1 1 1 \
0 1 2 2 2 1

0 0 0 1 1 0 2 0 2 2 2 1 1 1 1 0 1 0 0 0 0 0
```

The type STATE is the generic type. The character '\' enables to continue a sequence on the following line.

Consider the following example of multivariate sequences:

```
2 VARIABLES

VARIABLE 1 : STATE
VARIABLE 2 : STATE

1 0 | 0 0 | 1 0 | 2 0 | 2 1 | 2 1 | 1 0 | 1 0 | 1 0 | 0 1 | 0 1 | 1 1 \
0 1 | 2 0 | 2 1

0 0 | 0 0 | 1 0 | 2 0 | 2 1 | 1 1 | 1 0 | 1 0 | 0 0 | 0 0
```

The character '|' enables to separate successive vectors.

Consider the following example of sequences with an explicit index parameter of type POSITION:

```
2 VARIABLES

VARIABLE 1 : POSITION
VARIABLE 2 : STATE

10 1 | 12 0 | 13 1 | 14 2 | 15 2 | 20 2 | 22 1 | 23 1 | 27 1 | 30 0 | 31 0 | 32 1 \
35 1 | 37 0 | 40 1 | 45

5 0 | 7 0 | 10 0 | 11 0 | 15 1 | 18 1 | 20 0 | 21 0 | 22 0 | 25 0 | 25
```

This explicit index parameter is given as a first variable and the other variables (at least one) should be of type `STATE` . The index values should be increasing along sequences and the sequence ends with a final index value.

The explicit index parameter of type `POSITION` can be replaced by inter-position intervals:

```
2 VARIABLES

VARIABLE 1 : POSITION_INTERVAL
VARIABLE 2 : STATE

10 1 | 2 0 | 1 1 | 1 2 | 1 2 | 5 2 | 2 1 | 1 1 | 4 1 | 3 0 | 1 0 | 1 1 \
3 1 | 2 0 | 3 1 | 5

5 0 | 2 0 | 3 0 | 1 0 | 4 1 | 3 1 | 2 0 | 1 0 | 1 0 | 3 0 | 0
```

Consider the following example of sequences with an explicit index parameter of type `TIME`:

```
2 VARIABLES

VARIABLE 1 : TIME
VARIABLE 2 : STATE

3 1 | 7 4 | 10 8 | 14 10 | 18 15 | 21 16 | 25 18 | 28 19 | 31 20 | 35 22 | 39 23 | 42 24 \
45 25 | 49 25

3 1 | 7 2 | 10 6 | 14 9 | 18 13 | 21 14 | 25 15 | 28 16 | 31 17 | 35 17
```

The only difference with the explicit index parameter of type `POSITION` is that the index values should be strictly increasing along sequences and that no final index value is required.

The explicit index parameter of type `TIME` can be replaced by time intervals:

```
2 VARIABLES

VARIABLE 1 : TIME_INTERVAL
VARIABLE 2 : STATE

3 1 | 4 4 | 3 8 | 4 10 | 4 15 | 3 16 | 4 18 | 3 19 | 3 20 | 4 22 | 4 23 | 3 24 \
3 25 | 4 25

3 1 | 4 2 | 3 6 | 4 9 | 4 13 | 3 14 | 4 15 | 3 16 | 3 17 | 4 17
```

5.7.11 type TIME_EVENTS

The syntactic form of data of type {time interval between two observation dates, number of events occurring between these two observation dates} consists in giving, in a first column, the time interval between two observation dates (length of the observation period), in a second column, the number of events occurring between these two observation dates and, in a third column, the corresponding frequency. The time interval between two observation dates should be given in increasing order and then, for each possible time interval, the number of events should be given in increasing order. This is equivalent of giving successively the frequency distribution of the number of events for each possible time interval between two observation dates, ranked in increasing order.

```
#  frequency distribution of the number of events for an observation period of length 20
20  2   1
20  3   2
20  4   4
20  5  12
20  6  14
20  7   6
20  8   2
20  9   1

# frequency distribution of the number of events for an observation period of length 30
30   3   1
30   5   2
30   6   4
30   7  12
30   8  14
30   9   6
30  10   2
30  12   1
```

5.7.12 type TOPS

Consider the following example:

```
2 VARIABLES

VARIABLE 1 : POSITION
VARIABLE 2 : NB_INTERNODE

10 5 | 12 5 | 13 6 | 13 8 | 15 7 | 20 10 | 22 11 | 23 11 | 27 15 | 30 16 | 31 15 | 32 17 \
35 16 | 37 18 | 40 19 | 45

