
LTI Image Processing Library

Developer's Guide

Address LTI Computer Vision Library
Lehrstuhl für Technische Informatik
Ahornstr. 55
52074 Aachen

E-Mail ltilib@techinfo.rwth-aachen.de

WWW <http://www.techinfo.rwth-aachen.de/>
<http://ltilib.sourceforge.net/>

Coordinators Pablo Alvarado, Peter Dörfler

Der Lehrstuhl für Technische Informatik (LTI) und die Autoren übernehmen weder implizit noch explizit Haftung irgendwelcher Art. Der Benutzer trägt sämtliche Risiken, die aus der Verwendung der Informationen dieses Dokuments resultieren.

In keinem Fall kann der LTI für mittelbare oder unmittelbare, zufällige oder besondere Schäden oder Folgeschäden, die aus einem Mangel der Dokumentation resultieren, haftbar gemacht werden. Dies gilt auch, wenn auf die Möglichkeit eines solchen Schadens hingewiesen wurde.

Der LTI behält sich das Recht vor, dieses Dokument zu überarbeiten und gelegentlich zu verändern, ohne verpflichtet zu sein, zuvor irgendeine Person oder Organisation über solch eine Überarbeitung oder Änderung zu unterrichten.

The Chair of Technical Computer Science (LTI) and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

The LTI reserve the right to change the contents of this document at any time without notice.

All terms mentioned in this document that are known to be trademarks or service marks have been appropriately capitalized. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Contents

1	Introduction	1
1.1	History	1
2	LTI-Lib Architecture	3
2.1	Functors, parameters and states	3
2.2	Input and Output in the LTI-Lib	8
2.2.1	Example	10
2.3	Visualization Classes	11
2.4	Classifiers	11
2.5	Debug and Release Versions	12
3	C++ Programming Style Guide	13
3.1	Organisation of the data	13
3.1.1	Version control	13
3.1.2	File conventions	13
3.2	Tools	14
3.2.1	Debugging	14
3.2.2	Optimization	14
3.3	General programming conventions	14
3.3.1	Preamble	14
3.3.2	Name conventions	15
3.4	C++-Programming	15
3.4.1	File organization	15
3.4.2	Naming conventions	16
3.4.3	Class declaration and definition	17
3.4.4	Type casts	19
3.4.5	Cases	21
3.4.6	Heap	22
3.4.7	Text Formatting	22
3.4.8	Portability	23
3.4.9	Forbidden language features	24
3.4.10	Templates	25
3.4.11	Error handling	26
3.4.12	Status monitors	28
3.5	Creating new functors	28
3.6	Documentation	28
3.6.1	Documentation system	28
3.6.2	Documentation style	29

4	Class Hierarchy	31
4.1	Introduction	31
4.2	Data Structures	32
4.2.1	Basic Types	32
4.2.2	Vector, Matrix and Image Types	33
4.2.3	Image Regions and Contours	33
4.2.4	Sequences	33
4.2.5	Histograms	34
4.2.6	Pyramids	34
4.2.7	Trees	34
4.3	Functors	34
4.3.1	Mathematical Operations	34
4.3.2	Filters	35
4.3.3	Transforms and Modifiers	36
4.3.4	Other image processing functors	36
4.3.5	Input and Output	37
4.4	Viewer and Drawing Tools	37
4.5	Objects for multithreading	38
4.6	Classifiers	38
5	Download	39
5.1	Linux	39
5.2	Windows	39
	Bibliography	39
	Index	40

License

Copyright (C) 1998-2004

Lehrstuhl fuer Technische Informatik (LTI), RWTH-Aachen, Germany

This documentation is part of the LTI-Computer Vision Library (LTI-Lib)

The LTI-Lib is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License (LGPL) as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

The LTI-Lib is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with the LTI-Lib see the file LICENSE. If not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

More information can be found in the Internet:

<http://www.gnu.org/licenses/licenses.html>

1 Introduction

The LTI-Lib encloses a collection of algorithms and data structures commonly used in image processing and computer vision applications, developed at the Chair of Technical Computer Science (in German *Lehrstuhl fuer Technische Informatik* (LTI)), RWTH Aachen University. It is written in C++ in order to allow both object oriented programming as well as efficient resulting code. Its main goal is to provide a system independent library, adhering as closely as possible to the ANSI C++ standards. It is intended to work on different operating systems, but it specially supports for LINUX/GCC and MS WINDOWSNT/MS VISUAL C++ systems.

1.1 History

In the last years, several research groups at the LTI have worked on different applications in the field of computer vision. Before 1999, all projects using image processing algorithms (Sign Language Recognition, Mobile Service Robots and Visual Information Retrieval) developed their own software and applications independently. Two typical problems arose:

1. work efforts were being wasted due to unnecessary code duplication
2. reusing code was difficult or even impossible due to the lack for standardized interfaces.

These reasons forced the design of an internal library, which has been now used and extended for a while in all groups at the LTI. The choice to develop a new library was based on several factors:

First, we required a C++ library that followed a consistent object-oriented concept. This was intended to simplify the software development, enhancement and maintainability. Many libraries were implemented in C, or in a C-like C++ that was not appropriate to enhance or to adapt to our requirements. Others lacked a fundamental concept that could be employed in further developments.

Second, the library should provide not only algorithms for image processing, but also for mathematical and statistical tasks, allowing a flexible interchange of data between all modules. Many existent libraries did not fulfill these requirements.

Third, some of the regarded libraries were not maintained any more, some had no documentation at all. Commercial libraries were well documented and had nifty rapid prototyping tools, but due to a “closed source” concept, it would be impossible to enhance or to change existing algorithms without reimplementing them. This last point is unacceptable for research purposes.

Fourth, very powerful and widespread tools used in research (like MATLAB) can be used to search for solutions of relatively small problems, but they showed to be usually too slow for more complex applications or complete prototypes involving the whole scope from image acquisition and processing to feature extraction and classification. We usually required implementations that can run in real-time or that involve huge amounts of data.

The creation of a new library at the LTI was not a start-from-scratch project, due to the existence of previous code and sufficient expertise from all research projects. At the beginning, an interface was specified and the most important algorithms were adapted to it. The LTI-Lib has grown to more than 750 classes (more than 350000 lines), covering image processing, mathematics, statistics, neural networks, hardware interfaces, etc. with all objects following the same concept. The LTI-Lib-2 project started as a natural evolution of the previous concepts middle 2003. In these version more modern concepts can be used as the restrictions imposed by the Microsoft Visual C++ 6.0 compiler have been removed since the newer MS compilers (.NET 2003 and newer) are more ANSI C++ compliant. Also, additional design changes have been done based on the experience with the LTI-Lib-1.

The use of *one* library saves development time, which otherwise would be required in re-implementing commonly used algorithms. This time can now be invested in the development of new solutions or optimizing already existing ones. This way, not only one developers group will benefit from improvements, but all the users of the library. Its use increases also the code readability, due to the fact that all, complex and simple tasks, will now follow general known specifications. Further development or maintenance are therefore easier.

The following chapters explain the concepts behind the LTI-Lib interface, and presents all specifications required in the program coding.

2 LTI-Lib Architecture

The LTI-Lib is easy to use due to the specification of a consistent programming interface for all classes. The preservation of this consistency is partially achieved through the use of the PERL-script `ltiGenerator`. Based on a few rudimentary data provided by the programmer (like class name and parent class) this script builds some template files, containing all standard definitions. After that, only the functionality needs to be implemented. This chapter explains all basic concepts required to understand the meaning of this classes.

2.1 Functors, parameters and states

Most algorithms require *parameters*, i.e. user defined values that modify its behavior. For example, the file name is a parameter of an image loader class, or the size of a filter kernel is a parameter of a convolution algorithm. All algorithms in the LTI-Lib are encapsulated in so called functor-classes. They always enclose a class called `parameters`, that can be explicitly declared or just inherited from the parent class.

