Akademie věd České republiky
Ústav teorie informace a automatizace

Academy of Sciences of the Czech Republic
Institute of Information Theory and Automation

RESEARCH REPORT

J. ANDRÝSEK, M. PIŠTĚK, V. ŠMÍDL, O. ŠTERBÁK, M. TKÁČ,
M. TÝNOVSKÝ AND I. VÁŇOVÁ

Mixtools 3000 Foundation

No. 2179 December 2006

MŠMT 1M0572, AV ČR 1ET 100 750 401

ÚTIA AVČR, P.O.Box 18, 182 08 Prague,
Czech Republic
Fax: (+420)286890378, http://www.utia.cz, E-mail:
utia@utia.cas.cz

Contents

Chapter 1

Aim of the paper

This paper is a first step in a larger project that aims is to create a toolbox for support of dynamic distributed decision making under uncertainty. This toolbox is designed as a new generation of the software platform that serves for testing of various decision-making-related algorithms. The new framework will replace older system MIXTOOLS[1], from which it inherits the main low-level algorithmic base. However, the upper level of the algorithmic structure was completely redesigned and thus the system was renamed to MIXTOOLS3000.

When it became apparent that the old system can not handle new class of problems, full analysis of the problem was undertaken and its results published [2]. The main conclusion was that MIXTOOLS3000 will be based on the old MIXTOOLStoolbox, and it will be written in MATLABand ANSI C in order to re-use most of the existing code. On the other hand, it was decided to use object-oriented approach in order to achieve better extensibility and flexibility of the code. These two tasks are somewhat contradictory because MATLABsupport for object-oriented programming is very weak. Therefore, we decided to build our own implementation of the main feature of the object-oriented programming.

Brief overview of the software analysis is summarized in Chapter 2. The main solution of object-oriented Matlab programming is presented in Chapter 3. The use of this approach is explained in Chapter **??**. An important part of the proposed solution is a mechanism for quality control, which is presented in Chapter 5.

Chapter 2

Background

In this Chapter, we summarize the results of [2] that are relevant for our task. Specifically, we review the basic requirements on the toolbox which were used as the main guidelines in selection between alternative solutions.

2.1 Requirements on the toolbox

We recognize that new optimal algorithms can only arise from proper use of the mathematical theory. The theory of statistical decision making was developed in [3], elaborated to engineering form by [4, 5] and updated in [6]. Hence, we require the software to be as close to this theory as possible. However, translation of the theory into software is a challenging task, since the theory describes real world in terms of abstract mathematical structures, such as density functions and functionals. The process of decision making is then defined in terms of operations on these structures. The set of all possible structures is extremely rich and operations over many of them are not computationally tractable. Therefore, each attempt at software analysis of the theory has to, inevitably, restrict its scope to a certain, computationally tractable, sub-set. This initial restriction is a very important step, since it represents a trade-off between modelling abilities of the toolbox and simplicity of its implementation.

The analysis distinguishes two principal parts of the software package:

Framework which is a general description of the distributed dynamic DM. It specifies data structures, and algorithms.

Implementation of the framework in a specific programming language.

Specification of the framework should be independent of its implementation. This can be achieved by the use of general modelling language in which the data structures and algorithms will be described. Various implementations of the framework may arise. These implementations may be application-specific with different intellectual property rights. However, all implementation should follow the framework specifications to be mutually compatible.

2.1.1 Software framework

As mentioned in the previous Section, full generality of the theory cannot be captured by any software. Therefore, we seek such a class of models that is as general as possible, but at the same time computationally tractable and applicable in real-life.

In this Section, we summarize the necessary requirements for a candidate families:

Requirements 1 (Requirements on software framework) *The considered framework should support:*

1. ***Multivariate dynamic models;*** *the environment we intend to work with is expected to have: (i) both discrete and continuous variables, with mutual dependencies between them, and (ii) dynamic nature, i.e. present behavior depends on the previous observations.*
2. ***Lego-like concept;*** *the software framework should provide basic building blocks that can be seamlessly composed into complicated structures. These blocks should: (i) cover data structures corresponding to basic structural elements in the theory, (ii) include composition tools corresponding to operators in the theory, and (iii) allow easy addition of new types of all elements (within the framework).*
3. ***Design of DM strategies;*** *the framework should allow re-evaluation of DM strategy at any time. I.e. all elements that are necessary for this task must be supported by the software.*
4. ***User interface;*** *user is a human being who determines the desired behaviour of the overall system. Therefore, the software framework should provide tools of interaction with non-expert users: (i) users' aims, available knowledge, used model and constraints should be made in user's terms independently of the processing method, (ii) Presentation of results has to be close to the application domain.*

2.1.2 Software implementation

The software is to be used in full-scale applications, which induces high requirements for quality and maturity of the code. The following points seem to be indispensable.

Requirements 2 (Requirements on Implementation) *The supported development platform should be:*

1. ***numerically stable and efficient;*** *which is important in industrial applications,*
2. ***portable to a wide range of platforms****; it should run on anything from a supercomputer to an industrial microcontroller,*
3. ***suitable for implementation of object-oriented algorithms****; which is necessary for seamless implementation of the framework which will be defined using object-oriented methodology,*
4. ***able reuse the code that is already available****; most of the development in the area was done in* MATLAB *and C. The new tool should be easily connectible to these tools,*
5. ***economically affordable****; applications of the framework in non-for-profit organizations is also considered, therefore we should not rely on any expensive proprietary tools,*
6. ***user-friendly****; allowing easy testing of new algorithms and tuning knobs,*

These requirements are heavily based on the approach that was developed and incrementally improved by the AS group. A lot of expertise and effort has been invested into the development of the bridge between academy and industry. Seamless transition of expertise was done sequentially in: (i) MATLAB, as academic environment for rapid development and testing, (ii) pure ANSI C, for portability and implementation in MATLAB-free applications, and (iii) Mex files, for connectivity of MATLAB and ANSI C.

2.2 State of the art

In this Section, we review the existing solutions in the light of the requirements described above.

We have tested software packages from the following research areas:

Optimal advising: package Mixtools,
`http://guest:guest@marabu.utia.cas.cz:1800/svn/mixtools/`,

Bayesian networks: package BNT,
`http://www.cs.ubc.ca/~murphyk/Software/BNT/bnt.html`,

Graphical models: project *gR*,
`http://www.r-project.org/gR/`,

Bayesian neural networks: project *fbm*,
`http://www.cs.toronto.edu/~radford/fbm.software.html`,

Nonlinear filtration: project *nftool*, `http://control.zcu.cz/nftools/`,
and *rebel* `http://choosh.ece.ogi.edu/rebel/`

Bayesian decision making: project *IND*,
`http://ic.arc.nasa.gov/ic/projects/bayes-group/ind/`

Here, we have selected only the most advanced, *freely available* tools. There are many more software projects in areas of state-space modelling and Bayesian estimation, see e.g. survey on `http://leuther-analytics.com/bayes/free-bayes-software.html`. However, the number of tools for Bayesian decision-making is rather limited. Evaluation of all projects with respect to our requirements (Requirements 1, and 2) were discussed in [2] in detail.

Most of the projects are implemented in MATLAB or C. However, implementation is the less important factor, since most of the packages fails our requirements on the framework. All the studied tools have some advantages and disadvantages, the most common disadvantage is that the application area of each tool is too narrow for our purpose. All projects do well within their field of expertise, but none of them fulfills all of our requirements.

From those tools, we choose Mixtools and BNT for further detailed analysis, since these meet most of our requirements.

2.2.1 Mixtools

Mixtools is a MATLAB Toolbox developed in our department specifically for the purpose of optimal Bayesian advising [7, 6, 8].

The model: The basic observation model considered within this toolbox is a mixture of ARX models (autoregressive models with exogenous input). Both continuous and discrete observations are supported via mixture of Gaussian, and Markov chain regression models, respectively.

Lego-like concept: The basic structure is a mixture of ARX (or Markov) models. It is possible to define new-type of components in the mixture, however it requires relatively lot of effort. Composition tools are ready only for creation and manipulation with mixtures. Another disadvantage of this package is centralized data-handling mechanism.

Decision making: the DM strategy is designed using *fully probabilistic approach*. This approach is capable recursive evaluation of DM strategy.

User interface: many tools for support of non-informed user are available.