5 2 | 7 4 | 10 5 | 11 6 | 15 7 | 18 8 | 20 9 | 21 11 | 22 11 | 25 12 | 25
```

The syntactic form of the type `TOPS` is a variant of the syntactic form of the type `SEQUENCES`. 'Tops' can be seen as sequences with an explicit index parameter of type `POSITION`. This index parameter represents the position of successive offspring shoots along the parent shoot and a final index value gives the number of internodes of the parent shoot. The second variable of type `NB_INTERNODE` gives the number of internodes of the offspring shoots.

The explicit index parameter of type `POSITION` can be replaced by inter-position intervals:

```
2 VARIABLES

VARIABLE 1 : POSITION_INTERVAL
VARIABLE 2 : NB_INTERNODE

10 5 | 2 5 | 1 6 | 0 8 | 2 7 | 5 10 | 2 11 | 1 11 | 4 15 | 3 16 | 1 15 | 1 17 \
3 16 | 2 18 | 3 19 | 5

5 2 | 2 4 | 3 5 | 1 6 | 4 7 | 3 8 | 2 9 | 1 11 | 1 11 | 3 12 | 0
```

5.7.13 type TOP_PARAMETERS

A model of 'tops' is defined by three parameters, namely the growth probability of the parent shoot, the growth probability of the offspring shoots (both in the sense of Bernoulli processes) and the growth rhythm ratio offspring shoots / parent shoot. Consider the following example:

```
TOP_PARAMETERS

PROBABILITY : 0.7
AXILLARY_PROBABILITY : 0.6
RHYTHM_RATIO : 0.8
```

The following constraints apply to the parameters:

$0.05 \leq$ `PROBABILITY` ≤ 1
$0.05 \leq$ `AXILLARY_PROBABILITY` ≤ 1
$1/3 \leq$ `RHYTHM_RATIO` ≤ 3

5.7.14 type VECTOR_DISTANCE

The parameters of definition of a distance between vectors are the number of variables, the distance type (`ABSOLUTE_VALUE` or `QUADRATIC`) if there is more than one variable, the variable types (`NUMERIC`, `SYMBOLIC`, `ORDINAL` or `CIRCULAR`) and eventually the weights of the variables (default behaviour: the variables have the same weight), and in the symbolic case, explicit distances between symbols (default behaviour: 0 / 1 for mismatch / match). Consider the following example:

```
4 VARIABLES

DISTANCE : ABSOLUTE_VALUE

VARIABLE 1 : NUMERIC  WEIGHT : 0.4

VARIABLE 2 : ORDINAL  WEIGHT : 0.2

VARIABLE 3 : SYMBOLIC  WEIGHT : 0.2

4 SYMBOLS

0
1  0
1  1  0
2  2  2  0

VARIABLE 4 : CIRCULAR  WEIGHT : 0.2
PERIOD : 12
```

5.7.15 type VECTORS

In the syntactic form of the type VECTORS, each row corresponds to an individual and each column corresponds to a variable. Consider the following example:

```
0   1   20
1   2   96
0   4  152
1  12  218
0  14   42
0   6   57
1   3  111
1   2  172
1   1  154
0   2   31
1   1  139
```

Appendice A AML FILE EXAMPLE

➢ Wij.aml

```
################################################
#
#     WIJICK Basis (101 Hybrid Trees)
#
#     Database: E. Costes  (costes@ensam.inra.fr)
#
################################################

amapmod_dir= "../../.."

mtg = MTG(amapmod_dir + "databases/MTGFiles/AppleTree/wij.mtg")

mtg10 = MTG(amapmod_dir + "databases/MTGFiles/AppleTree/wij10.mtg")

plants = VtxList(Scale->1)
shoots = VtxList(Scale->2)
gus = VtxList(Scale->3)


plant(_i) = (Foreach _v In plants: Select(_v, Index(_v)==_i))@1

###############################################
#
#  Exploration functions
#
###############################################

# 1. sampling function
# number of internodes per entity

innb(_x) = Size(Components(_x, Scale->4))

# 2. Sample
# set of growth units for a given order and year

gus_o(_order, _year) = Foreach _x In gus : \
  Select(_x, Order(_x) == _order And Index(Complex(_x)) == _year)

# Iteration : construction of the sample of values

sample_innb_gu(_order, _index) = Foreach _x In gus_o(_order, _index) :\
  innb(_x)

(#
gus_oc(_order, _year, _cycle) = Foreach _x In gus : \
  Select(_x, Order(_x) == _order And \
             Index(Complex(_x)) == _year And Index(_x) == _cycle)

sample_innb_guc(_order, _index, _cycle) = \
  Foreach _x In gus_oc(_order, _index, _cycle) : innb(_x)

h90 = Histogram(sample_innb_guc(0,90,1))
h91 = Histogram(sample_innb_guc(0,91,1))
h92 = Histogram(sample_innb_guc(0,92,1))
```