This means, when you use the LTI-Lib you do not call some functions or class methods with lots of confusing arguments, some meaning input data, others the output data, and additionally a long list of parameters. A default parameters object is usually stored within the functor class, and all methods that provide the algorithmic functionality expect from the user only the input data and the output objects where the results are going to be written. You can of course change the used parameters to fit the functor's functionality to your needs.

The parameters of a functor have to be distinguished from its state, which consists of all those attributes of the class that are computed during the execution of the algorithm, but are not directly provided or required by the user. For the *Motion History Images*(`lti::temporalTemplates`) for example, the last presented image must be kept in order to compute the next iteration. This image is not a parameter, but a part of the functor's state. These concepts are shown in Fig. 2.1.

There are several reasons for an independent parameters class. You can create several instances with different value sets and change at once the functionality of your functor with a simple `setParameters()`. You can load and save your

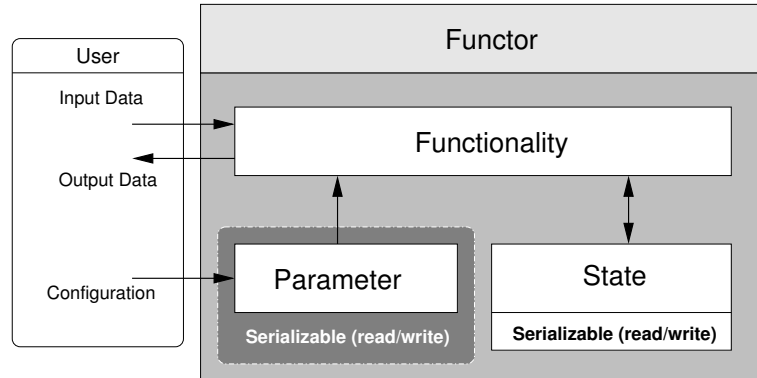


Figure 2.1: Architecture of a functor. The user can change the behavior of the functor through the parameters. The functor can also have a state, that eventually (like the parameters) can also be saved.

parameters object in a file, or can give it to a graphical user interface where the user can choose between several values.

The parameters contain values directly specified by the user and they should not be modified by the functor. If a programmer thinks his/her algorithm must change a parameter, this is just a sign that this parameter should be copied somewhere else at the beginning of the algorithm, like a local variable or a state variable (an attribute of the functor class).

An usual question is: why do I need to call the method `getParameters()` to get the parameters instance? would it not be faster if each functor-class had its own parameters instance that it could use directly?

The answer relies partially on memory management issues. It would be very expensive if all classes in the functor hierarchy would have their own instance of the parameters, because all inherited parameter attributes would be present several times. With the functor hierarchy shown on the left side of Fig. 2.2 an instance of the functor `lti::optimalThresholding` would have two parameter objects: the one of its own with five attributes (`precision` and the five attributes of the parent class) and the parameters-instance of `lti::thresholdSegmentation` with its four attributes. In other words, the four attributes of the parent class are present twice!

To avoid this problem, there exist just one instance of the parameters in the functor class. Each class casts this instance to the proper parameters type using the overloaded method `getParameters()`.

Another important reason for the use of just one parameters-instance in the functor class appears when the inherited class calls methods of the parent classes, the later ones could not see the proper parameters-instance but only the own one, which could contain other values than those specified by the user.

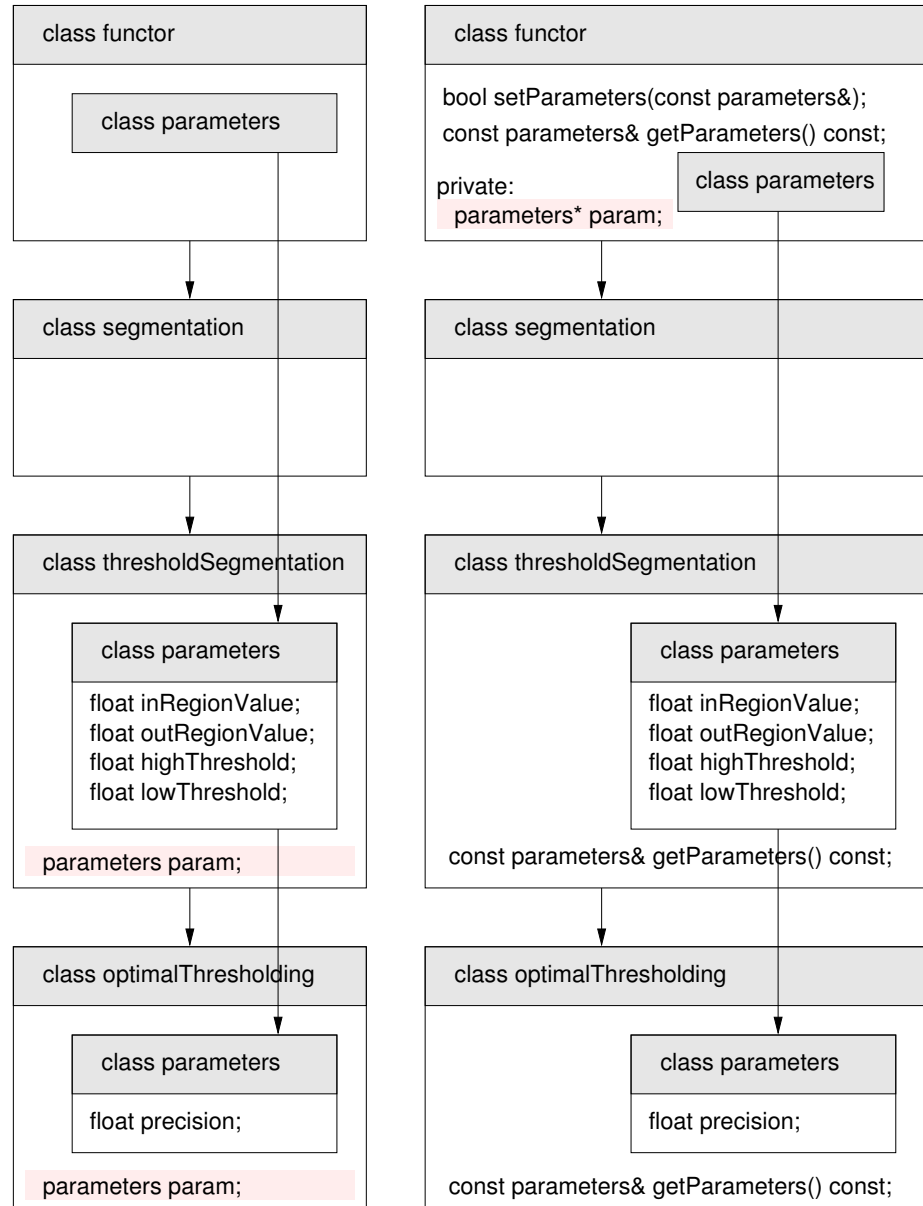


Figure 2.2: Example for the functor hierarchy: on the left side the naive way with several parameters-class instances (this implies a very inefficient memory management). On the right side the LTI-Lib way with just one parameters-class instance.