Implementation: the toolbox is implemented in MATLAB, Mex, and ANSI C programming environments.

2.2.2 BNT

The BNT package implements state-of-the-art algorithms for Bayesian networks [9]. This toolbox has two principal distinctions from other tools typically used in the area of graphical and Bayesian-network modelling. First, autoregressive (i.e. dynamic) models are considered. Second, it includes basic support for decision making.

The model: the basic model is a dynamic Bayesian network. In principle, it is a pdf restricted by the assumptions of conditional independence between some variables. These assumptions are described in terms of graph, where pdfs are nodes in the graph, and edges denote mutual dependence of nodes. Currently, the following types of nodes are supported: Gaussian models, hidden Markov models, perceptron neural networks, and discrete models.

Lego-like concept: the conditional independence assumption is an excellent tool for separation of basic building blocks and their composition. Namely, nodes on the graph represent the basic building blocks, and the graph (network) of their dependence is the composition tool. This way, complex structures can be easily created.

Decision making: is made via so called utility functions, which are assigned to each node in the graph. This mechanism supports both, recursive evaluation of DM strategy, and formalization of the communication strategy as DM problem. However, it is readily available only for one-step-ahead DM. Extensions to longer DM horizon is possible, however, it would be computationally inefficient, since it requires to build the network for all variables within the DM horizon.

User interface: is quite limited. Only expert users of the toolbox are supported. However, additional projects are trying to fill the gap, such as BNTeditor `http://bnt.insa-rouen.fr/BNTEd.html`.

Implementation: the toolbox is primarily written in MATLAB, however, a preliminary port to C++ is available at Intel, `http://www.intel.com/research/mrl/pnl/`.

2.2.3 Summary

None of the currently available toolboxes for Bayesian decision making matches our requirements (Requirements 1, and 2). Therefore, we decided to create in this text, we develop an analysis of an the desired toolbox for distributed Bayesian decision making.

We intend to exploit as much experience accumulated in the current software packages as possible. We will draw inspiration from both: Mixtools, and BNT packages. The Mixtools are more mature in their technology of implementation, and design of DM strategy. On the other hand, the range of models supported by BNT is impressive and unrivaled.

2.3 Implementational tools

The basic requirements of the framework—namely extensibility, flexibility and intuitiveness of its use—have been raised in computer science many time before. One approach that was designed to meet these requirements is known as object-oriented (OO) approach [10]. However, the traditional line of implementation—i.e. MATLAB and ANSI C—has no native support for OO approach. Hence, our requirements are mutually incompatible. This conflict could be solved by two possible approaches: (i) re-implement all available code in a native OO language, or (ii) creation of custom support of OO approach in the traditional line.

	numerical stability and efficiency	portability	OO approach	code reuse	economically affordable	user friendly
MATLAB	+/−	+/−	−	+	−	+
ANSI C	+	+	−	+	+	−
C++	+	+/−	+	−	+	−
JAVA	−	+/−	+	−	+	+/−

Table 2.1: Comparison of programming languages.

A survey of OO languages was done in [2] and its results are summarized in Table 2.1.
Here, we comment on each of the studied languages:

MATLAB: has some form of support for object-oriented approach, however, its support is very poor and inefficient. Therefore, we can not consider MATLAB as ready for OO approach. Its main attraction is user-friendliness, its main drawback is the lack of computational efficiency and economical affordability.

Java: Java is a popular OO programming language. It is well supported in the MATLAB environment, namely Java classes can be called from MATLAB. Its main attraction is support of OO approach and connectivity to MATLAB. Its main drawback is computational efficiency.

The following experiment with library JAMA library (`http://math.nist.gov/javanumerics/jama/`) was performed on Pentium 400MHz:

Test: 100 multiplications of matrix 100x100: on Pentium 400MHz

Results: MATLAB 0.8s, JAMA 4s, calling JAMA from MATLAB (>10s)

ANSI C: is a low-level programming language. Its main advantages are computational efficiency and portability. Its main drawback is the lack of support for OO approach and user-friendliness. However, the latter can be remedied by interoperability with MATLAB via the technology of Mex files.

C++: is a re-design of the C language to support the OO approach. Therefore is has all the advantages of ANSI C, except for portability and connection with MATLAB. Its main drawback is therefore the lack of user friendliness.

The overall conclusion was that none of the above languages is suitable for our needs. We have to use a combination of languages to meet most of our requirements. In this situation, the requirement on continuity of research starts to be the dominant factor in the selection process. Therefore, we will continue to use the traditional tool-chain. Preliminary considerations listed in [2] suggest that it is possible to support basic properties of the OO approach consistently in MATLAB and ANSI C.

2.4 Aims and tasks of the work

The aims of this work can be summarized as follows:

1. create a methodology how to extend the traditional tools (MATLAB–MEX–ANSI C) for support of OO approach. The strength of MATLAB—such as the ease of notation of mathematical formulas, and their presentation—should not be affected by the new extension. In other words, the OO properties are required only for higher-level structures, low-level structures (i.e. matrices and vectors) should not be changed,

2. create a methodology of testing if the implementations in two languages (i.e. MATLAB and ANSI C) provide the same results,

3. implement the basic structures of Bayesian decision making: (i) random variables, (ii) distributions of random variables (pdf), and (iii) functions of random variables.

These tasks will be addressed as follows: In Chapter 3, we introduce the runtime object framework. We describe its features and how they are implemented. We also present exhaustive tutorial on creating your own class. Chapter **??** guides you through producing executable MEX file. Chapter 5 discusses methods for checking correctness of produced scripts or correspondence of MATLAB and ANSI C implementations. A separate document "MIXTOOLS 3000 Class Reference" contains documentation of the central part of MIXTOOLS 3000 – the provided classes and their methods.

Chapter 3

Runtime object framework

3.1 Motivation

The original Mixtools library was being developed continuously throughout several years. Functionality was extended usually by adding new functions or patching the existing ones. Over the years the library became hard to maintain, extend and use. For MIXTOOLS 3000 we decided to use object-oriented approach. We expect it will lead to better readability, user-friendliness, maintainability and extensibility. Beside the basic principles of object oriented approach – inheritance, polymorphism and encapsulation, we needed to achieve runtime effectiveness, i.e. to make the library perform reasonably fast. We also wanted to introduce a method for runtime parameter checking for methods as this is not present in MATLAB.

3.2 Introduction

The first major task in implementing MIXTOOLS 3000 was design and implementation of a runtime object framework (ROF). It is a foundation of MIXTOOLS 3000 because it provides support for objects. As such, it is also the first thing that needs to be explained to understand how MIXTOOLS 3000 works. The specification did not require a fully featured ROF, but rather a simple, effective, easy-to-implement framework that would provide basic object functionality.

The MATLAB implementation and ANSI C implementation of any method must be interchangeable to allow mixed use of MATLAB M-files and Mex-files. It requires that the object frameworks in MATLAB and C are compatible enough. That means the design of objects and realization of virtual methods have to be very similar in both languages.

When designing the ROF, we had two alternatives. Either to use the object framework built in MATLAB 6.5 and implement the same model in C. Or to create our own customized object framework and implement it in both MATLAB and C.

3.2.1 Why did we reject the MATLAB framework

Some experiments were carried out to test the capabilities and find possible problems with the MATLAB framework. The most important issue was the possibility of porting it to ANSI C for transparent use in MEX-files as well as in standalone MATLAB-free solution.

The MATLAB framework is documented only briefly. We were unable to find how to correctly create an instance of a class in MEX-file so that this instance could then be used in MATLAB environment on-the-fly – without saving the instance first and then loading it.

This problem itself would be serious enough to reject usage of MATLAB framework. However, there are other issues. This framework hasn't been widely accepted and a new framework was in development for future MATLAB releases. Even as of writing this document, the new framework is in Beta stage and not yet stabilized.

Implementation of inheritance introduces another obstacle. When using inheritance in MATLAB, the child object contains the parent object in a property with the name of the parent class. That really breaks the simple and universal way of accessing data properties and forces us to create and use special get and set methods for each class.

3.3 Main features

3.3.1 Data attributes and class information

MATLAB version

Objects in MIXTOOLS 3000 are implemented using MATLAB structures. MATLAB structure is a class (or data type) that enables to store named data attributes of any type in one logical entity or group. MATLAB does not allow to define specific

structured types. Instead attributes are added and removed from a structure at runtime as needed. MATLAB also does not specify a standard way to associate methods with structures – except for the mentioned object model.