```
Plot(h90,h91,h92)
#)

as_oy(_order, _year) = Foreach _x In shoots : \
  Select(_x, Order(_x) == _order And Index(_x) == _year)

sample_innb_as(_order, _index) = Foreach _x In as_oy(_order, _index) : \
  innb(_x)

h90 = Histogram(sample_innb_as(0,90))
h91 = Histogram(sample_innb_as(0,91))
h92 = Histogram(sample_innb_as(0,92))

h290 = Cluster(h90,"Step",2)
h291 = Cluster(h91,"Step",2)
h292 = Cluster(h92,"Step",2)

h91_1 = Histogram(sample_innb_as(1,91))
h91_2 = Histogram(sample_innb_as(2,91))

h291_1 = Cluster(h91_1,"Step",2)
h291_2 = Cluster(h91_2,"Step",2)

h92_1 = Histogram(sample_innb_as(1,92))
h92_2 = Histogram(sample_innb_as(2,92))

h292_1 = Cluster(h92_1,"Step",2)
h292_2 = Cluster(h92_2,"Step",2)

##############################
#
#  Histogram extraction
#
##############################

# Histogram of the number of internodes for gus order 0 and year 90

histo_nb_gu0_90 = Histogram(sample_innb_gu(0,90))
Plot(histo_nb_gu0_90)

h909192 = Foreach _i In [90:92] : Histogram(sample_innb_gu(0,_i))

Plot(h909192@1, h909192@2, h909192@3)

(# Detail des histogrammes fonction des ordres
Foreach _a In [90:92] : Plot(Histogram(sample_innb_gu(0,_a)))
Foreach _a In [90:92] : Plot(Histogram(sample_innb_gu(1,_a)))
Foreach _a In [91:92] : Plot(Histogram(sample_innb_gu(2,_a)))
#)

h90_1 = ValueSelect(h909192@1, 0,31)
h91_1 = ValueSelect(h909192@2, 0,31)
h92_1 = ValueSelect(h909192@3, 0,31)

Plot(h90_1, h91_1, h92_1)

(#
h90_2 = ValueSelect(h909192@1, 32,100)
h91_2 = ValueSelect(h909192@2, 32,100)
h92_2 = ValueSelect(h909192@3, 32,100)
```

```
Plot(h90_2, h91_2, h92_2)
#)

# Are the two distributions identical ?
ComparisonTest(W, h90_1, h91_1)

# Mixtures

mixt1 = Estimate(histo_nb_gu0_90, "MIXTURE", "NB", "NB")
Plot(mixt1)

############################
#
#  Extraction of sequences
#
############################

is_branching(_x) = If Size(Sons(_x, EdgeType->'+')) != 0 Then 1 Else 0
seq_branch(_plant) = Foreach _x In Trunk(_plant, Scale->4) : \
  is_branching(_x)

seq_wij = Sequences(Foreach _x In plants : seq_branch(_x))
# Plot(seq_wij, "Intensity")


######################################
#
#  Geometric representation
#
######################################


p10 = plant(10)
dr1 = DressingData(amapmod_dir + "databases/MTGFiles/AppleTree/wij.drf")

#####################################################
# 1. Coarse representation (scale of annual shoots)
#####################################################

f1 = PlantFrame(p10, Scale -> 2, DressingData-> dr1)
Plot(f1)


#################################################
# 2. Measured diameters can be taken into account
#################################################

botdiam(_x) = Feature(_x,"diabase")      # value on gus in 1/100 mm
topdiam(_x) = Feature(_x,"diasom")  # value on gus in 1/100 mm
length(_x) = Feature(_x,"longueur")      # value on gus in 1/10 mm
nbfruit(_x) = Feature(_x, "nbfruit")     # value on gus

botdiam_as(_x) = botdiam(Components(_x)@1)
topdiam_as(_x) = topdiam(Components(_x)@1)
len_as(_x) = length(Components(_x)@1)

botdiam_gu(_u) =  If Class(_u) == 'I' \
             Then 13 \
             Else If Father(_u) != Undef \
                  Then If Class(Father(_u)) == 'I' \
```

```
                    Then 4 \
                    Else Undef \
               Else Undef
topdiam_gu(_u) =  If Class(_u) == 'I' \
            Then 13 \
            Else If Size(Sons(_u,EdgeType->'<')) != 0 \
                 Then If (Class(Sons(_u,EdgeType->'<')@1)=='I') \
                    Then 4 \
                    Else Undef \
               Else Undef
len_gu(_u) =  If Class(_u) == 'I' Then 4 Else Undef

botdiam_in(_e) =  If Class(Complex(_e)) == 'I' \
            Then 2000 \
            Else If Father(_e) != Undef \
                 Then If Class(Complex(Father(_e))) == 'I' \
                    Then 400 \
                    Else Undef \
               Else Undef
topdiam_in(_e) =  If Class(Complex(_e)) == 'I' \
            Then 2000 \
            Else If Size(Sons(_e,EdgeType->'<')) != 0 \
                 Then If (Class(Complex(Sons(_e,EdgeType->'<')@1))=='I') \
                    Then 400 \
                    Else Undef \
               Else Undef
len_in(_e) =  If Class(Complex(_e)) == 'I' Then 400 Else Undef