The functionality of a functor is always accessed by its methods `apply`. They expect input data, (usually constant references to objects like matrices or images), and references to output objects (references to containers where the result is written). Functors also provide the so called *shortcut*-methods that simplify the use of specific functionality. For example, to load an image file, the image loaders provide the shortcut `load` that expects the file name and the image where the result should be left. Otherwise, you would require to create a parameters object, set there the filename, give this parameters-instance to the functor, and at last call the `apply` method:

```
// an image
lti::image img;

// functor to load images in Windows BMP format:
lti::loadBMP loader;

// parameters for the loader
lti::loadBMP::parameters loaderParam;

// the file to load
loaderParam.filename = "testimage.bmp";

// load the image into img
loader.setParameters(loaderParam);
loader.apply(img); // load the image!
```

It is much easier and comfortable to employ following shortcut:

```
// an image
lti::image img;

// functor to load images with Windows BMP format:
lti::loadBMP loader;

// load an image
loader.load("testimage.bmp",img);
```

All functors must nevertheless provide an interface based on a `parameters`-object and `apply`-methods, in order to provide complex higher-level applications a uniform way to access the functionality of the functor.

Within the `apply` methods you can avoid an unnecessary copy of the parameters-instance getting a constant reference to them, for example:

```
/*
```

```

* apply method for myFunctor. Do something on the src vector and
* leave the result in the dest vector.
* @param src the input vector
* @param dest the output vector
* @return true if successful, false otherwise
*/
myFunctor::apply(const vector<double>& src, vector<double>& dest) {

    // Get a const reference to the functor parameters
    const parameters& param = getParameters();

    // use the parameters to do something, assuming you have
    // an attribute called "justCopy" in the parameters
    if (param.justCopy) {
        dest.copy(src)
    } else {
        // do something else ...
    }

    return true;
}

```

Please remember, the parameters should *never* be changed within the apply-methods:

```

bool myFunctor::apply(...) {

    ...

    const parameters& param = getParameters();

    param.justCopy = false; // is not possible, param is const!

    // you should NEVER EVER do something like this:
    const_cast<parameters*>(&param)->justCopy = false;

    // do something like:

    // a local copy of the parameters' attribute
    bool justCopy(param.justCopy);

    justCopy = false; // the local copy may be changed!

    ...
}

```

There exist just one `getParameters`-method and it returns a *constant* reference to the parameters-instance, this due to the fact that the parameters must not be changed by the functor.

As mentioned above, the functor may provide some *shortcuts* that allow changing specific parameter values. These methods, of course, should not be called within the `apply` methods.

Besides the parameters, a functor may have a state, where it stores information irrelevant for the user but necessary for later computations. An example for a functor with separated state and parameters is the `lti::principalComponents` object. Here you find a parameter `autoDim` which indicates that another parameter `resultDim` should be detected automatically. In the `apply` method this last value is not changed. The computed transformation matrix is part of the functor state, which can be used later to transform other vectors. This matrix is not something that the user can give directly, but can be saved and loaded with other parts of the functor's state (this is done when you load or save the whole functor).

All functors with a state relevant for later computations can be saved and loaded, i.e. they overload the methods `read` and `write`.

2.2 Input and Output in the LTI-Lib

WARNING: Some redesigning of the IO-Handlers concept is taking place in the LTI-Lib-2, which may conduce to more or less radical changes in the near future.

Serializable objects in the LTI-Lib (i.e. objects that can be written or read from disk) never directly use `std::fstream` objects. The main reason is that we need to provide a way to support different file formats at the same time. The desired file format is determined through a so called `lti::ioHandler`. At this time there are two file formats. A Lisp-like one writes or reads ASCII strings in or from a given stream, where different scopes are delimited with parenthesis. A binary format produces shorter files and is faster to be read or written, but can not be edited by hand.

A uniform way to load or save LTI-Lib-objects and internal types (`int`, `float`, `double`, `std::string`, etc.) is provided through four global functions that passes them properly to a given `ioHandler`. These are:

```
bool lti::write(ioHandler& handler,
               const T& data);

bool lti::read(ioHandler& handler,
               T& data);

bool lti::write(ioHandler& handler,
               const std::string& name,
               const T& data,
```

```
        const bool complete=true);

bool lti::read(ioHandler& handler,
              const std::string& name,
              T& data,
              const bool complete=true);
```

The first two functions write or read the contents of an object of type `T` in or from the given `ioHandler`. The third and fourth methods write the data together with a name that identifies this object. To read the data, the given name must match the one used as the data was saved.

With a handler of type `lti::lispStreamHandler` following lines

```
lti::write(handler,"a",5);
lti::write(handler,"b",9);
```

produce the following output in the output stream associated with the handler:

```
(a 5)
(b 9)
```

The parenthesis around each line can be left out if the fourth parameter of the functions (`complete`) is set to `false`. Note that the default value for this parameter is `true`.

The `lti::lispStreamHandler` can find an object using its name:

```
int x,y;
lti::read(handler,"a",x);
lti::read(handler,"b",y);
```

After these lines it applies `x==5` and `y==9`. Some `ioHandler` (for example `binaryStreamHandler`) require that the read order matches the one used when writing. If this is not true, the read methods will return `false`. Other `ioHandler` (like `lispStreamHandler`) search for the data using the given name as key, so that you can use a different reading order. Following lines would also result in `x==5` and `y==9`:

```
int x,y;
lti::read(handler,"b",y);
lti::read(handler,"a",x);
```

The `ioHandler` concept makes it possible to define new file formats *without* requiring to reimplement all read and write methods of the LTI-Lib-classes.

Due to the fact that the read and write methods use a rigorous syntax, it is also relative simple to parse the files.

Please note that the variables used in the previous examples could also have any other type defined in the LTI-Lib. All numerical standard types (`int`, `double`, etc.), the Standard Template Library (STL) types `std::vector`, `std::list` and `std::map` (if you include the file “`ltiSTLInterface.h`”) and the most LTI-Lib functors, parameters and data structures can be serialized.

2.2.1 Example

How can I save and load some parameters in my program?

```
// ltilib functors and their parameters
lti::csPresegmentation segmentor;
lti::csPresegmentation::parameters segParam;

lti::orientationFeature orientor;
lti::orientationFeature orientParam;

// ... initialize the parameters ...

// how can we write the parameters in a file named "param.txt"?
lti::lispStreamHandler handler; // the stream handler
std::ofstream out("param.txt"); // the std::fstream used

// if the output stream is ok, write the data
if (out) {
    // the handler have to write the data using the stream "out":
    handler.use(out);

    // write the parameters:
    lti::write(handler,"orientParam",orientParam);
    lti::write(handler,"segmentParam",segParam);
    lti::write(handler,"anInteger",5);
}

// how can we read the data from "param.txt"?
std::ifstream in("param.txt");

if (in) {
    int x;

    handler.use(in);

    // read the data
```

```

    lti::read(handler,"orientParam",orientParam);
    lti::read(handler,"segmentParam",segParam);
    lti::read(handler,"anInteger",x);
}

```

2.3 Visualization Classes

Not everything in an image processing or computer vision library can be considered as functor. Examples for this are the so called drawing and visualization objects.

Drawing objects does not execute one algorithm. They provide different tools to draw simple geometric constructs on images or other output media. To use a drawing object you need to provide it with your *canvas*, i. e. you need to specify the image where you want to draw. This is done with the method `use`. After that, you can choose the color you want to use with the method `setColor`. All lines, circles or points you draw after this, will be painted using the given color.

Following example draws a circle and a line on a color image:

```

lti::image img(256,256); // our canvas

lti::draw<rgbPixel> drawing; // drawing tool

drawing.use(img); // where should "drawing" paint on?
drawing.setColor(lti::Blue); // Blue color
drawing.circle(lti::point(128,128),
               20,true)); // filled circle, radius 20
drawing.setColor(lti::Red); // Red color
drawing.line(10,10,128,128); // A red line

```

Viewer objects do not modify any data, but provide simple ways to visualize them. The presentation of the data persists as long as the viewer object exists.

