The gmdl Modeling and Analysis System

Daniel Gillblad (*Editor*)

D. Gillblad *dgi@sics.se*
A. Holst *aho@sics.se*
P. Kreuger *piak@sics.se*
B. Levin *blevin@sics.se*

SICS Technical Report T2004:17, ISSN 110-3154, ISRN:SICS-T–2004/17-SE

29 November 2004

SUMMARY

This document provides a defining description of the gmdl modeling and analysis environment. gmdl is an extension to and library for the programming language Scheme, a dialect of the Lisp programming language. gmdl was designed to provide powerful data analysis, modeling and visualization with simple, clear semantics and easy to use, well defined syntactic conventions. It also provides an extensive set of necessary functionality for general data analysis and modeling tasks, including various statistical measures, models, and data exploration and adaptation functions, as well as mathematical and numerical functionality.

The introduction offers a brief history of the system and of this document.

The first two chapters give an overview of the fundamental ideas of the system and describe the basic modeling and analysis work flow. They also provide a description of the basic types, the syntactic conventions used and describe the most essential gmdl functionality.

Chapter 3 describes basic procedures included in gmdl that are not found in standard Scheme.

Chapter 4 describes how gmdl handles data, including data formats, data types and filters.

Chapter 5 describes gmdl functions, models and distributions, including syntax conventions and built-in procedures.

Chapter 6 describes the gmdl graphics and plotting system, ranging from low-level graphics primitives to data plots.

Chapters 7 and 8 describes the built-in mathematical, numerical and matrix functions.

Some examples of the use of the system follows the system description chapters.

The document concludes with a list of references and an alphabetic index.

CONTENTS

INTRODUCTION

A data analysis and modeling system should be designed to fulfill a number of criteria, sometimes in conflict with each other. For example, we want to provide both a simple interface as well as being able to handle a large variety of problems. We also want the system to be extensible and adaptable, while maintaining a simple model abstraction so that a problem can be described in a consistent manner. By removing weaknesses and restrictions of the system itself, in the way models, data and other functionality are being described, we can form an efficient and practical modeling environment that will support most data analysis tasks.

gmdl uses a relatively small number of primitives and procedures to provide a flexible data analysis and modeling environment. By relying on general abstractions of models and data, it is possible to reduce the functionality to a rather small set of standard procedures. This both reduces the initial learning threshold and makes it easier for a user to learn new extensions to the system.

A general data analysis system must be fully programmable by the user. Almost all data analysis tasks differ slightly from each other, and while some subtleties might be possible to handle in a system that cannot be fully programmed or extended, it is bound to one day run into a task that cannot be handled completely within the system.

gmdl is based on the Scheme programming language, which is both simple and expressive and is flexible enough to support most major programming paradigms. Unlike many data analysis systems in use today, this means that gmdl provides a sensible programming environment. gmdl is also intended to be interactive to as large an extent as possible. Data analysis is exploratory by nature, and this should be handled and encouraged by the system. This might sometimes come into conflict with the need for efficiency in large calculations, but this conflict can usually be avoided in the system design.

Redundant functionality has been avoided as far as possible in the design of gmdl. This will hopefully make the system more intuitive as a whole, but might deter some first time users since basic functionality might actually seem to be missing. With a basic understanding of the gmdl system and the usual work flow, this will hopefully not be a problem.

Background

The gmdl system began as an easy-to-use interface to a variety of software libraries, most notably the MDL library by Daniel Gillblad and the plotting and graphics libraries by Anders Holst. It then developed into a more complete modeling and data analysis environment, with support for fast numerical calculations, interactive data analysis and plotting while maintaining full high-level programmability in Scheme. The main focus in the beginning was on learning systems and process analysis, but as the scope became wider the system quickly extended to support more general numerical calculations by introducing new syntactic forms and types, as well as providing extensive support for data pre-processing.

The data types and functionality provided by the system was by no means stable in the early versions. This has become better with time as the system matured and the need for changes to the basic structure disappeared. Still, it can not be regarded as completely stable, but rather a work in progress. As gmdl might see increasing use by people not directly connected to the development process, other views and suggestions might lead to changes both in the basic structure of the system as well as the introduction of additional functionalities that has a large impact on the usage.

This document is intended for the entire gmdl user community, and permission to copy it in whole or in part is granted without fee. Implementors of extensions to gmdl is encouraged to use this manual as part of their own documentation, but also to use the syntax and interfaces defined here for their own extensions.

Acknowledgments

We would like to thank Anders Lansner and Douglas Wikstrom for their help on discussing and developing the gmdl system.

We also thank everyone involved in developing the LISP [3] and Scheme [2] programming languages, as well as everyone involved in the Guile project for their tremendous work on creating a simple yet powerful programming language and a portable and extensible implementation thereof.

The Industrial Applications and Methods Laboratory at Swedish Institute of Computer Science and the Numerical Analysis and Computing Science department at the Royal Institute of Technology in Stockholm, Sweden, both supported the preparation of this report.

DESCRIPTION OF THE GMDL SYSTEM

1. The gmdl system

The gmdl system intends to be an easy-to-use and flexible data modeling and analysis system. As a side effect, it also provides rather extensive facilities for implementation of fast numerical procedures. This document describes the standard functionality and basic concepts of the system. It is not intended as a practical introduction to working with gmdl or a tutorial, but rather as a definition and reference manual. Still, it is the hope and intention of the authors that it is easy to read and understand, and that it provides enough information for the beginner to start working with the system immediately after reading the text.

1.1. Modeling and analysis with gmdl

Modeling and analysis in gmdl is based on an abstract and unified view of numerical functions, models of data and data itself. These concepts are all represented as abstract types that have a hierarchical dependency structure. For example, a model can always be used as a numerical function, and all procedures operating on a numerical function per definition also operate on models.

As a brief introduction, a model is a parameterized representation of data. As such, there are a number of functions that usually can be calculated, e.g. estimation of the model from data. Data modeling in gmdl is based on a clear separation between data and the models that operate on it. All models always interact with data or entries of a specified format. A model is intrinsically linked, not to a certain data set, but to a data format and a set of input attributes. In essence, a model can be viewed as a set of numerical functions with specified connections to a data format.

The system also uses a convention for numerical functions, to make it easier to exchange functionality between different parts of gmdl, as well as consistent ways to specify data and data formats. Data and formats are separate entities, and several data objects can be of the same format. A data format is built by a number of field formats, all describing the properties of a part of each entry in a data object.

All of these concepts and their relations, as well as a description of their functionality, will be presented in this document. gmdl also comes with a number of standard models, functions, distributions etc. whose functionality also is described, as well as provided functionality for more general numerical calculation.

1.2. Scheme and gmdl

gmdl is based on the Scheme programming language, described in e.g. [1] and [2]. The main reasons that gmdl uses Scheme as its programming language are found in its clarity, simplicity and consistency. gmdl introduces a number of primitives and conventions, but tries to remain true to the basic ideas and objectives of the language.

1.2.1. The Scheme programming language

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme.

1.2.2. Syntax

Since gmdl is based on Scheme, it employs a fully parenthesized prefix notation for programs and (other) data. By experience, we know that this might introduce minor headaches for first time users, but the problems are usually quickly overshadowed by the consistency and simplicity of the syntax.

gmdl makes absolutely no changes to the syntax specified in the Scheme standard. Every syntactically correct Scheme program is also a syntactically correct gmdl program. gmdl does, however, introduce a small number of new syntactic forms. These are essentially just "syntactic-sugar", defined to make data analysis and modeling tasks easier.

The formal syntax of Scheme, and therefore the better part of gmdl syntax, is described in [2].

1.3. Notation and terminology

1.3.1. Primitive, library, and optional features

The gmdl system must, in its most basic form, support all features that are not marked as being *optional.* Different versions of gmdl may not provide all optional features and add others, as long as they do not conflict with the standard definitions in this document. The main reason for this requirement is to (some degree) support portable code, relying on a basic set of procedures and definitions, but also to provide some backward compatibility to older versions of the system.

Some features do not necessarily have to be provided at startup. These are marked as *library*, and may or may not be available by default when gmdl is started. They must, however, be provided by the system in the form of a library or module that can be loaded into the gmdl environment when necessary. These functions constitute the *gmdl standard library.* The reason for not providing some features at startup is usually that they might consume scarce resources, most likely memory.

Note: These definitions are for many reasons, mainly the fact that gmdl is not a programming language standard but rather a systems standard, different from the definitions used in the Scheme report (?). Try not to be confused.

1.3.2. Error situations and unspecified behavior

When speaking of an error situation, this document uses the phrase "an error is signaled" to indicate that gmdl will detect and report the error. If such wording does not appear in the discussion of an error, then gmdl will not detect or report the error. This is common when procedures can proceed to calculate a consistent result, although this result for most but perhaps not all practical purposes is unusable.

It is an error for a procedure to be passed an argument that the procedure is not explicitly specified to handle, even though this might not be explicitly mentioned in this document.

If the value of an expression is said to be "unspecified," then the expression must evaluate to some object without signaling an error, but the value depends on the implementation. This situation is not common in gmdl.

1.3.3. Entry format

Chapters 3 to 7 are mainly organized into entries. Each entry describes one gmdl feature or a group of related features, where a feature is either a syntactic construct or a built-in procedure. An entry begins with one or more header lines of the form

template *category*

for required, primitive features, or

template *qualifier category*

where *qualifier* is either "library" or "optional" as defined in section 1.3.1.

If *category* is "syntax", the entry describes an expression type, and the template gives the syntax of the expression type. Components of expressions are designated by syntactic variables, which are written using angle brackets, for example, ⟨expression⟩, ⟨variable⟩. Syntactic variables should be understood to denote segments of program text; for example, ⟨expression⟩ stands for any string of characters which is a syntactically valid expression. The notation

⟨thing_1⟩ ...

indicates zero or more occurrences of a ⟨thing⟩, and

⟨thing_1⟩ ⟨thing_2⟩ ...

indicates one or more occurrences of a ⟨thing⟩.

If *category* is "procedure", then the entry describes a procedure, and the header line gives a template for a call to the procedure. Argument names in the template are *italicized*. Thus the header line

`(vector-ref` *vector k*`)` procedure

indicates that the built-in procedure `vector-ref` takes two arguments, a vector *vector* and an exact non-negative integer *k* (see below). The header lines

`(make-vector` *k*`)` procedure
`(make-vector` *k fill*`)` procedure

indicate that the `make-vector` procedure is defined to take either one or two arguments.

It is an error for an operation to be presented with an argument that it is not specified to handle. We try to follow the convention that if an argument name is also the name of a type, then that argument must be of the named type. For example, the header line for `vector-ref` given above dictates that the first argument to `vector-ref` must be a vector. The following naming conventions may also, but not by necessity, imply the following type restrictions:

obj	any object
list, $list_1$, ... $list_j$, ...	list
z, z_1, ... z_j, ...	complex number
x, x_1, ... x_j, ...	real number, continuous
y, y_1, ... y_j, ...	real number, continuous
q, q_1, ... q_j, ...	rational number
n, n_1, ... n_j, ...	integer, discrete
k, k_1, ... k_j, ...	exact non-negative integer

1.3.4. Evaluation examples

The symbol "⟹" used in program examples should be read "evaluates to." For example,

```
(* 5 8)                    ⟹  40
```

means that the expression `(* 5 8)` evaluates to the object `40`. Or, more precisely: the expression given by the sequence of characters "`(* 5 8)`" evaluates, in the initial environment, to an object that may be represented externally by the sequence of characters "`40`". See [2] for a discussion of external representations of objects.

1.3.5. Naming conventions

By convention, the names of procedures that always return a boolean value usually end in "`?`". Such procedures are called predicates.

By convention, the names of procedures that store values into previously allocated locations usually end in "`!`". Such procedures are called mutation procedures. By convention, the value returned by a mutation procedure is unspecified.

By convention, "`->`" appears within the names of procedures that take an object of one type and return an analogous object of another type. For example, `list->vector` takes a list and returns a vector whose elements are the same as those of the list.

By convention, procedures operating on a more complex type are named by the type or an operation thereof and the operation, `type-procedure` or similar. For example `vector-ref`, `data-read` and `matrix*` operate on vectors, data, and matrices respectively.

By convention, procedures for creating complex types are named `make-type`, e.g. `make-array` or `make-gaussian`.

2. Overview of gmdl

This chapter gives an introduction to basic functionality and concepts in gmdl. It also describes some conventions used in the system, as well as describing the memory models used. A general knowledge of most of the concepts described and their consequences is necessary for a good understanding of the later chapters in this document.

2.1. Basic concepts and conventions

2.1.1. The unknown value

In many data analysis and numerical tasks it is necessary to handle unknown values. gmdl provides a standard object for this purpose, written as `#?`. The unknown constant evaluates to itself, so it does not have to be quoted in programs.

```
#?                    ⟹  #?
'#?                   ⟹  #?
```

The unknown value carries no type information. There is no such thing as an unknown integer, unknown complex number etc. in gmdl, just the general unknown object. Many, but not all, standard procedures in gmdl are designed to handle unknown values in a consistent way. They are referred to as being *unknown safe.*

`(unknown? `*obj*`)` procedure
`(not-unknown? `*obj*`)` procedure

The procedure `unknown?` is used to determine whether an object is unknown or not, and returns `#t` if and only if *obj* is unknown, `#?`. Otherwise `#f` is returned. The complementary procedure `not-unknown?` returns `#t` if and only if *obj* is not unknown, otherwise returning `#f`.

`(any-unknown? `*obj*`)` procedure
`(all-unknown? `*obj*`)` procedure

The procedure `any-unknown?` returns `#t` if and only if any of the elements of the container *obj* is unknown, otherwise `#f` is returned. Similarly, `all-unknown?` returns `#t` if and only if all elements of the container are unknown. The container *obj* could be of any type depending on the implementation, but at least lists and vectors must be supported.

2.1.2. The irrelevant value

gmdl also provides a standard object for representing irrelevant values, e.g. for explicitly describing an expected argument to a procedure as irrelevant in the current context. It is written as `#-`. The irrelevant constant evaluates to itself, so it does not have to be quoted in programs.

```
#-                    ⟹  #-
'#-                   ⟹  #-
```

Like the unknown value, the irrelevant value carries no type information.

`(irrelevant? `*obj*`)` procedure
`(not-irrelevant? `*obj*`)` procedure

The procedure `irrelevant?` is used to determine whether an object is irrelevant or not, and returns `#t` if and only if *obj* is irrelevant, `#-`. Otherwise `#f` is returned. The complementary procedure `not-irrelevant?` returns `#t` if and only if *obj* is not irrelevant, otherwise returning `#f`.

2.1.3. Discrete and continuous values

There is often a need to separate between continuous and discrete data. This is often done in gmdl procedures, and the handling of discrete and continuous values might be very different. It is necessary to be aware of this distinction, or unexpected errors or erroneous results may occur. In some cases, procedures operate exclusively on discrete or continuous values, and will signal an error if the argument is not of the specified type.

By convention, a discrete value in gmdl is an exact numerical value, i.e. `exact?` evaluates to `#t` for the value. A continuous value is an inexact numerical value, i.e. `inexact?` evaluates to `#t`. A value may be written in and converted to discrete or continuous form by the use of a prefix. The prefixes are `#z` for discrete values and `#r` for continuous values. With no given prefix, a number with no decimals will be assumed to be discrete and a number with decimals will be interpreted as continuous. For example, `1` would be interpreted as discrete and `1.0` as continuous.

`(disc? `*obj*`)` procedure
`(cont? `*obj*`)` procedure

These procedures test whether *obj* is a discrete or continuous value. `disc?` returns `#t` if and only if *obj* is a discrete value and `#f` otherwise, while `cont?` tests if *obj* is continuous in the same manner. These procedures produce the same results as `exact?` and `inexact?` respectively, but are provided for increased readability.

```
(cont? 3)          ⟹  #f
(disc? 3)          ⟹  #t
(cont? 3.0)        ⟹  #t
(disc? 3.0)        ⟹  #f
(cont? #?)         ⟹  #f
```

`(disc-unknown? `*obj*`)` procedure
`(cont-unknown? `*obj*`)` procedure

The procedures `disc-unknown?` and `cont-unknown?` tests if *obj* is discrete or unknown and continuous or unknown, i.e. they return the same values as `disc?` and `cont?` with the exception that they both return `#t` if *obj* is `#?`.

```
(cont-unknown? 5)      ⟹ #f
(disc-unknown? 5)      ⟹ #t
(cont-unknown? #?)     ⟹ #t
(disc-unknown? #?)     ⟹ #t
```

`(disc->cont `*z*`)` procedure
`(cont->disc `*z*`)` procedure

`disc->cont` returns a continuous representation of z. Similarly, `cont->disc` returns a discrete representation of the argument z. In both cases, the value returned is the numerically closest representable value to the argument. Both procedures return `#?` when z is unknown.

```
(disc->cont? 6)        ⟹  6.0
(disc->cont? 6.0)      ⟹  6.0
(cont->disc? 7.2)      ⟹  7
(cont->disc? #?)       ⟹ #?
```

2.2. General functionality

2.2.1. Additional numerical primitives

gmdl provides an extensive range of numerical operations. Most of them are described in chapter 7, but here we will introduce some basic functionality for making iterative operations, something that is very common in numerical programming tasks, easier.

`(++ `*z*`)` procedure
`(-- `*z*`)` procedure

These are incremental and decremental operators similar to those found in e.g. the C programming language. `++` returns $z + 1$, while `--` returns $z - 1$.

`(++! `*z*`)` procedure
`(--! `*z*`)` procedure

The `++!` procedure sets the variable z to $z + 1$, and `--!` sets z to $z - 1$. These procedures return no value.

`(set+! `*z v*`)` procedure
`(set-! `*z v*`)` procedure
`(set*! `*z v*`)` procedure
`(set/! `*z v*`)` procedure

These procedures can be used to set z to $z + v$, $z - v$, $z * v$ and z/v respectively. They are just shorthand notation for `(set! z (+ z v))` etc.

2.2.2. Unspecified return values

It is sometimes useful to be able to explicitly return no value, or an *unspecified* value, from a function. gmdl provides the procedure

`(unspec)` procedure

for this purpose. The procedure does not return a value, and can be used in e.g. the end of `(begin ...)` statements to make sure that no value is returned. It is equal to the expression `(if #f #f)`.

2.2.3. Expressions

gmdl provides a slightly extended set of expressions, mostly focused on iteration, compared to the standard Scheme language,

```
(for ((⟨variable₁⟩ ⟨init₁⟩ ⟨step₁⟩)        syntax
      ...)
     ⟨test⟩
     ⟨expression₁⟩ ⟨expression₂⟩ ...)
```

Similar to `do`, `for` is an iteration construct. It specifies a set of variables to be bound, how they are to be initialized at the start, and how they are to be updated on each iteration. When a test condition is no longer met, the loop exits.

`for` expressions are evaluated as follows: The ⟨init⟩ expressions are evaluated (in some unspecified order), the ⟨variable⟩s are bound to fresh locations, the results of the ⟨init⟩ expressions are stored in the bindings of the ⟨variable⟩s, and then the iteration phase begins.

Each iteration begins by evaluating ⟨test⟩; if the result is true, then the ⟨expression⟩ expressions are evaluated for effect, the ⟨step⟩ expressions are evaluated in some unspecified order, the ⟨variable⟩s are bound to fresh locations, the results of the ⟨step⟩s are stored in the bindings of the ⟨variable⟩s, and the next iteration begins.

If ⟨test⟩ evaluates to a false value, the iteration stops and the `for` expression returns. The return value of the expression is unspecified.

The region of the binding of a ⟨variable⟩ consists of the entire `for` expression except for the ⟨init⟩s. It is an error for a ⟨variable⟩ to appear more than once in the list of `for` variables.

A ⟨step⟩ may be omitted, in which case the effect is the same as if (⟨variable⟩ ⟨init⟩ ⟨variable⟩) had been written instead of (⟨variable⟩ ⟨init⟩).

```
(let ((vec (make-vector 5)))
  (for ((i 0 (++ i)))
       (< i 5)
       (vector-set! vec i i))
  vec)                          ⟹  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9))
      (sum 0))
  (for ((x x (cdr x)))
       (not (null? x))
       (set+! sum (car x)))
  sum)                          ⟹  25
```

It is quite often necessary to transform the interface of a procedure to fit a special purpose, e.g. binding some of the arguments to specific values or re-arrange the order of the arguments. The procedure

`(argbind (`⟨ivar⟩ ...`) (`*proc* ⟨pvar⟩ ...`))` library procedure

provides an easy to use mechanism for this purpose. `argbind` returns a new procedure with interface (⟨ivar⟩ ...) to the procedure (*proc* ⟨pvar⟩ ...).

```
(let ((newadd (argbind (x) (+ 1 x 2))))
  (newadd 3))                   ⟹  6

(let ((newsub (argbind (x y) (- y x 1))))
  (newsub 4 10))                ⟹  5
```

2.3. Storage models

An identifier may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a location is called a *variable* and is said to be *bound* to that location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value or object stored in the location to which a variable is bound is called the variable's value. By abuse of terminology, the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Variables and objects such as pairs, vectors, and strings implicitly denote locations or sequences of locations. A string, for example, denotes as many locations as there are characters in the string. (These locations need not correspond to a full machine word.) A new value may be stored into one of these locations using the `string-set!` procedure, but the string continues to denote the same locations as before.

An object fetched from a location, by a variable reference or by a procedure such as `car`, `vector-ref`, or `string-ref`, is equivalent in the sense of `eqv?` last stored in the location before the fetch.

Many objects in gmdl, like data, filters, distributions etc., are not used with call-by-value semantics. Instead, when used as an argument in a procedure call, these objects strictly use call-by-reference semantics. This means that locally bound identifiers to an argument in a procedure, as well as the argument itself, will refer to the exact same data as the identifiers referring to the corresponding object in the calling environment. Changes to the data by use of any variable bound to "the same" object will affect the other bindings.

Most objects that use call-by-reference semantics have an "&" in at end of the type name in their external representation, usually similar to `#<type-name&` *info*`>`. For example, the external representation of a Gaussian distribution with unspecified fields is

```
(make-gaussian 2)          ⟹
                             #<gaussian& (0 1)>
```

Rationale:

Models, distributions and especially data are often complex and large objects, not suitable for duplication unless specifically required. This is why these objects use call-by-reference semantics in gmdl. It can be argued that e.g. simple distributions do not need call-by-reference semantics, but since it is useful for many complex distributions, for consistency all distributions follow this scheme. The convention is intended to make the system easier to use, since there is no need for the user to be concerned about unnecessary object duplication. The obvious drawback is of course that there are many situations where the user must be aware of the convention, or errors will inevitably occur.

`(expunge` *var*`)` procedure

The `expunge` procedure can be used for removing large objects from working memory. The procedure un-binds the variable *var* and tries to free the memory used by the variable, usually by running the garbage collector. If the memory is used by other variables or referenced by other complex type variables, it will not be freed. If there is no specific need to remove an object from memory at a certain time, it is more efficient not to use this procedure but instead relying on the garbage collector to reclaim the memory when necessary.

3. Basic procedures

The gmdl environment contains a slightly expanded basic set of procedures compared to standard Scheme. Most of these procedures provide functionality for performing simple but very common tasks within gmdl.

3.1. List and vector procedures

3.1.1. Systematic data generation and selection

These procedures are provided to make it easy for the user to generate systematic data and to select subsets of data, two rather common task tasks within gmdl.