# Multiscale functions

bottom_diameter(_x) = \
  Switch Scale(_x)  \
  Case 2 : botdiam_as(_x) \
  Case 3 : botdiam_gu(_x) \
  Case 4 : botdiam_in(_x) \
  Default : Undef

top_diameter(_x) = \
  Switch Scale(_x) \
  Case 2 : topdiam_as(_x) \
  Case 3 : topdiam_gu(_x) \
  Case 4 : topdiam_in(_x) \
  Default : Undef

len(_x) = \
  Switch Scale(_x) \
  Case 2 : len_as(_x) \
  Case 3 : len_gu(_x) \
  Case 4 : len_in(_x) \
  Default : Undef

alpha(_x) = \
  Switch Scale(_x) \
  Case 3 : (If Class(Father(_x)) == 'I' Then 5 Else Undef) \
  Default : Undef


# Geometry (top and bot diam, length) is only measured at the scale of
shoots
```

```
f2 = PlantFrame(p10, Scale->2, DressingData-> dr1, Length->len, \
                TopDiameter->top_diameter, BottomDiameter ->
bottom_diameter)
f3 = PlantFrame(p10, Scale->3, DressingData-> dr1, Length->len, \
                TopDiameter->top_diameter, BottomDiameter->bottom_diameter)
f4 = PlantFrame(p10, Scale->4, DressingData-> dr1, Length->len, \
                TopDiameter->top_diameter, BottomDiameter->bottom_diameter)

Plot(f2) # annual shoots
Plot(f3) # growth units
Plot(f4) # internodes

########################################
#
#  Predicates
#
########################################

has_geom(_x) = bottom_diameter(_x) != Undef Or top_diameter(_x) != \
  Undef Or len(_x) != Undef
has_allgeom(_x) = bottom_diameter(_x) != Undef And top_diameter(_x) != \
  Undef And len(_x) != Undef
has_fruits(_x) = nbfruit(_x) != Undef

########################################
#
#  Coloring plants
#
########################################

color_order(_v) = \
  Switch Order(_v) \
  Case 0 : Green \
  Case 1 : Red \
  Case 2 : Blue \
  Case 3 : Yellow \
  Case 4 : Violet \
  Default : White

color_fruits(_x) = \
  If Class(Complex(_x, Scale->3)) == 'I' \
  Then \
    Switch Index(Complex(_x, Scale->2)) \
    Case 92 : LightBlue \
    Case 93 : Violet \
    Default : Black \
  Else Green

color_index(_x) =  \
  Switch Index(Complex(_x, Scale->2)) \
  Case 90 : Black \
  Case 91 : Green \
  Case 92 : Red \
  Case 93 : Violet \
  Default : Yellow

color_bourse_index(_x) = \
  If  Class(Complex(_x, Scale->3)) == 'I' \
  Then Switch Index(Complex(_x, Scale->2)) \
       Case 90 : LightBlue\
       Case 91 : Green \
```

```
       Case 92 : Red \
       Case 93 : Violet \
       Default : Yellow \
  Else Black

color_class(_x) = \
  If Class(Complex(_x, Scale->3)) == 'I' \
  Then Violet \
  Else \
    Switch Class(_x) \
    Case 'B' : Black\
    Case 'C' : Green \
    Case 'E' : Red \
    Default : Yellow

color_bourse_order(_v) = \
  If Class(Complex(_v, Scale->3)) == 'I' \
  Then Switch Order(_v) \
       Case 0 : Green \
       Case 1 : Red \
       Case 2 : Blue \
       Case 3 : Violet \
       Default : Yellow \
  Else Black

color_height(_x) =  \
  If Scale(_x) >2 \
  Then Switch Height(Complex(_x, Scale->3)) \
       Case 0 : Yellow\
       Case 1 : Violet \
       Case 2 : LightBlue \
       Case 3 : Red \
       Case 4 : Green \
       Case 5 : Blue \
       Default : Black \
  Else Black

color_measured_shoots(_x) = \
  If has_fruits(Complex(_x, Scale->3))\
  Then Red \
  Else \
    If has_geom(Complex(_x, Scale->2)) \
    Then Green \
    Else Black

Plot(f4, Color -> color_fruits)
Plot(f4, Color -> color_order)
Plot(f4, Color -> color_index)
Plot(f4, Color -> color_class)
Plot(f4, Color -> color_height)
Plot(f4, Color -> color_bourse_order)
Plot(f4, Color -> color_bourse_index)
Plot(f4, Color -> color_measured_shoots)