You can show the previously drawn image with following code:

```

lti::viewer viewer("This is art"); // our viewer object
viewer.show(canvas); // show our master piece
getchar(); // just wait

```

2.4 Classifiers

Other important object classes that do not fit into the functor paradigm are the classifiers. They provide methods to learn from data and to use the learned

information to analyze new data. There are different interfaces for the supervised and unsupervised classifiers. Both types can be categorized into instance classifiers that learn single vectors (like traditional neural networks) and sequence classifiers that also considered time aspects (like the Hidden Markov Models).

In the LTI-Lib all classifiers deliver the results using the same data structures `lti::classifier::outputVector`, so that the processing of their results does not depend on the specific classifier used.

2.5 Debug and Release Versions

The LTI-Lib has to provide very efficient methods and functions, and also ways to find programming errors comfortably. The design paradigm frequently used with MS Visual Studio has been adopted. There are *debug* libraries and *release* libraries. The former are usually slower because of many boundary checks. The latter is used in the final programs. It is assumed that in the debugging process all possible errors have been fixed, and some unnecessary checks are not done, implying faster code. The *debug* libraries are postfixed with a `d` and the *release* libraries with a `r`.

3 C++ Programming Style Guide

The Programming Style Guide provides several necessary rules to keep a clean and consistent development environment. Through this rules it is easier for a developer to find his way in the code, even if he is not the original author.

The main goal of the LTI-Lib is to provide a system independent library, which follows as much as possible the ANSI standards. The system should work on different UNIX-Platforms and on WINDOWS, but it is specially maintained for LINUX and WINDOWSNT machines. To achieve this, strict programming discipline is required. System dependent “tricks” are not allowed.

In those cases where is not possible to write system independent code (for example, due to direct hardware access), the platform dependent parts must be isolated in the smallest possible units. An access class have to be specified, which provides access to all special supported operations. No direct platform dependent access is allowed.

3.1 Organisation of the data

3.1.1 Version control

All code files will be administrated using a CVS-database. The repository will be on the CVS-Server. A few hints to use CVS are available. There are many graphic front-ends to simplify the use of this version control system, like WINCVS for WINDOWS (<http://www.wincvs.org/>) or CERVISIA for LINUX/KDE (<http://cervisia.sourceforge.net/>).

Each developer should not forget to comment its changes when committing into the repository. To avoid confusion, he/she should try to get only one copy of the LTI-Lib (for the CVS-Server there is no problem, but some developers have lose some work because they did changes somewhere else...).

3.1.2 File conventions

- Header files have always the extention `.h`.
- Source files have always the extention `.cpp`.

- Different words in a file name are written together, and each word (except the first one) must begin with an uppercase letter. The first word must be `lti`. Examples: `ltiFunctor.h` `ltiLinearFilter.h`
- Files which contain implementation for inline or template methods or functions must be clearly denoted with `_template` or `_inline`. For example `ltiMatrix_template.h` (see also section 3.4.10).

3.2 Tools

3.2.1 Debugging

Search your *bugs* with a Debugger. On WINDOWS NT you can use the MS VISUAL C++-Debugger. On LINUX there are many front-ends for the GNU-Debugger gdb, like XXGDB, KDBG and DDD. A class must work without problems before it is integrated in the rest of the LTI-Lib. To test your new functors, you can use the `tester` project provided with the library.

Other commercial products like PURIFY can help searching for memory problems like memory leaks or outer-bounds access of arrays.

If a program crashes, it usually helps to catch the `lti::exceptions` and ask their content with the method `what()` (see also section 3.4.11). It also helps to check if the called `apply()` or other functor methods return `false`, and in that case the cause of the problem can be checked with the `getStatusString()` method.

3.2.2 Optimization

Try to implement efficient code, but in a way that it remains maintainable. You can use GPROF or the MS VISUAL C++ profilers to detect the critical parts of your code, which may require special attention by optimization.

3.3 General programming conventions

3.3.1 Preamble

Each violation to the programming conventions must be documented. It is not always possible to respect all conventions. There are even cases where they are contradictory and following one will violate another. Each case have to be studied separately to decide which solution makes more sense. Discuss these cases with other LTI-Lib developers.

3.3.2 Name conventions

The names of classes, methods, functions and variables must always make sense. They all must be in English. Comments must be also in English. These way, your code can be understand all around the world.

3.4 C++-Programming

C++ was chosen as the main language for the LTI-Lib because it is possible to efficiently implement time and memory intensive algorithms (which are common in computer vision) without sacrificing code maintainability and using modern object oriented approaches. It is also important to know where to look up for information. There are two helpful books : [2] is a very good introductory book for C++ and [5], in which almost everything about C++ can be found, even if it is sometimes difficult to find.

3.4.1 File organization

- each file with source code begins with a comment including information about the file, like licence conditions, filename, authors, creation date, etc. If a special comment is present (see \$Id in the next example), CVS automatically enters some information in the file history:

```
/*-----
 * project ....: LTI Digital Image/Signal Processing Library
 * file .....: ltiThread.h
 * authors ....: Thomas Rusert
 * organization: LTI, RWTH Aachen
 * creation ...: 03.11.99
 * revisions ..: $Id: ltiThread.h,v 1.6 2003/02/17 07:17:01 author Exp $
 */
```

- you should avoid multiple inclusion of header files:

```
#ifndef HEADER_FILE_NAME_H
#define HEADER_FILE_NAME_H
...
#endif
```

- each class definition required in other header or implementation files, must have its own header file.
- header files of the LTI-Lib must be included with `#include "filename"`, system files with `#include <filename>`.
- cyclical `#includes` have to be avoided.

- each function has to show a comment documenting the interface and the functionality of the function or method. The comment format must follow the Doxygen style.

3.4.2 Naming conventions

All LTI-Lib declarations must be done within the namespace `lti`, except those members which extend other libraries (like **Standard Template Library (STL)**).

- names of data types, structs, classes, typedefs, enums, functions and methods begin with a lowercase letter. Only the enum constants and the template parameters begin with an uppercase letter.
- the name of enumeration types have to begin with the lowercase letter 'e' (for example `eBoundaryType`). The constants of the enumeration have to begin with uppercase letters (e.g. `Mirror`, `Periodic`).
- names with more than one word are written without separators, and each word (except the first one) must be written in uppercase. For example `linearEquationSystemSolutionMethod`. The only exception are the type names used in template classes to designate different properties of template parameters. This can be done to keep some concordance with the STL. For example in `matrix<T>`, the type of `T` can be accessed through the `value_type` type. In case of doubt, prefer the LTI-Lib standard (`thisWay` instead of `this_way`). In other words, underscores should only be found a very few times in template classes.
- names must not begin with an underscore (`_`).
- use names that make sense (not `mIn` for `maxIndex`)
- use C++ comments `//`. Comments used for documentation generation with DOXYGEN, must follow the Doxygen conventions, i. e. begin with `/**`, end with `*/`. In documentation comments with multiple lines, each line must begin with `'*'`. See also section 3.6.
- names must sufficiently differ. Differences in lowercase or uppercase letters only lead usually to confusion.
- protected and private attributes must have a trailing underscore, to make it possible in the implementation of class methods to differentiate between local variables and the internal attributes.
- precompiler macros (which, by the way, have to be avoided in the programming context) are written completely in uppercases and must have an `"_LTI_"` prefix. Several words are separated here with underscores.

Table 3.1 summarizes the naming conventions in the library. Please avoid the use of other conventions (especially the `m_{type}` conventions, you can read the reasons in [1]).