`(..` *first*`)` procedure
`(..` *first last*`)` procedure
`(..` *first last step*`)` procedure
`(list-seq` *first*`)` procedure
`(list-seq` *first last*`)` procedure
`(list-seq` *first last step*`)` procedure

Both `..` and `list-seq` generate a sequence of equally spaced numbers stored in a list. `..` is in fact just shorthand notation for `list-seq`. *first* defines the value of the first element, *last* the value of the last element to be included and the optional argument *step* the distance between the numbers. The default value of *step* is 1. The numbers in the sequence are generated by starting at *first* and then adding *step* until the value is larger than *last* unless *step* is less than zero, in which case values are added until the value is less than *last*. If only the argument *first* is given, the procedure will return the list containing only this value.

```
(list-seq 0 3)            ⟹ (0 1 2 3)
(list-seq 0 3 2)          ⟹ (0 2)
(list-seq 0 1.7 0.5)      ⟹ (0.0 0.5 1.0 1.5)
(list-seq 3 0 -1)         ⟹ (3 2 1 0)
(list-seq 5)              ⟹ (5)
```

If any argument given to `list-seq` is a real number, all elements of the resulting list will be real numbers. Otherwise, all numbers will be integers.

`(list-repeat` *lst pattern*`)` procedure

The procedure `list-repeat` is useful for generating sequences with a particular pattern. *pattern* must be either a single number or a list of numbers, usually of the same length as *lst*. If *pattern* is a single number, then `list-repeat` simply repeats *lst pattern* times. If *pattern* is a list, then each element of *lst* is repeated the number of times indicated by the corresponding element of *pattern*.

```
(list-repeat (list 1 2 3)
             2)            ⟹ (1 2 3 1 2 3)
(list-repeat (list 1 2 3)
             (list 3 2 1)) ⟹ (1 1 1 2 2 3)
```

When *pattern* contains more elements than *lst*, the additional elements in *pattern* will simply be ignored. Only elements in *lst* that have a corresponding element in *pattern* will be repeated, i.e. if *pattern* is shorter than *lst* not all patterns in *lst* will be repeated.

`(list-which` *lst test*`)` procedure

Returns a list containing the indexes of all elements in *lst* for which the procedure *test* does not evaluate to `#f`. *test* should take one argument, the content of an element in *lst*.

```
(list-which (list 3 1 0 4 2)
            (lambda (x)
              (>= x 3)))    ⟹ (0 3)
```

`(list-select` *lst selection*`)` procedure
`(list-remove` *lst selection*`)` procedure

The procedure `list-select` selects a single element or a group of elements from a list. If *selection* is a single number, the procedure returns the element at *selection* in *lst*. If *selection* is a list of numbers, a list containing the elements specified in *selection* is returned.

```
(list-select (list 'a 'b 'c 'd)
             2)            ⟹ c
(list-select (list 'a 'b 'c 'd)
             (list 2 3 1 1))⟹ (c d b b)
```

`list-remove` removes a single element or a group of elements from a list. If *selection* is a single number, the list containing all elements in *lst* except the element specified by *selection* is returned. Similarly, if *selection* is a list, the list containing all elements in *list* except the ones specified in *selection* is returned. The contents of *selection* does not have to be ordered.

```
(list-remove (list 'a 'b 'c 'd)
             2)            ⟹ (a b d)
(list-remove (list 'a 'b 'c 'd)
             (list 1 2))   ⟹ (a d)
(list-remove (list 'a 'b 'c 'd)
             (list 2 1 1)) ⟹ (a d)
```

Both `list-select` and `list-remove` require that all elements specified in *pattern* are valid, i.e. integer values larger than or equal to 0 and smaller than the length of *lst*.

`(vector-seq` *first*`)` procedure
`(vector-seq` *first last*`)` procedure
`(vector-seq` *first last step*`)` procedure
`(vector-repeat` *vec pattern*`)` procedure
`(vector-which` *vec*`)` procedure
`(vector-select` *vec selection*`)` procedure
`(vector-remove` *vec selection*`)` procedure

These procedures correspond directly to their list counterparts, but instead of operating on and returning lists, these procedures operate on and return vectors. Please consult the corresponding documentation for the list counterparts for a more detailed description.

3.2. Environment and system interface

gmdl provides a number of environment and system related commands that are used to e.g. interact with the underlying system, and to specify how gmdl should behave under certain circumstances.

3.2.1. Printing detail and options

The level of detail of printed objects can be set to three different levels: brief, detailed and full, where brief provides the least detail and full the most detail. Exactly what is printed for different objects and different detail level settings is implementation dependent. However, the levels should be meaningful. For example, a matrix printed at the brief level only prints a message saying that the object is a matrix and its dimensions, while at the detailed level the whole contents of the matrix is displayed when the matrix dimensions are reasonably small. The full level would then correspond to always printing the whole contents of the matrix, regardless of its dimension.

`(print-detail)` procedure
`(set-print-detail!` *detail*`)` procedure

The procedure `print-detail` returns the symbol `brief`, `detailed` or `full` depending on the current print detail level. To set the print detail level, use `set-print-detail!` where *detail* should be one of the symbols mentioned above.

3.2.2. Program information

The following procedures can be used to obtain general information about the gmdl system.

`(about)` procedure
`(copyright)` procedure
`(citation)` procedure
`(webpage)` procedure

`about` displays a brief description about gmdl on the standard output, while `copyright` displays a copyright notice. The procedure `citation` displays information on how to cite the gmdl system, and `webpage` displays a link to the gmdl web page.

`(gmdl-compile-time)` procedure
`(gmdl-version)` procedure
`(mdl-version)` procedure

`gmdl-compile-time` returns a string containing the date and time of when the running system was compiled. `gmdl-version` returns a list containing three elements, the *major*, *minor*, and *micro* version of the gmdl system. The procedure `mdl-version` returns the version of the underlying mdl library on the same format.

3.2.3. File utilities

gmdl contains a number of file utilities that are useful when performing data analysis. Their names often reflect their standard UNIX [4] counterpart.

`(pwd)` procedure
`(cd)` procedure
`(cd` *dir*`)` procedure

`pwd` returns a string containing the current working directory. The procedure `cd` changes the current working directory to the directory specified by the string *dir*. If *dir* is not specified, the current working directory is changed to the home directory of the user.

`(ls)` procedure
`(ls` *dir*`)` procedure

`ls` displays the contents of the directory specified in *dir* on the current output port. If no arguments are given, the procedure displays the contents of the current working directory.

`(file-display` *filename*`)` procedure
`(file-display` *filename port*`)` procedure
`(file-head` *filename*`)` procedure
`(file-head` *filename n*`)` procedure
`(file-head` *filename n port*`)` procedure

These procedures display the contents of a file specified by the string $filename$. `file-display` displays the complete file on *port*. If no output port is specified, the current output port is used. The `file-head` procedure displays the first n lines of the file on *port*. Again, if no output port is specified, the current output port is used. If n is not specified, the 10 first lines are displayed.

`(file-wc` *filename*`)` procedure

`file-wc` counts the number of lines, words and characters in the file specified by the string *filename*. The procedure returns a list containing the total number of lines, the total number of words (separated by white space), the total number of characters, the maximum number of characters of any line, and the filename.

3.3. Data conversion and interpretation

3.3.1. Date and time

Many data analysis tasks involve interpreting and representing dates and time. gmdl provides a number of standard procedures to help manage such tasks. Below we will make a distinction between *time strings*, which only represent a time and never include a date, and *date strings*, which are often just composed of a date representation but may also include a time specification such as the time of day.

`(time-string->sec` *str*`)` procedure
`(date-string->sec` *str*`)` procedure

`time-string->sec` returns the number of seconds after midnight as specified by the string *str*. If *str* cannot be interpreted as a time by the procedure, `#f` is returned. The procedure `date-string->sec` returns the number of seconds after midnight the 1st of January, 1970. If *str* cannot be interpreted as a date `#f` is returned.

`(time-string->time` *str*`)` procedure
`(date-string->date` *str*`)` procedure
`(date-string->edate` *str*`)` procedure

`time-string->time` returns the time specified by the string *str* as a list of three elements, the first element representing the number of hours, the second the number of minutes, and the third the number of seconds. If *str* cannot be interpreted as a time by the procedure, `#f` is returned.

`date-string->date` returns the date specified by *str* as a list of three elements, the first element representing year, the second month and the third day of the month. Similarly, `date-string->edate` returns an *extended* date as a list representing the date and time in *str*. The returned list have six elements. The elements represent, in order, year, month, day of month, hours, minutes, and seconds. Again, if the procedures cannot interpret *str* as a date, `#f` is returned.

`(sec->date` *s*`)` procedure
`(sec->edate` *s*`)` procedure
`(date->sec` *lst*`)` procedure
`(date->absolute` *lst*`)` procedure
`(absolute->date` *d*`)` procedure

`sec->date` converts the number of seconds s after 1st of January 1970 to a date represented as a list of the year, month and day of month. `sec->edate` returns a list including the same information, but also the hour, minute and second of the day. The procedure `date->sec` converts the date or extended date *lst* on list form to the number of seconds after 1st of January 1970.

The `date->absolute` procedure converts a date *lst* using list representation as above to an absolute date. `absolute->date` converts the absolute date d to a date on list form. Both procedure assumes the use of the Gregorian calendar [5].

4. Formats, data and filters

Data for modeling and analysis is generally represented in gmdl in one of several *data object* types. A data object is always linked to a *format*, i.e. a description of the entries in the data object. A format is a specific type in gmdl, and one format can be shared by several data objects. Both formats and entries in data objects are composed of a number of fields, each with a specific field format. A *field format* is a specific type, and a data format is a set of field formats. All models operate on data objects and entries of a specific format.

To transform data to a suitable representation, *filters* can be used. A filter is an object type in gmdl. A set of standard filters is always provided, and new filters can be specified easily. Most often, the filtered data is transformed into a new, *virtual* data object so that model functionality and other standard procedures can be used in a straightforward manner. A virtual data object does not store data explicitly, but refers to other data objects and procedures, calculating a transformation of data when necessary.

In this chapter we will describe how gmdl handles data, formats and transformation of data, and describe related functionality and the data types involved. We will also give a brief description on how to work with data and data transformations in the usual modeling or analysis setting.

4.1. Introduction to data formats and field formats

Data formats are used in gmdl to describe the format of a data instance. A data instance is an ordered set of values, each on a certain format, and can be stored in e.g. data objects. Data objects, models, etc. use data formats to describe the contents of the object or what kind of data the object expects for its procedures. Fig. 4.1 describes the relationship between models, data and formats. In gmdl, data formats are often used to check whether the use of data is consistent or not, i.e. that procedures are called with the right kind of data and that interacting objects use the same data format.

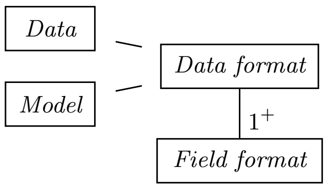

Figure 4.1: Relations between formats, data and models

Data formats can also be used to describe information about the whole data object or the specific fields, e.g. the sample rate of a data object or minimum and maximum value of a continuous field. Field formats are also used to associate a description to an outcome in a field with discrete values. Discrete values are always stored as numbers, $0, 1, 2 \ldots$ in gmdl, but the interpretation of these values might be e.g. $red, blue, green, \ldots$ or similar.

A format specifies both an internal and an external representation of data. The internal representation is used for calculations, in which e.g. discrete values are represented as integers. This is how data is represented within gmdl. The external representation can be different, and the format e.g. specifies what text strings or symbols the integer values in a discrete field represents, or how to represent time and dates on easy to read formats. This is how data is displayed and written to text files.

Both data and field formats can be tested for equality using the `equal?` procedure. The formats are considered equal if they represent data in the same way, i.e. all field formats are equal, although field names and meta-information are not considered. Two discrete fields are considered equal if the number of outcomes are the same, no attention is given to the names of the outcomes. Similarly, markers are not compared between the formats.

4.2. Data formats

A format is a record structure containing a set of field formats, a set of values representing meta-information about the object (indexed by a key string), and a set of markers. The meta-information can be used to specify e.g. how often the data was sampled or other notes about the data. Markers are used to store locations of and information about specific indeces in data.

The names of all procedures operating on formats start with `dformat` (short for "data format", to separate it from "field format").

`(make-dformat)` procedure
`(make-dformat` *flist*`)` procedure

The `make-dformat` procedure is used to construct a new data format. If no argument is given to the procedure, the empty data format (with no fields) is returned. Given a list of field formats $flist$, the data format consisting of the field formats in $flist$ in order is returned.

`(dformat?` *obj*`)` procedure

The procedure `dformat?` returns `#t` if and only if *obj* is a data format, otherwise `#f` is returned.

`(dformat-labels?` *dform*`)` procedure
`(dformat-dfile-labels?` *dform*`)` procedure
`(dformat-separator` *dform*`)` procedure

There are several options available for the external textual representation of data in a format. `dformat-labels?` returns a `#t` if the fields are labelled, or have field names, and whether these field names should be read from format specifications or not. Otherwise it returns `#f`. The value can be changed using the `set!` procedure, e.g using `(set! (dformat-labels? f) #t)`.

In many cases, field names are available in the actual data file. A data format can optionally read field names, or labels, from the data file if available. Any previous specification of field names is discarded if data is read into a data object using the data format. If the procedure `dformat-dfile-labels?` returns `#t`, field names will be read from data files. The value can be changed using `set!` in the same manner as for `dformat-labels?`.

The character separating values in data entries can be set to any valid character or whitespace. This value is especially important when reading data from a text file, since the wrong value might produce inconsistent results or errors. Setting the separator character to whitespace means that any number of blank characters (spaces, horizontal tabs etc.) can separate the values in a data entry. The procedure `dformat-separator` returns the specified separator character for the data format. If the separator is general whitespace, the symbol `whitespace` is returned. The format can be specified using the `set!` procedure as described above.

`(dformat-length` *dform*`)` procedure
`(dformat-ref` *dform*`)` procedure
`(dformat-ref` *dform i*`)` procedure

`dformat-length` returns the number of field formats, or *length* of the data format. `dformat-ref` returns the field format at position *i*, where *i* is an integer larger than or equal to 0 and less than the total number of fields in the format. The argument *i* can also be a list of valid fields, in which case a list of field formats is returned. If no argument *i* is given, a list of all field formats is returned.

`(dformat-add!` *dform fform*`)` procedure
`(dformat-insert!` *dform i fform*`)` procedure
`(dformat-remove!` *dform i*`)` procedure
`(dformat-swap!` *dform* i_1 i_2`)` procedure

These procedures modify the field formats of the data format *dform*. The procedure `dformat-add!` adds the field format *fform* to the data format. If *fform* is a list of field formats, these formats are added in order. `dformat-insert` inserts *fform* at position *i*. Both procedures increase the number of fields in the data format by one or more. `dformat-remove!` removes the field format at position *i*, decreasing the number of fields by one, and `dformat-swap` swaps positions of the field formats in i_1 and i_2.

`(dformat-field-name` *dform i*`)` procedure
`(dformat-set-field-name!` *dform i name*`)` procedure
`(dformat-field-index` *dform name*`)` procedure

The name of field *i* is returned by `dformat-field-name`. If the name has not been specified, the procedure returns the empty string. When *i* is a list of fields, a list of field names is returned. A field name can be specified using `dformat-set-field-name!`, where *i* is the field and *name* the new field name. Note that the field name is actually contained within the field format, so using this procedure will modify the corresponding field format itself. These procedures are provided for convenience only. Again, *i* and *name* can be lists of fields and strings respectively.

The procedure `dformat-field-index` returns the index (position) of the first field with field name *name*. If no such field is present in the format, `#f` is returned.

`(dformat-write` *dform file*`)` procedure
`(dformat-display` *dform*`)` procedure
`(dformat-display` *dform port*`)` procedure

A format specification *dform* can be written to a file format suitable for machine reading using the procedure `dformat-write`, where *file* is the name of the file to be written. The actual format is implementation dependent, but must be complete and interpreted by the corresponding read procedure `dformat-read!`.

`dformat-display`, on the other hand, displays the format *dform* in such a way that it is easy to read on the port *port*. If no port is specified, the current output port is used.

`(dformat-guess!` *dform file*`)` procedure
`(dformat-read!` *dform file*`)` procedure

`dformat-guess!` tries to guess the format of the file with filename *file*, and sets the format *dform* to this format. Exactly what files the procedure operates on and what algorithm the format generation is based on depends on the implementation.

The procedure `dformat-read!` reads the format specification in *file*, and sets *dform* to the specified format. The read procedure may operate on a number of different file formats, but is required to be able to read the output of `dformat-write`.

`(dformat-meta-ref` *dform key*`)` procedure
`(dformat-meta-set!` *dform key value*`)` procedure
`(dformat-meta-remove!` *dform key*`)` procedure
`(dformat-clear-meta!` *dform*`)` procedure
`(dformat-meta->list` *dform*`)` procedure

A data format can store meta-data, i.e. information that can be considered common for the whole format. Such data could be information about the sample rate, origin of data or other descriptions of the format. All meta-data stored in a format is indexed by a unique key used for referencing the meta-data. This key is always a symbol,

The procedure `dformat-meta-ref` returns the data referenced by *key* in *dform*. The variable *key* must be a symbol. If no data matching *key* is found in data `#f` is returned. To change or add meta data to a format, `dformat-meta-set!` is used. If no previous meta-data with key *key* is available in *dform*, *value* is added to the set of meta data with key *key*. If there already is an entry with this key, its value will be set to *value*. Again, *key* must be a symbol, but *value* can be any basic gmdl value, e.g. a string or a number.

`dformat-meta-remove!` removes the meta-data specified by *key*. To clear all meta-data in a format *dform*, `dformat-clear-meta!` can be used. The conversion procedure `dformat-meta->list` returns all meta-data in *dform* as a list of lists, containing the keys and the corresponding values.

4.3. Field formats

Field formats are used to describe the individual elements in a data entry, and an ordered set of field formats constitute a data entry description as used in the data format type. Field formats have, like the data format, both an internal and external representation of data. Discrete attributes are always represented as integers in gmdl, but their external representation might be the corresponding labels or descriptions of each outcome. Time and date formats work in a similar way, where they are represented internally as a single number but externally as a text string that is easy to read.

Testing for equality between field formats can be performed using the `equal?` procedure. Two field formats are considered equal if they represent data in the same way. Field names are not considered. Two discrete field formats are considered equal if the number of outcomes are the same, no matter what labels are used for the outcome.

4.3.1. Field format functionality

`(fformat?` *obj*`)` procedure

The procedure `fformat?` returns true if and only if *obj* is a field format.

`(fformat-name` *fform*`)` procedure
`(fformat-set-name!` *fform name*`)` procedure

The `fformat-name` procedure returns the name of the field format *fform*. The name is usually used to describe the name of the corresponding attribute, and can also be modified by a data format that the field format belongs to. `fformat-set-name!` sets the name of *form* to *name*, where *name* is any string.

`(fformat-interval` *fform*`)` procedure

Return the interval specified for the field format *fform* as a pair of two values, the minimum and the maximum value. The symbol `'...` represents negative or positive infinity, depending on the position in the pair. An interval is not really applicable to all field format types, in which case the interval minus infinity to infinity will be returned.

`(fformat-interpret` *fform str*`)` procedure
`(fformat-represent` *fform obj*`)` procedure

These procedures convert between a field formats internal and external representation of data. `fformat-interpret` converts a text string *str*, describing the external representation, into its internal representation in gmdl using field format *fform*. The opposite conversion is performed by `fformat-represent`, which converts a gmdl type *obj* into its external representation as specified by the field format *fform*.

In all field formats, `#?` is represented externally as the string `"#?"`, and all values outside the domain of the field format will be represented as an unknown. External representations that cannot be interpreted by a field format are represented internally as `#?`.

`(fformat-display` *fform*`)` procedure
`(fformat-display` *fform port*`)` procedure

`fformat-display` displays information about the format *fform* and its properties in an easy to read form on port *port*. If no port is specified, the current output port is used.

4.3.2. Field format types

There are several field format types available in gmdl that cover most basic forms of data. The names of the types start with `fform-` for field format. Below is a listing of the available field formats and what types of data they are used for:

`fform-unknown`	`Unknown`
`fform-discrete`	`Discrete (not ordered)`
`fform-int`	`Integer (ordered discrete)`
`fform-cont`	`Continuous`
`fform-string`	`String`
`fform-time`	`Time`
`fform-date`	`Date`

Here we will describe the different field format types along with the corresponding construction procedure and associated functionality. Please note that, although not explicitly stated in the procedure definition, a field name can always be given as an optional last argument to the field format construction procedures.

`(fformat-unknown?` *obj*`)` procedure
`(fformat-disc?` *obj*`)` procedure
`(fformat-int?` *obj*`)` procedure
`(fformat-cont?` *obj*`)` procedure
`(fformat-string?` *obj*`)` procedure
`(fformat-time?` *obj*`)` procedure
`(fformat-date?` *obj*`)` procedure

These procedures return true if and only if *obj* is of the implied field format.

`(make-fformat-unknown)` procedure

The procedure `make-fformat-unknown` creates an unknown field format. This format is used e.g. when data objects are automatically extended to accommodate a greater number of fields, but can also be used when there is little or no information about the format of the field, or the data is mixed. Generally, the unknown field format interpret numbers that could only be continuous (e.g. `"3.4"`) as continuous data, numbers that could be integers (e.g. `"32"`) as integer data, and everything else as strings. Converting from internal to external representation follow the same criteria.

`(make-fformat-disc` *n*`)` procedure
`(make-fformat-disc` *n start*`)` procedure
`(fformat-disc-values` *fform*`)` procedure
`(fformat-disc-values` *fform i*`)` procedure

If *n* is an integer larger than 0, the procedure `make-format-disc` returns a discrete field format with *n* values. The external representation used is assumed to be on the form `"0", "1", "2"` etc. for *n* values. If an integer argument *start* is given, the external representation is assumed to start from this value instead of 0. The argument *n* can also be given as a list of strings, where the strings represent the external representation of the values in order. The resulting discrete field format has as many values as the length of *n* in this case. `fformat-disc-values` returns a list of the external representation of each outcome as strings unless the optional argument *i* is provided, in which case the integer value *i* represents the index of an outcome for which the external representation will be returned.

`(make-fformat-int)` procedure
`(make-fformat-int` *interval*`)` procedure

`make-fformat-int` creates an integer field format. The integer field format is used to represent ordered discrete data that does not necessarily have a finite number of values. An *interval* can optionally be specified as a pair containing the minimum and maximum value of the attribute. The symbol `'...` represents positive and negative infinity, which is also the default minimum and maximum value.

`(make-fformat-cont)` procedure
`(make-fformat-cont` *interval*`)` procedure

The procedure `make-fformat-cont` returns a continuous field format, used to represent continuous or real data. An *interval* can optionally be specified as a pair containing the minimum and maximum value of the attribute. The symbol `'...` represents positive and negative infinity, which is also the default minimum and maximum value.

`(make-fformat-string)` procedure

`make-fformat-string` is used to create a string format. The string field format represents all data as strings, no matter their content.

`(make-fformat-time)` procedure

The time field format tries to interpret data as discrete values representing seconds after midnight, and can be created by using `make-fformat-time`. Note that the precision is limited to seconds.

`(make-fformat-date)` procedure

The date field format is very similar to the time field format in that it interprets data as discrete values representing seconds. It differs in the respect that it does not represent seconds after midnight, but rather the total number of seconds since a fixed, implementation dependent time and date. The field format is created using `make-fformat-date`.

4.4. Data objects

A data object in gmdl is essentially an ordered set of values, referenced by an index and a field, and a data format describing the stored data. The data pointed out by a specific index is always an ordered set of values described by the data format. The index itself is usually a single integer value, but could also be e.g. a list of integers (in the case of multi-dimensionally indexed data) or a symbol.

4.4.1. Data object types

Several kinds of data objects are available in gmdl, to provide for efficient resource allocation for different applications. Data objects are divided into two main categories, storage data objects and virtual data objects. Storage data objects store the data explicitly on one form or the other, without referring to other data objects or procedures. Virtual data objects on the other hand always do not actually store data directly, but refer to other data objects, filters etc.