###################################
#
#  Displaying a set of plants
#
###################################
```

```
i_in_ancestors(_x) = Size(Foreach _v In Ancestors(_x) : \
  Select(_v, _v != _x And Class(_v) == 'I')) != 0
ii(_x) = If Class(Complex(_x)) == 'I' And \
  Index(Complex(_x, Scale -> 2)) == 93 \
         Then If i_in_ancestors(Complex(_x)) Then True Else False \
         Else False
i92(_x) = Class(Complex(_x)) == 'I' And \
  Index(Complex(_x, Scale -> 2)) == 92
i93(_x) = Class(Complex(_x)) == 'I' And \
  Index(Complex(_x, Scale -> 2)) == 93
color_II(_x) = If ii(_x) \
               Then Red \
               Else If i92(_x) \
                    Then Yellow \
                    Else If i93(_x) \
                         Then Violet \
                         Else Green

color_bourse_regul(_x) = \
  If  Class(Complex(_x, Scale->3)) == 'I' \
  Then If ii(_x) \
       Then Red \
       Else If i92(_x) \
            Then Yellow \
            Else If i93(_x) \
                 Then Violet \
                 Else Black \
  Else Green

# Virtual orchard : 3D Visualization of an orchard of apple trees

inflo_ram(_u) = (Size(Foreach _v In Descendants(_u) : \
  Select(_u, _v != _u And Class(_v) == 'I')) != 0)

I92 = Foreach _u In gus : Select(_u, Index(Complex(_u)) == 92 \
  And Class(_u) == 'I')
I93 = Foreach _u In gus : Select(_u, Index(Complex(_u)) == 93 \
  And Class(_u) == 'I')
pI92 = Sort(ToSet(Foreach _u In I92 : Select(Complex(_u,Scale->1), True)))
pI93 = Sort(ToSet(Foreach _u In I93 : Select(Complex(_u,Scale->1), True)))
pII = Sort(ToSet(Foreach _u In I92 : \
  Select(Complex(_u,Scale->1), inflo_ram(_u) == True)))

f_pII = PlantFrame(pII, Scale->4, DressingData-> dr1, Length->len, \
  TopDiameter->top_diameter, BottomDiameter->bottom_diameter, \
  TrunkDist->10000)

# particular coloring of fruit locations 92 (yellow), 93 fruit (violet)
# location of consecutive fruiting (92 and 93) (red)

Plot(f4, Color -> color_bourse_regul)
Plot(f_pII, Color -> color_bourse_regul)
```

➢ Wij.drf

```
# Dressing file for apple trees
SMBPath = ../../SMBFiles

SMBModel internode = nentn105
SMBModel apple = pommecyl

BranchPattern applebranch  = wij.crv

Class U = internode
Class I = internode
Class B = internode

Class T = internode
Class Y = internode
Class P = apple

# For virtual symbols

FruitClass = P
LeafClass = Z

LeafAlpha = 80
LeafBeta  = 144
LeafLength = 50
LeafTopDiameter = 50
LeafBottomDiameter = 50

# For branch forms

Form 1 = apple2 # axes 2 ortho

# For a vertical Trunk: DefaultTrunkCategory = -1
DefaultTrunkCategory = -1

Phyllotaxy = 144

DiameterUnit = 1000
LengthUnit = 100

DefaultDistance = 10000
NbPlantsPerLine = 5
```

➢ Wij.crv

```
7
apple1
4
0 0 0
0 6 3
0 14 1
0 24 6
apple12
4
0 0 0
0 3 10
0 -5 20
0 0 30
apple2
3
0 0 0
0 7 3
0 8 20
apple22
5
0 0 0
0 0 10
0 0 20
0 20 22
0 30 25
apple222
6
0 0 0
0 0 10
0 10 20
0 20 30
0 30 -10
0 40 -20
apple23
4
0 0 0
0 7 3
0 14 0
0 20 6
apple3
3
0 0 0
0 7 3
0 8 20
```

➢ Wij10.mtg

```
################################
# Pommiers wijick x Baujade
# Bordeaux
# plante 10
################################
CODE: FORM-A

CLASSES:
SYMBOL      SCALE DECOMPOSITION     INDEXATION  DEFINITION
$     0     FREE  FREE  IMPLICIT
P     1     CONNECTED   FREE  EXPLICIT
A     2     <-LINEAR    FREE  EXPLICIT
U     3     <-LINEAR    FREE  EXPLICIT
I     3     <-LINEAR    FREE  EXPLICIT
E     4     FREE        FREE  EXPLICIT
C     4     FREE        FREE  EXPLICIT
B     4     FREE        FREE  EXPLICIT