Table 3.1: Naming Conventions in the LTI-Lib

C++ Item	Name convention
class, struct, union	<code>theClassName</code>
inner type	<code>inner_type</code>
public attribute	<code>publicAttribute</code>
protected attribute	<code>protectedAttribute_</code>
private attribute	<code>privateAttribute_</code>
class method	<code>classMethod()</code>
public method for access to inner attributes	<code>getClassAttribute()</code> <code>classAttribute()</code>
public method for complex computations	<code>seekSomething()</code> <code>searchSomething()</code> <code>findSomething()</code> <code>computeSomething()</code> <code>generateSomething()</code> <code>calculateSomething()</code>
enum	<code>eEnumType</code>
constant in enum	<code>ConstantInEnum</code>
macro name	<code>_LTI_MACRO_NAME</code>

3.4.3 Class declaration and definition

Visibility sections

- The order for the visibility sections in class definitions must be:
 1. `public`
 2. `protected`
 3. `private`
- `friend` declarations are not allowed! The only valid exception is if a class needs access to protected attributes of enclosed classes (like iterators). Otherwise the class design must be revised.
- methods should not be defined in the class declarations, i. e. in the header file there should not be any code implementation. Template and inline functions should be implemented in `_inline.h` or `_template.h` files, which are appended using `#include`. Due to a bug of MS VISUAL C++(version 6.0), the implementation of enclosed classes of a template

class or template methods must be done directly in the class declaration. This is one of the reasons why the LTI-Lib-2 cannot be compiled with this older compiler. (see also section 3.4.10)

Methods and global functions

- global functions are not allowed (except for general tools like minimum, maximum or general mathematical functions like `sin()`, `cos()`, `tan()`, etc.)
- a method that does not change the state of an object must be declared as `const`. These are for example methods to access the value of some protected attributes.
- classes with attributes must define a copy-constructor and a copy member. These receive as parameter a `const` reference to an object of the same type. The documentation must specify, if these methods do a deep copy or a shallow copy (i.e. if the class contains pointers, if the whole pointed data are copied, or only the pointers.) In the LTI-Lib almost every copy method and copy constructor do a deep copy.
- classes that allocate data with `new` must define the operator `'='`, which also receives a `const` reference to an object of the same type.
- methods and functions which do not change the contents of the parameters, must declare them as `const`. This means, the “input” parameters for a method should always be declared as `const`.
- classes with virtual functions and own attributes must declare the virtual method `clone()`, which have exactly the same interface than the `clone()` method of the parent class. These method looks as follows:

```
rootClass* Foo::clone() const {  
    return new Foo(*this);  
}
```

- classes with virtual methods must have a virtual destructor.
- a public method must not return references or pointers to local variables, even if these variable are declared `static`. The reason for this restriction is that in multi-threaded programs this can cause crashes of unpredictable behaviours if parallel accesses takes place.
- a public method must not return references or pointers to protected attributes, except if the returned values are declared `const`.
- variable argument lists (...) in function definitions are not allowed, due to the lack of type checking possibilities. You can overload methods to provide the desired functionality.

- the names of the arguments must be the same in the declaration and definition of functions and methods. The names of the arguments must be self explanatory.
- the returned value of a function or method must be given explicitly.
- The methods in classes should provide one of following name conventions:
 - Methods with a name beginning with `get` or the direct attribute noun do not make expensive computations. For example, `size()` in the LTI-Lib is never as expensive as the `size()` methods in STL containers.
 - If a method returning a scalar value requires some computations to produce the desired result, the method should be prefixed with `compute`, `calculate` or `search`. For example `searchMaximum()` seeks the maximum value in a container.

Constants and variables

- Declare constant values with `const` or `enum` instead of `#define`.
- Do not use “magic numbers” in your code (`int value = 0x42`).
- Declare your variables in the smallest possible scope (if this does not affect considerably the efficiency of your program!).
- Initialize your variables before their first use (otherwise you will get megabytes of warnings from tools like PURIFY or VALGRIND without any logical reason).
- Due to the fact that pointer operations are always one of the most usual error sources, try to use references instead. If you allocate some memory, check your code for memory leaks.

3.4.4 Type casts

- Type casts in C-style are not allowed! (`new=(NewType)old`). C++-Casts must follow following conditions:
 - with polymorphic classes, it can be necessary to cast a static type into a dynamic one. This can be achieved using `dynamic_cast`. This way type checking will take place at compilation time and at run time, to ensure that the used types are compatible:

```
class A {  
};
```



```

class B: public A {
};

class C {
};

A a1,*a2,*a3;
B b1,*b2;
C *c2;
a2=&b1;           // a2 has dynamic type B
a3=&a1;           // a3 has dynamic type A
b2=dynamic_cast<B*>(&a2); // Ok, notNull(b2)
c2=dynamic_cast<C*>(a2); // Error at compile time:
                        // Types not compatible
b2=dynamic_cast<B*>(a3); // b2 is Null!, because B is
                        // not the type of a3

```

- if you cannot avoid it (for example using some system functions) you can replace an old C-case with

```
new=reinterpret_cast<NewType>(old)
```

This is though one of the most dangerous error sources. The use of the C++ cast emphasises when reading code, that the real type of a variable is just being ignored.

C cast are not allowed due to the fact that they are not type secure, and the LTI-Lib follows a strong type security philosophy.

- the secure type conversion between static types, which cannot be implicitly casted, can be done in C++ with `static_cast`. The compiler can decide if a type conversion is allowed or not. For example:

```

float f;
double d;

d = f;           // valid
f = d;           // not valid
f = static_cast<float>(d); // valid

```

`static_cast` is also used in the LTI-Lib to solve conflicts between signed and unsigned types. For example:

```

// compare the sizes of an lti::vector and a
// std::vector
lti::vector ltivector(5);
std::vector stdvector(6);

```

```

    if (ltivector.size() ==
        static_cast<int>(stdvector.size())) {
    ...
    }

```

or if this comparison is not in a time critical section of your code, this slower version can be used:

```

// compare the sizes of an lti::vector and a
// std::vector
lti::vector ltivector(5);
std::vector stdvector(6);

// use the integer constructor to convert the
// unsigned type into a signed int
if (ltivector.size() == int(stdvector.size())) {
    ...
}

```

- the last example shows the other C++ alternative to cast types using constructors. For example:

```

float f;
int i;

i=5;
f = float(i); // float constructor receives an
               // int parameter

```

This possibility must be considered carefully, because it can be very slow depending on the used compiler.

- use 0 instead of NULL. To check pointers if they are 0 or not, use the predicates `isNull` or `notNull`. You can use `isZero` or `notZero` to check integer variables.
- Never ever use `const_cast`. This is confusing due to the fact that something declared `const` must remain `const`, and should not be changed by the code.

3.4.5 Cases

- each `case`-label must end with a `break`-statement.
- A `switch`-statement must have a `default`-option, which handles the unknown cases, even if this cases does not exist. This is just a measure for prevention of future bugs.

- also the `default`-label must end with `break`.

3.4.6 Heap

- the C-library functions `malloc`, `realloc`, `free`, etc. should not be used. Use `new` and `delete` instead.
- to free the memory of arrays use `delete[]`.
- after deallocating a memory block, set the pointer to 0.
- functions that allocate some memory must take care of freeing it at the end.
- check always if a pointer is valid before you use it (except if you are absolutely sure that it is valid). For example:

```
myPointer=comingFromSomewhereElse();
if (notNull(myPointer)) {
    doSomething(myPointer);
} else {
    throw lti::exception("Oooh, missing pointer");
}
```

- due to the fact that pointers are the most common error sources, you should try to work with references instead. You should always check after possible memory leaks, if you really need to use pointers.