A data object can be *mutable*, which refers to the ability to change its contents once created, and *adaptable*, meaning that the data object will automatically grow in size to accomodate data assigned to previously unused locations. When extending the size, all previously unused locations that have not been specified but that is now accessible will be initialized to the unknown value, `#?`.

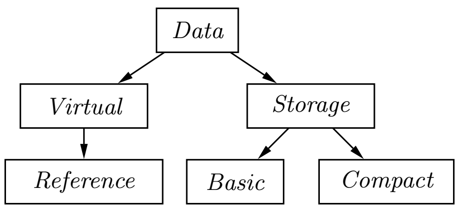

Figure 4.2: Data object types

The pre-defined data types in gmdl are basic data objects, compact data objects, reference data objects and virtual data objects, and will be described in more detail below.

4.4.2. General data object functionality

The following procedures operate on all kinds of data objects.

`(data?` *obj*`)` procedure

The procedure `data?` returns `#t` if and only if *obj* is a data object, otherwise `#f` is returned.

`(data-entries` *dobj*`)` procedure
`(data-fields` *dobj*`)` procedure
`(data-size` *dobj*`)` procedure

The procedure `data-entries` returns the number of entries currently in data object *dobj*. Similarly, `data-fields` returns the number of fields in *dobj*. `data-size` returns a list containing the number of fields and entries in the data object *dobj*.

`(data-ref` *dobj i*`)` procedure
`(data-ref` *dobj i field*`)` procedure

The procedure `data-ref` returns the values stored at index *i* in the data object *dobj* as a vector. If the *field* argument is provided, the value stored in field *field* at the entry specified by *i* in *dobj* is returned. *field* might be given either as an integer, specifying the location of the field, or a string, specifying the field name.

`(data->vector` *dobj*`)` procedure
`(data->vector` *dobj transp*`)` procedure
`(data-field->vector` *dobj field*`)` procedure

The data stored in a data object can be converted to a vector representation using `data->vector`. The procedure returns a vector of vectors, containing all values of all fields for each entry in the data object. If the procedure is provided with the optional argument *transp* as the symbol `'transpose`, the data is instead returned as a vector of vectors, containing all values for all entries and all fields. The procedure `data-field->vector` converts the contents of field *field* in data object *dobj* to a list.

`(data->list` *dobj*`)` procedure
`(data->list` *dobj transp*`)` procedure
`(data-entry->list` *dobj i*`)` procedure
`(data-field->list` *dobj field*`)` procedure

The data stored in a data object can be converted to a list representation using `data->list`. The procedure returns a list of lists, containing all values of all fields for each entry in the data object. If the procedure is provided with the optional argument *transp* as the symbol `'transpose`, the data is instead returned as a list of lists containing all values for all entries and all fields. The procedure `data-entry->list` converts the contents of entry *i* in data object *dobj* to a list, while `data-field->list` converts the contents of field *field* in data object *dobj* to a list.

`(data-display` *dobj*`)` procedure
`(data-display` *dobj port*`)` procedure
`(data-write` *dobj file*`)` procedure

The procedure `data-display` displays the contents of the data object *dobj* on port *port*. If no port is specified, the current output port is used. `data-write` writes the contents of the data object to the file specified by the string *file*. Note that while the procedure `data-display` can potentially be used to store the contents of a data object to a file, its output is not specifically intended to be machine readable. It does not necessarily output all meta-data in a consistent manner and does not strictly follow the format specification in the data format, since its output is mainly intended to be displayed on screen.

`(data-dformat` *dobj*`)` procedure

`data-dformat` returns the current data format of the data object *dobj*.

`(data-field-name` *dobj i*`)` procedure
`(data-set-field-name!` *dobj i name*`)` procedure
`(data-field-index` *dobj name*`)` procedure

These procedures are provided for convenience only, as they interact directly with the data object's data format. See the corresponding procedures `dformat-set-field-name`, `dformat-field-name`, and `dformat-field-index` for a closer description.

`(data-revision` *dobj*`)` procedure

All data objects store a revision number that is increased every time the data object is modified in any way. The current revision number of a data object *dobj* can be accessed using `data-revision`. The revision number is useful to keep track of whether a data object has changed or not since a certain point in time.

`(data-marker-ref` *form ref*`)` procedure
`(data-marker-set!` *form i key value*`)` procedure
`(data-marker-remove!` *form ref*`)` procedure
`(data-clear-markers!` *form*`)` procedure
`(data-markers->list` *form*`)` procedure

All data objects also store a set of markers. A marker is very similar to meta-data in a format in that it stores information and can be accessed using a key symbol, but the marker also has a reference to a certain index in the data. Markers are primarily used to mark and comment events in data sets, e.g. start and end points of test series, but can also store associated information in much the same manner as meta-data.

`(data-for-each` *dobj expr*`)` procedure

Performs the expression *expr* for each index in the data object *dobj*. *expr* should take two arguments, a data object (which will be *dobj* when `data-for-each` is evaluated) and an index.

`(data-which` *dobj expr*`)` procedure

Returns a list containing the indeces in the data object *dobj* for which the expression *expr* does not evaluate to `#f`. *expr* should take two arguments, a data object (which will be *dobj* when `data-which` is evaluated) and an index.

`(data-index-sort` *dobj less*`)` procedure
`(data-stable-index-sort` *dobj less*`)` procedure

Return a list containing the indices of the data object *dobj* in sorted order accoreding to the expression *less*. *less* should take three arguments, a data object (which will be *dobj* when the sorting is performed) and two indeces. The procedure *less* should evaluate to a value different than `#f` when the first index is to be considered less than the second index. `data-stable-index-sort` is guaranteed to be stable, while no such guarantee exists for `data-index-sort`.

`(data-merge-match` *dobj*$_1$ *dobj*$_2$ *less*`)` procedure
`(data-merge-match` *dobj*$_1$ *dobj*$_2$ *less equal*`)` procedure

`data-merge-match-sorted` matches the contents of data object *dobj*$_1$ with the contents of the data object *dobj*$_2$, using the procedures *equal* and optionally *less*. Both procedures should take four arguments, being, in order: the first data object, an index within that data object, the second data object, and an index within that data object. The procedures should return a value different from `#f` when the index in the first data object is considered to be strictly less than the index in the second data object, or equal, respectively. Otherwise, `#f` should be returned. The *equal* argument can be provided for increased clarity and correctness if *less* happens not to be strict.

The procedure returns a vector of the same length as the number of entries in *dobj*$_1$. Each element in the vector contains a list of all indeces in *dobj*$_2$ for which *equal* does not evaluate to `#f`.

4.4.3. Storage data object functionality

Storage data objects explicitly store data in one form or the other, without referring to other objects created earlier. The procedures below operate on all storage data objects.

`(storage-data?` *obj*`)` procedure
`(mutable-data?` *obj*`)` procedure
`(adaptable-data?` *obj*`)` procedure

The procedures above return `#t` if and only if *obj* is a storage data object, mutable data object and adaptable data object respectively, otherwise `#f` is returned.

`(data-clear!` *dobj*`)` procedure

Clears (if possible) all data in *dobj*, including markers and other meta data. The data format is not changed.

`(data-set!` *dobj i lst*`)` procedure
`(data-set!` *dobj i field val*`)` procedure

The procedure `data-set!` assigns the values in data object *dobj* at index *i* to the values in the list *lst*. If the data object is adaptable, then *lst* can have the same or larger size than the number of values stored at each index, while if *dobj* is not adaptable the sizes must match exactly. If a field number $field$ is provided then the value referenced by index *i* and specified field is assigned the value *val*. If *dobj* is adaptable, the index and field can have any value (as long as $field$ is larger than or equal to zero and *i* is within the index domain). Otherwise *i* and $field$ have to be within the current size of *dobj*. Note that both variants of the procedure require that *dobj* is mutable.

`(data-set-field!` *dobj field lst*`)` procedure
`(data-set-all!` *dobj lst*`)` procedure
`(data-set-all!` *dobj lst transpose*`)` procedure

`data-set-field!` sets the value of field $field$ in all entries of the data object *dobj* to the values of the list *lst*. If the data object is adaptable, *lst* may be equal to or longer than the current number of entries in the data object. If not, the lenght of *lst* must be equal to the number of entries in *dobj*. The procedure `data-set-all!` sets all data in the data object *dobj* to the contents of *lst*, which should be a list of lists containing the values of all fields for all entries. If the optional argument *transpose* is given and equal to the symbol `'transpose`, the data in *lst* is interpreted as a list of lists containing the values of all entries for all fields. If the data object is adaptable, the data in *lst* can exceed the size of *dobj*. Otherwise, the size of *lst* and the data object must match.

`(data-read!` *dobj file*`)` procedure

The procedure `data-read!` clears all data in the data object *dobj* and reads new data from the file specified by the string *field*. The actual format of the data in the specified file is implementation dependent, but it the procedure must at least be able to read data written by `data-write`. *dobj* must be a mutable data object.

The basic data object

The basic data object is a flexible data object that is both mutable and adaptable. It can store values of different types in all positions of the data object without restriction. When the data object needs to be expanded to more fields than currently specified in the data format, the data format will be expanded with unknown field formats to accomodate for the new data.

`(make-basic-data)` procedure
`(make-basic-data` arg_1`)` procedure
`(make-basic-data` arg_1 arg_2`)` procedure

Constructs a basic data object. If no arguments are given to the procedure, an empty data object with no fields is returned. If arg_1 is a data object, the procedure will return a new basic data object with the same contents and data format as arg_1. If arg_1 is a data format, the returned, empty, data object will have this format, while if arg_1 is a string (and arg_2 is not) the data format will be read from the file specified by arg_1. In both cases, the optional integer argument $arg2$ specifies the number of entries to allocate in the data object, all values being initialized to `#?`. If both arguments arg_1 and arg_2 are strings, the returned data object will use the format specified in arg_2 and contain data read from the file specified in arg_1.

`(basic-data?` *obj*`)` procedure

The procedure `basic-data?` returns `#t` if and only if *obj* is a basic data object, otherwise `#f` is returned.

`(basic-data-compact!` *bdobj*`)` procedure

The memory allocated for a basic data object may exceed what is actually being used for the stored data. The procedure `basic-data-compact` re-allocates memory for the basic data object *bdobj* in a manner that minimizes the memory consumption while storing the same data. This procedure can be very slow for large objects, and should not be used excessively.

The compact data object

The compact data object uses tries to use as compact representation as possible for each value stored in the data object, but the gains in memory efficiency also means a slight loss in flexibility. The compact data object is mutable but not adaptable and must usually be allocated to the correct size when created. It also needs a fixed data format to be able to choose an effiecient representation for the data, and all data in a certain field can only be of one specific type, e.g. integer values or continuous values.

`(make-compact-data` arg_1`)` procedure
`(make-compact-data` arg_1 arg_2`)` procedure

Constructs a basic data object. If arg_1 is a data object, the procedure will return a new compact data object with the same contents and data format as arg_1. If arg_1 is a data format, the returned (empty) data object will have this format, while if arg_1 is a string (and arg_2 is not) the data format will be read from the file specified by arg_1. In both cases, the optional integer argument $arg2$ specifies the number of entries to allocate in the data object, all values being initialized to `#?`. If both arguments arg_1 and arg_2 are strings, the returned data object will use the format specified in arg_2 and contain data read from the file specified in arg_1.

`(compact-data?` *obj*`)` procedure

The procedure `compact-data?` returns `#t` if and only if *obj* is a compact data object, otherwise `#f` is returned.

4.4.4. Virtual data object functionality

The reference data object

The reference data object is a simple virtual data object. It does not store any data itself but refers to a specified input data object. It does, however, keep most other parameters independent of the data object it refers to. The data format, size and number of fields etc. will reflect the specified input data object. The reference data object is useful e.g. for certain kinds of data selection.

`(make-reference-data` *dobj*`)` procedure
`(make-reference-data` *dobj selection*`)` procedure

Constructs a reference data object that refers to the data object *dobj*. If *selection* is a list of integer values, only the entries specified in *selection* will be considered to be known (see `reference-data-select!` below).

`(reference-data?` *obj*`)` procedure

The procedure `reference-data?` returns `#t` if and only if *obj* is a reference data object, otherwise `#f` is returned.

To make data selection easier, the contents of an entire field or entry can be set to unknown in the reference data object. The effect is simply that all values in a certain field or entry will be returned as `#?` instead of its value in the data object referred to (which may of course also be `#?`).

This setting will not affect the data that the reference data object refers to, and the field or entry can be set to known again at any point without affecting the original data.

`(reference-data-field-unkn?` *d f*`)` procedure
`(reference-data-set-field-unkn!` *d f v*`)` procedure
`(reference-data-entry-unkn?` *d i*`)` procedure
`(reference-data-set-entry-unkn!` *d i v*`)` procedure

These procedures return whether all values in specific fields *f* or entries *i* are considered unknown or not in the reference data object *d*, as well as sets these values to the boolean variable *v*. *f* and *i* may be either a single integer value or a list of integer values, in which case a list of boolean values are returned or all specified fields or entries are set to *v*. The argument *v* can also be a list of boolean values of the same length of *f* or *i*.

`(reference-data-select!` *d selection*`)` procedure

Selects the entries specified by the list of integers in *selection* in the reference data object *d*, i.e. marks all entries as unknown as described above except for the ones specified in *selection*.

The virtual data object

The virtual data object does not store any data in itself, but simply refers to other data objects or filters. Actually, the virtual data object is only an ordered collection of references to filters, allowing the data object to be used for flexible data selection and transformation. However, one of the most common uses for a virtual data object is to refer to a number of fields in perhaps several different data objects. The solution is simply to create filters that refers directly to the required fields in the data objects and insert them into the virtual data object. This is done automatically by many of the procedures that operate on virtual data objects to make them easier to use in practice.

To facilitate for further transformation of the input data in the virtual data object, each field has its own stack of filters. Adding a filter to a specific field in the virtual data object means that the output of this filter will be used instead of the specified input filter to the field (see below).

`(make-virtual-data)` procedure
`(make-virtual-data` *obj*`)` procedure
`(make-virtual-data` *obj fields*`)` procedure

Constructs a virtual data object. If no arguments are given, an empty virtual data object is returned. If *obj* is a list of filters, these filter will be added in order. If *obj* is a data object, filters referring to all fields in *obj* will be added in order to the returned virtual data object. The additional argument *fields* should be a list of fields in *dobj* that specifies what fields to use.

`(virtual-data?` *obj*`)` procedure

The procedure `virtual-data?` returns `#t` if and only if *obj* is a compact data object, otherwise `#f` is returned.

`(virtual-data-add-field!` *vd f*`)` procedure
`(virtual-data-add-field!` *vd do i*`)` procedure
`(virtual-data-add-all-fields!` *vd do*`)` procedure

The procedure `virtual-data-add-field!` appends one field to the virtual data object *vd*. If two arguments is given, the filter *f* is added. *f* may also be a list of filters, in which case each filter in the list will be added in order. If three arguments are used, *do* specifies a data object and *i* a field that will be added to the virtual data object. *i* may also be a list of fields, added in order. `virtual-data-add-all-fields!` adds filters referring to all fields in *do* in order to the virtual data object *vd*.

`(virtual-data-insert-field!` *vd i f*`)` procedure
`(virtual-data-insert-field!` *vd i do j*`)` procedure
`(virtual-data-remove-field!` *vd i*`)` procedure
`(virtual-data-swap-fields!` *vd* i_1 i_2`)` procedure

The procedure `virtual-data-insert-field!` performs an operation similar to that of `virtual-data-add-field!`, but instead of appending the filter to the end of the fields in the virtual data object it is inserted at position *i*. To remove filter *i* in the virtual data object *vd*, `virtual-data-remove-field!` is used. To simply swap the positions of filters i_1 and i_2, use the procedure `virtual-data-swap-fields!`.

`(virtual-data-delay-ref` *vd i*`)` procedure
`(virtual-data-set-delay!` *vd i d*`)` procedure

`virtual-data-delay-ref` returns the current delay for field *i* in the virtual data object *vd*. If *i* is a list, a list of delays for the fields in *i* is returned. To set the delay for field *i* to *d*, `virtual-data-set-delay!` is used. Again, *i* and *d* can be given as lists, in which case the procedure will set the delay of all fields specified in *i* to the value of the corresponding element in *d*.

`(virtual-data-filter-ref` *vd i*`)` procedure
`(virtual-data-add-filter!` *vd i f*`)` procedure

`virtual-data-filter-ref` returns the currently outermost filter for field *i* in *vd*, i.e. the latest added filter to the field, or, in the case that no filters have been added, the input filter. If *i* is a list, a list of the currently outermost filters for the fields specified in *i* is returned. The procedure `virtual-data-add-filter!` adds the filter *f* to the stack of added filters at field *i* in *vd*. The input of filter *f* will be set to the last added filter to field *i*, and if no other filters have been added to the input filter. The arguments *i* and *f* may also be given as lists, in which case the procedure will operate on each corresponding pair of elements in turn.

`(virtual-data-remove-filter!` *vd i*`)` procedure
`(virtual-data-remove-filters!` *vd i*`)` procedure
`(virtual-data-remove-all-filters!` *vd*`)` procedure

`virtual-data-remove-filter!` removes the filter that was latest added to field *i* in *vd*. If no filters have been added, the procedure does nothing. Similarly, `virtual-data-remove-filters!` removes all filters that have been added to field *i* in *vd*. In both procedures, if *i* is a list, the procedures perform the operation on all fields specified in the list in order. `virtual-data-remove-all-filters!` removes all added filters for all fields in *vd*.

`(virtual-data-permutation` *vd*`)` procedure
`(virtual-data-permute!` *vd lst*`)` procedure
`(virtual-data-clear-permutation!` *vd*`)` procedure

`virtual-data-permutation` returns the current permutation used by the virtual data object *vd*. The permutation can be set using the procedure `virtual-data-permute!`, where *lst* must be a list of valid indices. `virtual-data-clear-permutation` removes any modified permutation from the data object and returns it to standard ordering of indeces.

4.5. Filters

A filter is a function operating on any number of inputs, although perhaps most commonly one, to produce one single output value. Filters are used e.g. for data transformations and to specify the fields in virtual data objects. An output from a filter could be any value, and is referenced by an index similar to that of a data object. The specified inputs to a filter is, with few exceptions, an ordered set of filters. Exceptions include filters that reference data objects, external data ports, and most importantly arbitrary procedures.

4.5.1. General filter functionality

`(make-filter` *proc*`)` procedure
`(make-filter` *proc fformat*`)` procedure

This procedure will return a new filter using the procedure *proc* for determining the returned value for a specified index. The procedure should take exactly two arguments, the first being the calling filter object and the second the current index. Use e.g. the procedures `inner-ref` and `inner-ref0` to ease the construction of simple filter functions.

If the optional argument *fformat* is given as a field format, the format of the output of the filter will be considered to have this format. Otherwise, the format of the output will be assumed to be that of the first inner filter.

`(filter?` *obj*`)` procedure

The procedure `filter?` returns `#t` if and only if *obj* is a filter, otherwise `#f` is returned.

`(filter-get-inner` *filter*`)` procedure

`filter-get-inner` returns the input filters, or *inner* filters, of *filter* as a list. If *filter* has no inner filters, the empty list is returned.

`(filter-set-inner!` *filter inner*`)` procedure

The procedure `filter-set-inner!` sets the inner filters (or input filters) of *filter* to *inner*, where *inner* can be a list of filters or one single filter. Note that the inner filters may be ignored, depending of the type of filter.

`(filter-discr?` *filter*`)` procedure
`(filter-cont?` *filter*`)` procedure
`(filter-string?` *filter*`)` procedure

The procedures `filter-discr?`, `filter-cont?`, and `filter-string?` returns `#t` if and only if *filter* may return discrete, continuous and string values respectively. Otherwise, `#f` is returned.

`(filter-ref` *filter i*`)` procedure

The `filter-ref` procedure returns the value of *filter* at index *i*. Two consecutive calls to the procedure for the same index may not necessarily return the same value.

`(filter-inner-ref` *filter i*`)` procedure
`(filter-inner-ref0` *filter i*`)` procedure

The procedure `filter-inner-ref` returns a list of the values of the inner filters of *filter* at index *i*. The procedure is mostly usable when implementing simple filter functions. The procedure `filter-inner-ref0` returns the value of the first inner filter at index *i*, useful for implementing filter functions for filters that will take only one input.

`(filter-entries` *filter*`)` procedure

Similar to data objects, filters can also be queried about the total number of entries that can be indexed. This is performed using the procedure `filter-entries`. Note that for a filter, the total number of entries that can be indexed might be infinite, in which case the symbol `'...` will be returned.

4.6. Pre-defined filter types

gmdl provides a number of pre-defined filter types, usable in e.g. exploratory data analysis or for transforming data objects.

Non-recursive filter types

Non-recursive filters are filter that do not use other filters as inputs. These filters are usually used as the base source of data for a chain of filters, and can be used to interface with e.g. data objects or random number generators.

`(make-filter-data` *data field*`)` procedure

Creates a filter that accesses a data object *data* at field *field*. The number of entries as well as the first and last accessible entry of the filter will reflect the properties of the data object.

Single input filter types

Single input filters only use a single input, i.e. they only consider the first inner filter when calculating the output.

`(make-filter-log)` procedure

Creates a filter that returns the natural logarithm of the first inner filters value.

`(make-filter-exp)` procedure

Creates a filter that returns the exponent of the first inner filters value.

`(make-filter-add` *val*`)` procedure

Creates a filter that returns the first inner filters value added by *val*.

`(make-filter-multiply` *val*`)` procedure

Creates a filter that returns the first inner filters value multiplied by *val*.

`(make-filter-power` *val*`)` procedure

Creates a filter that returns the first inner filters value to the power of *val*.

`(make-filter-diff)` procedure

Creates a filter that returns the difference between the current value of the first inner filters current value and its previous value, $y_i = x_i - x_{i-1}$.

`(make-filter-threshold` *t dir*`)` procedure
`(make-filter-threshold` *t dir goal*`)` procedure

The procedure `make-filter-threshold` creates a filter where all values of the first inner filter that are equal to, above or below the specified threshold t are substituted with `#?`, while all other values are returned as they are. The argument *dir* should be a symbol `above` or `below` which specifies whether the filter should remove values above or below the threshold. If the additional argument *goal* is provided, the filter will substitute all values that fall outside the threshold with *goal* instead of `#?` as long as *goal* is a real number. *goal* can also be provided as the symbols `threshold` or `remove`, in which case the filter will substitute all values that fall outside the range with the threshold itself or `#?` respectively.

`(make-filter-run-avr` *n*`)` procedure

Create a filter that returns the mean of the n latest values (the current included) of the first inner filter.

`(make-filter-run-median` *n*`)` procedure

Create a filter that returns the median of the n latest values (the current included) of the first inner filter.

`(make-filter-linear` *w*`)` procedure

Create a filter that returns the weighted sum of the latest values (the current included) of the first inner filter. The list w specifies the weights used, where the weights are given in reverse order, starting with the weight for the current value.

`(make-filter-gamma` γ`)` procedure

Create a gamma filter, i.e. an exponential filter where the output value $y_i = \gamma x_i + (1 - \gamma)x_{i-1}$ and x_i, x_{i-1} are the current output of the first inner filter and its previous value respectively.

`(make-filter-back-gamma` γ *n*`)` procedure

Create a filter similar to the gamma filter, but in the reverse direction. γ specifies the ratio and n the number of indeces ahead that the filter considers for its calculation of the current value.

`(make-filter-highpass-run-avr` *n*`)` procedure

Creates a filter that returns the pairwise difference between the n last outputs of the first inner filter, $y_i = x_i - x_{i-1} + x_{i-2} - \ldots - x_{i-n-1}$. This creates a highpass filter effect. Note that n should be an even number for the filter output to have the same mean as the input.