Data attributes of our objects are represented as fields of MATLAB structures.

In MIXTOOLS 3000 a unique number is assigned to each class. This mapping is generated automatically by a script named `create_files.m` called after adding a new class. The mapping is stored in files – `classname.m` for MATLAB and `classid.h` for C. When initializing MIXTOOLS 3000 in MATLAB using `prodini.m` it is loaded to memory.

Information about the class (or type) of the object is stored in a compulsory field named `class` as a number.

Example, data attributes and field `class` of instance of class `rvfinal`:

```
rvf = rvfinal_new([2,1], DFLT, DFLT, DFLT, 'my_rvfinal')
rvf =
    class: 45
    name: 'my_rvfinal'
    id: 1
    time: 0
    size: [2 1]
    obs: 1
```

The class number 45 corresponds to class `rvfinal`. The instance also has 5 regular data attributes – `id`, `time`, **`size`**, `obs` and `name`.

Mex version

Objects (instances of classes) that we work with in MEX files must be compatible (read identical) with objects used in MATLAB. The reason is obvious. Functions in MEX files will be called by MATLAB and both will operate on the same representation of objects in memory. Thus the object design stays the same. To create, modify and otherwise manipulate the objects, we use the provided MATLAB interface for C.

Plain C version

This version shares nearly all the source code with the MEX file version except for the *implementation* of the MATLAB interface for C. That means, all the design principles are preserved. We still think of objects as MATLAB-like structures with fields that are added dynamically. And we use the same fields as in MATLAB version. In addition to reimplementation of this explicit behaviour we had to implement some implicit behaviour of MATLAB, e.g. dimensions – structure is in fact a structure array of dimensions 1x1.

Our reimplementation of the MATLAB interface introduced several simplifications that suffice our needs. Among the notable is that we only allow two-dimensional arrays (numeric, cell) and we only use one numeric type for all numbers – `double`.

3.3.2 Inheritance

In MIXTOOLS 3000 there are two design guidelines that together bring about inheritance. They both govern the process of construction of object or class, naturally. The first concerns inheritance of data fields. In a constructor of an object, to be able to set its data fields, the object's predecessor must be constructed first – providing an instance that can be further manipulated. Thus, in a constructor, you have to manually call a constructor of the predecessor – one of them, if the predecessor has more constructors. You must store the return value and then add or modify its data fields.

The second guideline concerns inheritance of methods. When building a vtable for a class, one has to base it upon a vtable of its superclass. And this exactly is done in a vtable building script for all classes that have a predecessor.

3.3.3 Methods

In MIXTOOLS 3000, each function or method is implemented in a separate file. This is necessary, because when MATLAB calls a function, it looks for the function in a file with the same name as the function. When looking for the file, MATLAB searches through current directory and directories specified in its search path. The required path strings are added to MATLAB search path during initialization by running `prodini.m` which calls `setpath.m`.

Implementations of one method for different classes share one script (or function) that usually carries out parameter checking and dispatching – calling the appropriate implementation. This script (or function) is stored in a file `methodname.m` (MATLAB version) or `methodname.c` (C version) located anywhere in MATLAB search path or compilation path. We call it a *generic method*. Implementation of a method for a specific class is then stored in separate file named `classname_methodname.m` or with `.c` extension again located anywhere in the respective path.

Each specific implementation is assigned a globally unique number (ID). It is the same for MATLAB and C versions. This mapping of specific implementations to IDs is stored in files – `methname.m` for MATLAB and `methid.h` for

C. Both are generated by the script `create_methid.m` called after adding a new method. The mapping is loaded to memory during initialization of MIXTOOLS 3000 using `prodini.m`.

The number assigned to a method implementation (ID) acts as implementation-independent "pointer" to the method implementation. Before we can actually call a method implementation with a given ID, we have to translate its ID to an implementation specific pointer – a function handle in MATLAB or a function pointer in C. That is achieved by using a translation table. The table is created by a script `create_fhandle_table.m` in MATLAB or by a function in `fpointer_table.c` in C. `fpointer_table.c` is generated by the script `create_fhandle_table.m` after adding a new method.

Example: `rvfinal_rvcount` and `rvlist_rvcount` are implementations of method `rvcount`. The ID of `rvfinal_rvcount` is 155. This implementation-independent ID is used to obtain a function handle of the method implementation if we call it from MATLAB script. Or to obtain a pointer if we call it from C code.

Why do we need implementation-independent pointers to method implementations? Because we need a virtual method table to implement virtual methods.

3.3.4 Virtual methods

Process of calling virtual method

Our method calling convention follows standard MATLAB convention. For example, to call method *methodname* on object *obj* with arguments *arg1*, *arg2*, *arg3*, you need to write `methodname(obj, arg1, arg2, arg3)`. The object you wish to call the method on must be the first argument of the call.

When the example call is executed, MATLAB will call the function in script methodname.m or mex function in compiled mex file methodname.dll. It can reside in any directory in its path. The responsibility of such function is to perform a basic check of correctness of the arguments (`arg1`, `arg2`, `arg3`) and then to call the appropriate implementation of the method – for the class the first argument `obj` is instance of. That's why we call such functions *generic functions*.

The argument checking section usually checks if all required arguments are present and if all arguments are of correct type. It also substitutes default values for unspecified arguments or for arguments specified explicitly as default. We can say, this section makes up for the compile-time parameter checking in C++.

Calling the right implementation of the method is handled by a special function called `callmethod`. It takes the method name as first argument, the object as second and then the rest of the method's arguments. It locates vtable for the object's class in the array of vtables according to the object's `class` field. In its vtable it looks for the "universal" pointer to the method implementation by its name. If it's found, then the right method implementation is called using the translation table from "universal" to implementation-dependent pointers.

The format of generic function brings about one limitation. Two classes cannot have methods with the same name while their arguments have different meaning, or while there is different number of arguments. To be precise, they can, but it would require a special and not very elegant workaround. In other words, there is only one namespace for method names – and even this namespace is shared with namespace of MATLAB's built-in methods.

The process of calling virtual methods is essentially the same in MATLAB and C. Thus the generic methods in C are functional equivalents of generic methods in MATLAB.

Example of generic method `sample` in MATLAB:

```
function x=sample(self,n)

global DFLT PRUDENT_MODE;

% standard argument check
%if(nargin<1) self=DFLT; end;
if(nargin<2) n=DFLT;   end;

% prudent mode - checking arguments
if( PRUDENT_MODE)
    ENOUGH( nargin, 1);
    NOT_DFLT( self);
end;

% substituting default values
if isdflt(n) n=1; end;

% calling method implementation
x = callmethod('sample',self,n);
```

Format of the vtables

In our object framework, for each class its vtable stores a mapping of virtual method names to implementation-independent pointers (IDs) and a list of ancestors of the class. Each vtable is a structure with two fields, `ancestors` and `methods`.

Field `ancestors` is a vector of numbers (indicating the class) of all ancestors of the object and of the object itself. The numbers are in order of the class hierarchy, starting with the most general (distant) ancestor to the object itself in the end. This identification of object's "ancestry" is used when checking if object's class is a subclass of a given class, see `ASSERT_CLASS`.

Field `methods` is a structure, however different for each class. For each virtual method in the class there is a field with the same name in the `methods` structure. Its value is a number – the ID of implementation of that method for this class.

All the vtables are assembled in one big cell array, stored in variable `VMT`. It is indexed by the numeric identifier of the class.

The format of vtables in pure C version of the library is again the same as in MATLAB or MEX version. We just use our own implementation of structures and cell arrays. That also explains why we use cell array for assembling individual vtables instead of structure array. Our implementation of structures does not support structure arrays, only cell arrays. And we wanted to use the same container in both versions of the library to prevent confusion.