3.4.7 Text Formating

An appropriate text formating scheme alleviates the maintenance and further developing of existing code. It is also helpful in order to keep the code overview. Following rules have shown their usefulness in practice:

- Use a two-spaces indentation. Almost all modern text editors allow you to set the width of a tab, and if it should be replaced with spaces. Please use this options. Your code should never contain tabs, because they may seem to be all right for you, but for other users they will not.
- Function names, their return type and their parameters should be in the same line:

```
void myFuncName(int param1, int param2);
```

- if the number of parameters is too high, and they do not fit in one line, try to give one parameter per line:

```
void myFuncName(const int&                                paramArraySize,
```

```
const lti::vector<float>& paramArray,
lti::class&                className);
```

- all control flow primitives (if, else, while, for, do) must be followed by a complete block, even if they consist in just one line:

```
if (pValue == 0) {
    executeFunction();
}
```

```
for (int members = 0; members[i] < maxMembers; members++) {
    // loop stops, if maxMember value reached
}
```

- the opening brace must stay exactly after the statement (and not at the new line). The closing brace at the end should be at the same indentation level than the statement.
- in the declaration or definition of pointer variables or references, the *- or &-operators must stay after the data type. I.e. `char* p` instead of `char *p`. The reason is that C++ is a strong typed language, and the type of `p` is a pointer to `char`. To declare more than one pointer/reference use one line per variable, or define previously a type with `typedef`.
- before and after `.`, `->` and `::` there should not be any spaces.
- All methods and declarations should be printable in a DIN A4-Page (i.e. less than 70 lines). Lines should not be longer than 80 chars. Source files should not have more than 1000 lines.
- the priority for the application of operators should be explicitly given using parenthesis, in those cases where it can be ambiguous.

Some helpful tips for coding, that can also be applied to C++ can be found in [3].

3.4.8 Portability

Many programming errors are noticed just when the code is compiled using another compiler or another operating system. Following tips should help writing portable code:

- do not make any assumptions about the size of data types. The ANSI-Standard specify only:

```
sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
```

- Variables should never be declared as `unsigned`, even if their values never become negative. Exception to this rule is when the complete

value range of an unsigned variable is required (for example the coding of RGBA-values in a 32-bit int variable). In these cases the **unsigned** must be explicitly given. In those situations where unsigned variables of other libraries (like STL) need to interact with the LTI-Lib you should use the **static_cast**-operator instead of the C-Cast method in order make assignments or comparisons.

- a pointer does not have always the size of an integer `sizeof(int)`!
- type casts between **signed/unsigned** must be done explicitly.
- do never make assumptions about the size or location of data in memory (for example the number of bytes of a structure).
- the evaluation order of the operands in an expression should not be assumed. The only exceptions are boolean functions: in `a && b` or `a || b` `a` is evaluated first. Depending on its value, `b` is evaluated or not.
- do not replace arithmetical operation with shift operations (this should be done by the compiler). If you cannot avoid this, use comments to denote the semantic of the shifts.
- avoid use of pointer arithmetic.
- be aware of the endianness when reading or writing bytes in files. (Big-/Little-Endian).
- do not ever use MFC special classes, like `CList`, `CString`, `CPtrList` (they are only available for MS VISUAL C++). Use instead standard data structures of the Standard Template Library (Standard Template Library) (`std::list`, `std::string`, etc.)¹

3.4.9 Forbidden language features

Following reserved words or language constructs are not allowed in the LTI-Lib.

- `goto`
- Trigraphs
- functions without proper declaration
- variable argument lists
- `friends` (see also 3.4.3)
- Old C header files like `<stdlib.h>`, `<math.h>`, `<stdlib.h>`. Please note that for all this headers an equivalent C++ one exists:

¹Documentation for the Standard Template Library can be found in [4].

C (don't use)	C++ (ok)
<assert.h>	<cassert>
<ctype.h>	<cctype>
<errno.h>	<cerrno>
<float.h>	<cfloat>
<limits.h>	<climits>
<locale.h>	<locale>
<math.h>	<cmath>
<setjmp.h>	<setjmp>
<signal.h>	<csignal>
<stdarg.h>	<cstdarg>
<stdio.h>	<cstdio>
<stdlib.h>	<cstdlib>
<string.h>	<cstring>
<time.h>	<ctime>

You should avoid specially the use of old deprecated stream files like <fstream.h>, <iostream.h>, etc. The LTI-Lib use the standard files (e.g. <fstream> or <iostream>) combining them with the old deprecated ones leads usually to unpredictable behaviour!

3.4.10 Templates

Templates are sort of komplex strong-typed macros. With them you can write code for which the types of variables or function parameters are still unspecified. When you use your template classes or functions you will explicitly give the type (or types) for the variables you left “untyped”. Templates are extensively used for example in the **Standard Template Library** (STL).

If you use templates, you should know that

- no C++ compiler has 100% support for templates.
- templates behave sometimes like macros, with all known problems (annoying compiling errors, lots of code duplication and replication).

They have however as advantage, that if properly used the code can be implemented in a very efficient manner. Due to the speed requirements in image processing, templates are used extensively in the LTI-Lib.

For an efficient and clean implementation of code using templates, the ANSI C++ standard specifies the statement **export** [5]. MS VISUAL C++ and GCC do not support this feature. Do not forget this when creating new functors for the LTI-Lib.

3.4.11 Error handling

C++ provides many possibilities to deal with errors. In the LTI-Lib you must follow following conventions:

assert

Assertions are considered by the compiler in debug mode only. They inform the programmer about critical errors in code, that should *NEVER* happen, e. g. outer bound access to matrices or other container classes.

In release versions all **assert** statements are ignored. If an error like the one mentioned before is not fixed, the program will crash without giving any information. Please debug your code using the debug version. In order to avoid crashes in bigger software systems that use the LTI-Lib, you should *NEVER* report usual errors in **apply** methods using assertions. For this task you should implement a defined behavior for the functor (see below).

exception

Exceptions can be used in those cases where multiple errors are possible (e. g. during file access: file does not exist or invalid file format). The constructor of the class `lti::exception` will expect a string with a short description of the problem. You can of course inherit from this class if you think it is convenient. The most common use for exceptions is to report hardware problems, which can require special treatment to recover, or where a specific state must be warranted, even when error occur.

All functor classes in the LTI-Lib throw an `lti::exception` if the parameters are requested but not set, or when the parameter type is wrong.

An example for the use of **exceptions**:

```
// channels
lti::channel inputChannel, outputChannel;
// a convolution operator
lti::convolution convolver;
// initialize the convolution operator
// ... user sets the parameters here
// try to convolve
try {
    // try to convolve the inputChannel with a kernel
    // given by the user
    convolver.apply(inputChannel,outputChannel);
} catch (lti::functor::invalidParametersException& exc) {
```

```
// some error occurred with the parameters
std::cout << "Do not forget to set the proper parameters!"
           << std::endl;
// recover from the error
outputChannel.clear();
} catch (lti::exception& exc) {
    // some unexpected error occurred
    std::cout << "Error in convolver: " << exc.what()
               << std::endl;
    // recover from the error
    outputChannel.clear();
}

// don't know what to do with all other errors, they should be
// caught by the enclosing scope
}
```

If you don't use the **try-catch** blocks, all thrown exceptions will arrive enclosing scope (the calling function).

A nice introduction to exceptions can be found in [5].

Defined error behavior

This should be the error handling method of preference for all functor **apply**-methods. If two images with different sizes are given to a functor which expects images with the same size, the functor should not crash or throw an exception. It just needs to return **false** or an invalid result, that can be checked by the calling functions.

For all those not critical error, which are usually caused by a wrong usage of functions and classes, please follow these rules:

On-Copy apply The returned data objects should be reset (e. g. with **clear()** for images, matrices and vectors) or should return an invalid value (e. g. the result of a distance computation, which should be always positive, can be in case of error negative). These special error conditions must be documented.

On-Place apply The input/output parameters should not be changed.

The **apply**-methods return a boolean with the value **true** if everything worked fine and **false** if a error occurred. Within the methods of the functor class you can use **setStatusString()** to report the cause of an error. The user can check the returned boolean value if there was a problem and **getStatusString()** to determine the cause of the problem.