`(make-filter-subtract-run-avr` *n*`)` procedure

Create a filter that returns the current value of the first inner filter subtracted by its running average over the n latest outputs.

`(make-filter-negate)` procedure

Creates a filter that negates the output of the first inner filter.

`(make-filter-delay` *d*`)` procedure

Creates a filter that delays the output of the first inner filter by d steps, $y_i = x_{i-d}$. Note that d can be a negative value as well as positive.

`(make-filter-reverse` *f*`)` procedure

Create a filter that reverses the output of the first inner filter around the fixpoint f, $y_i = x_{(2f-i)}$.

`(make-filter-interpolate)` procedure

Create a filter that substitutes all unknown values in the first inner filter by the linear interpolation between the last previosly known to the next known value.

`(make-filter-repeat)` procedure
`(make-filter-repeat` *n*`)` procedure

Create a filter that substitutes all unknown values in the first inner filter by repeating the last previously known value. If the additional argument n is provided, the filter will only repeat the last known value a maximum of n times until it starts returning unknown values.

`(make-filter-replace` *type*`)` procedure
`(make-filter-replace` *type n*`)` procedure

Create a filter that will replace all unknown output values in the first inner filter with a specified value. If *type* is a simple value, e.g. a discrete or real number, all unknown outputs will simply be replaced by this value. If *type* is the symbol `mean`, all unknown values will be replaced with the mean of the output of the first inner filter. Similarily, if *type* is the symbol `median`, all unknowns will be replaced by the median of the filter. If *type* is the symbol `interpolate`, the filter will substitute all unknown values in the first inner filter by the linear interpolation between the last previosly known to the next known value. In the same manner, if *type* is `repeat`, then the filter will substitute all unknown values in the first inner filter by repeating the last previously known value. Provided with the additional argument n, the filter will only repeat the last known value a maximum of n times until it starts returning unknown values.

`(make-filter-compact)` procedure

Creates a filter that replaces each unknown output of the first inner filter with the next non-unknown output from the same filter, only using each non-unknown output of the first inner filter once. The output is a compacted representation of the inner filter where all unknown values have been removed.

`(make-filter-remove-range` *begin end*`)` procedure

Creates a filter that returns `#?` for all indeces between *begin* and *end*, and otherwise the output of the first inner filter.

Multiple input filter types

Multiple input filters use one or more input filters to produce their output.

4.7. Data preparation and scripting

When preparing data for statistical analysis it is frequently useful to incrementaly construct a sequence of transformations of one or more data sources to produce data on a form suitable for the analysis. The scripting functionality is intended to support this process by providing a scripting language, a number of high level operations and library routines that can be easily adapted to handle a large class of preprocessing needs.

Most of the high level operations takes a data object as input and returns a virtual data-object as result. This makes the operations clean in the sense that the orignial data is always unchanged and for most operations it also makes the transformation very fast. On the other hand accessing the data in the resulting data-object may be slow. For this reason an explicit check point mechanism has also been implemented that will transform the virtual data-object to a compact data-object and optionally write a representaion of it to disk.

This means that, as you incrementaly construct your script, you can commit the result of executing stable parts of the script to disk and re-execute the parts of the script to generate the check point files only when really neccessary.

The scripting mechanism include a compiler and a top level interpreter, which handles the check point mechanism and provide abstractions of the primitive operations as commands where the input and output data objects are implicit. Please note that, when used in the context of a script, the virtual-data procedures (`virtual-data-consolidate`, `data-sort`, `data-match`, `data-replace`, `data-derive`, `data-select` and `data-merge`) in this section should be used without their first argument which is implicit. The script specific commands (`script-read`, `script-display`, `script-check` and `script-write`) and utility procedures take arguments as specified below.

4.7.1. Scripting reference documentation

`(define-script` *script-name body*`)` macro
`(define-script` *(script_name formals) body*`)` macro

`define-script` creates a new script with the name *script_name*. This script should be applied to a file specification which is either a string containing a path name or a list where the first argument is such a string and the second is the size of a sample from that file. The third element of the list, if present, should be an offset from the start of the file. Additional arguments to the script will be accessible through the formals of the script. Note that the formals may have to be unquoted (with ",") since the script is compiled through macro expansion.

The first command in the script must always be a `script-read` which will produce the initial data object operated on by the remaining commands in the script. See below for the semantics of the `script-read` command.

Note that in the script, the data objects produced by and occurring as first argument to the commands described below are implicit, as is the naming of check point files and samples.

Applying the script to a file specification will create a data object from it and return a data object representing the result of executing all the operations encoded by the commands of the script.

All the commands will be executed in sequence unless the script contains one or more `script-check` commands, in which case some of the commands will be executed only if the corresponding check point file does not exist or a regeneration of the check point is forced by the call to `script-check`. See below for the semantics of the `script-check` command.

`(script-read` *. format_specs . force*`)` script command
`(script-read` *file_spec . format_specs . force*`)` procedure

The `script-read` command returns a compact data object from the file specified by $file\_spec$. It does this by a sequence of (optional) operations as follows:

1. Guess the data format of the data file specified by $file\_spec$ if the format file corresponding to it does not already exist, or if the command is called with more than two arguments.
2. Modify the guess using $format\_specs$ which, if present, should be a list of lists where each element in the outer list starts with a field format specification and the rest of the list are the columns which should have that format. The field format specification format is described under `make-fformat` below.
3. Create a sample from the data file if $file\_spec$ is a list with more than one element. If so the second element is expected to be an integer specifying the size of the sample. The third element, if present should be an integer that specifies an offset from the start of the specified file to the start the sample.
4. Create a compact data object by reading in the data file specified by $file\_spec$.

`(script-display` *row column*`)` script command
`(script-display` *dobj row column*`)` procedure

Display content of a field in data object. The external epresentation of the datum will be used. Useful for debugging.

`(script-check` *. force*`)` script command
`(script-check` *d prefix file_spec version . force*`)` procedure

The `script-check` command returns a data object by either reading in the check point file specified by $file\_spec$ and *version* if it exists or by executing $prefix$ as a script. In the latter case it will also save the resulting data object as a new check point file.

Note that all the arguments of this command (except the optional $force$) are implicit when it occurs in a script. The script compiler will generate both the prefix and version and compute the name of the check point file based on the file specification and version. In the context of a script the command will be called with a prefix compiled from all commands occurring before it in the script. The last argument, if present will force the regeneration of the check point file even if it exists.

`(script-write` *file_name* `)` script command
`(script-write` *d prefix file_name version . force*`)` procedure

The `script-write` is used to save the the input data object to disk using a named file. If the the script was called with a sample specification the given name, $file\_name$, will be extended as in the naming scheme of the `script-check` above.

`(virtual-data-consolidate)` script command
`(virtual-data-consolidate` *virtual_dobj*`)` procedure

The `virtual-data-consolidate` commands creates a new compact data object in which the result of the executing all filters in the virtual data object supplied to it as input is explicitly stored.

This operation should in most cases be called before a check point is created but may be useful also e.g. before a sort, select or merge operation that will access many elements of a virtual data object and where the on-demand execution of the filters is expensive.

After the consolidation is complete all pointers to the input data object held by the script are released and the garbage collector called.

`(data-sort` *cmp columns*`)` script command
`(data-sort` *dobj cmp columns*`)` procedure
`(data-stable-sort` *cmp columns*`)` script command
`(data-stable-sort` *dobj cmp columns*`)` procedure

Sort data lexically on each of the *columns*, using *cmp* as a binary comparison operator. `data-stable-sort` is guaranteed to be stable, while `data-sort` warns of duplicate values.

`(data-replace` *fn frmt cdmn*`)` script command
`(data-replace` *dobj fn frmt cdmn*`)` procedure

Return a virtual data object cloned from *dobj* but in which the values all fields in the columns specified in *cdmn* have been mapped to new values of a corresponding new field format specified by *frmt*. *fn* is the mapping function, *frmt* a field format specification and *cdmn* a list of columns to apply the transform to.

The transform function takes exactly three arguments, an abstraction of the original data object (see the `abstr` function below for details), a row and a column index.

The field format specification *frmt* format is described under `make-fformat` below.

`(replace-filter` *fn . rgs*`)` procedure

`replace-filter` is a simple generic wrapper function for replace. The function will return a function of the type expected by `data-replace` that will apply the function *fn* to the internal representation of the fields specified in the replace command and the additional arguments in *rgs* interpreted as being of the same field format that of the designated field.

fn must be a function taking one plus the length of *rgs* arguments. The the function returned by `replace-filter` will return `#?` whenever *fn* returns `#f` and the return value of the call to *fn* otherwise.

`(data-derive` *fn frmt cdmn*`)` script command
`(data-derive` *dobj fn frmt cdmn*`)` procedure
`(data-insert` *fn frmt cl cdmn*`)` script command
`(data-insert` *dobj fn frmt cl cdmn*`)` procedure

The `data-derive` command returns a virtual data object cloned from *dobj* and to which a new column has been added. The values of the fields in the new column are derived by a function *fn* that is applied to an abstraction of the original data object *dobj* (see the `abstr` function below for details), a row index and the column indexes given in *cdmn*.

The `data-insert` command returns a virtual data object into with a new column has been inserted at the position given by *cl*. The column indexes in *cdmn* refer to those in the original data object even when the new column is inserted before one of those in *dobj*. All columns in the original data object larger than or equal to *cl* will be displaced one step in the virtual data object returned. The remaining arguments are the same as for `data-derive`.

The effect of `data-insert` can be emulated by adding the new column at the end with `data-derive` and specifying the corresponding permutation of the columns with e.g. `data-select`.

`(data-match` *c . frmts*`)` script command
`(data-match` *dobj c . frmts*`)` procedure

The `data-match` command returns a virtual data object cloned from *dobj* and to which one or more new fields has been added. It does this by matching the contents of the given field *c* to the strings enumerated by the (discrete) field formats of each new column which are specified by *frmts*. Each string in the discrete data format is interpreted as a regular expression when the match is computed. See section 4.7.2 for an example.

`(match` *strings*`)` procedure

`match` can be used in replace and derive operations to similar effect as the `data-match` command above. Match returns the internal representation of the matched string in a discrete field format where the range is defined as the enumeration *strings* or 0 if no match is found.

`(data-select` *tstfn cdmn*`)` script command
`(data-select` *dobj tstfn cdmn*`)` procedure

The `data-select` command returns a virtual data object cloned from *dobj* but in which only the columns given in *cdmn* and rows that fulfils the test function *tstfn* has are present. The test function should take exactly two arguments: an abstraction of a data object (see the `abstr` function below for details) and a row index.

The column indexes in *cdmn* need not be ordered. However, expansion of domain expressions occurring in *cdmn* is unreliable if they are not.

(`data-merge` *db_0 db_1 ccs vfn ffn n cdmn . rgs*) script command

(`data-merge` *db_0 db_1 ccs vfn ffn n cdmn . rgs*) procedure

The `data-merge` command merges data from two datasources, db_0 and db_1. One or more fields are derived from db_1 are and added to a new virtual data object cloned from db_0.

First, a match is generated. The match will consist of the indexes of the entries in db_1 that matches each row in db_0 where a match occurs when the entries of db_0 and db_1 are equal (`=`) at each of the column pairs given in *ccs*. Correctness of the matching algorithm *requires* each input data objects to be *sorted* on the columns given in *ccs*.

The remaining arguments of `data-merge` must occur in multiples of four where each quadruple vfn, ffn, n, *cdmn* specifies how to derive additional columns. For each column index c specified by *cdmn*, n new columns will be derived. The value function vfn computes the value of each new field while the format function ffn computes the format of each new column.

The value function vfn should take four arguments: an abstraction d of the data object db_1 (see the `abstr` function below for details), a list of row indexes *rws*, which represents the indexes in db_1 matched to the current row in db_0, a column index c in db_1 and integer i between 0 and $n-1$.

This is typically used to sample or enumerate fields in consecutive entries in db_1 to create several new fields in the resulting data object. E.g. to transpose the vector of values of a field in several matched entries into a number of new fields, a value function

```
(lambda (d rws c i) (d (list-ref rws i) c)
```

could be used. See section 4.7.2 for further examples.

Furthermore a number of useful and convenient merge function are provided by the scripting module. See e.g. `resample` and friends below.

The format function should be a function of five arguments: a column index c, an integer i between 0 and $n-1$ as above and a field format type *tp*, domain *dmn* and name *nm* respectively which when called for each new column will be instantiated with the field format parameters of the column c of db_1.

This is useful for deriving the new field format as a function of the format of the column from which its value is derived. The function should return a list of up to three arguments to which the `make-field-format` can be applied. E.g. to use a unique version of the original field name one could use the format function:

```
(lambda (c i tp dmn nm)
 (list tp dmn (format #f " a- a" nm i)))
```

The format function will be called once for each new column as the columns are generated.

(`resample` *d rws s op l fn . cdmn*) merge utility procedure

A generic sampling utility function for `data-merge`. The function takes at least six arguments where d and *rws* are as in the call to `data-merge` above, s is a column index in db_1, *op* a binary comparison operator and l a value in the domain of the column in db_1. fn should be a sampling function of the type described below and the remaining arguments are further column indexes in db_1.

The semantics of the procedure is as follows:

1. Read the entry indexes in *rws* and apply *op* to the field s of each row and the value l until it returns a value different from $\#f$. Let this index be call *rw*, *prv* be the list of indexes preceding and *nxt* the list of indexes following it in *rws*.
2. Return the result of applying the function fn to d, l, *prv*, *rw*, *nxt* and the column indexes in *cdmn*.

Typically this is used to search the values of a particular column s for a value that exceeds the value l in which case the column must be sorted on s at least in range of entries given by *rws* for a binary operation such as $>=$.

The sample function fn typically interpolates between consecutive values of the fields in *cdmn* using e.g. linear interpolation, gradient, derivatives or splines. The intention of supplying the prefix *prv* and suffix *nxt* of the selected row *rw* in *rws* is that some sample functions (e.g. splines) may require more than two entries to compute their value.

Two common sample functions are implemented as library routines. See `interpolate` and `gradient` below.

(`interpolate` *d k prv rw nxt x y*) merge utility procedure

Return a (linearly) interpolated value of x at position k in y. `interpolate` could be imlemented as follows:

```
(define (interpolate d k prv rw nxt x y)
  (if (null? prv) #?
      (- (d rw x)
         (* (- (d rw x) (d (car prv) x))
            (/ (- (d rw y) k)
               (- (d rw y)
                  (d (car prv) y)))))))
```

`(gradient` *d k prv rw nxt x y*`)` merge utility procedure

Return gradient of *x* with respect to *y*. `gradient` could be interpreted as follows:

```
(define (gradient d k prv rw nxt x y)
  (if (null? prv) #?
      (/ (- (d rw x) (d (car prv) x))
         (- (d rw y) (d (car prv) y)))))
```

`(abstr` *dobj*`)` scripting utility

Create an abstraction that is a function that when applied to exactly two arguments *row* and *column* it returns the *internal* representation of the data in the referred field.

If the abstraction is called with three or more arguments, the third argument should be a function that will be applied to the content of the field referred to by the first two arguments and the *internal* representation of all arguments following the third, interpreted by the same format as the one of the field referred to by the first two arguments.

`(expand-dom` *(domain_expression . (upper_bound))*`)` scripting utility

`expand-dom` takes a domain expression and optionally an upper bound and expands it to a list of all integers denoted by the expression.

A domain expression is in itself a list of intervals where each interval is either a single integer or a pair of a lower and upper bound. The lower bound i and integer and the upper bound either and integer or the symbol ... in which case it is interpreted the optional argument *upper_bound* must be present and substituted for ... in the expression before expansion.

`(lex` *binary_test test_pairs . (equality_return_value)*`)` scripting utility

General lexical ordering with optional value to return in equality case.

`(access` *function*`)` scripting utility

Transforms a function expecting a one or more values as arguments into a function which takes a data object abstraction (see the `abstr` function above for details), a row index and one or more column indexes as arguments. When applied the transformed function will access the values from the data object abstraction and apply the original function to the resulting *values*.

This is useful since most of the scripting commands expect a function of the transformed type as arguments but for simple case we man want to use a primitive operation in our script. For examples see section 4.7.2.

`(make-fformat` *frmt . (lbl)*`)` scripting utility

Create a new field format using the field format specification *frmt* which is either one of *int*, *cont*, *date*, *time*, *string* or a list where the first element is a base type, i.e. either one of the above or *disc*. The following elements of the list constitute a specification of the range of the format and optionally a string containing the name used to refer to the field. The range specification format varies with base type. See `make-fformat-int`, `make-fformat-cont` `make-fformat-disc` etc. for details.

4.7.2. Brief tutorial and usage examples of the scripting mechanism

Example 1 — Script structure, replace, select, sort and consolidation

```
(define-script (example1 SaveName)
  (script-read
   '(((int (0 . 999)) (0 . 3))
     (string 4 5 6)))
  (data-replace
   (lambda (d r c)
     (if (d r c string=?      ; Unknown
           "1858111700000000") #?
         (string-append
          (d r c substring 0 4) "-"
          (d r c substring 4 6) "-"
          (d r c substring 6 8) ","
          (d r c substring 8 10) ":"
          (d r c substring 10 12) ":"
          (d r c substring 12 14))))
   'date '(4 5 6))
  (data-replace
   (lambda (d r c)
     (if (d r 0 = 2002)
         (if (d r c >= 50)  (d r c + 600)
             (d r c + 700))
         (if (and (d r 1 <= 8) (d r c >= 30))
             (d r c + 700) (d r c + 800))))
    '(int (0 . 999)) '(2))
   (data-replace
    (replace-filter >= 0)
    '(cont (0 . 2000)) '(8))
   (data-replace
    (replace-filter >= 0)
    '(cont (0 . 21000)) '(9))
   (data-select
    #t '(2 3 4 (8 . 12) 14 (16 . 19) 21 23))
   (virtual-data-consolidate)
   (data-stable-sort <= '(0 1 2))
   (script-write ,SaveName)
  ))

; (example1 "<source path>" "<destination path>")
```

This script will first check if the file `"<source path>.frmt"` exists, and if not generate it by analysing the file "<source path>.data", enforcing interpretation of fields 0..3 as integers in the range 0..999 and fields 4..6 as strings. Then "<source path>.data" will be read using the format "<source path>.frmt" to create a compact data object which will then be passed to the next command in the script.

The next command (data-replace) will parse the fields 4..6 as strings and in each case return a new string which will be interpreted as a date. Note the use of the data object abstraction d in both the test (d r c string=? "1858111700000000") and the the extraction e.g. (d r c substring 0 4) where r and c are row (entry) and column (field) indexes respectively.

The following replace command recomputes the most significant digit in field 2 which has accidentally been deleted in the data using a year and a month field 0 and 1 respectively. None of these transforms are very interesting in themselves but illustrates nicely the generality of the replace mechanism.

The next two replace commands uses a primitive filter function replace-filter which replaces all values in fields 8 and 9 that are less than or equal than 0 with the unknown value #?.

The next command selects the columns 2..4, 8..12, 14, 16..19, 21 and 23 from the original data. Since the first argument is #t no selection of entries will be performed.

The result of all operations so far is then consolidated as a new compact data object (releasing all pointers to all data objects created before this point in the script) using virtual-data-consolidate. This is useful for two reasons. First, if the the selection is significantly smaller than the original data, the memory occupied by the larger object will be released. Second, since all operations are done on virtual data the values of transformed and added fields will be computed at each access. In cases where you expect to access each fields many times it is therefore useful to first consolidate the data so that access is simply a lookup operation.

The resulting data object is then sorted and saved to disk using the name supplied as argument to the script. Note that this argument needs to be unquoted since the the script is implicitly quoted by the macro expansion of define-script.

Example 2 Derived fields, check points and simple merge

```
(define-script (example2 SRC2 SRC3 SaveName)
  (script-read)
  (data-derive (access -)
   '(int (0 . 172800) "ct0-ch") '(71 29))
  (data-match 85
   '(disc ("No" "vb" "vb.i.st" "vb.i.f.*p") "VB")
   '(disc ("No" "hasp.*680" "hasp.*750") "HT"))
  (virtual-data-consolidate)
  (data-sort <= '(0 1))
  (script-check)
  ;; Merge source 2
  (data-merge
   (src2 ',SRC2)           ; Subscript
   '((0 . 0)(1 . 1))       ; Match fields
   (lambda (d rws c i)     ; Value function
    (d (list-ref rws i) c))
   (lambda (c i tp dmn nm) ; Format function
    `(cont (0 . 100) ,nm))
   1 '((2 . 29)))       ; Samples & source fields
   ;; Merge source 3
  (data-merge
   (src3 ',SRC3)  '((0 . 0)(1 . 1))
   (lambda (d rws c i)
    (if (null? rws) "No" "Yes"))
   (lambda (c i tp dmn nm)
    '(disc ("No" "Yes") "Sliver"))
   1 '(0))

  (script-write ,SaveName)
  )

; (example2 "<src1>" "<src2>" "<src3>" "<dst>")
```

This example illustrates the use of derived fields, the check point mechanism and some simple merge operations.

The data-derive command computes a time duration from two date fields (71 29). The access function transforms the primitive - arithmetic operations into function that first access the data in the supplied fields using the same data object abstraction as in the replace examples above. This means that the the arithmetic operations will be applied to the internal representation of the dates. The result will be interpreted according to the supplied field format specification '(int (0 . 172800).

The data-match command will match each case of the discrete (enumerated) field format specification against the content of field 85 and add a new field with that discrete type and the value of the corresponding match. Note that the first case in the discrete type will be used when no match is found and should therefore be chosen as a value that does not occur in field 85.

Next, the data produced so far is consolidated, sorted and then saved to disk as a check point file. The next time the script is run the existence of this file is checked. If the file exists and unless the check command contain additional arguments, the commands preceding the check command is ignored and the data is instead read from disk. This makes sense when parts of the script are time or memory consuming and not under active development.

The next two commands in the script are both data-merge. The merge command is the most complex of the operations considered so far but these are very simple cases. The first argument to the merge command is secondary data object (the primary is implicit). Here that data object is generated by a call to other scripts (`src2` and `src3`). Note the the source files to this scripts are given as arguments to the main script (`example2`).

The second argument should be list of pairs which indicate how entries in the primary data object should be matched to entries in the secondary. The match in both of these cases will be computed by identifying entries which has identical values in columns `0` and `1`.

The third argument should be a function that for each merged field computes a value at each entry. In the first case the value function simply returns the value of the field in the secondary data object of the entry at the first row in the matched rows. In the second case the value function returns the internal representation of the discrete value No if the match is empty and of Yes otherwise.

The fourth argument is the format function which in the first case will return a continuous format with values in the range 0..100 and with the same name as in the original data field. In the second case the field format is discrete with the same two possible values as those returned by the value function.

The last two arguments are the number of samples derived from each field in the secondary data object which in both these cases are `1`, and finally the source columns in the secondary data object. In the first case these are all columns in the range 2..29 and in the second case an arbitrary dummy column since this value is not used by the corresponding value function.