Example of vtables of two classes, `rv` and its subclass `rvfinal`. Notice, that `rvfinal` uses implementation of method `tstep` inherited from its parent `rv` (the same ID).

```
VMT{CLASSID.rv}.methods
ans =
                  dump: 89
         isrealization: 0
                 tstep: 117
                rvfind: 0
            rvsubselect: 0
                rvcount: 0
                 rvcell: 0
               rvexpand: 0
               getsizes: 0
           isobservable: 0
          setobservable: 0
    formaterealization: 0

VMT{CLASSID.rvfinal}.methods
ans =
                  dump: 89
         isrealization: 118
                 tstep: 117
                rvfind: 119
            rvsubselect: 120
                rvcount: 121
                 rvcell: 122
               rvexpand: 123
               getsizes: 124
           isobservable: 125
          setobservable: 126
    formaterealization: 127

VMT{CLASSID.rvfinal}.ancestors
ans =
    18    33    34

[ classname(18) ' <- ' classname(33) ' <- ' classname(34) ]
ans =
object <- rv <- rvfinal
```

Creation of `VMT` (array)

`VMT` assembles all the vtables in one big cell array. It is indexed by the numeric identifier of the class. The process of creation of `VMT` when MATLAB environment is available (MATLAB&MEX version) is different from the process in pure C version. Furthermore, the code responsible for creation of `VMT` for pure C version of the library is performed automatically by MATLAB script and thus depends on MATLAB. Detailed description of process of `VMT` creation in both MATLAB&MEX version and pure C version now follows.

Individual vtable building scripts

As the name already suggests, the responsibility of an individual vtable building script is to build a vtable for a specific class and place it at the appropriate position in the global VMT cell array. It is a MATLAB specific script (isn't supposed to have C equivalent) and for class *classname* it is stored in `classname_methods.m`.

It first checks if the required vtable is already created and if it is, then exits. Otherwise it calls the script for parent class to ensure that the parent's vtable is created. It then copies the parent's vtable, adds to it information about the given class and stores the result in VMT. The information being added is the IDs of class specific method implementations and update of the inheritance hierarchy (`ancestors`).

VMT for MATLAB&MEX version

The design of individual vtable building scripts provides for easy construction of VMT for all classes or for a subset of them. You only need to call the respective scripts in any order and VMT will be correctly created. This is done by `create_vmt.m` that recursively searches all folders in the MIXTOOLS 3000 root to call all available vtable building scripts. `create_vmt.m` is called during initialization in `prodini.m`.

VMT for pure C version

As soon as the VMT for MATLAB&MEX version is created, the script `create_vmt.m` starts traversing it and producing C code that will build the pure C version of VMT. It stores the code in `vmt.c`. In other words, the code for generating pure C version of VMT is created using the actual VMT for MATLAB&MEX. As a consequence, porting the individual vtable building scripts (`classname_methods.m`) to C is not needed at all. Thus eliminating the possible serious errors when porting vtables to C. Code from `vmt.c` is called and the VMT created during initialization of the pure C version of the library.

Rejected alternative

Usually, not all available classes will be used (instantiated) in a single computation session. This suggests an idea that the individual vtables could be created during computation in the moment when they are needed. It would be a moment when a new class is instantiated for the first time. The new vtable would be then inserted to the global VMT. This approach would be fine if each version of the library had its own copy of VMT. However, in this alternative, the MATLAB version and the MEX version must share the VMT, so that the VMT is updated for both versions, when one of them instantiates a new class for the first time.

The MEX version works with a copy of the shared global data (VMT). Thus if any of the data were modified, then when the MEX version returns control to MATLAB, MATLAB has to copy the modified data back to update the original.

This way, addition of vtable in MEX version would require copying the whole VMT back to MATLAB. This is the reason why we rejected this alternative and decided to build the complete VMT during initialization instead.

3.3.5 NoRedef methods

NoRedef methods is a name for a special sort of methods specific to our object model. Once a NoRedef method is defined in a class it is not intended to be redefined in any of its subclasses and even in any other class. The behaviour of the method is the same for all classes and does not directly depend on the instance that is given as the first parameter – unlike in virtual method. NoRedef method is in fact an analogy of *template method* design pattern.

NoRedef method is created when you place its implementation directly into generic method and omit the usual `callmethod` call in the end. However, NoRedef method may contain calls to other methods, that may be virtual – and those calls will still behave like calls to virtual methods. In this case, the NoRedef method acts as a template method. Its structure is given, but individual steps are virtual – dependent upon the class of actual instance they are called on.

There are no class specific implementations of NoRedef method and hence there are no entries in vtable for that method.

Example of NoRedef method `evalsomerv` that calls other virtual methods and thus works like a template:

```
function fno = evalsomerv(self,rv,values)

global PRUDENT_MODE
if( PRUDENT_MODE)
    ENOUGH( nargin, 3);
    NOT_DFLT( self);
    NOT_DFLT( rv);
    NOT_DFLT( values);
    ASSERT_CLASS( self, 'fn');
    if(~isrvfinlist(rv))
```

```
        error('argument rv is not rvlist of rvfinals');
    end;
end;

% 'hard-coded' implementation instead of callmethod call
which = rvfind(rv,self.rvs);           % call to virtual method
fno = evalsome(self, which, values);   % call to virtual method
```

Chapter 4

Programming with the framework

This section is a practical guide, that illustrates how to use our ROF to create new classes. To define a class, You need to perform at least three steps – write a constructor (includes definition of attributes), declare methods of the class (in a vtable building script) and implement those methods. This guide describes recommended structure of files produced in these steps. The files are made up of several sections. Some sections are preceded by a comment indicating the name of the section. This guide is partly a manual of style that we used to create MIXTOOLS 3000 library. So You don't have to obey it strictly, however we found it to be quite usable and useful.

In the code examples throughout the guide, the special documentation comments are left out to make the examples clearer.

4.1 Constructors

A constructor in our ROF is made up of these sections in the following order:

1. header
2. globals declaration
3. standard argument check
4. (optional) prudent mode
5. inheritance
6. default values
7. (optional) prudent mode
8. attributes

The constructors also contain documentation for their class in form of documentation comments.

The code examples that follow are not isolated, they form a functional whole. So the later examples relate to the previous ones. They are based on enorm_new.m script or enorm_new.c respectively.

4.1.1 Header

In our object framework we do not have special requirements for a function header. It's mentioned here mainly to show the names of arguments of the constructor, because they will be used in the rest of the examples.

MATLAB version:

```
function pdf     = enorm_new(rv,mu,R)
```

C version:

```
void enorm_new(int no, mxArray **out, int ni, const mxArray **in)
```

4.1.2 Globals declaration

In MATLAB we need to declare global variables before we can use them. And we will usually need to use these two:

DFLT a constant that indicates a default value of an argument

PRUDENT_MODE indicator of the state of PRUDENT MODE

CLASSID structure representing mapping from class names to their IDs

```
global DFLT PRUDENT_MODE CLASSID;
```

4.1.3 Standard argument check

The purpose of this section is to mark those optional arguments that were not given at all as having default value. Here we do not check, if obligatory arguments were given, we will do it later in prudent mode section.

```
% standard argument check
%if(nargin<1), rv=DFLT;end;     % obligatory argument
if(nargin<2), mu=DFLT;end;
if(nargin<3), R=DFLT;end;
```

In C, this part serves one more purpose. The input arguments come in the form of array of pointers. To make the code more readable, we rename the input arguments to the same names as in MATLAB script. We need to declare the argument names first.

```
mxArray *rv, *R;
VECTOR(mu);

/* standard argument check */
if (ni<1)
    rv = GLOBAL_DFLT;
else
    rv = (mxArray*)(in[0]);

if (ni<2)
    mu = GLOBAL_DFLT;
else
    mu = (mxArray*)(in[1]);

if (ni<3)
    R = GLOBAL_DFLT;
else
    R = (mxArray*)(in[2]);
```

4.1.4 Prudent mode

Two of our main objectives are speed and clarity of the code. So we try to isolate most of the error checking in sections called prudent mode. These blocks of code are executed only when the library works in a kind of debug mode called "prudent mode". Prudent mode section contains statements checking for errors or consistency. We have a convention that every such error checking statement that may cause a fatal error belongs to the prudent mode section. This section may appear in more than one place in the code. In constructors prudent mode sections usually appear on places indicated in the list in the beginning of Constructors section. The error checking in prudent mode sections ranges from checking obligatory parameters for presence, through checking class of an object, to checking more complex relations between objects. To learn more about prudent mode, see section 4.6.1 Error checking.

```
if( PRUDENT_MODE)
    ENOUGH( nargin, 1);
%   NOT_DFLT( rv);  % this is checked in the parent constructor
end;                % it does not belong here
```

Prudent mode sections in constructors are further special. They should only contain error checking specific for (or required by) this class and not checks that are already done by constructors of parent classes. You can see the point in the above example.