3.4.12 Status monitors

If an error occurs in a functor, classifier or other computationally expensive LTI-Lib object, the failing method can report the error through the functor's status string. Internally, the functors will set a status string with messages that indicate what went wrong, and the user can obtain those strings using the method `getStatusString()`. You can additionally configure the behavior of the whole library with objects inherited from `lti::statusMonitor`, which will additionally provide the chance to print out the errors as soon as they occur, to always throw an exception if you wish so, or, the default behavior, to do nothing.

3.5 Creating new functors

All functors in the LTI-Lib have a common framework. The PERL-Script `ltiGenerator` (to be found in `ltilib/tools/perl/`) provides a simple way to create this class frame. You just need to fill out some fields in the text file `ltiTemplateValues.txt` and then call the script.

The expected values in `ltiTemplateValues.txt` are:

`classname` the name of the new class.

`parentclass` the name of the class from which the new class will inherit.

`author` the author (or authors) of the class.

`date` (Optional) creation date (if leaved empty, the current date will be taken).

`includes` (Optional) A comma separated list with all header files required.

`filename` (Optional) The name of the file where this class should be found. If left empty, the name will be automatically created from the class name, following the naming conventions of the LTI-Lib.

`parameters` The list of parameters. Of course you can add later new parameters manually. You just need to update the code in the parameters constructor and in the methods `copy`, `read` and `write`.

`applytype` The initial parameters types for the `apply`-methods.

3.6 Documentation

3.6.1 Documentation system

You can find the online-documentation under

`http://ltilib.sourceforge.net/doc/html/index.html`

This will always correspond to the latest released version. It will be created using the comments in the header-files of each class and the file `ltilib/doc/src/`. The documentation system DOXYGEN is used for this task (for detailed information see [6]).

Important is to notice that doxygen comments begin with `/**` and end with `*/`.

```
/**
 * This is how a comment should look like
 *
 * All doxygen commands start with \ or @ (e.g. \param).
 * Use % to avoid, that the following word gets linked.
 */
```

3.6.2 Documentation style

The documentation of all classes must be written in English. It must contain a brief description, and a detailed explanation with all relevant information like references to literature with theory to the implemented algorithms. All methods and attributes of the classes must also be documented. For all attributes of the parameters-classes you have to give the used default values.

The brief and detail descriptions must begin with an uppercase letter.

Example:

- Classes

```
/**
 * Computes the optical flow between two consecutive images
 * according to Horn-Schunks gradient based method. ...
 *
 * Theory: "The Computation of Optical Flow", Beauchemin
 * & Barron, ACM Computing Surveys, Vol.27 No. 3
 * Algorithm: "Computer Vision", Klette, Schluens &
 * Koschan, Springer, pp.190
 */
```

- Methods

```
/**
 * Computes the optical flow for the given sequence of images.
 * @param theSeq The sequence of images, which must have at least two
 * @param mag magnitude of the optical flow
```

```
* @param arg angle of each optical flow vector.  
* @return true if successful, false otherwise.  
*/  
bool apply(const sequence<images>& theSeq,  
           channel& mag,  
           channel& arg) const;
```

- Attributes

```
/**  
 * Number of iterations. iterations = 1,2,...  
 *  
 * Default value: 100  
 */  
int iterations;
```

Of course you should also comment your source code, so that other programmers can work on it, but the sources are not going to be parsed by Doxygen.

4 Class Hierarchy

All Objects in the LTI-Lib are in the namespace `lti`. The basis class in the LTI-Lib is `object`.

The class `lti::mathObject` is the parent class for many aggregate classes, like vectors and matrices. Most algorithms in the library operate on this kind of data. `lti::functor` objects implement the algorithms. `lti::exception`-objects inherit from `std::exception`, which are thrown in case of critical errors.

The basic types in the LTI-Lib are listed in Table 4.1 (all of them defined in namespace `lti`).

More details on these and other classes can be found in the Doxygen documentation of the library or in <http://ltilib.sourceforge.net>. Next sections provide an short overview of the documentation there¹.

4.1 Introduction

The LTI-Lib is a collection of algorithms and data structures frequently used in the image processing.

¹Please note that this section is not at all up-to-date. Only the online documentation contains the actual functors, which are *many* more than the ones listed here. The goal of these paragraphs is only to provide a fast overview of the main topic of the library.

Table 4.1: Basic Types in the LTI-Lib

<code>byte</code>	8-bit Typ signed
<code>ubyte</code>	8-bit Typ unsigned
<code>int16</code>	16-bit Typ signed
<code>uint16</code>	16-bit Typ unsigned
<code>int32</code>	32-bit Typ signed
<code>uint32</code>	32-bit Typ unsigned
<code>point</code>	2D points with <code>lti::int32</code> coordinates
<code>rectangle</code>	Two 2D points (upper-left and bottom-right)
<code>rgbPixel</code>	a 32-bit long RGB pixel representation

These pages will give some informations about the usage of the LTI-Lib. For details about the usage of each class follow the link "More..." at the header of each class documentation.

The data structures used in the LTI-Lib are described in the section Data Structures (4.2). All the algorithms implemented in the LTI-Lib are Functors (4.3).

The algorithms can be organized in the following groups:

- Mathematical Operations (4.3.1)
- Filters (4.3.2)
- Transforms and Modifiers (4.3.3)
- Other image processing functors (4.3.4)
- Input and Output (4.3.5)

4.2 Data Structures

There are six groups of data structures:

- Basic Types (4.2.1)
- Vector, Matrix and Image Types (4.2.2)
- Image Regions and Contours (4.2.3)
- Sequences (4.2.4)
- Pyramids (4.2.6)
- Trees (4.2.7)
- Histograms (4.2.5)

4.2.1 Basic Types

The basis types allow the representation of usual data structures like:

- Color pixels: `lti::rgbPixel`, `lti::trgbPixel<T>`
- Points in a 2D space: `lti::point`, `lti::tpoint<T>`
- Rectangles: `lti::rectangle`, `lti::trectangle<T>`

4.2.2 Vector, Matrix and Image Types

The LTI-Lib Algorithms manipulate two types of containers: one-dimensional (`lti::vector`) or two dimensional containers (`lti::matrix`).

Both of them are implemented as C++ templates. This allow an efficient reuse of the code with different types of data.

A vector of integers can be created with the following code:

```
lti::vector<int> aVct(5,0); // this creates a vector with 5 elements
                           // and initialize them with 0
```

A 3x4 matrix of double (3 rows and 4 columns) will be created with

```
lti::matrix<double> aMat(3,4,1.0); // creates a 3x4 matrix of double
                                   // and initialized elements with 1.0
```

There are some "alias" for frequently used matrix types:

We have for example the type `lti::image`, which is defined as a matrix of color pixels (`lti::rgbPixel`).

A channel is in this library an image with monochromatic information. There are two types of channels: 8-bit channels with pixel values between 0 and 255 (`lti::channel8`) and the `lti::channel`, which is an matrix of floats.

4.2.3 Image Regions and Contours

The data structures for the representation of Vector, Matrix and Image Types (4.2.2) are not enough to satisfy every requirement of the image processing algorithms. Some types to allow the representations of image regions are also required.

This function will be achieved by classes like `lti::pointList` and its sub-classes (`lti::areaPoints`, `lti::borderPoints` and `lti::ioPoints`).

4.2.4 Sequences

The `lti::sequence` is a sort of vector of other objects. It is indeed based on the `std::vector`, but with a LTI-lib standard interface.

4.2.5 Histograms

The histograms are a collection of classes used to generate n-dimensional statistics. The base class is `lti::histogram`. Usually a `lti::mapperFunctor` is used to map the input space into the histogram cell-space.