Example 3 — A more complex merge

The following example illustrates a more complex merge operation with a two distinct types of sampling of the data in the secondary data object. In both case a fixed number of samples (12) are computed from a variable number of matched entries in the secondary object. Both use the library procedure `resample` but with different sample functions, `interpolate` and `gradient`. Most of the data for which the gradient data is computed is also sampled by interpolation.

```
(data-merge (frnc ',Frnc) '((0 . 0)(1 . 1))
 (lambda (d rws c i)
  (resample d rws 5 >= (* (++ i) 100)
   interpolate c 5))
 (lambda (c i tp dmn nm)
  `(,tp ,dmn ,(format #f " a at  aC"
                       nm (* (++ i) 100))))
 12 '(3 4 (6 . 14))
 (lambda (d rws c i)
  (resample d rws 5 >= (* (++ i) 100) gradient c 3))
 (lambda (c i tp dmn nm)
  `(cont ,(if (= c 4) '(0 . 300) '(0 . 30))
         ,(format #f " a incr/min at  aC"
                     nm (* (++ i) 100))))
 12 '(4 5 7 8 9 12 13 14))
```

The value function of the first set of samples simply calls the resample library routine with the data object `d`, the matched rows `rws`, the index `5` of a column to scan for sample points, a binary relation `>=` and a limit `(* (++ i) 100)` indicating at what value of column `5` to compute the sample. The next argument of the `resample` function is a function to compute the value of the sample, the column to sample and the index of a column with which to sample with respect to. In this case the functions is `interpolate` and the column is the same as the scan column which means that we will compute the interpolated value of column `c` at the point where column `5` reaches the value `(* (++ i) 100)`. Since the function will be called for each value of `i` in 0..12 we will generate samples representing the state of each column in `'(3 4 (6 . 14))` at values $100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200$ of column `5` respectively.

The format function returns the same format as that of the sampled column but with the name extended with a representation of the sample point.

The value function in the second case is similar to the first, the only difference being that a gradient is used instead of the interpolated value. The sample points are identical but the gradient is instead computed with respect to column `3` which contains time stamps. This means that the gradient value represents a time derivative of the value of the sampled columns, something which is frequently useful.

The format function in the second case is a little more involved than in the first since the range derivative is not a simple function of the range of the measured value. Here we have inspected the generated data to confirm that the derivatives of the values of column `4` which encodes a distance are in the range 0..300 and all other sampled columns which encodes temperatures, in the range 0..30. Again the names of the column are extended to illustrate the type of value it contains.

Enjoy!

5. Functions, models and distributions

5.1. Functions

A numerical function in gmdl operate on one or multiple values of one or several numerical types, depending on the domain and intended dimensionality of the function. Similarly, the returned value can be of any dimensionality and numerical type. The dimensionality of a numerical function is allowed to be either fixed or flexible, in which case

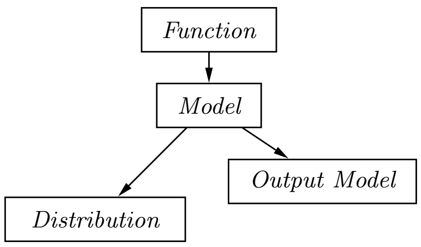

Figure 5.1: Functions, models and distributions.

the dimensionality of the output usually is either fixed or depends on the dimensionality of the input.

5.2. Models

A model in gmdl is any parameterized (or parameter free, for that matter) model of data. A model contains a reference to a data format and an ordered set of fields within this format which the model operates on. A number of procedures are useful for almost all kinds of models, e.g. estimating the model from a certain data set and predicting the unknown values in a data vector. These procedures are always available for all models but the way they operate depends heavily on the type of model used. For some models, some procedures might not actually perform any operations at all. Below is a description of these general model functions and a number of specific models available in gmdl.

5.2.1. General model functionality

These procedures operate on all models.

`(model?` *obj*`)` procedure

The procedure `model?` returns `#t` if and only if *obj* is a model, otherwise `#f` is returned.

`(model-estimate?` *mdl*`)` procedure
`(model-set-estimate!` *mdl v*`)` procedure

All models have a boolean parameter deciding if the model should actually be estimated when requested to. The default value for a model is usually `#t`. The main use of setting the parameter to `#f` is if the model is part of another model that could request an estimation, but the parameters of the model should actually be fixed to their current value. `model-estimate?` returns the current value of the estimation parameter for *mdl*. To set the parameter to the boolean value *v*, `model-set-estimate!` is used.

`(model-dformat` *mdl*`)` procedure
`(model-field` *mdl n*`)` procedure
`(model-max-field` *mdl n*`)` procedure
`(model-dim` *mdl i*`)` procedure

The procedure `model-dformat` returns the data format for the model *mdl*. `model-field` returns the field that dimension *n* in the model operate on. `model-max-field` returns the maximum field index for the model. `model-get-dim` returns the first dimension in the model for which the field index *i* is used. If the field index is not used in the model, `#f` is returned.

`(model-val` *mdl n vals*`)` procedure
`(model-any-unknown?` *mdl vals*`)` procedure
`(model-all-unknown?` *mdl vals*`)` procedure

The procedure `model-val` returns the value of the model *mdl*'s *n*:th dimension in the data vector *vals*. `model-any-unknown?` returns `#t` if any of the fields used in the model *mdl* is `#?` in the data vector *vals*, otherwise `#f` is returned. Similarly, `model-all-unknown?` returns `#t` if all fields used in *mdl* are `#?` in *vals*, otherwise `#f` is returned.

`(model-get-fields` *mdl*`)` procedure

The procedue `model-get-fields` returns a list of the fields that the model operates on in order.

`(model-same-fields?` mdl_1 mdl_2`)` procedure
`(model-superset-fields?` mdl_1 mdl_2`)` procedure
`(model-subset-fields?` mdl_1 mdl_2`)` procedure

`model-same-fields?` returns `#t` if the models mdl_1 and mdl_2 operate on the same fields, otherwise `#f` is returned. In the same manner, `model-superset-fields?` and `model-subset-fields?` return `#t` if and only if the fields of mdl_1 is a superset or subset of the fields of mdl_2 respectively. For these latter two procedures, mdl_2 may also given as a list of fields.

`(model-predict` *mdl vals*`)` procedure

The procedure `model-predict` is mainly used to predict or classify unknown data. All models provide this procedure, but exactly how it operates depends on the type of model. It always returns a vector of values, that is usually a copy of *vals* where the unknown values of the vector have (if possible) been substituted with the output of the model.

`(model-estimate!` *mdl dobj*`)` procedure
`(model-estimate!` *mdl dobj prob*`)` procedure

`model-estimate!` estimates the model *mdl* from the data object *dobj*. The optional value *prob* provides the estimation procedure with a probability or relative importance for each entry in the data object. *prob* must be a vector of the same size as the data object, where each element is a continuous value between zero and one.

`(model-display` *mdl*`)` procedure
`(model-display` *mdl port*`)` procedure

`model-display` displays information about the model *mdl* on port *port*. If no port is given, the current output port is used.

`(model-write` *mdl file*`)` procedure
`(model-read` *mdl file*`)` procedure

The procedure `model-write` writes the model *mdl* and all objects it depends on to the file specified by the string *file* in a machine readable format. `model-read` returns the model specified in the file *file*, which must be on the same format as produced by the procedure `model-write`.

5.3. Testing and evaluating models

Testing and evaluation of models is an essential part in data model development. gmdl provides data types and procedures to make this task easier.

5.3.1. The test results type

Test results in gmdl can be stored in a specific data type to make testing and evaluation easier. The data stored in the test results object data type is referenced using a set of symbols, where each symbol relate to a certain test result by convention. The symbols are:

`name`	`Name of test`
`field-name`	`Name of tested field`
`model`	`Description of model`
`description`	`Description of test`
`date`	`Date of test`
`patterns`	`Number of patterns`
`unknowns`	`Number of unknowns`
`exact-match`	`Number of exact matches`
`res-mean`	`Result mean`
`true-mean`	`True mean`
`res-var`	`Result variance`
`true-var`	`True variance`
`covar`	`Co-variation`
`corr-coef`	`Correlation coefficient`
`rms`	`The root mean square error`
`mean-absolute`	`The mean absolute error`
`rrs`	`The relative squared error`
`relative-absolute`	`The relative absolute error`

All numerical values default to zero, and string values to the empty string. The date defaults to the current date and time as supplied by the system.

`(make-test-results)` procedure

Constructs a new test result. All parameters are initialised to their default values.

`(test-results?` *obj*`)` procedure

The procedure `test-results?` returns `#t` if and only if *obj* is a test result object, otherwise `#f` is returned.

`(test-results-ref` *tr param*`)` procedure
`(test-results-set!` *tr param val*`)` procedure
`(test-results-clear!` *tr*`)` procedure

The procedure `test-results-ref` returns the value of the parameter *param* in the test results object *tr*. `test-results-set!` sets the value of *param* to *val* in *tr*. In both procedures, *param* must be a valid symbol (see above). `test-results-clear!` initialises all parameters in *tr* to their default values.

`(test-results-display` *tr*`)` procedure
`(test-results-display` *tr port*`)` procedure
`(test-results-write` *tr file*`)` procedure
`(test-results-append` *tr file*`)` procedure

`test-results-display` displays the (relevant) parameters and their current value in the test results object *tr* on port *port*. If no port is specified, the current output port is used. Parameters that e.g. do not have any value may be left out when the parameters are displayed. `test-results-write` writes the test results object *tr* to the file specified by the string *file*. In a similar manner, `test-results-append` appends this information to the end of an already existing file. If no such file exists, it will be created by the procedure if possible.

5.3.2. Test and evaluation procedures

`(model-test` *mdl data fields*`)` procedure
`(model-test` *mdl data fields all*`)` procedure
`(model-test` *mdl data fields all rdata*`)` procedure
`(model-test` *mdl data fields all rdata act*`)` procedure

Test a model *mdl* on the data set *data*. The argument *fields* is a list specifying which fields that will be tested. *all* is an optional boolean variable specifying whether the model should try to predict all values of the fields in *fields* for all instances, even if the correct value is unknown for the instance. The default value is `#f`. The optional argument *rdata* should be a mutable and adaptable data object in which the predicted results of the model will be stored. If *rdata* is the symbol `'create`, then a data object to store the results will be created and returned. The procedure returns a list of test results, each element corresponding to the test results of each field in *field*, unless the argument *rdata* is `'create`. In this case the procedure will return a list containing a list of test results as its first element and a data object containing the results in the second. If the optional argument *act* is `#t`, two columns for each tested field will be used in the resulting data object; the first containing the actual value and the second the predicted. The defualt behaviour is to only write the predicted values to the data object. Note that relative errors cannot be estimated.

`(model-test!` *mdl edata data fields*`)` procedure
`(model-test!` *mdl edata data fields all*`)` procedure
`(model-test!` *mdl edata data fields all rdata*`)` procedure
`(model-test!` *mdl edata data fields all rdata act*`)` procedure

Works like `model-test`, except for the fact that the model is estimated on the data object *edata* before testing. The relative errors are estimated by this procedure.

`(model-cross-validate!` *mdl d f*`)` procedure
`(model-cross-validate!` *mdl d f n*`)` procedure
`(model-cross-validate!` *mdl d f n a*`)` procedure
`(model-cross-validate!` *mdl d f n a rd*`)` procedure
`(model-cross-validate!` *mdl d f n a rd act*`)` procedure

Cross-validate a model *mdl* using the data object *d*. The argument *f* is a list specifying the fields that will be tested. The optional argument *n* specifies the number of parts to split the data into during the cross validation. The model will be estimated on all parts but one, and then tested on the remaining part. The procedure is repeated for all parts, i.e. *n*-fold cross-validation is performed. If *n* is the symbol `'size`, *n* is selected to be the size of the data object, i.e. leave-one-out cross-validation is performed. This is the default behaviour if *n* is not specified. *a* is an optional boolean variable specifying whether the model should try to predict all values of the fields in *fields* for all instances, even if the correct value is unknown for the instance. The default value is `#f`. The optional argument *rd* should be a mutable and adaptable data object in which the predicted results of the model will be stored. If *rd* is the symbol `'create`, then a data object to store the results will be created and returned. If the optional argument *act* is `#t`, two columns for each tested field will be used in the resulting data object; the first containing the actual value and the second the predicted. The defualt behaviour is to only write the predicted values to the data object. The procedure returns a list of test results, each element corresponding to the test results of each field in *f*, unless the argument *rd* is `'create`. In this case the procedure will return a list containing a list of test results as its first element and a data object containing the results in the second.

5.4. Specific models

gmdl provides a number of common models. Below, some of these models are described.

5.4.1. k-nearest neighbour

The k-nearest neighbour algorithm [6] is a basic example of an instance based learning algorithm. The algorithm does not (at least in its most simple form) estimate any parameters from the training data set, but only stores a reference to the data set for future use. When a prediction is made, the algorithm goes through all instances in the training data set to calculate which instances are the closest according to a distance measure, and selects the k closest ones. These instances are then used to predict the unknown attributes. For discrete attributes, the algorithm performs the classification by voting, and for continuous attributes by calculating the mean value of the attribute in the k neighbours.

The algorithm can optionally weigh the votes for discrete attributes and the mean value value calculation for continuous attributes by the distance. The weight is by default calculated as $w_k = 1/d(x_i, x_j)^2$, where $d(x_i, x_j)$ is the distance between the intances x_i and x_j. If two or more classes get the same, maximum, vote, the class with the lowest internal representation number will win. If the distance to any of the instances in the training data set is zero, these will be used for prediction by using non-weighted voting or mean value calculation.

The default distance function is the euclidian distance, $d(x_i, x_j) = \sqrt{\sum_r (a_r(x_i) - a_r(x_j))^2}$, where x_i and x_j are instances and $a_r(x)$ the attributes of these instances. If the attribute is discrete, $a_r(x_i) - a_r(x_j)$ is 0 if $a_r(x_i) = a_r(x_j)$ and 1 otherwise. Optionally, the continuous parameters can be normalized before used in the disctance calculation. The normalization parameters are estimated from the training data set. This will give equal or at least more similar significance to all attributes, both continuous and discrete. Each dimension in the model can also be assigned a weight, reflecting its importance in the distance calculation. The resulting default distance function can then be written as $d(x_i, x_j) = \sqrt{\sum_r w_r^2 (a_r(x_i) - a_r(x_j))^2}$, where w_r represents the weight. By default, all weights are set to 1.

`(make-k-nn` *form fields*`)` procedure
`(make-k-nn` *form fields k*`)` procedure

Creates a k-nearest neigbour model with the data format *form* over the fields *fields* using the k nearest neighbours. *fields* must be a list of fields available in *form*. If k is not specified, the value of the variable `k-nn-default-k` is used instead. The algorithm will only consider the attributes of the specified fields when calculating the distance between instances, and only the specified fields can be used for prediction.

`(k-nn? obj)` procedure

The procedure `k-nn?` returns `#t` if and only if *obj* is a *k*-nearest-neighbour model, otherwise `#f` is returned.

`(k-nn-k knn)` procedure
`(k-nn-set-k! knn k)` procedure

The procedure `k-nn-k` returns *k*, the number of neighbours used in *knn*. To set the value of *k*, use `k-nn-set-k!`.

`(k-nn-normalize? knn)` procedure
`(k-nn-set-normalize! knn n)` procedure

`k-nn-normalize?` returns `#t` if the the *k*-nearest-neighbour model *knn* normalizes the values of the attributes when calculating distances, otherwise `#f` is returned. To set whether *knn* should normalize its attributes or not, `k-nn-set-normalize!` is used. *n* must be a boolean value.

`(k-nn-distance-weighted? knn)` procedure
`(k-nn-set-distance-weighted! knn dw)` procedure

The procedure `k-nn-distance-weighted?` returns `#t` if *knn* weights the influence of each of the *k* nearest neighbours by their distance. `k-nn-set-distance-weighted!` sets whether *knn* should use distance weighting or not, where *dw* is a boolean value.

`(k-nn-weights knn)` procedure
`(k-nn-weights knn dim)` procedure
`(k-nn-set-weights! knn weights)` procedure
`(k-nn-set-weights! knn dim weight)` procedure

A weight can be assigned to each feature (or dimension) of the *k* nearest neighbour model. `k-nn-weights` retrieves the current weight of the dimension *dim* in model *knn*. If *dim* is not specified, a list of the weights for all dimensions in the model is returned. The procedure `k-nn-set-weights!` sets the weight of dimension *dim* in *knn* to *weight*. If only two arguments are provided, *weights* must be a list of weights of the same length as the number of dimensions in *knn*.

5.4.2. Neural networks

Neural networks (e.g. [7], [8], [9]), or at least what is often referred to when neural networks are mentioned, can be represented in gmdl using the *neural network model*. The model can represent a number of common network structures, perhaps most importantly the *multi layer perceptron* [10]. A neural network is represented by a set of neural layers, which are essentially a set of units using a certain transfer function. The layers are connected by neural connections, transferring activity from one layer to the next by possibly weighted links. Each layer can have several input and output connections, but each connection only has one input layer and one output layer. Several different types of layers and connections are available.

`(make-nnet form inputs outputs)` procedure

Create a neural network model with the data format *form* using the fields specified in the list *inputs* as inputs and *outputs* as outputs. The created neural network will have an input layer and an output layer that are not connected to each other, i.e. hiddden layers and connections must be set up correctly before the network can be estimated or used for predictions. A full network is normally constructed by first creating the network and a number of hidden layers, and then connected by creating connections between the networks input layer, the hidden layers and the output layer. The resulting network should not contain cycles. Note that the neither the input or the output layer uses a transfer function, i.e. the activity is passed directly from the input to the output. This means that what traditionally is referred to as a e.g. two layer network, one hidden layer and one output layer, here would have to contain two “hidden” layers. The last hidden layer, containing the same number of units as the output layer of the network, would then be connected to this output layer by a direct connection that does not modify the activities.

The number of units in the input and output layer depends on the format of the input and output fields. If a field is continuous, one unit will be used in the layer to represent its value. If a field is discrete, *n* units will be used in the layer to represent it, where *n* is the number of outcomes for the field. Thus, the number of input and output units may be larger than the number of input and output fields. When a discrete value is represented in a layer, one of the *n* units used to represent its value is set to one while the others are zero. When classification of a discrete value is made, the network uses the discrete value that is represented by the output node with the highest value amongst the units that represent the field.

`(nnet? obj)` procedure

Returns `#t` if and only if *obj* is a neural network model, otherwise `#f` is returned.

`(nnet-input-layer nnet)` procedure
`(nnet-output-layer nnet)` procedure

Returns the input and output layer of the neural network model *nnet*.

`(nnet-parameter-ref` *nnet param*`)` procedure
`(nnet-set-parameter!` *nnet param val*`)` procedure

Returns or sets parameter *param* of the neural network model *nnet*. The argument *param* must be a valid symbol and *val* a valid argument for the specified symbol. The symbols used are

`lrate`	`The learning rate`
`momentum`	`The momentum`
`max-epochs`	`Maximum number of epochs`
`min-epochs`	`Minimum number of epochs`
`lower-error-lim`	`Lower error limit`
`rel-error-lim`	`Relative error limit`
`lrate-mod-limit`	`Learning rate mod. limit`
`lrate-mod-factor`	`Learning rate mod. factor`
`mom-mod-factor`	`Momentum mod. factor`
`randomize-data`	`Randomize data`
`estimate-scaling`	`Estimate scaling`
`modify-lrate`	`Modify the learning rate`

Most of the parameters mostly influence the estimation of the model, such as the learning rate `lrate` and momentum `momentum` used by the network. `max-epochs` specify the maximum number of epochs during training. If the estimation reaches this number of epochs, it will be stopped regardless of the performance of the network. `min-epochs` sets the minimum number of epochs, i.e. this number of epochs will always be completed by the estimation procedure regardless of the network performance. The estimation will also be stopped as soon as the network error is below the absolute lower error limit `lower-error-limit`, as well as when the relative change in error from one epoch to the next falls below `rel-error-limit`. If this relative change in error falls below `lrate-mod-limit` and the boolean variable `modify-lrate` is `#t`, then the learning rate will be multiplied with `lrate-mod-factor` and the momentum with `mom-mod-factor`. `lrate-mod-factor` is usually lower than 1.0 (typically 0.5), since we generally want to lower the learning rate if network performance is not improved. `mom-mod-factor` is essentially used for compensating the lowered learning rate with a slight increase in momentum. Note that the momentum will never be set to a value above 0.99, since a momentum over 1.0 may cause the weights to not converge. If the boolean variable `randomize-data` is `#t`, then the order of the instances in the data set is randomized in every epoch. This is the preferred procedure. If the boolean variable `estimate-scaling` is true, then scaling parameters for all continuous inputs and outputs will be estimated from the data set, scaling these parameters to values between -1.0 and 1.0.

`(make-nlayer` *type n*`)` procedure
`(make-nlayer` *type n val*`)` procedure

Create a neural layer with *n* units using the transfer function specified by the symbol *type*. Valid values for *type* are

`linear`	`Linear layer`
`sigmoid`	`Sigmoid layer`
`tanh`	`tanh layer`
`log`	`Logarithmic layer`

The `linear` layer uses a linear transfer function $f(x) = c * x$, where the constant c is specified by the optional argument *val*. The `sigmoid` layer uses the sigmoid transfer function $f(x) = 1/(1 + \exp(-x))$, while the `tanh` uses $f(x) = (\exp(x) - \exp(-x))/(\exp(x) + \exp(-x))$ and `log` the natural logarithm as $f(x) = \log(1+x)$ when $x > 0$ and $f(x) = \log(1 - x)$ otherwise.

`(nlayer?` *obj*`)` procedure

Returns `#t` if and only if *obj* is a neural layer, otherwise `#f` is returned.

`(nlayer-units` *nlayer*`)` procedure

Returns the number of units in the neural layer *nlayer*.

`(nlayer-inputs` *nlayer*`)` procedure
`(nlayer-outputs` *nlayer*`)` procedure

Returns a list of the input and output connections respectively for the neural layer *nlayer*. If no input or output connections exist, the empty list is returned.

`(nconnection?` *obj*`)` procedure

Returns `#t` if and only if *obj* is a neural connection, otherwise `#f` is returned.

`(nconnection-input` *nc*`)` procedure
`(nconnection-output` *nc*`)` procedure

Return the input or output layer of the neural connection *nc*. If the neural connection is not connected, `#f` is returned.

`(nconnection-disconnect!` *nc*`)` procedure

Disconnect the neural connection *nc*. The connection will be removed from the network and its input and output set to `#f`. It will also be removed from its input layers list of outputs as well as its output layers list of inputs.

`(make-nconnection-direct` nl_1 nl_2`)` procedure

Create a direct neural connection between neural layer nl_1 and nl_2, where nl_1 and nl_2 must contain the same number of units. The direct connection is a 1-to-1 connection between two layers, i.e. there is only one connection from each unit in nl_1 to the corresponding unit in nl_2. The weight for these connections is always 1.0, and it is not modified when estimating the model. The direct neural connection is mainly provided to make a wider range of network topologies possible.

`(nconnection-direct?` *obj*`)` procedure

Returns `#t` if and only if *obj* is a direct neural connection, otherwise `#f` is returned.

`(make-nconnection-full` nl_1 nl_2`)` procedure

Create a full neural connection between neural layer nl_1 and nl_2, where nl_1 and nl_2 can contain any number of units. The full connection is an all-to-all connection between nl_1 and nl_2, i.e. there is one weighted connection for each unit in nl_1 to all units in nl_2. The connection also holds a bias value for each of the units in the output layer nl_2, which can be thought of as a weighted connection from the constant value 1.0 to the unit. All weights are modified during estimation of the network, including the bias weights. When the connection is created as well as when estimation is started, all weights are initialised to a random value between $-2.4/n_i$ and $2.4/n_i$, where n_i is the number of units in the input layer.

`(nconnection-full?` *obj*`)` procedure

Returns `#t` if and only if *obj* is a full neural connection, otherwise `#f` is returned.

`(nconnection-full-bias` *fnc*`)` procedure
`(nconnection-full-weights` *fnc*`)` procedure

The procedure `nconnection-full-bias` returns a vector containing the current bias weights of the full neural connection fnc. `nconnection-full-weights` returns a vector of vectors containing the current weights of fnc.

5.4.3. The ensemble model

An ensemble model combines predictions from a number of component models. The ensemble model can use any kind of model over any set of fields as a component model, i.e. the component models can provide predictions for different fields. This is useful for combining simpler models that themselves only predict a subset of the total set of fields to a complete predictor for all fields. When predicting the value of a discrete attribute the component models are combined by voting. If two or more classes get the same vote, the class with the lowest internal representation number will win. For continuous attributes, the component models are combined by calculating the mean value of the predictions provided by the component models.