Prudent mode section in C:

```
if( PRUDENT_MODE)
{
    ENOUGH( nargin, 1);
}
```

4.1.5 Inheritance

In this section we need to construct the parent object by calling its constructor. We save the result and build our new object on it by adding or modifying its attributes. Then we shouldn't forget to set correct class information for object being created.

```
% inheritance
pdf       = epdf_new(rv);
pdf.class = CLASSID.enorm;
```

In C version, notice, that we pass pointer to the original argument (rv) and not pointer to its copy to the constructor of parent. We have to be very careful about the memory management in the C version. Copying an argument is a responsibility of the constructor that really needs the copy. The copy is usually needed when the constructor wants to use it as a value of an attribute. We will see an example in next section.

```
/* inheritance */
mxArray *pdf; /* will store result */

call_function(epdf_new,1,&pdf,1,rv);
setclass(pdf, CLASSID_enorm);
```

4.1.6 Default values

In this section, we assign actual default values to the optional arguments that are marked as being default.

```
% default values
dim = rvcount(rv);
if(isdflt(mu)) mu=zeros(dim,1); end;
if(isdflt(R))
    R = struct;
    R.L = eye(dim,dim);
    R.D = ones(dim,1);
end;
```

In C version, we additionally need to create copies of those arguments that we want to modify or use as a value of an attribute. Of course, only arguments whose value is given (not marked as default) should be copied. Again, we have to be careful to prevent memory leaks. To recap, copying an argument is a responsibility of the constructor that really needs the copy. If the constructor created a copy (or created default value) but did not use it as a value of an attribute, it should destroy the copy (value) when it is no longer needed.

```
/* default values and duplication of new parameters */
mxArray *mxDim, *mxL, *mxD;
int dim, i;

call_function(rvcount, 1, &mxDim, 1, rv);
dim = (int)mxGetScalar(mxDim);

if(isdflt(mu))
    VECTOR_FILL(mu, mxCreateDoubleMatrix(dim,1,0));
else
    VECTOR_FILL(mu, mxDuplicateArray(mu));

if(isdflt(R))
{
    mxL = mxCreateDoubleMatrix(dim,dim,0);
    mxD = mxCreateDoubleMatrix(dim,1,0);
    R = mxStruct(2, "L", mxL, "D", mxD);

    MATRIX_FILL(L, mxL);   /* initializes new aliases */
    VECTOR_FILL(D, mxD);   /* L, D of mxL, mxD that can be */
    // operated on by special macros
```

```
    for (i=1; i<=dim; i++)
    {
        COMPONENT(D,i) = 1;
        ELEMENT(L,i,i) = 1;
    }
}
else
{
    R = mxDuplicateArray(R);
}
```

4.1.7 Prudent mode

Typical reason for placing prudent mode section after default values section is to perform error checking on optional arguments. To make such checks readable, we need to have meaningful values substituted into the optional arguments.

MATLAB version:

```
if( PRUDENT_MODE)
    if(length(R.D) ~= dim),
        error('R must have the same dimension as random value'); end;
    if(prod(size(mu)) ~= dim),
        error('mu must have the same dimension as random value'); end;
end;
```

C version:

```
if (PRUDENT_MODE)
{
    if (length(mxD) != dim)
        error("R must have the same dimension as random value");
    if (mxGetM(mu)*mxGetN(mu) != dim)
        error("mu must have same dimension as random value");
}
```

4.1.8 Attributes

Finally, we assign values of object's attributes either obtained from arguments or calculated from them.

MATLAB :

```
% attributes
pdf.mu = mu;
pdf.R  = R;
```

C:

```
/* attributes */
addattribute(pdf,"mu",mu);
addattribute(pdf,"R",R);
```

4.1.9 Comments on structure of code files

In the old version of the library, the obligatory arguments of every function (including constructors) were always checked for presence and that they are not default. The optional variables were checked for the same two conditions. If the optional argument was not specified or was explicitly specified as default, then it would be assigned an appropriate default value in the initialization block of the function. This was typically done in one statement like this:

Example for argument called type, 4th, optional:

```
if ( (nargin < 4) || isdflt(type) )  type = 2; end;
```

In the new version of the library we isolated the checks of obligatory arguments in the prudent mode. We also divided the checks of optional arguments in two parts. In the first – standard argument check – we check if the argument is given. If it is not, we explicitly mark it as having default value by assigning a `DFLT` global variable to it. Usually we later check if the argument is designated as default, in which case we assign it the actual appropriate default value. However in constructors we sometimes need to specify that an argument should have a default value which we shall not determine in the constructor. This is used when we want to pass the optional argument to a constructor of our super-class and indicate

that it should have a default value. The constructor of the super-class then can determine what the actual default value should be.

Example corresponding to the previous:

```
if (nargin<4)   type = DFLT; end;
% ...
% prudent mode
% ...
if (isdflt(type))   type = 2; end;
```

We find this structure of code useful for two reasons. It divides the code into rather small maintainable and clear sections. It also allows us to create templates for various kinds of source code files, like constructors or methods. Thus all constructors, for example, have rather unified layout. Another motivation is that we want the MATLAB code and the C code to have very similar layout to be able to easily identify the part of the C code that corresponds to a given part of the MATLAB code and vice versa. The examples should explain what we mean:

Example: The original – "condensed" – structure of the code:

```
if ( (nargin<1) || isdflt(arg1) ) % obligatory argument
    error(' Argument arg1 must be given! Default value is forbidden.');
end;
if ( (nargin<2) || isdflt(arg2) ) % optional argument passed
    arg2 = DFLT;                  % to constructor of parent
end;
if ( (nargin<3) || isdflt(arg3) ) % optional argument
    arg3 = real_default_value;
end;

obj = create_parent_somehow(arg1, arg2);
```

The corresponding C code:

```
if (  (ni<1)  ||  isdflt( (mxArray*) in[0] )  )
    error(" Argument arg1 must be given! Default value is forbidden.");
else
    arg1 = (mxArray*)(in[0]);    // not copied

if (  (ni<2)  ||  isdflt( (mxArray*) in[1] )  )
    arg2 = GLOBAL_DFLT;
else
    arg2 = (mxArray*)(in[1]);

if (  (ni<3)  ||  isdflt( (mxArray*) in[2] )  )
    {
    /* prepare default value */
    arg3 = default_value;
    }
else
    arg3 = mxDuplicateArray(in[2]);   // copied

/* create parent somehow */
```

Example: The new way structure of code – in more steps:

```
% standard argument check
%if (nargin<1)   arg1 = DFLT; end;
if (nargin<2)   arg2 = DFLT; end;
if (nargin<3)   arg3 = DFLT; end;

% prudent mode
if( PRUDENT_MODE)
    ENOUGH( nargin, 1);
end;

% inheritance
obj = create_parent_somehow(arg1, arg2);

% default values
if (isdflt(arg3)) arg3=real_default_value; end;
```

The corresponding C code:

```
/* standard argument check */
if(ni<1)
    arg1 = GLOBAL_DFLT;
else
    arg1 = (mxArray*)(in[0]);

if(ni<2)
    arg2 = GLOBAL_DFLT;
else
    arg2 = (mxArray*)(in[1]);

if(ni<3)
    arg3 = GLOBAL_DFLT;
else
    arg3 = (mxArray*)(in[2]);

/* prudent mode */
if( PRUDENT_MODE)
{
    ENOUGH( nargin, 1);
}

/* create parent somehow */

/* assigning defaults */
if (isdflt(arg3))
{
    /* prepare default value */
    arg3 = default_value;
}
else
    arg3 = mxDuplicateArray(arg3);   // copied
```

In the other categories of source files (for generic methods and their implementations) we use the same structure of the code – we divide it in the above mentioned blocks. The blocks, that are not used are simply left out. We do it mainly to *keep the same coding style* throughout the whole project.