DESCRIPTION :
LEFT  RIGHT RELTYPE     MAX
A     A     <     1
A     A     +     ?
U     U,I   <     1
U     U,I   +     ?
I     U,I   +     10
E     E,C,B <     1
C     E,C,B <     1
B     E,C,B <     1
B     E,C,B +     1

FEATURES:
NAME  TYPE

diabase     INT
diasom      INT
longueur    INT
nbfruit     INT
rem   STRING

MTG:
ENTITY-CODE                          diabase  diasom  longueur   nbfruit   rem
/P10
 /A90/U1                                      900    1880       6700
 ^/E1<C2<C3<B4<<B7<C8<C9<B10<B11<B12
   +A91/U1/E1<C2<C3<B4
 ^<B13<<B15
   +A91/U1/E1<B2
 ^<B16
   +A91/U1/E1<B2
 ^<B17
   +A91/U1/E1<B3
 ^<B18
   +A91/U1/E1<B4
 ^<B19
   +A91/U1/E1<B5
 ^<B20
   +A91/U1/E1<B6
 ^<B21
  +A91/U1/E1<A92/U1                           500    100        540
```

Appendice B BIBLIOGRAPHY

1. 1998. *Architecture et modélisation en arboriculture fruitière, Actes du 11ème colloque sur les recherches fruitières*. INRA-Ctifl, Montpellier, France.

2. **Barthélémy, D.**, 1991. Levels of organization and repetition phenomena in seed plants. *Acta Biotheoretica*, 39: 309-323.

3. **Bouchon, J., de Reffye, P. et Barthélémy, D.** (Eds), 1997. *Modélisation et simulation de l'architecture des végétaux*. Science Update. INRA Editions, Paris, France, 435 pp.

4. **Caraglio, Y. et Dabadie, P.**, 1989. Le peuplier. Quelques aspects de son architecture. In: " Architecture, structure, mécanique de l'arbre " Premier séminaire interne, Montpellier (FRA) 01/89, pp. 94-107.

5. **Cilas, C., Guédon, Y., Montagnon, C. et de Reffye, P.**, 1998. Analyse du suivi de croissance d'un cultivar de caférier (*Coffea canephora* Pierre) en Côte d'Ivoire. In: *Architecture et modélisation en arboriculture fruitière*, 11ème colloque sur les recherches fruitières, Montpellier, France 5-6/03/1998, INRA-Ctifl, pp. 45-55.

6. **Costes, E. et Guedon, Y.**, 1997. Modelling the sylleptic branching on one-year-old trunks of apple cultivars. *Journal of the American Society for Horticultural Science*, 122(1): 53-62.

7. **Costes, E., Sinoquet, H., Godin, C. et Kelner, J.J.**, 1999. 3D digitizing based on tree topology : application to study th variability of apple quality within the canopy. *Acta Horticulturae*, 499: 271-280.

8. **de Reffye, P.**, 1982. Modèle Mathématique aléatoire et simulation de la croissance et de l'architecture du caféier Robusta. 3ème Partie. Etude de la ramification sylleptique des rameaux primaires et de la ramification proleptique des rameaux secondaires. *Café Cacao Thé*, 26(2): 77-96.

9. **de Reffye, P., Dinouard, P. et Barthélémy, D.**, 1991. Modélisation et simulation de l'architecture de l'Orme du Japon *Zelkova serrata* (Thunb.) Makino (Ulmaceae): la notion d'axe de référence. In: 2ème Colloque International sur l'Arbre, Montpellier (FRA) 9-14/09/90. *Naturalia Monspeliensa*, Vol. hors-série, pp. 251-266.

10. **de Reffye, P., Edelin, C., Françon, J., Jaeger, M. et Puech, C.**, 1988. Plant models faithful to botanical structure and development. In: SIGGRAPH'88, Atlanta (USA) 1-15/08/88. C.G.S.C. Proceedings, Vol. 22, pp. 151-158.

11. **de Reffye, P. *et al.***, 1995. A model simulating above- and below- ground tree architecture with agroforestry applications. In: *Agroforestry : Science; Policy and Practice*, 20th IUFRO World Congress, F.L. Sinclair (Ed.), Tampere, Finlande 06-12/08/1995. *Agroforestry Systems*, Vol. 30, pp. 175-197.

12. **Dempster, A.P., Laird, N.M. et Rubin, D.B.**, 1977. Maximum likelihood from incomplete data via the EM algorithm (with discussion). *Journal of the Royal Statistical Society, Series B*, 39: 1-38.

13. **Ferraro, P. et Godin, C.**, 1998. Un algorithme de comparaison d'arborescences non ordonnées appliqué à la comparaison de la structure topologique des plantes. In: SFC'98, Recueil des Actes, Montpellier, France 21-23/09/1998, Agro Monpellier, pp. 77-81.

14. **Ferraro, P. et Godin, C.**, 1999. A distance measure between plant architectures. *Annals of Forest Sciences*, 57(5/6): 445-461.

15. **Fisher, J.B. et Weeks, C.L.**, 1985. Tree architecture of *Neea* (Nyctaginaceae) : geometry and simulation of branches and the presence of two different models. *Bulletin du Muséum National d'Histoire Naturelle, section B Adansonia*, 7(4): 385-401.

16. **Fitter, A.H.**, 1987. An architectural approach to the comparative ecology of plant root systems. *New Phytologist*, 106(Suppl.): 61-77.