`(make-ensemble-model` *mlst cfields*`)` procedure

Creates an ensemble model, using the models in the list *mlist* as its component models.

`(ensemble-model?` *obj*`)` procedure

The procedure `ensemble-model?` returns `#t` if and only if *obj* is an enemble model, otherwise `#f` is returned.

5.4.4. Boosting

Boosting is a general method for improving the accuracy of any given learning algorithm or component model. The boosting algorithm used here is usually referred to as AdaBoost, introduced in 1995 by Freund and Schapire. AdaBoost maintains a weight for each instance in the training data set, and the higher the weight the more the instance influences the classifier learned. At each trial, the vector of weights is adjusted to reflect the performance of the corresponding classifier, and the weight of misclassified instances is increased. The error for a single classification is 1 if the predicted and actual classes differ, otherwise it is zero. For continuous attributes, the error if 1 if the predicted value is outside the correct value plus / minus the specified error limit, otherwise it is considered correct and the error is 0. Note that if several class fields are used, a classification is only considered correct (i.e. the error for the classification is 0) if and only if the errors for all class fields are 0. The total error for a classifier is essentially the error frequency of the whole data set (the sum of errors divided by the number of instances in the data object) and is a number between 0 and 1. The component classifiers are each given a weight that reflects this error rate.

AdaBoost can operate on both discrete and continuous attributes, although the former is more straightforward. When predicting the value of a discrete attribute the component models are combined by weighted voting, each model having a vote equal to the weight calculated during estimation. If two or more classes get the same vote, the class with the lowest internal representation number will win. For continuous attributes, the component models are combined by calculating a weighted mean value over the predictions of the component models that reflects each models weight.

The AdaBoost algorithm can be choosen to be either the normal AdaBoost algorithm or AdaBoost.M1, differing mainly in the calculation of the weights. For details about the algorithms, see [11].

`(make-ada-boost` *mlst cfields*`)` procedure

Creates an ada-boost model, using the models in the list *mlist* as its component models and the fields specified in the list *cfields* as class fields, i.e. the fields the error will be calculated upon.

`(ada-boost? obj)` procedure

The procedure `ada-boost?` returns `#t` if and only if *obj* is an ada-boost model, otherwise `#f` is returned.

`(ada-boost-number-of-trials mdl)` procedure

Returns the number of trials used for the ada-boost model *mdl*. This number also represent the number of component models used and estimated by the ada-boost model.

`(ada-boost-break-on-error mdl)` procedure
`(ada-boost-set-break-on-error! mdl b)` procedure

The procedure `ada-boost-break-on-error` returns `#t` if the model interrupts the estimation of component models when the total error for a model exceeds 0.5. To set whether the model should break under these circumstances or not, use `ada-boost-set-break-on-error!` where *b* is a boolean value. If the model does interrupt estimation, not all component models will be estimated or used during classification.

`(ada-boost-error-limit mdl)` procedure
`(ada-boost-set-error-limit! mdl limit)` procedure

`ada-boost-error-limit` returns the continuous error limit for the ada-boost model *mdl*. To set the error limit for the model to *limit*, use `ada-boost-set-error-limit!`.

`(ada-boost-algorithm mdl)` procedure
`(ada-boost-set-algorithm! mdl alg)` procedure

`ada-boost-algorithm` returns the algorithm used by the ada-boost model *mdl*, which is either the symbol `ada-boost-normal` (for normal AdaBoost) or `ada-boost-M1` (for AdaBoost.M1). To set the algorithm of *mdl*, use `ada-boost-set-algorithm!` where *alg* is one of the two symbols described above.

5.4.5. Bagging

Bagging, or bootstrap aggregating, tries to create a better classifier by combining a number of simpler classifiers [12]. These component classifiers can be any kind of model, even another bagging model. When the bagging model is estimated from data, it creates replicate training sets by sampling with replacement from the total data set. The size of these data sets is called the training set size, and each component model is estimated on one of these data sets separately.

Bagging models in gmdl operate on both discrete and continuous attributes. When predicting the value of a discrete attribute, or classifying the attribute, the component models are combined by voting. If two or more classes get the same number of votes, the class with the lowest internal representation number will win. For continuous attributes, the component models are combined by calculating the mean value of the predictions for the attribute.

`(make-bagging mlst tssize)` procedure

Creates a bagging model using the models in the list *mlst* as component models and a training set size of *tssize*.

`(bagging? obj)` procedure

The procedure `bagging?` returns `#t` if and only if *obj* is a bagging model, otherwise `#f` is returned.

`(bagging-training-set-size mdl)` procedure
`(bagging-set-training-set-size! mdl n)` procedure

`bagging-training-set-size` returns the current training set size of the bagging model *mdl*. To set the training set size of *mdl* to *n*, `bagging-set-training-set-size!` is used.

5.5. Distributions

gmd provides a number of statistical distributions, useful e.g. for data modeling purposes. A distribution in gmdl is a kind of model, which means that all procedures that operate on all models also operate on distributions. This also means that some of the operations normally performed on distributions, like estimation from data, are also performed on models. Procedures for such procedures will be found under general model procedures and not distribution specific procedures.

Only two higher order distributions, i.e. distributions that are composed by other distributions in some manner, are provided in gmdl. These are mixture models and graphical models, and allow for the construction of most common statistical models as well as more complex ones. See [20] for further details about the modeling procedures.

5.5.1. General distribution functionality

The procedures below operate on all distributions.

`(distr? obj)` procedure

The procedure `distr?` returns `#t` if and only if *obj* is a distribution, otherwise `#f` is returned.

`(distr-priored?` *distr*`)` procedure
`(distr-set-priored!` *distr p*`)` procedure

`distr-priored?` returns true if and only if the distribution *distr* would be considered to have a valid prior distribution when estimation of the distribution is performed, and `#f` otherwise. To set whether a distribution *distr* should be considered having a prior, use `distr-set-priored!` where *p* must be a boolean value.

`(distr-marginal` *distr fields*`)` procedure

Calculates the marginal over the fields in the list *fields* of distribution *distr*. The fields in the list that are not included in *distr* are ignored. If no fields in *fields* are included in *distr*, an empty distribution is returned.

`(distr-conditional` *distr vals*`)` procedure

Calculates the conditional of the distribution *distr* over the fields in the data vector *vals* that are `#?` given the known fields. If all fields in *distr* are known, an empty distribution is returned. If all fields in *distr* are known, a copy of *distr* is returned.

`(distr-expectation` *distr vals*`)` procedure

Returns the expectation of the distribution *distr* given the known values the data vector *vals* as a vector of the same length as the dimensionality of the distribution. Note that calculating the expectation is not a well defined operation for all distributions.

`(distr-random` *distr vals*`)` procedure

Returns a copy of the data vector *vals* with the fields of *distr* that are unknown substituted with random values, drawn according to the distributon and the known values in *vals*.

`(distr-prob` *distr vals*`)` procedure
`(distr-log-prob` *distr vals*`)` procedure

`distr-prob` calculates the probability (or probability density) of a data vector *vals* for distribution *distr*. Some or all of the fields of *distr* can be left unknown, in which case the marginal over the known fields is used for the calculation. If all fields in *distr* are unknown, 1.0 is returned. Similarly, `distr-log-prob` returns the natural logarithm of the probability. Note that it is often numerically more stable to calculate the logarithm of the probability.

`(distr-log-model-prob` *distr*`)` procedure

Returns the natural logarithm of the model probability of *distr*, useful for Bayesian calculations.

`(distr-likelihood` *distr data*`)` procedure
`(distr-log-likelihood` *distr data*`)` procedure

Returns the likelihood and the natural logarithm of the likelihood of the samples in a data object *data* being drawn from the distribution *distr* respectively.

`(distr-entropy` *distr*`)` procedure

Returns the entropy of the distribution. Note that the entropy is always calculated using the natural logarithm (the result will be in nats, not bits).

`(distr-kullback-leibler` $distr_1$ $distr_2$`)` procedure

Calculates and returns the Kullback-Leibler distance between the distribution $distr_1$ and $distr_2$. Note that only a subset of distances between different distribution types can be reasonably calculated and that the measure is not symmetrical.

`(distr-estimate-prior!` *distr data*`)` procedure
`(distr-estimate-prior!` *distr data prob*`)` procedure

Estimate the prior distribution of *distr* from the data object *data*. The optional value *prob* provides the estimation procedure with a probability for each instance in the data object. *prob* must be a vector of the same size as the data object, where each element is a continuous value between zero and one.

`(distr-multiply` $distr_1$ $distr_2$`)` procedure

Returns the result of multiplying and then normalizing the distributions $distr_1$ and $distr_2$. The distributions must be over the same fields, and while distributions of the same type usually can be multiplied, it is not at all certain that two distributions of different types can be multiplied.

5.5.2. Specific distributions

gmdl provides a number of statistical distributions, most of them multivariate. These are described below.

The discrete distribution

About the discrete distribution. Representation of continuous variables. Representation in vector form. Representation of priors.

`(make-disc-distr` *form fields*`)` procedure

Creates a new discrete distribution over the fields specified in the list *fields* using the data format *form*.

`(disc-distr?` *obj*`)` procedure

Returns `#t` if and only if *obj* is a discrete distribution, otherwise `#f` is returned. When created, the distribution is initialised to be equally distributed.

`(disc-distr-prob` *dd*`)` procedure
`(disc-distr-prob` *dd pos*`)` procedure
`(disc-distr-set-prob!` *dd pvec*`)` procedure
`(disc-distr-set-prob!` *dd pos p*`)` procedure

`disc-distr-prob` returns the probability of the outcome specified by *pos*, where *pos* either is a list of integers specifying the outcome for each dimension of the distribution or a single integer value, which is interpreted as a position in the vector representation of the distribution. If *pos* is not specified, the vector representation of the distribution is returned as a vector.

The procedure `disc-distr-set-prob!` sets the probability of the outcome specified by *pos* to *p*, where *pos* can be a list or an integer value in the same manner as for `disc-distr-prob`. If only two arguments are used for the procedure, *pvec* must be a vector specifying the probability of each outcome in the discrete distributions vector representation. Note that the probabilities for the outcomes will be set exactly to the specified values, even though they may not represent a proper distribution.

`(disc-distr-prior-prob` *dd*`)` procedure
`(disc-distr-prior-prob` *dd pos*`)` procedure
`(disc-distr-set-prior-prob!` *dd pvec*`)` procedure
`(disc-distr-set-prior-prob!` *dd pos p*`)` procedure

These procedures operate in the same manner as `disc-distr-prob` and `disc-distr-set-prob!`, the difference being that they return and set the prior distribution of *dd*. Please refer to these procedures for a more detailed description.

`(disc-distr-prior-alpha` *dd*`)` procedure
`(disc-distr-set-prior-alpha!` *dd* α`)` procedure

`disc-distr-prior-alpha` returns the current value of α for the discrete distribution *dd*. To set the value of α, the procedure `disc-distr-set-prior-alpha!` is used.

`(disc-distr-normalize!` *dd*`)` procedure

Normalizes the discrete distribution *dd*.

The Gaussian distribution

Mixture Models

Graphical Models

6. Graphics and plotting

The gmdl system provides both basic drawing primitives as well as higher level data plotting functionality, so that new visualization routines can be created as well as easy access to common plotting functionality. We will here describe both the basic primitives as well as the higher order procedures.

6.1. The Canvas

Most graphics in in gmdl is drawn on a *canvas*. The contents of the canvas is displayed on screen or is written direct to file, e.g. as PostScript or PDF. Each canvas needs a *display procedure*, were all drawing in the canvas is performed. This procedure is called when needed, which might be quite frequently in some interactive cases. If this is the case, it might be a good idea to make sure that the procedure is relatively fast. The graphics model is closely related to and based on OpenGL [13], so a basic familiarity with this system will help in understanding this chapter.

6.1.1. Creating and changing properties of a canvas

`(make-canvas` *dp*`)` procedure
`(make-canvas` *dp width height*`)` procedure
`(make-canvas` *dp width height name*`)` procedure
`(make-canvas` *dp width height name bgcol*`)` procedure

Create a canvas using the display procedure *dp*. The display procedure should take the current canvas as the only argument. The width and height of the drawing area is set to *width* and *height* if specified, otherwise the values of the variables `canvas-default-width` and `canvas-default-height` are used. The name of the canvas is set to the string *name* if specified, otherwise the canvas will be given a name by the system. *bgcol* specifies the background color of the canvas (more on colors in gmdl below). If no background color is specified, white is used.

`(canvas?` *obj*`)` procedure

Returns `#t` if and only if *obj* is a canvas. Otherwise `#f` is returned.

`(canvas-redraw` *canv*`)` procedure

Redraws the canvas *canv* by calling its display procedure. The redraw procedure usually does not need to be called explicitly as this is normally handled by the canvas itself, but is useful e.g. for creating animations.

`(canvas-set-needs-redraw!` *canv proc*`)` procedure

This procedure sets the *needs redraw procedure* of *canv* to *proc*. The procedure should take one one argument, a canvas, and return a value different from `#f` if the canvas needs to be redrawn and `#f` otherwise. The canvas will run this procedure regularly in the background to see if there is any need to redraw the contents of the canvas. By default, no such procedure is used, and the canvas is only redrawn if e.g. the window size is changed by the user. The needs redraw procedure is very useful e.g. in interactive plotting situations, making it possible to redraw the canvas whenever the user changes the plotted data in any way. Note, however, that the procedure is called relatively often and should be very fast not to cause any noticeable computational overhead.

`(canvas-remove-window!` *canv*`)` procedure

Removes the window belonging to *canv* that is displayed on screen (if any).

`(canvas-write` *canv file*`)` procedure
`(canvas-write` *canv file type*`)` procedure

Writes the contents of the canvas *canv* as is to the file specified by the string *file*. The symbol *type* specifies the file format to use, and can be `'ps`, `'eps`, `'pdf` or `'tex`, representing PostScript, Encapsulated PostScript, PDF and TeX format respectively. The default format is Encapsulated PostScript.

`(canvas-win-width` *canv*`)` procedure
`(canvas-win-height` *canv*`)` procedure
`(canvas-win-size` *canv*`)` procedure
`(canvas-set-win-size!` *canv width height*`)` procedure

These procedures returns and sets the actual size of the displayed window belonging to *canv*. `canvas-win-width` returns the width, `canvas-win-height` the height and `canvas-win-size` a list of two elements, the width and the height. To set the size of the window to *width* and *height*, `canvas-set-win-size!` is used.

Canvas properties

`(canvas-set-anti-alias!` *canv val*`)` procedure

Sets whether anti-aliasing should be used for *canv*. *val* must be a boolean value. The default is not to use anti-aliasing.

`(canvas-set-gray-scale!` *canv val*`)` procedure

Sets whether all colors in *canv* should be converted to gray scale. *val* must be a boolean value. The default is not to convert all colors to gray scale.

6.1.2. Viewing and projections

All graphics in a canvas is specified in three-dimensional space. This means that we have to have some means of choosing a projection from this three-dimensional space to a suitable two-dimensional plane for viewing on-screen or on paper. Below we describe a few procedures for selecting and setting this projection. The projection can be changed at any time in a display procedure, but this is not recommendable. Instead, choose and set the projection as the first thing you do in a display procedure and leave it to that, transforming model space instead if necessary.

The default view is an orthographic projection (see below) with left edge at 0, right edge at the width of the window, bottom edge at 0, and top edge at the height of the window. The near clipping plane is set at -1 and the far plane at 1.

`(canvas-ortho` *lft rght btm top near far*`)` procedure

Sets the current view to an orthographic projection, which maps objects directly onto the screen without affecting their relative size. It is suitable e.g. for two-dimensional plotting and computer-aided design applications. The projection is done to the x, y plane. lft and *rght* specifies the left and right edges of the plane, and *btm* and *top* the bottom and top edges. *near* and far specifies the near and far clipping planes, i.e. no graphics with a z-position less than *near* and higher than far will be drawn.

`(canvas-frustum` *lft rght btm top near far*`)` procedure

Sets the current view to a perspective projection, i.e. the farther away from the actual viewpoint an object is, the smaller it will appear. This occurs because the viewing volume for a perspective projection is a frustum of a pyramid (a truncated pyramid whose top has been cut off by a plane parallel to its base). Objects that fall within the viewing volume are projected toward the apex of the pyramid, where the viewpoint is. Objects that are closer to the viewpoint appear larger because they occupy a proportionally larger amount of the viewing volume than those that are farther away, in the larger part of the frustum.

The frustum's viewing volume is defined by the arguments: $(lft, btm, -near)$ and $(rght, top, -near)$ specify the (x, y, z) coordinates of the lower left and upper right corners of the near clipping plane; *near* and far give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

`(canvas-look-at` e_x e_y e_z c_x c_y c_z u_x u_y u_z`)` procedure

Sets the current view to a perspective projection. Often, programmers construct a scene around the origin or some other convenient location, then they want to look at it from an arbitrary point to get a good view of it. As its name suggests, this procedure is designed for just this purpose. It takes three sets of arguments, which specify the location of the viewpoint, define a reference point toward which the camera is aimed, and indicate which direction is up.

The desired viewpoint is specified by e_x, e_y, and e_z. The c_x, c_y, and c_z arguments specify any point along the desired line of sight, but typically some point in the center of the scene being looked at. The u_x, u_y, and u_z arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume).

6.1.3. Transforms and drawing coordinates

All actual drawing in a canvas is actually performed in a transformed model space. By default, the transform of the canvas is the one-to-one transform that does actually not modify the drawing space at all. This can, however, be changed, and it is possible to transform a set of drawing primitives by for example translation, scaling or rotation.

A three-dimensional transformation, as used by gmdl, can be described mathematically by a 4x4 matrix:

$$\begin{pmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{pmatrix}$$

In gmdl, the matrix is represented as a vector with 16 fields, m_1 to m_{16}. Transformations in a canvas can be "stacked" on top of each other, e.g. one transformation might transform an earlier transformation and so on.

`(canvas-translate` *transl expr1 expr2 ...*`)` procedure

Evaluates the expressions *expr*1, *expr*2 etc. in an environment where the canvas has been translated by *transl*. *transl* must be a vector of three elements, specifying the translation on the x, y and z axis.

`(canvas-scale` *scale expr1 expr2 ...*`)` procedure

Evaluates the expressions *expr*1, *expr*2 etc. in an environment where the canvas has been scaled by *scale*. *scale* must be a vector of three elements, specifying the scaling of the x, y and z axis.

`(canvas-rotate` *rot expr1 expr2 ...*`)` procedure

Evaluates the expressions *expr*1, *expr*2 etc. in an environment where the canvas has been rotated by *rot*. *rot* must be a vector of four elements, the first specifying the rotation angle, and the following three the rotation axis.

`(canvas-identity` *expr1 expr2 ...*`)` procedure

Evaluates the expressions *expr*1, *expr*2 etc. in an identity environment, i.e. all outer canvas transformations are ignored and the identity transform is used instead.

`(canvas-load-transf` *transf expr1 expr2 ...*`)` procedure

Evaluates the expressions *expr*1, *expr*2 etc. in an explicitly stated transform environment $transf$, i.e. all outer canvas transformations are ignored and the transform $transf$ is used instead. $transf$ must be a vector with 16 elements specifying the complete transformation matrix as described earlier.

`(canvas-add-transf` *transf expr1 expr2 ...*`)` procedure

Evaluates the expressions *expr*1, *expr*2 etc. in an environment where $transf$ has been added. $transf$ must be a vector with 16 elements specifying the complete transformation matrix as described earlier.

6.1.4. Color

Colors used on a canvas are represented in RGBA color space a by a vector of four numbers: the amount of red, green and blue, and an alpha value. The color is defined by the amounts of red, green and blue, and the alpha channels sets the transparency.

`(canvas-color!` *col*`)` procedure

Set the current drawing color to *col*. All subsequent drawing is performed using this color until it is changed by the same procedure. *col* must be a vector with four elements, representing the amount of red, green, blue and alpha in that order. All values must be between zero and one.

6.1.5. Fonts

The font used for writing text on a canvas can be chosen from a rather wide range of fonts, styles, and sizes. Fonts are specified by their X11 name, and all available X11 fonts can be used on a canvas. However, if the graphics on the canvas is intended for output to file on PostScript or PDF, then some care must be taken in choosing a font that is among the standard fonts available on the system were the graphics are intended to be displayed. No fonts will be embedded into the produced file.

`(canvas-font!` *canv font*`)` procedure

Set the font of the canvas *canv* to $font$. All subsequent text drawing is performed using this font until it is changed by the same procedure. $font$ must be a string specifying an available X11 font.

`(canvas-font-width` *canv font*`)` procedure
`(canvas-font-ascend` *canv font*`)` procedure
`(canvas-font-descend` *canv font*`)` procedure

Return the width, ascend and descend of the font used by the canvas *canv*.

6.1.6. Drawing primitives

There is a number of drawing primitives that can be used on a canvas, which are usually used to specify more complex graphics by grouping them into procedures. The primitives should be general enough to describe most 3D graphics. No graphics primitives need to specify the canvas as an argument, since this is known and set by the canvas calling the display procedure. If the primitives are used when not called directly or indirectly by a canvas, they will have no effect whatsoever. The color and settings used for drawing are the ones currently used by the canvas.

Settings

`(canvas-point-size!` *size*`)` procedure

Sets the point size to *size*. All points drawn on the canvas will use this size until the point size is set to a different value.

`(canvas-line-width!` *width*`)` procedure

Sets the line width to *width*. All lines drawn on the canvas will use this width until the line width is set to a different value.

Primitives

`(canvas-point` x y z`)` procedure

Draw a point at (x, y, z).

`(canvas-line` x_1 y_1 z_1 x_2 y_2 z_2`)` procedure

Draw a line from (x_1, y_1, z_1) to (x_2, y_2, z_2).

`(canvas-disc` x y z w h`)` procedure

Draw a filled disc (an ellipse), centered at (x, y, z), with width w, and height h. The width and the height is given in the (x, y) plane, and the z parameter only represents a translation on the z-axis of the ellipse.

`(canvas-rect` x_1 y_1 x_2 y_2 z`)` procedure

Draw a filled rectangle on the canvas, with the first corner at (x_1, y_1), and the opposite corner at (x_1, y_1). The corner positions are given in the (x, y) plane, and the z parameter only represents a translation on the z-axis of the rectangle.

`(canvas-text` x y z *str*`)` procedure
`(canvas-text` x y z *str align*`)` procedure

Draw the text in the string *str* on the canvas *canv* at (x, y, z), using the current font. The argument *align* should be a symbol `left`, `right` or `center`, specifying how the text should be aligned around (x, y, z). If the *align* argument is not specified, left alignment is used.

`(canvas-draw` *type* p_1 p_2 p_3 ...`)` procedure

This is the general drawing primitive for a canvas. The arguments p_1, p_2 etc. represent points in space, each given as vectors containing three values: the x, y, and z position of each point. The *type* argument determines how these points should be interpreted and drawn on screen. It must be a symbol, taking on one of the following values:

`points`	`line-strip`
`lines`	`line-loop`
`polygon`	`triangle-strip`
`triangles`	`triangle-fan`
`quad`	`quad-strip`

`points` draws individual points, and `lines` pairs of points interpreted as individual line segments. For `polygon`, the points specify the boundary of a simple, convex polygon. `triangles` assumes triples (p_1, p_2, p_3 etc.) of points interpreted as triangles, and `quads` quadruples of points interpreted as four-sided polygons. `line-strip` draws series of connected line segments, as does `line-loop`, but with a segment added between last and first points. `triangle-strip` displays a linked strip of triangles, `triangle-fan` a linked fan of triangles, and `quad-strip` a linked strip of quadrilaterals.