4.2.6 Pyramids

The multiresolutional analysis requires frequently the use of pyramidal data structures. Gaussian, Laplacian and Gabor pyramids are examples of this data-structures. (see `lti::pyramid`, `lti::gaussianPyramid`, `lti::gaborPyramid`)

4.2.7 Trees

A container class for ordered n-Trees is implemented in `lti::tree`

4.3 Functors

The functors are objects which can operate on given data structures. They can be parameterized with different values of some specific members.

For example, the functor which saves an image in a BMP-File has parameters which include the name of the image file, the compression rate to be used, the color depth and so on. For a Gaussian filter we need only the variance and the dimension of the filter.

The LTI-Lib encapsulates these functor parameters in an object of an internal class called "parameters" (see `lti::functor::parameters`). The programmer knows, that all parameters required for a functor will be defined in its class parameters.

4.3.1 Mathematical Operations

- Noise and probability distributions
 - `lti::noise` adds noise with a given distribution to matrixes or vectors
 - `lti::randomDistribution` classes are used to generate random numbers which follow a specific probability distribution
- Linear Algebra

- `lti::linearEquationSystemSolutionMethod`
- `lti::matrixInversion`
- `lti::minimizeBasis`
- `lti::principalComponentsAnalysis`
- `lti::l1Distance`, `lti::l2Distance`
- `lti::varianceFunctor` calculates variance and covariance
- `lti::serialStatsFunctor` and `lti::meansFunctor`

4.3.2 Filters

- Kernels for linear filters
 - gaussian kernels (`lti::gaussKernel2D`, `lti::gaussKernel1D`)
 - gabor kernels (`lti::gaborKernelSep`, `lti::gaborKernel`)
 - oriented gaussian derivatives (`lti::ogd1Kernel`, `lti::ogd2Kernel`)
 - gradient approximation kernels (`lti::gradientKernelX`, `lti::gradientKernelY`)
 - binary kernels for the morphological operators (`lti::cityBlockKernel`, `lti::chessBoardKernel`, `lti::octagonalKernel`, `lti::euclideanKernel`)
- Linear Filters (convolves objects with linear kernels)
 - `lti::convolution`
 - `lti::squareConvolution` implements convolution with a rectangular filter kernel.
 - `lti::downsampling` Efficient implementation of a filter-downsampling pair.
 - `lti::decimation` efficient implementation of image decimation
 - `lti::upsampling` Upsampling-filter pair.
 - `lti::filledUpsampling` efficient implementation of upsampling with a rectangular filter.
 - `lti::correlation`
- Iterating functors This functors apply a simple function to all elements

of the matrix. Please note, that this can also be done giving the C-Style function to the `apply()` method of the matrix.

- `lti::absoluteValue`
- `lti::logarithm`
- `lti::addScalar`, `lti::multiplyScalar`
- `lti::square`, `lti::squareRoot`
- Morphological operators
 - `lti::dilation`
 - `lti::erosion`

4.3.3 Transforms and Modifiers

- Color spaces transformations
 - `lti::mergeImage` Merge three channels in a color image
 - `lti::splitImage` Splits an image in different color spaces
 - `lti::rgbColorQuant` color quantization in RGB space
- Gray value transformations
 - `lti::histogramEqualization`
- Coordinate Transforms
 - cartesian \leftrightarrow polar coordinate: `lti::cartesianToPolar` and `lti::polarToCartesian`
- Other Transforms
 - FFT: `lti::realFFT` and `lti::realInvFFT`
 - Optical Flow: `lti::opticalFlowHS`
 - Illuminant Color Normalization: `lti::colorNormalization`

4.3.4 Other image processing functors

- Edge detectors
 - `lti::susanEdges` Use SUSAN algorithm to extract the edges of an image

- Segmentation
 - `lti::thresholdSegmentation`
 - `lti::regionGrowing`
- Tools for segmentation
 - `lti::objectsFromMask`
 - `lti::boundingBox`

4.3.5 Input and Output

- The LTI-Lib supports loading and saving of Windows Bitmap files (BMP), Jpeg files (JPEG) and Portable Network Graphics files(PNG).
 - `lti::loadBMP`
 - `lti::saveBMP`
 - `lti::loadJPEG`
 - `lti::saveJPEG`
 - `lti::loadPNG`
 - `lti::savePNG`
- It can also acquire data directly from a frame grabber:
 - `lti::frameGrabber`
 - `lti::quickCam`
 - `lti::toUCam`
 - `lti::ltiFrameGrabber`

4.4 Viewer and Drawing Tools

- Viewers There are two viewers in the LTI-Lib
 - `lti::externViewer`

This viewer is used to invoke in a very easy way an external application.
 - `lti::viewer`

Is a GTK-based object, which runs in its own thread, to allow the

user a very easy way to show images, fast as easy as the `std::cout` stream!

- Drawing tools
 - `lti::draw`
 - `lti::draw3D`
 - `lti::epsDraw`
 - `lti::drawFlowField`

4.5 Objects for multithreading

Some objects allow a OS independent use of multithreading functions:

- `lti::thread`
- `lti::mutex`
- `lti::semaphore`

4.6 Classifiers

It is possible with the LTI-Lib to use some classifiers.

- `lti::rbf` Radial Basis Functions
- `lti::MLP` Back propagation networks
- `lti::svm` Support Vector Machines
- `lti::hmmClassifier` Hidden Markov Models

Classifications statistics can be generated using

- `lti::classificationStatistics`

There are many others supervised and unsupervised classifiers.

5 Download

You can get a tar ball of the LTI-Lib latest version from

<http://ltilib.sourceforge.net/doc/html/index.html>

5.1 Linux

From the shell and using GNU tar you can decompress the file with

```
tar xzvf ltilib.tar.gz
```

or

```
tar xjvf ltilib.tar.gz
```

After that you will need to build the library:

```
cd ltilib/linux
./configure
make
```

and as root or a user with write privileges on the prefix given to the configure (usually /usr/local) you need to install the library just a

```
make install
```

5.2 Windows

In `ltilib/win/tester` there is an example project that uses all LTI-Lib files directly. You can make some experiments with this project.

You can of course build the library. For this you will need a Perl-Script interpreter installed. In `ltilib/win/buildLib` execute the file `buildLib.bat`. This will create the libraries in the directory `ltilib/lib`. Additionally all header-files will be collected and placed in `ltilib/lib/headerFiles`.

More information creating your own projects using the LTI-Lib can be found in the on-line documentation.

Bibliography

- [1] Stephen C. Dewhurst. *C++ Gotchas. Avoiding Common Problems in Coding and Design*. Addison-Wesley, 2003. 17
- [2] Stanley B. Lippman and Josée Lajoie. *C++ Primer*. Addison-Wesley, Reading, MA, 3rd edition, 1998. 15
- [3] Sun Microsystems. Java Code Conventions.
<http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>,
April 1999. 23
- [4] Silicon Graphics Computer Systems, Inc. Standard Template Library, Programmer's Guide.
<http://www.sgi.com/Technology/STL/>, 1999. 24
- [5] Bjarne Stroustrup. *Die C++-Programmiersprache (3. Auflage)*. Addison-Wesley, Bonn, 1998. 15, 25, 27
- [6] Dimitri van Heesch. Doxygen.
<http://www.doxygen.org/>, April 1997-2003. 29

Index

- dynamic_cast, 19
- include, 15
- namespace, 16, 31
- template, 14
- LTI-Lib, 1
- export, 25
- lti, 31
- unsigned, 23

- advantages, 2
- arguments, 19

- C++ Books, 15
- copy-constructor, 18
- CVS, 13

- Debugger, 14
- Doxygen, 29

- enum, 16

- File conventions, 13
- filenames, *see* file conventions
- Forbidden language features, 24
- functor, 3

- name
 - class, 16
 - function, 16
 - method, 16
- Naming conventions, 16

- struct, 16

- template, 25
- Type casts, 19
- type definition, 16

- underscore, 16

- version control, *see* CVS