4.2 vtable building scripts

The role of vtable building script for a class is to construct a vtable for a given class and insert it into global array of vtables. These scripts are stored in files named `classname_methods.m`, where *classname* stands for the name of a class. Their structure is following:

1. initialization
2. inheritance
3. methods declaration

The subsequent examples are based on the script for class `enorm` named `enorm_methods.m`

4.2.1 Initialization

We first need to make accessible important global variables:

VMT the collection of individual vtables

CLASSID mapping from classname to numeric class ID

METHID mapping of method implementations to numeric IDs

Then we extract class ID for our class and perform a test if a vtable for this class exists in VMT. If it does already, we can terminate the script.

```
function enorm_methods

global VMT CLASSID METHID;

my_classid = CLASSID.enorm;

if (my_classid <= length(VMT)) && (~isempty(VMT{my_classid}))
    return;
end;
```

4.2.2 Inheritance

In this section, we have to explicitly specify, which class does our class inherit methods from. We initialize vtable of our class with the vtable of our parent class. However, we must ensure that the parent vtable exists already. Then we update the line of ancestors for our class.

```
% inheritance of parent methods
epdf_methods;                          % ensure parent's vtable exists
VMT{my_classid} = VMT{CLASSID.epdf}; % epdf is parent of enorm

VMT{my_classid}.ancestors(end+1) = my_classid;
```

4.2.3 Methods declaration

Finally we update our vtable by adding or modifying entries for methods introduced by this class. These methods fall into three categories:

implemented These are methods for which the class provides its own specific implementation. We store their universal ID (aka address) in a corresponding field (.methods.*methodname*) of the vtable of our class. The universal method ID can be obtained from global structure METHID in a field named *classname_methodname*.

abstract For abstract methods, we store special ID 0 in the appropriate field.

noredef NoRedef methods are really special. They are identified only by the name of the method. Such method is common to all classes. We don't store an ID for these methods in vtable, because their call is not realized using vtable at all. However, it is recommended to mention such a method in this section in a comment.

```
% methods
VMT{my_classid}.methods.sample = METHID.enorm_sample;
VMT{my_classid}.methods.evalpdflog = METHID.enorm_evalpdflog;
VMT{my_classid}.methods.moment = METHID.enorm_moment;
VMT{my_classid}.methods.edivergence = METHID.enorm_edivergence;
VMT{my_classid}.methods.maximum = METHID.enorm_maximum;
VMT{my_classid}.methods.visibound = METHID.enorm_visibound;
```

Example of special methods:

```
VMT{my_classid}.methods.abstract = 0;

% noredef - special method used by this class, cannot be redefined;
```

4.2.4 C equivalent

Unlike for other scripts, You don't have to create C equivalents of vtable building scripts. The C code that builds the whole collection of vtables is generated automatically during initialization of MATLAB version of MIXTOOLS 3000. The C code is generated based on the complete array VMT that is built by the individual vtable building scripts. In the pure C version it is provided as generated file.

4.3 Methods

Implementation of methods in our ROF is accomplished in two stages. To introduce method(s) with a new name that has not yet been used, You first have to create a *generic* method with that name. It serves as a gateway or entry point which MATLAB can use to call the method. Or from the opposite point of view, it enables You to call the implementation of the

method corresponding to the class it is invoked on. When generic method for a name is ready, You can then create one or more *implementations* of a method with that name for different classes. The implementation is defined by class name and method name and provides functionality of that method for objects that class.

For example, to implement new method `sample` for class `epdf` and its subclasses `enorm` and `emulti` You would definitely have to create generic method `sample`. Then according to your requirements, You might leave method `sample` of `epdf` abstract and create two implementations `enorm_sample` and `emulti_sample` for the subclasses. Or You might write universal implementation `epdf_sample` and let the subclasses inherit it.

4.4 Generic and NoRedef methods

Common roles of generic and noredef methods are argument checking and assignment of actual default values to default arguments. They also contain documentation for respective method in form of documentation comments. The only difference is that in the end generic method calls appropriate implementation of the method (late binding) while noredef has one common implementation hard-wired into it and that is executed. Their structure is:

1. header
2. globals declaration
3. standard argument check
4. prudent mode
5. default values
6. call implementation (generic) / implementation (noredef)

The following examples for method `sample` are based on files `sample.m` and `sample.c`.

4.4.1 Header

4.4.2 Globals declaration

4.4.3 Standard argument check

The purpose of these sections is the same as similar sections in 4.1.1 for constructors.

Code for MATLAB :

```
function x=sample(self,n)

global DFLT PRUDENT_MODE;

% standard argument check
%if(nargin<1) self=DFLT; end;
if(nargin<2) n=DFLT;   end;
```

C code:

```
void sample(int no, mxArray **out, int ni, const mxArray **in)

const mxArray *self;
mxArray *n;

/* standard argument check */
if(ni<1)
    self= GLOBAL_DFLT;
else
    self= in[0];
if(ni<2)
    n= GLOBAL_DFLT;
else
    n= (mxArray *) in[1];
```

4.4.4 Prudent mode

The basic purpose of this section – error checking – is the same as for the corresponding section in 4.1.4 for constructors. The section is only specific in the scope of the checks – what should be checked here. Only those checks that are general enough belong here. In other words, only checks common for all implementations of the method should appear here. These, for example, include checking if the object the method should be invoked on (usually the first argument) is given (non-default). Make sure, that the checks in this section do not depend on a property that is specific to only some implementations or classes.

MATLAB version:

```
% prudent mode
if( PRUDENT_MODE)
    ENOUGH( nargin, 1);
    NOT_DFLT( self);
    IS_CLASS( self, 'epdf');
end;
```

"Rather corresponding" C version:

```
/* prudent mode */
if( PRUDENT_MODE)
{
    ENOUGH( ni, 1);
    NOT_DFLT( self);
    ASSERT_CLASS( self, epdf);
}
```

4.4.5 Default values

In this section, we assign actual default values to the optional arguments that are marked as being default. We have a convention, that all default arguments have to be substituted with actual default values. So that a method implementation is called with proper values as arguments, not with the default markers. That serves to simplify design of the method implementation as well as maintain consistent default values across different implementations.

```
% default values
if isdflt(n) n=1; end;
```

Unlike in constructors we usually do not copy non-default arguments in C version. The originals are further passed to the method implementation. But we still have to take memory management into consideration. If we allocate for any default values we must remember to destroy them after the call to method implementation or when they are no longer needed. But we should only destroy those values we created. Thus we have to mark the values we created and later destroy them accordingly.

```
bool allocated_n = false;

/* default values */
if ( isdflt(n) )
{
    n = mxCreateScalar(1.0);
    allocated_n = true; /* must destroy n after use */
}
```

4.4.6 Implementation

This is the only section that is different for generic and noredef methods. While generic method calls appropriate method implementation to do the work, noredef method executes the implementation contained in this section.

```
% call method implementation
x = callmethod('sample',self,n);
```

As mentioned in the previous section, after calling the method implementation, we must dispose of the values, we previously created.

```
/* call method implementation */
const mxArray *new_input[2];

new_input[0]=self; new_input[1]=n;
```

```
callmethod("sample", no, out, 2, new_input);

/* destroy created values */
if (allocated_n)
    mxDestroyArray(n);
```

4.5 Method implementation

As the name implies this section describes how to write actual method implementation(s). Implementation of method *methodname* for class *classname* is stored in file named `classname_methodname.m` or `classname_methodname.c`. Its structure is:

1. header
2. initialization
3. prudent mode
4. implementation
5. return values

The code examples are loosely based on `enorm_sample.m` and `enorm_sample.c`.

4.5.1 Header

4.5.2 Initialization

Initialization is a unifying name for two things that need to be done in the beginning. It's globals declaration in MATLAB version and argument renaming in C version.

MATLAB version:

```
function x = enorm_sample(self,n);

global PRUDENT_MODE;
```

And C version:

```
void enorm_sample(int no, mxArray **out, int ni, const mxArray **in)

mxArray *self, *n;

self = (mxArray*) in[0];
n = (mxArray*) in[1];
```

4.5.3 Prudent mode

Error checking in method implementations mandatorily includes checking that all arguments are given. When the implementation is called from generic method, it should be guaranteed that all arguments are given and not default. Other than that the prudent mode section contains error checking specific for the implementation.

Prudent mode in MATLAB:

```
% prudent mode
if( PRUDENT_MODE)
    ENOUGH( nargin, 2);
end;
```

Prudent mode in C:

```
/* prudent mode */
if( PRUDENT_MODE)
{
    ENOUGH( nargin, 2);
}
```

4.5.4 Implementation

This is the place for "executive" part of the script/function that implements required functionality.

```
%implementation
x=cell(1,n);

[L,D]=ld2ld(self.R);
SQ=L'*sqrt(D);

for k=1:n
    e    = randn(self.rv.size);
    x{k} = self.mu + SQ*e;
end;
```

In the C implementation, keep an eye on the memory management and destroy all auxiliary variables you allocated. The implementation of this simple script in C is rather lengthy to include it here.