17. **Fournier, D., Guédon, Y. et Costes, E.**, 1998. A comparison of different fruiting shoots of peach trees. In: IVth International Peach Symposium, Bordeaux, France , Vol. 465(2), pp. 557-565.

18. **Frijters, D. et Lindenmayer, A.**, 1976. Developmental descriptions of branching patterns with paraclidial relationships. In: Formal Languages, Automata and Development, G. Rozenberg et A. Lindenmayer (Eds), Noordwijkerhout, The Netherlands , North-Holland Publishing Company, pp. 57-73.

19. **Godin, C., Bellouti, S. et Costes, E.**, 1996. Restitution virtuelle de plantes réelles : un nouvel outil pour l'aide à l'analyse de données botaniques et agronomiques. In: *L'interface des mondes réels et virtuels*, 5èmes Journées Internationales Informatiques, Montpellier, France 22-24/05/96, pp. 369-378.

20. **Godin, C. et Caraglio, Y.**, 1998. A multiscale model of plant topological structures. *Journal of Theoretical Biology*, 191: 1-46.

21. **Godin, C. et Costes, E.**, 1996. How to get representations of real plants in computers for exploring their botanical organisation. In: International Symposium on Modelling in Fruit Trees and Orchard Management, Avignon (FRA) 4-8/09/95, ISHS. *Acta Horticulturae*, Vol. 416, pp. 45-52.

22. **Godin, C., Costes, E. et Caraglio, Y.**, 1997. Exploring plant topology structure with the AMAPmod software : an outline. *Silva Fennica*, 31(3): 355-366.

23. **Godin, C., Costes, E. et Sinoquet, H.**, 1999. A method for describing plant architecture which integrates topology and geometry. *Annals of Botany*, 84(3): 343-357.

24. **Godin, C., Guédon, Y. et Costes, E.**, 1999. Exploration of plant architecture databases with the AMAPmod software illustrated on an apple-tree bybird family. *Agronomie*, 19(3/4): 163-184.

25. **Godin, C., Guédon, Y., Costes, E. et Caraglio, Y.**, 1997. Measuring and analyzing plants with the AMAPmod software. *In: Plants to ecosystems - Advances in Computational Life Sciences 2nd International Symposium on Computer Challenges in Life Science*. M.T. Michalewicz (Ed.). CISRO Australia, Melbourne, Australie, pp. 53-84.

26. **Guédon, Y.**, 1998. Analyzing nonstationary discrete sequences using hidden semi-Markov chains. Document de travail du programme Modélisation des plantes, 5-98. CIRAD, Montpellier, France, 41 pp.

27. **Guédon, Y.**, 1998. Hidden semi-Markov chains: a new tool for analyzing nonstationary discrete sequences. In: 2nd International Symposium on Semi-Markov models: theory and applications, J. Janssen et N. Limnios (Eds), Compiègne, France 09-11/12/1998, Université de Technologie de Compiègne, pp. 1-7.

28. **Guédon, Y., Barthélémy, D. et Caraglio, Y.**, 1999. Analyzing spatial structures in forests tree architectures. In: Salamandra Ed. Empirical and process-based models for forest tree and stand growth simulation, A. Amaro et M. Tomé (Eds), Oeiras, Portugal 21-27/09/1997, Salamandra Ed., pp. 23-42.

29. **Guédon, Y. et Costes, E.**, 1999. A statistical approach for analyszing sequences in fruit tree architecture. *Acta Horticulturae*, 499: 281-288.

30. **Hallé, F. et Oldeman, R.A.A.**, 1970. *Essai sur l'architecture et la dynamique de croissance des arbres tropicaux.* Monographie de Botanique et de Biologie Végétale, Vol. 6. Masson, Paris, 176 pp.

31. **Hallé, F., Oldeman, R.A.A. et Tomlinson, P.B.**, 1978. *Tropical trees and forests. An architectural analysis*. Springer-Verlag, New-York.

32. **Hanan, J. et Room, P.**, 1997. Practical aspects of plant research. *In: Plants to ecosystems - Advances in Computational Life Sciences 2nd International Symposium on Computer Challenges in Life Science*. M.T. Michalewicz (Ed.). CISRO Australia, Melbourne, Australie, pp. 28-43.

33. **Harper, J.L., Rosen, B.R. et White, J.**, 1986. *The growth and form of modular organisms*. The Royal Society, London.

34. **Honda, H.**, 1971. Description of the form of trees by the parameters of the tree-like body : Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 31: 331-338.

35. **Honda, H., Tommlinson, P. et Fisher, J.B.**, 1982. Two geometrical models of branching of tropical trees. *Annals of Botany*, 49: 1-12.

36. **Jackson, J.E. et Palmer, J.W.**, 1981. Light distribution in discontinuous canopies: calculation of leaf areas and canopy volumes above defined irradiance contours for use in productivity modelling. *Annals of Botany*, 47: 561-565.

37. **Jaeger, M. et de Reffye, P.**, 1992. Basic concepts of computer simulation of plant growth. In: The 1990 Mahabaleshwar Seminar on Modern Biology, Mahabaleshwar (IND) . *Journal of Biosciences*, Vol. 17, pp. 275-291.