6.1.7. The frame buffer

`(canvas-scissor` *area expr1 expr2 ...*`)` procedure

The procedure `canvas-scissor` displays all expressions *expr*1, *expr*2 etc. in an environment where all resulting drawing outside the specified *area* is ignored, thus allowing for selecting a subset of the window area to draw in. The area is specified by a vector of four elements: the x and y coordinates of the lower left corner and the width and height of the area. The coordinates are given as window coordinates, and are not affected by any transforms.

6.1.8. Event handling

Events, like mouse buttons and keys pressed, are handled by specifying callback procedures for each event type and canvas. By default, no procedures are called on any type of events except for a set of normal window operations, such as resizing, for which the display procedure is called.

`(canvas-set-button-event!` *canv proc*`)` procedure

Sets the procedure to be called for mouse button events in canvas *canv* to *proc*. The procedure *proc* should take five arguments: the calling canvas, the x position of the event, the y position of the event, the button number, and a modifier number. The button number usually ranges from 1 to 3, 1 being the left button and 3 the right button. The modifier number represents any modification keys pressed at the time of an event, such as shift, alt etc.

`(canvas-set-key-event!` *canv proc*`)` procedure

Sets the procedure to be called for key press events in canvas *canv* to *proc*. The procedure *proc* should take five arguments: the calling canvas, the x position of the event, the y position of the event, the key pressed, and a modifier number. The key is given as a character. The modifier number represents any modification keys pressed at the time of an event, such as shift, alt etc.

6.2. Plotting

Here we will present a number of higher-order plotting functions that are used to visualize data.

`(make-series-plot` *data fields*`)` procedure
`(make-series-plot` *data fields layout*`)` procedure

Plot $fields$ in data object *data* as series, i.e. let the x-axis represent the index in the data object and the y-axis the value in the specified field(s) of the data object. $fields$ may be either an integer/string specifying a single field, or a list of field specifications. All specified fields are plotted. The optional argument *layout* specifies how to display the plots, and can be either the symbol `'stack`, in which case each field is plotted separately with separate axis, or `'add`, where all plots are added on top of each other within the same axis. The default value is `'add`.

`(make-pairs-plot` *data xfields yfields*`)` procedure
`(make-pairs-plot` *data xfields yfields layout*`)` procedure

Plot pairs of $xfields$ and $yfields$ in data object *data*. Let the x-axis represent the value in the specified $xfields$ of the data and the y-axis the value in $yfields$, and plot a point for each index in the data object. $xfields$ and $yfields$ may be either an integer/string specifying a single field, or a list of field specifications. If they are specified as lists, they must be of the same length. All specified fields are plotted. As before, the optional argument *layout* specifies how to display the plots, and can be either the symbol `'stack`, in which case each field is plotted separately with separate axis, or `'add`, where all plots are added on top of each other within the same axis. The default value is `'add`.

`(make-trace-plot` *data xfields yfields*`)` procedure
`(make-trace-plot` *data xfields yfields layout*`)` procedure

Plot pairs of $xfields$ and $yfields$ in data object *data*. Let the x-axis represent the value in the specified $xfields$ of the data and the y-axis the value in $yfields$, and draw a line between consecutive indexes in the data object. This amounts to plotting $yfields$ against $xfields$. $xfields$ and $yfields$ may be either an integer/string specifying a single field, or a list of field specifications. If they are specified as lists, they must be of the same length. All specified fields are plotted. As before, the optional argument *layout* specifies how to display the plots, and can be either the symbol `'stack`, in which case each field is plotted separately with separate axis, or `'add`, where all plots are added on top of each other within the same axis. The default value is `'add`.

7. Mathematical and numerical functions

This chapter describes gmdl's built-in mathematical and numerical procedures. The initial (or "top level") gmdl environment starts out with a number of variables bound to locations containing useful values or procedures. gmdl provides both bindings to numerical constants, useful in many numerical calculations, and mathematical procedures that might occur in more complex numerical routines.

Note: Although a program may use a top-level definition to bind any variable, altering any top-level binding that has not been introduced by a definition has an unspecified effect on the behavior of the built-in procedures.

7.1. Mathematical functions

7.1.1. Not-a-number and infinity

Perhaps not necessarily related to mathematical routines but rather to the machine representation of numbers, are the special values *not-a-number* and *positive* and *negative infinity*. These may on occasion be returned from some numerical procedures instead of producing an error, since they still may contain useful information about the result. Below is a listing of these special values as defined by default in the initial gmdl environment:

`nan`	`Not-a-number`
`+inf`	`Positive infinity`
`-inf`	`Negative infinity`

`nan` (not-a-number) is used to specify values that can not be interpreted as a number, such as division by zero. `+inf` and `-inf` is defined as the representation of positive and negative infinity, e.g. useful for specifying infinite intervals. The external representation of the values are usually `#.#` for not-a-number, `+#.#` for positive infinity and `-#.#` for negative infinity.

`(nan?` *obj*`)` procedure
`(inf?` *obj*`)` procedure
`(+inf?` *obj*`)` procedure
`(-inf?` *obj*`)` procedure

These procedures are used to test whether an object is not-a-number or infinity. All procedures return true if and only if the object is of the type implied by the procedures name.

7.1.2. Mathematical constants

When writing efficient numerical routines, it is often necessary to define and use high-precision numerical constants to avoid unnecessary computation. gmdl provides some very common constants defined in the initial gmdl environment. Here is a listing of the constants along with their meaning:

`e`	e
`log2e`	$\log_2 e$
`log10e`	$\log_{10} e$
`sqrt2`	$\sqrt{2}$
`sqrt1/2`	$\sqrt{1/2}$
`sqrt3`	$\sqrt{3}$
`pi`	π
`pi/2`	$\pi/2$
`pi/4`	$\pi/4$
`sqrtpi`	$\sqrt{\pi}$
`2/sqrtpi`	$2/\sqrt{\pi}$
`1/sqrt2pi`	$1/\sqrt{2\pi}$
`1/pi`	$1/\pi$
`2pi`	2π
`ln10`	$\ln(10)$
`ln2`	$\ln(2)$
`lnpi`	$\ln(\pi)$
`euler`	`Euler's constant` (γ)
`degree`	$\pi/180$
`golden-ratio`	`The golden ratio` (ϕ)
`catalan`	`Catalan's constant`

Euler's constant ($\gamma \simeq 0.577216$), or the Euler-Mascheroni constant, appears in many integrals and asymptotic formulas. Catalan's constant ($\simeq 0.915966$) sometimes appears in asymptotic estimates of combinatorial functions. The golden ratio is defined as $\phi = (1+\sqrt{5})/2$. `degree` ($\pi/180$) is the degrees to radians conversion factor.

7.1.3. Mathematical and numerical procedures

gmdl provides a number of useful mathematical and numerical procedures as part of the implementation in addition to the routines specified in the Scheme programming language standard. For a complete listing of mathematical and numerical procedures, please also refer to the Scheme documentation.

Note: Many of these routines contain a trade-off between calculation speed and accuracy. While both the speed and accuracy of the operations should be sufficient for most needs, there is no guarantee that these functions are, most importantly, accurate enough for all applications.

`(eq-eps?` z_1 z_2 ϵ`)` procedure

It is sometimes useful to compare two numbers approximately, to allow for truncation and rounding errors. `eq-eps?` determines whether z_1 and z_2 are approximately equal to the given accuracy ϵ.

`(sign` *z*`)` procedure

Returns the sign of z, defined as 1 if $z >= 0$ and -1 otherwise.

`(expt` *z p*`)` procedure
`(expt2` *z*`)` procedure
`(expt3` *z*`)` procedure
`(expt4` *z*`)` procedure

`expt` returns z to the power of p. For small integer powers, the procedures `expt2`, `expt3` and `expt4`, returning z to the power of 2, 3 and 4 respectively, are faster and numerically more stable choices.

`(cbrt` *z*`)` procedure

Returns the cube root of z. The result will always be a real number.

`(sinc` *z*`)` procedure

Returns the sinc function of z, where $sinc(z) = \sin(\pi z)/\pi z$. This function is often used in signal processing.

`(factorial` *n*`)` procedure
`(factorial-ln` *n*`)` procedure
`(binomial-coefficient` *n m*`)` procedure
`(cumulative-binomial-probability` *n k p*`)` procedure

`factorial` returns the factorial of n ($n!$), while `factorial-ln` returns the natural logarithm of the factorial ($\ln(n!)$). `factorial-ln` is provided for maintaining numerical precision, and is a better choice for most numerical routines since the result does not grow as fast as the factorial. `binomial-coefficient` returns the binomial coefficient $\binom{n}{m} = n!/m!(n-m)!$. It gives the number of ways of choosing m objects from a collection of n objects. `cumulative-binomial-probability` calculates the probability of an event with probability p occurring k or more times in n trials. These procedures always return real numbers.

`(beta` a b`)` procedure
`(beta` a b z`)` procedure
`(beta-regularized` a b z`)` procedure
`(inverse-beta-regularized` a b s`)` procedure

`beta` returns the Euler beta function $B(a,b) = \Gamma(a)\Gamma(b)/\Gamma(a+b) = \int_0^1 t^{a-1}(1-t)^{b-1}dt$, useful e.g. in many statistical computations. The three-argument variant of `beta` returns the incomplete beta function $B_z(a,b) = \int_0^z t^{a-1}(1-t)^{b-1}dt$. The argument z is the upper limit of integration and should be a value between 0 and 1.

Sometimes it is more convenient to calculate the incomplete beta function in its regularized form, in which it is divided by the complete beta function. This is what is returned by `beta-regularized`. The inverse of the beta function is only calculated in its regularized form, and is returned by the procedure `inverse-beta-regularized`.

`(gamma-ln` z`)` procedure
`(gamma` z`)` procedure
`(gamma` z a`)` procedure
`(gamma-regularized` z a`)` procedure
`(gammac-regularized` z a`)` procedure
`(inverse-gamma-regularized` s a`)` procedure

The Euler gamma function $\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt$ can be calculated using the procedure `gamma`. For positive integers n, $\Gamma(n) = (n-1)!$. `gamma-ln` returns the natural logarithm of the gamma function to preserve numerical precision. This function is a better choice for must numerical computations. The two-argument version of `gamma` calculates the incomplete gamma function $\Gamma(a,z) = \int_z^\infty t^{a-1}e^{-t}dt$, which is useful in many statistical calculations.

`gamma-regularized` returns the regularized incomplete gamma function, i.e. the incomplete gamma function divided by its complete form. `gammac-regularized` returns the complement to the regularized incomplete gamma function or $1 - \Gamma(a,z)$. Similar to the beta function, the inverse of the gamma function is only calculated in its regularized form, and is returned by the procedure `inverse-gamma-regularized`.

`(erf` z`)` procedure
`(erfc` z`)` procedure
`(inverse-erf` s`)` procedure
`(inverse-erfc` s`)` procedure

The error function `erf` returns the integral of the Gaussian distribution, $\text{erf}(z) = 2/\sqrt{\pi}\int_0^z e^{-t^2}dt$. The complementary error function `erfc` simply returns the complement $\text{erfc}(z) = 1 - \text{erf}(z)$. `inverse-erf` gives the inverse of the error function, or the solution for z in $s = \text{erf}(z)$. Similarly, the inverse complementary error function is given by `inverse-erfc`. The error function is important for many calculations in statistics.

`(normal-pdf` x`)` procedure
`(normal-cdf` x`)` procedure
`(poisson-pdf` λ x`)` procedure
`(poisson-cdf` λ x`)` procedure
`(chi-square-pdf` n x`)` procedure
`(chi-square-cdf` n x`)` procedure
`(student-t-pdf` n x`)` procedure
`(student-t-cdf` n x`)` procedure
`(f-ratio-pdf` n_1 n_2 x`)` procedure
`(f-ratio-cdf` n_1 n_2 x`)` procedure

These functions calculate the probability density function (pdf) and cumulative density function (cdf) for a number of common probability distributions. `normal-pdf` and `normal-cdf` calculates the normal distribution probability density and cumulative distribution function respectively at x for a normal (Gaussian) distribution with mean 0 and standard deviation 1. The functions `poisson-pdf` and `poisson-cdf` calculates the density functions at x for a poisson distribution with intensity λ.

`chi-square-pdf` and `chi-square-cdf` calculates the density functions for a χ^2-distribution for n degrees of freedom at x. The Student's-t distribution density functions for n degrees of freedom at x can be calculated using `student-t-pdf` and `student-t-cdf`. Similarly, the density functions of the f-ratio distribution with n_1 and n_2 degrees of freedom is calculated by `f-ratio-cdf` and `f-ratio-cdf`.

`(exp-integral-e` n x`)` procedure
`(exp-integral-ei` x`)` procedure

`exp-integral-e` calculates the exponential integral function $E_n(x) = \int_1^\infty e^{-zt}/t\ dt$. The second exponential integral function `exp-integral-ei` is defined by $Ei(x) = -\int_{-x}^\infty e^{-t}/t\ dt$.

Polynomials

A polynomial $c_0 + c_1x + c_2x^2 + \ldots + c_{n-1}x^{n-1}$ is in gmdl represented as a list of length n containing the constants $c_0 \ldots c_{n-1}$ in order.

`(poly-eval` *poly x*`)` procedure
`(poly-solve` *poly*`)` procedure
`(poly-deriv` *poly*`)` procedure
`(poly-integ` *poly*`)` procedure
`(poly-display` *poly*`)` procedure
`(poly-display` *poly port*`)` procedure

`poly-eval` evaluates the polynomial specified in *poly* at x. `poly-solve` solves the polynomial equation $c_0 + c_1x + c_2x^2 + \ldots + c_{n-1}x^{n-1} = 0$, where the polynomial is represented as described above. The result is a list of the roots of the equation. The roots can be both real and complex numbers. If the degree of the polynomial is higher than 4, an iterative method to find the approximate locations of the roots is used. `poly-deriv` returns the derivative of *poly*, while `poly-integ` returns its integral with constant 0.

The `poly-display` procedures writes the polynomial *poly* in an easy to read format on the port *port*. If no port is specified, the current output port is used.

The discrete Fourier transform

The discrete Fourier transform (DFT) can be effectively calculated using the Fast Fourier Transform (FFT). There are several variants of this algorithm, with different properties and efficiency on different kinds of data and data sizes. The procedures provided in gmdl tries to automatically choose a suitable algorithm for the provided data. Procedures are available for data of any dimensionality.

`(fft` *x*`)` procedure
`(fft` *x f*`)` procedure
`(ifft` *x*`)` procedure
`(ifft` *x f*`)` procedure

The `fft` procedures calculates the forward discrete Fourier transform of a one dimensional complex array of values x of size n, and the `ifft` procedures the inverse or backward discrete Fourier transform. The array x can be a list, a vector, a data object or a matrix. If x is a data object or a matrix, f must be specified which in the case of a data object represents the field the transform should be calculated on, and in the case of a matrix specifies the column. The result is returned as the same or a similar type to the argument x, i.e. if x is a list, a list is returned etc. If x is a data object, the result will be returned as a basic data object, and in the case of a matrix the result is represented as a matrix with n rows and one column.

The actual transform calculated by the procedures correspond to $y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi jk\sqrt{-1}/n}$, where y is the resulting array for the `fft` procedures, and $y_k = \sum_{j=0}^{n-1} x_j e^{2\pi jk\sqrt{-1}/n}$ for the `ifft` procedures. All procedures compute an unnormalized transform, in that there is no coefficient in front of the summation in the DFT. In other words, applying the forward and then the backward transform will multiply the input by n.

The standard “in-order” output ordering is used, i.e. the k-th output corresponds to the frequency k/n (or k/T , where T is the total sampling period). For those who like to think in terms of positive and negative frequencies, this means that the positive frequencies are stored in the first half of the output and the negative frequencies are stored in backwards order in the second half of the output. (The frequency k/n is the same as the frequency $(n-k)/n$.)

7.2. Statistical functions

7.2.1. Measures and descriptive statistics

Some procedures for calculating descriptive statistics and measures on data are included in gmdl. The functions operate on specified fields in data objects and, unless obviously inappropriate, try to calculate the result on all numerical values in the data. This makes it possible to use the procedures on both continuous and discrete data, but might lead to unexpected results when data is mixed. All functions return `#?` when the result cannot be calculated, and signals an error if the specified field is not available in the data object.

`(measure-min` *data field*`)` procedure
`(measure-max` *data field*`)` procedure
`(measure-range` *data field*`)` procedure
`(measure-quartile25` *data field*`)` procedure
`(measure-median` *data field*`)` procedure
`(measure-quartile75` *data field*`)` procedure
`(measure-iqr` *data field*`)` procedure

The procedures `measure-min` and `measure-max` calculate the minimum and the maximum value respectively, while `measure-range` returns the difference between the maximum and minimum values. Note that the range is sensitive to outliers, which may make it an unreliable estimator of the spread of a sample. `measure-quartile25` returns the value of the feature below which 25 % of all values are. In general, this value is also referred to as the first or lower quartile. The central value of the sorted data is given by `measure-median`. If the number of values is even, the mean value of the two central values is used. In general, this value is also referred to as 2nd quartile. The third or upper quartile is given by `measure-quartile75` and is the value of the feature below which 75 % of all values are. `measure-iqr` returns the interquartile range (IQR), which is the difference between the lower and the upper quartile. If there are outliers in the data, then the IQR may be a more representative estimate of the spread of the body of the data than e.g. the standard deviation. To estimate the standard deviation from the IQR, multiply it by a constant $c_{iqr} = 0.7413$.

`(measure-percentile` *data field p*`)` procedure

Calculates a percentile of the data, i.e. the value of the feature below which the fraction p of all values are. Note that p must be a value between 0 and 1.

`(measure-mean` *data field*`)` procedure
`(measure-trimmed-mean` *data field*`)` procedure
`(measure-trimmed-mean` *data field p*`)` procedure
`(measure-geometric-mean` *data field*`)` procedure
`(measure-harmonic-mean` *data field*`)` procedure

The arithmetic mean of the data $\mu = 1/n \sum_{i=0}^{n} x_i$ can be calculated with `measure-mean`. The mean is sensitive to outliers and can be falsified strongly due to them. To deal with this issue, the procedure `measure-trimmed-mean` calculates the arithmetic mean of the data when the fraction $p/2$ highest and lowest values have been removed. p must be a value between 0 and 1. If no value of p is provided, 0.1 is used. `measure-geometric-mean` returns the geometric mean $\mu_g = \prod_{i=0}^{n} x_i^{1/n}$ and `measure-harmonic-mean` the harmonic mean $\mu_h = n/\sum_{i=0}^{n} 1/x_i$.

`(measure-moment` *data field order*`)` procedure

Returns the central moment of the data specified by *order*, where *order* usually is a positive integer. The central moment of order k of a distribution is defined as $m_n = E(x-\mu)^k$, where $E(x)$ is the expected value of x. Note that the first central moment is zero, and the second central moment the maximum likelihood estimation of the variance.

`(measure-rms` *data field*`)` procedure
`(measure-var` *data field*`)` procedure
`(measure-unbiased-var` *data field*`)` procedure
`(measure-var-sample-mean` *data field*`)` procedure

`measure-rms` calculates the root mean square (rms) of the data, $\sqrt{1/n \sum_i x_i^2}$. The variance is a measure for the dispersion of the data. It can be calculated either by `measure-var` which returns the maximum likelihood estimate $var_{mle}(x) = 1/n \sum_i (x_i - \mu)^2$ or the unbiased estimate $var_{unb}(x) = 1/(n-1) \sum_i (x_i - \mu)^2$ using `measure-unbiased-var`. The variance of sample mean $\frac{1}{n} var_{unb}(x)$ can be calculated using `measure-var-sample-mean`.

`(measure-std-dev` *data field*`)` procedure
`(measure-unbiased-std-dev` *data field*`)` procedure
`(measure-stderr-sample-mean` *data field*`)` procedure

The standard deviation, defined as the square root of the variance, is calculated by `measure-std-dev` for the maximum likelihood estimate and `measure-unbiased-std-dev` for the unbiased estimate (see `measure-var` and `measure-unbiased-var` for a closer description). The unbiased estimate of standard error (standard deviation) of sample mean can be calculated using `measure-stderr-sample-mean`.

`(measure-coefficient-var` *data field*`)` procedure
`(measure-mean-abs-dev` *data field*`)` procedure
`(measure-median-abs-dev` *data field*`)` procedure

`measure-coefficient-var` returns the ratio of standard deviation to mean. The mean absolute deviation $\mu_{abs} = 1/n \sum |x_i - \mu|$ is calculated by `measure-mean-abs-dev` and the median absolute deviation, or the median of $|x_i - median(x)|$, by `measure-median-abs-dev`.

`(measure-skewness` *data field*`)` procedure
`(measure-kurtosis` *data field*`)` procedure
`(measure-excess` *data field*`)` procedure

`measure-skewness` calculates the skewness of the data. It can be used to measure the deviation of a frequency distribution from a symmetric distribution. In case of normally distributed features, the skewness is equal to 0. Thus the skewness can be used to test a feature for normal distribution. `measure-kurtosis` returns the kurtosis of the data, and `measure-excess` the excess. Excess specifies whether a distribution of a feature has a kurtosis higher or lower than that of the normal distribution. In case of normally distributed features, the excess delivers the value of 0. As the skewness, the excess can be used to test a feature for similarity to a normal distribution.

`(measure-sum` *data field*`)` procedure

Returns the sum of all numeric values of a feature that are not unknown.

`(measure-nr-discrete` *data field*`)` procedure
`(measure-nr-continuous` *data field*`)` procedure
`(measure-nr-known` *data field*`)` procedure
`(measure-nr-unknown` *data field*`)` procedure

`measure-nr-discrete` and `measure-nr-continuous` simply counts the number of discrete and continuous data respectively. `measure-nr-known` and `measure-nr-unknown` counts the number of known and unknown values in data.

7.2.2. Confidence intervals

The following procedures calculate confidence intervals, which are returned as lists containing two elements. The first element contains the lower confidence interval limit and the second the upper limit. Both procedures operating on measured statistics and data are provided.

(`ci-normal` μ σ *cl*) procedure
(`ci-student-t` μ *sterr dof cl*) procedure

`ci-normal` returns the confidence interval centered at mean μ, with standard deviation σ at confidence level *cl* using the normal distribution. A Student's-t confidence interval can be calculated using `ci-student-t` where μ represents the mean, *sterr* the standard error, dof the degrees of freedom and *cl* the confidence level.

(`ci-chi2` σ^2 *dof cl*) procedure
(`ci-f-ratio` *ratio ndof ddof cl*) procedure

The procedure `ci-chi2` returns the confidence interval interval for the population variance, given the sample variance and degrees of freedom using the χ^2 distribution. σ^2 is the sample variance, dof the degrees of freedom and *cl* the confidence level. `ci-f-ratio` gives the confidence interval for the ratio of population variances *ratio*, given the ratio of sample variances in the numerator and denominator have *ndof* and *ddof* degrees of freedom respectively. Again, *cl* specifies the confidence level.

(`ci-mean` *data field cl*) procedure
(`ci-mean` *data field* σ^2 *cl*) procedure

The confidence interval for the population mean of field $field$ in data *data* with confidence level *cl* can be calculated using `ci-mean`. If the four argument version of the procedure is used and the variance σ^2 is specified the interval is based on the normal distribution, otherwise the Student's-t distribution is used.

(`ci-mean-diff` d_1 f_1 d_2 f_2 $\sigma^2_{eq?}$ *cl*) procedure
(`ci-mean-diff` d_1 f_1 d_2 f_2 σ^2_1 σ^2_2 $\sigma^2_{eq?}$ *cl*) procedure

`ci-mean-diff` returns the confidence interval for the difference between the population mean of field f_1 in data d_1 and the population mean of field f_2 in data d_2. If the variances σ^2_1 and σ^2_2 are specified, the interval is based on the normal distribution, otherwise the Student's-t distribution is used. In both cases, $\sigma^2_{eq?}$ is a boolean variable representing whether the variances are considered equal or not.