4.5.5 Return values

This section handles a specific issue with memory management in C version. The author of the implementation should be aware of the fact that the method can be called with lower number of requested output arguments (`no`) than the maximum possible. In such case it is the responsibility of the author to ensure that in the end the number of allocated and assigned output arguments is the same as requested. Author can either check in the end and destroy the extra unneeded arguments or prevent to create them in the implementation.

Although this approach is meaningful only in methods that can return more than one output we apply it out of habit and to be sure also in regular methods that return just one output.

For a discussion of special methods that modify one of their arguments in place, see section 4.6.2 In-place processing.

```
/* result */
if (no)
    out[0] = result; /* assign computed result to the array of outputs */
else
    mxDestroyArray(result); /* or destroy it if not requested */
```

4.6 Other features

4.6.1 Error checking

So called "PRUDENT MODE" (PM) is a mode of execution of MIXTOOLS 3000 library. It's concept is similar to debug mode. When PM is active, simple **error detection** and C++-like **type checking** is activated in the code. When PM is disabled, nearly none of that takes place. Remember, that MATLAB does not provide type checking and the C version of the library must behave in the same way. There is a small package of useful functions (or macros, in C language), which serve us for this purpose. We are going to discuss them separately, but now it is the right time for a little detour. All the project is about CPU time-saving. Execution of these checks takes significant amount of time, because they are rather plentiful in the code. That's why we still want to be able to turn these helpful but at times unnecessary features off. Our idea is following: develop and debug new functions with PM turned on, and when they are ready for real use (in massive computations) we could switch PM off to boost them. Such a switching is very simple, it's just a matter of changing global variable. In MATLAB, if a global variable `PRUDENT_MODE` is defined (in `prodini.m`) and true (non-zero), PM is active. In C, if a constant `PRUDENT_MODE` is defined (in `mixtools.h`) and true (non-zero), PM is active. However, for a Mex version of the library, if a macro `GET_PRUDENT_MODE_FROM_MATLAB` is defined (in `mixtools.h`) then the definition of `PRUDENT_MODE` constant reflects the value of global variable `PRUDENT_MODE` in MATLAB. PM also helps save time when porting MATLAB scripts to ANSI C, because the syntax is almost the same in both. We found this concept very useful.

List of prudent macros related to object framework:

ENOUGH(number, enough) Outputs standard error message if less than `enough` input arguments is passed to a function. `number` must be always set to number of input arguments – **`nargin`** (MATLAB) or `ni` (C). `enough` shall be an integer.

NOT_DFLT(value) Outputs standard error message if `isdflt(value)` is true, i.e. if argument `value` was specified explicitly as default. `value` shall be variable of any type.

ASSERT_CLASS(value, classname) Outputs standard error message if `value` is not instance of class `classname`. `value` shall be an instance of any class. `classname` shall be a class name in form of string in MATLAB and just a bare macro parameter in C.

ASSERT_CLASS2(value, classname1, classname2) Outputs standard error message if `value` is neither instance of class `classname1` nor instance of class `classname2`. `value` shall be an instance of any class. `classname` and `classname` shall be a class name in form of string in MATLAB and just a bare macro parameter in C.

NOT_INPLACE Outputs standard error message if function containing it is called without any output parameters. Useful in C code only, often a function call makes no sense if there is not any output. Should be present in functions, that are not expected to process any of their arguments "in place".

List of prudent macros specific to our application (theory):

ASSERT_REALIZATION(rv, value) Outputs standard error message if `value` is not a realization of given `rv` (i.e. random variable). `rv` shall be instance of any subclass of rv. `value` shall be cell array or vector.

4.6.2 In-place processing in C

In MIXTOOLS 3000 there are quite a few methods that modify objects. The object to be modified is one of their input arguments, usually the first one. To make the modified object available outside of the method, it must be returned as one of the output arguments. We believe that implementation of this MATLAB behaviour is not very effective. It means we cannot modify objects *in place*, which is rather inefficient mainly for bigger objects. The behaviour also affects design of MEX files (aka the C version). They must behave in the same way as MATLAB scripts *when they are called by* MATLAB. If we want to modify an input argument, we have to create a copy of it, modify the copy and return it as an output argument. Overcoming the unnecessary copying was one of the reasons to create alternate C version of library known as pure C version.

Our goal in the pure C version is to enable these methods to modify object in place. For us, *in place processing method* is a method, that is expected to modify one of its input arguments and should do it in place, when not called by MATLAB. Such method is recognized by its MATLAB signature. The name of one of its output arguments is the same as the name of one of its input arguments. And that is the argument to be modified. Example:

```
function epdf = updatecond(self, ovalue, value, epdf)
```

Mex version and pure C version share nearly all of the code. Mex version must modify a copy of the argument, pure C version should modify it in place. We could differentiate between those behaviours using conditional compilation or by doing check at runtime. We decided to use the latter option. The behaviour of the in place processing method is determined by the value of variable `no` that specifies the number of output arguments. When `no` is set to 0, the method is expected to modify the object in place, otherwise it must modify the copy of the object.

This can be accomplished by adding one line of code to the affected method implementations after the prudent mode section and before the implementation section. For example, if the argument to be modified is `epdf`, then the code could look like:

```
if (no > 0)  epdf = mxDuplicateArray(epdf);
```

If there is just one output argument, as in most cases, we can use the following line and then we can omit the return values section in the end.

```
if (no > 0)  out[0] = epdf = mxDuplicateArray(epdf);
```

There is a minor issue about the chosen solution. MATLAB may call a MEX function with the number of output arguments (`no`) equal to 0. It means that the user is not interested in the output of the method, however MATLAB accepts one output argument to store it in its **`ans`** variable. We solve this conflict by setting the value of `no` to 1 whenever a MEX function is called from MATLAB with `no` equal to 0. This is done in conditionally compiled stub, that is executed at the moment when MATLAB calls the MEX function.

4.6.3 Effective manipulation with global data

As MATLAB is copying the data each time the function is called, it takes a lot of time to do that. We work with huge matrices and we needed to speed up this process. That's why we created global data.

The data are huge so we have to be careful when working with them. Because of that the only allowed way to work with them is by calling functions with *global_* prefix. These functions work with structure `MIXTOOLS_WRAPPER`, which is cell vector of matrices.

4.6.4 Displaying objects

What to do if you want to add ability to your class to display its instance smartly? If you only have basic requirements, the method `dump` provided by class `object` (the topmost in class hierarchy) may be sufficient. It is designed to print the name of your object, its class and all attributes. Object dumping is recursive. Default recursion depth is set to 5.

The method `dump` provided by class `object` is quite general, thanks to recursive calling of generic dump function. But in spite of it, there are some cases when it must be overloaded. We can take method `dump` of class `decomposition` as a trivial example. Or, for more complicated situation, take a look at the following example of modified dump method with rich commentary.

```
function zoo_dump(self, levels, depth, indent)

global PRUDENT_MODE;
if( PRUDENT_MODE)
    ENOUGH( nargin, 4);
end;

names = setdiff(fieldnames(self), {'name', 'class'});
% all above is standard head of dump method

names = setdiff(names, {'hidden_elephant', 'name_of_elephant', ...
                        'color_of_elephant'});
% you do not want to print these atributes standardly

depth = depth+1;
% we want to increase text indent for this object

blank = blank4dump(depth);
% you must initialize this variable to set correct text ident

len = length(names);
for i= 1:len
    fprintf([ blank names{i} ': ']);
    % here you can see the example of text formatting with the 'blank' variable

    value = getfield( self, names{i});
    dump( value, levels, depth+1);
    % calling of standard dump method with text indent increased (see 'depth+1')
end;

fprintf('You can ignore text indent! The %s elepant is called ''%s'', ...
         but it is hidden right now.', self.color_of_elephant, ...
         self.name_of_elephant);

% non-standard printing of non-standard atributes;
% 'hidden_elephant' won't be printed at all!
```

Chapter 5

Verification and validation

Checking correctness or correspondence of MATLAB *and ANSI C implementations*

5.1 Introduction

We have the requirement that the first implementational platform for research in this field will be MATLAB and ANSI C, which will be interlinked via MEX-files. We need to test the correspondence of these files, so we know that the results of functions written in MATLAB and ANSI C are equal (within the bounds of floating point relative accuracy; we will use equal and same in this meaning throughout this chapter).