38. **Mitchell, K.J.**, 1975. Dynamics and simulated yield of Douglas-fir. *Forest Science*, 21(4): 1-39.

39. **Prusinkiewicz, P. et Lindenmayer, A.**, 1990. *The algorithmic beauty of plants*. Springer Verlag.

40. **Prusinkiewicz, P.W., Remphrey, W.R., Davidson, C.G. et Hammel, M.S.**, 1994. Modeling the architecture of expanding *Fraxinus pennsylvanica* shoots unsing L-systems. *Canadian Journal of Botany*, 72: 701-714.

41. **Rapidel, B.**, 1995. *Etude expérimentale et simulation des transferts hydriques dans les plantes individuelles. Application au caféier (Coffea arabica L.)*. Thèse Doctorat, Université des Sciences et Techniques du Languedoc (USTL), Montpellier, France, 246 pp.

42. **Remphrey, W.R., Neal, B.R. et Steeves, T.A.**, 1983. The morphology and growth of Arctostaphylos uva-ursi (bearberry): an architectural model simulated colonizing growth. *Canadian Journal of Botany*, 61: 2451-2458.

43. **Rey, H., Godin, C. et Guedon, Y.**, 1997. Vers une représentation formelle des plantes. *In: Modélisation et Simulation de l'Architecture des Végétaux*. J. Bouchon, P. de Reffye et D. Barthélémy (Eds). *Science Update*. INRA Editions, Paris, France, pp. 139-171.

44. **Room, P. et Hanan, J.**, 1996. Virtual plants: new perspectives for ecologists, pathologists and agricultural scientists. *Trends in Plant Science Update*, 1(1): 33-38.

45. **Ross, J.K.**, 1981. *The radiation regim and the architecture of plant stands*. Junk W. Pubs., The Hague, The Netherlands.

46. **Sabatier, S., Ducousso, I., Guédon, Y., Barthélémy, D. et Germain, E.**, 1998. Structure de scions d'un an de Noyer commun, *Juglans regia* L., variété Lara greffés sur trois porte-greffe (*Juglans nigra*, *J. regia, J. nigra* x *J. regia*). In: *Architecture et modélisation en arboriculture fruitière*, 11ème colloque sur les recherches fruitières, Montpellier, France 5-6/03/1998, INRA-Ctifl, pp. 75-84.

47. **Sinoquet, H., Adam, B., Rivet, P. et Godin, C.**, 1998. Interactions between light and plant architecture in an agroforestry walnut tree. *Agroforestry Forum*, 8(2): 37-40.

48. **Sinoquet, H., Godin, C. et Costes, E.**, 1998. Mesure de l'architecture par digitalisation 3D. In: *Numérisation 3D, Design et digitalisation, Création industrielle et artistique*, Actes du Congrès, Paris, France 27-28/05/1998.

49. **Sinoquet, H., Rivet, P. et Godin, C.**, 1997. Assessment of the three-dimensional architecture of walnut trees using digitising. *Silva Fennica*, 31(3): 265-273.

50. **Sinoquet, H., Thanisawanyangkura, S., Mabrouk, H. et Kasemsap, P.**, 1998. Characterisation of the light environment in canopies using 3D digitising and image processing. *Annals of Botany*, 82: 203-212.

51. **Zhang, K.**, 1993. A new editing based distance between unordered labeled trees. In: *Combinatorial Pattern Matching* CPM 93, 4th Annual Symposium, Padova, Italie 2-4/06/1993.

Appendice C COPYRIGHT

---------- GPL ----------

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it.By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the

original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

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

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

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

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

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

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

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

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

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

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

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

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

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or

binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any

other system and a licensee cannot impose that choice. / This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH

ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) 19yy <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; see the file COPYING. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright © 19yy name of author
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
> This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other

than `show w' and `show c'; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

INDEX