(`ci-var` *data field cl*) procedure
(`ci-var-ratio` d_1 f_1 d_2 f_2 *cl*) procedure

`ci-var` give the confidence interval for the population variance of field $field$ in data *data* based on the χ^2 distribution. `ci-var-ratio` returns the confidence interval for the ratio of the population variance of field f_1 in data d_1 to the population variance of field f_2 in data d_2 based on the F-ratio distribution. In both procedures, *cl* sets the confidence level.

7.2.3. Hypothesis testing

Basic hypothesis testing can be performed using the following procedures, all returning the p-value for the specific test. Available are both procedures for calculating p-values for basic distributions and procedures for performing common hypothesis tests on data.

(`ht-normal` *stat ts*) procedure
(`ht-student-t` *stat dof ts*) procedure
(`ht-chi2` *stat dof ts*) procedure
(`ht-f-ratio` *stat ndof ddof ts*) procedure

The procedure `ht-normal` returns the p-value for test statistic *stat* in terms of a normal distribution with mean 0 and unit variance. Similarly, `ht-student-t` returns the p-value in terms of a Student's-t distribution with dof degrees of freedom, and `ht-chi2` the p-value in terms of a χ^2 distribution, also with dof degrees of freedom. `ht-f-ratio` returns the p-value in terms of the f-ratio distribution with $ndof$ degrees of freedom in the numerator and $ddof$ degrees of freedom in the denominator. In all the procedures, *stat* is the test statistic and *ts* is a boolean variable specifying whether the test should be two-sided or not.

(`ht-mean` *data field* μ *ts*) procedure
(`ht-mean` *data field* μ σ^2 *ts*) procedure

`ht-mean` returns the p-value for the test that the population mean of field $field$ in data *data* is equal to μ. If the variance σ^2 is specified, the test is based on the normal distribution. Otherwise, the Student's-t distribution is used. *ts* specifies if the test should be two-sided.

(`ht-mean-diff` d_1 f_1 d_2 f_2 δ $\sigma^2_{eq?}$ *ts*) procedure
(`ht-mean-diff` d_1 f_1 d_2 f_2 δ σ_1 σ_2 *ts*) procedure

The first, seven argument version of `ht-mean-diff` returns, if $\sigma_{eq?}$ is `#f`, the p-value for Welch's approximate t-test that the difference in population means of field f_1 in data d_1 and field f_2 in data d_2 is δ. If $\sigma_{eq?}$ is `#t`, it returns the p-value for the test that the difference in population means is δ based on the Student's-t distribution. $\sigma_{eq?}$ is a boolean variable specifying whether the variances are equal or not. If the variances σ_1 and σ_2 are known and specified, the procedure returns the p-value for the test that the difference in population means is δ based on the normal distribution. Again, *ts* is a boolean variable specifying whether the test should be two-sided or not.

(`ht-var` *data field* σ^2 *ts*) procedure
(`ht-var-ratio` d_1 f_1 d_2 f_2 *ratio ts*) procedure

`ht-var` gives the p-value for the test that the population variance of field $field$ in data *data* is σ^2. `ht-var-ratio` returns the p-value for the test that the ratio of population variances of field f_1 in data d_1 and field f_2 in data d_2 is *ratio*. *ts* specifies whether the test is two-sided or not.

7.3. Random number generation

Proper random number generation is an essential part of e.g. several numerical and statistical methods. gmdl provides a random number generator type that can be used to provide a consistent interface independent of the actual algorithm used for number generation.

7.3.1. Random number generator functionality

Basically, most random number generators available in gmdl provide a new integer value between the minimum and maximum output value for the algorithm with equal probability every time a new random number is requested. A number of functions used to convert this number into a perhaps more usable form are also provided. Most random number generators have to be provided with a random seed, or a starting point for the pseudo-random algorithm. The same seed will always provide the same sequence of values for these generators. Since constructing and seeding a random number generator might be excessive work for some very small applications, a standard number generator

```
rndgen-default
```

is always available unless its binding is changed by the user. It is seeded at the start of the program with a "random" number (see `rndgen-random-seed!` below). The algorithm used by this default random number generator is implementation dependent.

`(make-rndgen` *type*`)` procedure

This procedure is used to create a new random number generator using the algorithm specified in *type*. The *type* argument must be one of the following symbols, each specifying a random number generator algorithm:

```
system
sys-entropy
mt19937
ranlxs
ranlxd
```

The algorithms themselves are described in more detail in section 7.3.2.

`(rndgen?` *obj*`)` procedure

The procedure `rndgen?` returns `#t` if and only if *obj* is a random number generator, otherwise `#f` is returned.

`(rndgen-max` *rndgen*`)` procedure
`(rndgen-min` *rndgen*`)` procedure

`rndgen-max` returns the largest value that `rndgen-get` can return. Similarly, `rndgen-min` returns the smallest value that `rndgen-get` can return. Usually this value is zero, but there are e.g. some generators with algorithms that cannot return zero, and for these generators the minimum value typically is one.

`(rndgen-seed!` *rndgen seed*`)` procedure
`(rndgen-random-seed!` *rndgen*`)` procedure

The procedure `rndgen-seed!` initializes (or "seeds") the random number generator *rndgen* with *seed*, which must be an integer value equal to or larger than zero. If the generator is seeded with the same value of *seed* on two different runs, the same stream of random numbers will generally be generated by successive calls to the routines below. If different values of *seed* are supplied, then the generated streams of random numbers should be completely different. This does not hold for all random number generators though, some use a different approach to random number generation in which case the procedures for setting seeds does nothing. If the seed *seed* is zero then the standard seed from the original implementation is used instead.

`rndgen-random-seed!` initializes the random number generator *rndgen* with a "random" number, either from the device /dev/urandom if available or based on the current time of the system clock. Please note that using time instead of /dev/urandom is *much* weaker, but is also more portable.

`(rndgen-get` *rndgen*`)` procedure

The `rndgen-get` procedure returns a random integer from the generator *rndgen*. The minimum and maximum values depend on the algorithm used, but all integers in the range $[min, max]$ are equally likely. The values of *min* and *max* can be determined using the auxiliary procedures `rndgen-max` and `rndgen-min` respectively.

`(rndgen-uniform` *rndgen*`)` procedure
`(rndgen-uniform-pos` *rndgen*`)` procedure

The procedure `rndgen-uniform` returns a real number uniformly distributed in the range $[0, 1)$. The range includes 0.0 but excludes 1.0. The value is typically obtained by dividing the result of `rndgen-get` by $\texttt{rndgen} - \texttt{max} + 1.0$ in (at least) double precision. Some generators compute this ratio internally so that they can provide floating point numbers with better randomness.

`rndgen-uniform-pos` returns a real number uniformly distributed in the range $(0, 1)$, excluding both 0.0 and 1.0. The number is obtained by sampling the generator with the algorithm for uniform numbers until a non-zero value is obtained. You can use this function if you need to avoid a singularity at 0.0.

`(rndgen-uniform-exact` *rndgen n*`)` procedure
`(rndgen-binary` *rndgen prob*`)` procedure
`(rndgen-boolean` *rndgen prob*`)` procedure

The procedure `rndgen-uniform-exact` returns a random integer from 0 to $n-1$ inclusive. All integers in the range $[0, n-1]$ are equally likely, regardless of the generator used. An offset correction is applied so that zero is always returned with the correct probability, for any minimum value of the underlying generator.

`rndgen-binary` returns 1 with the specified probability *prob* and 0 with probability $1 - prob$. Similarly, `rndgen-boolean` returns `#t` with the specified probability *prob* and `#f` with probability $1 - prob$.

`(rndgen-gaussian` *rndgen* μ σ`)` procedure

The `rndgen-gaussian` procedure returns Gaussian distributed random values with the specified mean μ and standard deviation σ.

7.3.2. Random number generator algorithms

The `system` generator uses the random function provided through standard system libraries. Use this generator if you want to make sure that your application is if not perfectly compatible then at least functional on all platforms.

The `sys-entropy` generator uses the device /dev/urandom to produce random numbers. It gathers environmental noise from device drivers, etc., and returns good random numbers, suitable for cryptographic use. Besides the obvious cryptographic uses, these numbers are also good for seeding TCP sequence numbers, and other places where it is desirable to have numbers which are not only random, but hard to predict by an attacker. In short, it produces good, cryptographically strong random numbers, but since it is based on gathering "noise" from the system, truly random numbers cannot be drawn faster than they can be generated by the "noise" in the system. For simulations where a large number of random numbers is needed, the use of this generator is not recommendable.

The MT19937 generator of Makoto Matsumoto and Takuji Nishimura (`mt19937`) is a variant of the twisted generalized feedback shift-register algorithm, and is known as the "Mersenne Twister" generator. It has a Mersenne prime period of $2^{19937} - 1$ (about 10^{6000}) and is equi-distributed in 623 dimensions. It has passed the DIEHARD statistical tests [14]. It uses 624 words of state per generator and is comparable in speed to most other generators. The original generator used a default seed of 4357 and choosing *seed* equal to zero in `random-set!` reproduces this. For more information, see [17]. The `random19937` generator uses the second revision of the seeding procedure published by the two authors above in 2002. The original seeding procedures could cause spurious artifacts for some seed values.

The generator `ranlxs` is a second-generation version of the RANLUX algorithm, which produces "luxury random numbers" [15], [16]. This generator provides single precision output (24 bits). It uses double-precision floating point arithmetic internally and can be significantly faster than the integer version of RANLUX, particularly on 64-bit architectures. The period of the generator is about 10^{171}. The algorithm has mathematically proven properties and can provide truly decorrelated numbers at a known level of randomness.

The `ranlxd` generator produces double precision output (48 bits) from the `ranlxs` generator.

8. Matrix calculations

8.1. The matrix type

gmdl offers a fast matrix type for basic numerical calculations and linear algebra. Since the matrix type is intended for numerical calculations, it can only hold numbers. Strings, lists, etc. cannot be stored in a matrix.

Many functions operating on matrices are provided in both a variant that returns the result of the calculation as a new matrix, and a variant that performs the calculation in place, modifying the existing matrix and not returning a value. The reason is that most numerical calculations can benefit greatly concerning speed and memory use from only allocating and copying memory when necessary.

Also, try to use the built-in matrix operations and mapping functions as often as possible instead of accessing the matrix element-wise, since the built in routines usually are faster. When implementing speed-critical functionality, please take a moment to consider what approach and procedures to use. Most of the time it becomes rather obvious what the best solutions are.

8.2. Matrix functionality

`(make-matrix` *n type transp*`)` procedure
`(make-matrix` *rows cols type*`)` procedure

These procedures are used to create new matrices. Contrary to most other notation in this document, the arguments *type* and *transp* are optional in both of the procedures.

The first version of the procedure require that n is an integer value larger than zero. If no further arguments are given the procedure returns a matrix with n rows and 1 column. *type* can be specified as "'transpose", "'zero", "'one" or "'identity". If "'zero" or "'one" is specified the returned matrix is initialized to 0 or 1 respectively, and if "'transpose" is given either as *type* or *transp* the returned matrix will contain n columns and 1 row. If the second argument is "'identity", the identity matrix with n rows and columns is returned.

The second version of the procedure require that both *rows* and *cols* are integer values larger than zero. The returned matrix will contain *rows* rows and *cols* columns, and *type* can be optionally specified as "'zero" or "'one" with the same effects as described above.

Note that no initialization of the created matrix is done unless explicitly requested for efficiency reasons.

`(matrix?` *obj*`)` procedure

The procedure `matrix?` returns `#t` if and only if *obj* is a matrix, otherwise `#f` is returned.

`(matrix-display` *m*`)` procedure
`(matrix-display` *m port*`)` procedure

`matrix-display` writes the matrix m to port *port*. If no port is specified, the current output port is used.

`(matrix-rows` *m*`)` procedure
`(matrix-cols` *m*`)` procedure
`(matrix-size` *m*`)` procedure

`matrix-rows` returns the number of rows in the matrix m, while `matrix-cols` returns the number of columns. `matrix-size` returns a list containing the number of rows and columns of the matrix m.

`(matrix-ref` *m row col*`)` procedure
`(matrix-set!` *m row col val*`)` procedure

The procedure `matrix-ref` returns the value of the element on row *row* and column *col* in matrix m. To set an element in a matrix m to a specific value, `matrix-set!` is used. Again, *row* specifies the row, *col* the column and *val* the value to which the element will be assigned. Trying to access an element outside the allocated matrix signals an error.

`(matrix-square?` *m*`)` procedure
`(matrix-zero?` *m*`)` procedure
`(matrix-identity?` *m*`)` procedure
`(matrix-symmetric?` *m*`)` procedure
`(matrix-orthogonal?` *m*`)` procedure
`(matrix-normal?` *m*`)` procedure
`(matrix-upper-tria?` *m*`)` procedure
`(matrix-lower-tria?` *m*`)` procedure

All of these procedures test a matrix m for a specific property, returning `#t` if m has the property and `#f` otherwise. `matrix-square` determines whether the matrix is square or not, i.e. if the number of columns equal the number of rows. `matrix-zero?` returns true if all elements in the matrix are 0, and `matrix-identity?` returns true if the matrix is the identity matrix. Testing if the matrix is symmetric, orthogonal or normal can be performed using `matrix-symmetric?`, `matrix-orthogonal?`, and `matrix-normal?` respectively. `matrix-upper-tria?` determines if the matrix is upper triangular, i.e. if the elements below the diagonal are 0. Similarly, `matrix-lower-tria?` determines if the matrix is lower triangular.

`(matrix+` m_1 m_2`)` procedure
`(matrix+!` m_1 m_2`)` procedure
`(matrix-` m_1 m_2`)` procedure
`(matrix-!` m_1 m_2`)` procedure
`(matrix*` m_1 m_2`)` procedure
`(matrix*!` m_1 m_2`)` procedure

These procedures perform basic arithmetics on matrices. Common for all these procedures is that m_1 must be a matrix, while the second argument m_2 can be either a matrix or a number. If m_2 is a matrix, normal matrix arithmetics is performed, while if it is a number the operation is carried out element-wise on all elements. Although this is natural for multiplication, it might be counter-intuitive for addition and subtraction but the operation is allowed for convenience and completeness. `matrix+` performs addition, `matrix-` subtraction and `matrix*` multiplication, returning new matrices containing the result. `matrix+!`, `matrix-!` and `matrix*!` perform the same operations but store the result in the matrix m_1 directly, modifying m_1 while nothing is returned. The addition and subtraction procedures require that m_2 either is a number or a matrix of the same size as m_1. Similarly, the multiplication procedures require that m_2 is either a number or a matrix with the same number of rows as the number of columns in m_1.

`(matrix*@` m_1 m_2`)` procedure
`(matrix*@!` m_1 m_2`)` procedure
`(matrix/@` m_1 m_2`)` procedure
`(matrix/@!` m_1 m_2`)` procedure

The procedures above perform multiplication and division element-wise, i.e. a certain element in m_1 is multiplied or divided by the corresponding element in m_2. m_1 and m_2 must be matrices of the same size. `matrix*@` performs element-wise multiplication and `matrix/@` performs division, returning the result as a new matrix. The procedures `matrix*@!` and `matrix/@!` perform the same calculations in place, modifying and storing the result in m_1 without returning any value.

`(matrix-map` *m proc*`)` procedure
`(matrix-map!` *m proc*`)` procedure

It is sometimes useful to be able to map a function element wise over all elements in a matrix. An example could be creating a new matrix with the square root of all elements in the original matrix or setting all elements in a matrix to the natural logarithm of its present value. Two mapping procedures for matrices are offered: `matrix-map` and `matrix-map!`, the difference being that the first returns a new matrix while the second performs the operation in place, thereby modifying the matrix. In both cases, m must be a matrix and *proc* a procedure taking one argument (the value of the element) and returning a single number. If this is not the case, an error will be signaled.

`(matrix-transp` *m*`)` procedure
`(matrix-transp!` *m*`)` procedure

The transpose of matrix m is returned by procedure `matrix-transp`. `matrix-transp!` calculates the transpose in place, modifying m.

`(matrix-trace` *m*`)` procedure
`(matrix-det` *m*`)` procedure

`matrix-trace` returns the trace, i.e. the sum of the diagonal elements, of matrix m. `matrix-det` calculates the determinant of m.

`(matrix-inv` *m*`)` procedure
`(matrix-inv!` *m*`)` procedure
`(matrix-solve!` *A b*`)` procedure

`matrix-inv` calculates and returns the inverse of the general matrix m if possible, that is if the determinant is not 0. The procedure $m$atrix-inv! performs the same calculation but modifies the existing matrix, replacing it with its inverse. `matrix-solve!` solves the equation system $Ax = b$, where b is modified to contain the result of the calculation. b must be a vector with one column and the same number of rows as the matrix A has columns.

`(matrix-chol-inv` *m*`)` procedure
`(matrix-chol-inv!` *m*`)` procedure
`(matrix-chol-solve!` *A b*`)` procedure

These procedures are used in the same way as the general inversion and solving procedures described earlier, but provide enhanced precision and calculation speed if we know that the matrix m or A is symmetric. The calculations are based on Cholesky factorization and will produce erroneous results if applied to non-symmetric matrices.

`(matrix-eigen` *m*`)` procedure

The procedure `matrix-eigen` calculates the eigenvalues and eigenvectors for the matrix m. It returns a list containing two matrices. The first matrix contains the eigenvalues for m sorted in decreasing order, and the second matrix the corresponding eigenvectors, each column representing one eigenvector.

`(number->matrix` *n*`)` procedure
`(list->matrix` *lst*`)` procedure
`(vector->matrix` *vec*`)` procedure

`number->matrix` converts the number n to a 1 by 1 matrix. The procedures `list->matrix` and `vector->matrix` converts lists and vectors to a matrix representation. They both return column vectors of size n, where n is the number of elements in *lst* or *vec*.

NOTES

Changes to gmdl

This section lists the changes that have been made to gmdl since the original version of this report was published.

- All functions now use the standard Scheme data types, and the `d-entry` data type for interfacing with MDL functions has been made obsolete.
- Many of the functions have gained longer and more cumbersome names, but now have a structure that makes understanding their relationships and what they operate on easier. The new names also remove some conflicts in the global name space.
- While gmdl gained a comprehensive set of procedures for efficient data manipulation and scripting, the internals and basic primitives also changed to facilitate for these higher level procedures. A scripting facility was also added to simplify the transformation of large and complex data sets.

ADDITIONAL MATERIAL

The gmdl home page at

`http://www.sics.se/~dgi/gmdl/`

contains extensions to and information on gmdl, as well as manuals, papers, programs, and other material related to gmdl.

EXAMPLE

These examples are the same as in the Revised^5 Report on the Algorithmic Language Scheme [2]. They are included here since they describe basic features such as defining and using procedures.

`Integrate-system` integrates the system

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

of differential equations with the method of Runge-Kutta.

The parameter `system-derivative` is a function that takes a system state (a vector of values for the state variables $y_1, \ldots, y_n$) and produces a system derivative (the values $y'_1, \ldots, y'_n$). The parameter `initial-state` provides an initial system state, and `h` is an initial guess for the length of the integration step.

The value returned by `integrate-system` is an infinite stream of system states.

```
(define integrate-system
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`Runge-Kutta-4` takes a function, `f`, that produces a system derivative from a system state. `Runge-Kutta-4` produces a function that takes a system state and produces a new system state.

```
(define runge-kutta-4
  (lambda (f h)
    (let ((*h (scale-vector h))
          (*2 (scale-vector 2))
          (*1/2 (scale-vector (/ 1 2)))
          (*1/6 (scale-vector (/ 1 6))))
      (lambda (y)
        ;; y is a system state
        (let* ((k0 (*h (f y)))
               (k1 (*h (f (add-vectors y (*1/2 k0)))))
               (k2 (*h (f (add-vectors y (*1/2 k1)))))
               (k3 (*h (f (add-vectors y k2)))))
          (add-vectors y
            (*1/6 (add-vectors k0
                               (*2 k1)
                               (*2 k2)
                               k3))))))))

(define elementwise
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (size proc)
    (let ((ans (make-vector size)))
      (letrec ((loop
                (lambda (i)
                  (cond ((= i size) ans)
                        (else
                         (vector-set! ans i (proc i))
                         (loop (+ i 1)))))))
        (loop 0)))))

(define add-vectors (elementwise +))

(define scale-vector
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

`Map-streams` is analogous to `map`: it applies its first argument (a procedure) to all the elements of its second argument (a stream).

```
(define map-streams
  (lambda (f s)
    (cons (f (head s))
          (delay (map-streams f (tail s))))))
```

Infinite streams are implemented as pairs whose car holds the first element of the stream and whose cdr holds a promise to deliver the rest of the stream.

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

The following illustrates the use of `integrate-system` in integrating the system

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

which models a damped oscillator.

```
(define damped-oscillator
  (lambda (R L C)
    (lambda (state)
      (let ((Vc (vector-ref state 0))
            (Il (vector-ref state 1)))
        (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
                (/ Vc L))))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and interpretation of Computer Programs, second edition.* MIT, Cambridge MA, 1996.

[2] Richard Kelsey, William Clinger, and Jonathan Rees, editors. The revised^5 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 33(9), pages 26–76, 1998.

[3] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[4] S. R. Bourne. *The UNIX System.* Addison-Wesley, Reading MA, 1982.

[5] Nachum Dershowitz and Edward M. Reingold. *Calendrical Calculations.* Cambridge University Press (1997).

[6] T. Cover. Estimation by The Nearest Neighbor rule. In *IEEE Transactions on Information Theory* 14(1), pages 50–55, 1968.

[7] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. In *Psychological Review* 65, pages 386–408, 1958.

[8] M. L. Minsky and S. A. Papert. *Perceptrons.* MIT Press, Cambridge MA, 1969.

[9] Simon Haykin. *Neural Networks: A comprehensive foundation.* Prentice-Hall, Upper Saddle River, New Jersey, 1994.

[10] D. Rumelhart et. al. Learning internal representations by error propagation. In Rumelhart D. E., McClelland J. L., and the PDP Research Group (eds.), *Parallel Distributed Processing, Explorations in the Microstructure of Cognition*, volume 1, chapter 8. MIT Press, Cambridge, MA, 1986.

[11] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997.

[12] Leo Breiman. *Bagging Predictors.* Technical Report No. 421, Department of Statistics, University of California, Berkely, September 1994.

[13] Jackie Neider, Tom Davis, and Mason Woo. *Opengl Programming Guide: The Official Guide to Learning Opengl, Release 1.* Addison-Wesley, 1993.

[14] G. Marsaglia. *The Marsaglia random number CDROM with the DIEHARD battery of tests of randomness.* Distributed by the author (geo@stat.fsu.edu) from Florida State University, 1996.

[15] M. Luscher. A portable high-quality random number generator for lattice field theory simulations. In *Computer Phys. Commun.* 79, pages 100–110, 1994.

[16] F. James. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Luscher. In *Computer Phys. Commun.* 79, pages 100–110, 1994.

[17] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. In *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1 (Jan. 1998), pages 3–30.

[18] Anders Holst. *The Use of a Bayesian Neural Network Model for Classification Tasks.* PhD Thesis TRITA-NA-P9708, dept. of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden.

[19] Daniel Gillblad and Anders Holst. Dependency Derivation in Industrial Process Data. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 599–602.

[20] Daniel Gillblad and Anders Holst. Hierarchical Graph Mixtures.

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS, KEYWORDS, AND PROCEDURES

The principal entry for each term, procedure, or keyword is listed first, separated from the other entries by a semicolon.