Main idea is to provide test scripts using random generators and generator of random systems for each mixtools3000 method. We have two types of test scripts which:

1. check whether the function result is correct
2. provide testing inputs and check if results of MATLAB and ANSI C function are equal

There are two auxiliary functions for testing: *rename_MEX* and *restore_MEX*. The former adds "MEX_" prefix to all (or all with specified prefix) MEX-files and the latter removes it. This way user can be sure that he is calling the right function. Otherwise MATLAB prefers calling MEX-file to M-file.

5.2 Checking functions for correctness

This type of testing is performed for functions whose results can be simply checked for correctness (e.g. ud2ld, udform, ldinv, ...).

The name of the test script contains prefix *"check_"* and the name of the function (e.g. *check_udform*).

Each function must accept one optional argument that can redefine the function to test, i.e. you can call `check_udform` or `check_udform('MEX_udform')`.

Each function must return 1 on success and 0 on failure.

5.2.1 failed = execute_checks(prefix, subdir, quiet)

Execute all check_*prefix** functions found in the *./subdir** subdirectories. If quiet is set to 1, no textual output is displayed.

Failed is a cell vector of names of failed functions. It can be used as input argument for rename_mex or remove_mex.

5.3 Preparing inputs and testing functions

Two steps: prepare inputs and test MATLAB and C files.
Auxiliary functions/scripts:

- get_mods
- paste_args
- cut_args

5.3.1 Prepare functions

input = prepare_*fname*(ntest, full, mods)

Function prepare_*fname* prepares inputs for function *fname* (e.g. prepare_rvempty prepares inputs for rvempty method of object rv). Each *prepare* function should prepare the most variable inputs so the test of function *fname* would be comprehensive. If there is a need for narrower inputs, user can call prepare function with optional parameter *mods*.

Parameter meanings are as follows:

ntest Number of prepared inputs

full Percentage of work this function carries out. It enables displaying progress of those functions that call other prepare functions. It affects number of dots being printed during preparation of inputs (i.e. if full is 100, there will be 30 dots printed; if full is 10, just 3 points will be printed)

mods Optional structure which can influence generation of inputs being prepared. It holds substructures named by objects or their methods. These substructures contain fields which change generation of inputs.
For instance, setting mods to following values restrict instantiate_rvlist to fill its rvs attribute just with two to five rvfinal objects (for instantiate function look below):

```
mods.rvlist.minlen = 2;
mods.rvlist.maxlen = 5;
mods.rvlist.rec_depth = 0;
```

Output of the prepare function is cell vector with its length equal to ntest (i.e. number of prepared inputs). Each element of this cell vector is structure which contains these fields:

args (required) requested inputs for function *fname* in form of cell vector. Its length must be same as the number of input parameters of function *fname*

nargout number of outputs from function *fname*

global defines global variables (e.g. for object memdatasource)

input = instantiate_*object*(ntest, full, mods)

Function instantiate_*object* is a special type of prepare function. It instantiates *object* and returns it in same cell array as prepare function does. This prepared object can be easily used in prepare functions.

output = run_file(input, fname)

This function returns results of function *fname*. Inputs for function *fname* are prepared by prepare function and stored in input. Results are returned in output in same format as input (see description for output in prepare_*fname*). Parameter fname is string.

output = function_result(fname, variant, ntest, full, mods)

This is de facto front-end for prepare_*fname* and run_file. Parameter variant can add prefix to fname (i.e. function which will be called; e.g. "MEX_", default is empty string). *ntest*, *full* and *mods* meaning is same as in prepare_fname function.

5.3.2 Test functions

User needs to compile C-files (see Chapter **??**) and call `rename_MEX` for testing purposes.

test_function(fname, variants, ntest, full, mods)

User can test if two *fname* function variants (usually MATLAB and MEX, i.e. default *variants* value is `{'', 'MEX_'}`) returns equal results by calling `test_function fname`. *ntest*, *full* and *mods* meaning is same as in prepare_fname function.

failed = execute_tests(prefix, subdir, quiet, variants, ntest, full, mods)

Execute all tests for functions whose prepare_*prefix** are found in *./subdir** subdirectories. If quiet is set to 1, no textual output is displayed.

Failed is a cell vector of names of failed functions. It can be used as input argument for rename_mex or remove_mex.

Variants, *ntest*, *full* and *mods* meaning is same as in test_function function.

5.3.3 Auxiliary functions

get_mods script

get_mods stores the elements of mods in local variables.
This will get mods for *mnorm* object for instance:

```
if isfield(mods, 'mnorm')
gm_mods = mods.mnorm;
get_mods;
end;
```

R = paste_args(cell1, cell2)

Pastes inputs prepared by prepare function in cell1 and cell2 to one cell in this order. Resulting length is that of the shorter cell.

[cell1, cell2] = cut_args(c, n)

Cuts prepared args to two parts. Puts n elements of them to cell1, rest to cell2.

Chapter 6

Files mentioned in the document

classid.h in `base`

- stores mapping from classnames to numbers (for C)
- generated by `create_classid.m`

classname.m in `base`

- stores mapping from numbers to classnames (for MATLAB) for error reporting purposes
- generated by `create_classid.m`

create_classid.m in `base`

- generates a mapping from classes (classnames) to numbers
- creates `classname.m`, `classid.h`, `classname.h`

create_fhandle_table.m in `base`

- generates a mapping (table) from method implementation IDs to MATLAB function handles
- creates `fpointer_table.c`

create_methid.m in `base`

- generates a mapping from specific (for a class) method implementations to numbers
- creates `methname.m`, `methid.h`, `methname.h`

create_vmt.m in `base`

- generates the VMT – array of all available vtables – by recursively searching all folders in the MIXTOOLS 3000 root to call all available vtable building scripts
- by traversing the VMT then creates `vmt.c`

fpointer_table.c in `base`

- contains C function generating a mapping (table) from method implementation IDs to C function pointers
- generated by `create_fhandle_table.m`

methid.h in `base`

- stores mapping from specific method implementations to numbers (for C)
- generated by `create_methid.m`

methname.m in `base`

- stores mapping from numbers to specific method implementation names (for MATLAB) for error reporting purposes
- generated by `create_methid.m`

prodini.m

- initialization and loading script for MATLAB version of MIXTOOLS 3000
- loads class-to-number and method implementation-to-number mappings

setpath.m in `base`

- sets search path for MATLAB – adds all subfolders of root MIXTOOLS 3000 directory to MATLAB search path

vmt.c in `base`

- contains the C code needed to build the pure C version of `VMT`
- generated by `create_vmt.m`

Bibliography

[1] P. Nedoma, M. Kárný, J. Böhm, and T. V. Guy, "Mixtools Interactive User's Guide", Tech. Rep. 2143, ÚTIA AV ČR, Praha, 2005.

[2] V. Šmídl, *Software analysis of Bayesian distributed dynamic decision making*, PhD thesis, Západočeská Univerzita Plzeň, 2005.

[3] A. Wald, *Statistical Decision Functions*, John Wiley, New York, London, 1950.

[4] A.A. Feldbaum, "Theory of dual control", *Autom. Remote Control*, vol. 21, no. 9, 1960.

[5] A.A. Feldbaum, "Theory of dual control", *Autom. Remote Control*, vol. 22, no. 2, 1961.

[6] M. Kárný, J. Böhm, T. V. Guy, L. Jirsa, I. Nagy, P. Nedoma, and L. Tesař, *Optimized Bayesian Dynamic Advising: Theory and Algorithms*, Springer, London, 2005.

[7] M. Kárný et al, *ProDaCTool Background*, Internal Report of IST-99-12058 Project, 2001.

[8] A. Rakar, T.V. Guy, P. Nedoma, M Kárný, and D. Juričić, "Advisory system prodactool: Case study on gas conditioning unit", *Journal of Adaptive Control and Signal Processing*, 2004, submitted.

[9] Kevin Murphy, "The bayes net toolbox for matlab", *Computing Science and Statistics*, vol. 33, 2001.

[10] P. Coad and Nicola J., *Object-Oriented Programming*, Prentice Hall, 1993.